

# Building scientific applications using pyre

Michael A. G. Aivázis  
California Institute of Technology

March 30, 2009

## **Abstract**

This is a short tutorial that describes how to use pyre to construct scientific applications.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Monte Carlo integration: a bit of theory</b>	<b>2</b>
<b>3</b>	<b>Two simple implementations</b>	<b>3</b>
3.1	The two-minute python script . . . . .	4
3.2	The ten-minute C++ program . . . . .	5
<b>4</b>	<b>The hunt for objects</b>	<b>7</b>
4.1	Encapsulating the random number generator . . . . .	7
4.2	Representing the region of integration . . . . .	9
4.3	Creating the driver . . . . .	10
4.4	Improving performance . . . . .	11
4.4.1	Containers: trading memory for speed . . . . .	11
4.4.2	Generators: the best of both worlds . . . . .	13
4.4.3	Alternative containers . . . . .	14
4.5	Reuse: improving and adjusting the interface . . . . .	15
4.6	Functors . . . . .	15
<b>5</b>	<b>Turning classes into components</b>	<b>16</b>
<b>6</b>	<b>The application framework</b>	<b>16</b>
<b>7</b>	<b>Hosting the application on the web</b>	<b>16</b>
<b>8</b>	<b>Building a graphical interface</b>	<b>16</b>
<b>9</b>	<b>Enabling remote access</b>	<b>16</b>
<b>10</b>	<b>Epilogue</b>	<b>16</b>
<b>11</b>	<b>References</b>	<b>17</b>

# 1 Introduction

Research codes are peculiar beasts. They are typically born to parents that are far too young and inexperienced to take proper care of them. Their early days are sickly and even their authors do not expect them to live any longer than the next paper, definitely not past the completion of their thesis. They grow up in haphazard ways, reflecting the evolution of their care provider’s understanding of some research problem, end up having far too many appendages sticking out of all the wrong places and display very few signs of any organizing principle, let alone design. Yet, many of them outlive their parent’s wildest expectations (or fears), have long, productive lives, and become focal points of entire research communities. Hated by all, but also used by all.

The good ones have buried in them precious intellectual capital; that’s the secret of their longevity. The reason they are constantly on the some-day-I-will-rewrite-this list<sup>1</sup> is almost always the lack of enough structure so that successive generations of foster parents can maintain and evolve the code. The missing structure goes by the name “modern software engineering practices” and it’s not on the list of skills that graduate students of respectable institutions are supposed to have.

Pyre, the software architecture described in this paper, is an attempt to bring state of the art software design practices to scientific computing. The goal is to provide strong scaffolding on which to build scientific codes by steering the implementation towards usability and maintainability. It’s not a substitute for the intelligence, experience and effort necessary to write robust, hardened software. But by encouraging you to ask the right design questions and make the right practices part of your software cycle, you should experience a dramatic improvement in the quality of code you write.

You will still need to shop around for a source control system and find a scalable way to build your software on multiple platforms. You will need to find a good solution for writing documentation, maintain and run test suites, and track bugs and feature requests. If you are really ambitious, you need a release management solution. Most people hope they can get away without all this overhead, but that’s just the mild form of delusion that comes from not knowing what you are in for. Writing software can easily degenerate into a chaotic practice, with small changes having potentially unbounded effects, even when it’s only you that’s doing the coding. The passage of time has a way of introducing interesting complexity in software systems.

In the next few sections, I will show you how to turn a throw-away script into a usable application. After a brief introduction to the method, we will start by writing a naïve implementation of Monte Carlo integration in both python and C++. Then, we will evolve the code by introducing object oriented concepts that will help us improve—and document—the design of the code. The last evolutionary step will be to cast the design as a collection of reusable components. Once we have that, we will explore user interfaces, web hosting, parallelism and more.

## 2 Monte Carlo integration: a bit of theory

Suppose you have a function  $f(x)$  that is sufficiently well behaved for all  $x$  in a region  $\Omega \subset \mathbb{R}^n$  and you wish to compute the definite integral

$$I_{\Omega}(f) = \int_{\Omega} dx f \tag{1}$$

The Monte Carlo integration method estimates the value of a definite integral by sampling the function at points  $x$  in  $\Omega$  that are chosen at random with uniform probability. Suppose that you have  $N$  such points forming a sample  $X_N$ . The Monte Carlo estimate for the integral is

$$I_{\Omega}(f; X_N) = \Omega \cdot \langle f \rangle = \Omega \frac{1}{N} \sum_{x \in X_N} f(x) \tag{2}$$

---

<sup>1</sup>a list that invariably matures into someone-else-should-some-day-rewrite-this

where  $\langle f \rangle$  is the sample mean of the function  $f$  and we have used  $\Omega$  as short-hand for the volume of the associated region. More details can be found in [1, 2]; see [3] for an excellent pedagogical introduction to the subject.

