Agile Computer Control of a Complex Experiment

Gaël Varoquaux

▶ **To cite this version:**

Gaël Varoquaux. Agile Computer Control of a Complex Experiment. Computing in Science and Engineering, 2008, 10 (2), pp.55-59. 10.1109/MCSE.2008.47 . hal-00502504

HAL Id: hal-00502504

https://hal.science/hal-00502504

Submitted on 15 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Agile Computer Control of a Complex Experiment

This article introduces techniques and tools useful for writing an experiment's control framework. In particular, the author discusses how to use the Python language to control hardware.

Today's experiments can involve dozens of devices that must work together. An experimentalist is often taught the arts of electronics, optics, and mechanics required to build and run an experiment, but software engineering is frequently left out,[1] so when the task grows in complexity, it's the software that becomes a weak point.

Scientists are skilled with computers, and many of them understand the intricacy of numerical computing. Yet, designing the sophisticated software architecture that controls an experiment requires different skills, and small- and mid-sized experimental labs often lack a software engineering culture. Bad design choices plague experimental labs even though the real experimental difficulty seldom lies in the software itself.

In this article, I give some guidelines for designing an experiment's control software based on my experience in various Bose-Einstein condensation labs.[2] I explore the tools and patterns that lead to successful projects—in particular, a flexible and reliable code base that lets scientists cope with a research lab's ever-changing goals and resources.

Avoiding Timing Problems

Scientists use computers for both instrument control and data acquisition. The ability to put elaborate logic in a computer program helps realize the dream of replacing the electronic boxes commonly found lying around a lab with computers. Modern computers are clocked at several gigahertz, so many people believe that they can simply replace servo-locks and electronic timers for on-the-fly data processing. The bad news is that computer systems, both hardware and software, are optimized for throughput, not for short response times. Latencies of several milliseconds forbid any control loops with frequencies higher than 100 Hz, apart from those on dedicated systems (such as real-time operating systems or embedded devices).

You can move timing issues off the computer and into well-clocked electronic devices by relying on an external clock to trigger hardware-related actions. If necessary, input and output buffers can freeze the experimental time base while the computer retains some flexibility. This pattern is very efficient for solving most real-time experimental problems—as long as the frequency doesn't exceed a few kHz, and the computer is fast enough to perform the work during a clock cycle. Computer-programmed embedded systems often provide good solutions, and some commercial systems are often easier to implement and more reliable than home-brewed complex software solutions.

Using the Right Tools

If speed is paramount, you'll have to write the control software in C or C++, possibly with the help of a framework such as LabWindows or Root (http://root.cern.ch). However, try to avoid low-level languages as much as possible.[3] Not all scientists are familiar with memory management or linking and compiling, so the use of a low-level language in-

1521-9615/08/$25.00 © 2008 IEEE
Copublished by the IEEE CS and the AIP

Gaël Varoquaux
LENS, University of Florence

 THIS ARTICLE HAS BEEN PEER-REVIEWED.

Authorized licensed use limited to: CR Saclay. Downloaded on June 12,2010 at 09:26:42 UTC from IEEE Xplore. Restrictions apply.

creases development time, makes it harder for newcomers to contribute, and increases the chances of bugs and design errors.

The solution for building experiment-control software must be accessible to beginners and suitable for large projects as well as small ones to allow for rapid development. It must have a rich standard library, so that developers don't lose time implementing visualization or disk operations. We'll see later that the solution must also have good support for multiple threads. Not many languages or frameworks meet these criteria—Labview, for example, isn't scalable because it's based on graphical computing and is suited only for small projects, and although Matlab is math-aware, it's single-threaded.

With the recent rise of powerful numerical modules,[4] Python offers a good alternative. This agile language is focused on ease of use and development speed, yet it retains advanced features. In the rest of this article, I focus on the use of Python, although many of the considerations described here don't necessarily depend on language choice. Regardless of platform, you can achieve a huge gain in productivity and reliability by using existing libraries. Python provides great modules for visualization (Matplotlib[5] or Chaco for 2D plots and TVTK[6] for 3D plots), but every major language has libraries for plotting, so implementing such tools is both a waste of time and a threat to your project's reliability and maintainability.

Controlling the Hardware

All the instruments being controlled by the software must have an internal representation. Instruments are connected to the computer through various technologies and use different sets of low-level instructions to receive and send information, none of which is relevant to the general goal of control software—for example, taking a picture with a camera connected to a firewire port shouldn't appear to be any different than taking a picture through a frame-grabber card. You should program to an interface, not an implementation—that is, you should address your hardware through a universal set of instructions that don't depend on the implementation. A separate layer will translate everything to instrument-related gibberish. This is important because it makes the main code base more readable and allows modularity, both of which help you diagnose bugs and let you easily replace the hardware. Object-oriented programming is well adapted to such modularity and abstraction. The details of the implementation are hidden in the objects' internals, and only the meaningful methods and attributes are exposed in the calling code.

