MAS3909 - Markov Processes

Group 31 Project

Calum Doran, Stephen Cole, Joseph Crone

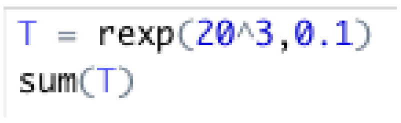
April 2020

Contents

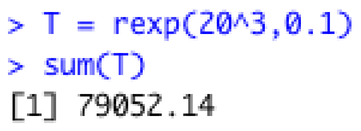
1 Poisson process (Practical 1)

1.1 Simulating observations and calculating jump times

For the first part of this practical we will use an exponential distribution with parameter $\lambda = 0.1$ to simulate inter-arrival times of a Poisson process. The Poisson process in question will be simuated over the interval, $0 \leq t \leq 10000$. We will use the R function, **`rexp(n,lambda)`**, to simulate these inter-arrival times, with the **`n`** argument being the number of simulations produced. We will choose an **`n`** argument that is large enough so that we cover the whole interval, meaning, the sum of all the inter-arrival times is larger than the 10000

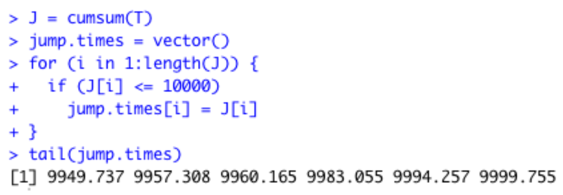


```
T = rexp(20^3,0.1)
sum(T)
```

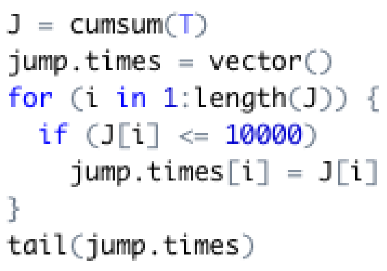


```
> T = rexp(20^3,0.1)
> sum(T)
[1] 79052.14
```

Here we have the code and output for simulating the inter-arrival times. We have used $n = 20^3$ and using the **`sum(...)`** function, we have checked that we have enough values to cover the interval.
Now using these inter-arrival times, we will calculate the jump times for this Poisson process. To do this we will use the **`cumsum(...)`** function. This function produces a cumulative sum for the vector inputted, say x, in other words, it produces a vector with entries $(x_1, x_1 + x_2, x_1 + x_2 + x_3, \dots)$. This new vector will be the jump times. We also want to remove any jump times that occur after $t = 10000$, as these are not part of the interval we are considering.



```
> J = cumsum(T)
> jump.times = vector()
> for (i in 1:length(J)) {
+   if (J[i] <= 10000)
+     jump.times[i] = J[i]
+ }
> tail(jump.times)
[1] 9949.737 9957.308 9960.165 9983.055 9994.257 9999.755
```



```
J = cumsum(T)
jump.times = vector()
for (i in 1:length(J)) {
  if (J[i] <= 10000)
    jump.times[i] = J[i]
}
tail(jump.times)
```

Here we have converted the inter-arrival times into jump times and then using a for loop to select all the jump times that are less than 10000. We have also used the **`tail(...)`** function to look at the last five values of the $jump.times$ vector to check the for loop has worked correctly.

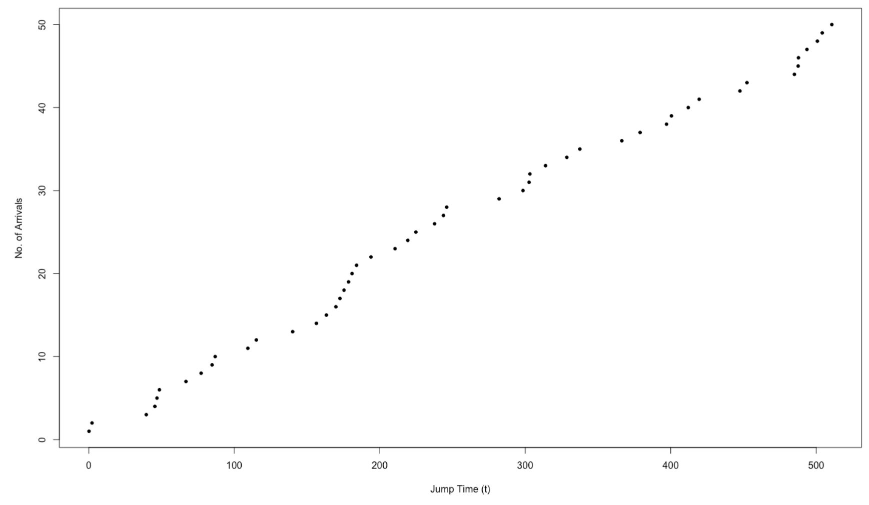


Figure 1: Plot of jump times for the first 50 arrivals.

1.2 Verifying distribution

Now we have the jump times for the interval, we want to check to see if the simulated values are distributed according to the Poisson process. We can do this by splitting the jump times into equally sized intervals and then comparing these the expected distribution of values, according to our Poisson process with parameter $\lambda = 0.1$.

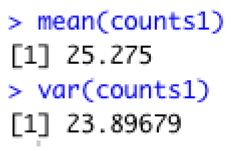
Our first task is to create equally sized intervals and then count the number of observations that have occurred in each interval. This can be done using the following code,

```
my.breaks1 = seq(0,10000,by=250)
counts1 = as.vector(table(cut(jump.times,breaks=my.breaks1)))
```

the size of the intervals we have created is 250, using our Poisson process, we can calculate the theoretical distribution of each of these intervals.
As the parameter is $\lambda = 0.1$ and the interval width is 250, the distribution for each individual interval, say U, will be,

$$
\begin{gathered}
U \sim Po(\lambda t) \\
\implies U \sim Po(0.1 \times 250) \\
\implies U \sim Po(25) \\
E[U] = var(U) = \lambda = 25
\end{gathered}
$$

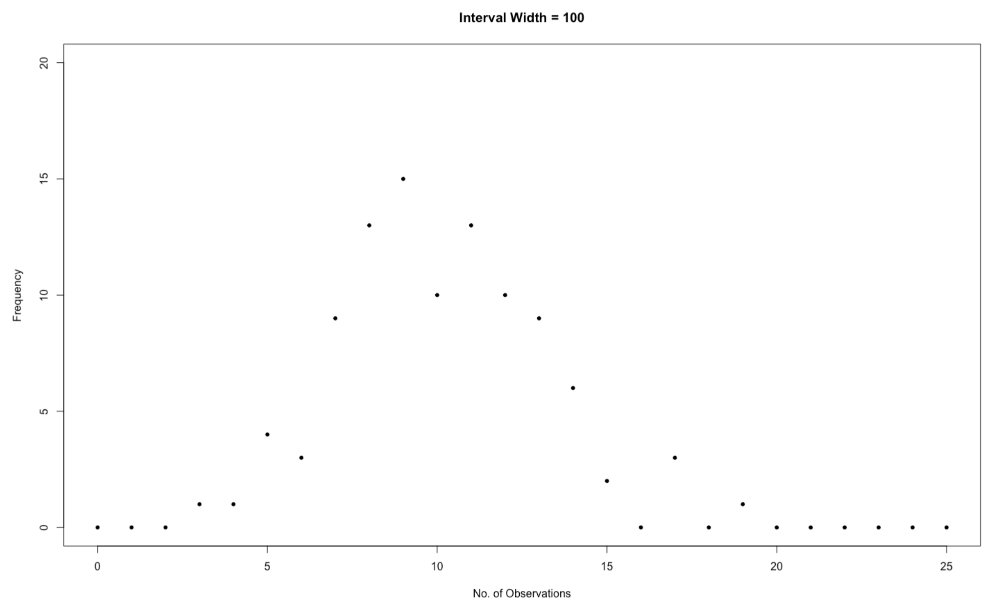
Now we have the theoretical mean and variance for the intervals, we can calculate the sample mean and variance then compare.



```
> mean(counts1)
[1] 25.275
> var(counts1)
[1] 23.89679
```

