Automated Wireless Research Tester
User Manual

User Guide

The first portion of this document is for users who are interested in using the software. In the following sections, we will go through how to set up your hardware and software before using the software, and then we will go through a tutorial for the software itself.

Hardware Setup

[this section goes over setting up the hardware, Thomas]

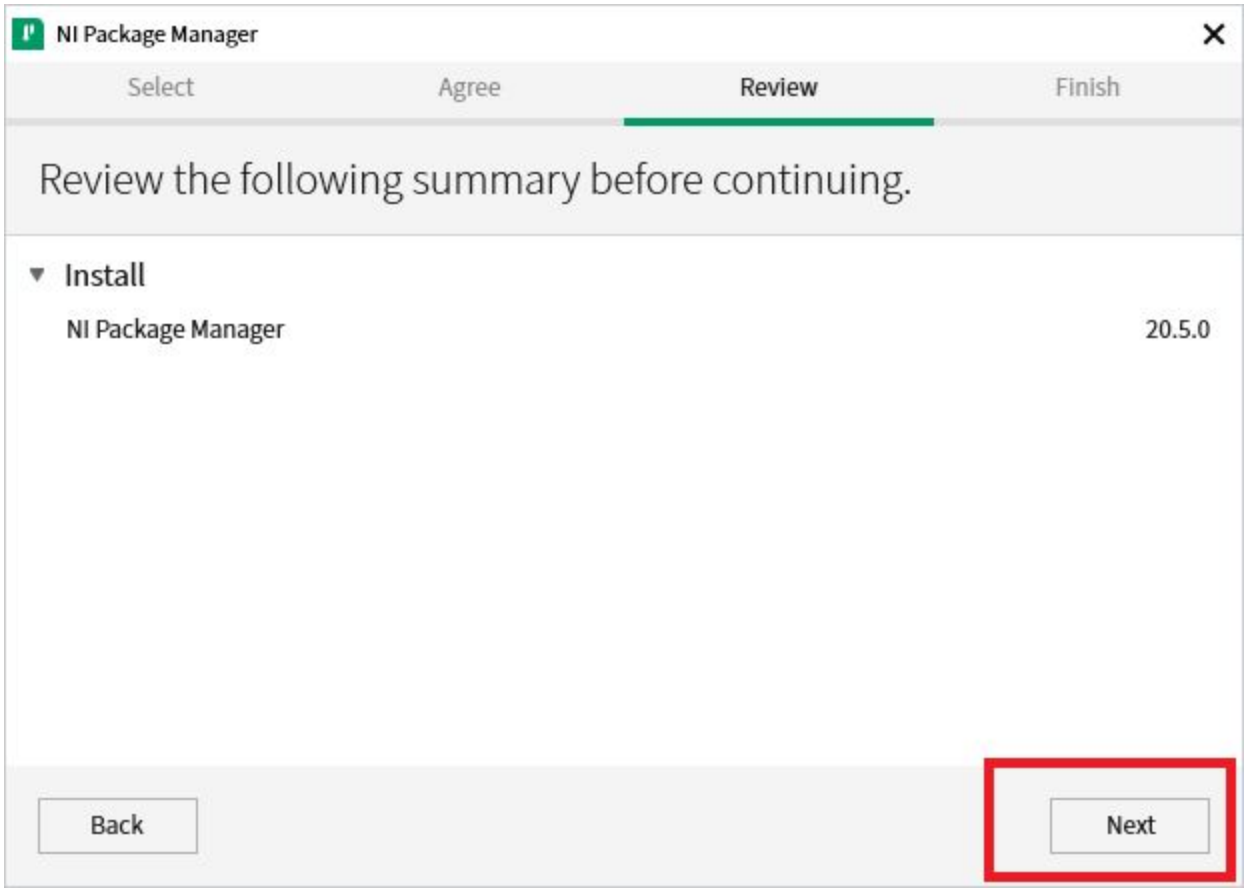
Software Installation

