HELAO User Guide

Jackson Flowers
KIT
November 9, 2021

Introductory Note

This document is intended to serve as a definitive guide to using the HELAO software framework for any laboratory automation application. It should lay out in detail, with examples, how to operate or interface with HELAO and write code that fits cleanly into the HELAO framework. When presenting standards and practices, the text should distinguish between what is required and what is merely best practice, or what is deeply integrated into the framework and what could be modified without much issue. It should quickly become clear that there is substantial room for these processes to be further improved, streamlined, and automated, and so it is my fervent hope that this document will be continually updated to incorporate new features of HELAO. However, this contrasts with my other fervent hope that HELAO development should no longer comprise a significant part of my doctoral work. It remains to be seen how these tensions will shake themselves out.

Overview

HELAO is a tool for laboratory automation, with particular focus on modularity, data management, and the integration of active learning. An operational HELAO instrument consists of an array of servers run and hosted via the fastAPI and uvicorn packages in python. These servers are organized in a three-level hierarchy: driver servers, which expose the functions of a single device via HTTP get request, are designed for maximum modularity. Above these are action servers, which host more complicated device functions, or functions dependent on one or more devices being in a certain physical configuration. These action functions generally call driver functions from one or more drivers, and are themselves exposed as HTTP get requests. Finally, the real workhorse of the system is the orchestrator server, which calls on the exposed functions of one or more actions servers in order to automate experiments.

The orchestrator accepts series of action functions ("experiments") via

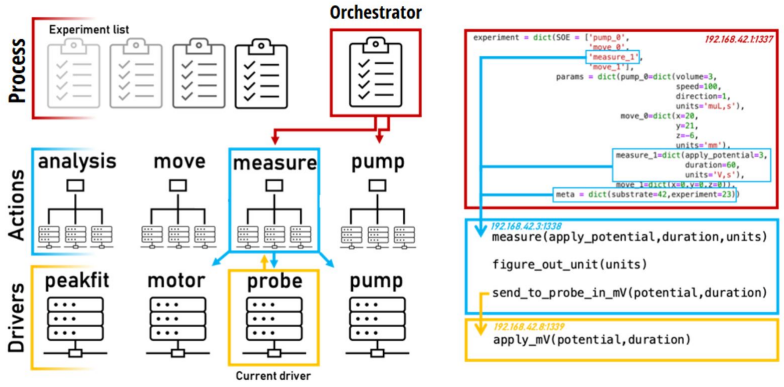


Figure 1: Diagram of HELAO. Figure by Helge Stein. Taken from the submitted but currently unpublished paper: "Enabling modular autonomous feedback-loops in materials science through hierarchical experimental laboratory automation and orchestration".

a single HTTP post endpoint, and performs these experiments in the order in which they are received by making get requests of the appropriate action severs. It is also possible, using the optional "thread" parameter of this endpoint, to send an experiment to be performed on a parallel thread, allowing coordinated orchestration of multiple instruments simultaneously. In addition to action functions, an experiment can also include calls to perform one of the orchestrator's internal functions, which exist to provide more complex scripting options to experiments.

It is our hope to add more internal commands as time goes on to allow more complicated experiments to be more easily scripted, and to put together some sort of graphical interface for monitoring the orchestrator and sending scripts to it, so that a user can design and send an experiment or series of experiments without needing to write python code. Currently, however, we use python files which we have taken to calling "process files" to generate experiments as python dictionaries and send them to the orchestrator.

As each action function returns its result to the orchestrator, the results of that function are automatically saved under a heading in an HDF5 file. When a series of experiments is complete, the orchestrator can automatically upload this HDF5 file to KIT's Kadi4Mat cloud storage system. We have developed a system of conventions as to how this information is stored, but hope that a future graphical interface could provide greater opportunity for user customization and visualization the data structures.

Finally, we have found that passing some kind of configuration information to our servers is essential. Currently, we have set things up so that we have one configuration file, containing a single python dictionary, per computer. Each server which can be hosted from that computer receives the config file on initialization, and a key specifying which entry within the file contains the information for that server. We also have a primitive graphical interface for automatically starting servers with the appropriate inputs.

The above paragraphs have laid out all of the basic elements of HELAO. What will follow now are short guides on how to use or write code for each of

these elements. Presumably, not every user is interested in every single one of these; so, with that assumption, the guides are written with the hope that each will, as much as possible, stand on its own – though that means there will likely be some repetition between them.

Writing a Driver Server

While we have written driver servers for all physical devices that we use with HELAO, they do not communicate directly with the orchestrator, and so it is possible to leverage much of the power of HELAO without doing so. A user preparing their own drivers for their own actions could in theory design them as they see fit. However, especially for code intended to be used more broadly, we recommend the following standards.

