http://www.math.nyu.edu/faculty/goodman/teaching/MonteCarlo20/

Class notes: Monte Carlo methods
Week 1, Introduction, error bars

Jonathan Goodman

September 8, 2020

1 About the course

This is a course on Monte Carlo methods, offered from an applied math point of view. I assumes that the student is familiar with probability at the level of the Courant class *Basic Probability.* I also assume a high level of "mathematical maturity". This means that the student has been exposed to much graduate level applied math, including linear algebra, multi-variate calculus, and numerical computing at the level of the Courant class *Scientific Computing.* For example, Section 2 refers to Stirling's approximation and the trapezoid rule for numerical integration. The student either should know these or be able to understand an explanation she or he finds on the web. Mathematical maturity also refers to some practical intuition and experience digesting mathematical explanations. For example, if the time needed to run an algorithm grows exponentially with n, then the algorithm will not run in practice with large n.

The assignments will call for programming in Python 3.X. Students who have done scientific programming in other languages (C, C++, Matlab, R, FORTRAN) should be able to pick up Python 3. Scientific programming is harder and requires more expertise than other programming. For example, one must be aware that floating point arithmetic is not exact and have some appreciation or experience with the size of roundoff error.

The classes and the exercises are designed not only to teach Monte Carlo methods, but to expose you to aspects of modern applied mathematics more broadly. For this, we take the opportunity to explain a little about the backgrounds of real applications of Monte Carlo. You may have to do a little background reading to appreciate some of the specific applications. This is part of your training in applied mathematics. These weekly notes are intended to resemble a rambling lecture more than a carefully crafted reference text. You may already have noticed that sometimes the reader is called "you" and sometimes "the reader" or "the student".

Much of the class will be conducted online because of COVID-19. The material for the week will be posted a few days before the class meeting time. This will take the form of written notes and possibly some video. You should read or watch and do a quick online quiz. The notes and video will be on the public site (where these notes are), but the quiz will be on the NYU Classes site. More serious understanding will be tested in the assignments. As I am typing this, I do not know how much in-person contact will be possible or desirable. I will make myself available in online "office hours". The zoom link is on the NYU Classes page for the course. If you aren't registered and want to "attend" office hours, email me for an invitation. I hope you will find a way to share written mathematical reasoning online. By this I mean some way to hand write formulas and diagrams so that I or others can see. This can be done using a camera connected to the computer (my approach) or by screen sharing a device with a writing stylus such as an Apple ipad or a Microsoft Slate.

The course has a final project that students should do in groups of one (individual) to maybe 4. Each group will give a presentation during finals week and prepare a document that explains the project. The project is a major part of the class. You may choose a topic related to Monte Carlo depending on your talents and interests. Starting Week 4, I will suggest some topics and start organizing the groups.

The course grade will be graded based on frequent assignments and a final project. The assignments involve programming and theoretical work. At the outset, I intend to give 65% weight to assignments, 5% weight to weekly quizzes, and 30% weight to the final project. I plan to use the following grading scale:

- A for excellent work (must be earned)
- A- for high quality but not excellent work
- B+ for good work, possibly with some reservations
- B for students who possibly struggled but made a good faith effort
- Less than B for students who made less than a good faith effort. Like A, this must be earned.

My experience is that almost all students will get one of the standard grades A, A- B+, B, and in roughly equal proportions. Grades are not competitive, and I would love to skew toward the higher grades. A few people will earn substandard grades. Please contact me as soon as you find yourself slipping. We probably can work something out. The class should be a learning experience, not a stressful one. Feel free to email me about this at any time.

2 Introduction to Monte Carlo

The term *Monte Carlo methods* refers to using computer generated random variables in computing. Conceptually speaking, the easy part is creating algorithms to simulate a specified random process or random variable. I call this *direct simulation*. This course will discuss two related topics: How accurate is a simulation likely to be? Can you change the rules to do a different simulation that produces the same or nearly the same answer with less error?

To correct the first paragraph, some Monte Carlo methods are applied to problems that do not have probability in their statements. One sub-field of Monte Carlo, called *quantum Monte Carlo*, uses Monte Carlo methods to calculate the electronic structure of atoms and molecules. Monte Carlo methods also have been used to investigate the distribution of prime numbers.

A more insightful definition is that *Monte Carlo methods* are computational methods that use random numbers to compute quantities that are not random. This definition comes from Malvin Kalos. Suppose that X is a random variable and you want to know $A = \mathrm{E}[X]$. The *direct* approach would be to estimate A by taking the average of a set of samples of X. This is described in Subsection 2.2. This course describes *indirect* methods that estimate A more accurately or faster than the the direct method. Week 2 describes *importance sampling*, which is one such indirect method. The insight of the Kalos definition is that you are interested in X only as a way to estimate A. This means there may better ways to estimate A, algorithms that do not use X.

The term *Monte Carlo* was first used by people at Los Alamos National Laboratory in New Mexico. They had the idea to express certain high dimensional integrals as the expected values of random variables and then simulate the random variables using a random number generator. The name comes from the city of Monte Carlo, which is the capital of a tiny European country of Monaco. The city was the gambling center of Europe, which is why the European-born scientists at Los Alamos associated Monte Carlo with random variables. The word "Monte" means mountain, and "Carlo" is a version of Charles. In English, they could be called "Mount Charles" methods, but they are not.

2.1 Curse of dimensionaltiy

The application of Monte Carlo methods to high dimensional integration illustrates the *curse of dimensionality*, which says that problems in high dimensions (problems with many variables) are "cursed" by being hard to solve. Methods that apply to low dimensional problems, problems with one variable or just a few variables, become impractical in high dimensions. Computer scientists use the term *complexity* to describe the computer time or memory needed to solve a problem. Many algorithms have complexity that "scales exponentially" with the dimension. We say one function "scales like" another function (non-mathematicians might say "quantity" instead of "function") if there is a rough proportionality between them. If one quantity doubles, then the other roughly doubles. It is not a precise concept, but a useful one nonetheless.

Taylor series are an example of this exponential complexity. Consider an algorithm that uses a Taylor series expansion up to order n in d variables. For one variable, which is $d = 1$, this would be

$$a_0 + a_1 x + \cdots + a_n x^n \ .$$

There are $n+1$ terms. For two variables, which is $d = 2$, the expansion to order

n is

$$\begin{aligned} f(x,y) \;\approx\; & a_0 \\ & + a_{1,0}x + a_{0,1}y \\ & + a_{2,0}x^2 + a_{1,1}xy + a_{0,2}y^2 \\ & + \cdots \\ & + a_{n,0}x^n + \cdots + a_{0,n}y^n \; . \end{aligned}$$

There is one term of order zero, two terms of order 1, three terms of order 2, and $n+1$ terms of order n. For three variables, there are three terms of order 1, which correspond to linear terms x, y, and z. There are six terms of order 2, which correspond to x^2, y^2, z^2, xy, xz, and yz. There are ten terms of order 3. You already can see that the number of terms grows faster with n when d is larger. For d variables, there are $d+1$ first order terms, $d(d+1)/2 \approx \frac{1}{2}d^2$ quadratic terms, and $d(d+1)(d+2)/(3\cdot 2) \approx \frac{1}{6}d^3$ cubic terms.

A combinatorics problem

How many terms are there in d variables of order not more than n? This is a problem in combinatorics and here is the clever solution I learned once. Consider $n+d$ boxes, numbered from 1 to $n+d$. Choose exactly d of them and call the chosen boxes $1 \leq b_1 < b_2 < \cdots < b_d \leq n+d$. Let $\alpha_1 = b_1 - 1$ be the number of boxes to the left of b_1. Let $\alpha_2 = b_2 - b_1 - 1$ be the number of boxes between b_1 and b_2, not counting either end box. Continue in this way, so that $\alpha_d = b_d - b_{d-1} - 1$. Note that $\alpha_k = 0$ if $b_k = b_{k-1} + 1$, which is to say that b_{k-1} and b_k are adjacent. Color the chosen boxes black, so there are d black boxes. Color the rest of the boxes white, so there are n white boxes. The sequence $\alpha = (\alpha_1, \ldots, \alpha_d)$ is a *multi-index.* For each multi-index there is a monomial $x^\alpha = x_1^{\alpha_1} \cdots x_d^{\alpha_d}$. The degree of x^α is $|\alpha| = \alpha_1 + \cdots + \alpha_d$. This is the number of white boxes to the left of b_d, so it is less than n, which is the number of white boxes in all. This establishes a one to one correspondence between size d subsets of $d+n$ boxes and monomials of degree not more than n. The number of such subsets is "$n+d$ choose d", which is

$$N_{n,d} = \binom{n+d}{d} = \frac{(n+d)!}{n!\,d!} \; .$$

My intuition is that factorial expressions like this are big when n and d are large. *Stirling's formula* is a way to figure out how big. *Stirling's approximation*, or *Stirling's formula* is a term used for approximations of the factorial function. There are different versions for different degrees of accuracy. A starting version is that for large m,

$$m! \approx m^m e^{-m} \; . \tag{1}$$

When d and n are large, the e^{-m} parts cancel and we get

$$\begin{aligned} N_{n,d} &\approx \frac{(n+d)^{n+d}}{n^n d^d} \\ &= \frac{(n+d)^n}{n^n} \, \frac{(n+1)^d}{d^d} \\ N_{n,d} &\approx \left(1 + \frac{d}{n}\right)^n \left(1 + \frac{n}{d}\right)^d \end{aligned} \tag{2}$$