One can show that the error in the estimate decreases as  $1/\sqrt{N}$ . The initial reaction to this fact is that the convergence rate is rather slow. As a result, there are implementations that address this by being smart about how the region of integration is sampled; see [4, 5, 6]. On the bright side, the convergence rate is *independent* of the dimensionality of the integral, making this method very well suited for multi-dimensional integrals. Further, it is rather straightforward to write a parallel implementation and compensate for the slow convergence by computing on a large machine.

Implementations on computers employ pseudo-random generators to create the sample set. Most generators produce numbers between 0 and 1, so actual calculations require finding a box  $B_\Omega$  that contains  $\Omega$ , and building  $n$ -dimensional sampling points  $x$  by stretching and translating the unit interval to match the dimensions of  $B$ . The integration is then restricted to  $\Omega$  by introducing a function  $\Theta(\Omega)$  that takes the value 1 inside  $\Omega$  and vanishes identically outside:

$$\Theta(\Omega) = \begin{cases} 1 & x \in \Omega \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Eq. 1 becomes:

$$I_\Omega(f) = \int_{B_\Omega} dx \Theta(\Omega) f \quad (4)$$

There are now two classes of points in the sample  $X_N$ : those interior to  $\Omega$ , and the rest. Let  $\tilde{N}$  be the number of points in  $X_N$  that fall in  $\Omega$ ; then Eq. 2 becomes:

$$I_\Omega(f; X_N) = \Omega \frac{1}{\tilde{N}} \sum_{x \in X_{\tilde{N}}} f(x) \quad (5)$$

where  $X_{\tilde{N}}$  is the subset of the sample in  $\Omega$ . Now, let  $B$  be the volume of the sampling box and observe that

$$\Omega = \frac{\tilde{N}}{N} B \quad (6)$$

is a good estimate of the volume of the integration region. Further, the sum over  $X_{\tilde{N}}$  in Eq. 5 can be extended to  $X_N$  as long as  $f$  is multiplied by  $\Theta(\Omega)$ . We obtain

$$I_\Omega(f; X_N) = B \frac{1}{N} \sum_{x \in X_N} \Theta(\Omega) f(x) \quad (7)$$

In the remainder of this article, we will transform this rather innocuous expression into a sequence of computer programs of increasing complexity and, hopefully, flexibility. The goal is to construct a piece of software that will enable our end users to explore the method with as little programming on their part as possible.

### 3 Two simple implementations

Turning the math in the previous section into actual code is not very hard, especially if you don't try improve the effective convergence rate by sampling the integration region in tricky ways. To simplify things even further, we will forgo integration with non-trivial integrands for now. Instead, we will compute areas and volumes by letting  $f$  be the unit function over our chosen region.

As a warm up exercise, let's use Monte Carlo integration to compute  $\pi$ . We can get an estimate for  $\pi/4$  by computing the area of the upper right quadrant of the unit circle, as shown in Fig. 1. This will require generating points in the unit square and counting the fraction that fall in the shaded region.

The implementation strategy is very simple. Let  $N$  be the total number of points we would like to generate. We will set up two counters,  $t$  for the total number of points and  $i$  for the points that

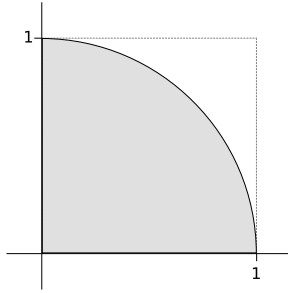


Figure 1: Estimating  $\pi$  by computing the area of a quadrant of the unit circle

fall inside the unit circle, and iterate  $N$  times building random point in the unit square, checking whether each point falls inside the unit circle and updating or counters appropriately, as shown in Alg. 1. Integration has been reduced to placing points in two bins. Note that  $\Theta(\Omega)$  in Eq. 7

---

**Algorithm 1:**  $\pi_N$ : the Monte Carlo estimate of  $\pi$

---

```

1  $i \leftarrow 0$ 
2  $t \leftarrow 0$ 
3 while  $t < N$  do
4    $x \leftarrow \text{random}()$ 
5    $y \leftarrow \text{random}()$ 
6   if  $\sqrt{x^2 + y^2} \leq 1$  then
7      $i \leftarrow i + 1$ 
8   end
9    $t \leftarrow t + 1$ 
10 end
11  $\pi_N = 4i/t$ 

```

---

corresponds to the conditional in line 6, and the dimensionality of the integrand is reflected in the number of calls to the random number generator per iteration.

It is straightforward to generalize this procedure to compute integrals of non-trivial integrands over higher dimensional regions. Perhaps you can already see that computing the integral of a non-trivial  $f$  should only affect our interpretation of the counter  $i$ . We will address this and other ways to generalize this algorithm in Sec. 4.

