Simulation

Chris Paciorek

2023-10-17

Table of contents

1. Monte Carlo considerations **3**
Motivating example . . . 3
Monte Carlo (MC) basics . . . 3
Monte Carlo overview . . . 3
Simulation uncertainty (i.e., Monte Carlo uncertainty) . . . 5
Final notes . . . 5
Variance reduction (optional) . . . 5

2. Design of simulation studies **6**
Basic steps of a simulation study . . . 6
Various considerations . . . 7
Experimental Design (optional) . . . 8

3. Implementation of simulation studies **9**
Computational efficiency . . . 9
Analysis and reporting . . . 10

4. Random number generation (RNG) **10**
Generating random uniforms on a computer . . . 10
Sequential congruential generators . . . 11
PCG generators . . . 14
Mersenne Twister . . . 15
The period versus the number of unique values generated . . . 15
The seed and the state . . . 16
RNG in Python . . . 16
Choosing a generator . . . 16
Using the Mersenne Twister . . . 17
Using PCG64 . . . 18
RNG in parallel . . . 19

5. Generating random variables **19**
Multivariate distributions . . . 19
Inverse CDF . . . 20

Rejection sampling 20
Adaptive rejection sampling (optional) 21
Importance sampling 21
Ratio of uniforms (optional) 21

PDF

References:

- Gentle: Computational Statistics
- Monahan: Numerical Methods of Statistics

Many (most?) statistical papers include a simulation (i.e., Monte Carlo) study. Many papers on machine learning methods also include a simulation study. The basic idea is that closed-form mathematical analysis of the properties of a statistical or machine learning method/model is often hard to do. Even if possible, it usually involves approximations or simplifications. A canonical situation in statistics is that we have an asymptotic result and we want to know what happens in finite samples, but often we do not even have the asymptotic result. Instead, we can estimate mathematical expressions using random numbers. So we design a simulation study to evaluate the method/model or compare multiple methods. The result is that the researcher carries out an experiment (on the computer, sometimes called *in silico*), generally varying different factors to see what has an effect on the outcome of interest.

The basic strategy generally involves simulating data and then using the method(s) on the simulated data, summarizing the results to assess/compare the method(s).

Most simulation studies aim to approximate an integral, generally an expected value (mean, bias, variance, MSE, probability, etc.). In low dimensions, methods such as Gaussian quadrature are best for estimating an integral but these methods don't scale well, so in higher dimensions (e.g., the usual situation with n observations) we often use Monte Carlo techniques.

To be more concrete:

- If we have a *method for estimating a model parameter* (including estimating uncertainty), such as a regression coefficient, what properties do we want the method to have and what criteria could we use?
- If we have a *prediction method* (including prediction uncertainty), what properties do we want the method to have and what criteria could we use?
- If we have a *method for doing a hypothesis test*, what criteria would we use to assess the hypothesis test? What properties do we want the test to have?
- If we have a *method for finding a confidence interval or a prediction interval*, what criteria would we use to assess the interval?

1. Monte Carlo considerations

Motivating example

Let's consider linear regression, with observations $Y = (y_1, y_2, \ldots, y_n)$ and an $n \times p$ matrix of predictors/covariates/features/variables X, where $\hat{\beta} = (X^\top X)^{-1} X^\top Y$. If we assume that we have $EY = X\beta$ and $\text{Var}(Y) = \sigma^2 I$, then we can determine analytically that we have

$$\begin{aligned} E\hat{\beta} &= \beta \\ \text{Var}(\hat{\beta}) = E((\hat{\beta} - E\hat{\beta})^2) &= \sigma^2 (X^\top X)^{-1} \\ \text{MSPE}(Y^*) = E(Y^* - \hat{Y})^2) &= \sigma^2 (1 + X^{*\top} (X^\top X)^{-1} X^*). \end{aligned}$$

where $Y^*$is some new observation we'd like to predict given X^*.

But suppose that we're interested in the properties of standard regression estimation when in reality the mean is not linear in X or the properties of the errors are more complicated than having independent homoscedastic errors. (This is always the case, but the issue is how far from the truth the standard assumptions are.) Or suppose we have a modified procedure to produce $\hat{\beta}$, such as a procedure that is robust to outliers. In those cases, we cannot compute the expectations above analytically.

Instead we decide to use a Monte Carlo estimate. To keep the notation more simple, let's just consider one element of the vector β (i.e., one of the regression coefficients) and continue to call that β. If we randomly generate m different datasets from some distribution f, and $\hat{\beta}_i$ is the estimated coefficient based on the ith dataset: $Y_i = (y_{i1}, y_{i2}, \ldots, y_{in})$, then we can estimate $E\hat{\beta}$ under that distribution f as

$$\hat{E}(\hat{\beta}) = \bar{\hat{\beta}} = \frac{1}{m} \sum_{i=1}^{m} \hat{\beta}_i$$

Or to estimate the variance, we have

$$\widehat{\text{Var}}(\hat{\beta}) = \frac{1}{m} \sum_{i=1}^{m} (\hat{\beta}_i - \bar{\hat{\beta}})^2.$$

In evaluating the performance of regression under non-standard conditions or the performance of our robust regression procedure, what decisions do we have to make to be able to carry out our Monte Carlo procedure?

Next let's think about Monte Carlo methods in general.

Monte Carlo (MC) basics

Monte Carlo overview

The basic idea is that we often want to estimate $\phi \equiv E_f(h(Y))$ for $Y \sim f$. Note that if h is an indicator function, this includes estimation of probabilities, e.g., for a scalar Y, we have $p = P(Y \leq y) = F(y) = \int_{-\infty}^{y} f(t)dt = \int I(t \leq y) f(t)dt = E_f(I(Y \leq y))$. We would estimate variances or MSEs by having h involve squared terms.

We get an MC estimate of ϕ based on an iid sample of a large number of values of Y from f:

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} h(Y_i),$$

which is justified by the Law of Large Numbers:

$$\lim_{m\to\infty} \frac{1}{m}\sum_{i=1}^{m} h(Y_i) = E_f h(Y).$$

Note that in most simulation studies, Y is an entire dataset (predictors/covariates), and the "iid sample" means generating m different datasets from f, i.e., $Y_i \in \{Y_1, \ldots, Y_m\}$ not m different scalar values. If the dataset has n observations, then $Y_i = (Y_{i1}, \ldots, Y_{in})$.

