Agile computer-control of a complex experiment
Gaël Varoquaux *

Today’s experiments can involve dozens of devices that need to work together to keep the experiment running. More and more often, a computer conducts the chorus of both commercial and home-made instruments. A experimentalist has often been taught the arts of electronics, optics and mechanics required to build and run his experiment, but software engineering has often been left out[1], and, when the experiment grows in complexity, software becomes a burden and a weak point.

Scientists are skilled with computers. Many understand well the intricacy of numerical computing. Yet, designing the sophisticated software architecture that controls an experiment requires different skills, and small and mid-size experimental labs often lack software engineering culture. Bad design choices plague experimental labs, and lead to slow progress, even though the real experimental difficulty seldom lies in the software.

In this article I will try to give a few guidelines to design the control software of an experiment based on my experience in various Bose-Einstein condensation labs[2]. I will explore the tools and patterns that have lead to successful projects: a flexible and reliable code-base, allowing to cope with the ever-changing goals and man-power of a research lab with short development cycles.

Avoiding timing problems
Computers can be used both for instrument control, and for data acquisition. The ability to put elaborate logic in a computer program opens the dream to replace electronic boxes lying around the lab with computers. Modern computers are clocked at several gigaHertz, so it is common belief that they will be able to replace servo-locks and electronic timers, or to do on-the-fly processing of data. The bad news is that computer systems, both hardware and software, are optimized for throughput and not for short response times. Latencies of several milliseconds forbid any control-loop with frequencies higher than 100Hz, apart on dedicated systems (real-time operating systems or embedded devices).

The timing issues can be moved out of the computer into well clock electronic devices by relying on an external clock to trigger the hardware-related actions. If necessary input and output buffers can help freeze the experimental time-base while the computer retains some flexibility. This pattern is very efficient at solving most real-time experimental problems, as long as the frequency does not exceed a few kHz, and the computer is fast enough to perform the work during a clock-cycle. Embedded systems programmed by the computer often provide good solutions and, in case of commercial systems (eg http://www.adwinproducts.com/), can be easier to implement and more reliable than home-brewed complex software solutions.

Using the right tools
If speed is paramount, even after moving all the timing-critical operations to external devices, the control software will have to be written in C or C++, possibly with the help of a framework such as LabWindows or Root (http://root.cern.ch). However one should try to avoid such low-level languages as much as possible [3]. Not all scientists are familiar with memory management or linking and compiling. The use of a low-level language increases development time, makes it harder to contribute for a new-comer, and increases the chances of bugs and design errors.

The solution used to build an experiment-control software needs to be accessible to beginners, suitable for large projects as well as small ones and to allow for rapid development. It needs to have a rich standard library, so that the developer does not loose time implementing visu-
alization or disk operations, and it should also be math-aware. We will see later that it must also have good support for multiple threads. Not many languages or frameworks meet these criteria. LabView is not scalable, as it is based on graphical computing, and will be suited only for small projects. MatLab is math-aware, but is single-threaded.

With the recent achievement of powerful numerical modules\cite{4}, Python provides a good candidate. It is an agile language focused on ease of use and speed of development, while keeping advanced features. I will focus in this article on the use of Python though many of the considerations exposed here exceed the choice of a language.

Whatever the choice of the platform may be, it is a huge gain in productivity and reliability to use existing libraries, for instance for visualization. Python provides great modules for visualization (matplotlib\cite{5} or chaco for 2D plot and \cite{6} for 3D plots), but every major language has libraries for plotting (eg. PLplot, for C), and implementing such commons tools is both a waste of time and a threat to the reliability and maintainability of your project.

Controlling the hardware
All the instruments to be controlled by the software need to have a internal representation. Instruments are connected to the computer through various technologies, they use different sets of low-level instructions to receive and send information. This is not relevant to the general goal of the control-software: taking a picture with a camera connected to the firewire port should not 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 an interface, a universal set of instructions that do not depend on the implementation, and a separate layer will translate this to instrument-related gibberish. This is important as it makes the main code-base more readable, and it allows modularity, which both helps to diagnose bugs, and gives the option to easily replace the hardware.

Object oriented programming is well adapted to such modularity and abstraction. The details of the implementation are hidden in the internals of the objects and only meaningful methods and attributes, the interface, are exposed in the calling code.

