R computing for Business Data Analytics

Instructor: Dr. Howard Hao-Chun Chuang
Assistant Professor, Department of Management Information Systems
National Chengchi University, Taipei, Taiwan 116
chuang@nccu.edu.tw

Last Revised: October 2014

5.1 Estimation

- Statistical inference

Suppose some data on students' attrition rate in NCCU are collected. One can use the data (i.e., *sample*) to draw conclusions/make statements about students' attrition rate in Taiwan (i.e., *population*). Infer the population using the sample is what statistical inference is about. An important aspect of statistical inference is that it involves *uncertainty*. This is because the collected data only represent a subset of the population. Probability models are the tool used to quantify the amount of uncertainty in an inference.

In discussing statistical inference, we will limit ourselves to the following case:

1. We have a random sample $x_1, x_2, \ldots, x_n$ and we tend to believe that the data comes from some distribution *f* that is referred to as the *population*.
2. We specify *a priori* what type of distribution we will fit (e.g., normal distribution). According to our assumption, the functional form of *f* is known except for parameters θ (μ and σ^2 in the case of the normal distribution).
3. We use the data $x_1, x_2, \ldots, x_n$ to estimate the unknown parameters θ. Specifically, we want to choose the estimates $\hat{\theta}$ that best fit the observed data.

We can further categorize statistical inference into the following five types:

1. *Point estimation*: Computing a value from the data that is thought to be near the TRUE (but unknown) value of the parameter.
2. *Interval estimation*: Computing from the data a range of plausible values for the unknown parameter.
3. *Goodness of fit*: Using the data to assess whether the assumed distribution is a reasonable model.

4. *Hypothesis testing*: Using the data to choose between two statements (*H0* & *Ha*) concerning the unknown parameter (you still remember anything about it?).
5. *Prediction*: Using the data to predict future values coming from the population.

In this lecture, we focus on *point estimation* (1.) and *interval estimation* (2.). That is, we cover a range of approaches to finding the single best estimate of a parameter, given some data and a model, as well as approaches to determining a range of possible values that a parameter could take.

While choosing the distribution that best fits an observed sample is an important practical task, here we do not cover this challenge – *goodness of fit* (3.). Nevertheless, for your own education, please go over a short and sweet tutorial – *fitting distributions with R* – written by Vito Ricci (http://cran.r-project.org/doc/contrib/Ricci-distributions-en.pdf).

The last two types of inference – *hypothesis testing* (4.) and *prediction* (5.) – will be left for the second half of the semester.

- Point estimation

We start with finding values for the parameters that provide the best fit between the proposed model *f* and the collected data. Once found, those values are called *parameter estimates.* So, how can we find the best estimates? The theoretically best approach – *maximum likelihood estimation* – chooses the parameters to maximize a function of the data called the likelihood function, which measures how likely it is to observe our given sample. Let's define several terms before we see the example.

If $x_1,\ldots,x_n$ is a random sample from a distribution $f(x\,|\,\boldsymbol{\theta})$, then the *likelihood function* is

$$\text{lik}(\boldsymbol{\theta}) = \prod_{i=1}^{n} f(x_i\,|\,\boldsymbol{\theta})$$

The *maximum likelihood estimates* are values such that

$$\text{lik}(\hat{\boldsymbol{\theta}}_{MLE}) \geq \text{lik}(\boldsymbol{\theta}) \;\; \forall \boldsymbol{\theta}$$

Often we use the *log-likelihood function* for the sake of computation:

$$\text{loglik}(\boldsymbol{\theta}) = \log(\prod_{i=1}^{n} f(x_i\,|\,\boldsymbol{\theta})) = \sum_{i=1}^{n} \log(f(x_i\,|\,\boldsymbol{\theta}))$$

Example: Suppose $x_1, x_2, \ldots, x_n$ is a random sample from the gamma distribution:

$$f(x) = \frac{1}{\Gamma(\alpha)} \lambda^{\alpha} x^{\alpha-1} e^{-\lambda x} \text{ for } x \geq 0 \text{ and } \alpha, \lambda > 0$$