Back to the regression example

Let's relate that back to our regression example. In that particular case, if we're interested in whether the regression estimator is biased, we want to know:

$$\phi = E\hat{\beta},$$

where $h(Y) = \hat{\beta}(Y)$. We can use the Monte Carlo estimate of ϕ:

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} h(Y_i) = \frac{1}{m}\sum_{i=1}^{m} \hat{\beta}_i = \widehat{E(\hat{\beta})}.$$

If we are interested in the variance of the regression estimator, we have

$$\phi = \mathrm{Var}(\hat{\beta}) = E_f((\hat{\beta} - E\hat{\beta})^2)$$

and we can use the Monte Carlo estimate of ϕ:

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} h(Y_i) = \frac{1}{m}\sum_{i=1}^{m} (\hat{\beta}_i - E\hat{\beta})^2 = \widehat{\mathrm{Var}(\hat{\beta})}$$

where

$$h(Y) = (\hat{\beta} - E\hat{\beta})^2.$$

Finally note that we also need to use the Monte Carlo estimate of $E\hat{\beta}$ in the Monte Carlo estimation of the variance.

We might also be interested in the coverage of a confidence interval. In that case we have

$$h(Y) = 1_{\beta \in CI(Y)}$$

and we can estimate the coverage as

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} 1_{\beta \in CI(y_i)}.$$

Of course we want that $\hat{\phi} \approx 1 - \alpha$ for a $100(1-\alpha)$ confidence interval. In the standard case of a 95% interval we want $\hat{\phi} \approx 0.95$.

Simulation uncertainty (i.e., Monte Carlo uncertainty)

Since $\hat{\phi}$ is simply an average of m identically-distributed values, $h(Y_1), \ldots, h(Y_m)$, the simulation variance of $\hat{\phi}$ is $\text{Var}(\hat{\phi}) = \sigma^2/m$, with $\sigma^2 = \text{Var}(h(Y))$. An estimator of $\sigma^2 = E_f((h(Y) - \phi)^2)$ is

$$\hat{\sigma}^2 = \frac{1}{m-1} \sum_{i=1}^{m} (h(Y_i) - \hat{\phi})^2$$

So our MC simulation error is based on

$$\widehat{\text{Var}}(\hat{\phi}) = \frac{\hat{\sigma}^2}{m} = \frac{1}{m(m-1)} \sum_{i=1}^{m} (h(Y_i) - \hat{\phi})^2.$$

Note that this is particularly confusing if we have $\hat{\phi} = \widehat{\text{Var}(\hat{\beta})}$ because then we have $\widehat{\text{Var}}(\hat{\phi}) = \widehat{\text{Var}}(\widehat{\text{Var}(\hat{\beta})})$!

The simulation variance is $O(\frac{1}{m})$ because we have m^2 in the denominator and a sum over m terms in the numerator.

Note that in the simulation setting, the randomness in the system is very well-defined (as it is in survey sampling, but unlike in most other applications of statistics), because it comes from the RNG that we perform as part of our attempt to estimate ϕ. Happily, we are in control of m, so in principle we can reduce the simulation error to as little as we desire. Unhappily, as usual, the simulation standard error goes down with the square root of m.

> **Important**: This is the uncertainty in our simulation-based estimate of some quantity (expectation) of interest. It is NOT the statistical uncertainty in a problem.

Back to the regression example

Some examples of simulation variances we might be interested in in the regression example include:

- Uncertainty in our estimate of bias: $\widehat{\text{Var}}(\hat{E}(\hat{\beta}) - \beta)$.
- Uncertainty in the estimated variance of the estimated coefficient: $\widehat{\text{Var}}(\widehat{\text{Var}}(\hat{\beta}))$.
- Uncertainty in the estimated mean square prediction error: $\widehat{\text{Var}}(\widehat{\text{MSPE}}(Y^*))$.

In all cases we have to estimate the simulation variance, hence the $\widehat{\text{Var}}()$ notation.

Final notes

Sometimes the Y_i are generated in a dependent fashion (e.g., sequential MC or MCMC), in which case this variance estimator, $\widehat{\text{Var}}(\hat{\phi})$ does not hold because the samples are not IID, but the estimator $\hat{\phi}$ is still a valid, unbiased estimator of ϕ.

Variance reduction (optional)

There are some tools for variance reduction in MC settings. One is importance sampling (see Section 3). Others are the use of control variates and antithetic sampling. I haven't personally run across these latter in practice, so I'm not sure how widely used they are and won't go into them here.

In some cases we can set up natural strata, for which we know the probability of being in each stratum. Then we would estimate μ for each stratum and combine the estimates based on the probabilities. The intuition is that we remove the variability in sampling amongst the strata from our simulation.

Another strategy that comes up in MCMC contexts is *Rao-Blackwellization*. Suppose we want to know $E(h(X))$ where $X = \{X_1, X_2\}$. Iterated expectation tells us that $E(h(X)) = E(E(h(X)|X_2)$. If we can compute $E(h(X)|X_2) = \int h(x_1, x_2) f(x_1|x_2) dx_1$ then we should avoid introducing stochasticity related to the X_1 draw (since we can analytically integrate over that) and only average over stochasticity from the X_2 draw by estimating $E_{X_2}(E(h(X)|X_2)$. The estimator is

$$\hat{\mu}_{RB} = \frac{1}{m} \sum_{i=1}^{m} E(h(X)|X_{2,i})$$

where we either draw from the marginal distribution of X_2, or equivalently, draw X, but only use X_2. Our MC estimator averages over the simulated values of X_2. This is called Rao-Blackwellization because it relates to the idea of conditioning on a sufficient statistic. It has lower variance because the variance of each term in the sum of the Rao-Blackwellized estimator is $\text{Var}(E(h(X)|X_2)$, which is less than the variance in the usual MC estimator, $\text{Var}(h(X))$, based on the usual iterated variance formula: $V(X) = E(V(X|Y)) + V(E(X|Y)) \Rightarrow V(E(X|Y)) < V(X)$.

2. Design of simulation studies

Consider the paper that is part of PS5. We can think about designing a simulation study in that context.

First, what are the key issues that need to be assessed to evaluate their methodology?