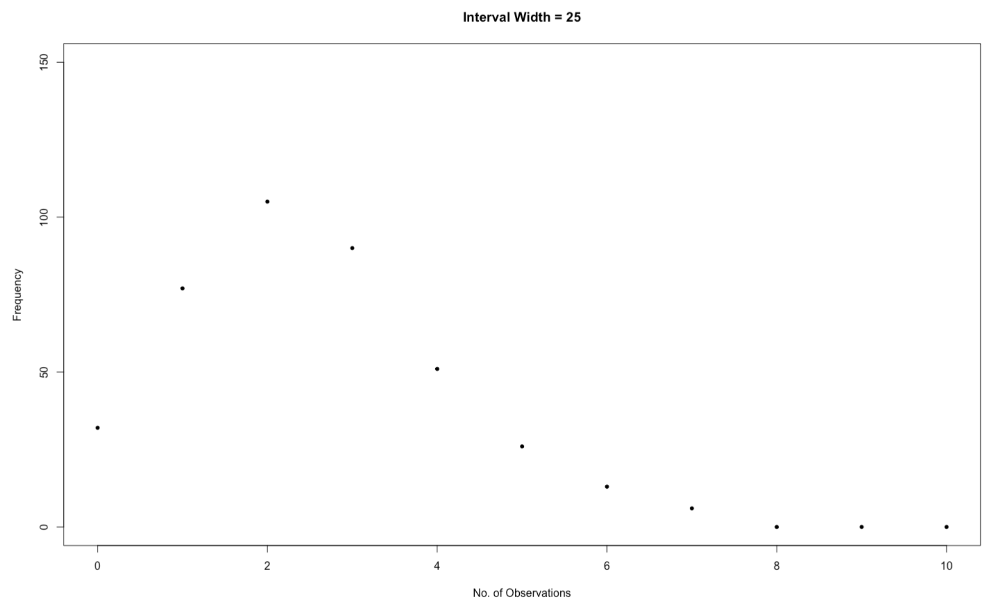
As we can see the mean is very close to the theoretical value and while the variance is not as close as the mean, it is still quite close to the value we would expect.
We can repeat this process but with different sized intervals and adjust the theoretical distribution accordingly.
Firstly, we will use an interval width of 100.

```
my.breaks2 = seq(0,10000,by=100)
counts_sample2 = as.vector(table(cut(jump.times,breaks=my.breaks2)))
obvs_sample2 = c(0:25)
freq_sample2 = vector(length = 26)
for (i in 0:25){
  for (j in 1:100){
    if (counts_sample2[j] == i){
      freq_sample2[i+1] = freq_sample2[i+1] + 1
    }
  }
}
plot(obvs_sample2, freq_sample2, ylim = c(0,20), pch = 20, ylab = 'Frequency', xlab = 'No. of Observations',
     main = 'Interval Width = 100')
```



Here we have a plot of the number of observations seen in an interval on the x-axis and the frequency at which it occurred.
Similar to how we calculated the theoretical distribution before, we know that for an interval width of 100, we should expect the distribution to be $Po(0.1 \times 100) \sim Po(10)$ with mean = 10 and variance = 10.
We can change the interval width and compare with the theoretical distribution again, this time we will change the interval width to 25, giving us a theoretical $Po(2.5)$ distribution. The mean and variance for this distribution both being 2.5.

```
my.breaks3 = seq(0,10000,by=25)
counts_sample3 = as.vector(table(cut(jump.times,breaks=my.breaks3)))
table(counts_sample3)
obvs_sample3 = c(0:10)
freq_sample3 = vector(length = 11)
for (i in 0:10){
  for (j in 1:400){
    if (counts_sample3[j] == i){
      freq_sample3[i+1] = freq_sample3[i+1] + 1
    }
  }
}
plot(obvs_sample3, freq_sample3, ylim = c(0,150), pch = 20, ylab = 'Frequency', xlab = 'No. of Observations',
     main = 'Interval Width = 25')
```



Using the same process as before, we see that the plot has its highest value at 2, this is what we would expect for this distribution with a mean of 2.5.

1.3 Combining two Poisson processes

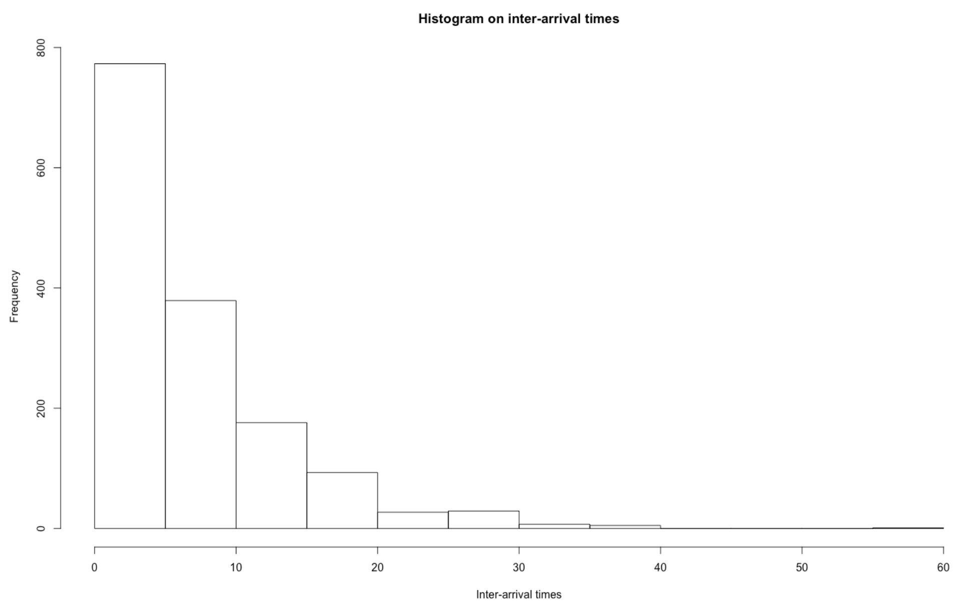
For this section of the practical we will use two separate Poisson processes, simulating jump times in a similar way to the last section and then combine them to form a single Poisson process. We will then convert these jump times back into inter-arrival times and compare this simulated Poisson process to see if it is similar to our expected distribution.

```
T1 = rexp(50^2,0.1)
sum(T1)
J1 = cumsum(T1)
jump.times1 = vector()
for (i in 1:length(J1)) {
  if (J1[i] <= 10000)
    jump.times1[i] = J1[i]
}
tail(jump.times1)
```

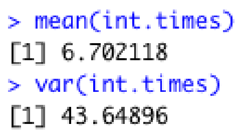
```
T2 = rexp(50^2,0.05)
sum(T2)
J2 = cumsum(T2)
jump.times2 = vector()
for (i in 1:length(J2)) {
  if (J2[i] <= 10000)
    jump.times2[i] = J2[i]
}
tail(jump.times2)
```

```
jump.times = c(jump.times1,jump.times2)
jump.times = sort(jump.times)
int.times = jump.times[2:length(jump.times)]-jump.times[1:length(jump.times)-1]
```

In the code above we have used the same method as before to simulate jump times for two Poisson processes with parameters $\lambda = 0.1$ and $\lambda = 0.05$ respectively. Next have then combined both jump time vectors into one single vector and sorting them to place them in order. Finally, we have converted the jump times into inter-arrival times.
We can now also calculate the theoretical distribution of this combined Poisson processes. When we created the intial two, independent sets of inter-arrival times, we used the distributions $Exp(0.1)$ and $Exp(0.05)$. We can calculate the resulting distribution using the two distributions and the fact that the distributions are independent.
If $S_i \sim Exp(\lambda)$ and $T_i \sim Exp(\mu)$ independently for $i = 1, 2, \dots$ and we define $U_i = min\{S_i, T_i\}$. Then $U_i \sim Exp(\lambda + \mu)$ and the observations form a Poisson process.
Using this, $S_i \sim Exp(0.1), T_i \sim Exp(0.05) \implies U_i \sim Exp(0.15)$ with mean $= 1/0.15 \approx 6.67$ and variance $= 1/0.15^2 \approx 44.44$.
Using a histogram we can visualise the inter-arrival times of the simulated Poisson process.



From the histogram we see that most of the inter-arrival times occur below 0.2, as expected for a exponential distribution with parameter $\lambda = 0.15$.
We can also calculate the mean and variance of the data for comparison.



```
> mean(int.times)
[1] 6.702118
> var(int.times)
[1] 43.64896
```

Here we see that the mean and variance of the data is very similar to the theoretical mean and variance of 6.66 and 44.44 respectively.