The log-likelihood function is

$$\begin{aligned} \text{loglik}(\alpha, \lambda) &= \log\left(\prod_{i=1}^{n} \frac{1}{\Gamma(\alpha)} \lambda^{\alpha} x_i^{\alpha-1} e^{-\lambda x_i}\right) \\ &= n\alpha \log(\lambda) + (\alpha - 1)\sum_{i=1}^{n} \log(x_i) - \lambda \sum_{i=1}^{n} x_i - n \log(\Gamma(\alpha)) \end{aligned}$$

From calculus we know that to find α and λ that maximize the function, we need to solve

$$\frac{\partial \text{loglik}(\alpha, \lambda)}{\partial \alpha} = n \log(\lambda) + \sum_{i=1}^{n} \log(x_i) - n \frac{\Gamma'(\alpha)}{\Gamma(\alpha)} = 0$$

$$\frac{\partial \text{loglik}(\alpha, \lambda)}{\partial \lambda} = \frac{n\alpha}{\lambda} - \sum_{i=1}^{n} x_i = 0$$

We solve the second equation (don't worry, I WON'T ask you to do this math in the midterm)

$$\hat{\lambda}_{MLE} = \frac{n\hat{\alpha}_{MLE}}{\sum_{i=1}^{n} x_i} = \frac{\hat{\alpha}_{MLE}}{\bar{x}}$$

We substitute this into the first equation and get

$$n \log(\hat{\alpha}_{MLE}) - n \log(\bar{x}) + \sum_{i=1}^{n} \log(x_i) - n \frac{\Gamma'(\hat{\alpha}_{MLE})}{\Gamma(\hat{\alpha}_{MLE})} = 0$$

This cannot be solved in closed form. An iterative numerical method of solving the equation must be used. Instead of this, I will show you a brute-force approach in the coming example.

Example: Kew rainfall (adapted from the Jones, Maillardet, and Robinson 2009 book)

The rainfall at Kew Gardens in London has been systematically measured since 1697 to 1999. First, take a look at the frequency distribution of the total July rainfall in millimeters

```
> kew=read.table("…/kew.txt", col.names=c("year", "jan", "feb", "mar", "apr", "may",
"jun", "jul", "aug", "sep", "oct", "nov", "dec"))
> kew[ , 2:13] = kew[ , 2:13]/10  #transform units from 0.1mm to mm
> hist(kew[ ,8])  #see the frequency distribution of July rainfall
```

Does this look like the normal distribution?

Let's say we expect the gamma distribution $X \sim \Gamma(\lambda, \alpha)$ to be a good fit to the *aggregated* rainfall data. Even though it is computationally intensive, *Maximum likelihood estimation* is

usually preferred as it has nice theoretic properties. In our course we will restrict ourselves to the mechanics, for a theoretical discussion please read up on *mathematical statistics*. Below is how you can do it computationally.

```
> jul.rain=kew[ , 8]
> gammaloglik=function(par){
>                   loglik=0
>                   for(i in 1: length(jul.rain)){
>                             loglik.i=ifelse(log(dgamma(jul.rain[i], par[1], par[2]))==-Inf,
>                                       0, log(dgamma(jul.rain[i], par[1], par[2])))
>                             loglik=loglik+loglik.i
>                   }
>                  -loglik
>}
```

Here I return the *negative* loglikelihood because the optimizer will MINIMIZE the function. Also I specify an *ifelse* condition to avoid numerical issues of log(0).

```
> kew.jul.mean= mean(kew[ , 8]) #sample mean
> kew.jul.var=var(kew[ , 8]) #sample variance
> mle=nlminb(c(1, 1), gammaloglik, lower=c(1e-5,1e-5))
```

nlminb is nice for constrained optimization. Others include *optimize*, *optim*, *nlm*, etc.

```
> alpha.mle=mle$par[1]
> lambda.mle=mle$par[2]
```