Second, what do we need to consider in carrying out a simulation study to address those issues? I.e., what are the key decisions to be made in setting up the simulations?

Basic steps of a simulation study

1. Specify what makes up an individual experiment (i.e., the individual simulated dataset) given a specific set of inputs: sample size, distribution(s) to use, parameter values, statistic of interest, etc. In other words, exactly how would you generate one simulated dataset?
2. Often you'll want to see how your results will vary if you change some of the inputs; e.g., sample sizes, parameter values, data generating mechanisms. So determine what factors you'll want to vary. Each unique combination of input values will be a scenario.
3. Write code to carry out the individual experiment and return the quantity of interest, with arguments to your code being the inputs that you want to vary.
4. For each combination of inputs you want to explore (each scenario), repeat the experiment m times. Note this is an easily parallel calculation (in both the data generating dimension and the inputs dimension(s)).
5. Summarize the results for each scenario, quantifying simulation uncertainty.
6. Report the results in graphical or tabular form.

Often a simulation study will compare multiple methods, so you'll need to do steps 3-6 for each method.

Various considerations

Since a simulation study is an experiment, we should use the same principles of design and analysis we would recommend when advising a practicioner on setting up a scientific experiment.

These include efficiency, reporting of uncertainty, reproducibility and documentation.

In generating the data for a simulation study, we want to think about what structure real data would have that we want to mimic in the simulation study: distributional assumptions, parameter values, dependence structure, outliers, random effects, sample size (n), etc.

All of these may become input variables in a simulation study. Often we compare two or more statistical methods conditioning on the data context and then assess whether the differences between methods vary with the data context choices. E.g., if we compare an MLE to a robust estimator, which is better under a given set of choices about the data generating mechanism and how sensitive is the comparison to changing the features of the data generating mechanism? So the "treatment variable" is the choice of statistical method. We're then interested in sensitivity to the conditions (different input values).

Often we can have a large number of replicates (m) because the simulation is fast on a computer, so we can sometimes reduce the simulation error to essentially zero and thereby avoid reporting uncertainty. To do this, we need to calculate the simulation standard error, generally, $s/\sqrt{m}$ and see how it compares to the effect sizes. This is particularly important when reporting on the bias of a statistical method.

We might denote the data, which could be the statistical estimator under each of two methods as Y_{ijklq}, where q indexes treatment, j, k, l index different additional input variables, and $i \in \{1, \ldots, m\}$ indexes the replicate. E.g., j might index whether the data are from a t or normal, k the value of a parameter, and l the dataset sample size (i.e., different levels of n).

One can think about choosing m based on a basic power calculation, though since we can always generate more replicates, one might just proceed sequentially and stop when the precision of the results is sufficient.

When comparing methods, it's best to use the same simulated datasets for each level of the treatment variable and to do an analysis that controls for the dataset (i.e., for the random numbers used), thereby removing some variability from the error term. A simple example is to do a paired analysis, where we look at differences between the outcome for two statistical methods, pairing based on the simulated dataset.

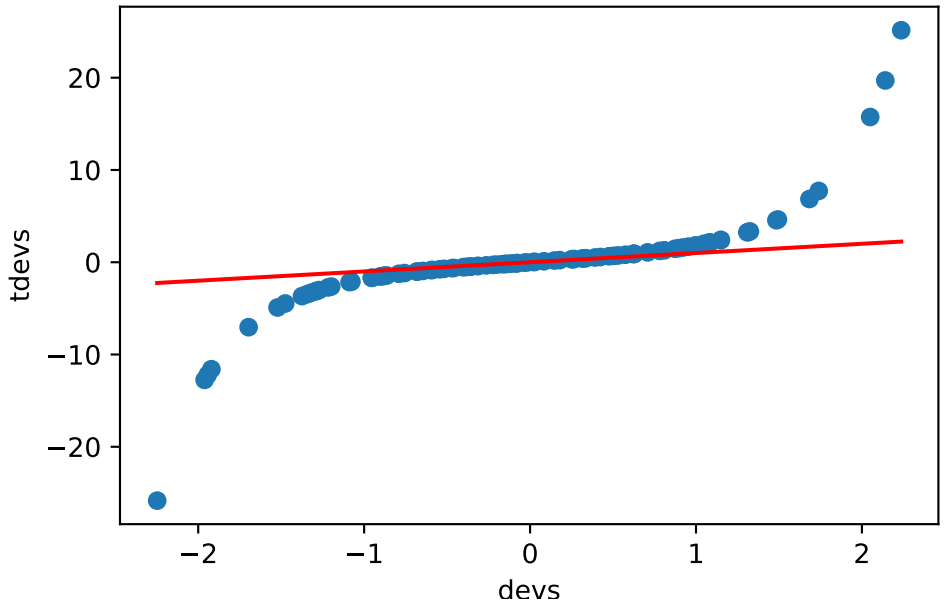
One can even use the "same" random number generation for the replicates under different conditions. E.g., in assessing sensitivity to a t vs. normal data generating mechanism, we might generate the normal RVs and then for the t use the same random numbers, in the sense of using the same quantiles of the t as were generated for the normal - this is pretty easy, as seen below. This helps to control for random differences between the datasets.

```
from scipy.stats import t, norm

devs = np.random.normal(size=100)
```

```
tdevs = t.ppf(norm.cdf(devs), df=1)

plt.scatter(devs, tdevs)
plt.xlabel('devs'); plt.ylabel('tdevs')
plt.plot([min(devs), max(devs)], [min(devs), max(devs)], color='red')
plt.show()
```



Experimental Design (optional)

A typical context is that one wants to know the effect of multiple input variables on some outcome. Often, scientists, and even statisticians doing simulation studies will vary one input variable at a time. As we know from standard experimental design, this is inefficient.

The standard strategy is to discretize the inputs, each into a small number of levels. If we have a small enough number of inputs and of levels, we can do a full factorial design (potentially with replication). For example if we have three inputs and three levels each, we have 3^3 different treatment combinations. Choosing the levels in a reasonable way is obviously important.

As the number of inputs and/or levels increases to the point that we can't carry out the full factorial, a fractional factorial is an option. This carefully chooses which treatment combinations to omit. The goal is to achieve balance across the levels in a way that allows us to estimate lower level effects (in particular main effects) but not all high-order interactions. What happens is that high-order interactions are aliased to (confounded with) lower-order effects. For example you might choose a

fractional factorial design so that you can estimate main effects and two-way interactions but not higher-order interactions.

