
MATHEMATICAL PHYSICS DEPARTMENT
MASTER OF COMPUTATIONAL SCIENCE DEGREE 2006-2007

NUMERICAL ALGORITHMS

Dr Derek O'Connor

Assignment No. 1 : Monte Carlo Methods.

OUT: Wed 4 Oct 2006

IN : Wed 18 Oct 2006

1 PURPOSE

The purpose of this assignment is to give you a brief introduction to the ideas of *pseudo*-random numbers and their use in Monte Carlo methods for evaluating integrals, probabilistic algorithms, and simple simulation.

2 RANDOM NUMBERS

Random numbers have become very important in many areas of computation, including prime number testing, sorting, algorithm evaluation, and numerical integration and probability.

2.1 MATLAB and Random Numbers

To get an immediate idea of what pseudo (computer-generated) random numbers look like type in the following commands at the MATLAB prompt :

```
>> rand(10,1)
>> x = rand(1000,1);
>> xbar = mean(x)
>> sdev = std(x)
>> hist(x,20)
```

The first command will display a list of 10 randomly-generated numbers $x \in [0, 1)$. The next statement generates a 1000×1 matrix, i.e., a vector, whose elements x_i are uniformly distributed on $[0, 1)$. Now we know that the uniform distribution $U(a, b)$ has mean $(a + b)/2$ and standard deviation $\sqrt{(b - a)^2/12}$, so $U(0, 1)$ has mean 0.5 and standard deviation 0.2886...

The next two commands calculate the mean and standard deviation of x , and the final command plots a histogram of the 1000 elements of x with 20 buckets.

2.2 Simulating Random Walks

A random walk in one dimension is as follows : make a sequence of moves and at move i take 1 step to the right with probability p or 1 step to the left with probability $1 - p$. Alternatively, we may think of it as a gambling game : make a sequence of coin-tosses and at each toss you win \$1 with probability p and lose \$1 with probability $1 - p$.

```
function CSum = RandWalk(n,p);
    UpsDowns = 2.*(rand(n,1) <= p)-1;
    CSum = cumsum(UpsDowns);
    plot(0:n-1,CSum)
```

The statement $2.*(rand(n,1) \leq p)-1$ generates a random walk as follows : $rand(n,1)$ generates a random column vector of size n , where each element is $U(0,1)$. Then $(rand(n,1) \leq p)$ generates a vector of 0s and 1s. Finally, $2.*(rand(n,1) \leq p)-1$ generates a vector UpsDowns of -1s and +1s. Type `»help cumsum` to see what cumsum does.

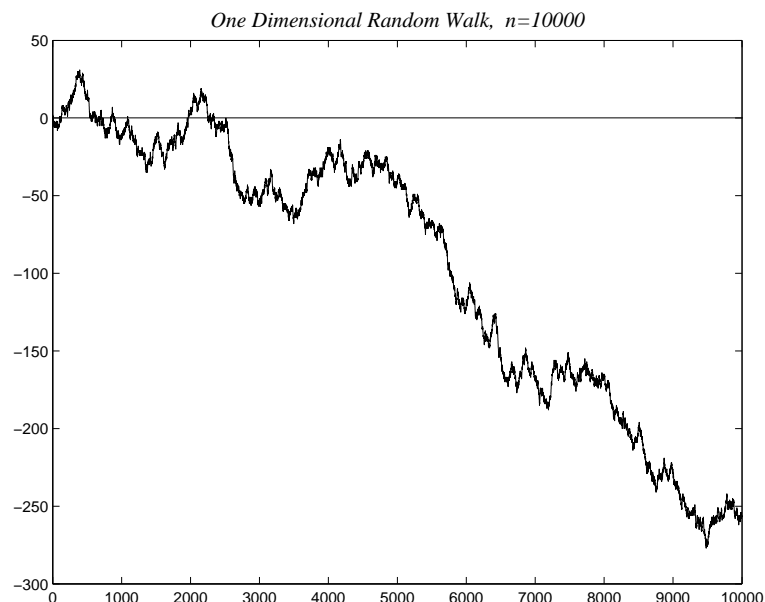


Figure 1: Random Walk in One Dimension with $n = 10^4$ and $p = 0.5$

One realization of a random walk with $p = 0.5$ is shown in Figure 1. You will notice that it seems to be biased : your winnings are negative most of the time. This is one of the many ‘paradoxes’ of probability theory. This behaviour is not paradoxical, merely counter-intuitive.¹

¹Can you explain the ‘paradox’?

3 MONTE CARLO SAMPLING

This is a topic that is often confused with the more general subject called *Simulation*.

J. Halton's definition is

The Monte Carlo Method represents the solution of a problem as a parameter of a hypothetical population, and using a random sequence of numbers to construct a sample of the hypothetical population, obtains a statistical estimate of the solution.

This is done in the hope that the estimator of the parameter is a good estimator of the solution to the problem.

Modern Monte Carlo methods were developed during the *Manhattan Project* (Atom Bomb) by Von Neumann and Ulam. It was used to evaluate nasty multiple integrals over complicated domains of integration. For this reason Monte Carlo sampling is often associated with Numerical Integration. Indeed, it is still the only reasonable method for evaluating multiple integrals or summations in high dimensions.

3.1 Hit-or-Miss Monte Carlo Sampling

This is a very crude method for estimating the value of

$$I = \int_a^b f(x) dx, \quad \text{where } 0 \leq f(x) \leq 1 \text{ on } [a, b]$$

Imagine we have drawn the function $f(x)$ on a piece of cardboard of width $(b - a)$ and height 1. Now throw a sequence of n darts at the cardboard and every time a dart hits the curve or below it, count this as a hit; otherwise count it as a miss. Let n_h be the number of hits and n_m the number of misses, with $n = n_h + n_m$. We call n_h/n the *Hit Ratio*.

Computationally we perform the following steps n times :

1. Generate a random point (x_i, y_i) in the rectangle $a \leq x_i \leq b$, $0 \leq y_i \leq 1$.
2. If $y_i \leq f(x_i)$ then increase n_h by 1.

The amount of work at each step is 2 random number generations and 1 function evaluation, or $T_{hm}(n) = (2r_g + f_e)n$.

Analysis. We claim that

$$\int_a^b f(x) dx = I \approx \hat{I}(n) = (b - a) \times \frac{n_h}{n}.$$

Let us simplify the problem by assuming that for $0 \leq x \leq 1$ we have $0 \leq f(x) \leq 1$ and that we want to evaluate $\int_0^1 f(x) dx$.

Define, on the unit square $0 \leq x \leq 1$, $0 \leq y \leq 1$, the *indicator function*

$$g(x, y) = \begin{cases} 1, & y \leq f(x), \\ 0, & y > f(x). \end{cases}$$

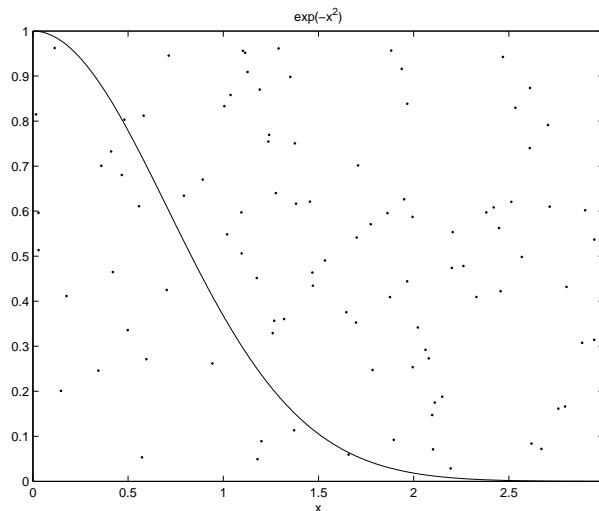


Figure 2: Hit-or-Miss Monte Carlo Sampling

Now, generate a sequence $\{g(u_{2k-1}, u_{2k}), k = 1, 2, \dots, n\}$, where each u_k has a uniform distribution on $[0, 1]$. This sequence of random variables is a *Bernoulli Process*, which we denote by $B(p)$. That is, a sequence of iid random variables $Z_1, Z_2, \dots, Z_k, \dots$ where $Z_k = 1$ with $\Pr(Z_k = 1) = p$ and $Z_k = 0$ with $\Pr(Z_k = 0) = q = 1 - p$. The mean or expected value is $E(Z_k) = p$, and the variance is $\text{Var}(Z_k) = p(1 - p)$.

