Subject: Resources for gifted student programmers who are new to Igor
Date: Wed, 2 Mar 2022 17:52:49 -0500
From: Howard Rodstein <support@wavemetrics.com>
To: Igor Discussion List <info-igor@lists.info-igor.org>

First the student needs to follow at least the first half of the Igor guided tour (Help->Getting Started). This is absolutely essential.

Once the student programmer has some hands on experience using Igor, the following help files need to be read:

```
Experiments, Files and Folders
Commands
Procedure Windows
Programming
Interacting with the User
Programming Techniques
Debugging
```

Next I recommend a process of experimentation and consulting the help.

Once the student programmer starts to grasp Igor programming concepts, I recommend rereading the Programming help file. It takes some iteration of experimentation and rereading before things start to really make sense. (You'll have to trust me that they do make sense at least to some degree :)

Here are some other good resources:

1. Search the online help for examples (Help->Search Igor Files) or (Edit->Find In Files)

2. Search WaveMetrics procedures for examples (Help->Search Igor Files)

3. The snippets at https://www.wavemetrics.com/code-snippets

It's useful for a student programmer who uses an Igor procedure package, whether from WaveMetrics or from a third party, to read the underlying procedure code or part of it. Single-stepping through parts of the code is helpful for clarification of vaguely understood programming features.

While programming, turn Igor's debugging features on (Procedure->Debug on Error and the like).

Once the student programmer begins to grasp of Igor programming, working on an actual project is invaluable. Start small, get something useful to work and then elaborate on it.

In Igor programming as well as any other programming, good programming practices are very important. Here are some that I try to implement:

- Write code as clear and readable as possible using expressive names. Someone (maybe the original programmer) will need to understand it in the future. The code should be self-documenting, via expressive naming practices, to the extent feasible. Add comments to give an overview or explain the programmer's motivation.

- Write modular code with subroutines that depend only on their inputs (not on globals) as much as possible. This makes subroutines more readable, testable, and reuseable.

- Use of globals is appropriate when user inputs or preferences need to be remembered. Access to globals should generally be in high-level code, not in low-level subroutines.

- Massive functions that go on forever are hard to read and understand. Break them up into units that can be meaningfully named. Then use a higher-level function to call the units. If expressive names are used, the higher-level function will be pretty-much self documenting, easy to read and easy to understand. Implementation details will be relegated to the lower-level functions which readers can skip unless they want to know how parts are implemented.

- Organize the code into logical file units with file names that state the file's function.

- Test the code thoroughly. It is not possible to test all possible combinations of inputs but you can strive to exercise each, or nearly each, line of code. If possible write automated tests that you can run to make sure you have not broken something when you make a change.

Howard Rodstein
WaveMetrics