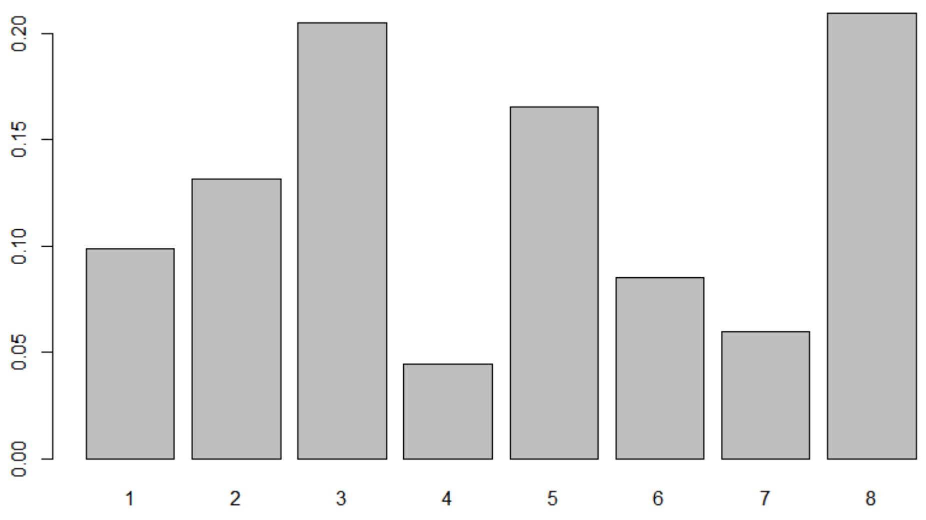
2 Finite state-space Markov processes (Practical 2)

2.1 The Rate Matrix Q

We know that a probability vector π is a stationary distribution if $\pi Q = 0$. By taking the transpose of this equation,

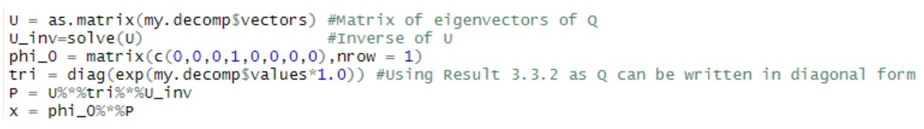
$$(\pi Q)^T = 0, \qquad Q^T \pi^T = 0$$

we can see that π^T is a column eigenvector of Q^T. We can use this to find and plot the stationary distribution of Q.



2.2 The distribution of X(t)

To plot X(t) we first had to calculate U, to enable us to compute the transition matrix P(t). Using the transition matrix we were able to create a function computing the distribution of X(1.0).



```
U = as.matrix(my.decomp$vectors) #Matrix of eigenvectors of Q
U_inv=solve(U)                   #Inverse of U
phi_0 = matrix(c(0,0,0,1,0,0,0,0),nrow = 1)
tri = diag(exp(my.decomp$values*1.0)) #Using Result 3.3.2 as Q can be written in diagonal form
P = U%*%tri%*%U_inv
x = phi_0%*%P
```

We then used this to create four barplots showing the distribution of X(t) for $t = 0.5, 1.0, 2.0, 4.0$.

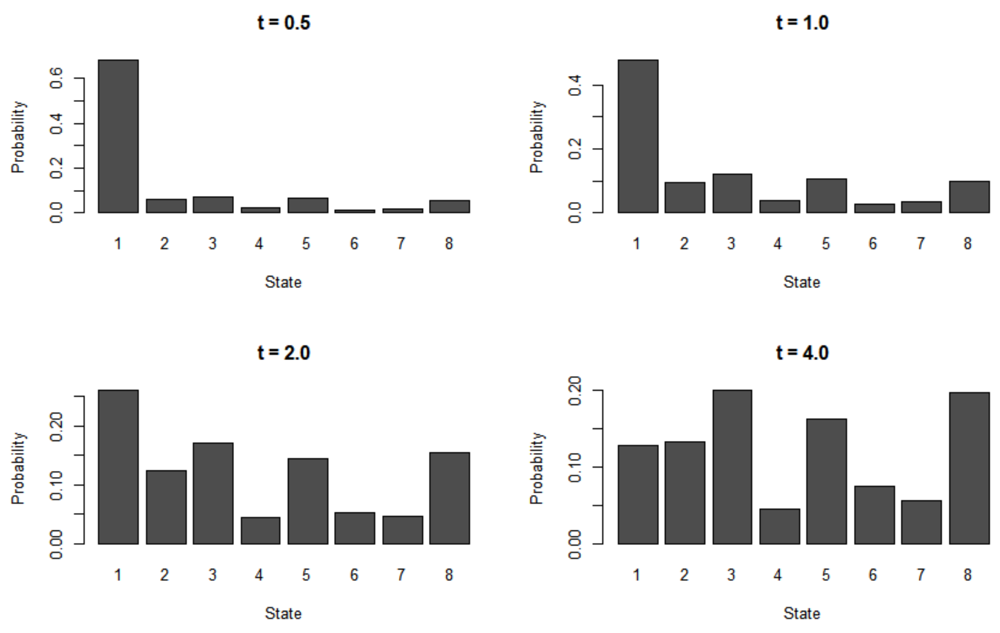


Figure 2: Distribution for $X(t)$ when $X(0) = 1$.

We ran this function again but made X(0) = 4 be the initial condition. As before $t = 0.5, 1.0, 2.0, 4.0$.

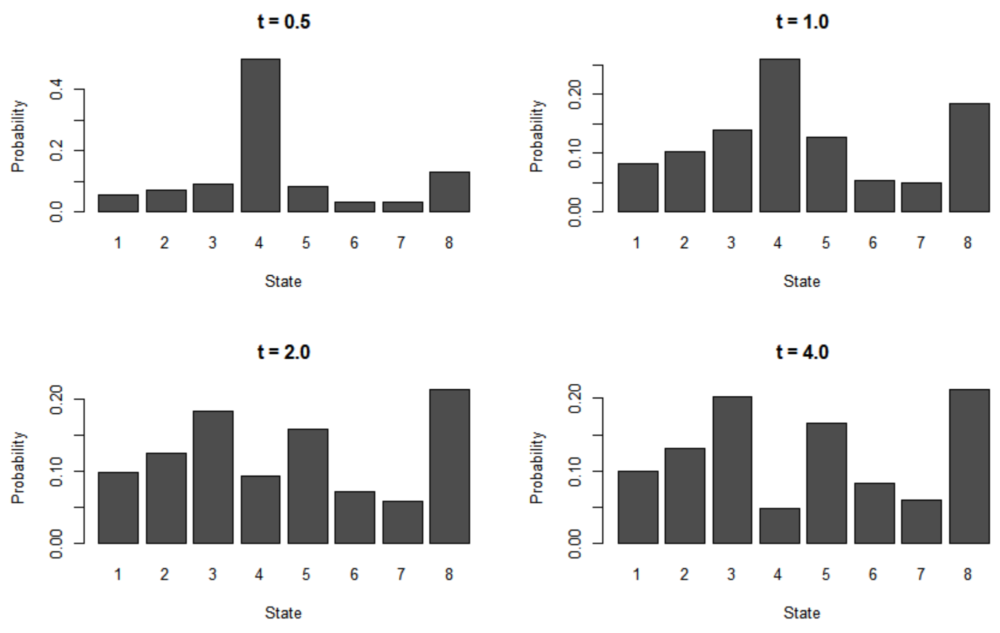


Figure 3: Distribution for $X(t)$ when $X(0) = 4$.

We can see in Figure 2, that for low values of t, we expect a high probability to be in the state that we started in, in comparison to the other states. This is the case for both initial conditions as shown in Figure 3. However, as time t increases, we expect the probabilities of being in each state to spread out. In other words, we are evenly likely to end up in each state for larger values of t.

2.3 Simulating Realizations

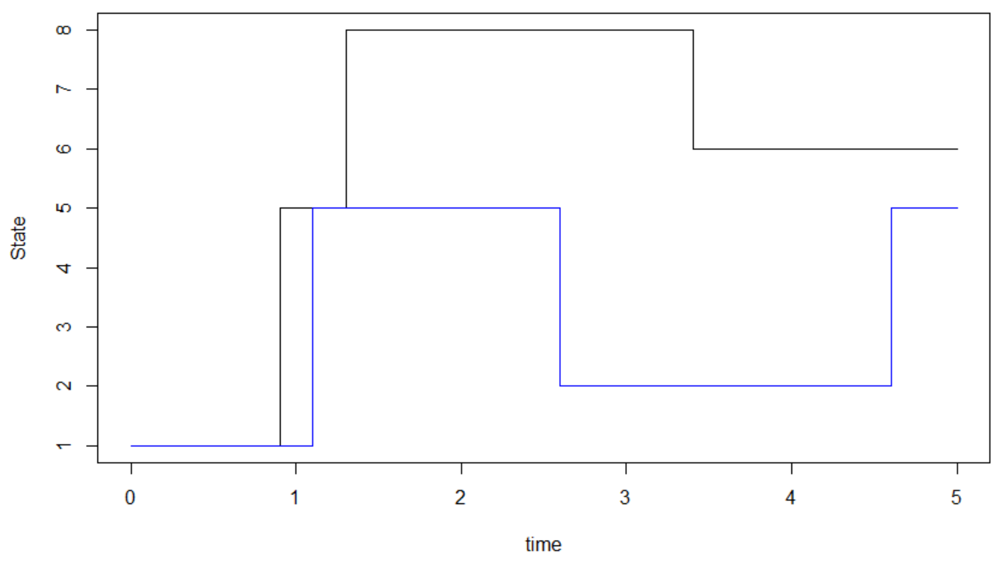
We want to simulate realizations of X(t) for a set of times $t = 0, \Delta t, 2\Delta t, ..., n\Delta t$, and we know P($\Delta t$). We x_k denote the simulated value of X($k\Delta t$). Then we initialize the algorithm with a fixed initial state X(0). Then, if we have simulated realizations $x_1, x_2, ..., x_{k-1}$,and$x_{k-1} = i$ then the distribution at the next time step is:

$$Pr(X(k\Delta t) = j|X((k-1)\Delta t) = i) = p_{xj}(\Delta t)$$

If we simulate from this distribution we obtain the realization x_k. Here is some R code for performing this algorithm,

```
deltat = 0.1; n = 50;        #the change in time and the number of interations
d = eigen(Q)
P = d$vectors %*% diag(exp(deltat*d$values)) %*% solve(d$vectors)  #Calculating the transition matrix based on Q
x = vector("numeric",n+1)    #initalise the output vector
x[1] = 1                     #intial state
for (k in 1:n) {
  xi = P[x[k],]              #probalilty of moving from the current state to the others(including the original state)
  i = sample(1:8,1,prob=xi)  #selects a state based on the probalities calculated
  x[k+1] = i                 #stores this state in a vector
}
```

Next we used the code to simulate two realizations of X(t) for $t\epsilon[0, 5.0]$, starting from X(0) = 1.