Now see how good the parameter estimates are

```
> hist(kew[ ,8], breaks=20, freq=FALSE, xlab= "rainfall(mm)", ylab= "density", main=
" July rainfall at kew, 1697 to 1999") #plot the density using our data
> t=seq(0, max(kew[,8]), 0.5)
> lines(t, dgamma(t, alpha.mle, lambda.mle), col='red')
```

The maximum likelihood method provides a decent fit to the data. An alternative approach is to solve the equation below in *R* using any root-finding technique (e.g., the Newton method in lecture 2). How do we do it?

$$n\log(\hat{\alpha}_{MLE}) - n\log(\bar{x}) + \sum_{i=1}^{n}\log(x_i) - n\frac{\Gamma'(\hat{\alpha}_{MLE})}{\Gamma(\hat{\alpha}_{MLE})} = 0$$

```
> jul.rain=jul.rain[jul.rain>0]
> n=length(jul.rain)
> f=function(a){
> alpha=a
> return(n*log(alpha)-n*log(mean(jul.rain))+sum(log(jul.rain))
> n*numericDeriv(gamma(alpha),"alpha")/gamma(alpha))
> }
```

Apply the *uniroot* function

```
> alpha.root=uniroot(f,c(1e-5, 10))$root
> lambd.root=alphamle/mean(jul.rain)
```

Are the roots better than what we found earlier?

```
> gammaloglik(c(alpha.root, lambd.root))
```

So far we have mainly focused on obtaining parameter estimates given data and assumed models. A big question to ask is: What is the right model to start with? The answer is, it depends (you probably think the answer is not helping at all). The truth is, it really depends on the type of data (e.g., discrete or continuous), the look of the frequency distribution (i.e., histogram), the skewness, the number of zeros, the dispersion (variance-to-mean ratio), etc. As I said, DO NOT fit everything with the normal distribution, despite those nice properties it has. You will be more proficient in specifying the right probability model *f* when you have more experiences in analyzing data (learning by doing, period!). For the time being, do not let yourself baffled by the modeling choice. Hopefully, by the time we finish all lectures, you will have a much better sense of choosing the distribution for collected data.

- Interval estimation

You must have heard of "confidence interval" multiple times in your statistics course. You may wonder why statistics professors make a big fuss about confidence interval (CI). CI is so important because it reflects *the precision of our parameter estimates* (which are obtained by method of moments, maximum likelihood, etc.).

For example, a famous result in mathematical statistics is $\bar{x} \xrightarrow{P} E[X]$ (according to the Weak Law of Large Numbers). However, how fast does it converge? For an estimate (e.g., $\bar{x}$)

to be really useful, we need to know how precise the estimate is. Interval estimation allows us to assess the precision of parameter estimation.

Suppose $x_1, x_2, \ldots, x_n$ are an iid sample with mean μ and variance σ^2. Let $P(Z < z_\alpha) = \alpha$ (pnorm(z_α , 0, 1)= α ; z_α =qnorm(α , 0, 1)). Then a 100(1- α)% CI for μ is

$$(\bar{x} - z_{1-\alpha/2}\frac{s_x}{\sqrt{n}}, \bar{x} + z_{1-\alpha/2}\frac{s_x}{\sqrt{n}})$$

where $z_{1-\alpha/2} = z_{\alpha/2}$ $\because$ the normal distribution is symmetric about 0

Example: Derive a 95% CI for simulated Poisson data

```
> qnorm((1-0.05/2), 0, 1) #it should return approximately 1.96
> set.seed(100) #set an arbitrary simulation seed
> n=2000 #the number of simulated data points
> la=2 #the assumed population parameter
> x=rpois(n, la)
> xbar=mean(x)
> s=sd(x)
> L=xbar-1.96*s/sqrt(n)
> U=xbar+1.96*s/sqrt(n)
> cat("estimate is", xbar, "\n")
> cat("95% CI is (", L, ",", U, ")\n", sep= "")
```

Up to this point, I hope you have realized calculating some metrics of interest from the data in *R* is NOT difficult at all. All you have to do is understand the meaning of the formula in the book/paper. You should have confidence in doing the computation by yourself instead of relying commercial statistical software packages.