The task of writing the methods to talk to the hardware itself often implies to get your hands dirty. Python has modules to control VISA (Virtual Instrument Software Architecture) capable instruments, or devices supporting Measurement Computing(TM)'s Universal library, but you are most likely to have build your own interface to your hardware. If the instrument is connected to a standard bus (serial, GPIB) you will have to use the adequate python module to send the proper instructions over the bus, as described in the devices' manual.

If the instrument is controlled by its own proprietary library it will come with an SDK (Software Development Kit) intended for linking with C programs. You will have to interface this with your high-level language, Python. The best way to do this is probably 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 "ctypes" module. You can even pass arrays created in Python to C, and back \url{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.

You may ask: why use Python at all if I have to work in C ? First of all, not every one on the team has to learn C, once an instrument is linked-to the work does not need to be redone. Second, using two languages enforces good coding discipline\cite{7}: the low-level, device-dependent code is pushed in the C code, and the Python code stays clear and readable. Finally, the general control-software code is likely to be reworked many times through the life of the experiment, and it need to be as agile as possible. The gain in time by using Python is well worth the effort.

Unit-testing the experiment
As a code-base grows it becomes necessary to implement tests of the elementary operations it relies on. Unit-testing tests the internals of a program, to see if they are working to specification \cite{8}. It allows to work on an existing program without breaking it by testing that the modifications do not introduce bugs. If unit-tests fail
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: Simple illustration of using threads with Python. The delayed_print is called in a separate thread, so that it does not block the execution of the program. 'Main thread done' is displayed before 'My thread done'.

Figure 2: Simple illustration of using threads with Python. The delayed_print is called in a separate thread, so that it does not block the execution of the program. 'Main thread done' is displayed before 'My thread done'.

eyen to just each time you receive one. However this is awkward. Not only will a failed transmission bring down the whole experiment, but also a change in the experimental sequence will enforce major rewrite of the software. In a similar way, a data analysis needs to be fast enough to be finished in the time-lapse between two incoming hardware signal, elsewhere you will 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 an experimental-control software it can be wise to use a thread to run the interface, one for the numerical-intensive data analysis, and one per bus that needs monitoring, this allows the computations not to block the interface both for the user, and for the hardware. If a call to a certain piece of equipment is long, it can also be made non-blocking with the use of a separate thread.

A "threading" module is available with Python that allows to set up threads easily (see fig. 2). However one must be careful when programming with different threads; an objects accessed by more than one thread is likely to cause "race-conditions": the behavior of the program becomes critically sensitive to relative timings between events, most often causing erratic crashes of the program caused by two threads modifying the object at the same time, and leaving it in an inconsistent state. A good rule of thumbs is to have only one thread modifying an object,
from collections import deque

class EventQueue(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

eq = EventQueue()
eq.dispatch(delayed_print, '1')
print "2"
eq.dispatch(delayed_print, '3')
print "4"

Figure 3: Event-loop dispatcher. The dispatch method of the EventQueue adds functions on the event stack and starts the event loop if not yet running. With the example application code, number are displayed in the order: 2, 4, 1, 3.

and others only reading its attributes. Of course more complex schemes with locks preventing concurrent modifications of objects are possible, but they require more experience with these matters.

Event-driven programming

In programs as we usually think of them, instructions have been laid down by the developer in the order they will be executed. Having the computer react to experimental events requires a paradigm shift.

Event-driven programming solves this by listening for events and accumulating callbacks on an event queue. A worker thread empties the event queue, executing the callbacks one after the other (see fig. 3). This pattern insures that events are not lost, and are processed in the order they are received. It also limits the number of threads: all the callbacks are executed in a sequential way, one after the other. This makes implementation much easier than starting a new thread per event. In an experiment with a larger number of instruments talking to the computer on different buses, this can be implemented with a listener thread, looping over the different buses and polling each instrument. The listener thread can feed events to the worker thread.

Building GUIs

Software-control of an experiment also means providing information to the experimentalist about the experiment, and allowing her to interact with it. The software must therefore have an interactive graphical user interface (GUI).

Building a GUI is hard because it requires laying-out graphical entities, and because the flow of the program is not chosen by the developer, but by the user interaction.