The quantities in parentheses are greater than one and they are being exponentiated. For example, if $n = d$ (Taylor series order equals the number of variables), then both terms in parentheses are equal to 2. Then, with some more approximations to simplify expressions,

$$N_{d,d} \approx 2^{2d} \; . \tag{3}$$

That's the curse of dimensionality. For $d = 1$ or $d = 2$, it would be feasible to take Taylor series up to order $n = 100$. But that is impossible for $d = 10$, say. With $d = 10$ and $n = 10$, the approximate formula (3) gives 10^{20}. The US Department of Energy, and similar agencies in Russia, Japan, and China, are competing hard and spending billions of dollars per year to build an "exascale" computer. "Exa" is for 10^{18}, which is 100 times smaller than 10^{20}. Exa is bigger than "peta-scale" (10^{15}), which is bigger than "tera" (10^{12}) and "giga" (10^9), etc. Exercises 1 and 2 have more on this point.

The curse of dimensionality makes it hard to create Monte Carlo methods and hard to understand them. Geometric intuition that works in low dimensions is wrong in high dimensions. Week 2 will have more examples of this.

The curse of dimensionality is the reason for Monte Carlo methods.

2.2 Sample averages and error bars

Suppose $X = (X_1, \ldots, X_d)$ ia a multi-component random variable with probability density $\rho(x)$. Suppose there is a function $V(x)$ and we want to know

$$A = \mathrm{E}[\, V(X)] \; . \tag{4}$$

Suppose there is an algorithm to generate independent samples $X_k \sim \rho$. Then the *sample average* (or sample *mean*) is a "direct" way to estimate A:

$$\widehat{A}_N = \frac{1}{N} \sum_{k=1}^{N} V(X_k) \; . \tag{5}$$

The "hat", as in $\widehat{A}$, indicates that $\widehat{A}$ is an estimate of A.

Error bars are appraisals of the statistical accuracy of Monte Carlo estimates. The number A in (4) is not random. [This is a central point in Monte Carlo, a point that allows Monte Carlo to go far beyond direct simulation.] However, the estimate $\widehat{A}$ is random. Suppose $\sigma_{\widehat{A}}$ is the standard deviation

$$\sigma_{\widehat{A}} = \text{std dev}(\widehat{A}) = \left(\text{var}(\widehat{A})\right)^{\frac{1}{2}} \ . \tag{6}$$

If $\text{var}(V(X)) < \infty$, and if we use the sample average estimate (5), then the standard deviation of the estimator is related to the standard deviation of $V(X)$ by the formula you should know (or quickly look up)

$$\sigma_{\widehat{A}} = \frac{1}{\sqrt{N}}\sigma_V = \frac{1}{\sqrt{N}}\left(\text{var}(V(X))\right)^{\frac{1}{2}} \ . \tag{7}$$

The standard estimator of σ_V, which you also should know or look up, is

$$\widehat{\sigma}_V^2 = \frac{1}{N}\sum_{k=1}^{N}\left(V(X_k) - \widehat{A}\right)^2 \ . \tag{8}$$

In Monte Carlo practice, you make the samples X_k, you evaluate the quantity of interest $V_k = V(X_k)$, you calculate the sample average (5), and you evaluate the sample variance (8). Then you estimate the standard deviation of the estimator using

$$\widehat{\sigma}_{\widehat{A}} = \frac{1}{\sqrt{N}}\left(\widehat{\sigma}_V^2\right)^{\frac{1}{2}} \ . \tag{9}$$

When you're done, you report something like

$$A = \widehat{A}\left(\pm\widehat{\sigma}_{\widehat{A}}\right) \ . \tag{10}$$

Whenever you do Monte Carlo estimation, the answer you report should be in this form – estimated value $\widehat{A}$, and error bar $\widehat{\sigma}$.

When we say σ is the *error bar*, we mean that the actual error is on the order of σ. The actual error is $\widehat{A} - A$. If N is large enough so that the central limit theorem (look this up if you don't remember it) applies, and if you use the error bar (9), then

$$\begin{aligned}
\left|\widehat{A} - A\right| &\leq \ \widehat{\sigma}_{\widehat{A}} \quad \text{about 66\% of the time,} \\
\left|\widehat{A} - A\right| &\leq 2\widehat{\sigma}_{\widehat{A}} \quad \text{about 95\% of the time,} \\
\left|\widehat{A} - A\right| &\leq 3\widehat{\sigma}_{\widehat{A}} \quad \text{about 99\% of the time.}
\end{aligned}$$

We use this to interpret the error bar report (10). On a graph, the *error bar* is a line segment $[\widehat{A} - \sigma, \widehat{A} + \sigma]$. The line segment may be drawn as a bar in a plot. The graph does not imply that A is inside the error bar (the interval). The probability of this is only about 66%. Rather, the size of the error bar tells the person looking at the graph about how large the error is likely to be. If you put 20 Monte Carlo estimates with error bars in a single plot, about a third of them, roughly 7, will not contain the actual A.

A statistician might call our error bar (10) a *one standard deviation* error bar. This error bar is what statisticians call a *confidence interval.* Our *one standard deviation* error bar/confidence interval is smaller than is recommended in statistics classes. They usually recommend two standard deviations, which would be $\widehat{A}(\pm 2\sigma)$. This is the "95% confidence interval". This difference is cultural and not always followed. In this class, I will always give one standard deviation error bars and suggest that you do also. The difference, for me, is the expectations of the person looking at your results. If it's a Monte Carlo practitioner (like you), she/he will expect that the error is on the order of the error bar, not much bigger and not much smaller. If she/he is a non-technical person used to "consuming" statistical data, then she/he is likely to expect that the true value is very likely to be inside the error bar, so that she/he can have confidence in the information you've provided. To me, a plot of 20 Monte Carlo runs with two standard deviation error bars is a little silly, as the actual error is usually much less than the "estimated" error. As I said, this is cultural and esthetic, not scientific. It is a good idea to tell the consumer which error bar you are using.

Don't put error bars on error bars. This advice comes from my Monte Carlo mentor Malvin Kalos (co-author of the great *Monte Carlo Methods* book with Paula Whitlock). The error bar is a rough guess at the size of the error. A rough guess at the error bar is good enough for this purpose. When we get to *Markov chain Monte Carlo*, often called *MCMC*, it will be hard to make error bars at all. MCMC error bars are never as accurate as (10). Even for the direct estimate (8) there are issues people ask about that should usually be unimportant. In statistics classes it is common to use $N-1$ instead of N in the denominator of the variance estimator (8). This is because "$N-1$ is the number of degrees of freedom left after you take away one by estimating the mean". It's also because with $N-1$ the estimator would be *unbiased.* The *bias* of an estimator is the difference between the expected value of the estimator and the quantity being estimated. In this case

$$\sigma_V^2 = \mathrm{var}(V(X)) = \mathrm{E}\left[\frac{1}{N-1} \sum_{k=1}^{N} \left(V^k - \widehat{A} \right)^2 \right] .$$

I have two answers to this. One is that if it makes a noticeable difference, then the sample size N is too small. It is rare to have a Monte Carlo sample with N as small as 10. But if $N = 10$, it would be fine to divide by 10 instead of 9 and have a 10% bias in the error bar.

Another answer is to ask why you want an unbiased estimator at all. The error bar involves the standard deviation, which is the square root of the variance. If Y is a random variable whose expected value is σ_V^2, then $Y^{\frac{1}{2}}$ whose expected value is less than σ_V.