Many formulas have been derived to estimate CIs of various parameter estimates. It is simply impossible for us to cover them all. What really matters is that you understand the *underlying meaning* of CI and why data analysts care a lot about interval estimation. The example hopes to plant the idea of CI in your mind deeply (*Inception*), so that you will be able to interpret CI correctly in the future.

Let's explore how fast $\bar{x} \xrightarrow{P} E[X]$ for our simulated Poisson data.

```
> set.seed(100)
> n=2000
```

```
> la=2
> plot(c(1, n), c(la-sqrt(la), la+sqrt(la)), type="n", xlab= "sample size k", ylab= "k point average")
> abline(h=la)
> lines(1:n, la+1.96*sqrt(la/1:n), lty=2, col= 'red')
> lines(1:n, la-1.96*sqrt(la/1:n), lty=2, col= 'red')
> x=rpois(n, la)
> xbar=cumsum(x)/1:n #what are we doing here?
> lines(1:n, xbar, type= "l", col= 'blue')
```

As you probably can see from the figure, as the sample size *k* increases, the sample mean $\bar{x}$ converges to the theoretical mean. *Simulation* allows us to make this assessment easily. Let's take a deep dive into simulation in the next section.

5.2 Simulation

5.2.1 Ideas and experiments

- Monte-Carlo simulation

Simulation provides a straightforward way of approximating probabilities and statistics (e.g., $\bar{x}$). One can simulate a random experiment many times, and approximate a probability of an outcome by the relative frequency of the outcome in the simulation experiment. Since World War II, the use of simulation to better understand *stochastic/probabilistic/uncertain patterns* is called Monte-Carlo simulation (in homage to the famous Monte-Carlo casino). In section 6.2.1, we focus on two *R* functions – sample and replicate – that greatly simplify simulation experiments.

- Simulating a game of chance

Snoopy and Charlie Brown are playing a coin-tossing game using a FAIR coin. Each time a head comes up, Snoopy wins a dollar from Charlie Brown. Otherwise Snoopy loses a dollar to Charlie Brown. Snoopy starts with zero dollars and decides to play the game for 50 tosses.

```
> win=sample(c(-1, 1), size=50, replace=TRUE)
> cum.win=cumsum(win)
> cum.win
```

Suppose Snoopy and Charlie Brown play once a week for a whole month. We wish to see what happens to Snoopy's pocket money by the end of the month.

```
> par(mfrow=c(2, 2))
> for(j in 1:4){
>     win=sample(c(-1, 1), size=50, replace=TRUE)
>     plot(cumsum(win), type='l', ylim=c(-15, 15))
>     abline(h=0)
> }
```

Despite how famous and popular Snoopy is, the figure tells us that Snoopy is no different from normal people who sometimes win and sometimes lose.

We can future write a function to consider Snoopy's fortune F at the end of each game.

```
> snoopy=function(n=50){ win=sample(c(-1, 1), size=n, replace=T)
> sum(win)}
```

Repeat the game 1000 times

```
> F=replicate(1000, snoopy())
```

Summarize the simulation experiment

```
> table(F)
> plot(table(F))
```

What is the chance for Snoopy to break even (i.e., F=0) in the game?

```
> sum(F==0)/1000
```

This *simulated probability*, should be close to the *theoretical probabil*ty. That is, Snoopy breaks even if there are exactly 25 heads in the binomial experiment of 50 trials.

```
> dbinom(25, size=50, prob=0.5)
```

Actually, we can modify the function to learn about new statistics from simulation.

```
> snoopy=function(n=50){
> win=sample(c(-1, 1), size=n, replace=T)
> cum.win=cumsum(win)
> c(F=sum(win), L=sum(cum.win>0), M=max(cum.win))}
```

Simulate the game 1000 times again.

```
> S=replicate(1000, snoopy())
```

A simulation like this allows us to answer questions that are difficult to address analytically.