### 3.1 The two-minute python script

The python implementation is a straightforward translation of the pseudocode of Alg. 1. The python standard library [7] includes the package `random` with a random number generator. Two minutes of typing produce:

Listing 1: `mcint.py`: Estimating  $\pi$  in python

```

1 # get access to the random number generator functions
2 import random
3
4 # sample size
5 N = 10**6
6
7 # initialize the counters
8 total = 0
9 interior = 0
10

```

$N$	$\pi_N$	$\Delta$	$t(\text{sec})$
$10^0$	0	1	.014
$10^1$	3.6	$1.46 \times 10^{-1}$	.014
$10^2$	3.36	$6.95 \times 10^{-2}$	.014
$10^3$	3.076	$2.09 \times 10^{-2}$	.015
$10^4$	3.156	$4.59 \times 10^{-3}$	.027
$10^5$	3.14496	$1.07 \times 10^{-3}$	.144
$10^6$	3.144028	$7.75 \times 10^{-4}$	1.265
$10^7$	3.142112	$1.65 \times 10^{-4}$	12.624
$10^8$	3.14170136	$3.46 \times 10^{-5}$	130.430

Table 1: Estimating  $\pi$ : accuracy and cost of the python implementation

```

11 # integrate by sampling some number of times
12 while total < N:
13     # generate a random point
14     x = random.random()
15     y = random.random()
16     # check whether it is inside the unit quarter circle
17     if (x*x + y*y) <= 1.0: # no need to waste time computing the square root!
18         # update the interior point counter
19         interior += 1
20     # update the total number of points
21     total += 1
22
23 # print the result
24 # note that since python 3.0, division of integers yields floats
25 print('pi: %.8f' % (4 * interior / total))

```

Note that by making the determination of whether the point is inside the quadrant using the square of the distance saves us quite a large number of calls to `sqrt`. This type of consideration is ubiquitous when trying to get good performance out of Monte Carlo methods. Also, note that we did not have to convert our integer counters to floats before the computation of the area fraction, since division of integers results in floating point numbers whenever necessary, a behavior that is standard since version 3.0 of the python interpreter.

There are two obvious questions to answer: how accurate and how fast is this calculation? To answer these questions, we can perform some experiments where we vary the total number of points  $N$  and record the result and the time it took. Table 1 shows the results of a series of such experiments. Varying the sample size involves editing the script, making the corresponding change to the value of  $N$  and making another run. We could have exposed this variable to the end user by attaching it to a command line argument, but there is no reason to go through all this at this point. We will see a better way to get this kind of flexibility in Sec. 6.

As estimators of  $\pi$  go, this is not a good one: it converges rather slowly and it is rather expensive. Sample sets larger than  $10^8$  are not practical in pure python. A major consideration when using Monte Carlo methods is the quality of the random number generator. How much better can we do by switching over to a low level language, such as C++?

### 3.2 The ten-minute C++ program

The implementation in C++ is also fairly straightforward. One complication is that the random number generator in the standard C library has rather poor properties and it is not well suited for stochastic calculations; fortunately, GSL, the GNU Scientific library [8], has a large number of good generators whose properties and cost are rather well documented. Precompiled packages of GSL are available for a variety of platforms, and you can always download and compile it from its freely available source code.

Note that we are only using C++ as a better C, taking advantage of the strict type checking, the

$N$	$\pi_N$	$\Delta$	$t(\text{sec})$
$10^0$	0	1	.002
$10^1$	3.2	$1.86 \times 10^{-2}$	.002
$10^2$	3.16	$5.86 \times 10^{-3}$	.002
$10^3$	3.2	$1.86 \times 10^{-2}$	.002
$10^4$	3.1456	$1.28 \times 10^{-3}$	.004
$10^5$	3.13528	$2.01 \times 10^{-3}$	.026
$10^6$	3.140756	$2.66 \times 10^{-4}$	.230
$10^7$	3.141948	$1.13 \times 10^{-4}$	2.277
$10^8$	3.1417769	$5.86 \times 10^{-5}$	22.749
$10^9$	3.1415631	$9.41 \times 10^{-6}$	227.735

Table 2: Estimating  $\pi$ : accuracy and cost of the C++ implementation

ability to declare variables where we need them, and the prettier comment syntax. Later we will see a couple of other reasons to compile C programs with C++ compilers. For now, here is what a simple translation of the pseudocode of Alg. 1 looks like:

Listing 2: mcint.cc: Estimating  $\pi$  in C++

```

1 #include <cmath>
2 #include <iostream>
3 #include <gsl/gsl_rng.h>
4
5 int main(int, char*[]) {
6     // local variables
7     const int N = pow(10, 8);
8
9     // initialize the counters
10    int interiorPoints = 0, totalPoints = 0;
11
12    // create the random number generator
13    gsl_rng * generator = gsl_rng_alloc(gsl_rng_ranlxs2);
14
15    // integrate by sampling some number of times
16    for (int i=0; i<N; ++i) {
17        // create a random point
18        double x = gsl_rng_uniform(generator);
19        double y = gsl_rng_uniform(generator);
20        // check whether it is inside the unit quarter circle
21        if ((x*x + y*y) <= 1.0) { // no need to waste time computing the square root
22            // update the interior point counter
23            interiorPoints++;
24        }
25        // update the total number of points
26        totalPoints++;
27    }
28
29    // print the results
30    std::cout << "pi: " << 4*((double)interiorPoints)/totalPoints;
31
32    return 0;
33 }

```

The accuracy and cost of the C++ implementation is shown in Table 2. It is very clear that the C++ implementation is almost an order of magnitude faster. However, the cost of making the necessary changes to the value of  $N$  is a little higher for the C++ program, since the cycle involves the additional steps of compiling and linking. Again, a flexible program would expose this variable to the end user; we will address this issue in Sec. 6.

