STAT 513/413: Lecture 13
Monte Carlo integration

(the real stuff)

Rizzo 6.1, 6.2

Include code, please

a)

n = 50

K	Probability of K run at least once
2	0.9976
3	0.9698
4	0.8050
5	0.5612
6	0.3076
7	0.1676

n = 100

K	Probability of K run at least once
2	<1.0
3	0.999
4	0.9648
5	0.8078
6	0.5582
7	0.3206

*<1.0 indicates a probability significantly close to 1.0, but is slightly less than 1.0.

	50	100
2	1.00000	1.00000
3	0.99998	1.00000
4	0.98149	0.99966
5	0.82057	0.97149
6	0.54471	0.80939
7	0.30812	0.54405

b)

n = 50

K	Average Number of Runs
2	6.4008
3	3.1504
4	1.5428
5	0.7508
6	0.3704
7	0.1804

n = 100

K	Average Number of Runs
2	12.5818
3	6.2312
4	3.1026
5	1.5090
6	0.7420
7	0.3760

	50	100
2	24.50000	49.50000
3	12.00000	24.50000
4	5.87500	12.12500
5	2.87500	6.00000
6	1.40625	2.96875
7	0.68750	1.46875

But not like this

```
# Question 3
# a)
#function of cdf, cdf calculated manually
# we could've used integrate to generate cdf in R if needed
cdf <- function(x) {-(x^[illegible])/[illegible] * (x-[illegible])}
#function to get random
random.generator <- function(n) {sapply(runif(n,min = [illegible], max = [illegible]), cdf)}

#generate a random number, for more numbers increase n, here n is 1
random.generator([illegible])


# b)

# function for inverse cdf
invcdf <- function(q){
    uniroot(function(x)(cdf(x) - q), range(x))$root
}

#function to get random
random.generator2 <- function(n) {sapply(runif(n,min = [illegible], max = [illegible]), invcdf)}

#generate a random number, for more numbers increase n, here n is 1
random.generator2([illegible])

# c
#confirm with plot
x <- seq([illegible],[illegible], by = [illegible])
plot(x,cdf)

#Question 4
run_count <- function(k, n){
    count <- [illegible]
    curr <- [illegible]
    curr_elem <- [illegible]

    for(i in n){
        if (i != curr_elem){
            if (curr == k){
                count <- count+[illegible]
                curr <- [illegible]
            }
            else{
                curr <- [illegible]
            }

        }
        else{
            curr <- curr + [illegible]
        }
        curr_elem <- i
    }
    return (count)
}
```

Even this does not help

```
run_count <- function(k, n){
    count <- 0
    curr <- [illegible]
    curr_elem <- [illegible]

    for(i in n){
        if (i != curr_elem){
            if (curr == k){
                count <- count+1
                curr <- [illegible]
            }
            else{
                curr <- [illegible]
            }

        }
        else{
            curr <- curr + 1
        }
        curr_elem <- i
    }
    return (count)
}
```

Finally, good to be in R spirit

```
# Set parameters
set.seed(318)
n = 10000

# Main Code
X_1 <- 1:n
for (i in X_1) {
  while (TRUE) {

    u = runif(1, 0, 2)
    v = runif(1, 0, 0.75)

    if ((3/4)*u*(2-u) >= v) {
      X_1[i] = u
      break
    }

  }
}
```


What is this formatting? Python? Check *any* book on R how the R code looks like in print!

Also, if you want to be serious with R: this is not the programming style of R! Loops? In R, you can do all the “Main Code” in *three* lines...

Are your computers that slow?

$N = 100$ is almost never a big deal; sometimes neither $N = 10000$

(You understand that probability is about large numbers? If not yet, the hope is that after this course you will. It is *the* most important knowledge you get here, much more important than coding skills or Monte Carlo technicalities.)

