Gamry Electrochemistry ToolkitPy™

Echem ToolkitPy is a new Python API released for Gamry potentiostats. **ToolkitPy** provides a powerful way to control your Gamry Instruments' potentiostats and other devices.

Why use ToolkitPy?

Written for Python users, ToolkitPy gives direct access to all real-time controls available originally in The Gamry Framework Software. Everything has been streamlined so that you can run signals and acquire data natively within Python. We now offer more variation in data handling, changes to signal types, easy interfacing with ancillary devices, and increased data point limits.

Software developers will be able to build up stand alone programs that are just as powerful as our own bundled software or write custom scripts that call our potentiostat to run in an integrated environment. Automation is easier with ToolkitPy.

Laboratory instructors wanting to integrate programming skills in the curriculum can have student use ToolkitPy within integrated development environments (IDE) such as PyCharm. Students can also run a Gamry potentiostat and analyze the resulting data all from one Jupyter Notebook.

We (Gamry) also use ToolkitPy to build out and create custom scripts for our potentiostat users needing state-of-the-art signals for pushing the boundaries of electrochemical science. We can save you time and create customizable code.

Python-friendly experimental control

Historically, our potentiostats are controlled by the Gamry Framework Software. The code that controls experiments (or scripts) in Gamry Framework is publicly available and can be altered by any *user.* These scripts are written in procedural language based on function calls and sequential execution of statements called EXPLAIN™. Users can create a custom experiment in EXPLAIN that can be controlled in Framework. This has always been available to users as a simple yet powerful customization option. However, it requires investing time in EXPLAIN, which may not yield a good return when compared to Python. ToolkitPy re-invests that time into learning API calls that can integrate with existing Python code. This allows you to more easily create scripts that both control the instrument and analyze data.

Data acquisition

ToolkitPy gathers data into a NumPy array, which is a very efficient data structure. The data array can be exported to any file type for subsequent data processing. Or you can also perform data manipulation techniques post-experiment using one of the many libraries for scientific computing in Python (NumPy, SciPy, or Pandas).

The array can be shared with other external equipment operating in the Python environment. Many third-party instruments and controllers have existing Python libraries. Scripting your experiments in ToolkitPy should allow for seamless integration with these devices, such as pumps, machines, cameras, etc. Depending on compatibility, synchronized operation and automation is possible. More interestingly, you could control your Gamry potentiostat from within a third-party instrument’s software so long as Python version compatibility is maintained.

ToolkitPy data acquisition is automated. Therefore, once run is called, a separate thread is created that automatically pumps the data to the NumPy array. This is a distinct difference from our more general platform, The Electrochemistry Toolkit, where we would have to call a Cook() to take the points out of the instrument's memory at a set interval.

Point limit

An individual experiment in Framework can only acquire 262144 data points. This is usually not an issue for most types of experiments; however, certain applications, especially neuro-stimulation, require a large number of data points. More broadly, the data point limit is approached when you need to record fast transients over a long period of time. With ToolkitPy, you can extract as many points as your RAM or hard disk can handle*.

**Experiments run in fast mode (sampling rates faster than 100 µs) can only record up to around 2 million points.*

How ToolkitPy works

A pre-step is to import common libraries and any other dependencies.

```
import time as Time
import numpy as np
```

Importing ToolkitPy

The first step to utilize ToolkitPy in any .py file is to import toolkitpy with the following:

```
import toolkitpy as tkp
```

Like all libraries, toolkitpy needs to be imported to utilize all of its functions. You do not need to import it as "tkp"; however, that is commonly how all of our source code imports the library. We recommend using it.

Initializing ToolkitPy

The second step required is to initialize toolkitpy in order to use its built-in functions. Use the following command:

```
tkp.toolkitpy_init("cv.py")
```

It is important to assign a unique string to avoid conflicts if you are running several .py files utilizing the toolkitpy libraries at the same time. The name of the string can be anything as long as it's unique.

Creating a pstat object

Step three requires you to create a pstat object in order to send commands to the Gamry potentiostat. There are two methods to creating a pstat object, described below.

A. *Grab the first Pstat:*

```
pstat = tkp.Pstat("Pstat"])
```

This method will allow you to grab the first pstat connected to your computer. This option is not ideal when you have more than one pstat connected (multi-channel potentiostat setup) and you want to select a particular channel.

B. *Create a list of pstats and grab from the list:*

You can create a list of pstats by using the enum_sections() command:

```
pstat_list = tkp.enum_sections()
print(pstat_list[0])
```

This will then return a List object containing the names of the available pstats that you have connected to your computer. It's helpful here to include a print() function, but not required. You can then choose the pstat from that list by using the unique identifier of the pstat returned in the list:

```
pstat = tkp.Pstat(pstat_list[0])
```

Knowing which potentiostat you are using is important for current limits, voltage limits, or frequency limits.

Up to this point, all we have done is acquire the potentiostat we want to use through ToolkitPy. The next section describes how you tell the potentiostat what to do. Here, we use cyclic voltammetry as an example given its universal appeal in many electrochemical applications. But this is also where you can be creative and build out unique and interesting commands to send to the potentiostat.

Creating a signal

For Cyclic Voltammetry, we use the function **pstat.signal_r_up_dn_new()**. Decoded, it's shorthand for a **signal** that applies a linear **ramp up** to a scan limit and back **down** to a second scan limit, cycling in-between. It takes all the usual arguments needed to specify a CV experiment in Framework. Shown in **Figure 1**, these also include the initial E, final E, scan rate, step size, and number of cycles.

