

Lecture Notes on Quasi-Monte Carlos

Andrew Larkoski

November 9, 2016

Happy Thanksgiving tomorrow! In this second day of this shortened week, I want to discuss quasi-Monte Carlo methods for integration. Let's recall why Monte Carlos were virtuous for integration. In low numbers of dimensions, deterministic integration algorithms like Simpson's method are the most accurate for the fewest number of sample points. However, the timing to evaluate Simpson's method integral scales exponentially in the number of dimensions of the integral:

$$T(N) = \mathcal{O}(N^D), \quad (1)$$

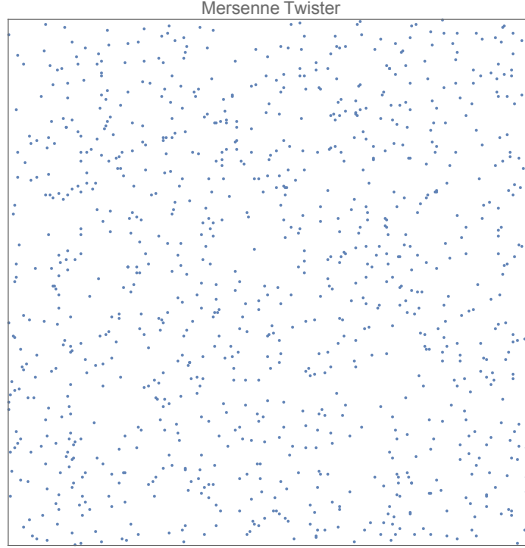
where N is the number of grid points in one dimension and D is the number of dimensions. This is a manifestation of the "curse of dimensionality": for a given accuracy, we must increase the number of grid points by a huge amount as the number of dimensions increases. Thus, it becomes increasingly difficult to use Simpson's method for high dimensional integrals.

By contrast, Monte Carlo integration, which randomly samples the integrand, is much more well-behaved in high dimensions. The accuracy of Monte Carlo scales like

$$|I - I_{\text{MC}}^N| = \mathcal{O}(N^{-1/2}), \quad (2)$$

where I is the true value of the integral, I_{MC}^N is the Monte Carlo approximation with N dart throws. This is independent of the dimension; the accuracy always scales as $N^{-1/2}$. During our week on Monte Carlos, we discussed where this came from as effectively an implementation of Poisson statistics.

For all of its virtues, however, Monte Carlos still have some issues. Consider the distribution of Monte Carlo points (just uniform pseudorandom numbers) over the unit square:



Note that there are regions of high density (lots of points) and regions of low density (few points). This is sub-optimal for integral evaluation.

In the high density regions, the integrand is sampled very well, and so the integral over high density regions is well-approximated. However, in the low density regions the integrand is hardly sampled at all, and so the integral is totally unknown in that region. For optimal calculation of the integral, we would want better, more evenly spaced, points.

So, for optimal integral evaluation, we want to combine aspects of Simpson's method (or the like) with Monte Carlo. We want to:

- Avoid the curse of dimensionality like Monte Carlo,
- Even point spacing like Simpson.

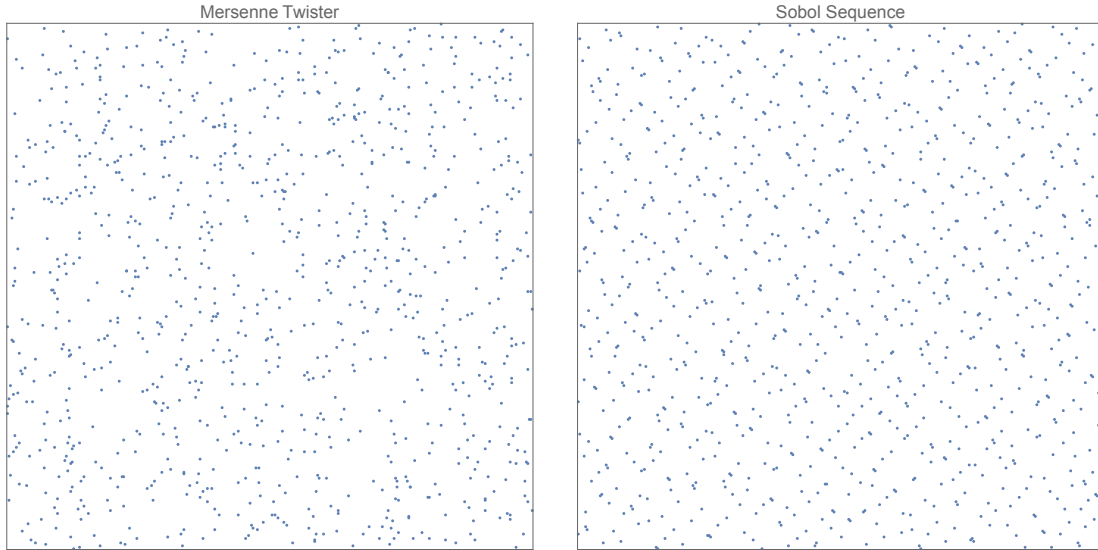
If we can accomplish both of these things, then we would have an even better integrator than Monte Carlo.