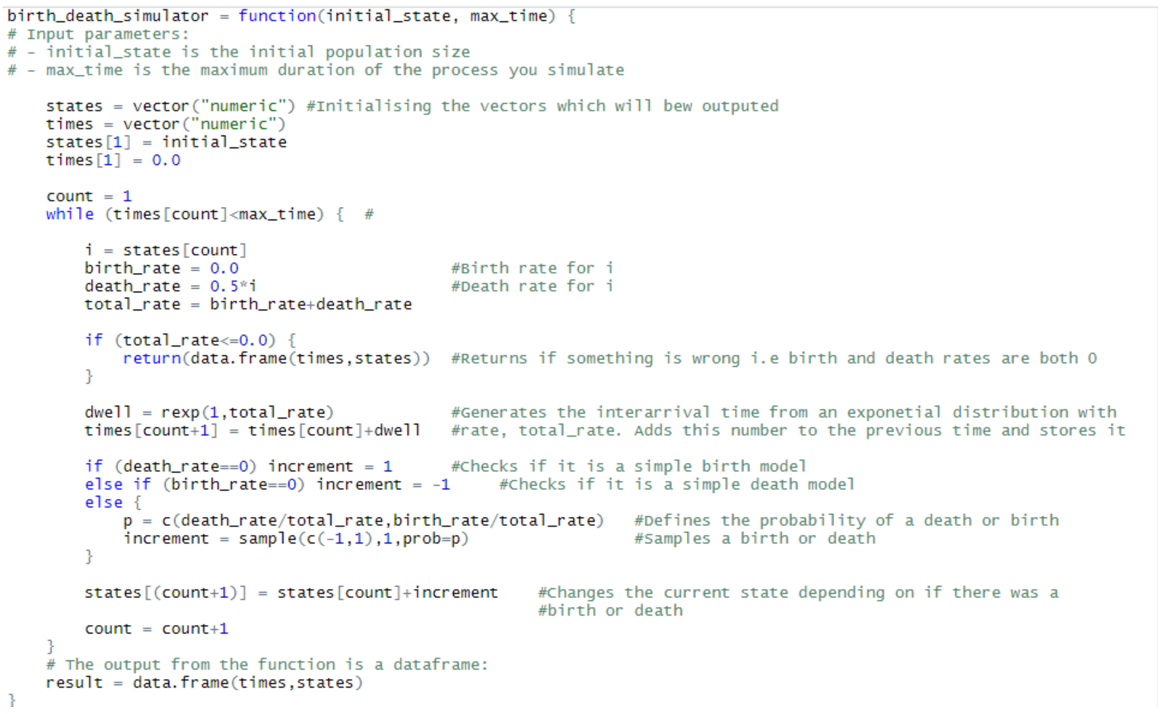


3 Simulating birth-death processes (Practical 3)

In this section we will simulate realizations from a variety of birth-death models to investigate their behaviour. As a basis we will use the R function birth-death-simulator, this will be edited as we simulate different realizations.

3.1 Birth Death Simulator

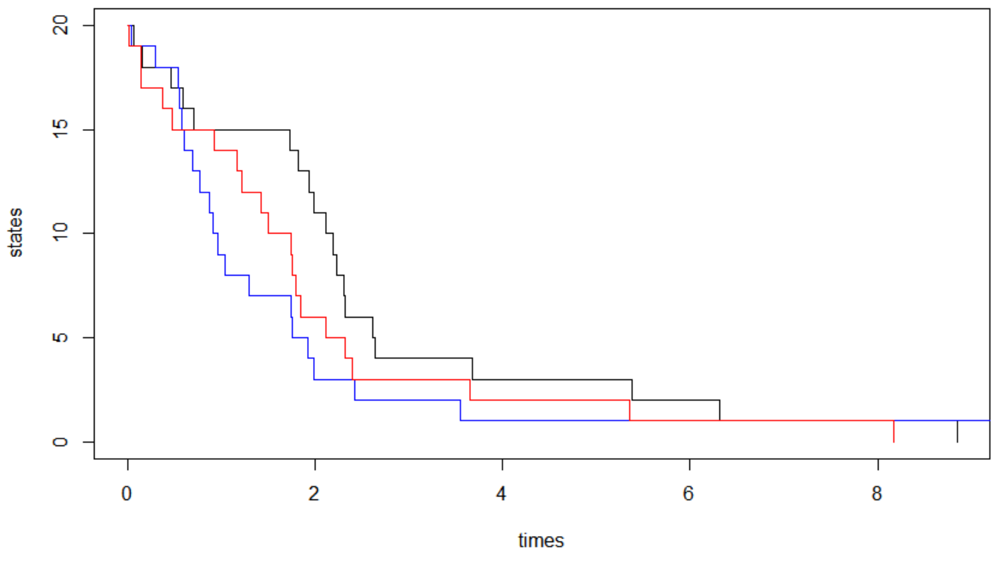
To start I will show the R function we will be using, as well as explaining how it simulates an arbitrary birth-death process.



```r
birth_death_simulator = function(initial_state, max_time) {
# Input parameters:
# - initial_state is the initial population size
# - max_time is the maximum duration of the process you simulate

    states = vector("numeric") #Initialising the vectors which will bew outputed
    times = vector("numeric")
    states[1] = initial_state
    times[1] = 0.0

    count = 1
    while (times[count]<max_time) {  #

        i = states[count]
        birth_rate = 0.0                     #Birth rate for i
        death_rate = 0.5*i                   #Death rate for i
        total_rate = birth_rate+death_rate

        if (total_rate<=0.0) {
            return(data.frame(times,states))  #Returns if something is wrong i.e birth and death rates are both 0
        }

        dwell = rexp(1,total_rate)           #Generates the interarrival time from an exponetial distribution with
        times[count+1] = times[count]+dwell  #rate, total_rate. Adds this number to the previous time and stores it

        if (death_rate==0) increment = 1     #Checks if it is a simple birth model
        else if (birth_rate==0) increment = -1     #Checks if it is a simple death model
        else {
            p = c(death_rate/total_rate,birth_rate/total_rate)    #Defines the probability of a death or birth
            increment = sample(c(-1,1),1,prob=p)                  #Samples a birth or death
        }

        states[(count+1)] = states[count]+increment     #Changes the current state depending on if there was a
                                                        #birth or death
        count = count+1
    }
    # The output from the function is a dataframe:
    result = data.frame(times,states)
}
```

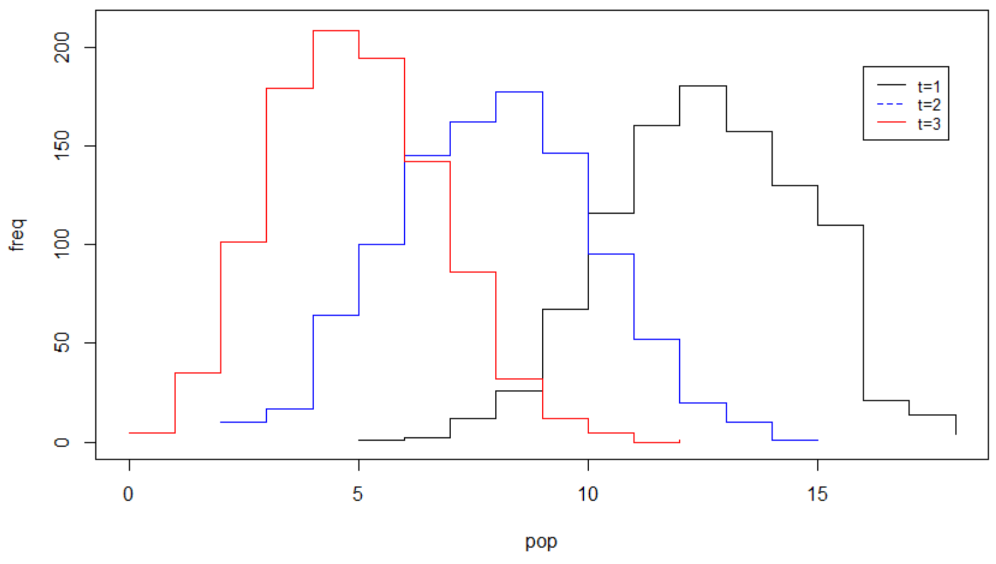
3.2 Simple Death Process

Here we ran birth-death-simulator three times with an initial population of 20, maximum time 100, a birth rate $\lambda = 0$ and a death rate $\mu_i = 0.5 \times i$. We then plotted the results off all three on one graph.