Installing Drivers

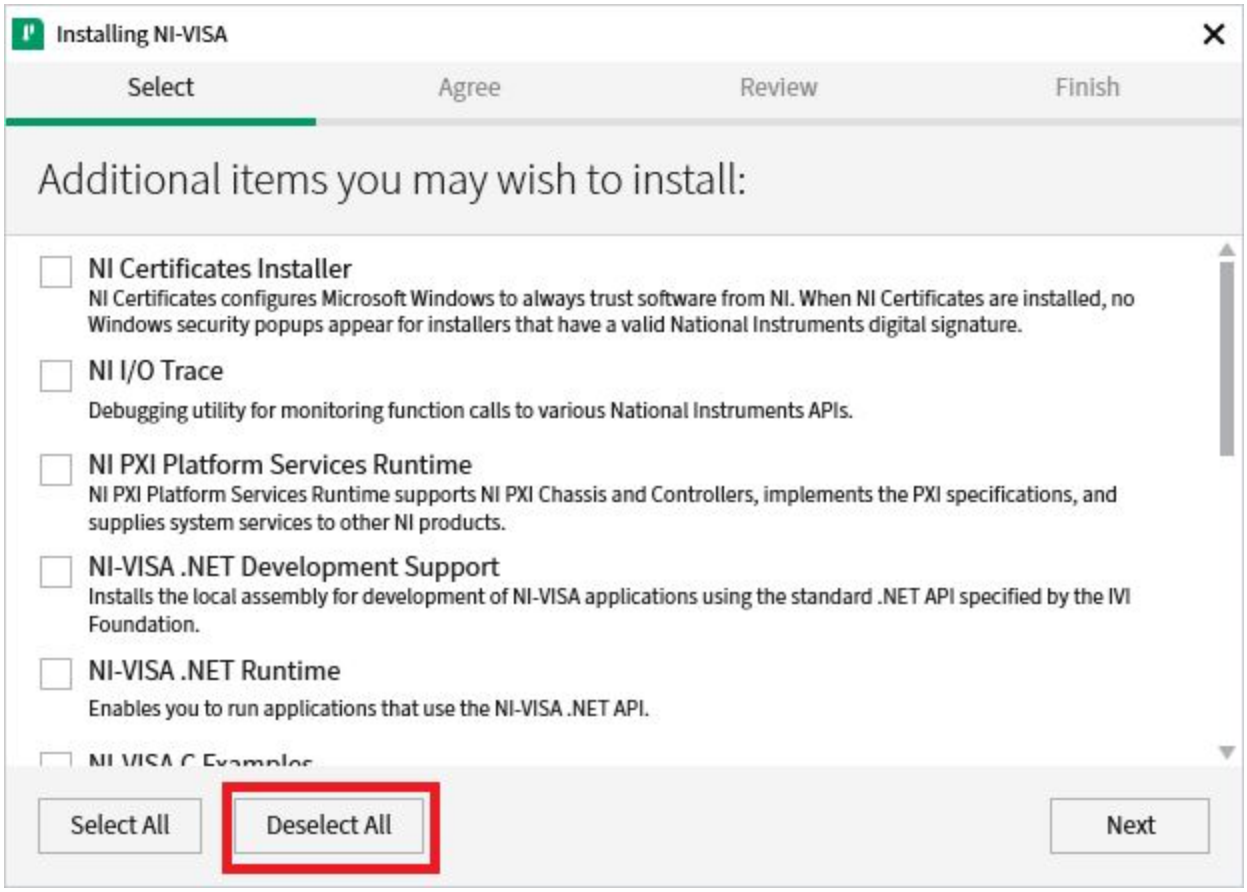
This software uses drivers from National Instruments in order to control both the VNA and the positioner. Please go to the following links and download the drivers:

- NI-VISA Download (Version 19.5 or newer): https://www.ni.com/en-us/support/downloads/drivers/download.ni-visa.html
- NI-488.2 Download (Version 19.5 or newer): https://www.ni.com/en-us/support/downloads/drivers/download.ni-488-2.html