The task of writing methods to talk to the hardware itself often implies getting your hands dirty. Python has modules to control instruments controlled by the VISA (Virtual Instrument Software Architecture) standard or some vendor-specific libraries, but you'll most likely have to build your own interface to the hardware. If the instrument is connected to a standard bus, you'll have to use the appropriate Python module to send the proper instructions over the bus, as described in the device's manual.

If the instrument is controlled by its own proprietary library, it will come with a software development kit intended for linking with C programs. You'll have to interface this with your high-level language (in this case, Python), and the best way to do this is to write a small set of C routines that acts as a wrapper to the instruments library. Linking these routines to Python is quite simple using the `"ctype"` module. You can even pass arrays created in Python to C and back (http://scipy.org/Cookbook/Ctypes2), pushing all the memory-management problems out of C. I have indeed found that improper uses of `"malloc"` and `"free"` are the source of countless bugs in programs written by inexperienced users.

At this point, you might ask, "Why use Python at all if I have to work in C?" First, not everyone on the team has to learn C: once an instrument is linked, the work doesn't need to be redone. Second, using two languages enforces good coding discipline:[7] the low-level, device-dependent code is pushed out in the C code, but the Python code stays clear and readable. Finally, the general control-software code is likely to be reworked many times over the life of the experiment, so it must be as agile as possible. The time you gain by using Python is well worth the effort.

Unit Testing the Experiment

As a code base grows, it becomes necessary to test the elementary operations on which it relies. Unit testing examines a program's internals, to see if they're working to specification.[8] If unit tests fail, they give valuable information about where the bug lies.

Systematic testing is a well-established software engineering technique.[9] A common testing trick is to replace a complex object with a simpler one that the test can use to explore the program's behavior—for instance, a replacement for a camera could return a well-controlled image to test the processing. Having "mock objects" replace the experiment's hardware lets you test the software without relying on or affecting the experiment itself. In a lab environment, hardware isn't always reliable, so

Authorized licensed use limited to: CR Saclay. Downloaded on June 12,2010 at 09:26:42 UTC from IEEE Xplore. Restrictions apply.

in case of a failure, running tests replacing or inspecting instruments narrows down the problem quickly. By spending time implementing an object that behaves like an instrument, the developer is also forced to understand the instrument more fully. Finally, unit testing allows the control software to run without instruments, which means you can modify and test without shutting down the experiment or the team. This, in itself, is a huge gain.

Performing Operations in Parallel

An experimental run involves sending instructions to—and waiting for answers from—different instruments, which often means waiting for signals on different buses. You might be able to evaluate before hand in which order you should receive these signals and then wait for one after the other, switching the bus you listen to each time you receive one, but this is awkward. Not only will a failed transmission bring down the whole experiment, but a change in the experimental sequence will also force a major software rewrite. Similarly, data analysis must be fast enough to finish in the time lapse between two incoming hardware signals or else you'll miss the second one.

Performing operations in parallel allows for much more flexibility and robustness. A portion of a program that can run concurrently with other portions while sharing objects is called a *thread*. In experimental-control software, it's wise to use one thread to run the interface, one to perform numerical-intensive data analysis, and one for each bus that needs monitoring. This makes it so computations don't block the interface, either for the user or the hardware. If a call to a certain piece of equipment is long, you can make it nonblocking by using a separate thread.

A threading module in Python makes it easy to set up threads (see Figure 1). However, you must be careful when programming with different threads: an object accessed by more than one thread is likely to cause *race conditions*, in which the program's behavior becomes critically sensitive to the relative timings between events. These situations can cause erratic crashes: two threads modifying an object at the same time leave it in an inconsistent state. A good rule of thumb is to have only one thread modify an object; any other threads should just read the object's attributes. Of course, more complex schemes with locks that prevent concurrent modifications of objects are possible, but they require more experience.

Event-Driven Programming

In most typical programs, the developer lays down instructions in the order in which they'll be executed. Having the computer react to experimental events requires a paradigm shift.

```
from threading import Thread
from time import sleep

def delayed_print(message):
    sleep(1)
    print message

Thread(target=delayed_print,
      args=('My thread done',)).start()
print 'Main thread done'
```

Figure 1. Using threads in Python. The `delayed_print` is called in a separate thread so that it doesn't block the program's execution. `Main thread done` is displayed before `My thread done`.

Event-driven programming solves this by listening for events and accumulating callbacks on an event queue. A worker thread empties the event queue, executing callbacks one after the other (see Figure 2). This pattern ensures both that events aren't lost and that they're processed in the order in which they're received. It also limits the number of threads: all callbacks are executed sequentially, which makes implementation much easier than starting a new thread per event. In an experiment with a large number of instruments talking to the computer on different buses, you can implement this with a listener thread that loops over the different buses and polls each instrument. The listener thread can feed events to the worker thread.