This was the goal of Russian mathematician Ilya Sobol: to find the sequence of points $\{x_i\}_{i=1}^N$ such that

$$\lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(x_i) = \int_0^1 f(x) dx \quad (3)$$

converges as fast as possible for any integrable function $f(x)$ on $x \in [0, 1]$. (This domain is representative: it could be any domain.) One sequence that does this is now named after Sobol, called the Sobol sequence. The Sobol sequence is a “low-discrepancy” sequence (its points don't vary wildly like (pseudo)random numbers) and it is a deterministic sequence (unlike random numbers, in principle).

To visualize the difference between Sobol sequence numbers and (pseudo)random numbers, we can use Mathematica to plot 1000 Sobol numbers and pseudorandom numbers (with the Mersenne Twister algorithm). Here are the comparison plots:



Sobol sequence numbers have a much more regular pattern than pseudorandom numbers. The Sobol numbers also clump up less than random numbers.

This is advantageous for integration. It has been shown that the error on an integral using quasi-Monte Carlo (i.e., Monte Carlo method with Sobol sequence points) is:

$$|I - I_{\text{qMC}}^N| = \mathcal{O}\left(\frac{\log^D N}{N}\right), \quad (4)$$

where N is the number of points and D is the dimension. Unlike strict Monte Carlo, this does depend on the number of dimensions, but much weaker than Simpson's method. Also, for sufficiently large N , quasi-Monte Carlo always beats the accuracy of Monte Carlo, because there exists a minimum N_{qMC} such that

$$\frac{\log^D N_{\text{qMC}}}{N_{\text{qMC}}} < N_{\text{qMC}}^{-1/2}. \quad (5)$$

While the advantages of quasi-Monte Carlo integration are only large in the $N \rightarrow \infty$ limit, quasi-Monte Carlo methods are standard for high-dimensional performance integration evaluation. First, within Mathematica, one can specify how to perform a numerical integral. The `NIntegrate` function in Mathematica performs numerical integration using a huge variety of options: among them, Simpson, Monte Carlo, quasi-Monte Carlo, but also many more. In particular, something we didn't discuss, `NIntegrate` adapts the integration grid, depending on the derivative of the function in a given region. To call `NIntegrate` in its default settings use:

```
NIntegrate[f[x], {x, a, b}]
```

For multi-dimensional integrals, `NIntegrate` starts on the outside and works in:

```
NIntegrate[f[x,y],{x,a1,b1},{y,a2,b2}]
```

First the y integral is done, then the x integral. To call the quasi-Monte Carlo method requires just adding a call:

```
NIntegrate[f[x],{x,a,b},Method->"QuasiMonteCarlo"]
```

In Mathematica, there are 22 different integration methods.

However, perhaps the most powerful numerical integration program is called CUBA, and you can download it at <http://www.feynarts.de/cuba/>. CUBA has been written in C++, Fortran, and as a plugin to Mathematica. I'll discuss the Mathematica plugin.

Once you've downloaded CUBA, unzip/untar the compressed file. Then, in the folder, open up a command line and enter

```
> ./configure MCFLAGS=-st  
> make
```

This compiles the program. To run it, we just open Mathematica and link to one of the integrators in CUBA. There are four built-in numerical integrators in CUBA: Suave, Cuhre, Divonne, and Vegas. I'll just discuss Vegas and its use.