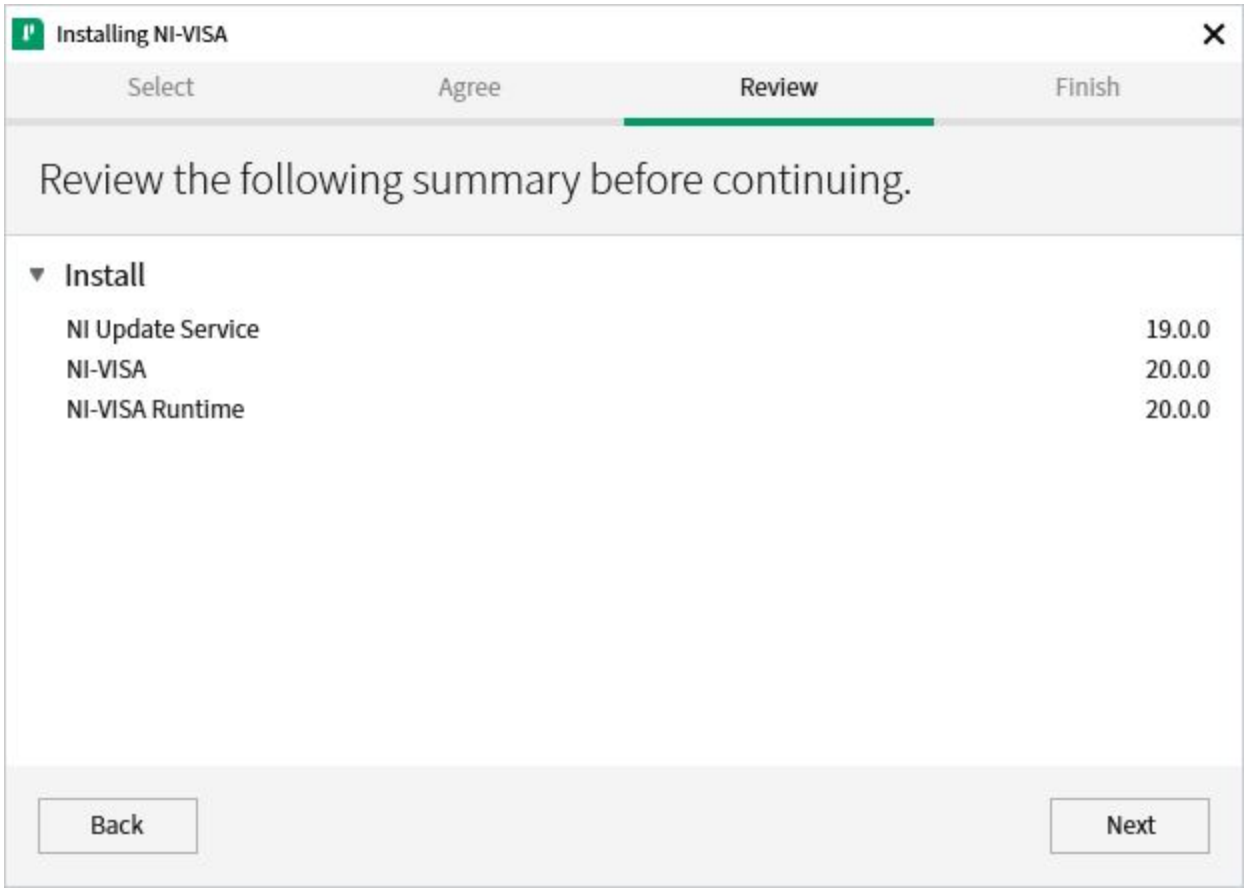
The installer will prompt you to install the Package Manager first, click Next to continue:



Next, the installer will give you the option to install additional items. These items are not required for our software. You may click “Deselect All” and continue:



Next, the installer will list all the items that will be installed. It should look something like this:

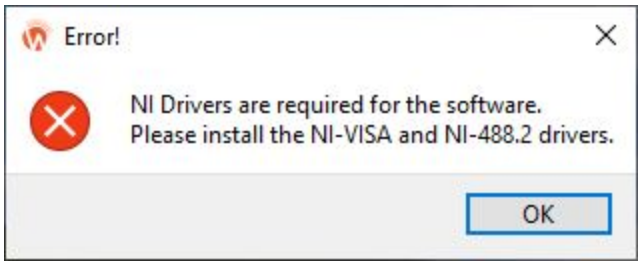


Continue on and finish installing. The process will be the same as above for the NI-488.2 Driver.