In interpreting the results, I suggest focusing on the decomposition of sums of squares and not on statistical significance. In most cases, we expect the inputs to have at least some effect on the outcome, so the null hypothesis is a straw man. Better to assess the magnitude of the impacts of the different inputs.

When one has a very large number of inputs, one can use the Latin hypercube approach to sample in the input space in a uniform way, spreading the points out so that each input is sampled uniformly. Assume that each input is $\mathcal{U}(0,1)$ (one can easily transform to whatever marginal distributions you want). Suppose that you can run m samples. Then for each input variable, we divide the unit interval into m bins and randomly choose the order of bins and the position within each bin. This is done independently for each variable and then combined to give m samples from the input space. We would then analyze main effects and perhaps two-way interactions to assess which inputs seem to be most important.

Even amongst statisticians, taking an experimental design approach to a simulation study is not particularly common, but it's worth considering.

3. Implementation of simulation studies

Luke Miratrix (a UCB Stats PhD alum) has prepared a nice tutorial on carrying out a simulation study, including helpful R code. So if the discussion here is not concrete enough or you want to see how to effectively implement such a study, see *simulation_tutorial_miratrix.pdf* and the similarly named R code file.

Computational efficiency

Parallel processing is often helpful for simulation studies. The reason is that simulation studies are embarrassingly parallel - we can send each replicate to a different computer processor and then collect the results back, and the speedup should scale directly with the number of processors we used. Since we often need to some sort of looping, writing code in C/C++ and compiling and linking to the code from Python may also be a good strategy, albeit one not covered in this course.

A handy function in Python is `itertools.product` to get all combinations of a set of vectors.

```
import itertools

thetaLevels = ["low", "med", "hi"]
n = [10, 100, 1000]
tVsNorm = ["t", "norm"]
levels = list(itertools.product(thetaLevels, tVsNorm, n))
```

Analysis and reporting

Often results are reported simply in tables, but it can be helpful to think through whether a graphical representation is more informative (sometimes it's not or it's worse, but in some cases it may be much better). Since you'll often have a variety of scenarios to display, using trellis plots in ggplot2 via the `facet_wrap` function will often be a good approach to display how results vary as a function of multiple inputs in R. In Python, it looks like there are various ways (`RPlot` in pandas, seaborn, plotly), but I don't know what the most standard way is.

You should set the seed when you start the experiment, so that it's possible to replicate it. It's also a good idea to save the current value of the seed whenever you save interim results, so that you can restart simulations (this is particularly helpful for MCMC) at the exact point you left off, including the random number sequence.

To enhance reproducibility, it's good practice to post your simulation code (and potentially simulated data) on GitHub, on your website, or as supplementary material with the journal. Another person should be able to fully reproduce your results, including the exact random number generation that you did (e.g., you should provide code for how you set the random seed for your randon number generator).

Many journals are requiring increasingly detailed documentation of the code and data used in your work, including code and data for simulations. Here are the American Statistical Association's requirements on documenting computations in its journals:

"The ASA strongly encourages authors to submit datasets, code, other programs, and/or extended appendices that are directly relevant to their submitted articles. These materials are valuable to users of the ASA's journals and further the profession's commitment to reproducible research. Whenever a dataset is used, its source should be fully documented and the data should be made available as on online supplement. Exceptions for reasons of security or confidentiality may be granted by the Editor. Whenever specific code has been used to implement or illustrate the results of a paper, that code should be made available if possible. [....snip....] Articles reporting results based on computation should provide enough information so that readers can evaluate the quality of the results. Such information includes estimated accuracy of results, as well as descriptions of pseudorandom-number generators, numerical algorithms, programming languages, and major software components used."

4. Random number generation (RNG)

At the core of simulations is the ability to generate random numbers, and based on that, random variables. On a computer, our goal is to generate sequences of pseudo-random numbers that behave like random numbers but are replicable. The reason that replicability is important is so that we can reproduce the simulation.

Generating random uniforms on a computer

Generating a sequence of random standard uniforms is the basis for all generation of random variables, since random uniforms (either a single one or more than one) can be used to generate values from other distributions. Most random numbers on a computer are *pseudo-random*. The numbers are chosen from a deterministic stream of numbers that behave like random numbers but are actually a finite sequence (recall that both integers and real numbers on a computer are actually discrete and there are finitely

many distinct values), so it's actually possible to get repeats. The seed of a RNG is the place within that sequence where you start to use the pseudo-random numbers.

Sequential congruential generators

Many RNG methods are sequential congruential methods. The basic idea is that the next value is

$$u_k = f(u_{k-1}, \ldots, u_{k-j}) \mod m$$

for some function, f, and some positive integer m . Often $j = 1$. *mod* just means to take the remainder after dividing by m. One then generates the random standard uniform value as u_k/m, which by construction is in $[0, 1]$. For our discussion below, it is important to distinguish the *state* (u) from the output of the RNG.

Given the construction, such sequences are periodic if the subsequence ever reappears, which is of course guaranteed because there is a finite number of possible subsequence values given that all the u_k values are remainders of divisions by a fixed number . One key to a good random number generator (RNG) is to have a very long period.

An example of a sequential congruential method is a basic linear congruential generator:

$$u_k = (au_{k-1} + c) \mod m$$

with integer a, m, c, and u_k values. (Note that in some cases $c = 0$, in which case the periodicity can't exceed $m - 1$ as the method is then set up so that we never get $u_k = 0$ as this causes the algorithm to break.) The seed is the initial state, u_0 - i.e., the point in the sequence at which we start. By setting the seed you guarantee reproducibility since given a starting value, the sequence is deterministic. In general a, c and m are chosen to be 'large'. The standard values of m are Mersenne primes, which have the form $2^p - 1$ (but these are not prime for all p). Here's an example of a linear congruential sampler (with $c = 0$):

```
n = 100
a = 171
m = 30269

u = np.empty(n)
u[0] = 7306

for i in range(1, n):
    u[i] = (a * u[i-1]) % m

u = u / m
uFromNP = np.random.uniform(size = n)

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)
plt.plot(range(1, n+1), u)
```

```
  plt.title("manual")
  plt.xlabel("Index"); plt.ylabel("Value")

  plt.subplot(2, 2, 2)
  plt.plot(range(1, n+1), uFromNP)
  plt.title("numpy")
  plt.xlabel("Index"); plt.ylabel("Value")

  plt.subplot(2, 2, 3)
  plt.hist(u, bins=25)
```

```
(array([6., 3., 6., 4., 1., 4., 6., 3., 4., 9., 4., 5., 1., 1., 2., 7., 1.,
       2., 5., 2., 6., 5., 4., 4., 5.]), array([0.01833559, 0.05743434, 0.09653309, 0.13563183, 0.174
       0.21382933, 0.25292808, 0.29202683, 0.33112557, 0.37022432,
       0.40932307, 0.44842182, 0.48752057, 0.52661931, 0.56571806,
       0.60481681, 0.64391556, 0.68301431, 0.72211305, 0.7612118 ,
       0.80031055, 0.8394093 , 0.87850804, 0.91760679, 0.95670554,
       0.99580429]), <BarContainer object of 25 artists>)
```

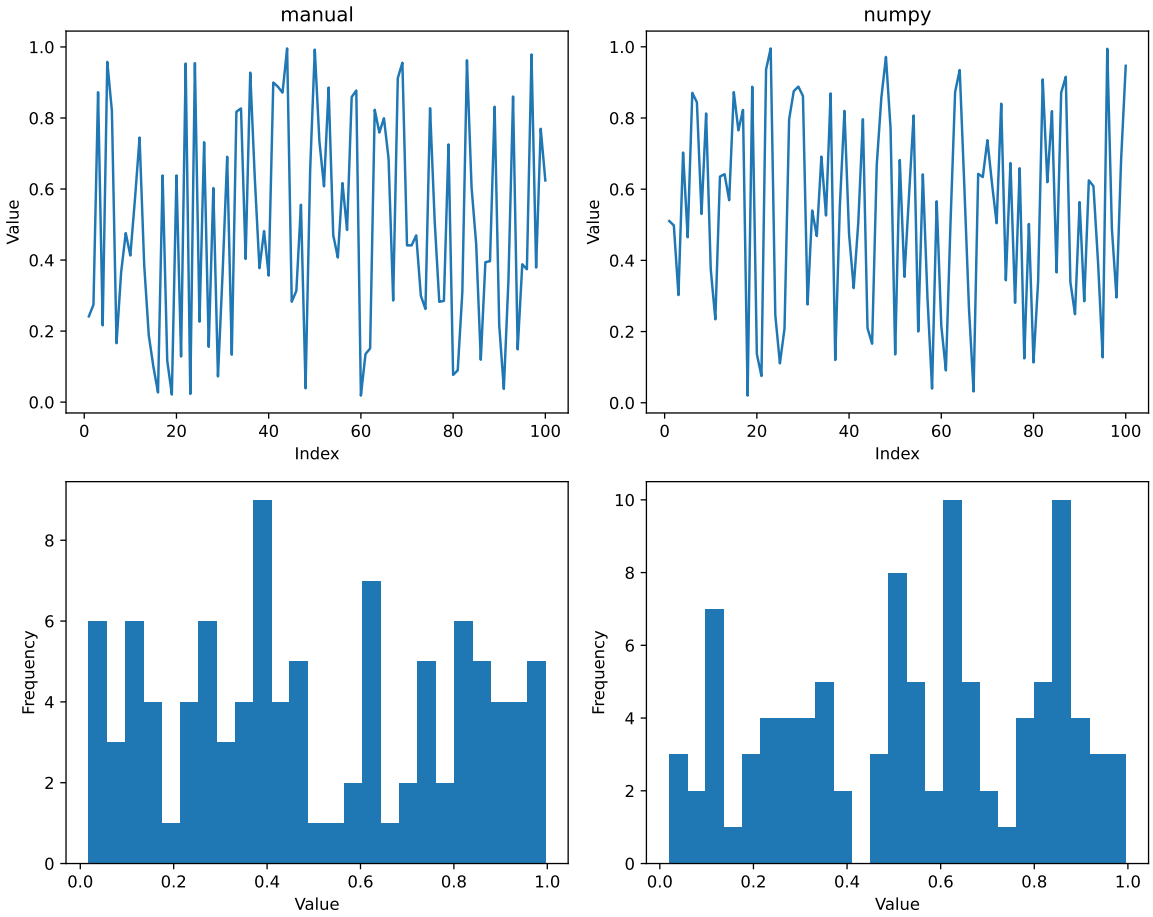
```
  plt.xlabel("Value"); plt.ylabel("Frequency")

  plt.subplot(2, 2, 4)
  plt.hist(uFromNP, bins=25)
```

```
(array([5., 2., 3., 1., 6., 5., 3., 3., 3., 2., 6., 3., 5., 7., 4., 3., 3.,
       4., 3., 5., 6., 4., 3., 5., 6.]), array([0.03311289, 0.07157103, 0.11002917, 0.14848731, 0.186
       0.22540358, 0.26386172, 0.30231986, 0.340778  , 0.37923614,
       0.41769427, 0.45615241, 0.49461055, 0.53306869, 0.57152683,
       0.60998496, 0.6484431 , 0.68690124, 0.72535938, 0.76381752,
       0.80227565, 0.84073379, 0.87919193, 0.91765007, 0.95610821,
       0.99456635]), <BarContainer object of 25 artists>)
```

```
  plt.xlabel("Value"); plt.ylabel("Frequency")

  plt.tight_layout()
  plt.show()
```



A wide variety of different RNG have been proposed. Many have turned out to have substantial defects based on tests designed to assess if the behavior of the RNG mimics true randomness. Some of the behavior we want to ensure is uniformity of each individual random deviate, independence of sequences of deviates, and multivariate uniformity of subsequences. One test of a RNG that many RNGs don't perform well on is to assess the properties of k-tuples - subsequences of length k, which should be independently distributed in the k-dimensional unit hypercube. Unfortunately, linear congruential methods produce values that lie on a simple lattice in k-space, i.e., the points are not selected from q^k uniformly spaced points, where q is the the number of unique values. Instead, points often lie on parallel lines in the hypercube.

