Pyexp - Python for Experiments

by Laurent Pointal
laurent.pointal@lure.u-psud.fr

S.I.E. (Service Informatique Expériences)
LURE (CNRS/CEA/MENRT)

I presented the Pyexp project in may 1999 at the first Python France Day. Since, some pyexp tools were developed, but other priorities have delayed the project itself. Now, it is a priority for some experiments at LURE.

Differents Levels

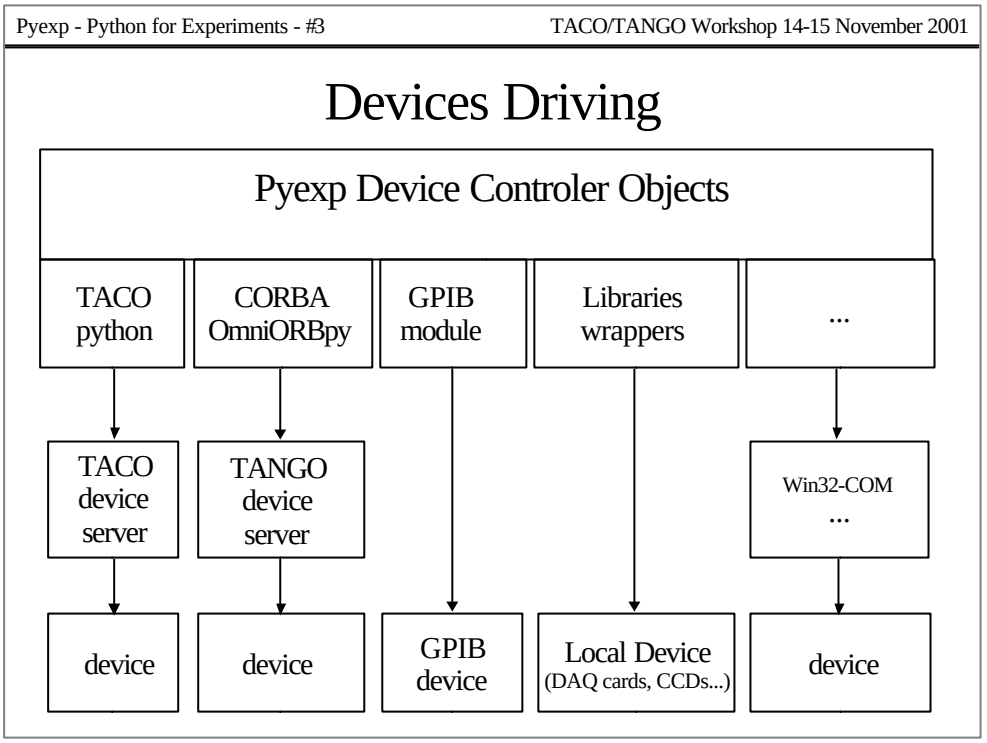
- Devices driving
- Graphical User Interface
- Experiment control
  - User scripting
  - Abstraction with physical elements
  - Hidden parallel processing and error management framework

Device driving
- Use of existing device control systems.
- Easy development of controlers for new devices.

GUI
- Easy interface construction 'kit' (eventually accessible to advanced users), with a set of reusable dialogs to select experiment parameters for standard experiments.
- Complete application to display and plot data during acquisition and manipulate them.

Experiment control þ Python scripts
- Users can write their own scripts ; script macros must allow to easily express experiment processing structures (loops nesting...) and parameters.
- Standard scripts for standard experiments, with a link to get parameters from GUI if available.
- Abstraction with physical elements make scripts more reusable.
- Modern computing tools (multithreading...) are efficient, but complex to understand and use for simple scientific programmers.
- Link with device driving layer.
- Run on different operating systems.

The right tool at the right place.