In this Bernoulli process the probability of getting a hit is $p = (\text{area under curve})/(\text{area of sampling box}) = \int_0^1 f(x) dx = I$. Thus p is Halton's 'parameter of a hypothetical population'.

We have now transformed the deterministic problem of estimating $I = \int_0^1 f(x) dx$ into the statistical problem of estimating the parameter p of the Bernoulli process. We use the standard statistical method of taking a random sample of the (hypothetical) population $\{g(u_{2k-1}, u_{2k}), k = 1, 2, \dots, n\}$, and calculating the *sample mean*

$$\bar{g}(n) = \frac{1}{n} \sum_{k=1}^n g(u_{2k-1}, u_{2k}).$$

Standard statistical theory shows that $\bar{g}(n)$ is an unbiased estimator of the mean of $g(x, y)$, which is $p = I$. The statistic $\bar{g}(n)$ is itself a random variable, which has, by the Central Limit Theorem, a Normal Distribution :

$$\lim_{n \rightarrow \infty} \bar{g}(n) = \bar{g} \sim N(\mu_{hm}, \sigma_{hm}^2), \quad \text{where } \mu_{hm} = p = I, \quad \text{and } \sigma_{hm}^2 = \frac{p(1-p)}{n} = \frac{I(1-I)}{n}. \quad (1)$$

The *standard error* (deviation) of the estimator $\bar{g}(n)$ is $\sigma_{hm} = \sqrt{I(1-I)/n}$. This is a function of I , the *area* under $f(x)$, and is $O(n^{-\frac{1}{2}})$. Thus, to halve the error we must *quadruple* n .

3.2 Mean Value Monte Carlo Sampling

This method is better than Hit-or-Miss and is an interesting application of the Mean Value Theorem for Integrals. This theorem states that if $f(x)$ is continuous on $[a, b]$ then there exists a real number c in (a, b) such that

$$\int_a^b f(x) dx = f(c)(b - a).$$

Geometrically the theorem says that the area under the curve $f(x)$ between a and b is equal to the area of the rectangle whose width is $b - a$ and height is $f(c)$, for some $c \in [a, b]$.

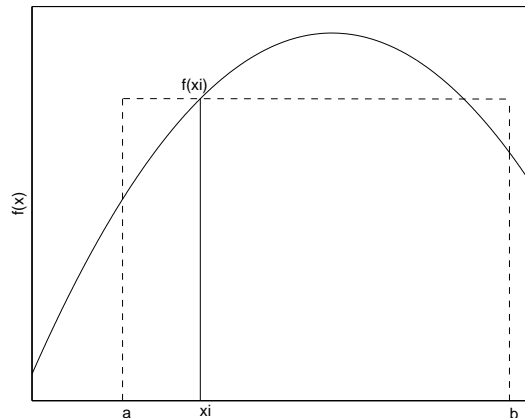


Figure 3: MVT Monte Carlo Estimation of Integral $\int_a^b f(x) dx$

In general we do not know c , especially if f is a complicated function. Instead we will guess the value of c and use it to calculate an estimate of the integral. In fact we will make a sequence of guesses by randomly picking values in $[a, b]$. Thus we will generate a sequence of random values $\{x_1, x_2, \dots, x_n\}$ and for each x_i evaluate $f(x_i)(b - a)$. This is the area of the rectangle whose base is a to b and whose height is $f(x_i)$ (see Figure 1.3). We then average these guesses to get an estimate of the integral. That is,

$$\int_a^b f(x) dx \approx \hat{I}(n) = \frac{1}{n} \sum_{i=1}^n f(x_i)(b - a) = \frac{(b - a)}{n} \sum_{i=1}^n f(x_i).$$

Computationally the MV Monte Carlo Method performs the following steps n times :

1. Generate a random point x_i in the interval $a \leq x_i \leq b$.
2. Evaluate $f(x_i)$ and accumulate this value.

Analysis. The amount of work at each step is 1 random number generation and 1 function evaluation, or $T_{mv}(n) = (r_g + f_e)n$. Compared to HM Monte Carlo we are using half the number of generations and get a smaller standard error σ_{mv} .

Now, the x_i 's are (pseudo-) random variables and therefore $\hat{I}(n)$ is a random variable. It can be shown that the expected value of $\hat{I}(n)$ converges to the value of the integral. That is,

$$\lim_{n \rightarrow \infty} E(\hat{I}(n)) = \int_a^b f(x) dx, \text{ and}$$

$$\sigma_{mv}^2 = \text{var}(\hat{I}(n)) = \frac{1}{n} \int_a^b (f(x) - I)^2 dx = \frac{1}{n} \int_a^b f^2(x) dx - \frac{I^2}{n}.$$

We see that σ_{mv} is a constant that depends on the *shape* of the function and not just the area I . In HM Monte Carlo $\sigma_{hm} = \sqrt{I(1-I)/n}$ depends on the area I only. Because the MC Monte Carlo uses more information about the function we would expect it to be a better method. Indeed, it can be shown (see

Hammersley & Handscombe, 1964) that

$$\sigma_{hm}^2 - \sigma_{mv}^2 = \frac{1}{n} \int_a^b f(x)(1-f(x)) dx > 0,$$

which means that *MV Monte Carlo is always better than HM Monte Carlo*. For some reason this fact is not widely appreciated, even today.

Although the Monte Carlo method may seem a rather frivolous method for evaluating an integral, it is still the only general method for evaluating multiple integrals over complicated domains.

4 PROBABILISTIC ALGORITHMS

Some very hard problems can be solved quickly if we introduce randomness into the solution. Testing the primality of a number n using the *Sieve of Eratosthenes* requires an amount of work that is $O(\sqrt{n})$. The size of this problem is $s = \log_2 n$, the number of bits needed to represent n . Thus $n = 2^s$ and the sieve method is $O(2^{\frac{s}{2}})$, which is exponential in the size of the problem.

Consider the following very fast algorithm for *Primality Testing*

```

function Prime(n)
  m := rand(2, sqrt(n))
  if m divides n then
    return False
  else
    return True
endfunc Prime

```

We know that if n is composite then one of its prime factors is less than \sqrt{n} . We randomly choose a number less than \sqrt{n} and test to see if it is a factor of n . Note that m may not be prime.

If this method returns False then we know with certainty that n is composite because m divides n . What happens if it returns True? This means that the randomly-chosen m does not divide n , but there may be other numbers that divide n , which we have not checked. Hence, when Prime returns False this answer is true, and when it returns True this answer may be false. Got it?

What is the probability p that function Prime(n) returns False? Consider the number $n = 2623 = 43 \times 61$. The algorithm randomly chooses m from $[2 \dots \sqrt{2623} = 51]$. Thus it has a 1/50 chance of picking 43 and so this algorithm is wrong 98% of the time. For larger values of n this probability gets larger.

4.0.1 P-Correct Random Algorithms

Random algorithms that give the correct answer with probability p are called **P-Correct**. This means that if we submit a given problem to a p -correct algorithm many times, it will give the correct answer $100p\%$ of the time. How can this help us to get a fast prime tester?

The following algorithm by M. Rabin (1980) is the first good primality testing algorithm.

```

function PrimeRabin(n)
  k := rand(2, n - 2)
  if n is strongly pseudo-prime
    to base k then
    return True, definitely.
  else
    return False, probably.
endfunc PrimeRabin

```

The pseudo-prime test can be done in polynomial time so Rabin's algorithm is polynomial. If n is not prime there is at least a 75% chance that the algorithm returns *false*. Hence the chance of it being wrong is less than 25%. If it returns *true* then n is definitely prime.