Last but not least

So, for the full credit:

- you answer all questions asked

- if it is mathematics, include *some* justification

- if it is testing in R, include the transcript how it tested

- if it is algorithm, implement it, and include the R code

- and then include the *transcript* showing how *that code* ran, the result *that code* produced and, if applicable, include the pictures *that code* produced

EVERYTHING COMPUTER WORK MUST BE REPRODUCIBLE
- otherwise no points

The programming part can be all monospaced

```
Probability of occurring and expected number of runs
of length k=2,3,4,5,6,7 in sequences of length 50 and 100

Expected values easier: closed-form solution:

> rr=cbind(2^(-(1:6))*(49:44),2^(-(1:6))*(99:94))
> rownames(rr)=as.character(2:7)
> colnames(rr)=as.character(c(50,100))
> rr
        50      100
2 24.50000 49.50000
3 12.00000 24.50000
4  5.87500 12.12500
5  2.87500  6.00000
6  1.40625  2.96875
7  0.68750  1.46875
```



```
My R function generating n coin tosses counting runs (16 lines):

runs = function(r, n=50, N=100000)
## the numbers of runs of k same elements in n coin tosses
## repeated N times
{
  runs = numeric(N)
  for (k in 1:N) {
    y = diff(floor(2*runif(n)))
```

... and so on...

And now to the topic: expected value = integral

Suppose that $c\ell(x)$ is a probability density of X

and we would like to compute the integral

$$\vartheta = \int_A g(x)\ell(x)\,dx = \frac{1}{c}\int_A g(x)c\ell(x)\,dx = \frac{1}{c}\mathsf{E}(g(X))$$

Notation: let $\mu = \mathsf{E}(g(X))$

Simple Monte Carlo algorithm:

If all X_i follow the distribution with density $cf(x)$

then $\mu = \mathsf{E}(g(X_1)) = \mathsf{E}(g(X_2)) = \ldots$ is estimated by

$$S_N = \frac{1}{N}\sum_{i=1}^{N} g(X_i)$$

(Note: this is written as $\overline{g(X)}$; do not confuse it with $g(\bar{X})$)

The desired integral ϑ is then estimated by $T_N = \frac{1}{c}S_N = \frac{1}{cN}\sum_{i=1}^{N} g(X_i)$

The typical instance

A "typical case" is when $\ell(x) = 1$ and $A = [a, b]$,

and the desired integral is $\displaystyle\int_a^b g(x)\ell(x)\,dx = \int_a^b g(x)\,dx$

Then the appropriate density has $c = \dfrac{1}{b-a}$ so that

$c\ell(x) = \dfrac{1}{b-a}$ is the density of the uniform distribution on $[a, b]$

The integral

$$\begin{aligned}\vartheta = \int_a^b g(x)\,dx &= (b-a)\int_a^b g(x)\frac{1}{b-a}\,dx \\ &= (b-a)\int g(x)c\ell(x)\,dx = (b-a)\mu\end{aligned}$$

is estimated by $\displaystyle T_N = (b-a)S_N = (b-a)\frac{1}{N}\sum_{i=1}^{N} g(X_i)$

Unbiased estimation

The target of interest, the integral, is defined outright as an expected value

So every unbiased estimator is estimating the integral

In particular, the simple Monte Carlo method yields an unbiased estimator:

$$\mathsf{E}(\mathcal{S}_N) = \mathsf{E}\left(\frac{1}{N}\sum_{i=1}^{N} g(X_i)\right) = \frac{1}{N}\sum_{i=1}^{N} \mathsf{E}(g(X_i)) = \mu$$

(as we have already seen before)

For the "typical case" we get

$$\mathsf{E}(T_N) = \mathsf{E}((b-a)\mathcal{S}_N) = (b-a)\,\mathsf{E}\left(\frac{1}{N}\sum_{i=1}^{N} g(X_i)\right) = (b-a)\mu = \vartheta$$

The measure of precision: variance

Once $E(g(X))$ [exists and] is finite, then all theory applies: the important thing to watch then is

$$\mathrm{Var}(S_N) = \frac{1}{N^2}\sum_{i=1}^{N}\mathrm{Var}(g(X_i)) = \frac{\mathrm{Var}(g(X))}{N}$$

Note once again: $\mathrm{Var}(g(X))$ is not $g(\mathrm{Var}(X))$; it depends on g

