ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

Extensions

- one of the major strengths of python is the ease with which you can get access to code written in low level languages
  - for leverage, reuse, performance

  the interpreter itself is written in `C`, so you can get access to code written in any language that is link compatible
- python provides
  - a `C` library that grants you access to nearly every aspect of the interpreter
  - a simple set of rules for constructing shared objects that can be imported, just like any other python module
- typically, the solution to this problem has three layers
  - the archive and headers of the library you want to expose to python
  - the interface layer that translates data from python to `C`, which becomes the importable shared module
  - a layer in pure python that provides an object oriented veneer to the library; in our case this layer will also double as the component specification
- if tempted to collapse some of these, *resist*

The bottom layer

we would like use the random number generators in `GSL`; to get a feeling about what is involved, recall our earlier `C++` solution

```
#include <iostream>
#include <gsl/gsl_rng.h>

int main(int, char*[]) {
   // the sample size
   const int N = 1.0e7;
   // initialize the counters
   int interior = 0;
   // allocate a random number generator
   gsl_rng * generator = gsl_rng_alloc(gsl_rng_ranlxs2);
   // integrate by sampling some number of times
   for (int i=0; i<N; ++i) {
      // create a random point
      double x = gsl_rng_uniform(generator);
      double y = gsl_rng_uniform(generator);
      // check whether it is inside the unit quarter circle
      if ((x*x + y*y) <= 1.0) { // no square roots
         // update the interior point counter
         interior++;
      }
   }
   // print the result
   std::cout << "pi: " << 4. * interior / N << std::endl;
   // free the generator
   gsl_rng_free(generator);
   // all done
   return 0;
}
```

we must allocate, use and free a `gsl_rng`

The top layer

for plug-and-play, we must build a component; perhaps something similar to `Mersenne`:

```
import pyre, itertools, gsl
from .PointCloud import PointCloud

class GSL(pyre.component, family="gauss.meshes.gsl", implements=PointCloud):
    """
    A point generator implemented using the large set of random number
    generators available as part of the gnu scientific library (GSL)
    """

    # interface
    @pyre.export
    def points(self, n, box):
        """
        Generate {n} random points in the interior of {box}
        """
        # unfold the bounding box
        intervals = tuple(box.intervals()) # realize, so we can reuse in the loop
        # loop over the sample size
        while n > 0:
            # make a point and yield it
            yield tuple(itertools.starmap(self.rng.uniform, intervals))
            # update the counter
            n -= 1
        # all done
        return
```

where `self.rng` is a *handle* for the `gsl_rng` pointer

Adding the new component to our package

in `gauss/meshes/__init__.py`

```
"""
Package that contains the implementations of point clouds
"""

# the interfaces
from .PointCloud import PointCloud as cloud

# the components
from .GSL import GSL as gsl
from .Mersenne import Mersenne as mersenne
```

the only reason to tuck this in `gauss.meshes` is the convenience of using the shorter name when configuring

```
[ mc ] ; configure our Monte Carlo integrator instance
samples = 10**6
mesh = gsl
region = ball
integrand = constant
```

access to `GSL` could have been provided by a user of the `gauss` package just as easily, without any access to our source code

Module access

- python extensions live in dynamically loaded libraries
  - `DLL`s on windows, shared objects on unix
  - building one is platform dependent; compilers have the right command line flags
- making an extension requires a *visible entry point* in a shared object, both of whose names python can infer correctly so that the statement

  ```
  import gsl
  ```

  can be converted into a search for a particular file, followed by a lookup of a particular symbol
- e.g. on unix, acceptable names for the `gsl` extension module are `gsl.so` or `gslmodule.so`
- looking for extensions is just another step in the sequence applied to the folders on `sys.path`, so the same rules apply as looking for regular modules

The visible entry point

and here is the definition of the entry point

```
// initialization function for the gsl module
// *must* be called PyInit_gsl
PyMODINIT_FUNC PyInit_gsl()
{
    // create the module
    PyObject * module = PyModule_Create(&gsl::module_definition);
    // check whether module creation succeeded
    if (!module) {
        // otherwise, raise an ImportError
        return 0;
    }
    // we have an initialized module
    // set the error handler
    gsl_set_error_handler(&errorHandler);
    // initialize the table of known random number generators
    gsl::rng::initialize();

    // return the newly created module
    return module;
}
```

python will call this function to create the module object, if the dynamic loading of the extension is successful

Building a table of known generators

```
namespace gsl {
   namespace rng {
      typedef std::map<std::string, const gsl_rng_type *> map_t;
      static map_t generators;
   }
}

// initialization of the known generators
void
gsl::rng::initialize()
{
   // iterate over the registered names
   const gsl_rng_type **current = gsl_rng_types_setup();
   while (*current !=0) {
      // add each one to my map
      gsl::rng::generators[(*current)->name] = *current;
      // get the next one
      current++;
   }

   return;
}
```

Module definition

the identifier `module_definition` refers to a `struct`

```
// the module documentation string
const char * const __doc__ = "sample module documentation string";
// the module definition structure
PyModuleDef gsl::module_definition = {
   // header
   PyModuleDef_HEAD_INIT,
   // the name of the module
   "gsl",
   // the module documentation string
   __doc__,
   // size of the per-interpreter state of the module
   -1, // don't touch
   // the methods defined in this module
   gsl::module_methods
};
```