(a) What is the likely number of tosses where Snoopy will be in the lead?

(b) What will be the value of Snoopy's highest fortune during the game?

(c) How likely Snoopy will have a maximum cumulative earning > 10?

The answer to (a) is just

```
> mean(S["L", ])
> c(quantile(S["L", ], 0.025), quantile(S["L", ], 0.975)) #what is this?
```

The answer to (b) is just

```
> mean(S["M", ])
```

Finally, the answer to (c) is

```
> sum(S["M", ]>10)/1000
```

Through this simple example, I hope you have started to realize the power of simulation. Let me give you another example of how simulation facilitates our decision-making.

Suppose 7-11 announces a new set of Hello kitty magnets. The set has 101 unique magnets in total. You can either buy *random* magnets (some of which could be repetitive) from 7-11 at cost.r=5 per magnet, or buy non-repetitive magnets from the dealer at cost.n=25 per magnet. The stochastic total cost will be

$$COST=cost.r*npurchased+cost.d*nmissed$$

Your goal is to collect the whole set, and you want to know how much you will have to pay.

```
> collect=function(npurchased){
> cost.r=5
> cost.d=25
> cardsbought=sample(1:101, size=npurchased, replace=TRUE)
> nhit=length(unique(cardsbought))
> nmissed=101-nhit
> cost.r*npurchased+cost.d*nmissed
> }
```

You decide to purchase 500 cards from 7-11 (good luck). The distribution of costs will be

```
> costs=replicate(1000, collect(500)) #simulate the purchase 1000 times
> summary(costs)
```

If you just buy the set from the dealer, the cost is 25*101=2525 dollars. What is your chance of spending <= 2525 if you buy 500 cards randomly from 7-11 first?

```
> sum(costs<=2525)/1000
```

5.2.2 Algorithms and examples

Most stochastic simulations have the same structure:

1. Identify a random variable of interest *X*, which has distribution function *f*.
2. Generate an iid sample $x_1, x_2, \ldots, x_n$ with the same distribution *f*.
3. Estimate E[*X*] using $\bar{x}$ and assess the precision of $\bar{x}$ using simulation-based CI.

It turns out that all random variables can be simulated by first simulating iid Uniform(0, 1) random samples. That is where we start.

- Generating pseudo-random numbers

We cannot generate *truly random numbers* on a computer. Instead, we generate *pseudo-random numbers* that have the appearance of random numbers, but are entirely deterministic. As such, any experiment performed using pseudo-random numbers can be repeated exactly. Below introduces the first reasonable and classical pseudo-random number generator – *congruential generators*.

Given an initial number $x_0 \in \{0,1,\ldots,m-1\}$ and two numbers *A* and *B*, we define a sequence of numbers x_n and a sequence of u_n that is nearly identical to iid *U*(0, 1) random variates by

$$x_{n+1} = (Ax_n + B) \bmod m, \ n = 0,1,...$$
$$u_n = x_n / m$$

The number x_0 is called the *seed*. If you know the seed, *m*, *A*, and *B*, then you can reproduce the whole sequence exactly. This allows you to reproduce simulation experiments.

For example, if we take *A*=3, *B*=0, m=7, and x_0=2, then

$$x_1 = 3 \times 2 (\bmod 7) = 6, \ u_1 = 0.857$$
$$x_2 = 3 \times 6 (\bmod 7) = 4, \ u_2 = 0.571$$
$$x_3 = 3 \times 4 (\bmod 7) = 5, \ u_3 = 0.714$$
$$x_4 = 3 \times 5 (\bmod 7) = 1, \ u_4 = 0.143$$
$$x_5 = 3 \times 1 (\bmod 7) = 3 \ u_5 = 0.429$$
$$x_6 = 3 \times 3 (\bmod 7) = 2, \ u_6 = 0.286$$
$$x_7 = 3 \times 2 (\bmod 7) = 6, \ u_7 = 0.857$$

Here x_1=x_7 and the cycle is too short. Clearly, the cycle length cannot be greater than *m*. An example of good congruential generator is $m=2^{32}$, *A*=1,664,525, and *B*=1,013,904,223.

