MEASURE

MEASURE is a program written in Python that automates and simplifies the mass calibration process. MEASURE was started by Alexander Moses and continued by Peter Wang. The mass calibration process is a lengthy process that NIST follows to calibrate customer weights, using NIST’s internal weights, which are traceable back to the International Prototype Kilogram (IPK) in Paris. Weights are calibrated on precision balances called comparators. There are many types of comparators for all the weight ranges, from milligrams up to hundreds and thousands of kilograms. There are also comparators that can hold two weights (two-pan), four, or even more. Furthermore, there are automated balances that can measure weights automatically, and manual ones where the reading has to be recorded. No matter the case, the customer weights and the NIST weights must be compared to each other in a certain order, defined by the “design matrix”. There are design matrices for all kinds of situations, whether there is one unknown customer weight or multiple.

When the comparator measurements are being recorded, environmental measurements must be recorded as well. Because this is such a high precision process, a small change in the temperature, humidity, or pressure could impact the result of the mass measurement. The design matrix, with all of its comparisons, decreases the variability and uncertainty in our measurements. For example, when we calibrate customer 100g weights with our own, we often get precision to hundredths of a milligram.

Once all the data has been collected, it has to be sent through the NIST mass code, which processes the data and outputs the final weight corrections and uncertainties. For instance: we get two customer weights that are supposed to be exactly 1 kilogram. When we compare it to our own weights, which we know are 1.00024 and 0.99932 kg (numbers are made up for demonstration), with a design matrix, we calibrate their masses to 1 kg plus or minus a small correction.

Before MEASURE, this process was done all by hand. The data collection had to be by hand, the environmental data collection had to be by hand, the historical data of our own weights had to be looked up beforehand, and all the math in the mass code had to be done by hand.

MEASURE provides a GUI (graphical user interface) that allows the metrologists to conduct a calibration much more easily. MEASURE also includes a MySql database interface, where comparator information, thermometer info, NIST weight info, customer weight info, and much more, are all stored. MEASURE allows metrologists to easily pick which weights they want to calibrate, how they want to calibrate them, and on which balance the weights should be calibrated on. Once all the settings are chosen, MEASURE will then communicate with the comparator through a COM port and automatically compare the weights with each other. At the same time, MEASURE communicates with the thermometer, hygrometer, and barometer to get environmental readings at set intervals. At the end of the automated calibration, two files will be generated: an “.ntxt” file and a “.json” file. The “.ntxt” file is an input file that can be run through the old (Fortran) mass code or the new (Python) mass code. The “.json” file is the condensed output of the new mass code. This “.json” file can be beautified through the GUI to produce a readable output file. If there needs to be adjustments made to the environmental readings (if there is a funky temperature reading, for example), the “.ntxt” file can be revised, and run through the new mass code again to produce a final “.json” file. Similarly, it can also be run separately through the old mass code to produce an output file. A video will be made detailing how to properly set up and use the GUI.

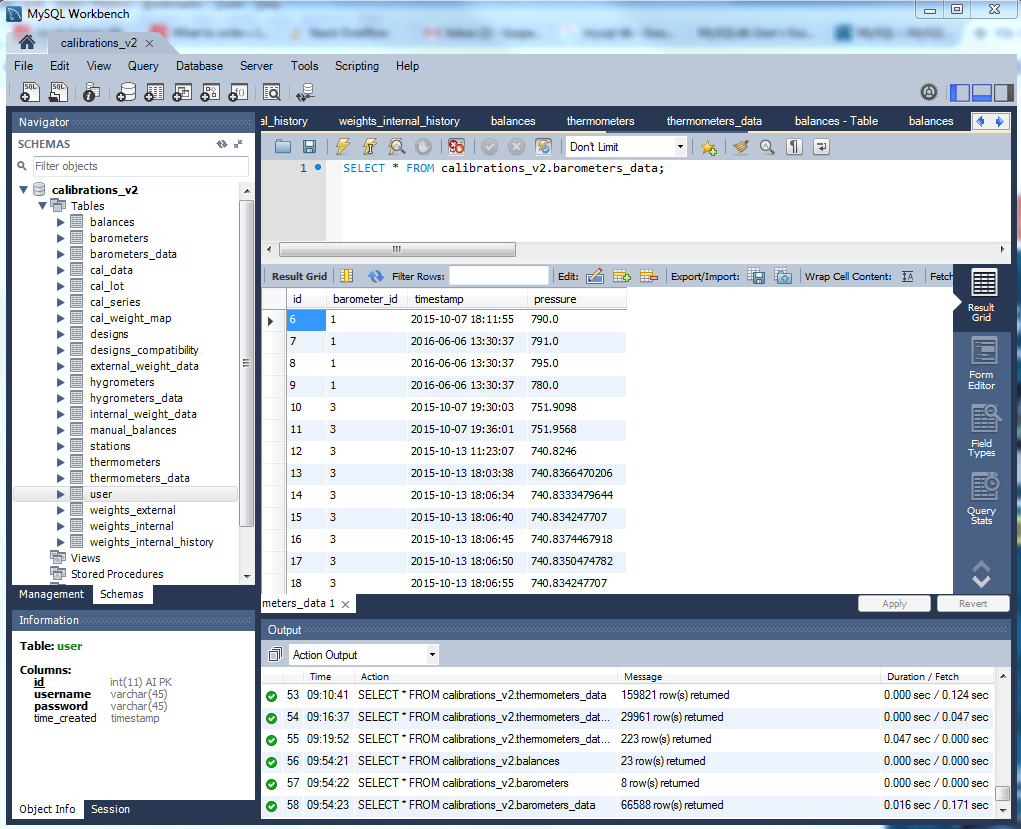
Before any of this can be used, however, Python and the correct packages need to be installed. As of today (8/5/16), the program has not been made into an executable. Python 2.7 was used for this program. Eventually, the syntax might need to be updated for Python 3, but that is far in the future. The main packages that need to be downloaded are:

* PyQt4 – a binding for creating GUIs with Qt, a popular tool for GUI development
* MySqlDB – provides a connection to the MySql database
* PySerial – provides access to the serial ports and communication with instruments
* Numpy – scientific computing package for Python similar to Matlab
* Matplotlib – graphing utility package similar to Matlab

To increase the usage of this program, users need to be able to add balances and commands to the program. To add a balance to the program, first add the balance and its settings in the database under the table “Balances”. Then, in config.py, follow the existing dictionary structures to input the new balance commands. Finally, in the comparator\_matching dictionary, match up the balance id to the set of commands it follows.

BEFORE USING THE PROGRAMS…

**How to use MySql Database**

MySql is an open source database that allows users to safely and quickly store large amounts of data. Both Python programs are connected to MySql. If your computer does not have it already, you should download MySql Workbench here: <http://dev.mysql.com/downloads/workbench/>. This allows you to easily view and alter tables and entries. To use the workbench, you should know a little SQL (Standard Query Language). If you’re not familiar with it, there is a great quick guide here: <http://www.tutorialspoint.com/sql/sql-quick-guide.htm>. Here is a screenshot of what MySql Workbench looks like: 

On the left, you will see a bolded “calibrations\_v2”. This is the name of the database we are using. Under it, you will see all the tables, from “balances” to “weights\_internal\_history”. When you hover over a table with the mouse, you will see three small buttons pop up in the same row. Try clicking on all three to see what happens. At the top of the right half, you will see a box where you can type stuff in. This is where all the SQL commands will go. In the picture, ‘SELECT \* FROM calibrations\_v2.barometers\_data;’ selects all the data from the table called barometers\_data. The data can be seen directly below this. You will see that there are 4 columns: id, barometer\_id, timestamp, and pressure. These are automatically populated by the Enviro program. If you need to make a change to any entry, just double click the entry, make the change, and click Apply in the bottom right to finalize the change.

The great thing about MySql is that you can form relationships between certain entries in one table to certain entries in another table. For example, we have one table called designs, which holds all of the design matrices that can be used in a calibration. We also have a table called designs\_compatibility, which holds the information that tells us which designs are compatible with which types of balances. Instead of having to redefine all the designs again in this table, we can simply refer to the existing designs in the designs table. A 41 design in the designs table might have an ID of 1. Therefore, when we want to say that a 41 is compatible with balance A, we can just say that 1 is compatible with A. In this case, 1 is a foreign key, because designs\_compatibility accesses information from designs. The foreign key/primary key concept is extremely useful in database management.

**How to get the programs working**

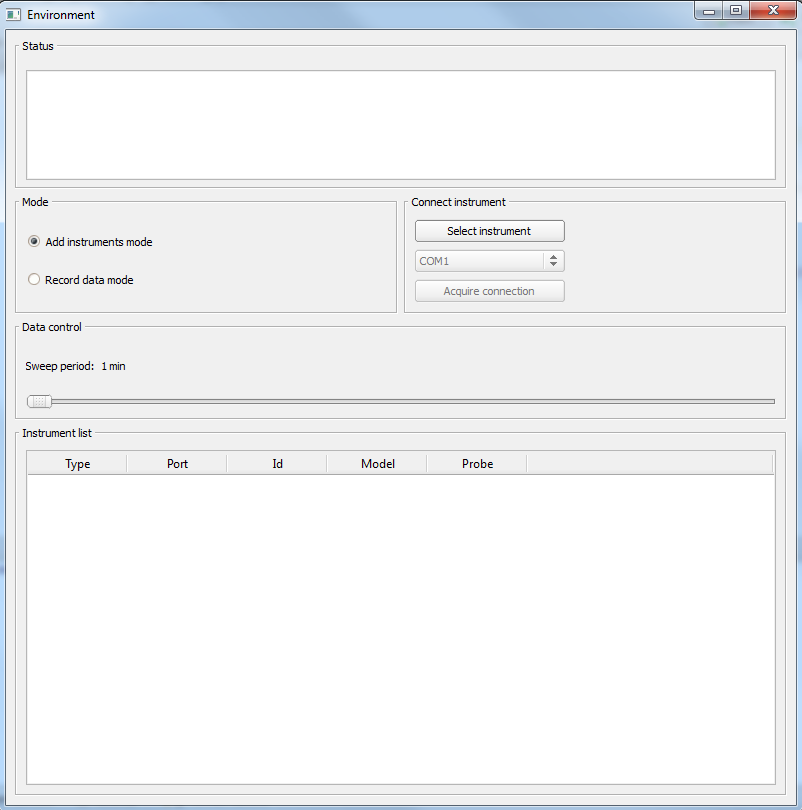
1. If Python 2.7 is not yet installed on your computer, download it from this website: <https://www.python.org/downloads/release/python-2712/>. Follow the instructions to install it properly on your computer
2. If you do not have a Python IDE (Integrated Development Environment) like Geanie or Pycharm, download Pycharm (I prefer Pycharm over Geanie as you can see output more easily). Pycharm can be downloaded here: <https://www.python.org/downloads/release/python-2712/>.
3. Open Pycharm
4. Go to File -> Open to get to the Dialog. Then, get to measure through the path L:\internal\684.07\Mass\_Project\Software\PythonProjects\measure. Click the measure folder and click OK.
5. Go to File -> Settings to get to the settings dialog.
6. Expand Project: measure on the left toolbar, and click on Project Interpreter. At the top, select your Python Interpreter using the drop down menu, or if nothing is present, click the cog and select add local. If you need to add local, find the Python you downloaded in step 1 (usually in your C drive under Python27) and add it.
7. In the same dialog, click the green + sign. In the search bar, type in qtconsole, and click Install Package at the bottom. Then download SQLAlchemy, numpy, pyserial, pip, and matplotlib.
8. After installing the packages, click Apply or OK.
9. Try running main\_ui.py by clicking the green run button at the top right corner. Make sure the program that’s running is actually main\_ui.py (look to the left of the green “play” button to see the name of the running program).
10. If it works, you’re all set!
11. If you get an error that says “\_\_\_\_\_ is not defined” or “\_\_\_\_\_\_ is not installed”, go back into settings and install that package.