2.3 Bad algorithms

Let $\widehat{A}_N$ be some Monte Carlo estimator of a quantity A that uses N "work". This could be N independent samples or some more complex strategy with N steps. The method is *correct* if $\widehat{A}_N \to A$ as $N \to \infty$. A correct Monte Carlo estimator may be "bad" in the sense that $\widehat{A}_N$ is not an accurate estimate of A unless N is very large, sometimes impossibly large. A Monte Carlo expert spends much of her or his time looking for complicated but good methods when there already is a simple but bad method. A bad method need not be wrong in the sense that it does not converge. It might just be that this theoretical convergence can't happen in your lifetime.

One simple measure of accuracy is the *relative mean square deviation*, or *RMSD*

$$R_N^2 = \mathrm{E}\left[\left(\frac{\widehat{A}_N - A}{A}\right)^2\right] . \tag{11}$$

The *absolute error* is $\widehat{A}_N - A$. More useful is the *relative error*,

$$\frac{\widehat{A}_N - A}{A} .$$

For example, a relative error of $-.05$ means that $\widehat{A}_N$ is lower than A by 5%. An absolute error of $-.05$ is very large if $A = .001$ but very small if $A = 1000$. The error (absolute or relative) is random because $\widehat{A}_N$ is random. An estimator has "roughly 5% accuracy" if $R_n^2 \sim (.05)^2$.

Bad methods arise naturally through the curse of dimensionality. The simple direct approach is probably bad in high dimensions. Direct methods are bad methods for *rare event* problems. Consider a random process that models an engineering system that "fails" with a small probability ϵ. If you estimate ϵ using N independent simulations of the system, the RMSD error measure will definitely be large if $N < \frac{1}{\epsilon}$. For $\epsilon = 10^{-6}$ (the failure probability of an internet switch), you must do at least 10^6 simulations to have any relative accuracy in your estimate of ϵ. Much of this course is devoted to understanding and overcoming these difficulties. This week, however, there is only a contrived example, which is Exercise 3.

3 Generating random variables

Monte Carlo calculations rely on random numbers from specified probability distributions. Monte Carlo *sampling* is the process of generating multi-component random variables from single component random variables with simple distributions. All of these are generated from sequences of independent random numbers U_k. A *random number generator* is an algorithm that creates the sequence U_k. A *sampler* is an algorithm that creates random variables from other distributions from a sequence U_k.

Subsection 3.1 describes random number and the random number generator in Python that will be used in this course. Especially if you're a math person used to Matlab or C++, please pay attention to subsubsection 3.1.1 on how it is implemented in Python. The `"="` sign in Python is easy to understand, but different from Matlab. Using it wrong, which every new Python programmer I've met has done, will make your program go wrong in ways that will be hard to figure out.

Subsection 3.2 is a taste of the problem of *sampling*, which is the problem of making a random variable you're interested in using the sequence U_k. The rest of the course will have many more sophisticated sampling techniques, many of which use the function/mapping method here.

3.1 Random number generators

A perfect *random number generator* would generate a sequence of independent random variables U_k, all uniformly distributed in $[0,1]$. Computer random number generators are really *pseudo* random number generators, because the numbers they make are not random. They are made by the pseudo random number generator algorithm. Experts can devise tests that computer-produced pseudo-random numbers will fail. These tests look for subtle correlations in the U_k. However, in 2020, good random number generators in Python, C++, Matlab, etc., are good enough for any Monte Carlo method I am aware of.

A random number generator is characterized by a data structure and two algorithms that act on it. Here is a description in pseudo-code. Pseudo-code is a way to explain algorithms without giving programming language specific details. The data structure is the *state*, which I call S, is a small collection of integer variables. The *update function*, which I call $\text{Next}(S)$, returns a new state computed from the old one. You get a sequence of states by applying the Next function, $S_{k+1} = \text{Next}(S_k)$. The "Next" function is a simple algorithm that characterizes the random number generator. The *output function*, which I call $\text{Out}(S)$, returns a floating point number in the interval $[0,1]$. You get a pseudo-random sequence by applying "Out" to a sequence of states: $U_k = \text{Out}(S_k)$. Different random number generators have different data structures and update and output functions. A perfect random number generator makes a sequence U_k that is an i.i.d. sequence uniformly distributed in $[0,1]$. Modern random number generators make numbers that are almost impossible to distinguish from true random numbers.

Obsolete *congruenial* random number generators illustrate how random number generators work, but modern random number generators work better. There are three large integers, typically prime, p, q, and r. The state update function is

$$S_{k+1} = \text{Next}(S_k) = qS_k + r \ (\text{mod } p) \ . \tag{12}$$

The output function is

$$U_k = \text{Out}(S_k) = \frac{1}{p} S_k \ .$$

In practice, p, q, and r were 128 bit integers, which means they were integers close to, but slightly smaller than, $2^{128} - 1$. These, and the 128 bit state S_k, were represented by four 32 but computer integers. The random number generator program had routines for doing 128 bit arithmetic when the integers were represented by four 32 bit integers. The output was a 64 but floating point number. With 128 bits of S and 64 bits of U, you could get any floating 64 bit floating point number in the interval $[0, 1]$.

A random number generator produce numbers that are uniformly distributed and independent. It is not hard to see that a congruential random number generator produces numbers that are uniformly distributed. Good congruential generators give numbers that are "random enough" for most Monte Carlo applications, but all of them have correlations that are easy to find if you know how to look for them. To see why S_k could be sort of independent from S_{k+1}, ask what happens from S_k and $S'_k = S_k + 1$. These have U_k and U'_k that are either are represented by exactly the same 64 bit floating point number, or one that is within normal roundoff of it. But, at the next step, if you ignore the reduction mod p, then $S'_{k+1} - S_{k+1} = q$. Then $U'_{k+1} - U_{k+1} = \frac{q}{p}$ is on the order of 1, if q is on the order of p. If $S_k + 2$ and $S_k + 3$, etc., also give identical or nearly identical output numbers, then at the next step, you have $U_{k+1} + 2\frac{q}{p}$, or $U_{k+1} + 3\frac{q}{p}$, etc. The numbers $\frac{q}{p}$, $2\frac{q}{p}$, $3\frac{q}{p}$, etc., are nearly uniformly distributed in the interval $[0, 1]$ (reducing mod 1). This is some idea why U_{k+1} is uniformly distributed, even conditioned on knowing U_k.

3.1.1 Python 3 details – very important

We will use the random number generator that comes with version 1.19 of `Numpy`. If you have an earlier version of Numpy, as I did when I started to prepare these notes, you must upgrade to the latest Numpy. You may have to know more about Python than the average math grad student to get the random number generator to work for you. With the right understanding, it will be easy and seem natural. Python was designed by computer scientists specializing in programming languages. The language has some simple principles that are different from C++ or Matlab. Understanding these principles will help you avoid some Python mistakes (and painful weeks of debugging) most math people make.

Object name binding

A Python *module* is a file that contains a sequence of Python commands. When you "run" a module, the Python *interpreter* executes the commands one by one. The interpreter executes a commend in the current Python *environment*. It takes information from the environment. Executing the command usually changes the environment in some way. As a first approximation, the environment consists of a list of active *names* and a list of *objects*. Each name is bound to an object, but there can be more than one name bound to a given object.

A simple *assignment* command has the form *name* = *expression*. To execute the command, the interpreter first *evaluates* the expression. The "value" of the expression is an *object*. This object either is created, or is found in the list of active objects. The interpreter then *binds* the name on the left to that object. If the name already "exists" (is in the list of active names), then that name is bound to the object. The object it was bound to before is forgotten. If the name is not in the environment, it is added to the list of active names. For example, when the interpreter executes the command `x = 2+3`, it first evaluates the right hand side. The value is an object which consists of an integer with the value 5. The interpreter then binds the name `"x"` to that object (an integer with the value 5). Suppose the next command is `y = x`. When the interpreter evaluates the right side, the value will be the object consisting of the integer 5. In this case, the value is the object that the name `"x"` is bound to, which is the integer 5. It then binds the name `"y"` to this object. The result is that names `"x"` and `"y"` are bound to the same object.