We have chosen the RANLUX random number generator, one the most costly but well behaved generators in the library. Even so, the C++ implementation is six to seven times faster. Since the python generator `random` is implemented in C, the difference in running times is mostly due to the cost of setting up the function call in the two languages. For many computations, the extra flexibility

that python has to offer is worth the performance hit; for others it is unacceptable. However, as we will see later, a little care, some common sense, and some careful profiling will make it possible to have the best of both worlds.

## 4 The hunt for objects

The two implementations in Sec. 3.1 and Sec. 3.2 are excellent examples of a very common occurrence in scientific programming. They are simple and to the point, and solve the problem as originally stated rather well. This kind of simplicity has many advantages: the code is easy to understand, changing it does not require understanding complicated interactions between unfamiliar pieces of code, and it is likely to have very good performance characteristics. Unfortunately, even minor attempts at generalization will require extensive surgery. That's not to say that the time spent in constructing these prototypes is wasted: it takes foresight that comes with extensive experience to be able to see the general solution without visiting a few dead ends. As they are, our two implementations will serve as benchmarks against which we will judge the next evolutionary steps, and in particular their performance so that we can gauge the flexibility trade-off.

There is a variety of approaches one could take to generalize our naïve implementation. One particular paradigm, object oriented programming [9], has proved effective and is currently very popular. Developers raised on procedural programming, i.e. the kind of programming you do in C or FORTRAN, typically face a bit of a challenge as they try to shed inappropriate habits. Those who have managed to climb the learning curve report an increased awareness of the role of process in software engineering, and overall improvement in the quality of code they write. However, typical goals such as reuse and flexibility are hard to attain regardless of the programming paradigm. On one hand, the problem itself has to cooperate by belonging to the class of problems that are reducible to fairly orthogonal and well characterized pieces. On the other, one has to be able to identify these pieces and turn them into high quality, maintainable code. Much of the time, these form a string of low probability events that require a significant intellectual investment. Fortunately, the example at hand is rather well behaved in this regard.

### 4.1 Encapsulating the random number generator

The first candidate for generalization is the random number generator. We should be able to choose among many different ones so that we can compare them. In order to take advantage of the object oriented machinery in languages like python and C++, let's work towards the specification of `Random`, an abstract base that will encapsulate random number generators. Abstract base classes are typically used to anchor class hierarchies. They are used to define the interface to which derived classes should adhere, but they themselves are not used directly, so they are not supposed to have any data members and all of their methods should generate exceptions. We will use `Random` to derive classes that hide the implementation particulars and conform to a standard interface that we will specify. This idea is that when we run into another random number generator that we would like to use, the effort necessary to incorporate it into our program is *isolated* to a small subset of the overall code, so that the likelihood of introducing a bug into the code is minimized. Ideally, we should need to do little more than create a new class that derives from `Random` and implement the standard interface using the new generator.

Having identified such a candidate is half the battle; but now a universe of possibilities opens up. What should its interface be, i.e. what is the list of its class methods? What arguments should these methods accept? Should there be any data members associated with instances of this and its derived classes? A good way to navigate through the possibilities is to do as little as possible at every step. Rather than trying to anticipate all the possible uses of `Random` and its derivatives, we can focus on endowing it with just enough interface to solve our problem. We need a sample  $X_N$  of  $N$  random points  $x$  that fall inside a box  $B$ , so let's create a method `point` that takes a suitable representation of  $B$  and returns such a point:

Listing 3: The base class for random number generators

```

1 class Random(object):
2     """
3     Base class for random number generators
4     """
5
6     def point(self, box):
7         raise NotImplementedError("please override 'point'")

```

Since `Random` is abstract, we have no non-trivial implementation to provide; instead `point` raises an exception so that developers of derived classes will be reminded to provide one.

At this stage of the design, we don't have any non-trivial constraints to satisfy. This is both a blessing and a curse. On one hand, we can pretty much dictate what the class hierarchy will look like since we are building it from scratch. On the other, there is no structure imposed from the outside that will help us steer the design in some particular direction. Figuring out which is a blessing and which a curse is left as an exercise. In these situations, experience strongly suggests that simplicity and laziness are the right guidelines: defer as many of the decisions to derived classes as is possible, keep the number of methods to a minimum, and make the interface as simple and generic as possible. Unanticipated complexities, such as creating an actual model of some process, or making the code perform at an acceptable level, will force us to rethink much of the design.