As we can see the plot are all fairly similar and all eventually end up in state 0. This is to be expected as there is 0 birth rate, so it can only go down states.

Next we used repeat-simulations a function which runs birth-death-simulator multiple times and returns the different population sizes that occurred at time max time as well as the frequency with which that population level occurred. We ran birth-death-simulator 1000 times for an initial population 20 and obtain a plot showing the population distribution at time $t = 1$ for these 1000 simulations, we then ran it for $t = 2$ and $t = 3$.

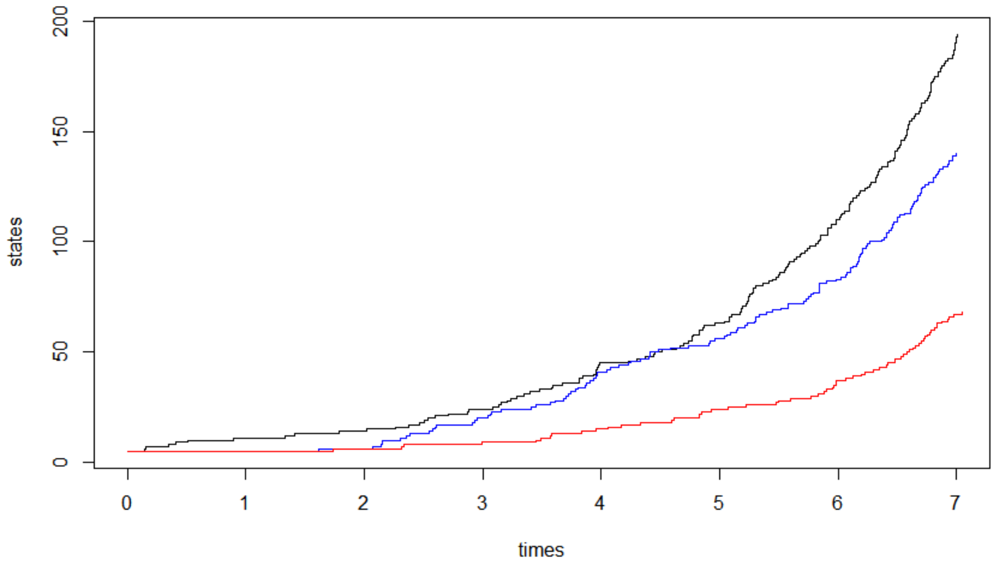


The simulation shows that as t increases, the frequency at which we have a small population increases. In other words, increasing time t, decreases the population of our simple death process. This can be highlighted by looking at the highest frequency for $t = 3$, which is at a population of less than 5 and frequency of above 200, and comparing it with the highest frequency for $t = 1$, which is at a population of around 13 and a frequency of roughly 175.

At time t the simple death process has a binomial distribution: X(t)-Bin(n, p), where $p = e^{-\mu t}$, so theoretical distributions for $t = 1$, $t = 2$ and $t = 3$ are X(1)-Bin($n, e^{-0.5}$), X(2)-Bin(n, e^{-1}) and X(3)-Bin($n, e^{-1.5}$).

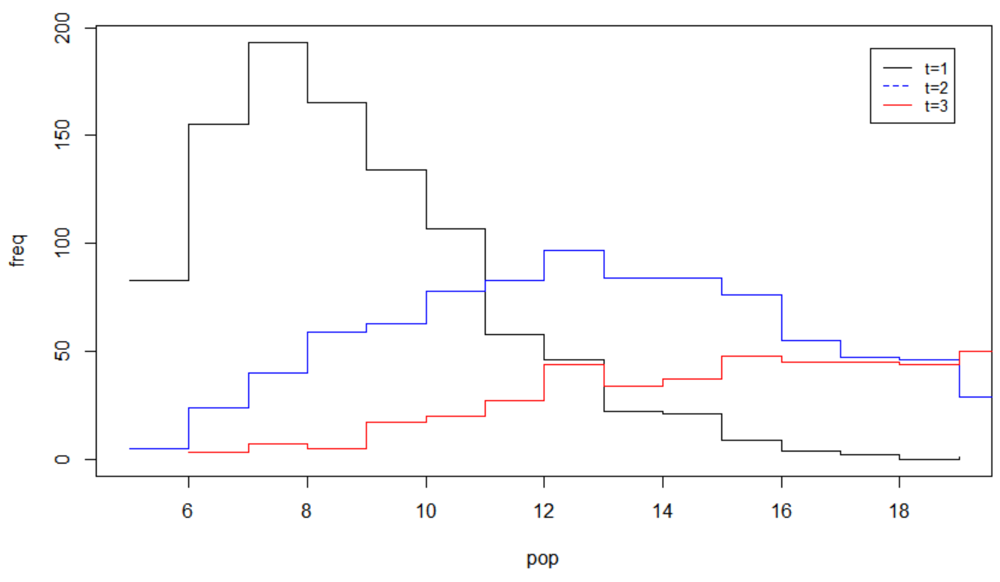
3.3 Simple Birth Process

Here we ran birth-death-simulator three times with an initial population of 20, maximum time 100, a birth rate $\lambda_i = 0.5 \times i$ and a death rate $\mu = 0$. We then plotted the results off all three on one graph.



As we can see the plot are all fairly similar and continue to increase . This is to be expected as there is 0 death rate, so it can only go up states.

We ran birth-death-simulator 1000 times for an initial population 5 and obtain a plot showing the population distribution at time $t = 1$ for these 1000 simulations, we then ran it for $t = 2$ and $t = 3$.

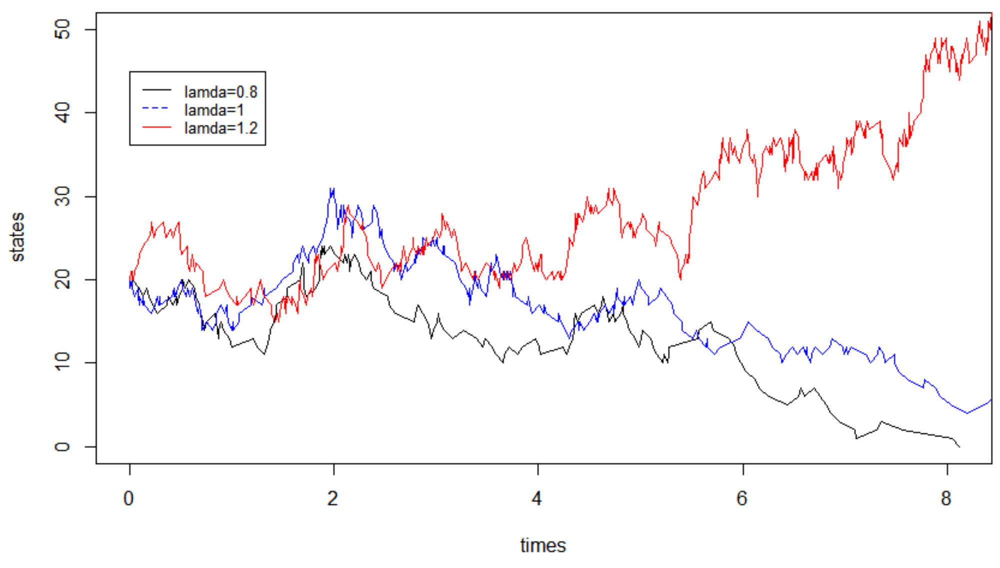


From the plot we see that the distribution is more varied as t increases. We also see that the mode increases with t as well.

The simple birth process X(t) with initial condition X(0) $= n$ has a theortical mean of $ne^{\lambda t}$. The would give our plots means of $5e^{0.5t}$, which to $t = 1, 2, 3$ is equal to 8.2436, 13.5914 and 22.4084. The means of the simulations are 8.373, 13.396 and 22.041 respectively, which are all very close to there theoretical counterparts.

3.4 Simple Birth and Death Process

Here we ran birth-death-simulator three times with an initial population of 20, maximum time 100, a birth rate $\lambda_i = \lambda \times i$ and a death rate $\mu_i = 1 \times i$. We then plotted the results for $\lambda = 0.8$,$\lambda = 1$ and $\lambda = 1.2$.



Increasing λ will increase the population growth. This is due to the fact that our death rate μ is not changing, but our value for λ is increasing. Therefore, we would expect higher values of λ to lead to increased population growth, which is highlighted in the plot above.