We typically split the driver into two python files, one containing a class exposing the device methods in python, and another to instantiate this class and host the methods of that class on a server. For a device "{device}" we would call these files "{device}_driver.py" and "{device}_server.py". Our main design philosophy when writing the drivers is maximum reuseability and modularity. Thus, the author of a driver should keep it relatively simple, such that all functions remain valid regardless of what context the device is used in. Code that is dependent on the device working in a specific configuration and/or with other specific devices should be left for the action server, which will make HTTP get requests of the more elementary driver functions [1]. Beyond that, the driver should simple be a class and accept any necessary configuration information as a dictionary argument to the "__init__" method.

```python
import serial

class pump():
    def __init__(self,conf):
        self.pumpAddr = conf['pumpAddr']
        self.ser = serial.Serial(conf['port'], conf['baud'], timeout=conf['timeout'])

    def primePump(self,pump:int,volume:int,speed:int,direction:int=1,read:bool=False):
        # pump is an index 0-13 indicating the pump channel
        # volume is the volume in µL, 0 to 50000µL
        # speed is a variable going from 20µl/min to 4000µL/min
        # direction is 1 for normal and 0 for reverse
        self.ser.write(bytes('{},PON,1234\r'.format(self.pumpAddr[pump]),'utf-8'))
        self.ser.write(bytes('{},WFR,{},{}\r'.format(self.pumpAddr[pump],speed,direction),'utf-8'))
        self.ser.write(bytes('{},WVO,{}\r'.format(self.pumpAddr[pump],volume),'utf-8'))
        return self.read() if read else None

    def runPump(self,pump:int,read:bool=False):
        self.ser.write(bytes('{},WON,1\r'.format(self.pumpAddr[pump]),'utf-8'))
        return self.read() if read else None
```

Figure 2: First 20 lines of a simple driver class: pump_driver.py. Full code can be found on github.com/helgestein/helao-pub.

The structure of the file hosting the server, "{device}_server.py", is quite a bit stricter, and so I defer to Fig. 3 to demonstrate the various features of

[1]The driver server shown in Fig. 3 has its methods called by the action server shown in Fig. 5 – comparing these might help clarify any confusion, though the question of what code belongs in the action server and what code in the driver server remains even for us something of a gray area, and the use of one's own judgement is encouraged.

a typical driver server file. Beginning at the top, FastAPI and uvicorn must be imported to host the web server, and the file must also be able to find and import the driver class and the instrument config file. Looking at lines 7-12 of Fig. 3, one can see that the config file and driver class must be in the appropriate subdirectories of the local HELAO distribution. Additionally, looking at lines 10 and 12 specifically, the choice of which config file to use and which key within that config file corresponds to the device are passed as arguments to the server upon instantiation. This is done for the sake of flexibility in how a device is configured, and sending these arguments in this way is also necessary for compatibility with our rudimentary GUI. The sections on configuration files and the GUI can be consulted for additional information.

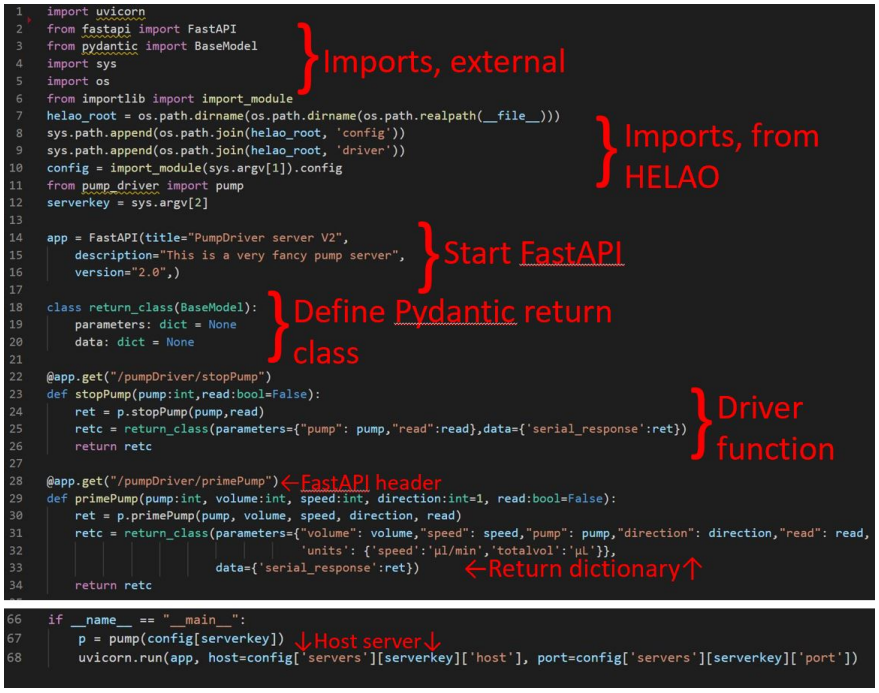


Figure 3: Example code taken from a typical driver server, pump_server.py, with features labelled. Full code can be found on github.com/helgestein/helao-pub.

Lines 14-16 of Fig. 3 should be copied to initialize the FastAPI server, though the text is not important, and lines 18-20 may optionally be copied for the return class as well – though the Pydantic return class is somewhat a living fossil in our code, and one could instead opt to replace all instances of the return class with a normal dictionary. Looking at the functions beginning on lines 22 and 28 of Fig. 3, each should call a single function from the driver object and expose it with FastAPI. The FastAPI endpoint of a function "{function}" of device "{device}" should be "/{device}Driver/{function}", for consistency – see the section on writing configuration files for more information. Additionally, we wrap any output (or lack thereof) from the driver function in a dictionary (or our return class, which will be received by higher-level servers as a dictionary) formatted to maximize the clarity and transparency of our

operations [2]. Finally, the "__main__" space of the function should initialize the driver object and then host the FastAPI server with uvicorn.

A successfully running server can be instantiated in a terminal as seen in Fig. 4. The terminal will receive information on requests processed by the server from uvicorn for as long as the server is active.