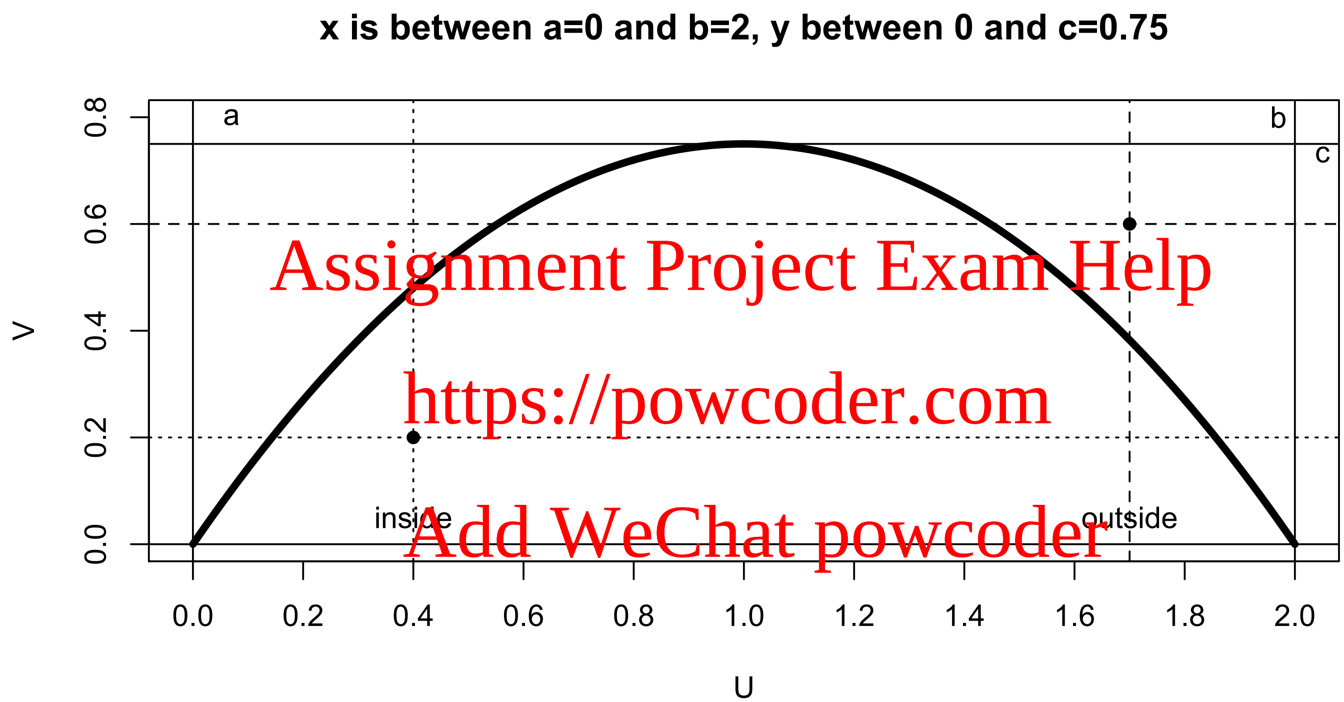
But we may estimate it by $\displaystyle\frac{1}{N-1}\sum_{i=1}^{N}(g(X_i) - S_N)^2$

And for the "typical case", the variance is

$$\begin{aligned}\mathrm{Var}(T_N) = \mathrm{Var}((b-a)S_N) &= (b-a)^2\,\mathrm{Var}(S_N)\\ &= \frac{(b-a)^2\,\mathrm{Var}(g(X))}{N}\end{aligned}$$

An alternative: hit-or-miss way of integration

Often the function f is bounded on $[a, b]$: say by 0 from below and by c from above (the example looks similar to acceptance/rejection method, but note the subtle difference)



Generate points (U, V), with U uniform on $[a, b]$ and V uniform on $[0, c)$; the desired result is

$$\frac{\text{number of points inside}}{\text{number of points alltogether, outside or inside}} \times \text{area}$$

$$\text{where area} = c(b - a)$$

Which one is better?

