PYTHONDAQ – A PYTHON BASED MEASUREMENT DATA ACQUISITION AND PROCESSING SOFTWARE

Daniel Jäger[1]
Technical University of Munich
TUM School of Engineering and Design
Chair of Turbomachinery and Flight Propulsion

Volker Gümmer
Technical University of Munich
TUM School of Engineering and Design
Chair of Turbomachinery and Flight Propulsion

ABSTRACT

This paper introduces PythonDAQ, an open-source Python package for measurement data acquisition, visualization, storage, and post-processing. The code is capable of acquiring measurement data from any sensor with digital data output, performs online calculations, and stores the measured and computed data. A client for live data visualization and tools for postprocessing are also contained in the software package. First, the code is introduced explaining the software architecture and the currently implemented features. Then, the usability is demonstrated by an application at the low-speed compressor test rig FRANCC. As last step, comparisons between PythonDAQ and commercial DAQ solutions, as well as the data acquisition software of a major aero-engine manufacturer are made.

NOMENCLATURE

CFD	Computational fluid dynamics
CGNS	CFD general notation system
CSV	Comma-separated values
DAQ	Data acquisition
FRANCC	Fundamental research and new concepts compressor
GUI	Graphical user interface
HDF	Hierarchical data format
MDF	Measurement data format
OPC UA	Open platform communications – unified architecture
SI	Système international d'unités

INTRODUCTION

Measurement setups for turbomachinery test rigs generally have a high number of channels, whereas, in modern setups, most of them come directly from sensors with digital data output, the rest indirectly from A/D-converters. The measurement data is then transferred to computers for further processing. As raw sensor data is often not directly human-interpretable, characteristic turbomachinery quantities are computed live from the sensor readings. The data is stored and displayed to the user as the last step.

To fulfill these tasks, the designer of a test rig has the following two options: Either use a specially designed commercial data acquisition software, which is generally expensive, or implement its own code. The latter is not a trivial task resulting in a large personal workload.

With this paper, a shortcut to the development process is provided. PythonDAQ is an open-source software package that intends to cover the whole workflow from data acquisition and storage until postprocessing. The software is designed in a modular way, such that different measurement setups can be built from an existing catalog of classes, avoiding starting from scratch. Features are kept as generally usable as possible, but focus is set on turbomachinery applications.

DATA ACQUSITION AND PROCESSING CYCLE AND POSTPROCESSING

To derive the software layout, the general data acquisition and processing cycle and the postprocessing workflow are discussed in this section. First of all, sensors are set up to correct operating parameters. The data acquisition itself is normally set up as a cyclic task, who's loop is started after the setup. The tasks inside the loop are shown in Figure 1 and can be divided into four steps:

First, the raw data is acquired from sensors via digital communication interfaces like Ethernet or serial interfaces with different kinds of protocols.

Then, sensor corrections are made - most commonly with a linear calibration applying gain and offset. Sensors that have a non-linear behavior, e.g., resistive temperature sensors, ([1] Sec. 3.2) should be compensated using non-linear calibration curves derived from multipoint calibration to minimize the error.

In some applications, the calibrated readings are enough to fulfill the measurement task. This

[1] Corresponding author. E-Mail: daniel.jaeger@tum.de

Pisa, Italy
28 – 30 September 2022

especially applies for simple measurement configurations or setups where no online visualization is needed, so that further data reduction is done in postprocessing steps only. In more complex setups, the calibrated sensor readings are often not directly human-interpretable. For example, the pressure readings in turbomachinery tests are normally measured relative to ambient pressure, where the relative pressure only is no clear indication of the operating point of the machine. Instead, several steps of computations are needed to derive characteristic operating data like the corrected speed of the machine, corrected massflow as well as pressure and temperature ratios ([2], pg. 18ff). Furthermore, dimensionless quantities like Reynold's and Mach numbers and flow and work coefficients can be computed. The online computation and visualization of this data is needed for the operation of the test rig. The steps of computations will be called online computations throughout the rest of this paper to distinguish them from computations made during postprocessing.

As last step of the cycle, the measured data is stored for postprocessing and visualized in live to the operators of the test rig.