Starting the Software

To start the software, locate `AutomatedWirelessResearchTester.exe` in the software folder. Simply double click on the file to run it. Feel free to add a shortcut to your desktop for quicker access.

Note: If you encounter the error message below when attempting to start the software, that means that you have not installed all the required drivers.



Using the Software

[this section goes over how to use the software, Maria]

Software Programming Guide

This remaining contents of the document will cover everything you need to know to modify the source code of the software. This will include the list of tools you will need for programming, a high-level overview of the main modules of the code, and finally how to repackage your source code into software.

Required Tools

The following table contains all of the resources (all free) you would need to modify and test your source code:

Note: *It is recommended that you create a virtual environment like anaconda to install these Python packages so that the different versions do not interfere with your local python environment.*

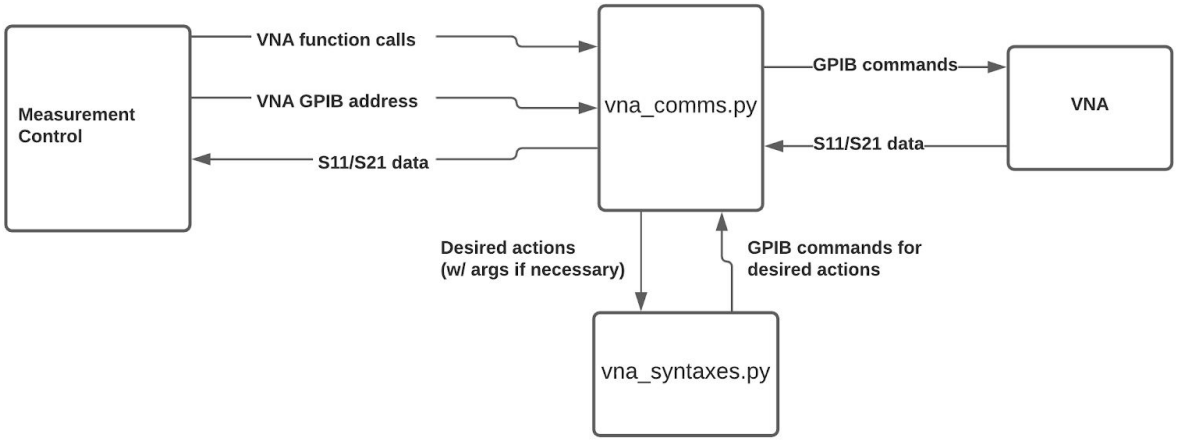
Name	**How it's used**	**How to install**
Python (3.8 recommended) (*Install this first!*)	It is the main programming language in the source code.	Visit Python to download
PyQt5 (5.15.1)	Python package used for programming our graphical user interface.	Type `pip install PyQt5==5.15.1` in the command prompt
matplotlib (3.2.2)	Python package used for generating plots.	Type `pip install matplotlib==3.2.2` in the command prompt
pandas (1.1.4) and numpy (1.19.3)	Python packages used for reading and processing data files.	Type `pip install pandas==1.1.4 numpy==1.19.3` in the command prompt
PyVISA (1.11.1)	Python package used for communicating with the positioner and VNA.	Type `pip install pyvisa==1.11.1` in the command prompt
Qt Designer	Tool to design the layout of the graphical user interface.	Visit Qt Designer to download
pyinstaller (4.0)	Tool to package python source code into an executable	Type `pip install pyinstaller==4.0` in the command prompt
National Instruments Drivers	Required drivers to control the positioner and VNA	See the Installing Drivers section

Measurement Control

Measurement Control - Positioner

Measurement Control - Vector Network Analyzer (VNA)

The VNA portion of measurement control is responsible for communicating data to and from the VNA. The figure below shows how measurement control interacts with the VNA modules:



In `vna_comms.py`, all of the VNA functions are located inside the `Session` class. These functions include:

- `reset_all`: Performs a factory reset on the VNA
- `reset`: Resets only measurement parameters on the VNA
- `setup`: Configures the VNA with the appropriate sweep/frequency list, averaging factor, and IF bandwidth.
- `get_data`: Will take 1 data point (either S21 or S11) at the frequencies indicated in `setup()`. The function returns a list of data points, with each data point being a Data class object, which contains information about measurement type, frequency, position, and value of data.
- `calibrate`: There are three separate calibration functions: One for each of standards that need to be calibrated (open, short, and load).
- `rst_avg`: Resets the averaging on the VNA. This is usually used whenever the positioner moves to the new position, and we don't want the averaging to contain data from previous positions.
- The `Session` class also maintains the communication session with the VNA (think of it like being on a phone call). Hence the name session.