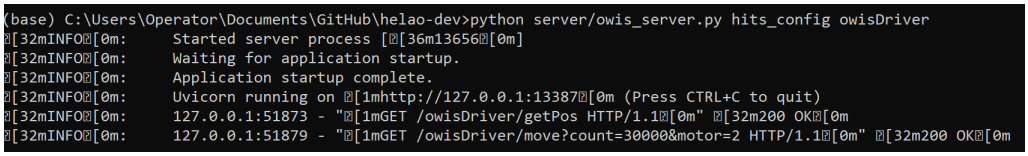


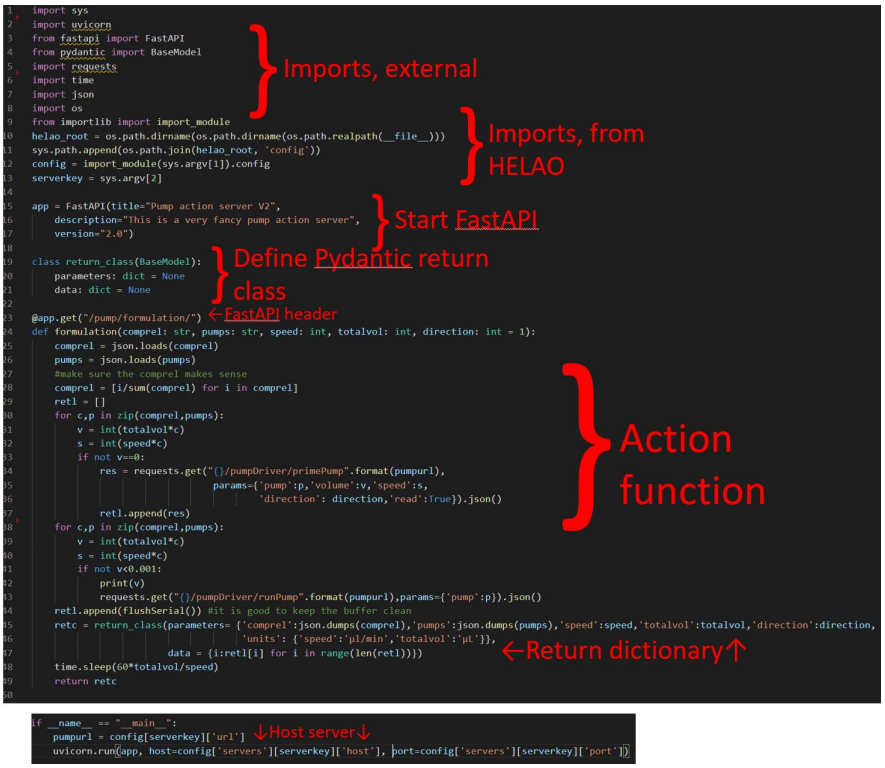
Figure 4: A server for a motor driver running in anaconda terminal. Full code can be found on github.com/helgestein/helao-pub.

Writing an Action Server

Action servers host the device functions which are directly called by the orchestrator. The orchestrator should be perfectly compatible with any action server that exposes its functions via HTTP get requests, and which return a JSON-safe and HDF5-safe dictionary. This gives the orchestrator substantial flexibility in interfacing with external code. We nonetheless do have a number of standards for how we write our action servers, which will be outlined below.

We typically write a single action server per device, but this standard is somewhat arbitrary, and one could hypothetically split one's action functions into however few or many servers are desired. Fig. 5 demonstrates part a typical action server for a pump. Readers of the previous section will notice many similarities between Fig. 5 and the driver server presented in Fig. 3. Looking to the first few lines of Fig. 5, we first import FastAPI and uvicorn to create and host the web server, and then ensure that the server can find and import the configuration file. In most cases, the configuration information associated with an action server will at minimum include the IP address(es) and port(s) of the driver server(s) from which it makes requests. Looking at lines 12 and 13 of Fig. 5, the config file to be used and the key of that config file which pertains to this particular action server are passed as arguments to the action server when it is started. This is done for the sake of flexibility in how an action server is configured, and sending these arguments in this way

[2]The standards for how these return statements are formatted are a bit complex. At the top level, one must always have two keys: "parameters" and "data". The "parameters" key holds a record of the parameters that were sent to the function, and contains a dictionary with the parameter names as keys and parameter values as values. The "data" key contains a dictionary of data values returned with key names clarifying the type of data. Alternatively, a function that has no parameters or no data to return may have a None value under the respective key instead. Finally, if relevant, the dictionaries under both "parameters" and "data" must have a key "units", which contains a dictionary where the keys are other keys under the "parameters" or "data" dictionary, and the values are the units in which those values are denominated. If a function has only a single parameter or returns only a single value, the value under the units key can simply be the units of that value. An example of how this shakes out in practice can be seen in lines 25 and 31-33 of Fig. 3




```
1  import sys
2  import uvicorn
3  from fastapi import FastAPI
4  from pydantic import BaseModel
5  import requests
6  import time
7  import json
8  import os
9  from importlib import import_module
10 helao_root = os.path.dirname(os.path.dirname(os.path.realpath(__file__)))
11 sys.path.append(os.path.join(helao_root, 'config'))
12 config = import_module(sys.argv[1]).config
13 serverkey = sys.argv[2]
14
15 app = FastAPI(title="Pump action server V2",
16     description="This is a very fancy pump action server",
17     version="2.0")
18
19 class return_class(BaseModel):
20     parameters: dict = None
21     data: dict = None
22
23 @app.get("/pump/formulation/")
24 def formulation(comprel: str, pumps: str, speed: int, totalvol: int, direction: int = 1):
25     comprel = json.loads(comprel)
26     pumps = json.loads(pumps)
27     #make sure the comprel makes sense
28     comprel = [i/sum(comprel) for i in comprel]
29     retl = []
30     for c,p in zip(comprel,pumps):
31         v = int(totalvol*c)
32         s = int(speed*c)
33         if not v==0:
34             res = requests.get("{}/pumpDriver/primePump".format(pumpurl),
35                                params={'pump':p,'volume':v,'speed':s,
36                                        'direction': direction,'read':True}).json()
37             retl.append(res)
38     for c,p in zip(comprel,pumps):
39         v = int(totalvol*c)
40         s = int(speed*c)
41         if not v<0.001:
42             print(v)
43             requests.get("{}/pumpDriver/runPump".format(pumpurl),params={'pump':p}).json()
44     retl.append(flushSerial()) #it is good to keep the buffer clean
45     retc = return_class(parameters= {'comprel':json.dumps(comprel),'pumps':json.dumps(pumps),'speed':speed,'totalvol':totalvol,'direction':direction,
46                                      'units': {'speed':'µl/min','totalvol':'µL'}},
47                         data = {i:retl[i] for i in range(len(retl))})
48     time.sleep(60*totalvol/speed)
49     return retc
50
51 if __name__ == "__main__":
52     pumpurl = config[serverkey]['url']
53     uvicorn.run(app, host=config['servers'][serverkey]['host'], port=config['servers'][serverkey]['port'])
```