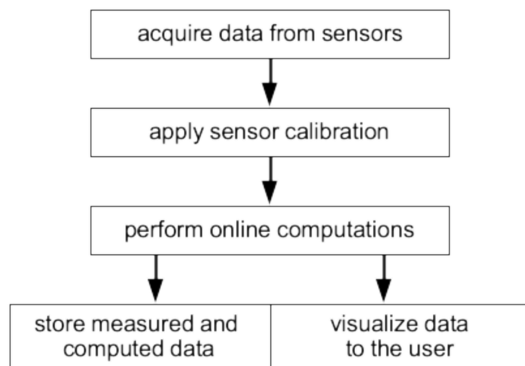

Figure 1: Steps performed during one DAQ cycle

After completing a measurement, the data is postprocessed. The basic steps of postprocessing in turbomachinery applications are: (cf. [2])

- Reading measurement data files.
- Statistically evaluating the data, e.g., by computing mean values and standard deviations.
- Computing further quantities from the acquired sensor readings and online computation results.
- Combining different measurements, e.g., spatial averaging of data from probe measurements.
- Plotting data as time series, XY plots, contour plots, etc.

SOFTWARE ARCHITECTURE

After discussing the principal measurement workflow in the previous section, the software architecture of PythonDAQ is now derived. As the name suggests, PythonDAQ is a pure Python software package which consists of several sub-packages and modules. They are discussed in detail. During development, care is taken to keep the number of dependencies at a minimum. All parts of the software are implemented in a platform-independent way, so that PythonDAQ can theoretically be used on any platform and operating system. Up to now, the software has been tested on Debian, Ubuntu, MacOS and Windows.

Generally, the software is designed as a server-client architecture. The server handles the whole data acquisition and processing cycle except the visualization. Data is continuously made accessible via an OPC UA server. The visualization is implemented as a client accessing the live data from the server. The idea behind this split is that the server application can run on a computer which is placed physically close to the device-under-test enabling the usage of serial communication or PC-based DAQ cards. The client on the other hand is intended to run on a PC or Laptop inside the test rig's control room. See ([1] Ch. 1) for a discussion of different system layouts.

Postprocessing applications are detached from the server-client architecture and can either be used in an automated way on the server or manually on any other computer.

Figure 2 shows the software layout of PythonDAQ splitting up the overall software package into the server-side and client and applications. This split is just made for better understanding, there are cross-connections between several sub-packages of the two sides. Python packages are shown in green color whereas specific GUI applications are shown in yellow.

The *dataserver* package is the core of the server side. It includes the *DataServer* class which manages the data acquisition and processing cycle. As cycle timer, the BackgroundScheduler of the Advanced Python Scheduler (apscheduler [3]) package is used. The *DataServer* also starts the OPC UA server, implemented using the python-opcua package [4], enabling remote access to the measurement data. The *dataserver* package also features a trigger server which allows triggering measurements of a predefined number of data points. This feature is especially useful for the traversing of aerodynamic probes.

The *datastorage* package contains several classes to write and read measurement data files in different file formats. Currently, writing measurement data in CSV and HDF5 file format is supported. Furthermore, traversing data can be mapped into a CGNS file which enables the usage of CFD postprocessing tools like ParaView [5]. For the postprocessing of data files from commercial DAQ systems, a reader class for MDF4 data files is implemented.

The *sensors* package contains classes for all sensors which are used for the specific measurement task. Currently, all sensors of the FRANCC compressor test rig (see section below) are implemented, but due to legal reasons (e.g., proprietary communication protocols), not all of them are published. The sensor classes inherit from a base sensor class which provides a uniform interface to the *DataServer*. Each sensor device can consist of multiple channels as this is the usual case for multichannel pressure scanners like the PSI9116 [6]. A linear calibration routine for all channels is applied inside the base class, any other calibration curve has to be implemented inside the specific sensor class.

Classes contained in the *computations* package work similar like the sensors and are used to perform any kind of online computations accessing the latest acquired sensor readings. Currently, the package implements several computations typically used in turbomachinery applications (see section below), but computations for any other kind of application are possible. Due to the persistence of the computation class instances, it is also possible to store data between different processing cycles, as needed to implement filters.

The *communication* package contains implementations of communication protocols which are not part of the basic Python installation. Currently, it holds an implementation of the Modbus RTU protocol [7] over different communication interfaces.