Basically, measurement control creates a `Session` class object. From there, measurement control will call specific VNA functions depending on where we are in the measurement. The

VNA function calls will then turn into GPIB commands, which will then be passed on to the VNA itself.

This is where `vna_syntaxes.py` comes in. `vna_syntaxes.py` contains a number of functions, with each function representing an action to be performed on the VNA. The functions take accept arguments including the VNA model and the command arguments (e.g. frequency), and it returns a string containing the properly formatted GPIB command for this specific action(s) on the specified VNA model.

Adding a New VNA

The structure in `vna_syntaxes.py` is set up in such a way that allows you to add additional VNAs that would be able to be used with the software. We will go through a short example to show you all the pieces you will need to add to `vna_syntaxes.py`:

1. Adding the model information: The first thing you will need to do is add your new VNA to the model class. See the screenshot below for an example:

```
17  class Model(Enum):
18      """Add additional VNAs here"""
19      HP_8753D = auto()
20      # ex: NEW_VNA = auto()
21
```

2. Identify the model: Now that you've added the model, there needs to be a way for the code to know which VNA model it is communicating with. An ***IDN query** (identification query command) is sent to the VNA, and the VNA a string with the model information. **To obtain the returned string, try running the script in** `vna_identify.py`. It will ask you the GPIB address of your VNA, and it will print the returned identification string.
Once you have obtained the string, add an else if statement to the block below in check_model:

```
23  def check_model(string):
24      """Indicate a unique portion of the returned *IDN? query string
25      which can identify the specific VNA model"""
26      if '8753D' in string:
27          return Model.HP_8753D
28      else:
29          raise Exception('Model is either not supported, or model is not
```

3. Adding all the commands: Now comes the tedious part. You will likely need to reference the programming guide for VNA so that you have information on all the different GPIB commands that you can send to your VNA. There comments for every function in syn that describe what actions the functions perform. You will need to find the command for your VNA that produces that same action, and add it to the function. For example:

```
206  def polar(model):
207      """This action should select the polar display format"""
208      commands = {
209          Model.HP_8753D: 'POLA'
210      }
211      return commands.get(model)
```

In this polar function, the GPIB command sent will change the display format on the VNA to polar form. You will need to add the command for your VNA in the commands block (e.g. `Model.Your_Model: 'YOUR COMMAND'`)
Repeat this for all of the remaining functions. After You should be set and you will be able to use your new VNA with the software.

For full, step-by-step details on how each function runs, please go to `vna_comms.py` and `vna_syntaxes.py`. There you will find detailed comments for the code.

Graphical User Interface (GUI)

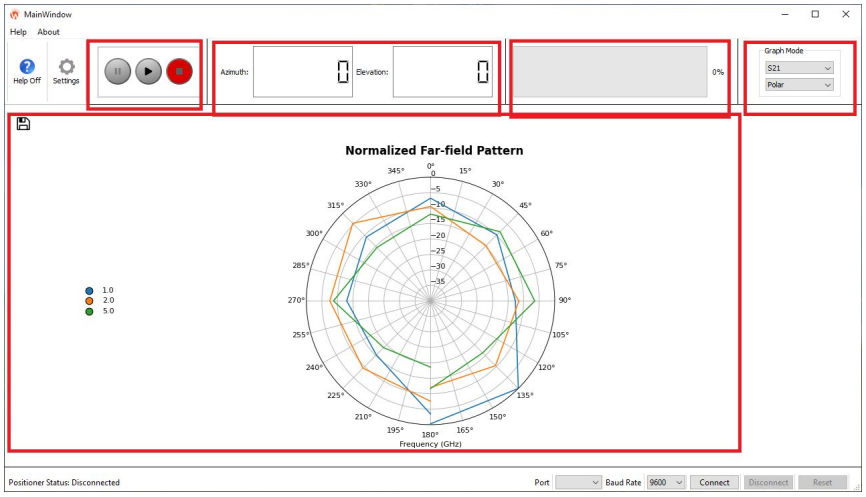
With the exception of the top-level code `mainWindow_app.py`, all code files for the GUI are located in the `gui` folder. In this section, we will go over what each file does, and how to modify these files if you wish to do so.