The general decline for $\lambda = 1$ may just be due to chance as there is an equal birth and death rate, but we can see the line starting to increase after time 8. Our observation for $\lambda = 0.8$ it to be expected, as we have a higher death rate than birth rate. Therefore, our system is expected to reach state 0, i.e. the population dies.

4 SIR and Lotka-Volterra models (Practical 4)

I will not be going into extensive detail as to how these models function. But instead, I shall be showing the outputs/plots of these processes and how changing the different parameters will affect the outcome.

4.1 The SIR model

4.1.1 Simulations

To show what the SIR model looks like and describes, we shall look at some simulations of the model with parameters $\beta = 2$ and $\gamma = 1$. We receive the plot for this SIR model with a fixed population size of $N = 100$ and just one infected individual at the outset,

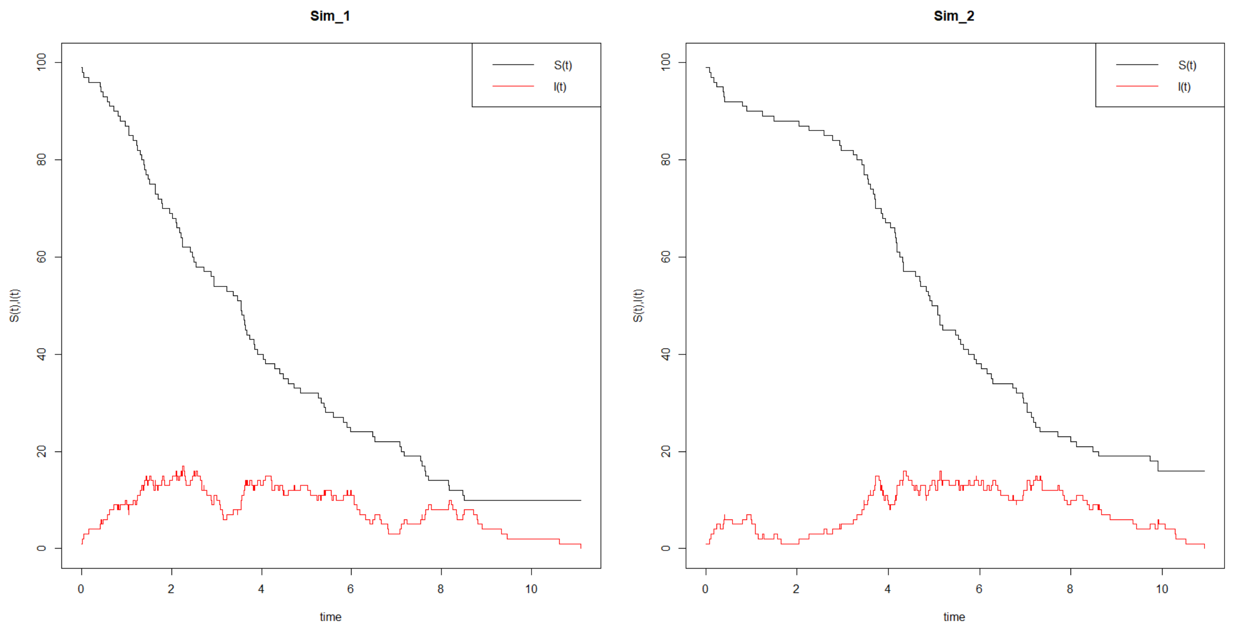


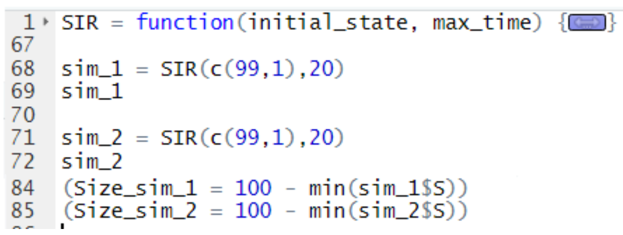
Figure 4: 2 different simulations for the SIR model with $\beta = 2$, $\gamma = 1$ and $N = 100$ starting with one infection on outset.

where $S(t)$ is the number of people susceptible to infections and $I(t)$ is the number of infected individuals. We can clearly see that as time increases, the number of people susceptible to the disease decreases. The number of infected people does not show a gradual increase as people recover from the disease. Therefore this evolution of the disease can be shown as

$$S(t) + I(t) + R(t) = N$$

where $R(t)$ are the individuals who have recovered from the disease and are immune to it.

We can calculate the epidemic size by subtracting the number of susceptible individuals at the end of the simulation from the population size, N. This can be calculated easily in R:

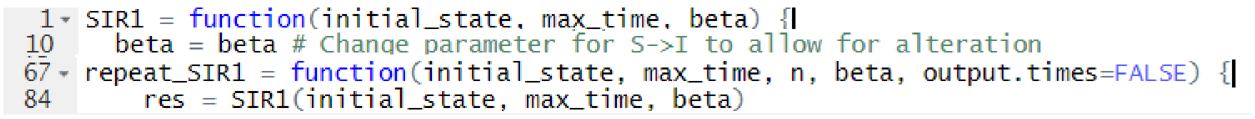


```
SIR = function(initial_state, max_time) {}

sim_1 = SIR(c(99,1),20)
sim_1

sim_2 = SIR(c(99,1),20)
sim_2
(Size_sim_1 = 100 - min(sim_1$S))
(Size_sim_2 = 100 - min(sim_2$S))
```

Using this code we can see that our first simulation has a final epidemic size of 90 and our second simulation has a final epidemic size of 84. We use min(sim_1$S) as the minimum number of susceptible individuals will always be at the end of the simulation.

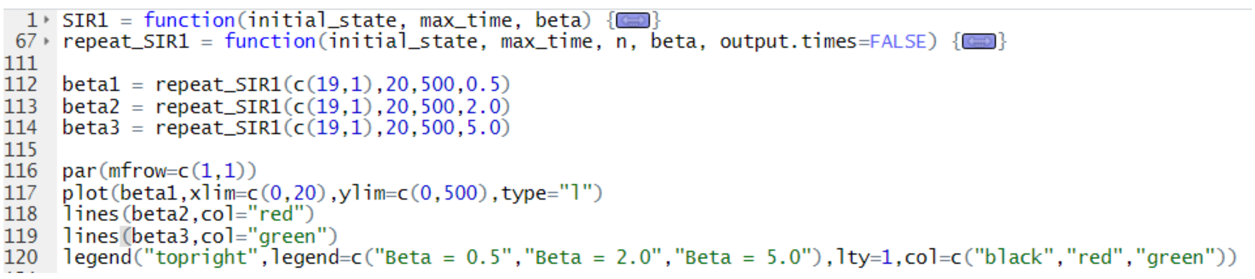
4.1.2 Size and duration of epidemics

To look at the size and duration we shall use the SIR and repeat_SIR functions. However, we can alter them to allow us to readily change the parameters, for β and γ. Below is the following changes within the functions, as well as, the program list for each β argument.



```
SIR1 = function(initial_state, max_time, beta) {
  beta = beta # Change parameter for S->I to allow for alteration
repeat_SIR1 = function(initial_state, max_time, n, beta, output.times=FALSE) {
    res = SIR1(initial_state, max_time, beta)
```

Figure 5: Adjustments made to SIR and repeat_SIR functions.



```
SIR1 = function(initial_state, max_time, beta) {}
repeat_SIR1 = function(initial_state, max_time, n, beta, output.times=FALSE) {}

beta1 = repeat_SIR1(c(19,1),20,500,0.5)
beta2 = repeat_SIR1(c(19,1),20,500,2.0)
beta3 = repeat_SIR1(c(19,1),20,500,5.0)

par(mfrow=c(1,1))
plot(beta1,xlim=c(0,20),ylim=c(0,500),type="l")
lines(beta2,col="red")
lines(beta3,col="green")
legend("topright",legend=c("Beta = 0.5","Beta = 2.0","Beta = 5.0"),lty=1,col=c("black","red","green"))
```

Figure 6: Program list for changing β.