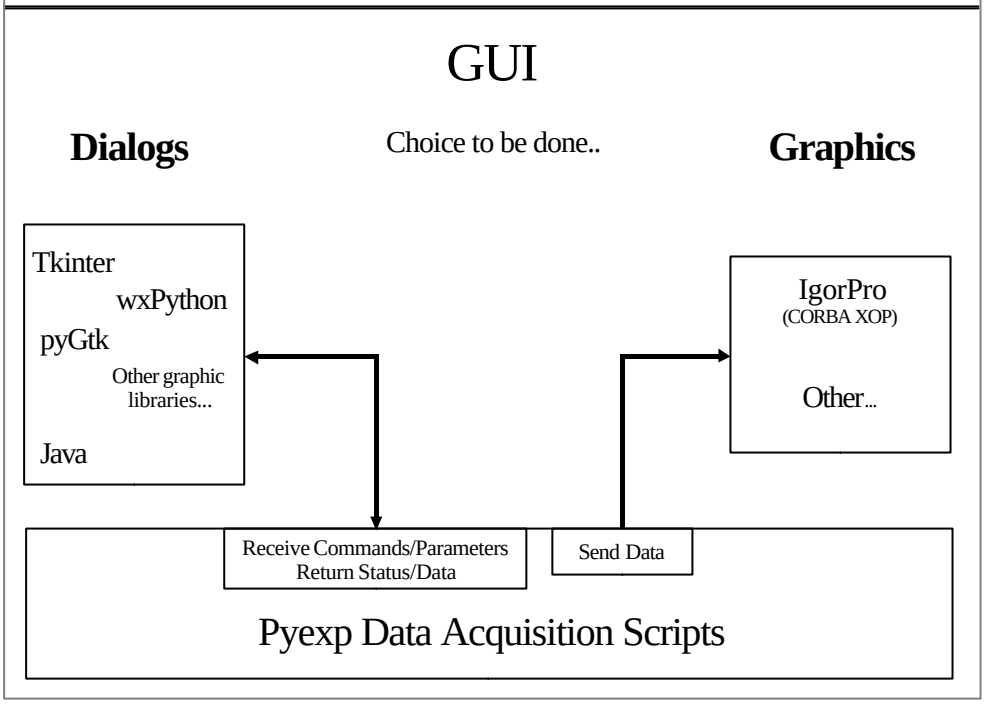


From a base device controler class, we define subclasses adapted to different devices protocols and/or kind.

Python is an excellent glue language, it has interfaces with most of existing protocols. And it is possible to write extension module wrappers for specific libraries (ex. we did it for IEEE488.1 functions of the National Instrument GPIB library).

With that structure, we develop device controler classes in Python, they can directly control devices, or they can link to existing device servers. New devices management can take place in Python code or in TACO or TANGO device servers.

Note: We have written a configuration tool based on configuration URLs (coding the protocol to use and the location to access configuration data) and 'entity' names at these locations, which allow to write code independently of the configuration system(s) used on the experiment. With that tool, pyexp can easily use any (write an adaptation class) and multiple (use different configuration protocols) configuration systems , and is then more adaptable and scalable.

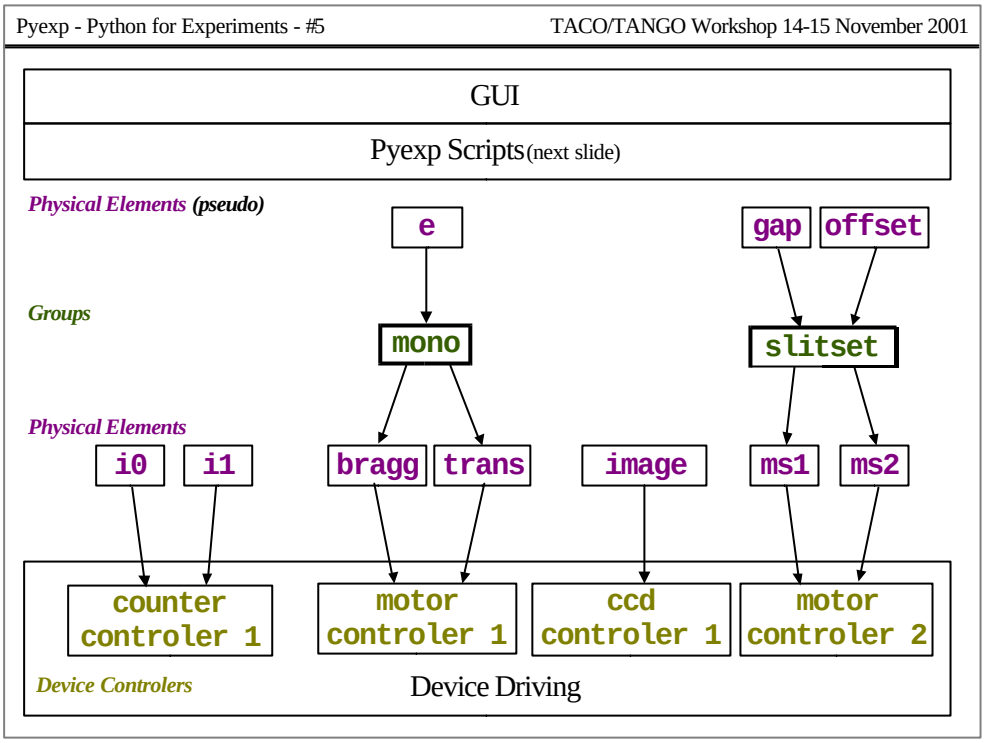


Technology choices for GUI are not closed.