Note that *R* does not use the congruential generatror. But we can still track those pseudo-random numbers. See the example below.

```
> set.seed(42)
> runif(2)

> RNG.state=.Random.seed
> runif(2)

> set.seed(42)
> runif(2)

> .Random.seed=RNG.state
> runif(2)
```

- Simulating discrete random variables

Earlier we have simulated Poisson random variates using *rpois*( ). As a matter of fact, we do not have to rely on those built-in *rxxx*( ) in *R* because given simulated Uniform(0, 1) random numbers, we can simulate all the others using the *inverse transformation* method.

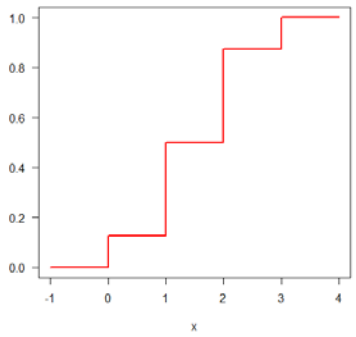
Let X be a discrete random variable with pdf $f(x) = P(X = x)$ and cdf $F(x) = P(X \le x)$.

The inverse transformation says that, given U~Uniform(0, 1)

```
> x=0
> while(F(x)<U){ x=x+1}
```

Let's take a look at X~binomial(n=3, p=0.5) and its cdf F(x)

```
> x=-1:4
> plot(x, pbinom(x,3,0.5), type='s', col='red', lwd=2)
```



For us to simulate X~binomial(n=3, p=0.5), all we have to do is to map simulated U~U(0, 1) to the corresponding x=0, 1, 2, 3 on the x-axis.

Of course you can easily simulate random binomial numbers using *rbinom*(n, size, prob). But I really want you to have a solid understanding of how those numbers are produced. You will be asked to program this algorithm for a X~binomial(n, p) in your homework.

- Simulating continuous random variables

The idea of inverse transformation is the same in the continuous case. Let X be a exponential random variable with pdf $f(x) = \lambda e^{-\lambda x}$ and cdf $F(x) = 1 - e^{-\lambda x}$. For a given U~Uniform(0, 1), if we set U=F(x) and solve for x, then we have

$$x = -\log(1-U)/\lambda$$

The formula simulates random numbers from exp(λ) given U and λ. In *R*, you could just use *rexp*(n, rate) to perform simulation.

Example: A bank has a single teller who is facing a queue of 10 customers. The time for each customer to be served is exponentially distributed with rate 3/minute. We can simulate the service times for the 10 customers.

```
> servicetimes=rexp(10, rate=3)
> sum(servicetimes)
```

As you have learnt how to simulate X~binomial(n, p), how do you simulate X~Bernoulli(p)?

```
> ?
```

A quick and dirty alternative to simulate n Bernoulli random variates would be

```
> guesses=runif(n)
> correct=(guesses<=p)
> correct
```

The last piece I want to discuss in section 6.2 is simulating the normal random variable given its importance and popularity. If $Z \sim N(0,1)$ then $\mu + \sigma Z \sim N(\mu, \sigma^2)$, hence it is sufficient to simulate standard normal random variates. The first way to simulate Z~N(0, 1) harnesses on the *central limit theorem*. Recall that U~U(0, 1), E[U]=1/2, and Var[U]=1/12. So, if $u_1,\ldots,u_{12}$ are iid U(0, 1) random numbers, then

$$Z = (\sum_{i=1}^{12} u_i) - 6 \xrightarrow{CLT} N(0,1)$$

The algorithm works quite well but it takes 12 uniform random numbers to deliver 1 standard normal random number. We have something better – *Box Muller* algorithm.

1. Simulate u_1 and u_2 from $U(0, 1)$
2. Set $\Theta = 2\pi u_1$ and $R = \sqrt{-2\log(u_2)}$
3. Return $X = R\cos(\Theta)$ and $Y = R\sin(\Theta)$

There are more algorithms for you to explore (if you'd like to). Before the end I simply want to show you a couple of examples in *R*.