Performance considerations are particularly difficult to anticipate in object oriented applications. The usual maxims about picking algorithms with low computational complexity and being careful about the internal structure of loops only capture a fraction of the problem. Object oriented programs re-shuffle complexity into the interactions of class instances. Developers are notoriously wrong about what to optimize even in the simplest procedural routines; it is considerably more difficult to anticipate the performance characteristics of an object oriented program. The only way to get a clear understanding of which section of the application require further optimization efforts is to use a profiler, a tool dedicated to analyzing code performance. Despite all the advances in software engineering technology, it is still very hard to profile a design without an actual implementation. This deals a rather strong blow to the proponents of formal design practices and makes agile programming a much more attractive methodology for scientific programming.

One thing is clear: the performance cost of function calls has the potential to dominate the execution profile of a code in an insidious way. The program may be too slow, yet the profiler does not uncover any obvious candidates for optimization. This is a rather unfortunate side effect of object oriented programming and is one of the reasons that many scientific programmers stay with the procedural languages or resort to C++ and templates. Still, much of the code in modern scientific application deals with the user and the computational environment; it is these parts that stand to benefit the most from the complexity management that object oriented techniques bring to the table.

Returning to the problem at hand, we can now derive a class `WichmannHill` that encapsulates the random number generator from the standard python library that we used in Sec. 3.1. We require that `box` has enough structure so that we can deduce the dimensionality of the requested point, as well as build the set of transformations that map from the unit line interval to the corresponding side of the box. The vector diagonal of the bounding box fits this bill perfectly: we specify that `box` is to be a pair of  $d$ -tuples specifying the coordinates of the point of the box closest to the origin and the point diagonally across from it. For example, the unit square on the plane might be represented by `[[0,0], [1,1]]`, while the unit box in three dimensions would be given by `[[0,0,0], [1,1,1]]`. We only use iterators to access `box`, so any iterable container with the structure shown will do.

Listing 4: Encapsulating the standard python generator

```

1 import random
2 from Random import Random
3
4 class WichmannHill(Random):
5
6     def point(self, box):
7         # unpack the bounding box
8         tail, head = box

```

```

9     intervals = zip(tail, head)
10
11     # build the point p by calling random the right number of times
12     p = [ random.uniform(left, right) for left,right in intervals ]
13
14     return p

```

We have unpacked the two corners of the bounding box in the assignment on line 8. The `zip` on line 9 returns a list of tuples, with each tuple containing one element from each of the arguments. It builds a list of the intervals along each axis, over which we iterate in the list comprehension on line 12 in order to extract the arguments to `uniform`.

The class `Random` and its derivatives should be generally useful to us and are good candidates for reuse. The interface should be broadened a bit, by adding a method to seed the random number generator with some initial value, something that most generators support. Another consideration is that we will be calling this method a large number of times during the integration, so perhaps we will have to modify the class interface after we evaluate the performance of this implementation in Sec. 4.3.

### Still to do

- ☞ references for object oriented programming (meyer, eiffel)
- ☞ references for "formal methods" and agile programming
- ☞ show example of how to encapsulate the GSL random number generator as well

## 4.2 Representing the region of integration

The next step in the factorization of the simple implementations in Sec. 3 is to build a class hierarchy that represents the region of integration. Designing with simplicity in mind would suggest a class `Shape` with a single method:

Listing 5: The base class for the representation of the region of integration

```

1 class Shape(object):
2
3     def contains(self, point):
4         raise NotImplementedError("please override 'contains'")

```

The representation of our region of integration involves two steps: providing an implementation for `contains` and endowing instances of `Circle` with location and size. These two attributes will be represented as data members of the class that are provided at construction time, and so they become arguments of the constructor of the class.

Listing 6: The class `Circle`

```

1 from Shape import Shape
2
3 class Circle(Shape):
4
5     def contains(self, point):
6         r2 = self.radius**2
7         x0, y0 = self.center
8         x, y = point
9         dx = x - x0
10        dy = y - y0
11
12        if dx*dx + dy*dy > r2:
13            return False
14
15        return True
16
17    def __init__(self, radius=1.0, center=(0.0, 0.0)):
18        self.radius = radius
19        self.center = center
20        return

```

$N$	C++ $t(\text{sec})$	python $t(\text{sec})$	naïve OO $t(\text{sec})$	containers $t(\text{sec})$	generators $t(\text{sec})$
$10^0$	.002	.014	.014	.014	.014
$10^1$	.002	.014	.014	.014	.014
$10^2$	.002	.014	.014	.015	.014
$10^3$	.002	.015	.020	.019	.018
$10^4$	.004	.027	.078	.063	.053
$10^5$	.026	.144	.625	.504	.401
$10^6$	.230	1.265	6.242	5.925	3.780
$10^7$	2.277	12.624	61.583	188.318	38.242
$10^8$	22.749	130.430			
$10^9$	227.735				

Table 3: Comparison of the cost of the various implementations

For convenience, the constructor on line 17 accepts default values for the center location and radius. It is best to keep such practices to a minimum because the implicit passing of information between caller and callee makes the code harder to read and understand, and they expose users of the class `Circle` to the risk that the default values of the arguments may change in the future. In this case there is a *natural* choice for these values, a unit circle centered at the origin, and so the risk is reduced.