In the gui folder, you will find files with the following names:

- `[something]_ui.ui`
- `[something]_form.py`
- `[something]_app.py`
- `.png` files

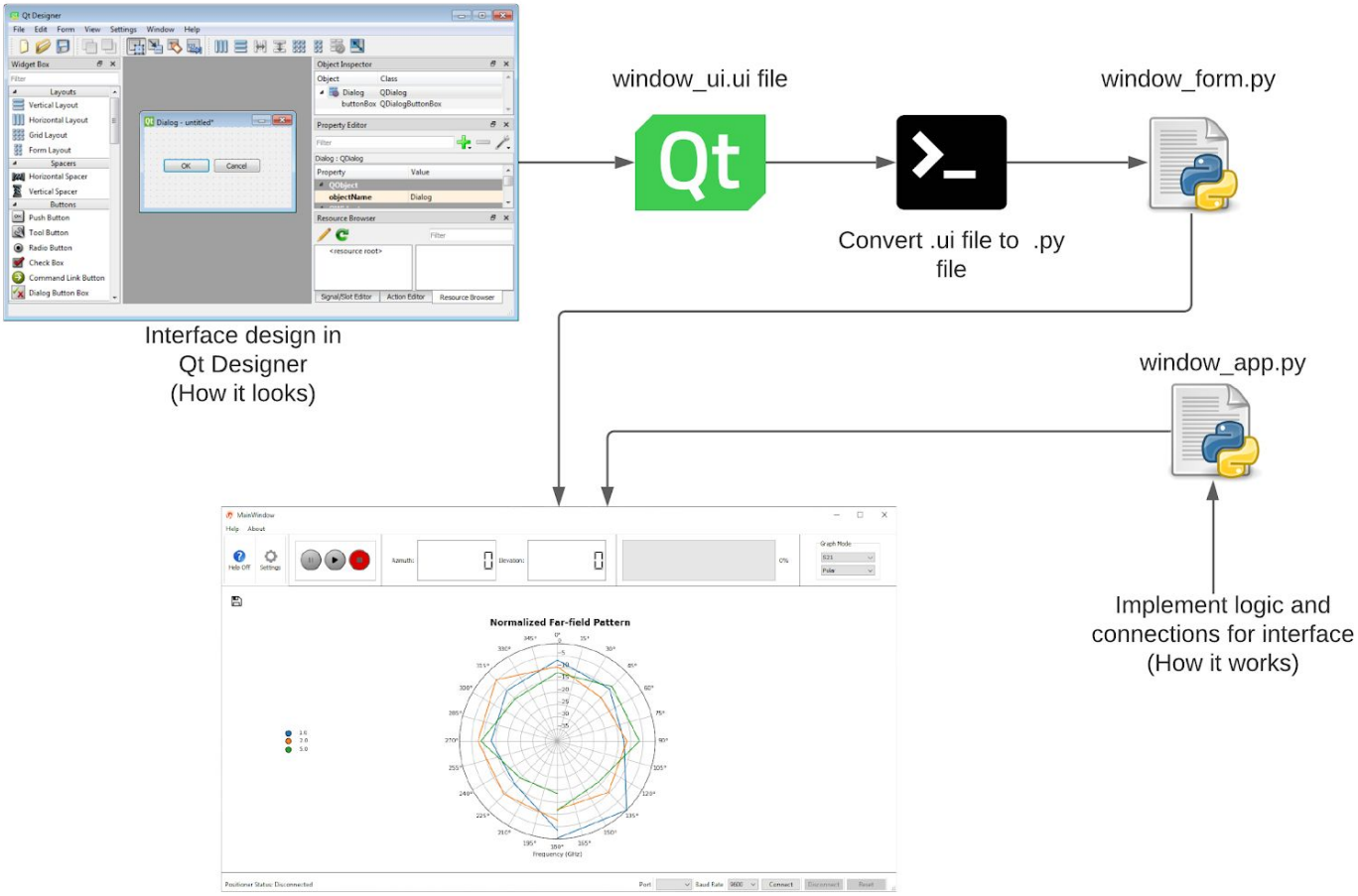
The entire user interface for the software is actually a combination of many smaller subinterfaces. There is an individual interface for the progress bar, the settings window, the graph mode selection toolbar, and so on. In the screenshot below, each red region represents a