The program list above produces the following plot:

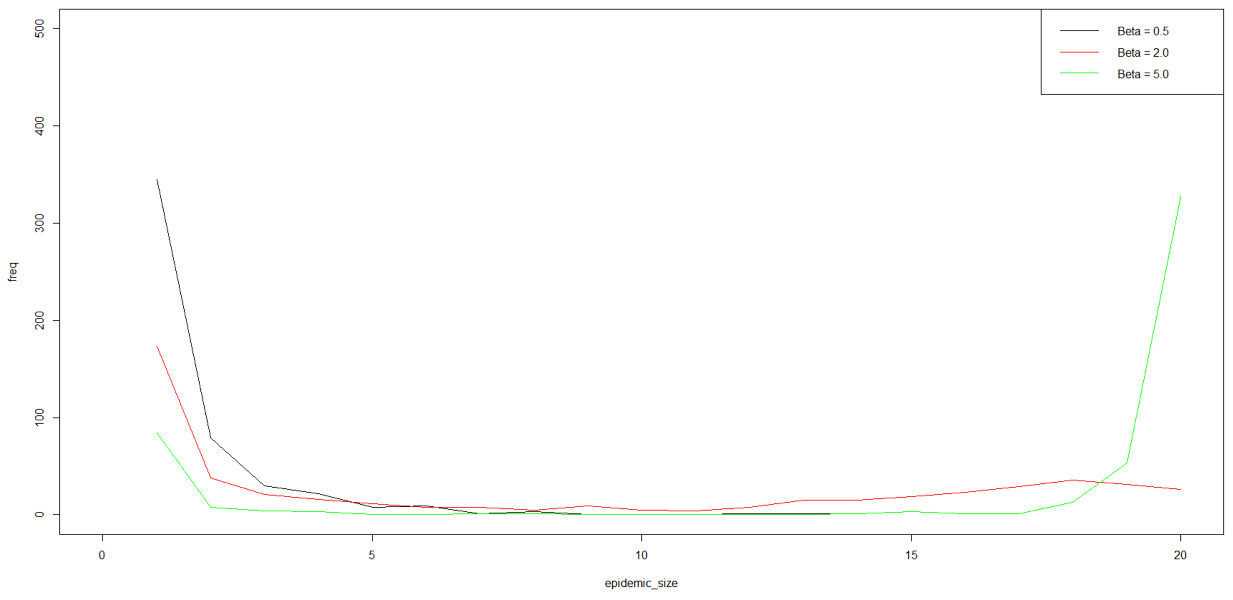


Figure 7: Program list for changing β.

As we can clearly see, the value of β changes the epidemic size of each iteration significantly. In fact, as β increases, the higher the frequency at which all of the population becomes infected for each simulation. For $\beta = 0.5$, we can see that our simulations finish early with only a few ending with a final epidemic size of 14. However, increasing β by 1.5 causes most of the simulations to shift towards the further end of the epidemic size and less simulations end with a low epidemic size. The same result is observed when β is increased by 3, but more drastically.

4.2 The Lotka-Volterra predator-prey model

4.2.1 Simulations

Using our **predator_prey** function, we can run multiple simulations for this model with constant rate parameters $c_1 = 1.0$, $c_2 = 0.005$ and $c_3 = 0.6$. We will run these simulations with an initial condition of 100 predators and 50 prey. We can then plot the results to see what the frequency of predators and prey look like as time goes on.

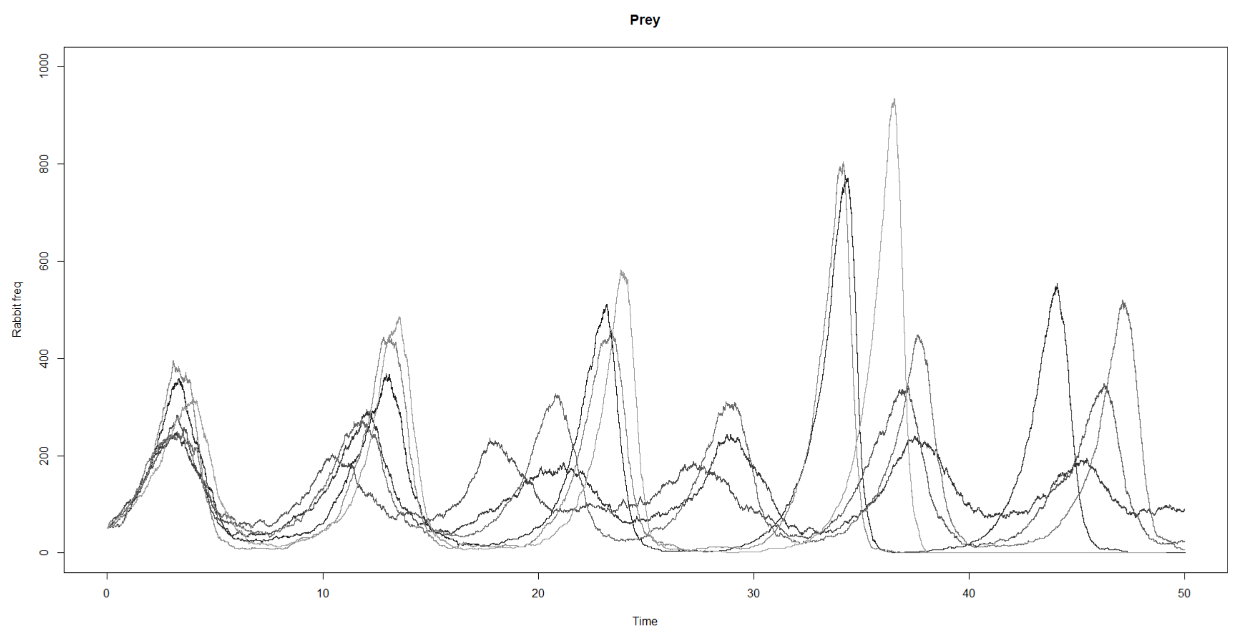


Figure 8: Grey plot of 5 trajectories for Prey.

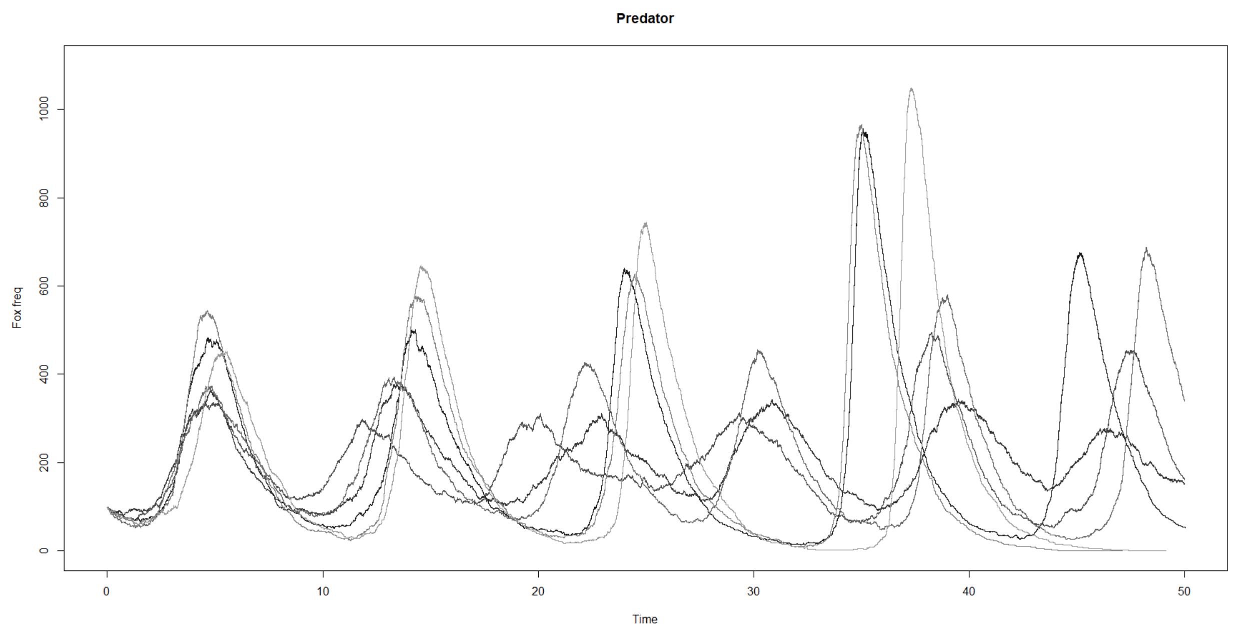


Figure 9: Grey plot of 5 trajectories for Predator.

From looking at the above figures, we can see that when rabbit frequency increases, the fox population increases shortly after which causes the prey population to decrease.