How can this algorithm be used if, despite its sophistication, it returns a wrong answer with probability $1/4$? Well, because it is random we may get a different answer if we re-test n a second time. If it returns *false* a second time, then the probability of getting the wrong answer twice is $1/4 \times 1/4 = 1/16$. If we use the algorithm k times on n then the probability of a wrong answer is $(1 - p)^k = 1/4^k$. This can be made arbitrarily small. Indeed for $k = 100$ we get a probability of error $= 2^{-200} = 10^{-60}$ and this is much smaller than the probability of a cosmic ray zapping one of the bits in your processor while the algorithm is running.

4.1 Matrix Equality Testing

Given three $n \times n$ matrices A , B , and C , determine if $AB = C$. The obvious deterministic solution is to multiply A and B , and check if the result equals C . This runs in time $O(n^3)$, using ordinary matrix multiplication. If we use the best (to date) fast matrix multiplication algorithm we get $O(n^{2.376})$, (Coppersmith and Winograd). In contrast, there is a probabilistic solution due to Freivalds [16] that runs in $O(n^2)$ time:

```

function Freivalds(A,B,C)
  Choose  $u \in \{0,1\}^n$  randomly and uniformly
  if  $A(Bu) \neq Cu$  then
    return false, definitely.
  else
    return true, probably.
endfunc EqualABC

```

Analysis. Generating a random vector u whose components are 0 or 1 requires $O(n)$ time. The main work is the **if**-test which checks the statement $ABu \neq Cu$. The product ABu is calculated as $v_1 = Bu$ in $O(n^2)$ time, followed by $v_2 = Av_1$ and $v_3 = Cu$, also in $O(n^2)$ time. In MATLAB this is simply $A*(B*u)$. The check $v_2 = v_3$ is $O(n)$. Thus the total time is $O(n^2)$.

The following theorem² gives a bound on the probability that the algorithm gives the wrong answer.

²R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995, page 162.

Theorem 1. Let $A, B,$ and C be $n \times n$ matrices over the field \mathcal{F} such that $AB \neq C$. If u is chosen from $\{0, 1\}^n$ randomly and uniformly then

$$\Pr(ABu = Cu) \leq \frac{1}{2}.$$

□

The implications of this theorem are

- If $AB = C$ then $\Pr(ABu = Cu) = 1$.
- If $AB \neq C$ then $\Pr(ABu = Cu) \leq \frac{1}{2}$.
- If we run the algorithm k times then the probability of a wrong answer is 2^{-k} .

Exercise 7.1 of Motwani and Raghavan asks for a verification that there is nothing magical about choosing u to have only entries drawn from $\{0, 1\}$. In fact, any two elements of \mathcal{F} may be used.

5 SIMULATION

5.1 Feller’s Biased Coin Problem

This is from William Feller, *An Introduction to Probability Theory and its Applications, Volume I, 3rd Edition*, 1968, Wiley, Problem 10, page 238.

“*Simulating a Perfect Coin.* Given a biased coin such that the probability of heads is $p \neq 0.5$, we simulate a perfect coin as follows. Throw the biased coin twice. Interpret HT as success and TH as failure; if neither event occurs repeat the throws until a decision is reached. (a) Show that this model leads to Bernoulli trials with $\Pr(\text{success}) = \frac{1}{2}$. (b) Find the distribution and number of throws required to reach a decision.”

We denote the outcomes from the biased coin as H or T and the unbiased outcomes from Feller’s algorithm as H' and T' . We can improve Feller’s algorithm by taking the shortcut shown in Figure 4 : if we get HH or TT then we then we repeatedly throw the coin once until we get HT or TH .

Here are some possible sequences of throws without the shortcut and with the shortcut

(TT) (TT) (HH) (HH) (HH) (TT) (TT) (TT) (HH) (TH) = T'	(TT) (TTTTTTTTTTTTTTTTH) = T''
(HH) (HH) (TT) (TT) (TT) (HH) (TH) = T'	(HH) (HHHHHHHHHHHT) = H''
(TT) (TT) (TT) (HH) (HH) (TT) (TT) (TT) (HH) (HT) = H'	(HT) = H''

Analysis. The biased coin has $\Pr(H) = p$ and $\Pr(T) = q = 1 - p$. We have $\Pr(HH) = p^2$, $\Pr(TT) = q^2 = (1 - p)^2$, and $\Pr(HT) = \Pr(TH) = pq = p(1 - p)$. By design, Feller’s algorithm has only two outputs, $H' = (HT)$ and $T' = (TH)$ and these have equal probabilities. Thus we have a Bernoulli process with $\Pr(\text{success}) = \Pr(\text{failure}) = 0.5$.

The distribution of the number of throws required to get either output is $P(k, p) = \Pr(\text{No. throws} = k)$.

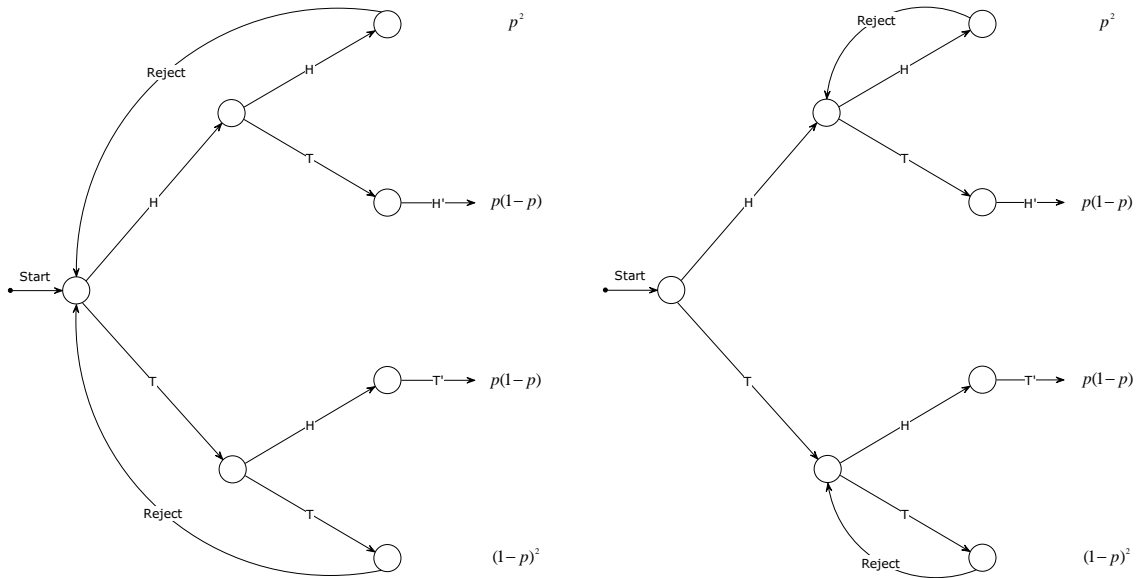


Figure 4: Feller's Biased Coin Algorithm and Shortcut

6 THE ASSIGNMENT

This assignment has three parts : (1) Simple Integration, (2) Testing Matrix Equality, and (3) Simulating a Fair Die. Each is independent of the others and can be done in any order.

6.1 Integration

Write MATLAB functions for Hit-or-Miss and Mean Value Monte Carlo sampling with the function headings HMMC(f, a, b, c, d, nsamp) and MVMC(f, a, b, nsamp). Use each method to evaluate

$$I_1 = \int_{-5}^5 \frac{1}{1+x^4} dx$$

$$I_2 = \int_0^{2\pi} \frac{1}{1+3\sin^2 t} dt$$

$$I_3 = \int_{-10}^{10} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$$

1. Run tests on each method on each function for sample sizes $n_{\text{samp}} = 10^1, 10^3, 10^4$.
2. Use MAPLE or MAXIMA or any other method to get the exact answer for each integral.
3. Draw the function and the random points generated. See Figure 2.
4. Draw up a comparison table of results.