Detailed Technical Descriptions of the Programs

If you’re reading this, you’re probably going to be working on the software next. If you just want to know how to use the program to conduct a calibration, please look at the video guide.

There are two main programs that will be used: MEASURE and Enviro. Both of these were designed using Qt Designer, and programmed using PyQt, a set of Python bindings for Qt which was written in C.

Before any calibration, the environmental conditions need to be recorded, so I will start the technical description with an explanation of the environmental program, **Enviro.** This is the program that pulls up the GUI for adding environmental instruments and recording data from them periodically. Personally I didn’t do that much work on this program; it was mostly written and developed by Alexander. However, it is very similar to MEASURE in that it communicates with the environmental devices through serial ports. We used the package PySerial for the communication.



When the main program is run, this is what shows up. There are two modes as seen on the left hand side. These two radio buttons are referred to in the code by recordRadio and addRadio respectively. On the right hand side, the Select Instruments button is referred to as selectButton, and the Acquire Connection button is the acquireButton. The combo box that houses all the COM port selections is called portCombo. In the \_\_init\_\_ function, the combo box is populated with all of the available serial ports. Now let’s step through the \_\_init\_\_ function. We first see that it takes one argument (window), which will be explained later. Super(QObject, self).\_\_init\_\_() initializes the class as a descendant of QObject, a class defined by PyQt. The passed window’s name will be set to the software name which is defined in another file. Uic.loadUi is one of the most useful things in all of PyQt. It loads the xml file that Qt Designer makes into Python, allowing you to access the widgets by their name. We load this into self.ui, so from now on, self.ui will be the object that contains all of the widgets of the entire UI. In the next line, we set the slider’s text to ‘1 min’, as you can see in the above picture. Then, we set the value to the percentage of the maximum value. Next, we initialize the database object relational mapping (ORM, for short). This is the connection between the UI/program and the MySql database. Basically (if you look in database\_orm.py) the ORM establishes a connection with the server, and we create objects that match up with the tables in the database. We use SqlAlchemy to retrieve, add, and alter data in the database from the program.

Then, back in the main program, we initialize some dictionaries and lists for future use. The serial ports are added to the combo box, the widgets are set to the way you see in the picture, a timer is started (use QTimer for GUIs, as they don’t make the browser hang), and the PyQt signals are connected to their respective slots in callback\_connector. To learn more about PyQt signals and slots, read here: <http://pyqt.sourceforge.net/Docs/PyQt4/new_style_signals_slots.html>.

When the instrument and the COM port are selected, the metadata is queried from the database and a connection is made. These are all added to the table below. When all instruments are added and the Sweep radio button is pressed, the program spawns a Sweep Thread. In this thread, the environmental instruments are sequentially queried for their reading, and the reading is then pushed into the database with the time, environmental instrument id, and reading recorded.

When the environmental program is running, you can safely start the measuring program to begin the calibration. measure is much more complicated than the environmental program, but it uses many of the same concepts.

