CNS*2006 Workshop: Interoperability of neural simulators
Organizer: E. De Schutter

A Better Interpreter For Computational Neuroscience

Eilif *"Python"* Muller (KIP), Andrew Davison (UNIC-CNRS),
Thierry *"XML"* Viéville (INRIA Sophia)

contact: emueller@kip.uni-heidelberg.de

FACETS
www.facets−project.org

Ruprecht-Karls Universität Heidelberg

Kirchhoff Institute for Physics

Overview

- Nearly all neural simulators have an interpreter interface.
- The interpreter defines the user experience.
- The interpreter is just an interface, and could be replaced by, exchanged for or interfaced to another interpreter.

Problem:

We seek interoperability between simulators and (FACETS) hardware systems.

Solution:

Seek interoperability at the interpreter level:

- Decide on one interpreter which is powerful, widely used and developed (external to Comp. Neurosci.), and has an optimal user experience (GUI, syntax).
- Interface simulators and hardware to this one interpreter.

Simulator	Interpreter	(Subjective) User Experience
NEURON	HOC	pseudo OOP, GUI and some analysis tools. CONS: Proprietary, underdeveloped as general interpreter.
GENISIS	GENISIS/SLI	? CONS: Proprietary
CSIM	MATLAB	PRO: general on-line analysis. CONS: weak as a programming language, expensive toolboxes, distribution problems, not open-source
NEST	NEST/SLI	Reminds me of my HP48gx
Surf-Hippo	Common Lisp	PRO: It's Lisp! CONS: weak numerics, weak Qt/GTK bindings, no hdf5, "all those parentheses make my head explode!"
MVASPIKE	Python	advanced dynamic OOL, batteries included: hdf5, MPI, Qt, GTK, SciPy, NumPy, Matplotlib, GSL, ROOT, like giving sight to the blind ...
FACETS Hardware	?	→ Python looks most promising

Things to Consider

Many of these interpreters still lack:

- Recent interpreter design innovations: OOP, Dynamic types.
- analysis tools à la MATLAB.
- distribution of the interpreter allowing scalable analysis on-line.
- GUI using Qt, GTK.
- bindings to tools such as hdf5, GSL, databases.
- interoperability with hardware systems.

All of these features come in at the interpreter level.

- A "One Interpreter" approach allows this development effort to be done once.
- Indeed we should seek an interpreter which has much of the work already done!

The Principle of Reusing Trusted Code

$\longrightarrow$ NEURON, NEST, GENESIS all implement interpreters of various maturities.

$\longrightarrow$ While mature modern dynamic object oriented open-source interpreters already exist: Lisp, Ruby, Python.

By adopting Python one leverages:

- An interpreter developed and deployed by NASA, Google, ILM, venture capital, etc.

  $\Rightarrow$ No need to expend Neuroscience resources developing general but "throw-away" interpreters.

  $\Rightarrow$ We teach Neuroscientists a programming language which will get them a job at Google when their grant proposal is rejected.

- A module library said to be rivaled only by PERL.
  - ⇒ "Batteries included": hdf5, MPI, Qt, GTK, SciPy, NumPy, Matplotlib, GSL, VTK, Blender?, OpenDX?, Diffpack, databases, ...
- A numerical and scientific Python community which have implemented extensive MATLAB-like features:
  - ⇒ avoid MATLAB for off-line analysis.
  - ⇒ allows distributed analysis of distributed simulations on-line.
  - ⇒ C-like efficiency when datasets are large enough (usually the case in practice).
- Recent interpreter design innovations:
  - ⇒ Clean and clear syntax can be learned in days.
  - ⇒ Fully dynamic language: name resolution and types.
  - ⇒ Modules (aka packages in Java) provide organizational structure above the class.
- A well defined C/C++ API for the inner-loop if necessary.

Hybrid Computing: The Python Movement

- Utilize Python as a middleware language for its flexible system integration capabilities.
- Efficient C++ to define new Python types, low-level libraries, remove performance bottlenecks.
- Boost.Python or SWIG automate the exposure of C++ classes to Python.