An assignment command, not the simple ones described above, can have the effect of modifying an object, rather than forgetting one. For example

```
import numpy as np
    .....
x    = np.zeros(3)
x[0] = 1.
```

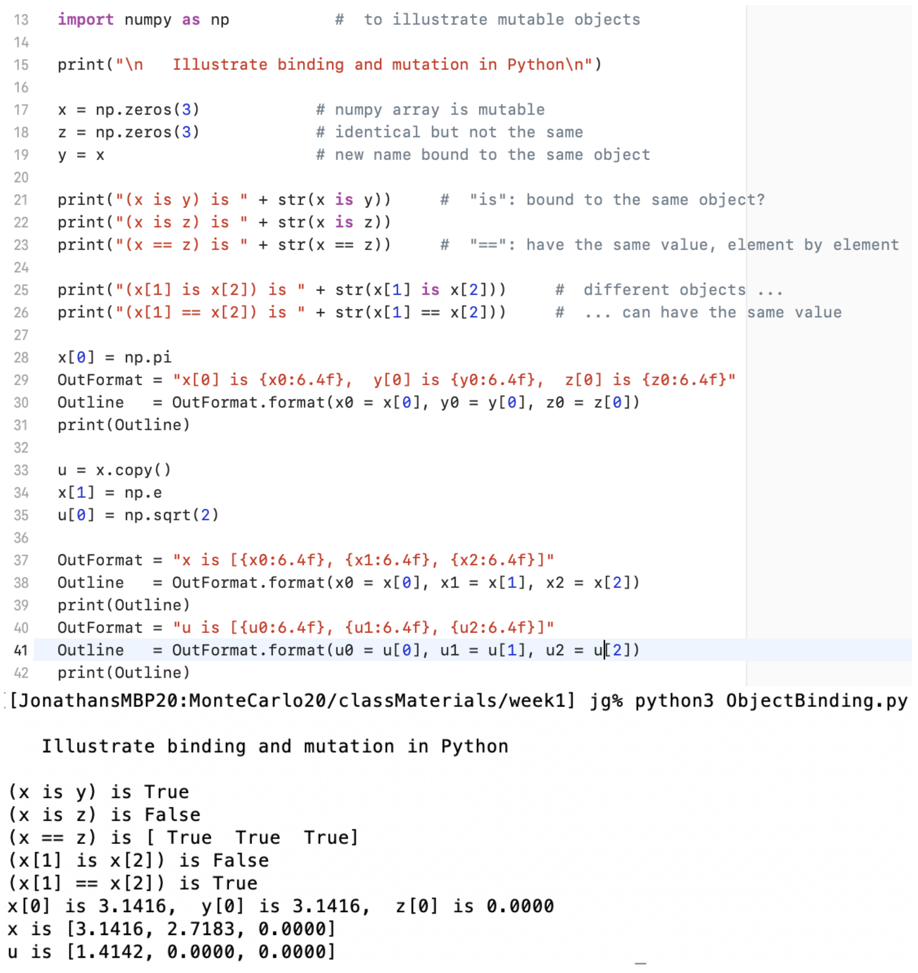
The command `x = np.zeros(3)` is a simple command. The interpreter first evaluates the expression on the right. This produces an object, which is a `numpy` array with three elements. The interpreter then binds the name `"x"` to that `numpy` array object. The next command modifies this object by changing one of its values. The name `"x"` continues to be bound to that object.

Figure 1 illustrates some features of evaluation and name binding, first some of the code, then the output. Line 17 creates the name `"x"` and binds this to the object with is a `numpy` array consisting of three zeros. Line 18 creates a different name, `"z"` and binds it to a different object, which happens to be identical to one `"x"` is bound to. Line 18 creates the name `"y"` and binds it to the object `"x"` is bound to. In Python, the logical operator *is* as in `(A is B)` returns `True` if the the names `"A"` and `"B"` are bound to the same object. The logical operator `==` returns `True` if the expressions `A` and `B` have the same value, sort of. The expression `(x is y)` evaluates to `True` because names `"x"` and `"y"` are bound to the same object. The expression `(x is z)` evaluates to `False` because `"z"` is bound to a different (but identical) object. When you apply the equality test operator `(x == z)`, Python applies the operation "elementwise" to the three entries. The output shows that these are all equal. Lines 26 and 27 illustrate the fact that two objects can be equal in the sense of `==` even though they are not the same object in the sense of `is`.

It is important to remember the difference between *binding* and *copying*. As was said above, but needs to be emphasized, the command of line 19 does not create a new object. Instead, it binds the name `"y"` to the existing object. The assignment command of line 28 modifies the object bound to `"x` by changing the first entry to $\pi = 3.14159\cdots$. The `numpy` name `np.pi` is bound to this value. Since the name `"y"` is bound to the same object, this has the effect of changing `y[0]` to π. Changing the value of `y[0]` is a *side effect* of `x[0] = np.pi`. A *side effect* of a command is a consequence of the command that is not explicitly directed by the command itself. The side effect of line 28 is that `y[0]` changes from 0 to π.

Line 32 shows how to make a copy of a `numpy` array. The expression `x.copy()` has the form `[name].[action]()`. Line 30 does also. See below for a little explanation of this. Executing this command creates a `numpy` array identical to the one bound to `"x"` and binds `"u"` to the new one. The `"u"` object created is identical to the `"x"` object, but modifying `x` does not modify `u` (line 34, sets the second entry of `x` to e), and modifying `u` does not modify `x` (line 35). The last two lines of output verify this. You can download the module here

`ObjectBinding.py`



```
import numpy as np          #  to illustrate mutable objects

print("\n   Illustrate binding and mutation in Python\n")

x = np.zeros(3)            # numpy array is mutable
z = np.zeros(3)            # identical but not the same
y = x                      # new name bound to the same object

print("(x is y) is " + str(x is y))     #  "is": bound to the same object?
print("(x is z) is " + str(x is z))
print("(x == z) is " + str(x == z))     #  "==": have the same value, element by element

print("(x[1] is x[2]) is " + str(x[1] is x[2]))     #  different objects ...
print("(x[1] == x[2]) is " + str(x[1] == x[2]))     #  ... can have the same value

x[0] = np.pi
OutFormat = "x[0] is {x0:6.4f},  y[0] is {y0:6.4f},  z[0] is {z0:6.4f}"
Outline   = OutFormat.format(x0 = x[0], y0 = y[0], z0 = z[0])
print(Outline)

u = x.copy()
x[1] = np.e
u[0] = np.sqrt(2)

OutFormat = "x is [{x0:6.4f}, {x1:6.4f}, {x2:6.4f}]"
Outline   = OutFormat.format(x0 = x[0], x1 = x[1], x2 = x[2])
print(Outline)
OutFormat = "u is [{u0:6.4f}, {u1:6.4f}, {u2:6.4f}]"
Outline   = OutFormat.format(u0 = u[0], u1 = u[1], u2 = u[2])
print(Outline)
```

```
[JonathansMBP20:MonteCarlo20/classMaterials/week1] jg% python3 ObjectBinding.py

   Illustrate binding and mutation in Python

(x is y) is True
(x is z) is False
(x == z) is [ True  True  True]
(x[1] is x[2]) is False
(x[1] == x[2]) is True
x[0] is 3.1416,  y[0] is 3.1416,  z[0] is 0.0000
x is [3.1416, 2.7183, 0.0000]
u is [1.4142, 0.0000, 0.0000]
```

Figure 1: Names and binding. From the module ObjectBinding.py

Classes and the `numpy` random number generator

The `numpy` random number generator is implemented using the Python *class* mechanism. Very roughly speaking, a *data type* is a kind of Python object (integer, float, character string, ...). Some data types are *built in*, which means defined by the Python language itself. Basic types like `int`, `float`, and `string` are built in. A *class* can be thought of as a data type defined by Python code, either in your code or in the code you *import*, including `numpy` and `matplotlib`. If `classname` is the name of a class, then `classname` can be the type of an object.

A *constructor* is one way to get an object of type `classname`. Another way is to call a method that returns an object of that class. The constructor is a Python method that does what it takes to set up an object of type *classname*.