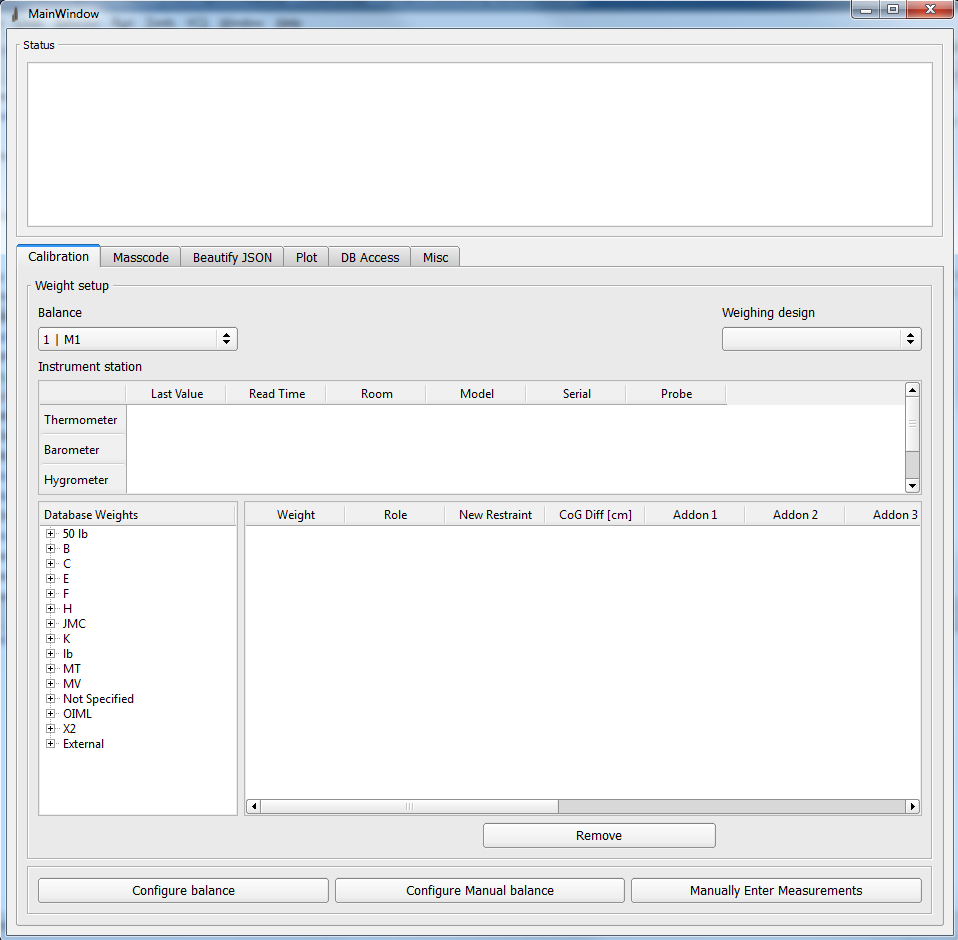
**MEASURE**

Measure is comprised of a main GUI and multiple sub-UIs. Each of these user interfaces has its own class that is sub-classed from either QObject or QWidget. For our uses, it doesn’t really matter which class is the parent class. QWidget is a descendant of QObject, but contains all of the necessary inherited functions we need. When main\_ui.py is run, the MainUI class is initiated with an application (QtGui.QApplication(sys.argv)), and a window (QtGui.QMainWindow()).

When looking at the code, the first thing you will see is the main dictionary (main\_dict) that is initialized. This dictionary contains all meta data for the entire calibrations. It has all the necessary information to create a full output file. It also provides some information that is used in processing data. For example, the design matrix, restraint vectors, check vectors, weight info, and many more are used by the mass code to correct the raw numbers. There are also miscellaneous pieces of information, like balance ID, user ID, station ID, and more, which are used just for the records.

In the \_\_init\_\_ function of MainUI, the Login UI is initiated before the main window even appears (window.show() makes the main UI appear). The login UI checks the username and password against the database. There is no protection against SQL injection yet, and should be added to the code. Once the username and password match, the database connection is initiated. Before the login is successful, the application does not have access to any information in the database.

After the login, the UI is populated using the PopulateUI class. This class populates the balance combo box, the weight tree, the design menu (after a balance is selected), and the weight table (after a design is selected). After this, two queues are instantiated, one for the input files (for the mass code analysis) and one for JSON files (for beautification). The callback connector is then called, which is defined a little further down. Finally, the window is shown, ending the initialization process. Here is a screenshot of the first window you will see:

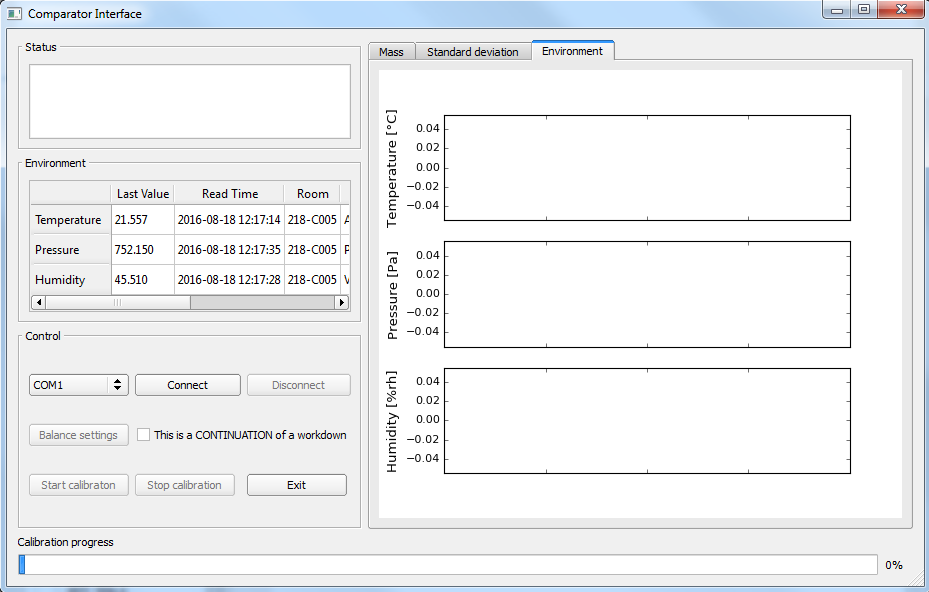


If you look in the code, you will see that I separated the signal slot connections by tab and I tried to order the connections in a logical way. None of the “heavy lifting” is done in the main UI thread. When the program needs to communicate with a balance or analyze data, the information is passed to another thread. For example, in the beautify\_json function (line 254 of main\_ui.py), the list of JSON files is pushed into a thread safe queue, and for each file, a separate thread is started that generates the output file from the JSON file. That way, anything going on in the Main UI won’t hang, as GUIs need to constantly be updated and listening for events.

**The Calibration Tab**

There are three buttons at the bottom of the Calibrations tab of the main window: configure balance, configure manual balance, and Manually Enter Measurements. Each of these brings up a distinct GUI. When any of these three buttons are clicked, the main dictionary is populated by the Populate Dictionary class (populate\_dictionary folder). In here, the files that start with data\_structure are pertinent to main\_dict. Masscode\_dicts and populate\_masscode\_dict are both used later for the new mass code, and will be discussed later.

The dictionary population works by reading the contents of the weight table filled out by the user, and processing them. Firstly, the main weights are read (data\_structure\_weights). The number of columns in the design matrix tells us how many weights are present in the design, so the weight specific parts of main\_dict are initialized to lists with length n, the number of weights. The program goes down the weights column one by one, reading in the text and parsing it. External weights have two pipes in the name, and are processed differently than internal weights, which only have one pipe in the name. E.g. ‘EXT|19|Amgen 100g’ vs ‘24|F 16’. The weight is identified, and the database is queried for the weight information. The restraint and check vectors are then populated in data\_structure\_vectors. The environment-related entries in main\_dict are populated in data\_structure\_enviro, where the instrument data is queried from the database. The centers of gravity are read from the weight table, and the add-on weights are scanned row by row, column by column, and the add-on weight data is queried from the database as well.   
 **Configure Balance**

****

When this button is clicked, an instance of ComparatorUi is created. This is wrapped in a try/except block, because if all of the weights aren’t populated (there are 4 slots but only 3 weights filled), an IndexError will be raised in the Populate Dictionary class. Upon successfully opening, the only buttons that are enabled are the Connect and Exit button. A serial port needs to be connected to using PySerial, as was done in the Enviro program. When a COM port is selected and ‘Connect’ is clicked, the program looks up the balance serial settings, which are taken from the database. This includes the baudrate, parity, bytesize, stopbits, and timeout. Therefore, it is extremely important to make sure the balance settings are stored in the database. Once a balance is connected, the ‘Balance settings’ button will be enabled. The ‘Balance settings’ button can be quite confusing. It is connected to the slot function ‘click\_settings’. From the \_\_init\_\_ function, there is a dictionary called settings\_ui\_dict that contains balance ID numbers as keys and the settings GUI for the value. Right now, there are two balances that use the same settings GUI, AT106HSettings. This is a separate GUI in sub\_ui. When ‘Balance settings’ is clicked, the balance ID is retrieved from main\_dict, and that ID is used to initialize the correct settings GUI. AT106HSettings has slots for the resolution (usually kept on the highest resolution), centerings (the number of times the balance should lift and lower the pan to center the weight), wait time (the amount of time before the calibration should start), stab time (the number of seconds after a weight has been lowered onto the pan before a measurement should be taken), int time (how many seconds the measurements should be taken for, usually 5 seconds), pre runs (number of fake runs before data is collected), runs (number of actual full runs), tare at start (if the balance should be tared at the start), and tare between runs (self-explanatory). These settings are passed back into ComparatorUi.

I don’t see anyone having to make another settings GUI in the future, but there might be a situation where a Sartorius balance has different settings that need to be configured, in which case a separate settings GUI should be made. For now, however, if a new comparator is added, the ID just needs to be connected to the AT106HSettings GUI in the \_\_init\_\_ of ComparatorUi.

There is also a little checkbox for workdowns. This button should only be checked if the current calibration is the continuation of a workdown. When checked a file dialog comes up and asks the user to select the JSON file of the previous (parent) calibration. This takes the “new restraint” from the previous calibration the “restraint” of the current one, and it sets all of the uncertainties to the correct ones as well.