Note that we were careful to avoid injecting the context of our exercise into the design of `Shape` and `Circle`. They appear rather generic and their intended use to perform Monte Carlo integration is not explicit. It is natural to ask a `Shape` whether it contains a given point, regardless of the context for the question. It is not always possible to avoid this type of pollution, but careful refactoring of the initial design often helps.

### 4.3 Creating the driver

We are now ready to put the object oriented solution together and see how it performs.

Listing 7: The driver for the objected oriented solution

```

1 # get access to the class definitions
2 from Circle import Circle
3 from WichmannHill import WichmannHill
4
5 # initialize the integration
6 N = 10**6
7 box = [(0,0), (1,1)]
8 generator = WichmannHill()
9 circle = Circle(center=(0,0), radius=1)
10
11 # initialize our counters
12 total = 0
13 interior = 0
14
15 # integrate
16 while total < N:
17     point = generator.point(box)
18     if circle.contains(point):
19         interior += 1
20     total += 1
21
22 print "pi:", 4*float(interior)/total

```

Before we break open the champagne in celebration of the elegance and simplicity of this implementation, we have the unfortunate matter of performance to contend with. A look at the third column of Table 3 should convince you rather quickly that we have a lot of work still left to do. The object

solution is an order of magnitude slower than our previous python implementation, and almost two orders of magnitude slower than the C++ implementation.

## Still to do

☞ add sentence excusing not going past  $10^8$

## 4.4 Improving performance

The lackluster performance numbers are not very hard to understand. Taking a closer look at Listing 7 we see that the body of the loop contains two function calls, one to `generator.random` and another to `circle.contains`. Each of these calls is going to get executed  $N$  times and we will have to pay the cost associated with calling python functions every time. Furthermore, if you look at the body of these two functions in Listing 4 and Listing 6, you will find quite a bit of code associated with initializing the local variables. That's another cost that we are paying once for each sample point. Looks simple enough, but it quickly adds up.

### 4.4.1 Containers: trading memory for speed

Perhaps we can amortize these costs over the sample set. One approach would be to call the random number generator once and have it return a container with all the points in the sample set. We would then pass this array to the `shape`, which would return a *mask*, i.e. an array of `True-False` values indicating whether the corresponding point was interior or not. Keep in mind that it is not acceptable to have `contains` just count the number of interior points for us, for a couple of reasons. First, this capability pollutes the design of `Shape` derivatives and specializes them too much for reuse. Second, we will eventually need to know which points are interior to the `Shape` so we can evaluate our integrand  $f$  there.

Turning to the implementation, `Random` would look like

Listing 8: An improved base class for random number generators

```
1 class Random(object):
2
3     def points(self, n, box):
4         raise NotImplementedError("please override 'points'")
```

where `n` is the number of points we want, and `box` is the bounding box representation as before. The new implementation of `WichmannHill` is

Listing 9: Generating  $N$  points at a time

```
1 import random
2 from Random import Random
3
4 class WichmannHill(Random):
5
6     def points(self, n, box):
7         # create the container for the sample
8         sample = []
9
10        # unpack the bounding box
11        tail, head = box
12        intervals = zip(tail, head)
13
14        # loop over the sample size
15        while n > 0:
16            # build the point p by calling random the right number of times
17            p = [ random.uniform(left, right) for left, right in intervals ]
18            # store the point
19            sample.append(p)
20            # update the counter
21            n -= 1
22
23        return sample
```

The trade off, of course, is that now we must pay the cost of managing a container for these points. Since  $n$  is potentially a large number we must be very careful inside the `while` loop. In particular it is important to use  $\mathcal{O}(1)$  operations while building the sample container. Fortunately, `append` is very well behaved in this regard: python lists are doubly-chained and the interpreter can add elements at the tail in constant time.

The class `Shape` doesn't require any change. We will merely reinterpret the dummy argument as a container of points:

Listing 10: The base class `Shape` requires little change

```

1 class Shape(object):
2
3     def contains(self, points):
4         raise NotImplementedError("please override 'contains'")

```

The implementation of `Circle` is a bit more complicated than Listing 5.

Listing 11: An implementation of class `Circle` with containers

```

1 from Shape import Shape
2
3 class Circle(Shape):
4
5     def contains(self, points):
6         mask = []
7         r2 = self.radius**2
8         x0, y0 = self.center
9
10        for x,y in points:
11            dx = x - x0
12            dy = y - y0
13            if dx*dx + dy*dy > r2:
14                mask.append(False)
15            else:
16                mask.append(True)
17
18        return mask
19
20    def __init__(self, radius=1.0, center=(0.0, 0.0)):
21        self.radius = radius
22        self.center = center
23        return

```

Again, we have to pay for the cost of managing `mask`, and the same considerations apply when we have to manipulate its entries.