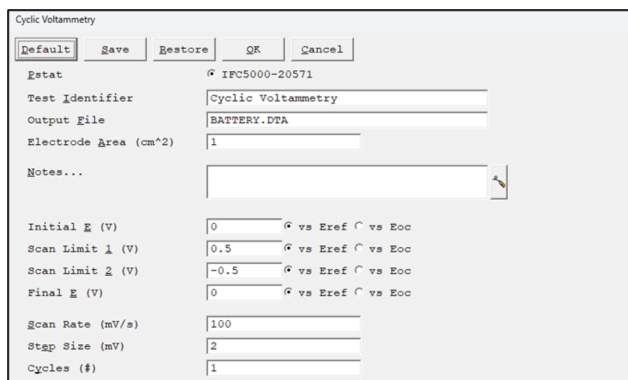

Figure 1. Parameters available in the setup window of a cyclic voltammetry script in the Gamry Framework software.

The complete signal is defined as the following:

```
signal = pstat.signal_r_up_dn([0,0.5,-0.5,0],[.1,.1,.1],[0,0,0],0.002,1,tkp.PSTATMODE)
```

The first argument is a list decoded as [Initial E, Scan Limit 1, Scan Limit 2, Final E]. The second argument is a list of scan rates—one for each section of the triangular wave. The third argument specifies a vertex hold, which allows for an accumulation and strip. These are followed by arguments for the step size, number of cycles, and a hardware setting call to run the ramp under potential control. This should give you a hint that the same signal could be called under current control, or GSTATMODE.

Note: You can find a variety of signals that have been created as templates for ToolkitPy.

Once the signal is defined, you must pass it to the potentiostat and initialize with these two steps:

```
pstat.set_signal_r_up_dn(signal)
pstat.init_signal()
```

Note the addition of "set" (blue arrow), which differentiates it from above.

Up to this point, the pstat knows what to do—it has a signal to apply. We only need to wrap this inside of what we call a **curve**.

Curve Initialization

Curves are data acquisition control objects and sometimes are referred to as DTAQs. Curves dictate stop at conditions and the data output. In this case, we are looking for the RCV curve. The call for the curve creation would be the following:

```
curve = tkp.RcvCurve(pstat,100000)
```

The second argument in curve specifies the maximum size of the NumPy array. It must be greater than the estimated number of points needed to record all data resulting from the applied signal.

Running the Curve

The final step is to turn ON the cell switch (allows current to flow through the system) and run the curve. A while loop can be used to acquire the data in the NumPy array and run common processes like writing to a file. Or you can simply print() the data altogether. The set of commands is below:

```
curve.run(True)
while curve.running():
        data = curve.acq_data()
```

Utilizing both the curve and signal to create the signal

Putting all the lines of code together would yield:

```
import time as Time
import numpy as np
import toolkitpy as tkp
tkp.toolkitpy_init("cv.py")
pstat_list = tkp.enum_sections()
print(pstat_list[0])
pstat = tkp.Pstat(pstat_list[0])
signal = pstat.signal_r_up_dn([0,0.5,-0.5,0],[.1,.1,.1],[0,0,0],0.002,1,tkp.PSTATMODE)
pstat.set_signal_r_up_dn(signal)
pstat.init_signal()
curve = tkp.RcvCurve(pstat,100000)
pstat.set_cell(True)
curve.run(True)

while curve.running():
        data = curve.acq_data()
        np.savetxt("Test_tkp.csv", data,
        delimiter = ',', newline = '\n', header
        = ','.join(data.dtype.names),comments =
        '')
del signal
del curve
del pstat
```

A few additional lines of code are added to save the array to a file and delete the signal, curve, and pstat objects. Altogether, this block of code represents the core functionality of ToolkitPy. However, it is not complete. Several elements such as potentiostat hardware settings, real-time plotting, and error handling are omitted for clarity.

Virtual Front Panel 2: A full implementation of ToolkitPy

We recommend that users interested in ToolkitPy take the time to experiment with our Virtual Front Panel 2 (VFP2) software. VFP2 can be installed alongside Framework and gives users real-time control over their Gamry Potentiostat(s). This application is intended for use as a replacement for a hardware front panel. The source code for this application is available to users as a demonstration of ToolkitPy and is installed alongside ToolkitPy.

System Information

ToolkitPy requires a Gamry potentiostat. Instruments supported include Interface 1000/1010, Interface 5000, Reference 600/600+/620, Reference 3000/3000AE. Auxiliary equipment includes the IMX8/ECM8 electrochemical multiplexer and RxE 10k Rotator. Microsoft Windows 10, 11, or higher is required as well as a suitable programming environment for coding such as Visual Studio Code.

Gamry's Software Suite Installer provides all necessary files for the installation of the ToolkitPy software development tool. It includes a Python Installer for a curated version of Python 3.7.9 (32-bit), ToolkitPy, and various site-package libraries such as NumPy 1.21.6 or Pyside2 5.15.2. Python Package Index (PyPI) should not be required.

ELECTROCHEMISTRY TOOLKITPy Rev. 1.0 11/4/2024

© Copyright 1990-2024 ©Gamry Instruments, Inc.

734 Louis Drive Warminster, PA 18974 USA +215-682-9330 Fax: +215-682-9331 info@gamry.com www.gamry.com