Building GUIs

Software control means providing information to the experimentalist about the experiment and letting that person interact with it. The software must therefore have an interactive GUI, but building one is hard because it requires graphical entities and developers don't choose the program's flow. Instead, the developer builds objects and specifies how they react to user actions. The toolkit used to provide basic objects also provides an event loop that catches user-generated events and calls the corresponding actions. Non-GUI-related work, such as polling for experimental data or processing it, should happen in different threads.

Out of the various toolkits available to build GUIs, wxPython is a versatile and powerful choice. As with all GUI frameworks, a wxPython program is made first by creating a graphical object, starting with a window, and then populating it. The

Authorized licensed use limited to: CR Saclay. Downloaded on June 12,2010 at 09:26:42 UTC from IEEE Xplore. Restrictions apply.

```
from collections import deque

class Dispatcher(deque):
  working = False

  def dispatch(self, function, *args):
    self.append((function, args))
    if not self.working:
      self.working = True
      Thread(target=self.__consume).start()

  def __consume(self):
    while self.__len__():
      fun, args = self.popleft()
      fun(*args)
    self.working = False

d = Dispatcher()
d.dispatch(delayed_print, '1')
print '2'
d.dispatch(delayed_print, '3')
print '4'
```

Figure 2. Event-loop dispatcher. The dispatch method of the `EventQueue` adds functions on the event stack and starts the event loop if it isn't running yet. In this example, numbers are displayed in the order 2, 4, 1, and 3.

```
from enthought.traits.api import *
from enthought.traits.ui.api import View
from enthought.pyface.api import GUI

class Dialog(HasTraits):
  index = Int()
  button = Button()

  def _button_fired(self):
    self.index += 1

  view = View('index', 'button', buttons=['OK'])

d = Dialog()
d.edit_traits()
GUI().start_event_loop()
print d.index
```

(a)

(b)

Figure 3. Interactive dialog created with traitsUI. (a) The `edit_traits` method creates a dialog representing the object. (b) Pressing the button fires the `_button_fired` callback that adds 1 to the `index` attribute. This modification reflects immediately in the dialog and later when the attribute of the object is printed: the last line prints "3" if the button is pressed three times.

program then calls `"MainLoop"`, which starts the event loop. The code looks very different from procedural batch programming, which is what most scientists are used to, so it can be baffling at first, but it's very expressive once you get used to it.

Building graphical objects and their callbacks can be a time-consuming and repetitive task. The code is cluttered with references to GUI elements, and important data-processing tasks can be hidden and hard to read, which leads to both bugs and frustration in the long run. Most of the time, a scientific application only requires displaying and editing some variables, the modification of which triggers the code to update the experiment's logics. The traitsUI[10] module can generate wxPython dialog panels from objects, which lets you modify their attributes and removes a lot of the boilerplate work, making the GUI a visual representation of objects in the code. GUI elements vanish from the code: traitsUI completely takes care of the correspondence between the object and its representation (see Figure 3).

Data-Driven Programming

Data retrieved from the experiment must be stored and displayed to the user. Similarly, the user must enter parameters that act on the experiment's control logics so that data can be exchanged between objects and across execution threads. Rather than explicitly propagating data changes, the sharing of data between objects by storing references instead of values helps make the code light and flexible.

A complex experiment-control program has anything but a linear flow. Control over the program comes from the program's own logic, the user, and the hardware to which the computer is connected. Event-driven programming techniques let the software respond to hardware and user interaction by letting external events trigger callbacks. However, this can also lead to strong coupling in the code: each procedure that processes the data retrieved from an instrument—or input by a user—must be listed as a callback.

The traitsUI package automatically updates the representation of object attributes when their value changes—it can even fire a procedure. This is somewhat akin to the idea of "don't call us, we'll call you": data processing is triggered by the change in the data itself (see Figure 4).

This pattern is very efficient at reducing the explicit coupling between objects. You can store the data and parameters that describe the experiment and its results in well-chosen classes, with methods to process the data and propagate it to the experiment or the user interface.

Authorized licensed use limited to: CR Saclay. Downloaded on June 12,2010 at 09:26:42 UTC from IEEE Xplore. Restrictions apply.

The techniques of loose coupling by sharing data across threads and objects and event-driven programming fall under the general design principle of control inversion. The program is built as an ensemble of objects and procedures linked together by data (see Figure 5).