Looking at programming language history:

- Python is now to C/C++ as C/C++ was to assembler 10-15 years ago.

⇒ Python is more than a simulator add-on: makes next incremental step since C++ in solving the complexity problem in software development.

⇒ The gained expressivity will make way for breakthroughs in Computational Neuroscience by improving programmability.

⇒ Develop more complex simulations in a shorter time.

How to implement PyANYSIMULATOR ?

Stage 1:

Keep the original interpreter running below Python and implement communication between the two.

PROS: A quick and general Python interface.
CONS: Less than optimal performance and memory efficiency.

Stage 2:

Gradually expose the class structure in Python to improve efficiency where required. Finally remove the old interpreter if appropriate.

PROS: Addresses performance and memory issues. Adds OOP to presently non-OOP simulators such as NEST/SLI.
CONS: Alot of work. Tools such as Boost.Python, SWIG or others exist to automate the process.

PyNEST and PyNEURON(?) both have an existing Stage 1 interface.

Stage 1 Example: PyNEST

Four commands are written in C/C++ and exposed to Python as the `pynestkernel` module:

- `run(string SLI_command)`
  Sends a string to be executed in NEST/SLI.

- `push(Python object)`
  Converts most Python datatypes into NEST/SLI types and pushes them onto the SLI stack.

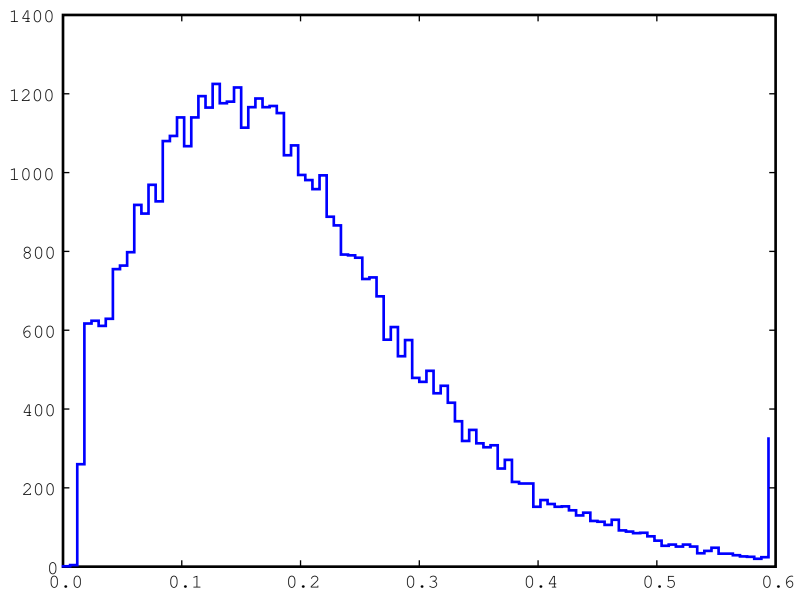
- `pop() returns Python object`
  Pops the last object on the SLI, converts to a Python object and returns it.

- `SLIstacksize() returns int`
  Returns the current size of the SLI stack as interger. Used for debugging purposes.

A high-level PyNEST interface with `create`, `set`, `get`, `connect`, `simulate`,... is built from these low-level commands.

Stage 1 Example: PyNEST

```
import pynest as pyn

cbn = pyn.create('iaf_sfa_neuron',1)
eParams = {'Theta':-57.0, 'Vreset': -70.0, 'TauR': 0.5,...}
pyn.setDict(cbn,eParams)

cbnD = pyn.create('pyspike_detector',1)
#... define static poisson input, setup recorders ...

pyn.simulate(1e7)

# Basic analysis: ISI Histogram
st = pyn.get(cbnD[0])['spike_times']
isi = st[1:]-st[:-1]

import NeuroTools.nstats as nstats
bins = arange(0.0,0.6,0.6/100)
h = nstats.histc(bins,isi)
import matplotlib.pylab as pylab
pylab.plot(bins,h,linestyle='steps')
```