Finally, utility classes and functions which are used inside the *sensor* and *computation* classes are placed in *utils*. These are functions to convert engineering units, fluid property classes for an ideal gas and humid air, statistical methods, etc.

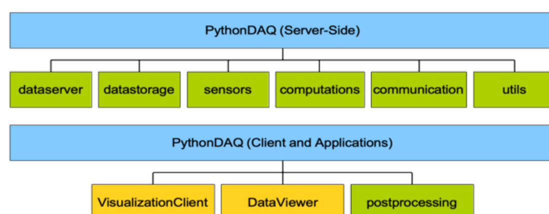

Figure 2: Software structure of PythonDAQ

A measurement application is set up by creating a setup script in Python. First, an instance of the *DataServer* class is created. Then, all sensors are defined by creating the appropriate class instances and setting their respective properties. The sensors are assigned to the server with the addSensor()-method. The same procedure is repeated for online computations. As last step, the data logging and measurement triggering is defined before calling the server's start()-method. The script is called from a Python console returning as soon as all setup was successful and the acquisition cycle is running. The application is then running in the background. This enables using the same Python console for manually triggering measurements or other tasks. In order to stop the program, the server's stop()-method is called. Figure 3 shows a simple server setup containing one sensor and data logging in the CSV format.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from dataserver import DataServer
from sensors import SensorSim