The driver for the container based solution is rather easy to put together.

Listing 12: The driver for the container based solution

```

1 # get access to the class definitions
2 from Circle import Circle
3 from WichmannHill import WichmannHill
4
5 # initialize the integration
6 total = 10**7
7 box = [(0,0), (1,1)]
8 generator = WichmannHill()
9 circle = Circle(center=(0,0), radius=1)
10
11 # create a sample
12 sample = generator.points(total, box)
13 # count the interior points
14 interior = len(filter(None, circle.contains(sample)))
15
16 print "pi:", 4*float(interior)/total

```

A striking difference with Listing 7, and all other implementations of the integration driver, is the absence of counters. Instead, we have used `filter` that accepts a list and returns another with the non-True elements removed, at which point counting the interior elements is just a matter of computing the length of the returned list.

## Still to do

☞ compare with the performance in Sec. 3.1 and Sec. 4.3

☞ talk about the table, and in particular about the  $10^7$  explosion

### 4.4.2 Generators: the best of both worlds

The container based solution has a certain elegance: the creation of our sample set and its use are separated into two distinct phases of the computation, the overhead associated with setting up the calls to the random number generator and the shape instance are amortized over the sample set, and we can use functions that operate efficiently on lists to perform parts of the calculation. The major drawback is having to manage the sample container, whose implications include some overhead in storing and retrieving the contents, and the potential that we run out of memory for sufficiently large samples.

Fortunately, there is a way to retain the elegance of containers without paying the memory cost by using *generators*, an extremely powerful programming construct borrowed from the language Icon[10] that is now available in python.<sup>2</sup> Hopefully, we will be able to avoid any confusion between random number generators and the programming construct.

Generators are functions that behave like iterators without necessarily having to manage actual storage for a container. You can think of them as iterator generators, hence the catchy name. What you actually get when you call a generator involves elaborate descriptions of some implementation details; for the purpose of this discussion, you can think of the result as something that can be used just like a normal iterator. Rather than getting lost in an abstract description, let's revisit `WichmannHill`. The improvements we will discuss in this section are focused entirely on implementation details; the interface as specified in our abstract classes will remain intact. Hence, we will derive our improved class from the version of `Random` shown in Listing 8. The new implementation of `WichmannHill` is shown in Listing 13.

Listing 13: The generator based implementation

```
1 import random
2 from Random import Random
3
4 class WichmannHill(Random):
5
6     def points(self, n, box):
7         # unpack the bounding box
8         tail, head = box
9         intervals = zip(tail, head)
10
11         # loop over the sample size
12         while n > 0:
13             p = [ random.uniform(left, right) for left, right in intervals ]
14             yield p
15
16             n -= 1
17
18         return
```

Almost everything looks the same as in Listing 9, except for the new keyword `yield` on line 14 and the `return` without a value on line 18. Indeed, we unpack the arguments to `point` as we did before, and proceed to loop over the desired sample size, generating random points. Rather than storing `p`, we `yield` it to our caller. This statement suspends the execution of `points` but saves everything that we have done so far, including the values of all the local variables such as the loop counter `n`. The caller doesn't actually get `p`, but an iterator object that executes this code and returns `p` when asked for its current value, and will resume the evaluation of `points` when asked to advance via its `next` method at whatever statement follows the `yield` statement on line 14 that suspended it. In our case it means that the counter will get decremented and the loop will execute another iteration until the next time the `yield` statement is reached. This process continues until

<sup>2</sup>Generators were first seen in version 2.2 and turned into a core part of the language in version 2.3

the loop is exhausted, at which point we run into the `return` statement on line 18. According to generator rules, `return` causes a `StopIteration` exception to be raised, signalling to the caller that we are done generating values from our virtual container.

If this is making your head hurt, it's all right. It will all make sense pretty soon. Later I will make your head *explode* when we touch on *metaclasses*.

Let's see what happens to `Shape`:

Listing 14: The class `Circle`

```
1 from Shape import Shape
2
3 class Circle(Shape):
4
5     def contains(self, points):
6         r2 = self.radius**2
7         x0, y0 = self.center
8
9         for x, y in points:
10            dx = x - x0
11            dy = y - y0
12
13            if dx*dx + dy*dy > r2:
14                yield False
15            else:
16                yield True
17
18        return
19
20    def __init__(self, radius=1.0, center=(0.0, 0.0)):
21        self.radius = radius
22        self.center = center
23        return
```

The method `contains` still accepts some kind of iterable object `points` and goes about the business of deciding whether the contents are interior or not. We set up a loop over `points` on line 9, but rather than storing the result in `mask` as we did in Listing 11, we either `yield False` on line 14, or `True` on line 16. When our loops exhausts `points` we reach the `return` statement on line 18, which raises a `StopIteration` exception and terminates our generator.