Figure 5: Example code taken from a typical action server, pump_action.py, with features labelled. Full code can be found on github.com/helgestein/helao-pub.

is also necessary for compatibility with our rudimentary GUI – the section on configuration files and the GUI can be consulted for additional information. Lines 15-17 should be copied to initialize the FastAPI server, though the text is not important, and lines 19-21 may optionally be copied for the return class as well, though our use of Pydantic is a bit of a living fossil, and one could instead opt to replace all instances of the return class with a normal dictionary.

The function beginning on line 23 of Fig. 5 is a typical example of an action function. This functions directs a multi-channel pump, in which the channels are assumed to be connected to a single tube on the output side, to pump the separate channels at varying speeds so as to produce the desired formulation. After calculating the pumping speeds and pump times needed to produce the desired total volume, overall speed, and formulation, it initializes the channels one-by-one and then activates them by making requests to the driver server of the pump. As demonstrated here, an action function should take the simple, configuration-independent building blocks offered by the driver servers to construct more complex functions designed for one or more devices arranged into a certain way, though action functions can be much more rudimentary than that presented here, sometimes just exposing a driver server function directly to the orchestrator. One can also make requests to multiple different driver servers within a single action function, as we often do when substantial feedback between devices is desired.

Again, as in the driver server, each function in the action server must have a function decorator for FastAPI, as can be seen on line 23 of Fig. 5.

For a function "{function}" an action server "{device}_action", we would use "{device}/{function}" for the FastAPI endpoint, for consistency – see the section on writing configuration files for more information. Additionally, we collect the output from the driver function(s) called in a dictionary (or our return class, which will be received by higher-level servers as a dictionary) formatted to maximize the clarity and transparency of our operations. [3] Finally, the "__main__" space of the function should host the FastAPI server with uvicorn. An action server which has been properly started and is running in a terminal window is shown in Fig. 6.

Figure 6: The terminal window for a running action server for a motor. Full code can be found on github.com/helgestein/helao-pub.

Writing a Configuration File

Configuration files are passed to the orchestrator, the driver servers, and the action servers on initialization, and store configurable information related to these servers. We typically have a single configuration file per computer, which is utilized by all servers hosted from that computer. The information in the config file can include such things as default speed settings on a motor for its driver server, or an electrochemical cycling procedure for a potentiostat action server. They also must include the IP addresses and ports of all servers hosted on the computer or to which servers on the computer make requests.

The config file should be written as a python file containing a single python dictionary. This dictionary will have a number of keys, most of which must store dictionaries themselves. Fig. 7 shows the first 20 lines of a typical config file. The first key of this dictionary, as can be seen, should always be the

[3] The formatting of the return statement for the action server is even more complex than that of the driver server, and the reader is advised to first read the footnote on the formatting of driver server return statements before continuing with this one. Lines 25-27 of Fig. 5 demonstrate one such return statement. Like the return statement of the driver server, the return dictionary of an action function has a key "parameters" and a key "data", and the standards for the value under "parameters" are exactly the same as described in the previous footnote. The value under "data" is more complicated, as we both want to ensure that all returns statements from all driver commands called by the action are included under this key, but also want the ability to highlight specifically relevant results in a clean way. The author of an action function thus has several options when deciding how to structure the return statement. Firstly, if there is just one request to a driver server function in the action function, one may simply return the result of that under "data". Or, if more than one request is made, they may be collected into a "list" – a dictionary with integers as keys, and returned that way. Alternatively, if some special result is to be highlighted, one should put a dictionary with the keys "raw" and "res" under "data". Under the "raw" key should go one of the options mentioned previously, and under "res" should go a dictionary of labelled values of interest. This dictionary, if necessary, should also contain the "units" key as described in the previous footnote.

```
1  config = dict()
2
3  config['servers'] = dict(pumpDriver=dict(host="127.0.0.1", port=13370),
4                           pump=dict(host="127.0.0.1", port=13371),
5                           autolabDriver=dict(host="127.0.0.1", port=13394),
6                           autolab=dict(host="127.0.0.1", port=13375),
7                           kadiDriver=dict(host="127.0.0.1", port=13376),
8                           kadi=dict(host="127.0.0.1", port=13377),
9                           forceDriver=dict(host="127.0.0.1", port=13378),
10                          force=dict(host="127.0.0.1", port=13379),
11                          forcesdcDriver=dict(host="127.0.0.1", port=13338),
12                          forcesdc=dict(host="127.0.0.1", port=13339),
13                          orchestrator=dict(host="127.0.0.1", port=13380),
14                          langDriver=dict(host="127.0.0.1", port=13381),
15                          lang=dict(host="127.0.0.1", port=13382),
16                          minipumpDriver=dict(host="127.0.0.1", port=13389),
17                          minipump=dict(host="127.0.0.1", port=13344),
18                          analysis=dict(host="127.0.0.1", port=13369),
19                          ml=dict(host="127.0.0.1", port=13363))
20
```

Figure 7: The first 20 lines of the config file "sdc_1.py". Full code can be found on github.com/helgestein/helao-pub.