# Initialize the data server
server = DataServer()

# Initialize a simulation sensor
sensor_1 = SensorSim()

# Add the sensor to the data server
server.addSensor(sensor_1)

# Configure a permanently writing logfile
server.addWriter("logfiles/logfile.csv")

# Start the data server
server.start()
```

Figure 3: Setup file for a basic server

On the client-side, especially the *VisualizationClient* is worth mentioning. It is a GUI application that allows connecting to the server and visualizing live data as text or in different kinds of graphs (time series, XY graphs, etc.). The graphical user interface is implemented using the PyQt5 library [8].

For the postprocessing of measurement data, two packages exist: The *postprocessing* package contains routines for the reduction of data (cutting, statistical evaluation), the combination of data (combining single point measurements), as well as postprocessing routines for aerodynamic probes. Furthermore, plot routines are contained in this package. All classes and routines can be used from a postprocessing script. It is still written manually for the specific measurement task, but due to the usage of predefined classes, the amount of code in the script is far less than writing a complete postprocessing procedure from scratch.

The *DataViewer* is a PyQt5-based application to visualize recoded measurement data. It is intended that the program will be extended to a full postprocessing tool incorporating the routines from the *postprocessing* package.

CURRENT STATE OF DEVELOPMENT AND LIMITATIONS

The implementation of PytonDAQ is not finished yet, thus still undergoing continuous development. The most recent stable release of the software can be downloaded from the repository at [9].

To the current date, a synchronous implementation of the data server is completed and

ready for usage. Synchronous implementation means that the sensors are polled for data, one at a time, before starting the computations one by one. This procedure is perfectly fine for applications with a limited number of sensors or if the actual execution time is far less than the desired cycle time.

Most of the sensors with digital communication interface respond within a few milliseconds after the request. Anyway, the communication latency of sensors in large turbomachinery test setups can sum up to a level which is not acceptable anymore. This observation is consistent with the general statements given in ([1] pg. 12). Due to the fact that most of the overall execution time is spent waiting for the sensors to respond, parallelization is the way to overcome this issue. Therefore, an improved implementation of the *DataServer* will be implemented in the future. The idea is to send the request command to all sensors first and then wait for the responses to come in via the communication interfaces. The latency of the slowest sensor is still limiting the overall execution time, but the waiting time is spent in parallel and thus, not adding up anymore. Such a procedure can be implemented using Python's *asyncio* package [10].

Online computations still need to be executed in series due to the fact that one computation may require the results of a previous one. Anyway, even in large setups, the computations take far less time than the sensor communication as can be seen for the FRANCC test case shown below.

Per design, the current version of PythonDAQ is limited to one rate of the whole measurement and processing cycle. For the application in turbomachinery test rigs, there are three scenarios where this limitation may be problematic:

1. There is one single sensor with a high latency limiting the acquisition rate of the whole setup.
2. A higher acquisition rate is needed for a few sensors of the setup.
3. It is intended to acquire and process data from high frequency measurements (more than 100 Hz).

The first scenario can be resolved by wrapping the specific sensor instance inside a new periodic thread running at lower acquisition rates. The upsampling to the overall cycle rate can be done applying a sample-and-hold procedure. A wrapper class for handling this scenario will be implemented in the future.

The second scenario is more complex because it involves downsampling the data and – which is even more problematic – requires a separated routine to store the measurement data at the higher rate. Thus, it is subject to future development.

The third scenario involves handling very high amount of data and thus, needs blockwise data processing, buffering and special precautions against data loss. Therefore, it is currently not planned to support the acquisition of high frequency measurements within PythonDAQ.

The implementation of sensor classes is complete and ready for usage. Anyway, if new types of sensors are needed for a specific test rig, these can be added to the *sensors* package at any time. Please refer to the source code [9] for a complete list of the currently supported sensor types.

The computations currently count, among others, with a number of computations especially designed for turbomachinery test rigs: [11]

- *CorrectedPressures* computes the absolute pressures of sensors adding the sensor readings to a given reference pressure or reference pressure sensor. Furthermore, the effects of geodetic pressure differences between the measurement positions and sensor positions are compensated.
- *Rake* is a class to handle data from pressure and temperature measurement rakes as they are typically used in the inlet and outlet of turbomachinery test rigs [2].
- *CompressorLevel* computes averaged flow quantities at one stage of a turbomachinery test rig.
- *Compressor* is used for overall operating data of a whole compressor like the total pressure ratio, reduced speed and massflow, among others. The outputs of this computation are required to determine the operating point of the machine.