Combining generators can yield better generators. The Wichmann-Hill is an option in R and is a combination of three linear congruential generators with $a = \{171, 172, 170\}$, $m = \{30269, 30307, 30323\}$, and $u_i = (x_i/30269 + y_i/30307 + z_i/30323) \bmod 1$ where x, y, and z are generated from the three individual generators. Let's mimic the Wichmann-Hill manually:

```
RNGkind("Wichmann-Hill")
set.seed(1)
saveSeed <- .Random.seed
uFromR <- runif(10)
a <- c(171, 172, 170)
m <- c(30269, 30307, 30323)
xyz <- matrix(NA, nr = 10, nc = 3)
xyz[1, ] <- (a * saveSeed[2:4]) %% m
for( i in 2:10)
    xyz[i, ] <- (a * xyz[i-1, ]) %% m
for(i in 1:10)
    print(c(uFromR[i],sum(xyz[i, ]/m)%%1))
```

```
[1] 0.1297134 0.1297134
[1] 0.9822407 0.9822407
[1] 0.8267184 0.8267184
[1] 0.242355 0.242355
[1] 0.8568853 0.8568853
[1] 0.8408788 0.8408788
[1] 0.3421633 0.3421633
[1] 0.7062672 0.7062672
[1] 0.6212432 0.6212432
[1] 0.6537663 0.6537663
```

```
## we should be able to recover the current value of the seed
xyz[10, ]
```

```
[1] 24279 14851 10966
```

```
.Random.seed[2:4]
```

```
[1] 24279 14851 10966
```

PCG generators

Somewhat recently O'Neal (2014) proposed a new approach to using the linear congruential generator in a way that gives much better performance than the basic versions of such generators described above. This approach is now the default random number generator in numpy (see `numpy.random.default_rng()`), called the PCG-64 generator. 'PCG' stands for permutation congruential generator and encompasses a family of such generators.

The idea of the PCG approach goes like this:

- Linear congruential generators (LCG) are simple and fast, but for small values of m don't perform all that well statistically, in particular having values on a lattice as discussed above.
- Using a large value of m can actually give good statistical performance.

- Applying a technique called *permutation functions* to the state of the LCG in order to produce the output at each step (the random value returned to the user) can improve the statistical performance even further.

Instead of using relatively small values of m seen above, in the PCG approach one uses $m = 2^k$, for 'large enough' k, usually 64 or 128. It turns out that if $m = 2^k$ then the period of the bth bit of the state is 2^b where $b = 1$ is the right-most bit. Small periods are of course bad for RNG, so the bits with small period cause the LCG to not perform well. Thankfully, one simple fix is simply to discard some number of the right-most bits (this is one form of *bit shift*). Note that if one does this, the output of the RNG is based on a subset of the bits, which means that the number of unique values that can be generated is smaller than the period. This is not a problem given we start with a state with a large number of bits (64 or 128 as mentioned above).

O'Neal then goes further; instead of simply discarding bits, she proposes to either shift bits by a random amount or rotate bits by a random amount, where the random amount is determined by a small number of the initial bits. This improves the statistical performance of the generator. The choice of how to do this gives the various members of the PCG family of generators. The details are fairly complicated (the PCG paper is 50-odd pages) and not important for our purposes here.

Mersenne Twister

A commonly used generator (including in both R and Python) is the Mersenne Twister. It's the default in R and "sort of" the default in numpy (see next section for what I mean by "sort of").

The Mersenne Twister has some theoretical support, has performed reasonably on standard tests of pseudorandom numbers and has been used without evidence of serious failure. (But note that O'Neal criticizes it in her technical report.) Plus it's fast (because bitwise operations are fast). The particular Mersenne twister used has a periodicity of $2^{19937} - 1 \approx 10^{6000}$. Practically speaking this means that if we generated one random uniform per nanosecond for 10 billion years, then we would generate 10^{25} numbers, well short of the period. So we don't need to worry about the periodicity! The state (sometimes also called the seed) for the Mersenne twister is a set of 624 32-bit integers plus a position in the set, where the position is `.Random.seed[2]` in R and (I think) `np.random.get_state()[2]` in Python.

The Mersenne twister is in the class of generalized feedback shift registers (GFSR). The basic idea of a GFSR is to come up with a deterministic generator of bits (i.e., a way to generate sequences of 0s and 1s), B_i, $i = 1, 2, 3, ...$ The pseudo-random numbers are then determined as sequential subsequences of length L from $\{B_i\}$, considered as a base-2 number and dividing by 2^L to get a number in $(0, 1)$. In general the sequence of bits is generated by taking B_i to be the *exclusive or* [i.e., 0+0 = 0; 0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 0] summation of two previous bits further back in the sequence where the lengths of the lags are carefully chosen.

numpy provides access to the Mersenne Twister via the `MT19937` generator; more on this below. It looks like PCG-64 only became available as of numpy version 1.17.

The period versus the number of unique values generated

The output of the PCG-64 is 64 bits while for the Mersenne Twister the output is 32 bits. The result is that the generators generate fewer unique values than their periods. This means you could get

duplicated values in long runs, but this does not violate the comment about the periodicity of PCG-64 and Mersenne-Twister being longer than 2^{64} and 2^{32}. Why not? Because the two values after the two duplicated numbers will not be duplicates of each other – as noted previously, there is a distinction between the output presented to the user and the state of the RNG algorithm.

The seed and the state

Setting the seed picks a position in the periodic sequence of the RNG, i.e., in the state of the RNG. The state can be a single number or something much more complicated. As mentioned above, the state for the Mersenne Twister is a set of 624 32-bit integers plus a position in the set. For the PCG-64 in numpy, the state is two numbers – the actual state and the increment (`c` above). This means that when the user passes a single number as the seed, there needs to be a procedure that deterministically sets the state based on that single number seed. The details of this are not usually well-documented or viewable by the user.

Ideally, nearby seeds generally should not correspond to getting sequences from the RNG stream that are closer to each other than far away seeds. According to Gentle (CS, p. 327) the input to `set.seed()` in R should be an integer, $i \in \{0, \ldots, 1023\}$, and each of these 1024 values produces positions in the RNG sequence that are "far away" from each other. I don't see any mention of this in the R documentation for `set.seed()` and furthermore, you can pass integers larger than 1023 to `set.seed()`, so I'm not sure how much to trust Gentle's claim. More on generating parallel streams of random numbers below.