"servers" key, under which we have a key for each server hosted on the computer or to which servers on the computer make requests. Under each of these keys is a dictionary, with the IP address of the server stored as a string and the port stored as an integer. The key for a driver server for device "{device}" should be "{device}Driver", and the key for an action server for that device should be "{device}"[4]. If multiple identical driver servers or action servers are to be used, the user can distinguish between them within the config file by adding a colon and an integer at the end of their respective keys. So if one wanted to communicate with two instances of "pump_driver.py", which has the key "pumpDriver" in Fig. 7, one would replace line 3 of Fig. 7 with two separate keys "pumpDriver:1" and "pumpDriver:2", each with its own IP address and port.

After the "servers" key, we then have top-level keys for each of the servers hosted on the computer, which use the same naming convention as shown under the "servers" key in Fig. 7. Under these keys will be the config dictionaries for each of these servers. The authors of these servers are free to structure the config dictionaries as they see fit. We find that these configurations are usually quite simple, but an example of a relatively elaborate one is shown in Fig. 8. Note that the config dictionary for an action server must include the addresses of any driver servers accessed by that action server.

Finally, config file will generally have two more keys, "instrument" and "launch", as shown in Fig. 9. "instrument" contains only a string with the name of the "instrument" – the name for the collection of devices currently being automated by the computer. This is used by the orchestrator for data management. The "launch" key is necessary to use the basic GUI we have written for starting and stopping servers. The keys for servers associated with driver servers should be put in the list under "driver", the keys for actions under "action", etc.

[4]This is identical to the convention we use for FastAPI endpoints within the servers themselves – refer to the sections on writing driver and action servers.

```
config['owisDriver'] = dict(serials=[dict(port='COM4', baud=9600, timeout=0.1),dict(port='COM11', baud=9600, timeout=0.1),
                                     dict(port='COM13', baud=9600, timeout=0.1)],
                                     currents=[dict(mode=1,drive=80,hold=40),dict(mode=0,drive=50,hold=30),dict(mode=0,drive=50,hold=50)],
                                     safe_positions=[1500000,None,600000])
config['owis'] = dict(coordinates=[None,None,
                                   {"sem":{"x":68,"y":91,"theta":pi,"I":True,"z":10.5},
                                    "fto":{"x":0,"y":0,"theta":0,"I":True,"z":0},
                                    "oneoff":{"x":0,"y":0,"theta":0,"I":True,"z":0}}],
                                   roles=['x','y','raman'],url="http://127.0.0.1:13387",ramanurl="http://127.0.0.1:13383")
```

Figure 8: Config dictionaries for "owis_driver.py" and "owis_action.py" in the config file "hits_config.py". For the driver, this includes the default serial connections, motor currents, and optional safe positions for three motors controlled by the driver. The action server, configuration information includes the URLs for the driver servers it is contact with, the roles for each motor within the instrument, and definitions for several instrument coordinate systems. Full code can be found on github.com/helgestein/helao-pub.

```
config['launch'] = dict(server=['autolabDriver', 'kadiDriver', 'langDriver', 'minipumpDriver', 'pumpDriver'], #, 'forceDriver'
                        action=['analysis', 'autolab', 'kadi', 'lang',
                                'measure:1', 'measure:2', 'minipump', 'ml', 'pump'], #, 'force'
                        orchestrator=['orchestrator'],
                        visualizer=['autolab_visualizer'],
                        process=[])
config['instrument'] = "sdc"
```

Figure 9: Lines 224-231 of the config file "sdc_1.py". Full code can be found on github.com/helgestein/helao-pub.

Using the Rudimentary HELAO Interface

While we have not yet found the time, energy, or expertise to develop a full interface for HELAO, we found that, given the large number of uvicorn servers involved in a typical HELAO system, it was quite a pain to open up a separate terminal window for each one, and so a basic interface has been written to start each server with the appropriate configuration data in a single window. The interface is run from a terminal window, and takes the name of a config file as argument. Assuming that name matches the name of a config file in the config folder of the local distribution of HELAO, and that the "launch" key of that config file has been set up properly, the interface will initialize with a list of buttons to open and close the servers defined in that config file, automatically passing the proper config information to each server. Fig. 10 is an image of the interface with several servers started. The terminal window in which the interface was opened will automatically receive the uvicorn messages of all servers started, as can be seen on the left of Fig. 10. Thus, the user should use the interface to open and close servers, and the adjoining terminal window to monitor the status of the servers and the requests made to them. The interface will also automatically track whether servers remain online.