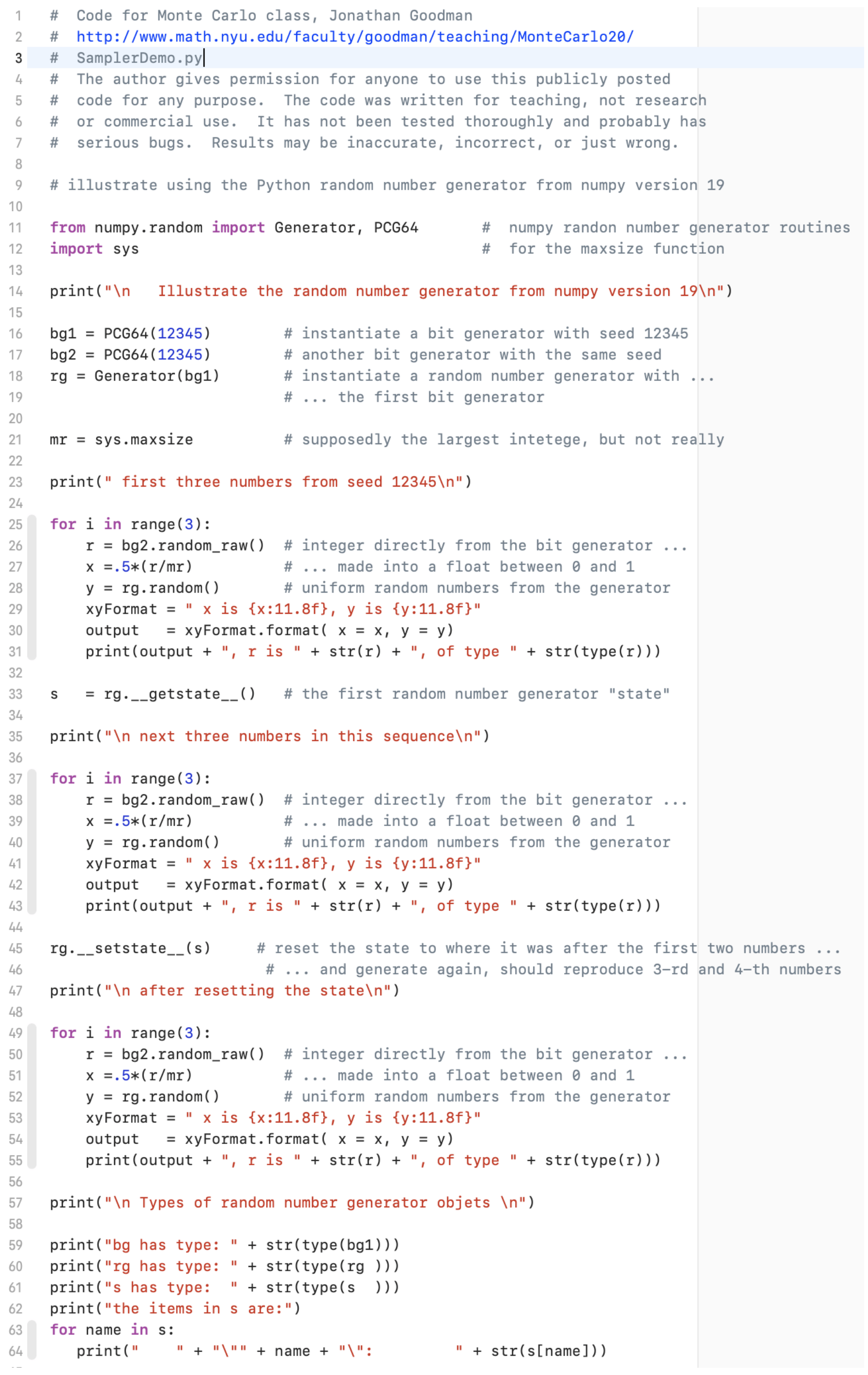
A command `x = [classname]([arguments])` uses the data in `[arguments]` to create an object of type `classname`. The name `"x"` is bound to that object.

In the following commands, the classes named `PCG64` and `Generator` have already been defined.

```
state = PCG64(1234)
rg    = Generator( state )
```

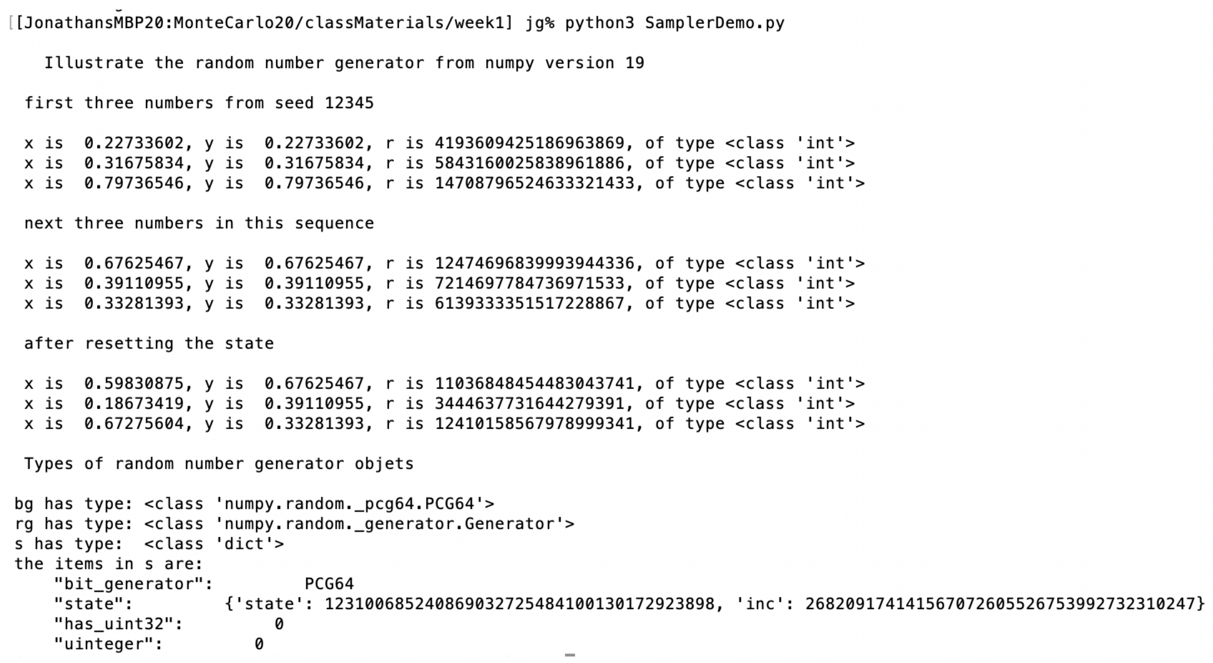
The second command calls the constructor method of the `Generator` class to create an object of type `Generator`. The name `"rg"` is bound to that object. The constructor has one argument, an object of type `PCG64`, which is a class that represents the state of the random number generator. The constructor for the `PCG64` class takes an integer argument, which is `1234` in this case. The constructor for `PCG64` takes the integer as a *seed* to construct a state. The state

SamplerDemo.py



```python
#  Code for Monte Carlo class, Jonathan Goodman
#  http://www.math.nyu.edu/faculty/goodman/teaching/MonteCarlo20/
#  SamplerDemo.py
#  The author gives permission for anyone to use this publicly posted
#  code for any purpose.  The code was written for teaching, not research
#  or commercial use.  It has not been tested thoroughly and probably has
#  serious bugs.  Results may be inaccurate, incorrect, or just wrong.

# illustrate using the Python random number generator from numpy version 19

from numpy.random import Generator, PCG64       #  numpy randon number generator routines
import sys                                      #  for the maxsize function

print("\n   Illustrate the random number generator from numpy version 19\n")

bg1 = PCG64(12345)        # instantiate a bit generator with seed 12345
bg2 = PCG64(12345)        # another bit generator with the same seed
rg = Generator(bg1)       # instantiate a random number generator with ...
                          # ... the first bit generator

mr = sys.maxsize          # supposedly the largest intetege, but not really

print(" first three numbers from seed 12345\n")

for i in range(3):
    r = bg2.random_raw()  # integer directly from the bit generator ...
    x =.5*(r/mr)          # ... made into a float between 0 and 1
    y = rg.random()       # uniform random numbers from the generator
    xyFormat = " x is {x:11.8f}, y is {y:11.8f}"
    output   = xyFormat.format( x = x, y = y)
    print(output + ", r is " + str(r) + ", of type " + str(type(r)))

s   = rg.__getstate__()   # the first random number generator "state"

print("\n next three numbers in this sequence\n")

for i in range(3):
    r = bg2.random_raw()  # integer directly from the bit generator ...
    x =.5*(r/mr)          # ... made into a float between 0 and 1
    y = rg.random()       # uniform random numbers from the generator
    xyFormat = " x is {x:11.8f}, y is {y:11.8f}"
    output   = xyFormat.format( x = x, y = y)
    print(output + ", r is " + str(r) + ", of type " + str(type(r)))

rg.__setstate__(s)     # reset the state to where it was after the first two numbers ...
                        # ... and generate again, should reproduce 3-rd and 4-th numbers
print("\n after resetting the state\n")

for i in range(3):
    r = bg2.random_raw()  # integer directly from the bit generator ...
    x =.5*(r/mr)          # ... made into a float between 0 and 1
    y = rg.random()       # uniform random numbers from the generator
    xyFormat = " x is {x:11.8f}, y is {y:11.8f}"
    output   = xyFormat.format( x = x, y = y)
    print(output + ", r is " + str(r) + ", of type " + str(type(r)))

print("\n Types of random number generator objets \n")

print("bg has type: " + str(type(bg1)))
print("rg has type: " + str(type(rg )))
print("s has type:  " + str(type(s  )))
print("the items in s are:")
for name in s:
   print("    " + "\"" + name + "\":         " + str(s[name]))
```