Let us do the simple Monte Carlo first, and then the hit-and-miss

```
> ngen = 10000
> x=runif(ngen,0,2); mean((3/4)*x*(2-x))*2
[1] 0.9965935
> x=runif(ngen,0,2); mean((3/4)*x*(2-x))*2
[1] 1.000751
> x=runif(ngen,0,2); mean((3/4)*x*(2-x))*2
[1] 1.002049
```



```
> ngen = 10000
> u = runif(ngen,0,2); v=runif(ngen,0,0.75); mean(v <= (3/4)*u*(2-u))*2*0.75
[1] 1.01865
> u = runif(ngen,0,2); v=runif(ngen,0,0.75); mean(v <= (3/4)*u*(2-u))*2*0.75
[1] 1.0035
> u = runif(ngen,0,2); v=runif(ngen,0,0.75); mean(v <= (3/4)*u*(2-u))*2*0.75
[1] 0.9894
```

Every difference here is in the eye of beholder (and also by chance), so let us try to estimate the variances; note, however, that we used twice as many random numbers for the hit-and-miss method

Variances: simple Monte Carlo

Note that we need to consider the variance *of the integral*: that is, if the mean (or probability) is multiplied by c to obtain the integral, the variance has then to be multiplied by c^2.

Everything is eventually divided by n - but to see the numbers better, we look at them before the division. In other words, our variances here are the variances of estimators, multiplied by n

First, the simple Monte Carlo; the estimate of variance is

```
> var((3/4)*x*(2-x))*2^2
[1] 0.1969241
```


It is also possible to obtain this theoretically: knowing that

$$2\,\mathrm{E}\left(\frac{3}{4}X(2-X)\right) = 2\int_0^2 \frac{3}{4}x(2-x)\frac{1}{2}\,dx = 1, \quad \text{we obtain}$$

$$2^2\,\mathrm{Var}\left(\frac{3}{4}X(2-X)\right) = 4\int_0^2 \left(\frac{3}{4}x(2-x)-1\right)^2\frac{1}{2}\,dx = 0.2$$

(But that is because we have rather a toy setup here)

Variances: simple Monte Carlo

And then the hit-and-miss method, where we use the $\hat{p}(1-\hat{p})$ *estimate* for $\sigma^2 = p(1-p)$

```
> p=mean(v <= (3/4)*u*(2-u))
> p*(1-p)*(2*0.75)^2
[1] 0.5069691
```

Also here we know the exact solution, as we know that $p = 1/(2(0.75)) = 1/1.5 = 2/3$, and therefore

$$p(1-p)(2(0.75))^2 = \frac{2}{3}\left(1-\frac{2}{3}\right)(1.5)^2 = \left(\frac{2}{3}\right)\left(\frac{1}{3}\right)\left(\frac{3}{2}\right)^2 = \frac{1}{2} = 0.5$$

So, "hit-and-miss" has larger variance - 0.5 to 0.2 - and also requires twice more random numbers...

Note: we are using the $\hat{p}(1-\hat{p})$ estimate, because we would like to evaluate "hit-and-miss" against "simple". If we wanted performance guarantee, we could use the upper bound $1/4$ - but then hit-and-miss would come out even worse

```
> (1/4)*(2*0.75)^2
[1] 0.5625
```

Remark

Note that in both averages it is the same number of summands, N, so both variances will be divided by N. “Twice more” is in terms of computational complexity: for one random uniformly distributed random number in simple Monte Carlo, we have to generate two uniformly distributed random numbers, in both coordinates, for the “hit-nad-miss” method. So in terms of n, how many times the random number generating procedure is ran, it is $n = N$ to $n = 2N$.

How about Hoeffding inequality here?

Recall: the bound is $2e^{-\frac{2N\varepsilon^2}{(b-a)^2}}$

given same N and ε, it is all about $(b-a)^2$

(the lower, the better)

In our particular case, simple Monte Carlo still fares better, as

$g(X_i) \in [0, 0.75]$ $\quad (b-a)^2 = 0.75^2 = 0.5625$

while the hit-and-miss method, working with 0-1 variables, has

$(b-a)^2 = 1$

Note also that for both methods, we have pretty exact performance guarantees by the Hoeffding inequality in both cases - without a need to estimate variance

(Inspiration: Monte Carlo Concepts, Algorithms, and Applications, by G. S. Fishman)