Vegas is an adaptive quasi-Monte Carlo integrator (adaptive = more dart throws where the function is very steep). To use it, we just link using the command:

```
Install["Vegas"]
```

(This needs to point to the directory where CUBA was installed.) To run Vegas, it is just like `NIntegrate`, e.g.,

```
Vegas[f[x],{x,a,b}]
```

Let's see how this works; let's integrate the function $f(x) = x$ on $x \in [0, 1]$:

```
Vegas[x,{x,0,1}]
```

The output is shown in Fig. 1. Look at the output: Vegas tells us how many integrand evaluations were performed, the estimated error, and the χ^2 (ignored here). The evaluation terminates once the estimated error is less than 10^{-3} times the estimated integral.

What if we integrate something more complicated, like

$$f(x,y) = e^{-x/y^2}, \quad (6)$$

on $x, y \in [0, 1]$? Well, if we run Vegas now, we find the output in Fig. 2. We don't reach 10^{-3} accuracy after 50,000 evaluations! So, we need to up the number of Sobol numbers to use. We can do this using `MaxPoints`:

```
Vegas[Exp[-x/y/y],{x,0,1},{y,0,1},MaxPoints->200000]
```

```

In[299]:= Vegas[x, {x, 0, 1}]

Iteration 1: 1000 integrand evaluations so far
[1] 0.499946 +- 0.00912348      chisq 0 (0 df)

Iteration 2: 2500 integrand evaluations so far
[1] 0.499802 +- 0.00306699      chisq 0.000281156 (1 df)

Iteration 3: 4500 integrand evaluations so far
[1] 0.499868 +- 0.00121244      chisq 0.000825048 (2 df)

Iteration 4: 7000 integrand evaluations so far
[1] 0.500404 +- 0.000545954      chisq 0.246156 (3 df)

Iteration 5: 10000 integrand evaluations so far
[1] 0.500016 +- 0.000356378      chisq 1.12742 (4 df)

... Vegas: Needed 10000 function evaluations.

Out[299]= {{0.500016, 0.000356378, 0.110103}}

```

Figure 1: Vegas output.

With 200,000 points, we are able to reach the desired accuracy. What if we want to increase the accuracy? We can do this with `PrecisionGoal`. Say, we want 10^{-4} accuracy rather than the default 10^{-3} . We then run

```
Vegas[Exp[-x/y/y], {x, 0, 1}, {y, 0, 1}, PrecisionGoal->4, MaxPoints->20000000]
```

We also crank up the number of points to 20,000,000 to ensure that the accuracy can be reached.

Cool!

```

In[302]:= Vegas[ $e^{-\frac{x}{y^2}}$ , {x, 0, 1}, {y, 0, 1}]

Iteration 1: 1000 integrand evaluations so far
[1] 0.269798 +- 0.0092531      chisq 0 (0 df)

Iteration 2: 2500 integrand evaluations so far
[1] 0.270486 +- 0.00381476    chisq 0.00666425 (1 df)

Iteration 3: 4500 integrand evaluations so far
[1] 0.269927 +- 0.00227444    chisq 0.0400652 (2 df)

Iteration 4: 7000 integrand evaluations so far
[1] 0.269892 +- 0.00162266    chisq 0.040539 (3 df)

Iteration 5: 10000 integrand evaluations so far
[1] 0.269874 +- 0.00126528    chisq 0.0408502 (4 df)

Iteration 6: 13500 integrand evaluations so far
[1] 0.27004 +- 0.00105696     chisq 0.0980477 (5 df)

Iteration 7: 17500 integrand evaluations so far
[1] 0.270038 +- 0.000901885   chisq 0.098075 (6 df)

Iteration 8: 22000 integrand evaluations so far
[1] 0.270113 +- 0.000799083   chisq 0.130769 (7 df)

Iteration 9: 27000 integrand evaluations so far
[1] 0.270098 +- 0.000712561   chisq 0.13249 (8 df)

Iteration 10: 32500 integrand evaluations so far
[1] 0.27006 +- 0.000641712    chisq 0.147531 (9 df)

Iteration 11: 38500 integrand evaluations so far
[1] 0.270076 +- 0.000585664   chisq 0.151226 (10 df)

Iteration 12: 45000 integrand evaluations so far
[1] 0.270075 +- 0.000539497   chisq 0.151272 (11 df)

Iteration 13: 52000 integrand evaluations so far
[1] 0.270049 +- 0.000498944   chisq 0.16625 (12 df)

*** Vegas: Desired accuracy was not reached within 52000 function evaluations.

Out[302]:= {{0.270049, 0.000498944,  $4.26707 \times 10^{-10}$ }}

```

Figure 2: Vegas output.