Exercise 1 - Solutions

1.1

The goal of this exercise is to estimate the area inside a polygon using Monte Carlo simulation. The idea behind using simulation as an integration technique stems from the fact that the normal numerical integration methods become tedious when dealing with low valued functions in multidimensional space. Good examples are probability distribution functions, which usually have values close to zero.

The basic procedure to integration with MC-technique is to generate a huge number of points and to check whether they place within the area (volume) of interest or in the rectangular area surrounding the polygon. The rectangle is used because it has an area, which is easily calculated. Checking whether a point lies inside or outside the polygon can be a bother, but in this case we can use a built-in Matlab function called *inpolygon*. We start off by defining the polygon, which in Matlab requires only the corner points

```
X = [0.2 0.4 0.8 0.5 0.2]';
Y = [0 0.25 0 1 0]';
```

Next we need to set the surrounding rectangle, which in this case is a unit square

```
ax=0; ay=0; bx=1; by=1;
A=(bx-ax)*(by-ay);
```

To plot the polygon and the surrounding rectangle we use following commands

```
figure; hold on;
plot(X,Y);
plot([ax bx bx ax ax]',[ay ay by by ay]');
```

Generate 2N random numbers, which represent the coordinates of the N random points

```
x=rand(N, 1);
y=rand(N, 1);
```

Next, scale the U(0, 1) distributed numbers (this is not necessary when inside a unit circle) to be inside the rectangle

```
x=ax+(bx-ax)*x;
y=ay+(by-ay)*y;
```

Now we have two vectors with dimensions N × 1. To check whether each point is inside the polygon or not we use the command

```
in=inpolygon(x,y,X,Y);
```

in is also a N×1 vector and it's elements are either 0 or 1 depending whether the point with the same index in x and y is outside or inside. The property which we are interested in is the number of points inside the polygon, which we can find out by using a command

```
S=length(x(in));
```

To see how many points lie within the polygon and how many outside, we can use the plot commands as follows

```
plot(x(in), y(in), 'r.');
plot(x(~in), y(~in), 'b.');
```

The value of the integral is calculated as the fraction of points inside the polygon multiplied by the area of the surrounding rectangle

```
lambda=S/N;
V=A*lambda;
```

Next we want to calculate the confidence interval for our result. For this we need to find out the variance of the result. Since each point is either inside or outside the polygon, we can use a binomial distribution for the integral. The variance of a binomial distribution can be calculated as

```
var_V=lambda*(1-lambda)/N;
```

Using the central limit theorem, we can approximate the distribution of the integral is close to being normal and thus the confidence interval is

```
V_upper=V+1.96*sqrt(var_V);
V_lower=V-1.96*sqrt(var_V);
```

The factor 1.96 is the value ($z_{1-\alpha/2}$) of the inverse of the normal cumulative distribution function (Φ^{-1}) for the desired confidence level 95% (i.e. $P(Z<z_{1-\alpha/2}) = \Phi(z_{1-\alpha/2})=0.975$).

To see how the sample size affects the width of the confidence interval, one needs to look at how we calculated the variance. Sample size N appears in the denominator of the variance, so increasing the sample size decreases the variance and also the width of the CI.

Next we need to find the appropriate sample size for a predefined absolute error ε. The thing we actually want to find is how many points we must generate so that the absolute error of the integral is in 95% of the cases within the limits A ± ε. If we look at the way we calculated the variance earlier, we notice that if the variance is a function of λ, it has a maximum when λ = 0.5. This maximum has a value σ^2 = 1/4N. From the Chebychev inequation we have

$$P(|\lambda_N - \lambda| \geq \epsilon) \leq \frac{\sigma^2}{\epsilon^2}$$

We want the left side of the inequality to be less than α, where $1-\alpha$ is our confidence level. So, just to be sure we use the inequality

$$\alpha \geq \frac{1}{4\epsilon^2 N}$$

(To understand how the reasoning goes, notice that now $\alpha \geq 1/4\epsilon^2 N \geq P(|\lambda_N - \lambda| \geq \epsilon)$). Finally we solve the inequality for N

$$N \geq \frac{1}{4\epsilon^2 \alpha} \quad .$$

With the figures used in our problem, we get the worst-case sample size to be

$$N \geq \frac{1}{4 \cdot 0.05^2 \cdot 0.05} = 2000$$

1.2

We first list all paths that lead from the start to the completion. The paths where e_1 is the first activity are {e_1,e_3,e_5}, {e_1,e_3,e_6}, {e_1,e_4,e_5}, {e_1,e_4,e_6}, {e_2,e_3,e_5}, {e_2,e_3,e_6}, {e_2,e_4,e_5}, {e_2,e_4,e_6}. To determine the expected duration, we simulate repeatedly the durations of the activities and observe the maximum duration of the above paths. In matlab:

```
function T = network_mc(N)
%Monte carlo -simulation for an activity network
%N the sample size

%Parameters for the exponential distributions
mu = [0.5 1 1.5 2 0.25 1];


%The main loop
for i = 1:N

    e = exprnd(mu);

    t(1) = sum(e([1 3 5]));
    t(2) = sum(e([1 3 6]));
    t(3) = sum(e([1 4 5]));
    t(4) = sum(e([1 4 6]));
    t(5) = sum(e([2 3 5]));
    t(6) = sum(e([2 3 6]));
    t(7) = sum(e([2 4 5]));
    t(8) = sum(e([2 4 6]));

    T(i) = max(t);
```

```
end

%Convergence
for i = 1:N;
Tav(i) = mean(T(1:i));
end

%Plot convergence
h1 = figure;
plot(Tav);

%Cumulative empirical distribution
for i = 1:1500
Td(i) = sum(T<i/100);
end

%Plot cumulative distribution
h2 = figure;
plot([0.01:0.01:15],Td/N);
hold;
plot([Tav(end) Tav(end)],[0 1],'r');
```