6.2 Matrix Equality Testing

Write the functions `Freivalds(A,B,C)` and `MatEqDet(A,B,C)` that return 1 if $AB = C$ and 0 otherwise. The function `MatEqDet(A,B,C)` uses ordinary matrix multiplication to test for $A * B = C$.

Write a test function that plots the times for `Frievald(A,B,C)` and `MatEqDet(A,B,C)`, with random matrices of sizes $n = 100 : 100 : 1000$. Count the number of times `Frievald(A,B,C)` is wrong.

6.3 Simulation

Write the function `RanBit(p)` which returns 1 with probability p and 0 with probability $1 - p$. Then write the function `BiasCoin(p)` that returns 1 (H) or 0 (T) with equal probability. This function uses `RanBit(p)` only.

Do the following tests:

1. Write a test function that calls each function N_{samp} times and plots the histogram of the H,Ts and 0,1s.
2. Write a test function that calls the function `BiasCoin(p)` N_{samp} times, counts the number of times that `RanBit(p)` is called for each call to `BiasCoin(p)`, and plots the histogram of this number. Calculate the mean and standard deviation of this number

Try to determine analytically the functional form of the distribution of the number of calls to `RanBit(p)` for each call to `BiasCoin(p)`. The histogram in (2) above should give you a hint.

7 NOTES

Although this assignment may seem long, no `MATLAB` function is longer than 10 lines. Each part can be worked on independently of the others, which greatly simplifies the work.

Plan and think before you code. Be sure you understand the purpose of each function before attempting to code and test it.

7.1 `MATLAB` Hints

The following `MATLAB` functions may be useful. Type `»help funname` to get information on each of these :

- | | | |
|-----------|-----------|-------------|
| 1. all | 4. rand | 7. quad |
| 2. size | 5. hist | 8. plot |
| 3. length | 6. cumsum | 9. semilogx |

See the `LaTeXNotes` for information on including `MATLAB` plots in a `LaTeX` files. I will write more notes on this subject later.

8 SOLUTION — Monte Carlo Integration

8.1 Hit-or-Miss Monte Carlo

We give two implementations of this algorithm : *looped* and *vectorized*. In writing MATLAB programs it is important to understand the difference between these because vectorized programs run faster, in general. Vectorizing a looped program is a MATLAB skill that is best learned by reading the MATLAB manual and looking at examples.

```
% Matlab code HMMC.m Version 1 - loops
% function val = HMMC(f,a,b,c,d,nsamp)
%
% Evaluate the integral of c <=f(x) <= d
% between the limits a <= x <= b
% using the Hit or Miss Monte Carlo method
%
% Inputs:  f inline function, the box a,b,c,d
%
function val = HMMC(f,a,b,d,nsamp)
%
% Generate nsamp points (xi, yi)
hits = 0;
for i = 1:nsamp
    x = a + (b-a)*rand(1,1);
    y = c + (d-c)*rand(1,1);
    if y <= f(x)
        hits = hits + 1;
    end;
end;
val = (b-a)*(d-c)*hits/nsamp
```

```
% Matlab code HMMC.m Version 2 - Vectorized
function meanstd = HMMC(f,a,b,d,nsamp)
%
    area = (b-c)*(d-c);
    x = a + (b-a)*rand(nsamp,1);
    y = c + (d-c)*rand(nsamp,1);
    hits = ( y <= f(x) )
    val = area*sum(hits)/nsamp
    sdev = area*std(hits)/sqrt(nsamp);
    meanstd = [val;sdev];
```

```
% Matlab code HMMC.m Version 3 - Vectorized
% with plotting
function meanstd = HMMC(f,a,b,d,nsamp)
%
    x = a + (b-a)*rand(nsamp,1);
    y = c + (d-c)*rand(nsamp,1);
    fx = f(x);
    hits = ( y <= fx )
    val = area*sum(hits)/nsamp
    sdev = area*std(hits)/sqrt(nsamp);
    meanstd = [val;sdev];
    ezplot(f,[a b]); hold on;
    plot(x,fx,'ko'); hold off;
```

Version 1 is similar to a C, Fortran, or Pascal program : the loop is explicitly stated and indexed to generate $x = U(a,b)$ and $y = U(0,d)$, one at a time. There is no need to store the generated points because hits records all the information we need.

Version 2 is in the preferred MATLAB form : everything is vectorized. MATLAB is an *interpreted* language which means that each statement must be interpreted (de-coded) and executed while the program is running. The more statements there are the more interpretation is done. Thus there are $3 * \text{interps} + 3 * \text{execs}$ in Version 1. In Version 2 there are $3 * \text{interps} + 3 * \text{nsamp} * \text{execs}$, which can make a big difference if nsamp is large. Note that x, y are vectors and the expression $(y \leq f(x))$ is a vector of ones (true) and zeros (false).

Version 3 is Version 2 with a plot of the function before we start and a plot of the random points when we have finished. Notice how simple the vectorized version of the Mean Value MC below is : 3 MATLAB statements. The MAPLE statements for the exact solutions are shown on the right.

8.2 Mean Value Monte Carlo

The vectorized version of this method requires just 4 lines of code and shows why MATLAB has become so popular : it allows you to do a lot of work with a small amount of code that is not obscure (once you understand vectorization).

<pre>% Matlab code MVTMC.m % function val = MVTMC(f,a,b,nsamp) % Evaluate the integral of f(x) between % the limits a <= x <= b using the % Mean Value Monte Carlo method % Inputs: f inline function,a,b % function val = MVTMC(f,a,b,c,d,nsamp) % % Generate nsamp points (xi, f(xi)) x = a + (b-a)*rand(nsamp,1); fx = f(x); val = (b-a)*sum(fx)/nsamp;</pre>	<pre>% Maple code for exact answers f1 := x -> exp(exp(x)); f2 := x -> (1-x^2)^(3/2); f3 := x -> exp(x+x^2); f4 := x -> 1 + sin(2000*x); evalf(int(f1(x),x=0..1)); 6.316563839027700 + 0.000000000000000 I evalf(int(f2(x),x=0..5)); 0.589048622548088 - 138.64345990764 I evalf(int(f2(x),x=0..1)); 0.589048622548086 evalf(int(f3(x),x=-5..5)); 0.988481927874795 10¹²</pre>
--	---

8.3 Testing

Each Monte Carlo method was run twice, on the three functions, using five sample sizes, as shown in Table 1. These tests do not meet publication standards but will suffice for demonstration purposes.

Sample Size. It is obvious from the results that increasing sample size decreases the standard deviation, thus increasing the accuracy. However, high accuracy requires huge samples because

$$\text{Acc}(n) = \frac{1}{\text{SDev}(n)} = \frac{c}{\sqrt{n}}.$$

This may be tolerable for the ‘toy’ functions here, but real-life functions may have hundreds of variables and require an expensive simulation for each function evaluation. For example, the British ICI company (or *enterprize* in today’s biz-speak) in the 1950’s did many experiments on optimizing the operation of their chemical plants. Each function evaluation required running a large chemical plant at chosen settings (variable values) for 1 or 2 days. That is expensive!

Function Shape. The shape of a function can greatly affect the results of Monte Carlo integration. The shape depends on the first and higher derivatives. Functions whose derivative are changing rapidly can be very difficult to integrate.

If a function has high narrow peaks then any type of crude sampling has a good chance of missing the important parts of this function. *Importance Sampling* is a Monte Carlo method that tries to overcome these problems by concentrating the sampling on the ‘nasty’ or important parts of the function. This prompts the question: where are the nasty parts of the function?

Note that ‘nasty’ functions cause difficulties for deterministic methods such as the Trapezoidal Rule, etc.