PyNN: Simulator/Hardware Independent Modeling

- Developed by Andrew Davidson, UNIC-CNRS
- Basic Idea: API specifies high level classes: Populations, Neuron types, connection methods, etc. which a simulator specific name space implements

Example:

```
# choice of platform before running setup operations
from PyNN.nest import *
or
from PyNN.neuron import *
or
from PyNN.hardware import *
or
from PyNN.genesis import *

eparams = {'vth':-57.0,'tau_m':10.0, ... }
epop = Population((1000,),IAF_SFA_Neuron, eparams, "pop. name")
```

Going Beyond the State of the Art with Python

- A master/slave framework for on-line distributed interpreting.
- Uses MPI4PY (MPI bindings for Python).
- We don't want to call it `PyMPImaster`...
- Currently we use it to implement a generalization of Google's MapReduce tool:
  - Free slaves process in turn a list of job parameters.
  - Process is defined by a list of slave stages and a master response handler defined by arbitrary Python code.
  - Allows on-line distributed simulation and analysis without writing to disk.
- Work in progress: integration with distributed NEST.
- More details in workshop: "Exploring large spiking networks using NEST"...

Software Development Timeline and Milestones

1. PyCSIM (Feb 2005): Replaced CSIM MATLAB interface with Python allowing multiple simulation cores using MPI.
   Lesson: With Python running on the Master node, setup and analysis did not scale.

2. FACETS Software Taskforce Meeting (May 2005): I proposed Python as a uniform interface to all neural simulators as foundation for software collaboration in FACETS.
   Synopsis: Official support for the idea was weak. The consortium defaulted to support development of NeuroML.

3. PyNEST (Arcachon CNS Course, August 2005): worked with Marc-Oliver Gewaltig and Markus Diesmann to build PyNEST where Python runs locally (no MPI support).

4. PyNEST+MPI4PY (April 2005): Using python MPI binding (MPI4PY) I built a master/slave framework using an approach similar to but more general than Google's MapReduce laying a foundation for scalable distributed setup and on-line analysis.

Software Development Timeline and Milestones

5. PyNN (May 2006): Andrew Davidson (CNRS-UNIC) implemented a prototype namespace and class structure to allow specification of a model which runs seamlessly on PyNEST or PyNEURON.

6. PyHAL (August 2006): Daniel Bruederle (KIP) will write a low-level Python interface to the Stage 1 FACETS neural hardware.

7. Distributed PyNEST (∼October 2006): I am working closely with the NEST consortium to merge the master/slave interpreter framework with the "to be released" distributed NEST.

8. PyHAL in PyNN (Septempber 2005): PyNN will be expanded to include support for Stage 1 FACETS hardware using PyHAL..

9. PyCSIM in PyNN (Late 2006): In collaboration with developers at TUGraz, the Python interface to CSIM will be rewritten and integrated in the PyNN framework.

10. Distributed PyNN (Early 2007): Integrate PyNN and Distributed PyNEST work.

Final Remarks

- Python over Lisp or Ruby as a Single interpreter interface: "Glue" for simulators, hardware, tools, workflow scripts, GUI, ∞
- Proof-of-principle implementations clarify system design details.
- Defined Roadmap/Milestones towards simulator/hardware independant model specification.
- Implemented foundations for a state-of-the-art distributed interpreter architecture allowing scalable distributed setup and analysis for all simulators with a Python interface.
- Long term: Provide Stage 2 PySimulators + tools with an OOP framework for interoperability at the interpreter level.
- My Provocative Statement: Python is like MOOSE but ubiquitous and mature.

Hope for more ...

⋆ Dialog about an exciting opportunity for CNS.

⋆ Participation from Modelers and Developers.

⋆ Adoption and Teaching: at Comp. Neurosci. courses, etc.

Acknowledgments

- Parellel Python efforts: Mike Hines, Olivier Rochel (MVASPIKE)
- Markus Diesmann, Marc-Oliver Gewaltig, Mortiz Helias
- Andrew Davison (PyNN)
- Daniel Bruederle (PyHAL)
- Jens Kremkow, Lars Buesing
- NumPy+Scipy developers
- Guido van Rossum (Python BDFL)