Writing a Process File – Sending Experiments to the Orchestrator

We hope at some point to have a graphical interface which removes or lessens the amount of direct python scripting required to run the HELAO orchestrator,

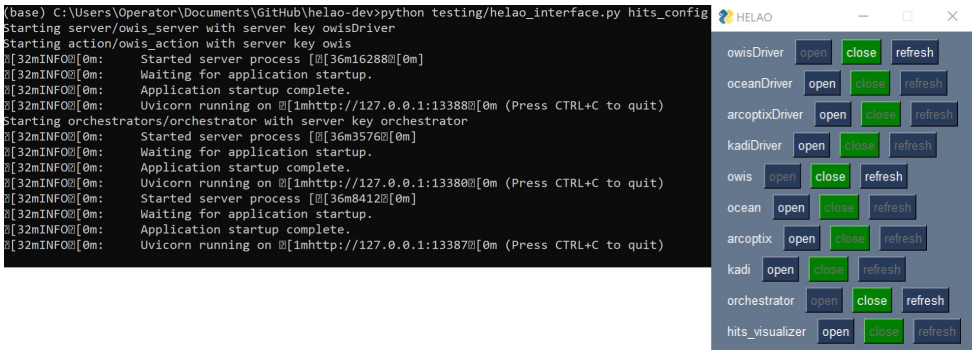


Figure 10: An image of the HELAO GUI (helao_interface.py) with terminal window. Full code can be found on github.com/helgestein/helao-pub.

but, for now, the only way to send experiments to the orchestrator is to write a python script to generate the experiments as python dictionaries and send them as parameters to the "orchestrator/addExperiment" endpoint of the orchestrator. What follows is a description and examples of this scripting, demonstrating both what can currently be done, and how to do it.

We refer to a single list of action functions sent to the orchestrator by a single function call as an "experiment". Again, lists of actions are sent to orchestrator by making an HTTP post request to the orchestrator function "orchestrator/addExperiment", with the experiment as a parameter. An experiment is formatted as a dictionary with three set keys: "soe", "params", and "meta". Any example of such a dictionary is shown in Fig. 11. Looking at this figure, the "soe" key contains a list of action functions in the order they are to be performed, the "params" key contains a dictionary with these action functions' names as keys, and dictionaries of the parameters to be passed to them as values, and, finally, the "meta" key accepts a dictionary of metadata to be passed into the result of the experiment. Action functions names under "soe" and "params" can optionally be followed by an underscore and an integer, to distinguish between multiple calls to the same function within an experiment.

```
dict(soe=['lang/moveWaste_0', 'minipump/formulation_0', 'lang/RemoveDroplet_0','lang/moveSample_0',
          'lang/moveAbs_0','lang/moveDown_0','autolab/setcurrentrange_0',
          'autolab/measure_0', 'lang/moveRel_0', 'lang/moveWaste_1'],
     params= dict(moveWaste_0= dict(x=0, y=0, z=0),
                  formulation_0= dict(speed= 70, volume= 80, direction= 1),
                  RemoveDroplet_0= dict(x=0, y=0, z=0),
                  moveSample_0= dict(x=0, y=0, z=0),
                  moveAbs_1 = dict(dx=dx, dy=dy, dz=dz),
                  moveDown_0 = dict(dz=0.213, steps=120, maxForce=0.44, threshold= 0.320),
                  setcurrentrange_0= dict(crange='100uA'),
                  measure_0= dict(procedure="ca", setpointjson= json.dumps({'applypotential': {'Setpoint value': pot},
                                  'recordsignal': {'Duration': ca_time}}),
                                  plot="tCV",
                                  onoffafter="off",
                                  safepath="C:/Users/LaborRatte23-3/Documents/GitHub/helao-dev/temp",
                                  filename="substrate_{}_{}_{}.nox".format(substrate, procedure, j, ca_time),
                                  parseinstructions='recordsignal'),
                  moveRel_0= dict(dx=0, dy=0, dz=-4),
                  moveWaste_1= dict(x=0, y=0, z=0)),
     meta=dict()
```

Figure 11: Example of an experiment dictionary for an electrochemical measurement.

When using HELAO, we send one or more of these experiments to the orchestrator, where they are, by default, performed in the order that they are received. We typically write python files to automatically generate a series of these dictionaries and send them to the orchestrator in a series of requests all

at once. One may choose to put all the actions one wishes to perform into one single list, or send a number of smaller experiments as separate requests – the choice is arbitrary, and will affect the structure of the data file generated, but not much else. As an example, when working with substrates, we typically send all the actions associated with a single point on the substrate as one experiment.

Figure 12: A simple but complete process file. Each experiment in the list sends a motor command to move a sample stage to a certain point beneath a Raman probe, and then sends a Raman commmand to take a spectrum. The parameters of the motor action function give a coordinate to move to, and express that that coordinate is in the coordinate system of the Raman probe. The parameters for the Raman action function are exposure time for the spectrum measurement, and a filepath to save a backup copy of the spectrum to.

A more complete example of a process file is shown and described in Fig. 12; though this is still a very rudimentary script, preparing a series of Raman measurements on a grid defined across a substrate. One can see that the first few lines of the process send an experiment containing only the command "orchestrator/start". This is an internal orchestrator command which announces that a new sequence of measurements is beginning, and sets up the data management accordingly. As parameters, "start" accepts arbitrary metadata to be passed to the orchestrator – more details in the section on orchestrator data management. One special parameter it can optionally take is "collectionkey", which defines a naming convention for the file which receives the data. After that experiment is passed to the orchestrator, Fig. 12 shows a "for" loop in which pairs of actions with parameters are generated and sent to the orchestrator. Of particular note is that the last such experiment will contain the orchestrator command "orchestrator/finish". This command is also related to data management, and announces to the orchestrator that the current sequence of measurements is complete and can now be uploaded to the cloud; it takes no parameters. Every process file should begin by sending a "start" command, and end with a "finish" command.