When the settings have been set, the ‘Start calibration’ button is enabled. This is the most confusing part of the program (for me at least). When it is clicked, ‘click\_start’ is called. A dialog is opened so the user can choose where to save the data (self.get\_file\_path()). Then, RecipeMaker class is called. This class is located in the control folder. RecipeMaker takes in a PyQt signal, the connection object, main\_dict, settings\_dict, and instruction\_dict (taken from the config.py file, contains all the balance commands). From these dictionaries, RecipeMaker makes two lists, self.m and self.a (bad names, I know). The first list, self.m, contains the type of command, while self.a contains the actual command. I think of it like a taco: self.m is the shell and self.a is the insides. The different types of ‘taco shells’ are contained in ingredient\_methods.py. There are different methods for each type of command. For example, if you just want to read the balance, you expect a response from the balance after you give it a command. This would be a use of short\_command, which expects a response. On the other hand, if you just tell the balance to close its doors, you don’t expect a response, so you would use short\_command\_no\_resp. There are also methods that wait the wait time and stability (stab) time.

Basically, RecipeMaker creates all of the methods that will be needed to run the calibration. As you can see in the class file, the make\_recipe method outlines the calibration structure. The two important lists, self.m and self.a (list of methods and list of commands respectively), are passed back to ComparatorUi. Then, in another thread, the recipe is executed by recipe\_execute.py. This function just loops through the two lists (self.m and self.a) and executes them pair by pair.

When a measurement is taken, cal\_data\_slot is called. This adds a vertical line to the environmental plot on the right, and it adds the measurement to the data dictionary (data\_dict), which is used for analysis. The format of data\_dict is the same wherever you encounter it: it consists of multiple nested dictionaries. The outer dictionary is the run number, the middle dictionary is the observation number, and the inner dictionary is the individual weights (A1, B1, B2, A2) in the ABBA structure. It looks like this:

{‘run 01’ : {‘observation 01’ : {A1: [measurement, temperature, pressure, humidity],   
 B1: [measurement, temperature, pressure, humidity],   
 B2: [measurement, temperature, pressure, humidity],   
 A2: [measurement, temperature, pressure, humidity]},  
 {‘observation 02’ : {A1: [measurement, temperature, pressure, humidity],   
 B1: [measurement, temperature, pressure, humidity],   
 B2: [measurement, temperature, pressure, humidity],   
 A2: [measurement, temperature, pressure, humidity]},  
 …  
 {‘observation 06’ : {A1: [measurement, temperature, pressure, humidity],   
 B1: [measurement, temperature, pressure, humidity],   
 B2: [measurement, temperature, pressure, humidity],   
 A2: [measurement, temperature, pressure, humidity]}},

‘run 02’ : ….}

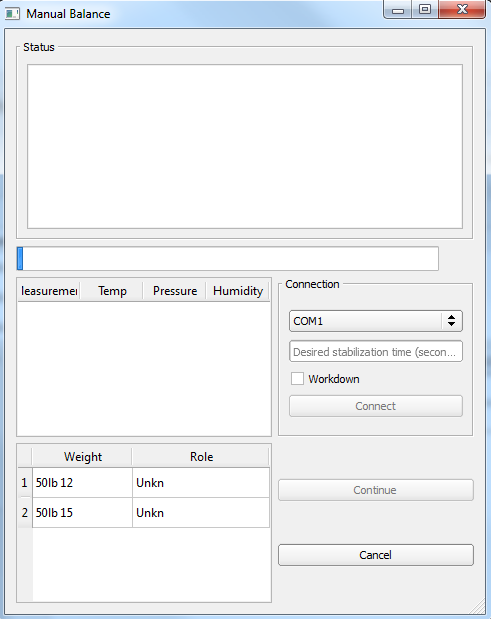
There are six observations for a 4-1 design, but there can more depending on the design matrix. When the dictionary is completely populated, an input file is generated by generate\_input\_file, and the new mass code is run.

The input file has the ending ‘.ntxt’, and can be run through both the old and the new mass code, though we are in the process of validating the new mass code so that we can switch to the new one once and for all. The input file generator is located in the folder file\_generators, and it takes in the main\_dict and the data\_dict and parses those dictionaries and formats the data for input into the mass code.

On the right hand side of the UI, there is a tab widget. Currently, only the environment tab works. This is located in the folder called ‘plots’. We use the Qt extension provided by matplotlib to put the graphs in the GUI. Two things you could work on in the future could be the other two tabs and prettying up

That covers the automatic balances; moving on to the manual balance, we have a button on the main UI that says ‘Configure Manual balance’.

**Configure Manual Balance**

****

When clicked, main\_dict is populated by the Populate Dictionary class, and an instance of ManualBalanceUI is created, just like the ComparatorUi. This brings up a separate GUI which is much simpler. As always, the UI is loaded using uic.loadUi, which allows Python to access the widgets by their names. The connection needs to be chosen and the stabilization time set before the ‘Connect’ button is enabled.

The only complicated thing in this class is the continue button. When the ‘Continue’ button is clicked, the proceed method is called. There are multiple if statements in this method. The first one checked is if the calibration is finished. We can tell if it’s finished if the row\_counter (counts the row of the design matrix we are on) exceeds the number of rows in the actual design matrix. If that is true, the input file is created, and the data is run through the new mass code, just like ComparatorUi. If the calibration is not done, we get the first and second sets of weights from the design matrix. Then, we check if it’s a measuring step or not. The way this UI works is that there are two main steps: putting the weight on the balance, and measuring it. When continue is pressed, one of two things happen: either the UI prompts the user to put a weight on the balance, or the UI starts counting down the stabilization time, and then reads the balance measurement.