Table 1: Hit or Miss and Mean Value Results

Method	Function	10	100	1000	10000	100000	Exact
Hit or Miss Run 1	$f_1(x)$	5.000000	1.900000	2.220000	2.229000	2.238200	2.216112
	$\sigma_1(x)$	1.666667	0.394277	0.131487	0.041621	0.013181	0.000000
	$f_2(x)$	4.398230	3.392920	3.078761	3.166725	3.154913	3.141592
	$\sigma_2(x)$	0.959772	0.314730	0.099376	0.031416	0.009935	0.000000
	$f_3(x)$	2.000000	1.400000	1.120000	0.978000	0.989000	1.000000
	$\sigma_3(x)$	2.000000	0.512865	0.145488	0.043134	0.013712	0.000000
Mean Value Run 1	$f_1(x)$	3.261117	1.673554	2.120463	2.245090	2.203584	2.216112
	$\sigma_1(x)$	1.352194	0.312227	0.106708	0.034455	0.010814	0.000000
	$f_2(x)$	2.877133	3.115299	3.152381	3.162984	3.142897	3.141592
	$\sigma_2(x)$	0.495432	0.153726	0.050563	0.015758	0.004965	0.000000
	$f_3(x)$	1.297717	0.570563	0.996050	1.016534	1.003671	1.000000
	$\sigma_3(x)$	0.793469	0.155619	0.068765	0.021693	0.006821	0.000000
Hit or Miss Run 2	$f_1(x)$	2.000000	2.800000	2.110000	2.253000	2.210600	2.216112
	$\sigma_1(x)$	1.333333	0.451261	0.129091	0.041780	0.013122	0.000000
	$f_2(x)$	3.141593	2.890265	3.204425	3.154787	3.135309	3.141592
	$\sigma_2(x)$	1.047198	0.314730	0.099376	0.031417	0.009935	0.000000
	$f_3(x)$	0.000000	0.600000	1.000000	0.962000	1.027400	1.000000
	$\sigma_3(x)$	0.000000	0.342893	0.137909	0.042798	0.013962	0.000000
Mean Value Run 2	$f_1(x)$	2.802398	2.649105	2.225233	2.269964	2.228612	2.216112
	$\sigma_1(x)$	1.200238	0.375473	0.108366	0.034707	0.010870	0.000000
	$f_2(x)$	3.031109	3.113300	3.227703	3.140440	3.141002	3.141592
	$\sigma_2(x)$	0.567956	0.141357	0.050588	0.015753	0.004963	0.000000
	$f_3(x)$	2.028454	1.207237	1.163242	1.009283	0.995699	1.000000
	$\sigma_3(x)$	1.069661	0.236966	0.072898	0.021608	0.006807	0.000000

9 Solution — Matrix Equality Testing

Freivalds's method is easy to implement :

```
function Answer = Freivalds(A,B,C);
[n,n] = size(A);
x = rand(n,1) < 0.5;
Answer = all(A*(B*x) == C*x);
```

Generating a random vector $x \in \{0, 1\}^n$ could not be simpler, but you need to know the vectorization capabilities of MATLAB. The second statement above generates a random n -vector u whose elements u_i are from $\text{Unif}(0, 1)$. Each u_i is compared with 0.5 and true (1) or false (0) is assigned to x_i . Thus we get a random $\{0, 1\}$ vector x . This requires $O(n)$ ops.

The third statement is where the main work is done. The parentheses are very important and cannot be omitted (or 'optimized' away by some smart compiler). The vector $(B*x)$ is formed first using $O(n^2)$ ops, the vector $A*(B*x)$ is formed second using $O(n^2)$ ops, and the vector $C*x$ is formed third, using $O(n^2)$ ops. The true-false vector $A*(B*x) == C*x$ is finally formed, using $O(n)$ ops. The MATLAB function `all(v)` returns true (=1) if all elements of a vector v are nonzero, using $O(n)$ ops. Thus Freivalds returns true if and only if $ABx = Cx$, which implies that $AB = C$. This function uses $O(n^2)$ ops.

The standard deterministic matrix equality test is this:

```
function Answer = Standard(A,B,C);
Answer = all(all(A*B == C));
```

Forming $A*B$ requires $O(n^3)$ ops and this dominates all other operations. $A*B==C$ forms an $n \times n$ true-false (0,1) matrix. `all(A*B == C)` forms a (0,1) vector and `all(all(A*B == C))` gives 0 or 1.

9.1 Timing Tests

Timing the two methods on random matrices of sizes $n = 100 : 100 : 1000$ gives Figure 5. This graph confirms that Freivalds's method is $O(n^2)$ while the standard method is $O(n^3)$.

9.2 Error Tests

Testing the 'correctness' of this algorithm raises many questions. Recall from Theorem (1) that if $AB \neq C$ then $\Pr(ABu = Cu) \leq 1/2$. We have not proved this but it is well to note that the 1/2 is a bound on the probability of the algorithm giving the wrong answer. Note also that the theorem says nothing about the size of the matrices and that the elements of these matrices may be from any field, not just \mathbb{Z} , the field of integers.

Here is the obvious way to test Freivalds's algorithm for correctness :

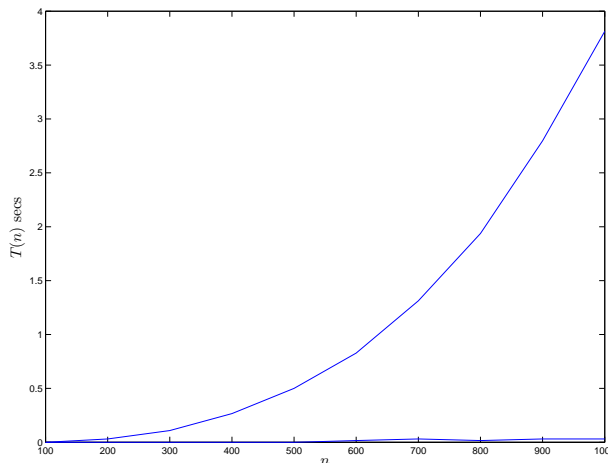


Figure 5: Freivalds’s vs Standard Matrix Equality Test

```

function TestFreivalds-1(n,n_s)
    n_e := 0
    for k := 1 to n_s do
        Generate n × n random matrices A,B,C.
        if Standard(A,B,C) ≠ Freivalds(A,B,C) is true then
            n_e := n_e + 1
        endif
    endfor
    return n_e
endfunc TestFreivalds-1
    
```

Let us determine the number of operations required by this test. For each $k = 1, 2, \dots, n_s$, generating three random matrices takes $O(n^2)$ ops, using Standard takes $O(n^3)$ ops, and using Freivalds takes $O(n^2)$ ops. Hence the total number of ops for the test is

$$T_1(n, n_s) = \sum_{k=1}^{n_s} [3O(n^2) + O(n^3) + O(n^2)] = \sum_{k=1}^{n_s} O(n^3) = n_s O(n^3) = O(n_s n^3) \text{ ops.} \quad (2)$$

This is an enormous amount of work. Worse still, most of it is not necessary. Let us see how we can eliminate this waste.

Recall that Freivalds’s method can go wrong only if and only if $AB \neq C$. Hence we need to test it only for those cases. If we generate three random matrices A, B, C , then $\Pr(AB = C)$ is very small, and vanishingly small as the size of the matrices increase.

Exercise 1 ($\Pr(AB = C)$). Calculate for three matrices generated by MATLAB’s $A = \text{rand}(n, n)$, etc., the probability that $AB = C$, for $n = 1, 2, 3$.

Indeed, let us assume that $AB \neq C$ for randomly generated matrices. Then the modified test function is

```

function TestFreivalds-2( $n, n_s$ )
   $n_e := 0$ 
  for  $k := 1$  to  $n_s$  do
    Generate  $n \times n$  random matrices  $A, B, C$ .
    % Assume  $AB \neq C$ 
    if Freivalds( $A, B, C$ ) is true then
       $n_e := n_e + 1$ 
    endif
  endfor
  return  $n_e$ 
endfunc TestFreivalds-2

```

Remember we are assuming that $AB \neq C$ and so, if Frieivalds says that they are equal, it must be wrong.

$$T_2(n, n_s) = \sum_{k=1}^{n_s} [3O(n^2) + O(n^2)] = \sum_{k=1}^{n_s} O(n^2) = O(n_s n^2) \text{ ops.} \quad (3)$$

Compared to (2), this is an order-of-magnitude reduction in the work.