When one invokes a RNG without a seed, RNG implementations generally have a method for choosing a seed (often based on the system clock). The numpy documentation says that it "mixes sources of entropy in a reproducible way" to do this.

Generators should give you the same sequence of random numbers, starting at a given seed, whether you ask for a bunch of numbers at once, or sequentially ask for individual numbers.

Additional notes

There have been some attempts to generate truly random numbers based on physical randomness. One that is based on quantum physics is http://www.idquantique.com/true-random-number-generator/quantis-usb-pcie-pci.html. Another approach is based on lava lamps!

RNG in Python

Choosing a generator

In numpy, the *default_rng* RNG is PCG-64. It has a period of 2^{128} and supports advancing an arbitrary number of steps, as well as 2^{127} streams (both useful for generating random numbers when parallelizing). The state of the PCG-64 RNG is represented by two 128-bit unsigned integers, one the actual state and one the value of c (the *increment*).

However, while the *default* is PCG-64, simply using the functions available via `np.random` to generate random numbers seems to actually use the Mersenne Twister, so the meaning of *default* is tricky.

I think that this text from `help(np.random)` explains what is going on:

```
Legacy
------

For backwards compatibility with previous versions of numpy before 1.17, the
various aliases to the global `RandomState` methods are left alone and do not
use the new `Generator` API.
```

We can change to a specific RNG using syntax (the `Generator` API) like this:

```
rng = np.random.Generator(np.random.MT19937(seed = 1))  # Mersenne Twister
rng = np.random.Generator(np.random.PCG64(seed = 1))    # PCG-64
```

but below note that there is a simpler way to change to the PCG-64.

Then to use that generator when doing operations that generate random numbers, we need to use methods accessed via the `Generator` object (`rng` here):

```
rng.random.normal(size = 3)     # Now generate based on chosen generator.
## np.random.normal(size = 3)   # This will NOT use the chosen generator.
```

In R, the default RNG is the Mersenne twister (`?RNGkind`).

Using the Mersenne Twister

If we simply start using numpy or scipy to generate random numbers, we'll be using the Mersenne Twister. I believe this is what the documentation mentioned above means by "aliases to the global `RandomState` methods".

We get replicability by setting the seed to a specific value at the beginning of our simulation. We can then set the seed to that same value when we want to replicate the simulation.

```
np.random.seed(1)
np.random.normal(size = 5)
```

```
array([ 1.62434536, -0.61175641, -0.52817175, -1.07296862,  0.86540763])
```

```
np.random.seed(1)
np.random.normal(size = 5)
```

```
array([ 1.62434536, -0.61175641, -0.52817175, -1.07296862,  0.86540763])
```

We can also save the state of the RNG and pick up where we left off. So this code will pick where you had left off, ignoring what happened in between saving to `saved_state` and resetting.

```
np.random.seed(1)
np.random.normal(size = 5)
```

```
array([ 1.62434536, -0.61175641, -0.52817175, -1.07296862,  0.86540763])
```

```
saved_state = np.random.get_state()
np.random.normal(size = 5)
```

```
array([-2.3015387 ,  1.74481176, -0.7612069 ,  0.3190391 , -0.24937038])
```

Now we'll do some arbitrary work with random numbers, and see that if we use the saved state we can pick up where we left off above.

```
tmp = np.random.choice(np.arange(1, 51), size=2000, replace=True) # arbitrary work

## Restore the state.
np.random.set_state(saved_state)
np.random.normal(size = 5)
```

```
array([-2.3015387 ,  1.74481176, -0.7612069 ,  0.3190391 , -0.24937038])
```

If we look at `saved_state`, we can confirm it actually corresponds to the Mersenne Twister.

Using PCG64

To use the PCG-64, we need to explicitly create and make use of the `Generator` object (`rng` here), which is the new numpy approach to handling RNG.

We set the seed when setting up the generator via `np.random.default_rng(seed)` (or `np.random.Generator(np.random.` `= 1))`).

```
rng = np.random.default_rng(seed = 1)
rng.normal(size = 5)
```

```
array([ 0.34558419,  0.82161814,  0.33043708, -1.30315723,  0.90535587])
```

```
rng = np.random.default_rng(seed = 1)
rng.normal(size = 5)
```

```
array([ 0.34558419,  0.82161814,  0.33043708, -1.30315723,  0.90535587])
```

```
saved_state = rng.bit_generator.state
rng.normal(size = 5)
```

```
array([ 0.44637457, -0.53695324,  0.5811181 ,  0.3645724 ,  0.2941325 ])
```

```
tmp = rng.choice(np.arange(1, 51), size=2000, replace=True)
rng.bit_generator.state = saved_state
rng.normal(size = 5)
```

```
array([ 0.44637457, -0.53695324,  0.5811181 ,  0.3645724 ,  0.2941325 ])
```

```
saved_state
```

```
{'bit_generator': 'PCG64', 'state': {'state': 216676376075457487203159048251690499413, 'inc': 1942902
```

```
saved_state['state']['state']   # actual state
```

```
216676376075457487203159048251690499413
```

```
saved_state['state']['inc']     # increment ('c')
```

```
194290289479364712180083596243593368443
```

`saved_state` contains the actual state and the value of `c`, the increment.

Question: how many bits does `saved_state['state']['state']` correspond to?

RNG in parallel

We can generally rely on the RNG in Python and R to give reasonable set of pseudo-random values. One time when we want to think harder is when doing work with RNG in parallel on multiple processors. The worst thing that could happen is that one sets things up in such a way that every process is using the same sequence of random numbers. This could happen if you mistakenly set the same seed in each process, e.g., using `np.random.seed(1)` on every process. Numpy now provides some nice functionality for parallel RNG, with more details given in the SCF parallelization tutorial.

5. Generating random variables

There are a variety of methods for generating from common distributions (normal, gamma, beta, Poisson, t, etc.). Since these tend to be built into Python and R and presumably use good algorithms, we won't go into them. A variety of statistical computing and Monte Carlo books describe the various methods. Many are built on the relationships between different distributions - e.g., a beta random variable (RV) can be generated from two gamma RVs.

Multivariate distributions

The `mvtnorm` package supplies code for working with the density and CDF of multivariate normal and t distributions.