We tries to consider the GUI side of the experiment as an external and optional tool.

It it possible to directly use graphic libraries which have Python binding, or even to link with Java programs via a CORBA interface (benefit of Java RAD tools, beans…).

For plotting, in place of developing new graphical scientific applications, it is possible to build an interface with existing ones. We did this for Igor Pro on Windows, where an XOP (Igor plugin module) creates a CORBA object. Client data acquisition scripts transmit data to this object, and scientists can use Igor as usually (data come in 'magically' in Igor waves).



The physical element abstraction allow users to deal with natural things in their scripts. They can be linked to device controlers, or to groups, or be standalone (ex. a software timer).

Groups are objects built to manage complex equipements. They can offer access for multiple physical elements to control different parameters of the equipment (like the **`slitset`** group). On the other side, they can have to drive multiple physical elements (like the **`mono`** and **`slitset`** groups). We can have groups for other things like diffractometers, hexapodes....

Physical elements (pe), groups and device controlers are all 'actors'. With their relations, they build a non cyclic oriented graph of actors. When operations are requested on one or more pe, we process by steps:

- *we pass the request among actors, starting from the concerned pe, and we get back action requests for a list of final target actors.*
- *action requests are grouped by target, and lists are checked (remove duplicates, and ask targets to check coherency).*
- *each target process its list of actions in a separate thread.*

Scripts

```
rsd = [e,i0,i1]
s = Loop (5) (
     Loop (evalues) (
        Set (e),
        Measure (i,i1),
        Read (rsd),
        Save (rsd),
        Display (rsd)
        )
     )
s.run()
```

Scripts macro instructions are subclasses of a **`ScriptNode`** class.

We define **`__call__`** method of **`ScriptNode`** objects to set up a tree of nodes. For users the open parenthesis is like the beginning of a nested bloc of the previous macro instruction. Inside parenthesis, indentation has no meaning, it only makes script more readable.

Loop can iterate on sequence objects, created from subclasses of a **`DataSequence`** class. Subclasses can manage values of the iteration as they want (constant value, linear or non-linear progression, read from a file, computation from a physical element value…).

Nodes can get/set values in their parent's nodes (ex. **`Set`** search for the nearest *'iterval'* value in its parent nodes to find the target for **`e`**).

Node objects have names, which allow to identify any macro in the script. In their code, they can request that 'patchs' be called (mimic the Spec cdef).

Normal macros processing can be modified with requests relatives to nodes (ex. when loosing light beam, pause until it come back, and request to restart last scan loop) - but that must be in the macros code.

Example : a Loop node

Built with: `Loop(LinearSeq(10000.0,1000,2.0),name="scan")`

attributes:

```
name = "scan"
parent = <reference to a node>
values = <the LinearSeq>
iterval = <set during run>
index = <set during run>
```

run code:

```
self.patchs("befnode")
self.index = 0
self.values.rewind()
while not self.values.finished():
    self.patchs("befiter")
    self.values.setindex(self.index)
    self.iterval=self.values[self.index]
    self.callsubnodes()
    self.patchs("aftiter")
    self.index += 1
self.patchs("aftnode")
```

Patchs Table

node	part	patch
*	befnode	patch1
scan	befiter	patch2
scan	aftiter	patch3
set	*	patch4
...	...	...

Here is an example for a **`Loop`** node.

At construction, the node may also receive an integer and use it to build a simple stepping sequence (use of Python dynamic typing).

Code to reset the loop from an external request, to process lists of sequences… must be added.

Patchs are not directly associated to nodes but to nodes names and nodes parts (like "befiter") which identify where in the node script the patchs are called). With that association by names, nodes are called whatever be the script. Tools allow to manage patchs (insert/remove, enable/disable, change calling order…).

Project Status

- Base tools are written (multithreading and synchronization, configuration data access, scientific data plot in Igor Pro...).
- Physical elements, groups and device controlers are well in progress.
- GUI tools are still not fixed.
- Macro scripts are planned to have base version running on march 2002 for one experiment.

Base tools are already used on a first experiment program written with Python (but without pyexp itself).

Dynamic construction and links between physical elements, groups and device controlers objects are done. We are working on preparing and transmitting action commands between these objects.

The choice of tools for GUI must be done while thinking of SOLEIL future experiments.

The march 2002 experiment will be our first demonstration of pyexp, with scientists using the software on a running experiment.

We hope this first test will make pyexp a good candidate when the data acquisition software choice will be done for SOLEIL.