separate user interface design:



As you can see, this is the reason there are so many files for the GUI.

So what is the process when it comes to working with these files? The figure below shows the general workflow of the user interface design:



First, we use Qt Designer to design how we want our interface to look. This includes what buttons, text boxes etc. we want to have, as well as where and how big they should be. Once the design is complete in Qt Designer, an `.ui` file is created.

However, due to packaging requirements, we cannot directly implement the .ui file into our program and expect it to work, so this is where the next step comes in. The `[something]_ui.ui` files must be converted into `[something]_form.py` files. **To do this, run the `ui2py.bat` batch file in the** gui **folder, and all `.ui` files will automatically be converted to `.py` files** (you should see the `.py` files being updated, while the `.ui` files remain unchanged). The `.ui` files and the `.py` files contain identical designs for the user interface.

There are also images and icons that are used in the user interface. Again, due to packaging requirements, we cannot directly use the .png files in our program. They have to be converted into binary data and stored in a `.py` file. So how is this done?

- The `resources.qrc` file should include the list of images that we want to convert into binary form.

- Next, run `resources.bat` to compile the images into the binary file. You will need to have PyQt5 installed in order to run the batch file.
- The output is the `resources_rc.py` file. Do not change the name or contents of the file, as PyQt5 will be looking for this specific file when compiling the program.

Now that we've taken care of how the GUI should look, it's time to work on how the GUI will run. This is what all the `[something]_app.py` files are for. They include what happens when certain buttons are pressed (signal triggers), threading the program for the smooth experience, and handling errors within the program.

The file with the most logic is `mainWindow_app.py`. We have tried to make the code as organized and well-commented as possible, but it still may seem confusing if you are not familiar with PyQt and all of its different objects. So here is a quick rundown of some **must-know** knowledge when it comes to PyQt:

- Depending on the type of QObject, they will **signals** and **slots**:
    - A signal emits information, usually when an event occurs (e.g. button presses, the window being resized, when an `emit()` function is called for that specific signal)
    - A signal **connects** to a slot. Think of the slot as receiving the signal. You will see a lot of these `connect` functions in the code. That is where the signals and slots are being established.
    - A slot is usually a function of some sorts, and it is called when the signal connected to it is emitted.
- The settings window and the main window use the QMainWindow class, while all other elements (progress bar, graphing window, etc.) use the QWidget class.
- The `[something]_ui.ui` and `[something]_form.py` files contain the names and properties of the QObject items in the user interface. If you want to implement logic for, say, a specific button, you will need to know the name of that button and what Qt Object type it is. Different QObjects have different properties, signals, and slots.

For more information visit Qt for Python for detailed documentation on every Qt Object that we've used in our source

Data Processing

Repackaging the Source Code

Packaging the source code was a complicated process for us, so we have tried to make the repackaging process as easy as possible for you! Running the `build.bat` script by either double-clicking on the file or running it in the command prompt will automatically start the packaging process.

Feel free to read through the `build.bat` script to see what is happening. Here is a recap of all the actions it performs:

- Installs all required python packages with the correct version
- Runs `pyinstaller` and starts the packaging process. It looks into `build.spec` for instructions for the packaging process
- Moves the new executable to the top-level source code directory
- Removes temporary files created during packaging

After a minute a two, the packaging script should be finished. You can copy the `.exe` file and use it anywhere you like (as long as OS and driver requirements are met, of course).

Note: Make sure your modified source code is working properly and error-free in a python environment before packaging it.