They say that judgment comes from experience and that experience comes from poor judgment. I hope this isn't completely true and that judgment can also be acquired through advice. I took the long way and got stuck in the software development tar pit. I was told that you don't study programming; rather, you learn it on the spot. Luckily, I'm not only an experimentalist but also the brother of a computer scientist, and my difficulties in the lab raised numerous discussions. Since then, I've found myself recoding some legacy applications written by a postdoc in which the poor fellow always chose the painfully hard solution. Thinking ahead and choosing the right tools have helped me be more productive and rebuild months of work in just a few weeks. Unfortunately, once we've learned our lessons in experimental labs, we move along to other tasks without passing that knowledge along. CiSE

```
class SquareFilter(HasTraits):
  input   = CFloat(0)
  output  = Float()

  def _input_changed(self):
    self.output = self.input**2

f = SquareFilter()
f.input = 2
print f.output
f.input = 10
print f.output
```

Figure 4. Dataflow programming with traits. Changing the `SquareFilter` object's `input` attribute automatically changes its `output` attribute. This example outputs "4." and "100." successively.

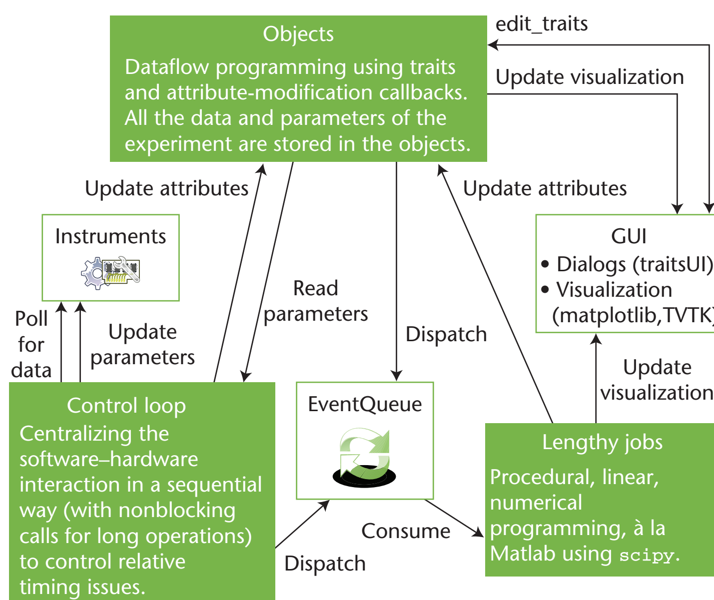

Figure 5. Schematic diagram. The full experimental program depicted here uses the building blocks presented in this article.

Acknowledgments

I thank Prabhu Ramachandran and David Morill for the help they gave me with threads and user interfaces. I also thank Thomas Pornin for teaching me the limits of real-time computing and steering me away from it, and Joseph Thywissen for the risk he took by letting me apply unusual ideas to the software controlling his experiment. Finally, all authors and contributors to the software mentioned here deserve a big thanks.

References

1. G.V. Wilson, "Where's the Real Bottleneck in Scientific Computing?" *Am. Scientist*, vol. 94, no. 1, 2006, p. 5.
2. C. Townsend, W. Ketterle, and S. Stringari, "Bose-Einstein Condensation," *Physics World*, vol. 10, no. 3, 1997, p. 29; http://physicsweb.org/articles/world/10/3/3/2.
3. J.K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," *Computer*, vol. 31, no. 3, 1998, p. 23.
4. E. Jones et al., "SciPy: Open Source Scientific Tools for Python," 2001; www.scipy.org.
5. J.D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 90–95.
6. P. Ramachandran, "TVTK, A Pythonic VTK," *EuroPython Conf. Proc.*, 2005; https://svn.enthought.com/enthought/attachment/wiki/TVTK/tvtk-paper-epc2005.pdf.
7. P.F. Dubois, "Ten Good Practices in Scientific Programming," *Computing in Science & Eng.*, vol. 1, no. 1, 1999, pp. 7–11.
8. G.V. Wilson, "Software Carpentry," 2006; http://swc.scipy.org.
9. G.K. Thiruvathukal, K. Läufer, and B. Gonzalez, "Unit Testing Considered Useful," *Computing in Science & Eng.*, vol. 8, no. 6, 2006, pp. 76–87.
10. G. Varoquaux, "Writing a Graphical Application for Scientific Programming Using TraitsUI," 2006; http://gael-varoquaux.info/computers/traitstutorial/.

Gaël Varoquaux *is a research fellow at LENS, University of Florence. His research activities include the experimental study of quantum-degenerate atomic gases for long-interrogation-time inertial-sensing atom interferometry. Varoquaux has a PhD in quantum physics from Institut d'Optique, Palaiseau. Contact him at gael.varoquaux@normalesup.org.*

Authorized licensed use limited to: CR Saclay. Downloaded on June 12,2010 at 09:26:42 UTC from IEEE Xplore. Restrictions apply.