1.3

The inverse transform method

In the inverse-transform method, we should first define the inverse of the cumulative distribution function. It is easily seen, that *F* has a jink at *c*, so consider separately the cases where $x \leq c$ and $x > c$. Solving $y=F(x)$ for *x* gives us

$$F^{-1}(y)=\begin{cases} 0 & \text{if } y<0 \text{ or } y>1 \\ \sqrt{y(b-a)(c-a)}+a, & \text{if } y\geq 0 \text{ and } y\leq \frac{c-a}{b-a} \\ b-\sqrt{(1-y)(b-a)(b-c)}, & \text{if } y\geq \frac{c-a}{b-a} \text{ and } y\leq 1 \end{cases}$$

Now, generating $U\sim U(0,1)$ and placing it in the equation above gives us a random variate from the triangular distribution. Writing the above in Matlab should be easy:

```
function y=trirnd(a,b,c)
%Generates a triangularly distributed random number
%a  minimum
%b  maximum
%c  mode

%Uniform [0,1] distributed random number
U = rand;
```

```
if U <= (c-a)/(b-a)
     y = sqrt(U*(b-a)*(c-a))+a;
elseif U > (c-a)/(b-a)
     y = b-sqrt((1-U)*(b-a)*(b-c));
end
```

The acceptance-rejection method

First refer to the lecture material on the acceptance-rejection method. A function *t(x)* that is equal to or larger than the probability density function *f(x)* of the triangular distribution for all *x* is simply a constant function whose value is *f(c)*, i.e. the value of *f* at its mode.

$$t(x)=\begin{cases}\dfrac{2}{b-a}, & \text{if } a\le x\le b\\ 0, & \text{otherwise}\end{cases}$$

Next, we define the function *r(x)=t(x)/C*, where *C* is the area under *t(x)*. We get

$$r(x)=\begin{cases}\dfrac{1}{b-a}, & \text{if } a\le x\le b\\ 0, & \text{otherwise}\end{cases}$$

which is the same as the uniform distribution on the interval [*a,b*]. A random variate from the triangular distribution is obtained as follows:

- Generate a random *Y* having the probability density function *r* and a random U from U[0,1] and independent of *Y*.
- If *U≤f(Y)/t(Y)*, return *Y* as the desired random variate. Otherwise go to step 1.

The matlab code to accomplish this could look something like

```
function y=trirnd(a,b,c)
%Generates a triangularly distributed random number
%a  minimum
%b  maximum
%c  mode


t = 2/(b-a);
r = (2/(b-a))/2;

accept = 0;

while accept == 0

     y = a+rand*(b-a);
     U = rand;
```

```
        if y <= c
             f = 2*(y-a)/((b-a)*(c-a));
        else
             f = 2*(b-y)/((b-a)*(b-c));
        end

        if U < f/t
             accept=1;
        end

end
```

1.4

The idea of the chi-square test is to assess, whether a generated random sample is uniformly distributed on the interval [0,1]. In the test, [0,1] is divided into *k* subintervals. Then, the number of random numbers that lie within each subinterval and the number of points we would expect to lie within each interval are calculated to define the test statistic. Essentially we are comparing the observed density to the one from which we are trying to generate random numbers.

Now, let us write a Matlab-function that performs the test. As input arguments, the function takes a vector composed of the random numbers in the sample and the number of subintervals k. The output y is the P-value of the test

```
function y = chitest(x,k)
```

First, calculate the sample size

```
N=length(x);
```

The subintervals of [0,1] are](i-1)/k,i/k], i=1,2,…,k. To calculate the observed values within each of these:

```
for i = 1:k
     f(i) = length(find(x>(i-1)/k & x<=i/k));
end
```

The test statistic becomes

```
X2 = sum(((f-N/k).^2)./(N/k));
```

The P-value of the test is the tail probability of the chi-square distribution at X2

```
y = 1-chi2cdf(X2,k-1);
```

Instead of the function *chi2cdf*, which is a part of the Statistics toolbox, you may construct the chi-square cumulative distribution function by yourself. See the exercise paper for the code.

Naturally, we reject the hypothesis of the sample being uniformly distributed, if the P-value is less than the confidence level we have chosen (e.g. 0.05).

To repeat the test for 100 independent sequences of random numbers and find out the proportion of tests where uniformity was rejected, you can write for instance:

```
for i = 1:100

     p(i) = chitest(rand(n,1),k);

end

y = find(P<0.05);
y = length(y)/N;
```

The proportion of rejected cases should be somewhere near the confidence level.