We can see that most of the work in TestFreivalds-2 is in generating the three random matrices. Now, let us move the generation of the matrices outside the sampling loop. This gives us

```

function TestFreivalds-3( $n, n_s$ )
   $n_e := 0$ 
  Generate  $n \times n$  random matrices  $A, B, C$ .
  % Assume  $AB \neq C$ 
  for  $k := 1$  to  $n_s$  do
    if Freivalds( $A, B, C$ ) is true then
       $n_e := n_e + 1$ 
    endif
  endfor
  return  $n_e$ 
endfunc TestFreivalds-3

```

Let us examine Frieivald's theorem again and see what it is doing : It is given the three matrices A, B, C , over which it has no control. Furthermore, it assumes that $AB \neq C$. The algorithm generates a random (0,1) vector u , calculates $v = Bu$, $w = Av$, $y = Cu$, and then tests if $w = y$.

In TestFreivalds-3 the function Freivalds is applied n_s times to the same 3 matrices. Thus this test is, in a sense, searching for random (0,1) vectors u such that $A(Bu) = Cu$. An error is recorded each time it finds one.

Let us re-arrange things slightly. Testing $AB = C$ is equivalent to testing $AB - C = 0$, or $D = 0$, where $D = AB - C$. Thus we may view Frieivalds's algorithm as a test of $D = 0$. It goes wrong only when $AB \neq C$ or when $D \neq 0$. This leads to the following test

```

function TestFreivalds-4(n,n_s)
    n_e := 0
    Generate an n x n random matrix D.
    % Assume D ≠ 0
    for k := 1 to n_s do
        Generate a random (0,1) n-vector u.
        if Du = 0 then
            n_e := n_e + 1
        endif
    endfor
    return n_e
endfunc TestFreivalds-4
    
```

This test saves generating 2 random matrices and 2 matrix-vector products. Note, however, that we are no longer testing $AB = C$. Instead we are trying to determine how often a random (0,1) vector u is such that $Du = 0$ when $D \neq 0$. This, we claim, has the same probability as the event $ABu \neq Cu$, i.e., the number of errors, n_e , is the same for TestFreivalds-3 and TestFreivalds-4.

Exercise 2 ($\Pr(D = 0)$). The TestFreivalds-4 function assumes that the random matrix D is not 0. Calculate $\Pr(D = 0)$, given that D is generated by MATLAB's `D=rand(n,n)`.

Results of Error Tests. The four TestFreivalds tests with $n = 100$, $n_s = 1000$ gave $T_1(n, n_s) = 10$ secs, $T_2(n, n_s) = 5$ secs, $T_3(n, n_s) = 0.5$ secs, $T_4(n, n_s) = 0.1$ secs, which shows the savings of 100 : 1. Each test gave similar error results which implies that they are equivalent.

Table 2 shows how the errors in Freivalds's method vary with matrix size n and sample size n_s .

Table 2: Errors n_e in Freivalds-4 for matrices of size $n = 1 \dots 20$

n_s	n										
	1	2	3	4	5	6	7	8	9	10	20
100	55	20	16	6	5	2	0	1	0	0	0
1000	521	252	121	52	29	19	10	8	2	1	0
10000	5024	2561	1243	596	307	157	85	40	22	6	0
100000	49950	25085	12554	6258	3058	1549	803	374	189	83	0

Here are some very tentative conclusions about Freivalds's method, based on these results.³

1. The number of errors increases linearly with sample size: $n_e = a + bn_s$.
2. The number of errors decreases geometrically with matrix size: $n_e = c2^{-n}$.
3. The number of errors seems to be $n_e = n_s 2^{-n}$, which is a combination of (1) and (2). Table 3 shows that this function fits the experimental results remarkably well.⁴

³I have not found much theory about this method, other than the proof of Theorem(1). I must try harder

⁴This is too good to be true. There must be a trivial theorem somewhere that gives this result.

Table 3: Errors $n_e = n_s 2^{-n}$ for matrices of size $n = 1 \dots 20$

n_s	n										
	1	2	3	4	5	6	7	8	9	10	20
100	50	25	13	6	3	2	1	0	0	0	0
1000	500	250	125	63	31	16	8	4	2	1	0
10000	5000	2500	1250	625	313	156	78	39	20	10	0
100000	50000	25000	12500	6250	3125	1563	781	391	195	98	0

10 Solution — Simulation

The recursive MATLAB function for Feller’s algorithm is as follows :

```

% =====
function face = BiasCoin(p);
%
% Flipping a biased coin to generate H or T with pr = 1/2.
% RanBit(p) returns 1 with prob. p and 0 with prob. 1-p
% Derek O'Connor, UCD, July 2006
test = RanBit(p);           % test = 0 or 1
test = test + RanBit(p)*2;  % test = 0,1,2, or 3
if test == 0 | test == 3
    test = BiasCoin(p);     % Reject & Repeat
end;
face = test;
    
```

10.1 Analysis.

We denote the number of flips needed to get an unbiased result (H' or T') by $N_f(p)$.

Distribution Function of $N_f(p)$. Given a biased coin with $\Pr(H) = p$, we wish to derive

$$\Pr(N_f(p) = k), \text{ the probability that it takes } k \text{ flips to obtain } H' \text{ or } T'. \tag{4}$$

First, let us consider the simpler problem : what is the probability that it takes $N_{fh}(p) = k$ flips of a biased coin to get the first H ? This requires that we get a sequence of $k - 1$ T s followed by a single H . Thus we get

$$\Pr(N_{fh}(p) = k) = \Pr(\underbrace{TT \dots T}_k H) = \underbrace{(1-p)(1-p) \dots (1-p)}_{k-1} p = (1-p)^{k-1} p = pq^{k-1}, \tag{5}$$

where $q = 1 - p$. This is called a **Geometric Distribution**, which is the discrete equivalent of the exponential distribution. The expected value of $N_{fh}(p)$ is

$$\begin{aligned} E[N_{fh}(p)] &= \sum_{k=0}^{\infty} kp(1-p)^{k-1} = p \sum_{k=0}^{\infty} kq^{k-1} = p \sum_{k=0}^{\infty} \frac{d}{dq} q^k \\ &= p \frac{d}{dq} \sum_{k=0}^{\infty} q^k = p \frac{d}{dq} \left(\frac{1}{1-q} \right) = \frac{p}{(1-q)^2} = \frac{1}{p}. \end{aligned} \tag{6}$$

Similarly, it can be shown that $\text{Var}[N_{fh}(p)] = (1 - p)/p^2$.

We now apply a similar analysis to $N_f(p)$. Feller's method repeatedly flips a coin twice until either $(HT) = H'$ or $(TH) = T'$ occurs. We may regard either of these outcomes as a success and either (TT) or (HH) as a failure. Thus, Feller's method is a Bernoulli process, $Z_1, Z_2, \dots, Z_k, \dots$, where

$$Z_k = \begin{cases} \text{success with prob.} & = \Pr(HT \text{ or } TH) = 2p(1 - p) = 2pq, \\ \text{failure with prob.} & = \Pr(HH \text{ or } TT) = p^2 + (1 - p)^2 = p^2 + q^2. \end{cases} \quad (7)$$

This answers part (a) of Feller's question.

If the first success occurs at the k th double flip then we have

$$\Pr(N_{df}(p) = k) = \Pr(\underbrace{FF \dots F}_{k-1} S) = \underbrace{(p^2 + q^2)(p^2 + q^2) \dots (p^2 + q^2)}_{k-1} 2pq = 2pq(p^2 + q^2)^{k-1},$$

where $N_{df}(p) = 2N_f(p)$ is the number of double flips. Letting $r = 1 - 2pq$ we get

$$\Pr(N_{df}(p) = k) = 2pq(p^2 + q^2)^{k-1} = (1 - r)r^{k-1}, \quad k = 1, 2, 3, \dots \quad (8)$$

This is a Geometric distribution with parameter $r = 1 - 2pq = 1 - 2p(1 - p)$.