All this comes together in driver. It is identical to the container based solution in Listing 12, so we won't repeat it here. See if you can follow how looping over the virtual contents is driving the generators to produce the associated values without paying the computational expense to set up the function or manage a container for the intermediate values.

The generator solution performs rather well. We are still a bit off the simple python implementation and a whole order of magnitude slower than C++, but such is the price for flexibility.

#### 4.4.3 Alternative containers

The container based solution suffers from two drawbacks: the high overhead associated with the function calls necessary to manage the container, and the large memory footprint of python containers. Generators enabled us to do away with the temporary storage but do little to minimize the total number of function calls in the body of the loop. Both of these solutions are substantially slower than the essentially inlined implementation of Listing 1. Is it possible to improve performance by using better containers than python provides?

At first glance, this looks like a viable strategy. We know the size of the point container rather early in the program. The points themselves are just pairs of floating point numbers, so perhaps we can allocate something like a C array of the right size and hand that to our random number generator. It's going to be hard to take advantage of the python random number generator this way, but we can easily access the generators in GSL, so that's not a big problem. Once our C container comes back filled with random points, we should be able to hand that to a C routine that will count the interior points. The whole thing can be orchestrated from python by writing wrappers for the low level routines. We can even recover the object oriented structure we have been building by using these python wrapped C routines as the implementations of the relevant methods of our classes.

At this point you might be very strongly tempted to write your own array class as a thin wrapper on top of C arrays, or even C++ containers. Resist. This thought has occurred to many a scientific programmer, and the software landscape is littered with such casually discarded dysfunctional software. Writing array classes correctly turns out to be much harder than it appears, mostly due to the fact that containers of real numbers have extremely rich algebraic structures. You either respect the semantics or you are contributing to the trash heap.

Fortunately, a few of those that didn't get—or didn't pay any attention to—such advice have tackled this problem and got it right. The package `numpy` [11] is such an example. We leave it as an exercise to build and profile a `numpy`-based solution that takes advantage of the compactness of representation, speed and convenience of the array implementation.

How do all these implementations fit together? Do we have to throw away the generators because we moved the representation to a `numpy` container? Have we locked ourselves out of using the python built in random number generator because we cannot feed it a C array? Thanks to the fact that interface, rather than type, is what python checks, we can make all these solutions inter-operate. The generator `Circle.contains` in Listing 14 does not know or care whether the point container it is iterating over is real or virtual, a python list or a `numpy` array. All that matters is that `points` is iterable and that each iteration produces a pair of floats. This lets us keep the object oriented structure and explore various strategies that trade off between complexity, memory and performance. Better yet, in Sec. 5 and Sec. 6 I will show you how you can let your users assume final responsibility by postponing making these decisions for as long as possible.

## 4.5 Reuse: improving and adjusting the interface

Listing 15: An improved base class for random number generators

```

1 class Random(object):
2
3     def seed(self, n):
4         raise NotImplementedError("please override 'seed'")
5
6     def point(self, box):
7         raise NotImplementedError("please override 'point'")
8
9     def points(self, box, n):
10        raise NotImplementedError("please override 'points'")

```

### Still to do

☞ fitting this solution in an existing design: introduce and explain Adaptors

## 4.6 Functors

### Still to do

☞ re-introduce the integrand  $f$  to the mix

☞ explain function-as-object

☞ show the complete driver

- 5 Turning classes into components
- 6 The application framework
- 7 Hosting the application on the web
- 8 Building a graphical interface
- 9 Enabling remote access
- 10 Epilogue

## 11 References

- [1] J. M. Hammersley. Monte Carlo methods for solving multivariable problems. *Ann. New York Acad. Sci.*, 86:844–874, 1960.
- [2] C. W. Ueberhuber. *Numerical Computation 2: Methods, Software, and Analysis*, chapter on Monte Carlo Techniques, pages 124–125 and 132–138. Springer-Verlag, 1997.
- [3] S. Weinzierl. Introduction to Monte Carlo methods. 2000. URL <http://arxiv.org/abs/hep-ph/0006269>.
- [4] G. P. LePage. A new algorithm from adaptive mutlidimensional integration. *Journal of Computational Physics*, 27:192–203, 1978.
- [5] G. P. LePage. VEGAS: An adaptive multidimensional integration program. preprint CLNS 80-447, Cornell University, March 1980.
- [6] W. H. Press and G. H. Farrar. Recursive stratified sampling for multidimensional Monte Carlo integration. *Computers in Physics*, 4:190–195, 1990.
- [7] Python documentation. <http://docs.python.org>.
- [8] The GNU scientific library. <http://www.gnu.org/software/gsl>.
- [9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
- [10] R. E. Griswold and M. T. Griswold. *The ICON programming language*. Peer-to-Peer Communications, third edition, 1996.
- [11] numpy. <http://numpy.scipy.org>.