To generate a multivariate normal, in Unit 10, we'll see the standard method based on the Cholesky decomposition:

```
L = np.linalg.cholesky(covMat) # L is lower-triangular
x = L @ np.random.normal(size = covMat.shape[0])
```

Side note: for a singular covariance matrix we can use the Cholesky with pivoting, setting as many rows to zero as the rank deficiency. Then when we generate the multivariate normals, they respect the constraints implicit in the rank deficiency. However, you'll need to reorder the resulting vector because of the reordering involved in the pivoted Cholesky.

Inverse CDF

Most of you know the inverse CDF method. To generate $X \sim F$ where F is a CDF and is an invertible function, first generate $Z \sim \mathcal{U}(0,1)$, then $x = F^{-1}(z)$. For discrete CDFs, one can work with a discretized version. For multivariate distributions, one can work with a univariate marginal and then a sequence of univariate conditionals: $f(x_1)f(x_2|x_1)\cdots f(x_k|x_{k-1},\ldots,x_1)$, when the distribution allows this analytic decomposition.

Rejection sampling

The basic idea of rejection sampling (RS) relies on the introduction of an auxiliary variable, u. Suppose $X \sim F$. Then we can write $f(x) = \int_0^{f(x)} du$. Thus f is the marginal density of X in the joint density, $(X,U) \sim \mathcal{U}\{(x,u) : 0 < u < f(x)\}$. Now we'd like to use this in a way that relies only on evaluating $f(x)$ without having to draw from f.

To implement this we draw from a larger set and then only keep draws for which $u < f(x)$. We choose a density, g, that is easy to draw from and that can *majorize* f, which means there exists a constant c s.t. , $cg(x) \geq f(x)\ \forall x$. In other words we have that $cg(x)$ is an upper envelope for $f(x)$. The algorithm is

1. generate $x \sim g$
2. generate $u \sim \mathcal{U}(0,1)$
3. if $u \leq f(x)/cg(x)$ then use x; otherwise go back to step 1

The intuition here is graphical: we generate from under a curve that is always above $f(x)$ and accept only when u puts us under $f(x)$ relative to the majorizing density. A key here is that the majorizing density have fatter tails than the density of interest, so that the constant c can exist. So we could use a t to generate from a normal but not the reverse. We'd like c to be small to reduce the number of rejections because it turns out that $\frac{1}{c} = \frac{\int f(x)dx}{\int cg(x)dx}$ is the acceptance probability. This approach works in principle for multivariate densities but as the dimension increases, the proportion of rejections grows, because more of the volume under $cg(x)$ is above $f(x)$.

If f is costly to evaluate, we can sometimes reduce calculation using a lower bound on f. In this case we accept if $u \leq f_{\text{low}}(y)/cg_Y(y)$. If it is not, then we need to evaluate the ratio in the usual rejection sampling algorithm. This is called squeezing.

One example of RS is to sample from a truncated normal. Of course we can just sample from the normal and then reject, but this can be inefficient, particularly if the truncation is far in the tail (a case in which inverse CDF suffers from numerical difficulties). Suppose the truncation point is greater than zero. Working with the standardized version of the normal, you can use an translated exponential with lower end point equal to the truncation point as the majorizing density (Robert 1995; Statistics and Computing). For truncation less than zero, just make the values negative.

Adaptive rejection sampling (optional)

The difficulty of RS is finding a good enveloping function. Adaptive rejection sampling refines the envelope as the draws occur, in the case of a continuous, differentiable, log-concave density. The basic idea considers the log of the density and involves using tangents or secants to define an upper envelope and secants to define a lower envelope for a set of points in the support of the distribution. The result is that we have piecewise exponentials (since we are exponentiating from straight lines on the log scale) as the bounds. We can sample from the upper envelope based on sampling from a discrete distribution and then the appropriate exponential. The lower envelope is used for squeezing. We add points to the set that defines the envelopes whenever we accept a point that requires us to evaluate $f(x)$ (the points that are accepted based on squeezing are not added to the set).

Importance sampling

Importance sampling (IS) allows us to estimate expected values. It's an extension of the simple Monte Carlo sampling we saw at the beginning of the unit, with some commonalities with rejection sampling.

$$\phi = E_f(h(Y)) = \int h(y)\frac{f(y)}{g(y)}g(y)dy$$

so $\hat{\phi} = \frac{1}{m}\sum_i h(y_i)\frac{f(y_i)}{g(y_i)}$ for y_i drawn from $g(y)$, where $w_i = f(y_i)/g(y_i)$ act as weights. (Often in Bayesian contexts, we know $f(y)$ only up to a normalizing constant. In this case we need to use $w_i^* = w_i/\sum_j w_j$.

Here we don't require the majorizing property, just that the densities have common support, but things can be badly behaved if we sample from a density with lighter tails than the density of interest. So in general we want g to have heavier tails. More specifically for a low variance estimator of ϕ, we would want that $f(y_i)/g(y_i)$ is large only when $h(y_i)$ is very small, to avoid having overly influential points.

This suggests we can reduce variance in an IS context by oversampling y for which $h(y)$ is large and undersampling when it is small, since $\text{Var}(\hat{\phi}) = \frac{1}{m}\text{Var}(h(Y)\frac{f(Y)}{g(Y)})$. An example is that if h is an indicator function that is 1 only for rare events, we should oversample rare events and then the IS estimator corrects for the oversampling.

What if we actually want a sample from f as opposed to estimating the expected value above? We can draw y from the unweighted sample, $\{y_i\}$, with weights $\{w_i\}$. This is called sampling importance resampling (SIR).

Ratio of uniforms (optional)

If U and V are uniform in $C = \{(u,v) : 0 \leq u \leq \sqrt{f(v/u)}$ then $X = V/U$ has density proportion to f. The basic algorithm is to choose a rectangle that encloses C and sample until we find $u \leq f(v/u)$. Then we use $x = v/u$ as our RV. The larger region enclosing C is the majorizing region and a simple approach (if $f(x)$and $x^2 f(x)$ are bounded in C) is to choose the rectangle, $0 \leq u \leq \sup_x \sqrt{f(x)}$, $\inf_x x\sqrt{f(x)} \leq v \leq \sup_x x\sqrt{f(x)}$.

One can also consider truncating the rectangular region, depending on the features of f.

Monahan recommends the ratio of uniforms, particularly a version for discrete distributions (p. 323 of the 2nd edition).