Data Management in the Orchestrator

The orchestrator takes dictionary return statements from the action server functions and places them under headers of an HDF5 file associated with a series of experiments. It is essential to understand how the orchestrator structures the data it receives from the action functions in order to to find

the data you need within to file, to manage the size and shape of this file, and to effectively script more complicated processes. We often call our HDF5 files "session" files, as they correspond to a single experimental session. This section will attempt to describe how these files are structured, how to control their structure, and how to read them.

The anatomy of a session file is described in Fig. 13. The action function returns are stored under headers based on their functions names. Any trailing underscore introduced in process files is included to distinguish between multiple calls to the same function within an experiment, so an action "{device}/{function}_{j}" would result in an action stored under the key "{function}_{j}". These keys are themselves stored under headers associated with each experiment. So, each call of "orchestrator/addExperiment" generates a new empty header in the session file. These files are indexed from zero and according to the thread they are running on. So the second experiment of a session in thread zero would have the header "experiment_1:0"[5]. Between the "experiment" headers and the top level of the file, we have the "run" headers. A new "run" header is generated whenever the orchestrator receives an "orchestrator/start" command[6]. Thus, the first experiments added to the session will go under the header "run_0". If this run does not finish and the process is restarted, the orchestrator will receive a new "start" command without previously having received a finish "command". This will signal it to prepare a new header, "run_1", instead of preparing a new HDF5 file. After a process finally completes successfully, the "finish" command will be received and no new runs will be added to the session file. It should also be noted that each level of header in the HDF5 contains a "meta" key. For the top-level header and for the action, this key only contains the date at which the session begins or the time at which the action completes. Arbitrary metadata can be introduced on the run level through the parameters of the "start" command, and on the "experiment" level through the "meta" key of the experiment dictionary.

For manipulating the HDF5 files, h5py is the primary python package to use, but has a bit of a learning curve. The hdf5dict python package, while providing much less functionality, is a package written on top of h5py that is much easier for a beginner to use. The code:

```
import hdfdict
d = hdfdict.load({filepath},lazy=False,mode='r+')
```

Will load a full HDF5 file into python as a dictionary, at which point its contents can be inspected by navigating through the keys (loading may take some time for large files). I am aware that the development environment Spyder might also include some useful functionality for visualizing the interior

[5]Every experiment is assigned to a thread signified by an integer value, by default thread zero. Orchestrator thread indices are central to how we run multiple parallel experiments. See the section on multithreading for more information.

[6]In the course of working with HELAO, we often found that one begins an experiment, something goes a bit wrong, and one wants to interrupt it and tweak it slightly before trying again. In the interest of full traceability, we do not want to throw out the data. Thus, we segregate it by defining a new header to contain the new data.

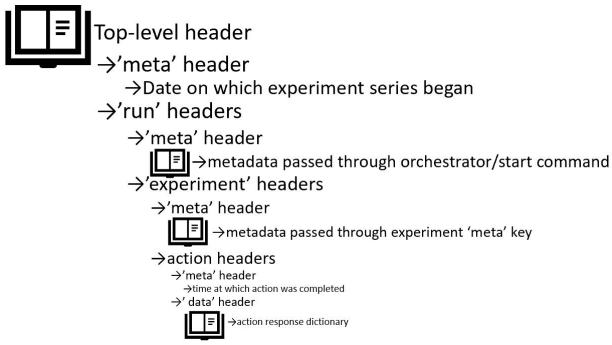


Figure 13: Anatomy of the HDF5 files generated by the orchestrator.

of and HDF5 file, but do not have any direct experience with it. I hope that in the future we are able to offer more comprehensive guidance on visualizing HDF5 files and converting data from them into other standard data formats.

Advanced Process Scripting – Multithreading

We are capable of writing somewhat more elaborate scripts than that shown in Fig. 12. There are two more built-in orchestrator functions: "wait" and "modify", which can be included as parts of an experiment. Furthermore, "orchestrator/addExperiment" has an optional second parameter, "thread", which enables the orchestrator to process multiple series of experiments simultaneously. Scripting with these features is notably more complicated than the linear series of experiments described previously. To provide a reasonably simple example, the dummy script I used to test these features is shown in Fig. 14. This process sends three experiments to the orchestrator, one of which is 30 actions long, one of which is 29 actions long, and one of which is 19 actions long. Note that the posts to the orchestrator on lines 20, 34, and 46 set the "thread" parameter to an integer value. When the orchestrator receives an experiment, it checks this "thread" value. If there is a list of experiments sent with this thread value already running, it appends it to that list. Otherwise, it starts a new list of experiments in parallel to those that are currently running. If no thread is included, the value is assumed to be 0. Since these experiments all have different thread values, they will begin running in parallel to one another as they are received by the orchestrator.