- *DimensionlessCoefficients* computes overall dimensionless quantities of the compressor like the flow and work coefficient.

The *VisualizationClient* currently features a structured overview of all available channels, the possibility to display data as numerical values as well as plotting data over time or comparing two channels in a XY-plot. At the moment of writing this publication, the possibilities to customize the plots are still limited to the scaling of the axes. In the near future, there will be the possibility to choose the plot color and line style, customizing the legend and plotting data from a CSV file. This last feature is especially useful to display reference data from a previous test run or from numerical simulations and observe any differences during the test.

The *DataViewer* currently just features plotting the measured data, but will be extended to a full postprocessing tool at a later point of development.

TEST AT THE LOW-SPEED COMPRESSOR TEST RIG "FRANCC"

After discussing the principles of the software, an application at the low-speed compressor test rig FRANCC at the Chair of Turbomachinery and Flight Propulsion, Technical University of Munich, is now presented and discussed.

The *DataServer* is installed on a PC running on Debian, being connected to the sensors which count with a digital data output. The total number of physical channels is 359. Table 1 shows the distribution of the channels on the individual type of measurement devices.

Table 1: Physical measurement channels

Type of Sensor	# of Devices	Total # of Channels
Pressure	16	308
Temperature	2	30
Massflow	1	10
Torque	1	6
Humidity	1	4
Speed	1	1
Total	**22**	**359**

From the physical measurements, live computations are made as discussed above, adding 355 computed channels, which results in a total of 714 channels. Table 2 shows that the majority of computed channels come from the pressure referencing and correction, whereas calculating turbomachinery-related operating values is the minor part.

Table 2: Computed channels

Type of Computation	# of Class Instances	Total # of Channels
CorrectedPressures	27	257
Rake	6	60
CompressorLevel	2	16
Compressor	1	18
Dimensionless-Coefficiens	1	4
Total	**37**	**355**

The server works with a loop/cycle time of 1 s, but only approx. 450 ms are spent in active state, i.e., data acquisition and processing. The data acquisition takes most of the time with approx. 430 ms, whereas the online computations and data storage take only approx. 20 ms. A permanently active logfile in CSV or HDF5 format is used to capture the entire test run. Additionally, measurements of 30 samples each are taken by manual triggering when measuring steady-state operating points.

For the traversing of aerodynamic probes, the in-house developed program *PythonDAQ-TraverseControl* is used. This program controls the whole probe movement process according to a predefined traversing grid. *TraverseControl* connects with the data acquisition via a TCP/IP connection, triggering a measurement of defined number of samples at each traversing point. The connection is enabled by activating the *TriggerServer* feature inside PythonDAQ.

In order to display live data to the operators of the test rig, the *VisualizationClient* is used on a different machine which is located in the rig's control room. It is running on Ubuntu Desktop. Figure 4 shows the main parts of a typical turbomachinery visualization containing exact textual data output for the compressor's operating point (corrected massflow, total pressure ratio, corrected speed in percent of design speed and the polytropic efficiency). The total pressure ratio is shown also in a graph plotted over the corrected massflow.

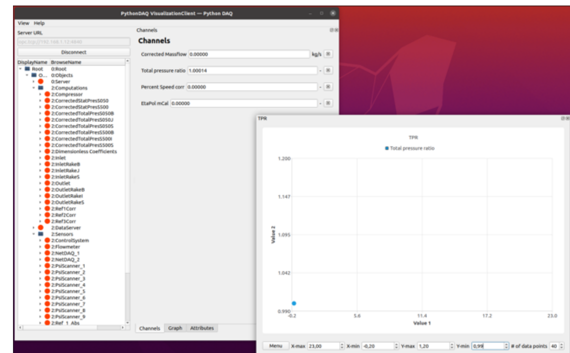

Figure 4: VisualizationClient showing operating data of the compressor test rig

The relatively low sampling rate of currently 1 s is suitable for all steady-state measurements. Anyway, for taking measurements with aerodynamic probes, this may be a problematic limitation: In order to statistically analyze measurement data assuming a normal distribution of data points (considering 95% point of Student's t-distribution t_{95}≈2.0), one measurement at one specific position of the probe should be repeated at least 30 times ([2] pg. 38). This results in 30 seconds + movement time + settling time for each position at which the probe is measuring ([2] pg. 65). For the analysis of the flow field inside a turbomachine, a relatively fine measurement grid is needed, resulting in a high number of probe positions and a very high overall measurement time considering the low data rate. If the acquired number of samples is reduced, the measurement time is decreased, but the level of confidence is decreased as well (cf. [12] Annex G). It has to be decided individually, whether the decreased level of confidence is tolerable or not. This explains, why the development of parallel data acquisition, as discussed in the previous chapter, will be the focus in the near future.

COMMERCIAL DAQ SOLUTIONS

The capabilities and the case setup of PythonDAQ is now compared with two commercial DAQ solutions, namely LabVIEW by National Instruments [13] and DASYLab by National Instruments/ measX [14].

LabVIEW is a graphical programming environment specialized on measurement data

acquisition. The software is very common among the teaching and research in universities. Function and class libraries exist inside the environment for the DAQ hardware provided by National Instruments. Furthermore, many vendors of sensors with digital data output provide libraries to use their hardware within a LabVIEW program which frees the user from implementing communication protocols and raw data conversion. Another advantage is that graphical user interfaces are easy and fast to implement. This makes LabVIEW the first choice in many labs. [13]