The expected value of $N_{df}(p)$ is

$$\begin{aligned} E[N_{df}(p)] &= \sum_{k=1}^{\infty} k(1 - r)r^{k-1} = (1 - r) \sum_{k=1}^{\infty} kr^{k-1} = (1 - r) \sum_{k=0}^{\infty} \frac{d}{dr} r^k \\ &= (1 - r) \frac{d}{dr} \sum_{k=1}^{\infty} r^k = (1 - r) \frac{d}{dr} \left(\frac{1}{1 - r} \right) = \frac{(1 - r)}{(1 - r)^2} = \frac{1}{1 - r} \\ &= \frac{1}{2pq} = \frac{1}{2p(1 - p)} \end{aligned}$$

This gives

$$E[N_f(p)] = E[2N_{df}(p)] = \frac{1}{p(1 - p)}, \quad (9)$$

Expected Number of Flips. This is an alternative way to derive $E[N_f(p)]$, using the method of Knuth and Yao (see Section 11). Let $F(p)$ be the expected number of flips until an unbiased result (H' or T') is obtained. A minimum of 2 flips are made. If these give either $HT = H'$ or $TH = T'$ we are done. If we get HH with probability p^2 we must start again. If we get TT with probability $(1 - p)^2$ we must start again. Hence we get the functional equation

$$F(p) = 2 + p^2F(p) + (1 - p)^2F(p) \quad \text{which gives} \quad F(p) = \frac{1}{p(1 - p)}. \quad (10)$$

which verifies equation (9). Note that this functional equation can be derived directly from the recursive function BiasCoin(p) above.

This function has a minimum at $p^* = \frac{1}{2}$, which gives $F(p^*) = 4$, and $F(p) \rightarrow \infty$ as $p \rightarrow 0, 1$. That is, the more biased a coin is, the more work is needed to get unbiased results.⁵

⁵This might be seen as an example of the moral precept, *Honesty is the best policy*.

10.2 Testing the Algorithm and Theory

Table 4: Frequencies for $N_f(p)$ with $N_{\text{samp}} = 10^4$ and $p = 0.1$

N_f	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Fr(N_f)	1772	1476	1204	983	811	666	534	473	361	303	256	203	161	118	123	114
N_f	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62	64
Fr(N_f)	97	64	44	43	42	32	18	18	21	10	12	10	5	6	3	4
N_f	66	68	70	72	74	76	78	80	82	84	86	88	90	92	94	96
Fr(N_f)	3	1	1	1	2	1	0	0	1	0	1	0	0	0	1	0
N_f	98	100	102	104	106	108	110	112	114	116	118	120	122	124	126	128
Fr(N_f)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

This sample has $\min(N_f) = 2$, $\text{Avg}(N_f) = 11.232$, $\max(N_f) = 126$, and $\text{S. Dev}(N_f) = 10.12$.

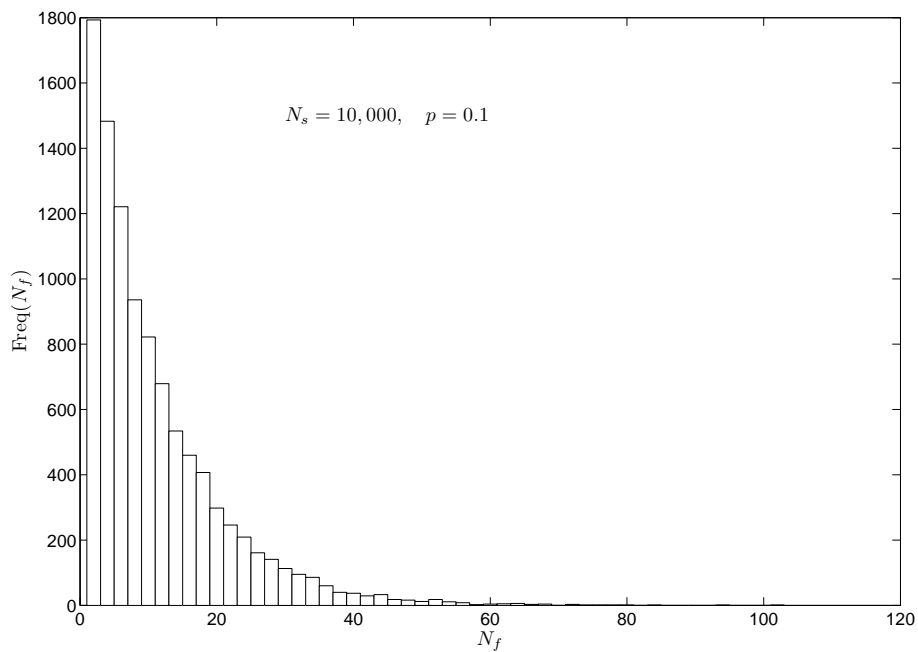


Figure 6: Feller's Biased Coin Method. $p = 0.1$

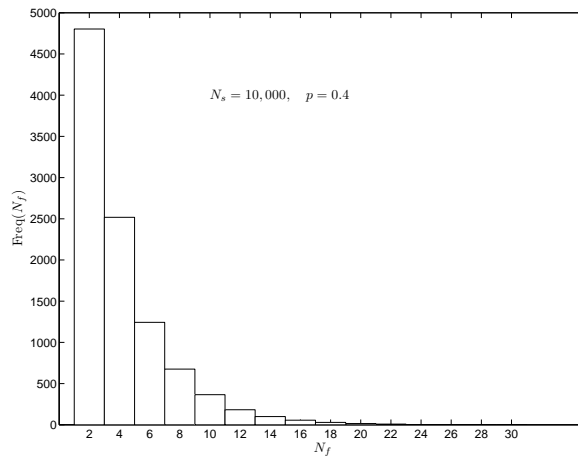


Figure 7: Feller’s Biased Coin Method. $p = 0.4$

Table 5: Frequencies for $N_f(p)$ with $N_{\text{samp}} = 10^4$ and $p = 0.4$

N_f	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
$\text{Fr}(N_f)$	4803	2518	1243	676	365	182	100	56	29	15	8	2	1	1	1

This sample has $\min(N_f) = 2$, $\text{Avg}(N_f) = 4.1804$, $\max(N_f) = 30$, and $\text{S. Dev}(N_f) = 3.036$.

11 Knuth and Yao’s Problem

This is similar to Feller’s problem and was used to develop a theory of discrete random number generation.

Donald E. Knuth and Andrew C. Yao, in their paper, “The Complexity of Random Number Generation”, in *Algorithms and Complexity*, edited by J. F. Traub, Academic Press, 1976, begin with this problem:

Problem You have the function `RanBit` which returns 1 with probability $\frac{1}{2}$ and 0 with probability $\frac{1}{2}$. Using this function only, write a function `Roll` that returns one of the set $\{1, 2, 3, 4, 5, 6\}$, each with probability $\frac{1}{6}$. Analyze your `Roll` function and find the average number of calls to `RanBit` it makes. Can you speed this up?

11.1 Solution

The problem is to simulate the roll of a die using a random bit generator. Using `RanBit` three times in succession, we get one of eight possible bit strings $\{(000), (001), \dots, (110), (111)\}$ each with probability $\frac{1}{2^3} = \frac{1}{8}$. We interpret this set of bit strings as the integer set $\{0, 1, \dots, 7\}$. If we get 0 or 7 then we reject these and start over again. Thus we get one of the set $\{1, 2, 3, 4, 5, 6\}$ with equal probability $= \frac{1}{6}$. This is shown in Figure 8

The recursive MATLAB function is as follows :

```

% =====
function face = Roll;
% -----
% Flipping a coin to generate the six faces of a die.
% RanBit returns 1 with prob. 0.5 and 0 with prob. 0.5
% Derek O'Connor, UCD, Jan 2006
test = RanBit*2^0;
test = test + RanBit*2^1;
test = test + RanBit*2^2;
if test == 0 | test == 7
    test = Roll;          % Reject and Repeat
end;
face = test;

```

11.2 Analysis.

We follow Knuth and Yao's analysis. The analysis of `Roll` is fairly simple : The first 3 statements `test = etc.` make 3 calls to `RanBit`. This produces an integer in $\{0, 1, \dots, 7\}$ each with probability $1/8$. Two out of these 8 integers are rejected and `Roll` is called (recursively) again. The probability of calling `Roll` again is $2/8$. Thus the number of calls to `RanBit` is a random number. Let F stand for the average number of calls to `RanBit`. Then the average number of calls is

$$F = 3 + \frac{2}{8}F, \quad \text{or} \quad F = 4. \quad (11)$$

Although 3 calls to `RanBit` give us more numbers than we need, we have to make 4 calls to it on average.