Figure 2: The code from SamplerDemo.py



```
[[JonathansMBP20:MonteCarlo20/classMaterials/week1] jg% python3 SamplerDemo.py

   Illustrate the random number generator from numpy version 19

 first three numbers from seed 12345

 x is  0.22733602, y is  0.22733602, r is 4193609425186963869, of type <class 'int'>
 x is  0.31675834, y is  0.31675834, r is 5843160025838961886, of type <class 'int'>
 x is  0.79736546, y is  0.79736546, r is 14708796524633321433, of type <class 'int'>

 next three numbers in this sequence

 x is  0.67625467, y is  0.67625467, r is 12474696839993944336, of type <class 'int'>
 x is  0.39110955, y is  0.39110955, r is 7214697784736971533, of type <class 'int'>
 x is  0.33281393, y is  0.33281393, r is 6139333351517228867, of type <class 'int'>

 after resetting the state

 x is  0.59830875, y is  0.67625467, r is 11036848454483043741, of type <class 'int'>
 x is  0.18673419, y is  0.39110955, r is 3444637731644279391, of type <class 'int'>
 x is  0.67275604, y is  0.33281393, r is 12410158567978999341, of type <class 'int'>

 Types of random number generator objets

bg has type: <class 'numpy.random._pcg64.PCG64'>
rg has type: <class 'numpy.random._generator.Generator'>
s has type:  <class 'dict'>
the items in s are:
    "bit_generator":         PCG64
    "state":         {'state': 123100685240869032725484100130172923898, 'inc': 268209174141567072605526753992732310247}
    "has_uint32":         0
    "uinteger":         0
```

Figure 3: Output from SamplerDemo.py

3.2 Making samples using functions

If U is a random variable with some distribution, and $X = f(U)$ then X also is a random variable. Let us suppose that f is monotone, so that if $x = f(u)$, then u is unique. Let $\rho(x)$ be the PDF of X. Then, if $x = f(u)$ and $0 \leq u \leq 1$, then

$$\rho(x) = \frac{1}{f'(u)} \ . \tag{13}$$

Here is an informal derivation. The PDF of X is defined by

$$\rho(x)dx = \Pr(x \leq X \leq x + dx) \ .$$

If $X = f(U)$ and $x = f(u)$, then $dx = f'(u)du$. Since U is uniformly distributed, $\Pr(u \leq U \leq u + du) = du$. Therefore (13) follows from

$$\begin{aligned} \rho(x)dx &= \Pr(x \leq X \leq x + dx) \\ \rho(x)f'(u)du &= \Pr(u \leq U \leq u + du) \\ \rho(x)f'(u)du &= du \ . \end{aligned}$$

The formula (13) may seem odd at first. You can get more confidence in the formula by checking that it makes dimensional sense. There is more on dimensions and units next week. Here, we simply ask why the the probability density $\rho(x)$ is small when $f'(u)$ is large. If $f'(u)$ is large then a small interval in u space is mapped to a large interval in x space. This means the probability in a u interval is spread to a large x interval, which makes any particular x interval less likely. You can look back at the derivation and see that it is a formal expression of this reasoning.

The exponential random variable is produced using the log function. An exponential random variable S with *arrival rate* parameter λ has PDF

$$\rho(s) = \lambda e^{-\lambda s} \ . \tag{14}$$

Subsection 3.4 explains why λ is called a rate. You can generate an exponential random variable from a uniform random variable U with $S = f(U)$ and

$$f(u) = \frac{-1}{\lambda} \log(U) \ . \tag{15}$$

The log is defined because $U > 0$. The log is negative because $U < 1$. This makes S positive. The time S is smaller (generally speaking, since it's random) when the arrival rate λ is larger, which explains λ in the denominator.

3.3 Simulation

A random variable may be the result of a *random process*. A random process may have a random state X_t that "evolves" in time as t increases according to some stochastic model. Subsection 3.4 gives an example. The state X_t as a function of time is the *sample path*. We write $X_\cdot$, or X, or $X_{[0,T]}$ to refer to the whole path rather than the state at a specific time. The notation $X_{[0,T]}$ indicates that the path is simulated up to a specific "final time" T. It is common that a path is simulated until some stopping condition is satisfied.

The *quantity of interest* (abbreviated *QOI*), V, is a single number that depends on the path in some way. [A number that is a function of a random object is a *statistic*.] We express this using notations such as $V(X_\cdot)$ or $V = F(X_{[0,T]})$. The random variable V has some probability distribution, but this distribution is usually too complicated for there to be a simple formula for the PDF.

The Monte Carlo problem is to estimate $\mathrm{E}[V]$. We do this by creating N independent samples of X, which we call $X_1, \ldots, X_N$. The corresponding samples of the V distribution are $V_k = V(X_k)$. This is one situation in which we generate independent samples of a random variable whose PDF is not known. The distribution of V is determined by the random process, not by explicitly specifying a PDF.

2D path example

As an example, consider a simplified model of a polymer in two dimensions. A *polymer* is a large molecule that consists of a large number of identical parts called monomers. In a *chain polymer*, the monomers are connected to form a long chain – each monomer (except the first and last) connected to one in front and one behind. We model the monomers as points, with monomer k at location x_k. We suppose the distance between neighboring monomers is fixed, so $|x_{k+1} - x_k| = r$ for all k. The line segment from x_k to x_{k+1} is a "bond" The polymer is "infinitely floppy" or "infinitely flesible" if the angles between successive bonds is uniformly distributed. We assume that the polymer has l bonds and $l+1$ monomers. The *interaction potential* for two particles a distance D apart is $\phi(D)$. The total interaction potential is the sum of the interaction potential between all pairs, which is

$$V = \sum_{j \neq k} \phi\left(|x_j - x_k|\right) = \sum_{j=0}^{l-1} \sum_{k=j+1}^{l} \phi\left(|x_j - x_k|\right) \ .$$

This quantity depends only on the relative positions of monomers, so we may assume that $x_0 = 0$.

A 2D polymer with uniformly distributed bond angles has $x_k = (x_{1,k}, x_{2,k})$ and

$$\begin{aligned} &(x_{1,k+1}, x_{2,k+1}) = (x_{1,k}, x_{2,k}) + r(\cos(\theta_k), \sin(\theta_k)) \ , \\ &\qquad\qquad \theta_k \in [0, 2\pi] \ \text{ uniformly distributed and independent} \ . \end{aligned}$$

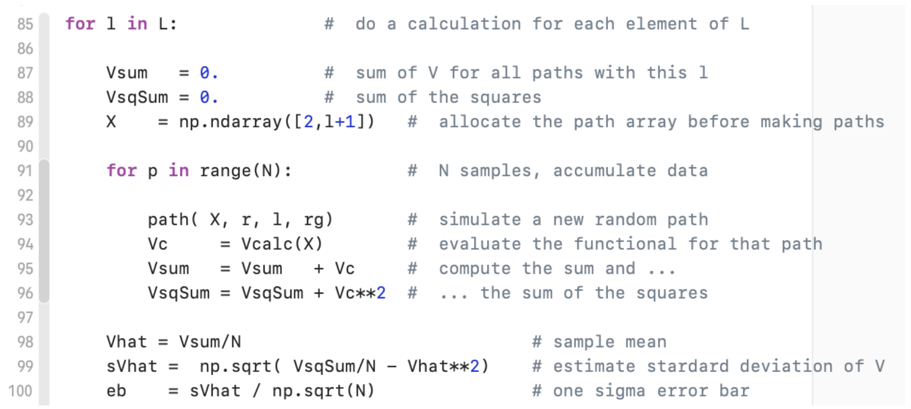
Note that having the bond angle uniformly distributed with respect to the $x-$axis (the formula above) is the same as having the bond angle from x_k to x_{k+1} uniformly distributed with respect to the bond from x_{k-1} to x_k. We choose the interaction potential to be

$$\phi(D) = e^{-D} \ .$$

The exponential decay is motivated by the *Debye Hückel* potential (see Wikipedia). The model has been "nondimensionalized" by setting the Debye Hückel length to 1 and having r be the ratio between the true bond length and the Debye Hückel length. [More on units and non-dimensioqnalized models is coming next week.]