The drawback is that LabVIEW is a programming environment and not a ready-to-use DAQ system, thus, the backbone of a DAQ solution (DAQ loop as described above) has to be implemented from scratch for each specific measurement task. Due to the graphical programming, adding or removing features to an existing program can easily end in messy code if no precautions are taken.

DASYLab on the other hand is a ready-to-use DAQ solution. For the data input, many standard protocols are already implemented. Furthermore, there is full support for the hardware by National Instruments and many other vendors of DAQ hardware. A measurement application can be set up in a graphical environment from predefined blocks for the used hardware, visualization and data storage. For hardware which is not supported out of the box, custom blocks can be created using Python programming language. [14]

The advantage of DASYLab compared to LabVIEW is that it is especially designed for DAQ applications, thus, the DAQ loop control does not have to be implemented manually. Furthermore, visualizations can be easily created. The price of approx. 1800€ for the full license [15] is relatively low for a commercial DAQ solution. Disadvantages are that generally no postprocessing and no online processing routines for the needs in turbomachinery applications are available in DASYLab. As a result, there is again the need for custom programming. Also, the user is limited to the Windows operating system.

There is numerous other commercial DAQ software of which the majority is designed to work with the vendor's own hardware. Thus, they are not suited for a highly heterogenous measurement setup as most commonly encountered in turbomachinery test rigs.

COMPARISON WITH INDUSTRY SOLUTION

Now, the capabilities of PythonDAQ are compared with the industrial DAQ solution used by a major aero-engine manufacturer at their turbomachinery test rigs.

The DAQ system consists of four major parts: The raw-data acquisition, main program, high-speed acquisition and the visualization. All parts are distributed along different computers running on Windows operating system.

The raw-data server acquires the data from all sensors with digital data output at a regular rate. The data is then written to a permanently writing logfile as well as made accessible via network. In contrast to PythonDAQ, no calibration and data conversion to SI unit is performed inside this part of the system. The configuration is made either directly using a graphical user interface or indirectly via configuration tables.

The main program handles all the online computations. It accesses the data from the network, applies computations and then publishes the data again on the network. This part performs computations in the following steps: As the first step, data is transformed to the correct engineering (SI) units by using unit conversion formulae and calibration charts for the specific measurement device. Reference pressures from the reference sensors can be added to differential measurements in order to obtain the absolute pressure at a specific measurement position.

As second step, sections inside the turbomachinery are defined for calculating average pressures, temperatures, etc. The section can consist of different types of probes. Also, different types of averages can be selected.

As third step, performance data is calculated considering the whole turbomachinery or just single stages. There is also the possibility to integrate user-defined calculations inside the program. All raw and computed data is written to a logfile providing redundancy to the raw-data acquisition logfile. Steady-state average values can be recorded over a predefined period of time.

Up to this point, the capabilities of the industry solution are very similar to the current implementation of PythonDAQ. Some additional features show the long development process the industry solution has gone through:

All configurations for the industry solution are made from a GUI being able to check and visualize the configuration while making changes. In contrast, PythonDAQ uses a Python setup file which has the advantages of being easily traceable by tools like git and being robust to further development. Anyway, editing a setup script manually is not that intuitive as using a GUI and is also riskier to configuration mistakes.

Inside the main program, reference conditions can be defined. While the test is running, the data is constantly compared to these reference conditions in terms of absolute values as well as in observing changing rates. This makes it easy to assure keeping the machine in steady-state and repeatable conditions. Such a feature is not yet implemented in PythonDAQ.

Furthermore, the main program has the possibility to connect to a database, where all

steady-state measurement points are stored together with different user-defined events. This automatically creates a logbook-style record of the whole test campaign making postprocessing and analysis easy and comprehensive.

Lastly, there is the possibility to re-do all computations of a whole test (single data point up the whole logfile), which may be necessary e.g., when correcting a calibration error in one of the measurement devices.

The high-speed data acquisition is capable of collecting data from vibration sensors, unsteady pressure sensors, etc. at high rates. Inside this program, online computations like calibration, checks and a highly elaborated set of signal processing and analysis routines are available. The data is logged to binary files and provided at low data rates via network for further usage inside the main program or visualization. As discussed above, high-speed data acquisition is intentionally left out of PythonDAQ, but the data from such a system can be still fed into a setup via OPC UA, Modbus TCP or any other digital data connection at low data rates.