Example: Simulate 10 independent normal variates with mean of -3 and a variance of 0.25.

```
> rnorm(10, -3, sd=sqrt(0.25)) #be careful about the sd argument in rnorm().
```

Example: Simulate x from the standard normal distribution, conditional on $0 \le x \le 3$.

```
> x=rnorm(10000)
> x=x[ (0<=x) & (x<=3)]
> hist(x, probability=TRUE)
```

This looks like a truncated/half normal distribution, right?

```
> lines(density(x, from=0, to=3), col= 'red', lty=2, lwd=2)
> curve(dnorm(x), add=TRUE)
```

5.3 Monte-Carlo integration

- Simple integration

My guess is most of you do not like to do integration analytically by-hand (I DON'T either). Monte-Carlo simulation is a quick and dirty (yet valid) way to *approximate* those integrals. This simulation mechanic turns out to be extremely useful when we encounter some difficult integrals in Bayesian data analysis, physics, etc. So, I feel it is imperative for you to have an intuitive understanding of Monte-Carlo integration.

Suppose $g(x)$ is any function that is integrable on the interval $[a, b]$, the integral

$$\int_a^b g(x)dx$$

Let U~uniform(a, b) and $u_1, u_2, \ldots, u_n$ be an independent random sample on the interval $[a, b]$. They have density $f(u)=1/(b-a)$ on that interval. Then

$$E[g(U)] = \int_a^b g(u)f(u)du$$

$$\Rightarrow \int_a^b g(u)du = \int_a^b g(u)f(u)du \times \frac{1}{f(u)} = \int_a^b g(u)\frac{1}{b-a}du \times (b-a) \approx \frac{1}{n}\sum_{i=1}^{n} g(u_i)(b-a)$$

Proving the validity of above approximation requires advanced calculus. For us, knowing how to apply the concept is good enough.

Example: Approximate the integral $\int_0^1 x^4 dx$

```
> u=runif(n=100000, 0, 1)
> mean(u^4)*(1-0)
```

The exact answer is 0.2. How good is our approximation?

Example: Approximate the integral $\int_2^5 \sin(x)dx$

- Multiple integration

Let V~uniform(c, d) and v_1, v_2,…,v_n be an independent random sample on the interval [c, d] Suppose $g(x, y)$ is an integrable function of two variables x and y. We can approximate the integral $\int_c^d \int_a^b g(x,y)dxdy$ by generating uniform pseudo-random variates, computing $g(u_i, v_i)$ for each one, and taking the average.

Example: Approximate the integral $\int_3^{10} \int_1^7 \sin(x-y)dxdy$

```
> u=runif(100000, 1, 7)
> v=runif(100000, 3, 10)
> mean(sin(u-v))*((7-1)*(10-3))
```

The uniform density is by no means the only density that can be used in Monte-Carlo integration. If the density of X is $f(x)$, $E[g(X)/f(X)] = \int [g(x)/f(x)]f(x)dx = \int g(x)dx$. So, we can approximate the latter by sample averages of $g(X)/f(X)$

Example: To approximate the integral $\int_1^\infty \exp(-x^2)dx$, write it as $\int_0^\infty \exp(-(x+1)^2)dx$, and use an exponential distribution for X:

```
> x=rexp(100000)
> mean(exp(-(x+1)^2) /dexp(x))
```

The exact answer is 0.1394. How good is our approximation?

Note that Monte-Carlo integration is not always successful: Sometimes the ratio $g(X)/f(X)$ varies so much that the sample does not converge. Try to choose $f(x)$ so the ratio is roughly constant, and avoid cases where $g(x)/f(x)$ can be arbitrarily large.