The mathematical model that we apply Monte Carlo to is that X is a path of $l+1$ monomers with uniformly distributed angles and equal lengths. We write X for such a random chain polymer in 2D. The quantity of interest is the total interaction potential $V(X)$. We generate an independent sample of the random variable V by generating an independent random path (monomer) X and then evaluating $V(X)$. Each path is a "simulation" of the monomer. We estimate $A = \mathrm{E}[V]$ by generating N independent paths and evaluating V for each path. We then take the sample average and make a one standard deviation error bar.

The script `MonteCarloEstimation.py` does this. The function defined after `def path( X, r, l, rg):` creates a random path. The arguments are space for the path X (see below), the length ratio r, the number of bonds l, and the random number generator `rg`. The function defined in `Vcalc( X):` evaluates and returns the quantity of interest V for the X given. The main program comes at the end. It instantiates a random number generator *rg* with a specified seed. It then does direct Monte Carlo estimation of A for the l values given in the list. Figure 4 illustrates the steps. For each l in the list of runs it generates N independent paths with l bonds, evaluates V, and computes teh sample mean (line 95 and 98) and the sample variance (line 95 and 99). Line 100 is the one standard deviation error bar.

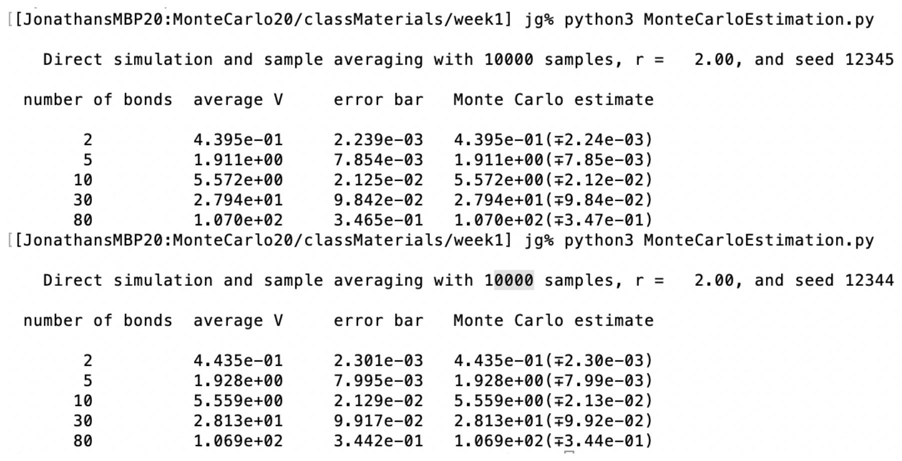


```
 85  for l in L:              #  do a calculation for each element of L
 86
 87      Vsum   = 0.          #  sum of V for all paths with this l
 88      VsqSum = 0.          #  sum of the squares
 89      X    = np.ndarray([2,l+1])   #  allocate the path array before making paths
 90
 91      for p in range(N):           #  N samples, accumulate data
 92
 93          path( X, r, l, rg)       #  simulate a new random path
 94          Vc     = Vcalc(X)        #  evaluate the functional for that path
 95          Vsum   = Vsum   + Vc     #  compute the sum and ...
 96          VsqSum = VsqSum + Vc**2  #  ... the sum of the squares
 97
 98      Vhat = Vsum/N                             # sample mean
 99      sVhat =  np.sqrt( VsqSum/N - Vhat**2)    # estimate stardard deviation of V
100      eb    = sVhat / np.sqrt(N)               # one sigma error bar
```

Figure 4: Code fragment from `MonteCarloEstimation.py` with the estimation loop.

When you download and run the code, you will see right away that Monte Carlo calculations even with modest numbers ($N = 10^4$, $l = 80$) can be very slow. In future weeks we will see how to get better performance out of Python. This allows bigger numbers, but not very much bigger. Monte Carlo like this takes a lot of compute time.

Figure 5 has the output from two runs with different seeds. These are essentially two independent samples of random Monte Carlo output. The last column has output in the standard format. For $l = 2$, the two estimates of V are .4359 and .4435. These differ by .006. According to basic probability, the standard deviation of the difference between two independent estimates is $\sqrt{2}\cdot\sigma$. We write Z for the difference between the estimates, measured in units of this standard deviation. In this case, it's $Z = (.4435 - .4359)/(\sqrt{2} \cdot .00223) = 2.2$. The difference is a little more than two standard deviations, which is big but not outrageous. For $l = 10$, the same calculation gives $Z = -.43$, which is less than one standard deviation. We don't have a theoretical way to check the estimates of the mean total interaction potential A, but the error bars seem to be about right.



```
[[JonathansMBP20:MonteCarlo20/classMaterials/week1] jg% python3 MonteCarloEstimation.py

   Direct simulation and sample averaging with 10000 samples, r =   2.00, and seed 12345

 number of bonds  average V     error bar   Monte Carlo estimate

       2          4.395e-01     2.239e-03   4.395e-01(∓2.24e-03)
       5          1.911e+00     7.854e-03   1.911e+00(∓7.85e-03)
      10          5.572e+00     2.125e-02   5.572e+00(∓2.12e-02)
      30          2.794e+01     9.842e-02   2.794e+01(∓9.84e-02)
      80          1.070e+02     3.465e-01   1.070e+02(∓3.47e-01)
[[JonathansMBP20:MonteCarlo20/classMaterials/week1] jg% python3 MonteCarloEstimation.py

   Direct simulation and sample averaging with 10000 samples, r =   2.00, and seed 12344

 number of bonds  average V     error bar   Monte Carlo estimate

       2          4.435e-01     2.301e-03   4.435e-01(∓2.30e-03)
       5          1.928e+00     7.995e-03   1.928e+00(∓7.99e-03)
      10          5.559e+00     2.129e-02   5.559e+00(∓2.13e-02)
      30          2.813e+01     9.917e-02   2.813e+01(∓9.92e-02)
      80          1.069e+02     3.442e-01   1.069e+02(∓3.44e-01)
```

Figure 5: Output from two runs of `MonteCarloEstimation.py`.

Coding requirements and standards

I expect the code you submit for the homework to follow standards that are illustrated in `MonteCarloEstimation.py` and in the output. It will take you a few minutes of coding time, but the result will be better code. Good code with easy to read and informative output will make you a more productive computational scientist. Studies show that good programmers (people who create code quickly that works and is easy for others to work with) rigidly follow a collection of coding rules. Different good programmers have different rigid coding styles. [Do a web search on "python coding style" to see good programmers arguing about which rigid rules are best.] When one coder works on code written by another, he or she usually tries to follow the coding style of the given code.

- Comments at the top of each file saying who (or which people) write the code, how to get in touch with them, and what it's for.
- Clear separation between different function definitions
- A docstring for each function describing the input and output in detail.
- Easy to identify variable names. In scientific computing code, they can match the names in the writeup.
- "White space" so the code is easy to read, including (when it's easy) aligning `=` signs. Aligning successive lines of code helps the reader, and it helps the coder spot typos.
- Collecting definitions of parameters together in one place. Here, the random number generator is instantiated in lines 69 to 72 and the physical parameters are in lines 74 to 76.

- Formatted floating point output is required. Python has more than one way to do this, but I use the mechanism of lines 78 to 81. Line 78 is a character string with format instructions. Line 79 says which floating point variables are to be formatted into the string. Lines 80 and 81 print the string with numbers formatted into it.
- Put the variables from the run into the output, as lines 80 and 81 do.
- Print output in organized tables with table headings. The heading is line 83. The output rows in the table are created in lines 101 to 103 in the same formatting process. The spacing in lines 83 and 101 are found by trial and error so the numbers in Figure 5 line up.
- Every aspect of the code is commented with comments that are as helpful as possible. An example of an unhelpful comment is

```
s = x + y     #  set s to be the sum of x and y
```

3.4 Simulating a Poisson process

A *Poisson arrival process* is a sequence of positive times $T_1 < T_2 < \cdots$. The probability of an arrival in a fixed small time interval $[t, t+dt]$ is λdt.

$$\Pr(\text{ arrival in } [t, t+dt]) = \lambda dt \ . \tag{16}$$

This λ is the *arrival rate*. Arrivals in disjoint time intervals are independent. The *inter-arrival times* are $S_k > 0$. These are defined by $T_k = T_{k-1} + S_k$. This formula makes sense for $k = 1, 2, \ldots$ if we artificially define $T_0 = 0$. In that case, $T_1 = S_1$, and $T_2 = S_1 + S_2$, etc. The inter-arrival times for a Poisson process are independent exponential random variables with rate parameter λ. This means they have PDF $\rho(s) = \lambda e^{-\lambda s}$. You can simulate a Poisson process by generating independent exponential inter-arrival times S_k using (15) and then adding them to generate the T_k.