As last part of the industry solution, the visualization is now discussed: The visualization tool is capable of connecting to multiple data ports of online data like the raw-data server, the high-speed data acquisition and the main program. Also, offline data from pre-recorded data files and from the previously mentioned data base can be visualized, making it a very universal tool. The data is visualized either as numeric values, as time series, XY plots comparing two channels, and in 2D profile and contour plots taking into account the geometry of the test vehicle. Compared to the industry solution, the *VisualizationClient* of PythonDAQ is just used for online visualization and still counts with less possibilities. Especially plots taking into account the geometry (2D profiles and contours) are not implemented yet.

As overall view it can be seen that the acquisition and processing possibilities of PythonDAQ are comparable to the industry solution. Features that are missing for the individual measurement task can be easily added due to the modular structure. The graphical setup makes the industry solution more intuitive to use and more robust against failures and misconfiguration. Also, the possibilities for visualization are more developed.

One major difference can be seen in the structure of the different programs. PythonDAQ is strongly object-oriented storing all relevant information of a sensor (position, probe type, …) inside the sensor's class instance. All computations can access this information through the program which results in a more centralized overall program structure. Instead, the industry solution is dataflow-oriented so that the raw-data acquisition only sees the sensor's communication interface. The main program instead just uses the data from the tag code on the network interface not caring about the source of data or the sensor any more.

CONCLUSION AND OUTLOOK

The measurement data acquisition and processing software PythonDAQ is introduced in this paper. At the current stage of development, it contains all features necessary to be used at turbomachinery test rigs like the low-speed compressor FRANCC. The data acquisition, online computation and data storage capabilities are running stable and reliable. A visualization tool allows the monitoring of measured and computed values throughout the test. A graphical viewer for data analysis as well as basic postprocessing routines exist.

PythonDAQ is still undergoing continuous development. The next features to be integrated into the software are:

- A more elaborated error handling and logging to improve the robustness against misconfiguration or hardware failures.
- Parallelization of the data acquisition using *asyncio*.
- Improving the capabilities of the *VisualizationClient*, especially enabling more customization for graphical display of data. Adding the possibility to use a pre-configuration avoids setting up the environment at each program start.
- Enhancing the postprocessing capabilities to different kinds of aerodynamic probes.
- Extending the *DataViewer* to a full postprocessing tool.

All developments reaching a stable state will be published at the online repository of the software at [9].

ACKNOWLEDGMENTS

PythonDAQ is developed in the context of the research project “Unsteady Tandem Flow”, funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 420268957 and the FVV eV – 6013600 from 2020 until March 2022.

Furthermore, the Chair of Turbomachinery and Flight Propulsion at Technical University of Munich is acknowledged for the financial support during the development time of PythonDAQ extending the period of the project “Unsteady Tandem Flow”.

Lastly, the authors would like to thank all developers of PythonDAQ. A full list of the contributors is provided inside the software package at [9].

REFERENCES

[1] Nawrocki, W. Measurement Systems and Sensors. Second Edition. Norwood, MA: Artech House 2016.

[2] Advisory Group for Aerospace Research & Development (AGARD). Recommended Practices for Measurement of Gas Path Pressures and Temperatures for Performance Assessment of Aircraft Turbine Engines and Components. AGARD-AR-245. Neuilly Sur Seine: AGARD 1990.

[3] Grönholm, A. Advanced Python Scheduler. github.com/agronholm/apscheduler. Accessed on 2022-07-15.

[4] FreeOpcUa. Python-opcua. github.com/FreeOpcUa/python-opcua. Accessed on 2022-07-15.

[5] Kitware Inc. ParaView. www.paraview.org. Accessed on 2022-07-15.

[6] Measurement Specialties, Inc., a TE Connectivity Company. Ethernet Intelligent Pressure Scanner. 9116 NetScanner System. Datasheet. www.te.com/commerce/DocumentDelivery/DDEController?Action=showdoc&DocId=Data+Sheet%7F9116%7FA1%7Fpdf%7FEnglish%7FENG_DS_9116_A1.pdf%7FCAT-SCS0002. Accessed on 2022-08-12.

[7] Modbus Organization, Inc. Modbus application protocol specification V1.1b3. 2012. www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf. Accessed on 2022-07-15.

[8] Riverbank Computing Limited. PyQt Documentation v5.15.4. www.riverbankcomputing.com/static/Docs/PyQt5/. Accessed on 2022-07-15.

[9] Jäger, D., et. al. PythonDAQ-Public. Source code repository for the public version of the PythonDAQ project. gitlab.lrz.de/ltf-experimentatoren/pythondaq-public.

[10] Python Software Foundation. Asyncio - Asynchronous I/O. docs.python.org/3/library/asyncio.html. Accessed on 2022-08-12.

[11] Selmayr, L. M. Ausarbeitung und Implementierung eines Systems zur Online-Verarbeitung von Messdaten an einem Turbomaschinenprüfstand. Term Paper (in German). Garching: Technical University of Munich 2022.

[12] Joint Committee for Guides in Metrology (JCGM). JCGM 100:2008. Evaluation of measurement data - Guide to the expression of uncertainty in measurement. Corrected version 2010.

[13] National Instruments Corp. What is LabVIEW? www.ni.com/en-gb/shop/labview.html. Accessed on 2022-08-12.

[14] measX GmbH & Co. KG. DASYLab - Versatile software for data acquisition. www.measx.com/en/products/software/dasylab.html. Accessed on 2022-08-12.

[15] BMC Solutions GmbH. DASYLab Mess-und Steuersoftware. www.bmc.de/DASYLab-FULL. Accessed on 2022-07-19.