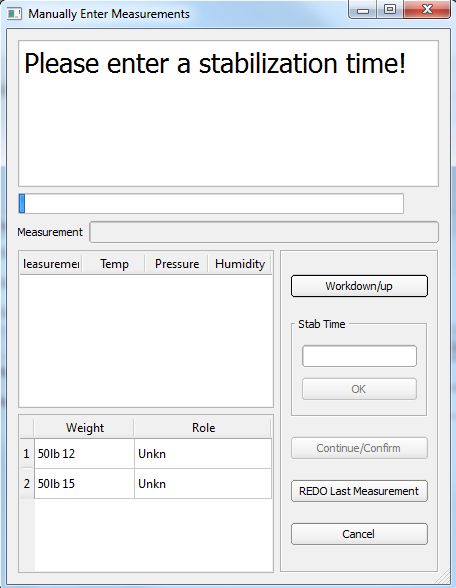
I enumerated it into steps as well so that the program would always know where it was in a measurement. Here are the 8 steps for each ABBA along with the relevant counters.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | row\_counter | isMeasure | set | AB |
| 1 | 0 | FALSE | 1 | 1 |
| 2 | 0 | TRUE | 1 | 1 |
| 3 | 0 | FALSE | 2 | 1 |
| 4 | 0 | TRUE | 2 | 1 |
| 5 | 0 | FALSE | 2 | 2 |
| 6 | 0 | TRUE | 2 | 2 |
| 7 | 0 | FALSE | 1 | 2 |
| 8 | 0 | TRUE | 1 | 2 |

This gets repeated for the number of rows there are in the design matrix. isMeasure alternates between True and False; the False rows are when the UI prompts the user to put a weight on, and the True rows are when the UI counts down and takes a measurement. The ‘set’ column is which set of weights are either on the balance or being measured: set 1 is equivalent to the ‘A’ weight (or weights in a work down) in the ABBA, and set 2 is the ‘B’ weight (or weights). The AB column tells us which half of the ABBA we are in. When AB = 1, we are in the half where we measure set 1 and set 2 for the first time. When AB = 2, we are in the second half where we measure set 2 and set 1 for the second time. This is important for keeping track of when to reset everything for the next row of the design matrix. After step 8, everything is reset to the original state (step = 1, isMeasure = False, set = 1, AB = 1).

When isMeasure is true and the continue button is clicked, the stabilization time counts down. We use a QTimer to count the stabilization time, as time.sleep() would hang the GUI. time.sleep() pauses the process, which also pauses the GUI. Therefore, I used the Qt supported QTimer and QTest.qWait to wait the stabilization time. Then, the ‘read’ command is sent to the balance in an asynchronous thread. In the line pool = ThreadPool(processes=2), I’m not really sure how many processes to allot to the pool; I picked two arbitrarily because I couldn’t really find a good number to use that was backed up by evidence. Maybe you can figure this out and allot the correct number of processes. I didn’t use a normal Thread because I needed to return the thread to return an answer (the measurement) back to the caller. I ended up doing this using Queues. When the measurement is retrieved from the Queue, the program tries to open the balance door if possible. If not, an exception is caught. Something that doesn’t work for some reason: when the exception is caught, I wanted the status browser to display “Got reading: \_\_\_. Press Continue”, but for some reason, it doesn’t show up. The measurement is then saved into data\_dict, and into an external temporary file. The history table in the GUI is also updated with the past 4 measurements. The progress bar is also a little clunky in that it changes when one row of the design matrix has been completed. This makes it less accurate than I would like it to be. For example, in a 41 design, there are 6 rows (6 ABBAs) in the design matrix. Therefore, the progress bar only changes by 1/6 every 4 measurements. Eventually, it would be nice if it incremented every time a measurement has been taken, but it isn’t a huge concern for us right now.

**Manually Enter Measurements**

****

The third and final calibration UI, Manually Enter Measurements, is the most general. This GUI can be used with or without a connection with the balance. As you can see from the picture, it is very similar to the manual balance UI in design and in logic, with a minor change. Instead of querying the balance for the measurement, the UI prompts the user to manually enter the measurement and pairs the measurement with the current environmental conditions. The great thing about this GUI is that it covers everything. It has many pretty much the same features as the Manual Balance UI. The line edit for the measurement receives the focus when the stabilization finishes counting down.

One thing to note is that the enter key is bound to a default button, which happens to be the workdown/workup button. If you want, you can manually override that and bind the enter key to the continue/confirm button. I also started implementing the Redo Last Measurement button, but I was not able to finish it. I hoped that this button would allow the user to erase the last measurement taken (if it was faulty or a wrong weight was put on) and take the measurement again. The difficult thing about implementing this button is that we would need to reverse all the changes made in step, row\_counter, set, and AB. I recommend printing all of these variables out while you go through a full calibration so you can see how they change. If there is a definite way to reverse the last measurement and reverse all the increments made to those variables, implementing the redo button should be pretty straightforward. If you do get to this, it would be best to implement it in the manual balance UI as well.