The Poisson process is a model of the situation where there are many "agents" who decide independently and randomly when to do something. The Poisson process models the situation where there are many agents and each agent is unlikely to do it. The product of a large number of agents and small "do it" probability leads to a regular stream of events, which are the Poisson arrivals. One example is a call center. Suppose the agents are one million customers who use a certain software package. At any moment, a customer may decide to call customer support. This is rare for each specific customer. But many customers generate a steady stream of calls. The calls come at random times, which model as Poisson arrivals.

The probability model (16) leads to a formula for the PDF of the inter-arrival times. Let $P(s)$ the the cumulative distribution for S and $\rho(s)$ the PDF. These are related by

$$P(s) = \int_0^s \rho(s')\, ds' \ , \ \ P'(s) = \rho(s) \ , \ \ P(0) = 0 \ .$$

The model (16) leads to a differential equation for P or ρ, which leads to the formula (14). The derivation uses Bayes' rule for conditional probability and the following observation: If $S > s$, then $S < s + ds$ if there is an arrival in $[s, s + ds]$. This is a conditional probability. With 16, this leads to

$$\Pr(S < s + ds | S > s) = -\lambda ds \ .$$

The unconditional probability is

$$\Pr(s \leq S \leq s + ds) = \rho(s) ds \ .$$

The Bayes' rule calculation from these facts is

$$\begin{aligned}
\lambda ds &= \Pr(S < s + ds | S > s) \\
&= \frac{\Pr(S < s + ds \text{ and } S > s)}{\Pr(S > s)} \\
&= \frac{\Pr(s \leq S \leq s + ds)}{1 - \Pr(S < s)} \\
&= \frac{\rho(s) ds}{1 - P(s)} \\
\lambda(1 - P(s)) &= P'(s) \ .
\end{aligned}$$

The last differential equation, and the initial condition $p(0) = 0$ leads to $P(s) = 1 - e^{-\lambda s}$ and then to (14).

4 Assignment 1, due September 16

Always check the class message board on the NYU Classes site from home.nyu.edu before doing any work on the assignment.

Corrections: Exercise 2 simplified. Exercise 5 corrected.

1. The Stirling approximation (1) is not "accurate" in the sense that

$$R_m = \frac{m^m e^{-m}}{m!}$$

does not go to 1 as $m \to \infty$. A version of Stirling's formula that is accurate in this sense includes the *prefactor*, $\sqrt{2\pi m}$. The assignment for week 2 will explain how to find this prefactor. The more accurate formula is

$$m! \approx \sqrt{2\pi m}\, m^m e^{-m} \ .$$

This has the property that $R_m \to 1$ as $m \to \infty$, if you use the more accurate approximation in the numerator. The prefactor shows that the Stirling's formula we used in (1) is wrong by a factor that goes to infinity as $m \to \infty$. For example, for $m = 10$, the formula (1) is off by a *factor* of about $\sqrt{20\pi} \approx \sqrt{20 \cdot 3.14} \approx \sqrt{63} \approx 8$.

Show that this does not change the practical conclusion in Section 2. Specifically, suppose we have enough computer time or storage for a billion (10^9 in the US meaning of "billion") Taylor series terms, and we will take $n = d$. Let $n_{\max}$ be the largest number of terms allowed, according to the simple approximation (3). The approximation $2^{10} \approx 10^3$ suggests that $n_{\max} \approx 15$. If you have 15 variables then you can use Taylor series up to order 15 and stay within the "one giga-term" limit ("giga" means 10^9 everywhere). How much does this conclusion change if you use the more accurate Stirling approximation and do the algebra more correctly? How different is the actual $n_{\max}$ from 15?

2. It is more common that the Taylor series degree n is fixed while the number of variables is much larger. Consider the situation where $n = 7$ (Taylor series up to order 7) and d much larger. The first factor on the right of (2) is approximately $(d/7)^7$. Show that the second factor is about e^n. What is $d_{\max}$ if you are constrained to one giga-term? You can be off by a small number, but within, say, 20%.

3. Define
$$A = \int_0^1 e^{\lambda u}\, du \ .$$
Consider the estimator that uses N independent uniformly distributed random variables U_k
$$\widehat{A}_N = \frac{1}{N} \sum_{k=1}^{N} e^{\lambda U_k} \ .$$
   (a) Show that this estimator is unbiased.
   (b) Show that it is "correct" in the sense that $R_N \to 0$ as $N \to \infty$.
   (c) Show that it is bad in the sense of Subsection 2.3 when λ is large. Show by direct calculation that $R_N^2 \approx \frac{\lambda}{2N}$ when λ is large.
   (d) For $\lambda = 50$, approximately what N do you need to estimate A to about 1% accuracy (1% error bar)?

4. The "standard, centered" *Cauchy* probability density is
$$\rho(x) = \frac{1}{\pi} \frac{1}{x^2 + 1} \ . \tag{17}$$
You can make a more general distribution by moving the "center" to $x_0 \neq 0$, as
$$\rho(x) = \frac{1}{\pi} \frac{1}{(x - x_0)^2 + 1} \ .$$

You can change the "width" by replacing x with x/L, as

$$\rho(x) = \frac{1}{\pi L} \frac{1}{\left(\frac{x-x_0}{L}\right))^2 + 1} \ . \tag{18}$$

We will see (Week 2) that a Cauchy random variable does not have a well defined mean (hence "center" $= x_0$ instead of "mean" $= \mu$). The variance also is infinite, hence "width" instead of "standard deviation".

Suppose θ is uniformly distributed in $[0, \pi]$. Consider the mapping

$$X = \frac{\cos(\theta)}{\sin(\theta)} \ .$$

Show that X has the standard centered Cauchy distribution. Explain how to make a random variable with the general Cauchy distribution (18)

5. Suppose T is a random positive time and

$$\Pr(T \leq t + dt \mid T > t) = \mu t \, dt \ .$$

Find the PDF of T.

6. (*Monte Carlo computation*). Here is a simple of an "integrate and fire" neuron. In this model, a nerve cell "fires" (sends off a chemical/electrical pulse) when its cell body reaches a certain threshold potential (also called voltage, but electric potential does not have to be measured in volts) F. Potential (charged ions, actually) leaks out of the cell body with leakage coefficient r. This means that without stimulation the potential X_t satisfies $\frac{d}{dt} X_t = -rX_t$ and $X_{t+s} = e^{-rs} X_t$. The cell body (in this simplified model) also receives stimulation from other nerve cells in the form of arriving pulses of strength 1. If T_k is the arrival time of a pulse, then $X_{T_k+0} = X_{t_k-0}$. The pulse arrivals form a Poisson process with arrival rate λ. If $X_{t_k+0} \geq F$, then the cell fires and the potential resets to zero. Feel free to do some web searching to learn more about synapses, dendrites, firing, neurotransmitters and so on.

Here is a different way to describe the same model. In a small time increment dt, there is an arrival with probability λdt. If an arrival happens, the cell body potential is incremented by 1, so $X_{t+dt} = X_t + 1$. If there is no arrival, the leakage lowers the potential by a deterministic amount $rX_t dt$. If $X_t > F$, then the neuron "fires" and the potential resets to zero:

$$X_{t+dt} = \begin{cases} X_t + 1 & \text{with probability } \lambda dt \\ (1 - rdt)X_t & \text{with probability } 1 - \lambda dt \\ 0 & \text{if } X_t > F \ . \end{cases}$$

Subsection 3.4 explains the equivalence between these descriptions.

Suppose $X_0 = 0$ and τ is the first firing time. Write a method that makes an independent sample path from this process. The method should take

F, λ, r, and the random number generator as arguments. It should return two quantities of interest, τ, and N_τ, the number of Poisson arrivals up to time τ. Use this method to generate enough independent samples of τ to estimate $\mathrm{E}[\tau]$ and $\mathrm{E}[N_\tau]$ for various values of the parameters. Make a table with parameter values, estimated expected values for the quantities of interest, and one standard deviation error bars. If you choose uninteresting parameter values, N_τ will nearly always be 1 (or zero, depending on whether you count the last arrival). If you choose impractical values, then N_τ will be so large that the simulation is impractical. Experiment with your code to see what parameter values are interesting and practical (or uninteresting or impractical).