Instead of specifying what will happen at which time, a GUI 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. Like with all GUI frameworks a wxPython program is made by first creating the graphical objects, starting with the window and populating it with objects, specifying their callbacks when needed. The "MainLoop" is then called, and the event-loop starts. The code looks very different from procedural batch programming, to which most scientists are used, and can be baffling at first, but it is very expressive once you get used to it. Learning new tools takes time, but complex problems require sophisticated tools.

Building the graphical objects and their callbacks can be a time-consuming and repetitive task. The code is cluttered with references to GUI elements and the important data processing tasks can be hidden and hard to read. This leads to bugs and frustration, in the long run. Most of the time a scientific application only requires to display and edit a number of variables. Modifying these variables triggers code to update the logics of the experiment. The traitsUI [10] module can generate wxPython dialog panels from objects allowing to modify their attributes. This removes a lot of the boilerplate work and makes
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 += "o"

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

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

Figure 4: Interactive dialog created with traitsUI. The edit_traits method of the object create a dialog representing it. Pressing the button fires the _button_fired callback that adds 1 to the index attribute. This reflects both immediately in the dialog, and when the attribute of the object is printed: the last line prints "3" if the button has been pressed three times.

the GUI a visual representation of objects in the code, which is great for its intuitive understanding. GUI elements vanish from the code as traitsUI takes completely care of the correspondence between the object and its representation (see fig. 4).

Data-driven programming

Data is retrieved from the experiment. It has to be stored and displayed to the user. Similarly the user enters parameters that act on the control logics of the experiment. Data is exchanged between objects and across execution threads. Rather than explicitly propagating data changes, sharing data between objects by storing references rather than values helps making the code light and flexible.

A complex experiment-control program has all but a linear flow. Control of program comes both from the program’s own logic, the user, and the hardware that the computer is connected to. Event-driven programming techniques allow the software to respond to hardware or user interaction by having external events triggering callbacks. However it can lead to strong coupling in the code: each procedure that processes the data retrieved from an instrument, or input by a user, has to be listed as a callback.

The traitsUI package automatically updates the representation of objects attributes when their value is changed, it can even fire a procedure. This allows to use the Hollywood Principle - "Don’t call us, we’ll call you": the processing of the data is triggered by the change in the data itself (see fig. 5).

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

Loose coupling achieved by sharing data across threads and objects, the heavy use of the Hollywood Principle and of event-driven programming fall under the general design principle of inversion of control. The program is built as an ensemble of objects and procedures linked together by data and that react to events. Ref. 8 contains

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 5: Data-flow programming with traits. Changing the input attribute of the SquareFilter object automatically changes its output attribute. This code snippet outputs "4." and "100." successively.
Figure 7: An application for the control of a Bose-Einstein condensation apparatus. The experiment is synchronized using an embedded device that runs sequences loaded by the computer. It triggs the different instruments that can be programmed by the computer. Amongst other things, a camera is trigged. The software has an acquisition loop that queries the camera for new data and spawns a processing job. All the information set by the user to control the experiment is held in an object used to generate the sequences, but also accessed for processing parameters.

an detailed example of such an application built using traitsUI.

Conclusion
They say that judgment comes from experience and that experience comes from poor judgment. I hope that this is not completely true and that judgment can also be acquired through advices. I took the long way, and sunk in the software development tar pit. I was told that you do not study programming but that you learn it on the spot. Luckily I am not only an experimentalist, but also the brother of a computer scientist, and my difficulties in the lab raised questions. A few years latter I have found myself recoding some legacy application written by a missing in action post-doc and where the poor fellow always chose the painfully hard solution, thinking ahead and choosing the right tools has allowed me to be more productive and rebuild months of work in a couple of weeks (see figure 7). Unfortunately quite often in experimental labs, once we have learned we move along to other tasks without passing the knowledge.

Acknowledgments
I would like to thank Prabhu Ramachandran and David Morill for the help they gave me with threads and user interfaces. I would also like to 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 mentioned software deserve a big thanks.

References
Instruments

Objects
Dataflow programming, à la Labview, using Traits and attribute-modification callbacks. All the data and parameters of the experiment are stored in the objects.

Control loop
Centralizing the software–hardware interaction in a sequential way (with non-blocking calls for long operations) to control relative timing issues.

Event Queue

GUI

Dialogs (TraitsUI)
Visualization (matplotlib, VTK)

Lengthy jobs
Procedural, linear, numerical programming, à la Matlab using scipy.

Figure 6: Schematic diagram of a full experimental control program using the building blocks presented in this article.