- Hit-and-miss method (*aka* rejection sampling)

Let's start with a simple example to develop your intuition about "hit-and-miss". We have a triangular density function

$$g(x) = 1 - |1 - x|, \ 0 \le x \le 2$$

We can plot the density

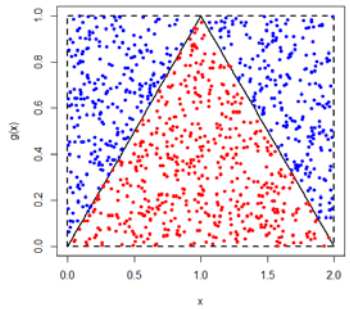
```
> x=seq(0, 2, by=0.05)
> plot(x, 1-abs(1-x), type='l', lwd=2, ylab="g(x)") #'l' is L not one.
```

How can we compute the area without integrating $\int_0^2 1 - |1 - x| \, dx$? First, we can see from the graph that the rectangular box outside the triangle has area=2.

```
> rect(0, 0, 2 ,1,lty=2,lwd=2) #create the rectangular box
```

Suppose I take a machine gun and randomly shoot at the rectangular n times, among which k bullets hit the triangle. It is reasonable to say that the area of the triangle is 2*(k/n).

```
> n=1000
> U1=runif(n, 0, 2)
> U2=runif(n, 0, 1)
> hit=which( (U2<=1-abs(1-U1)) == TRUE)
> miss=which( (U2<=1-abs(1-U1)) == FALSE)
> k=sum( (U2<=(1-abs(1-U1))) == TRUE)  #or just use length(hit)
> points(U1[hit], U2[hit], col= 'red', pch=20)
> points(U1[miss], U2[miss], col= 'blue', pch=20)
```




```
> 2*(k/n)
```

Is this close to the exact area=1? How about changing n=100000?

Below I will define the algorithm formally. Suppose we want to calculate $I = \int_a^b f(x)dx$ where f(x) is bounded below by c and bounded above by d.

Let A be the set bounded above by the curve $f(x)$ and by the box $[a,b]\times[c,d]$, then $I = |A| + c(b-a)$ (where $|A|$ is the area of A). In the example above, a=0, b=2, c=0, d=1, and

$$|A| = P((U_1,U_2)\in A)(b-a)(d-c) = P(U_2 \le g(U_1))(b-a)(d-c) \approx \frac{k}{n}(b-a)(d-c).$$

So, $I = \int_a^b f(x)dx$ can be approximated by $\frac{k}{n}(b-a)(d-c) + c(b-a)$.

Let's implement the approximation as a function in *R*.

```
> hit_miss=function(ftn, a, b, c, d, n){
> k=0
> for (i in 1:n){
>     x=runif(1, a, b)
>     y=runif(1, c, d)
>     hit=ifelse(y<=ftn(x), 1, 0)
>     k=k+hit
>     #cat("x =", x, "y =", y, "hit =", hit, "k =", k, "\n")
>}
>I=(k/n)*(b-a)*(d-c)+c*(b-a)
> return(I)
>}
```

Apply the function we just wrote to estimate $\int_0^1 x^3 - 7x^2 + 1dx = -1.0833$.

Taking the min and max of *each term*, we see that on [0, 1] the function is bounded below by c=0-7+1=-6 and above by d=1+0+1=2 (don't worry about the math, just digest the idea).

```
> f=function(x){x^3-7*x^2+1}
> hit_miss=function(f, a=0, b=1, c=-6, d=2, n=10)

> hit_miss=function(f, a=0, b=1, c=-6, d=2, n=100)

> hit_miss=function(f, a=0, b=1, c=-6, d=2, n=1000)

> hit_miss=function(f, a=0, b=1, c=-6, d=2, n=10000)
```

```
> hit_miss=function(f, a=0, b=1, c=-6, d=2, n=100000)

> hit_miss=function(f, a=0, b=1, c=-6, d=2, n=1000000)
```

What are we doing here? Do you notice that the simulation-based approximation improves so slowly even when *n* gets so large?

The hit-and-miss method converges very slowly when simulating complexed integrals. Nonetheless, given the horse power of contemporary computing devices, we will focus on the *effectiveness* of computing first and do not carry ourselves away with *efficiency*.