The function "dummy/lmao" in Fig. 14 takes a single integer 't', and waits for a number of seconds equal to 't'. "dummy/fakeml"[7] returns two random integers under the keys "val1" and "val2" under its "data" return key. Effectively, what this whole script does is instantiates two threads that "run" for five seconds each. A third thread detects when both have completed yields two random numbers which are picked up by the other two threads with the "modify" command. They then each "run" again for a random number of seconds, and wait on more numbers from the third thread, which in turn waits for the first two to complete. Thus, we are capable of synchronizing multiple

[7]Please forgive the names of these functions, this was simply to test that I could synchronize a machine learning server to two devices, and send them new experiment parameters from it.

```python
import requests
import sys
sys.path.append("config")
from jackspc_config import config
import json

if __name__ == "__main__":
    soe = ["orchestrator/start","dummy:1/lmao_0"]
    for i in range(9):
        soe += [f"orchestrator/wait_{i}",f"orchestrator/modify_{i}",f"dummy:1/lmao_{i+1}"]
    soe += ["orchestrator/finish"]
    params = {'start':{'collectionkey':'dummytest1'},'finish':None}
    params.update({"lmao_0":{'t':5}})
    params.update({f"lmao_{i}":{'t':'?'} for i in range(1,10)})
    params.update({f"wait_{i}":{'addresses':f"experiment_0:3/fakeml_{i}"} for i in range(9)})
    params.update({f"modify_{i}":{'addresses':f'experiment_0:3/fakeml_{i}/data/data/val1','pointers':f'lmao_{i+1}/t'} for i in range(9)})
    meta = {}
    experiment = dict(soe=soe,params=params,meta=meta)
    requests.post(f"http://{config['servers']['orchestrator']['host']}:{config['servers']['orchestrator']['port']}/orchestrator/addExperiment",
                  params= dict(experiment=json.dumps(experiment),thread=1))

    soe = ["dummy:1/lmao_0"]
    for i in range(9):
        soe += [f"orchestrator/wait_{i}",f"orchestrator/modify_{i}",f"dummy:1/lmao_{i+1}"]
    soe += ["orchestrator/finish"]
    params = {'finish':None}
    params.update({"lmao_0":{'t':5}})
    params.update({f"lmao_{i}":{'t':'?'} for i in range(1,10)})
    params.update({f"wait_{i}":{'addresses':f"experiment_0:3/fakeml_{i}"} for i in range(9)})
    params.update({f"modify_{i}":{'addresses':f'experiment_0:3/fakeml_{i}/data/data/val2','pointers':f'lmao_{i+1}/t'} for i in range(9)})
    meta = {}
    experiment = dict(soe=soe,params=params,meta=meta)
    requests.post(f"http://{config['servers']['orchestrator']['host']}:{config['servers']['orchestrator']['port']}/orchestrator/addExperiment",
                  params= dict(experiment=json.dumps(experiment),thread=2))

    soe = []
    for i in range(9):
        soe += [f"orchestrator/wait_{i}",f"dummy:1/fakeml_{i}"]
    soe += ["orchestrator/finish"]
    params = {'finish':None}
    params.update({f"fakeml_{i}":None for i in range(9)})
    params.update({f"wait_{i}":{'addresses':[f"experiment_0:1/lmao_{i}",f"experiment_0:2/lmao_{i}"]} for i in range(9)})
    meta = {}
    experiment = dict(soe=soe,params=params,meta=meta)
    requests.post(f"http://{config['servers']['orchestrator']['host']}:{config['servers']['orchestrator']['port']}/orchestrator/addExperiment",
                  params= dict(experiment=json.dumps(experiment),thread=3))

```

Figure 14: Dummy script demonstrating some of the more advanced scripting techniques available to the orchestrator.

parallel experiments and passing information between them, so long as we script carefully to begin with.

As can be seen in Fig. 14, this functionality hinges entirely on the "orchestrator/modify" and "orchestrator/wait" commands. These commands both function by looking into the HDF5 file being written by the orchestrator and, in the case of the "wait" command, simply checking whether one or more headers exist, or, in the case of the "modify" command, pulling a value from under that or those header(s). The "modify" command then puts that value or values into the parameter dictionary of the experiment. A parameter must have been initialized as '?' in the process to receive a value from "modify". Parameters initialized as '?' can be seen in lines 14 and 28 of Fig. 14.

The parameters of 'orchestrator/modify" and "orchestrator/wait", visible on lines 15, 16, 29, 30 and 42 of Fig. 14, may themselves look rather complicated. Both have a parameter "addresses", which can be either a string or a list of strings which point to headers which will be written in the HDF5 file where the orchestrator stores the results of the process. "modify" also takes a parameter "pointers", which is a string or list of strings pointing to locations in the "params" subheader of the experiment dictionary. So, for example, it is clear on line 16 of Fig. 14, that that modify command is intended to modify the parameter 't' of the action "lmao_{i+1}" in this experiment. Properly writing to the "addresses" parameter, on the other hand, requires understanding the structure and naming conventions used by the orchestrator as described in the previous section. The orchestrator automatically tracks the file and "run" header under which each orchestrator thread is currently saving data. The

strings under "addresses", therefore, are the path to a value within the run header. This can easily be predicted, as the name of the action and the structure of the action dictionary are known, and the "experiment" header can be inferred from the distance between the current experiment and the most recent "start" command in the process file, as well as the thread index.

A few more notes about multithreading follow. Firstly, we have not yet set up any serious requesting handling on our driver and action servers, so that if a server receives a request while it is already processing another request, the second request will fail. The orchestrator is smart enough not to send requests to an action from two threads at the same time, but will still run into problems if two separate actions in different threads try to access the same driver at the same time, so it is advised to take care if one is scripting multiple threads with overlapping hardware, though in principle it is possible and hopefully in the future will be a robust feature of HELAO. Another note: if a new thread is instantiated with an index one higher than that of a currently-running thread, it is unnecessary to use the "start" command at the beginning of that thread, and the experiments of that thread will automatically be saved under the same "run" header of the lower thread index. Otherwise, a new "start" command must be called to instantiate a new "run" header. All threads must separately conclude with their own "finish" commands, and only when all threads have signal that they are complete will the session file be considered done and uploaded.

Active Learning with the Machine Learning and Analysis Servers

To be written.