**The Masscode tab**

In the masscode tab, there are four buttons and one list widget. The Browse button brings up a file dialog (pre-designed by Qt) that allows the user to pick ‘.ntxt’ or ‘.txt’ files to run through the mass code. This browse button is referred to as inputButton in main\_ui.py (~line 160), and is connected to click\_input. Selected files appear in the list widget. When the run NEW masscode button is clicked, each of the input files are run through the Python mass code in another thread. This button looks for the JSON file of the same name so that it can combine information. If a JSON file is not found, it will create an abridged JSON file that will have partial information. It will not have “meta” information like the balance ID, nominal weights, and much more. However, it will have the results of the calibration: it will have the corrections and uncertainties.

This tab is extremely important because an ‘.ntxt’ file will always be created after the end of a calibration. Sometimes, the thermometer goes haywire and returns a bad temperature, which needs to be edited out before analysis can continue. Users can manually edit the ‘.ntxt’ file and then run it back through the mass code.

The Run OLD masscode button doesn’t seem to work; it calls the Fortran code using subprocess, but for some reason it doesn’t work. If subprocess is called outside the GUI it works fine. When the button is connected to it, it doesn’t work anymore. If you check run\_old\_masscode.py in the utility folder, you can check for yourself.

**The Beautify JSON tab**

This tab looks extremely similar to the masscode tab, and it acts very similarly too. The Select Files button brings up a file dialog that allows the user to choose JSON files that they want to turn into final output files. Basically, the Beautify button parses the JSON and writes the information to a text file in the same format as a Fortran output file. Fortran output files end with ‘.nout’, while JSON output files end with ‘.output’.

**The Plot tab**

This tab will be extremely important in the future as we continue to add more information to the database. It has limited capabilities right now, but hopefully in the future this tab can be made more robust to handle a lot.

Currently, the Browse button opens a file dialog just like the previous two tabs and it allows you to select any ‘.nout’ or ‘.output’ files. When the extract data button is clicked, the environmental data is parsed from the all of the output files and graphed. The average temperature, pressure, and humidity of each output file is plotted as a single point. So, for example, if you selected 10 output files and clicked extract data, there would be 10 data points on the temperature, pressure, and humidity graph. This is pretty useful right now to see if there are any egregious drops or rises in any environmental metric which would possibly affect the measurements.

The Send button parses the output file for the weight data and pushes it to the database. The file that deals with this button is parse\_send.py, located in the utility folder. Different parsing is used for ‘.nout’ and ‘.output’ files. If you look at the function it calls in database\_orm, you will see that parse\_send pushes the data from the output files into two different tables: internal weight data and external weight data. NIST weight data gets pushed into internal weight data, and customer weight data gets pushed into external weight data. This does NOT change the weights\_internal table; if you want to change the accepted value on an internal weight, it must be done separately in that table.

**The DB Access Tab**

This tab provides an interface to the MySql database. The radio buttons near the bottom allow the user to select which database table they want to view. This code is in populate\_ui/populate\_db\_acces.py. At the top, I define all the column headers for the different tables. For example, the stations table has 8 columns, ID through Room. When a radio button is clicked, the name of the radio button is passed into populate\_db\_access, which sets up the table accordingly. First we set the number of columns, and then we set the labels. Then, the database is queried for all of the data in the particular table, and the database table is populated row by row, column by column.

The buttons on the bottom all bring up separate sub UIs. They are very straightforward; they take in information from the user and spits it back into the MySql database. I’ll go through the Edit Existing Weight button and GUI just as an example.

When the Edit Existing Weight button is clicked, an instance of EditExistingWeightUI is created. It takes in an instance of MainUI so that it has access to the database. It is subclassed from QObject, then the window is set to a Dialog box, and the UI is loaded (again, using uic.loadUi, the magic phrase) into self.ui. The weight tree on the left is populated, all of the fields on the right are disabled, the event handlers are connected, and then window is executed. When an item in the weight tree is double clicked, the slot self.item\_clicked is called, which queries the database for the weight info of the selected weight, and then populates the fields to the right with the existing information. The name cannot be edited because the name is used to identify the weight in the database. When the submit button is clicked, all of the fields are parsed, and the information is pushed back into the database using the update() command in SqlAlchemy.

**The Miscellaneous Tab**

This tab has a random UI I created for Donna; I’m not really sure how often it will be used in the future, but you could make some interesting things and just throw it in the Misc tab. Currently, the only thing in this tab is a button that brings up a GUI that calculates air density. Have fun with this tab! If you want to experiment with anything, just do it in this tab so it won’t mess up anything essential.

I know it’ll take a while to wrap your head around this whole program; it took me over 3 weeks to fully understand what Alexander had written, and that was like a third of what we have now. I think the diagram below will help you understand the big picture and what we hope this program will be in the future.

Thermometer Hygrometer Barometer

Commands

Environmental data

Enviro

New mass code

New Mass Code (GUI)

FINAL OUTPUT

FINAL OUTPUT

Fortran Mass Code

.ntxt

.json

MEASURE

1. Automated Comparator
2. Manual Comparator
3. Manually Entered Measurements

Weight info, environmental info, design info, etc.

Measurements

Commands

GUI

Balance

Manual Entry

Environmental data

Environmental Instruments

Database