Can we speed up this function? Yes, by noticing that we do not need to repeat all 3 `test = etc.` statements. The bit strings (000) and (111) are equally probable and may be regarded as 0 and 1 outcomes of the first call to `RanBit` and so we need only two more calls to `RanBit`. This shortcut is shown as a dotted line in Figure 8.

Equation (11) now becomes

$$F_{\text{sc}} = 3 + \frac{2}{8}F', \quad \text{and} \quad F' = 2 + \frac{2}{8}F'. \quad (12)$$

This gives $F' = \frac{8}{3}$, and so $F_{\text{sc}} = 3\frac{2}{3}$, which is always less than $F = 4$.

Note Although the first statements of `Roll` produce an integer in $\{0, 1, \dots, 7\}$ each with probability $1/8$, the function, by construction, only produces an integer in $\{1, \dots, 6\}$ each with probability $1/6$.

Note It may seem that we need probabilities and averages in our analysis because we are using a random bit generator. This is not true, necessarily. Indeed, the analysis of most algorithms that contain an **if**-test requires probability statements.

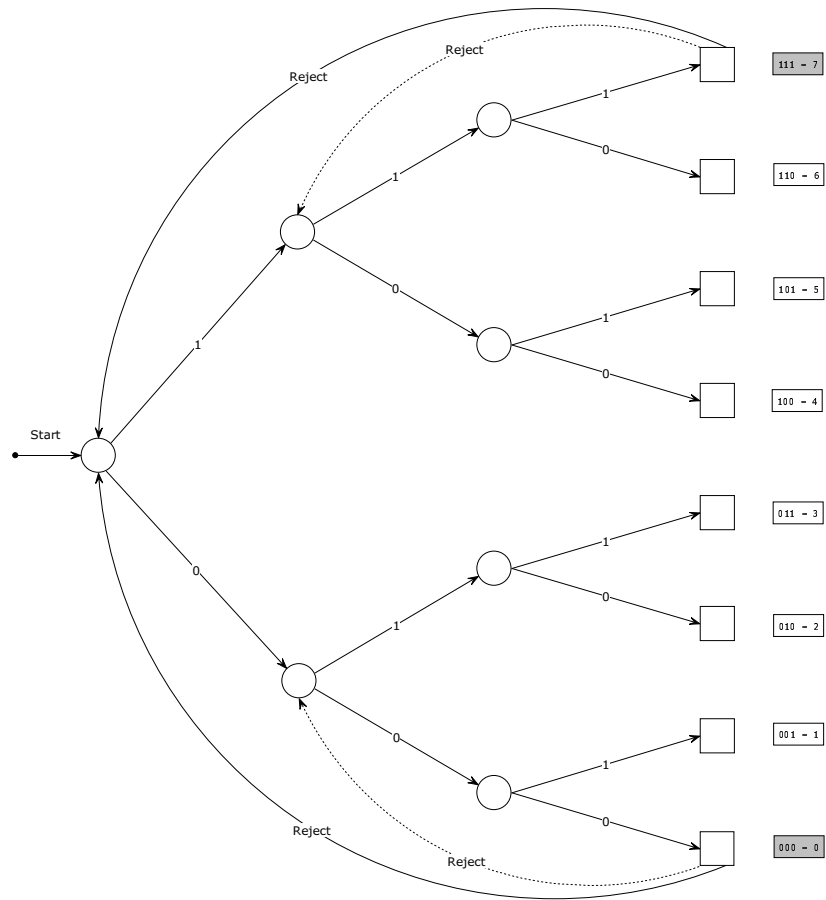


Figure 8: Rolling a Die using Random Bits, and a Shortcut

```

% =====
function face = RollSC;
% -----
% Flipping a coin to generate the six faces of a die.
% RanBit returns 1 with prob. p and 0 with prob. 1-p
% Uses the shortcut described above.
% Derek O'Connor, UCD, Jan 2006
test = RanBit*2^0;
test = test + RanBit*2^1;
test = test + RanBit*2^2;
if test == 0 | test == 7
    test = RollP(test);
end;
face = test;
% -----
function face = RollP(test);
% -----
test = test + RanBit*2^1;
test = test + RanBit*2^2;
if test == 0 | test == 7
    test = RollP(test);
end;
face = test;
    
```

12 Comments and Criticisms

1. The old canard, 'It must be due to floating point error', keeps cropping up. This is a useless statement (a lie usually) unless you can demonstrate how and where the floating point error occurred.
2. Related to the point above, some of you seem to have the notion on that FPNs are somehow 'fuzzy', imprecise, vague, things. They are not. They are very real. A string of 64 bits is very real, precise, and can be seen (on an oscilloscope). On the other hand, the irrational constant $\sqrt{2}$ does not exist, let alone the transcendental constant π . Remember, irrational and transcendental numbers are convenient mathematical fictions.
3. Some of you still have difficulty in laying out your reports. Despite repeated requests to put important code in the body of the report, I still get 'see the MATLAB code at the back'. Imagine you wrote a report whose Executive Summary⁶ said just 'See Appendix D-2 (Parts a,d, and z)'.
4. Simulating a Biased Coin : Did nobody check Feller's book, which has the solution?
5. Shortcut for Biased Coin : I does not work I never checked the output. Sorry. I thought I could use the Knuth & Yao method above.
6. I think I know why some people (this year and previous years) were getting inordinately long run-times. The timed code included I/O statements. A definite No-No. Think about it.
7. There are no prizes for the longest report.
8. Do not email me zip files larger than 500K. Larger files will be dumped by my spam washer.
9. If I use a MATLAB function that you do not understand and I have not explained then use »help funcname.

⁶For people who are so important they can spare only 15 secs on your 350 page report.

13 Historical Note

I have called this method of obtaining an unbiased result from a biased coin Feller's method because I found it in the book by William Feller, *An Introduction to Probability Theory and its Applications, Volume I, 3rd Edition*, 1968, Wiley, Problem 10, page 238.

The book by R. Motwani and P. Raghaven, *Randomized Algorithms*, Cambridge University Press, 1995, claims, in Problem 1.1, page 25, that it is due to von Neumann. They give the reference J. von Neumann, 'Various techniques used in connection with random digits' (notes by G.E. Forsythe), *National Bureau of Standards, Applied Mathematics Series*, 12:36–38, 1951. A more complete reference is A.S. Householder, G.E. Forsythe, and H.H. Germond, eds., *Monte Carlo Method*, National Bureau of Standards Applied Mathematics Series, 12 (Washington, D.C.: U.S. Government Printing Office, 1951): 36-38. This is the Proceedings of a symposium held June 29, 30, and July, 1, 1949, in Los Angeles, California, under the sponsorship of the Rand Corporation, and the National Bureau of Standards, with the cooperation of the Oak Ridge National Laboratory and organized by the Institute for Numerical Analysis (U.S.) This was historically important, being the first formal meeting on the Monte Carlo method and discussed, presumably, many of the techniques developed during and after the *Manhattan Project*.

William Feller obtained his MS from the University of Zagreb (Yugoslavia) and PhD from Göttingen, and was a professor of mathematics at Brown, Cornell and Princeton universities. He knew von Neumann, who was at the Institute for Advanced Studies. The first edition of Feller's book appeared in 1950, the second in 1957. This book is frequently quoted and is still used in many (of the better) courses.