4.2.2 Changing parameters

Keeping $c_1 = 1$, but changing the other parameters will cause changes in the populations of both predators and prey. If we vary c_2, the population will change at a longer time frame. For example, the peak of rabbit frequency occurs after the peak for our normal rate of c_2, when we decrease c_2. The peak occurs earlier when we increase c_2. This can be seen more clearly below:

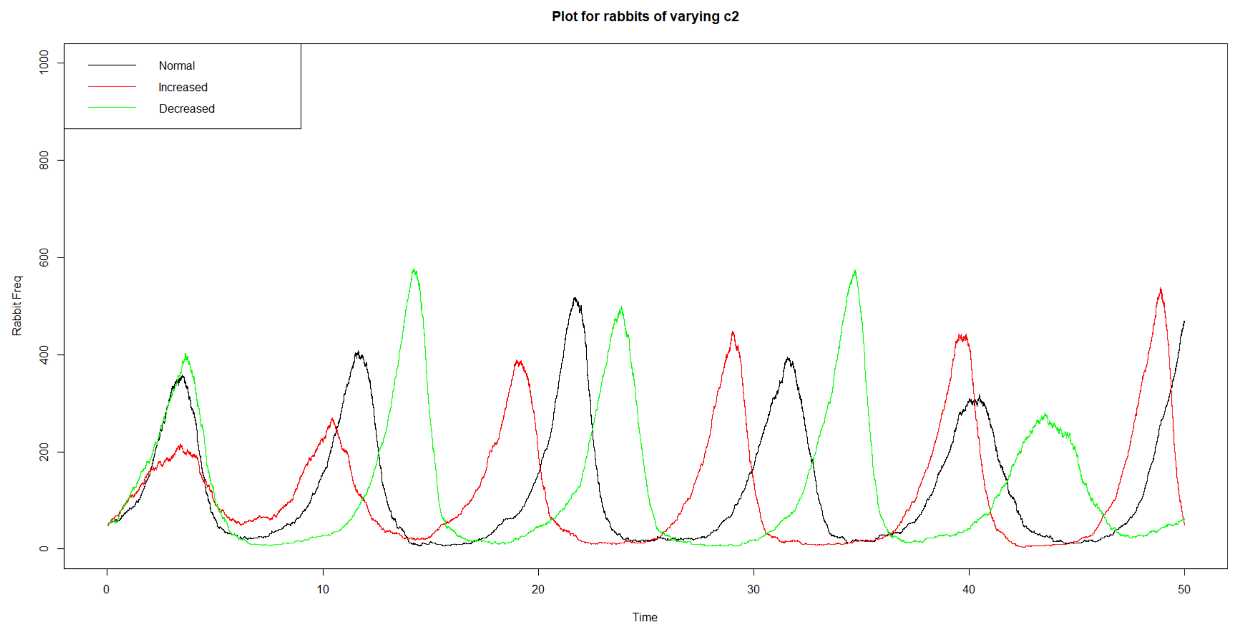


Figure 10: Plot of varying c_2 for prey.

We receive the same results with the predator population.

When changing c_3, we observe that our rate of increase/decrease changes. In other words, increasing c_3 will cause the overall population to increase more at faster/slower rate and vice versa. This can be shown more clearly below:

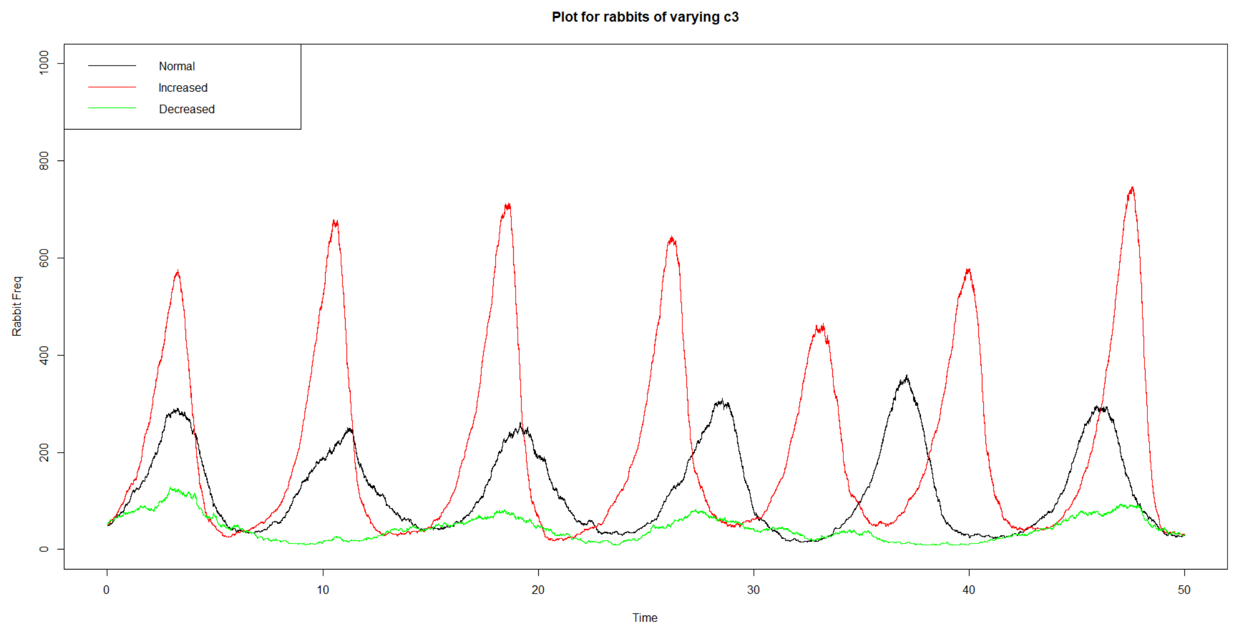


Figure 11: Plot of varying c_3 for prey.

Again, the same observation is shown for predators.

Changing both rate parameters will cause both effects to occur. We can check the dynamics of our population by using the **predator_prey_mean** function. We can then plot these for comparison.

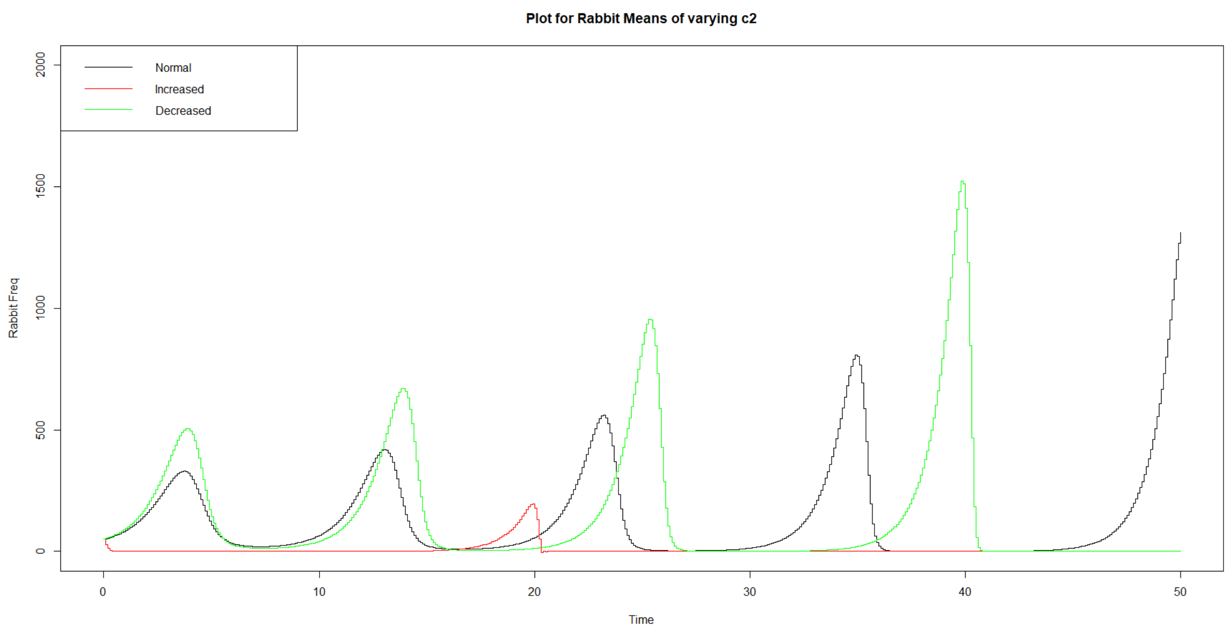
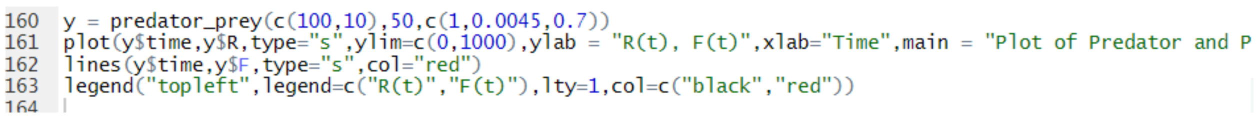


Figure 12: Plot for mean of varying c_2 for prey.

We can clearly see that varying c_2 will have the same effect on the mean as it does on the population.

To get a plot similar to that in the notes, we need to change our parameters according to the results that we have found from above. Therefore, if we need to find similar dynamic, but initially with a low predator population of 10 and high prey population of 100, we need to have a lower c_2 and a higher c_3 value. Shown from the following code:



```
y = predator_prey(c(100,10),50,c(1,0.0045,0.7))
plot(y$time,y$R,type="s",ylim=c(0,1000),ylab = "R(t), F(t)",xlab="Time",main = "Plot of Predator and P
lines(y$time,y$F,type="s",col="red")
legend("topleft",legend=c("R(t)","F(t)"),lty=1,col=c("black","red"))
```

We get the following plot:

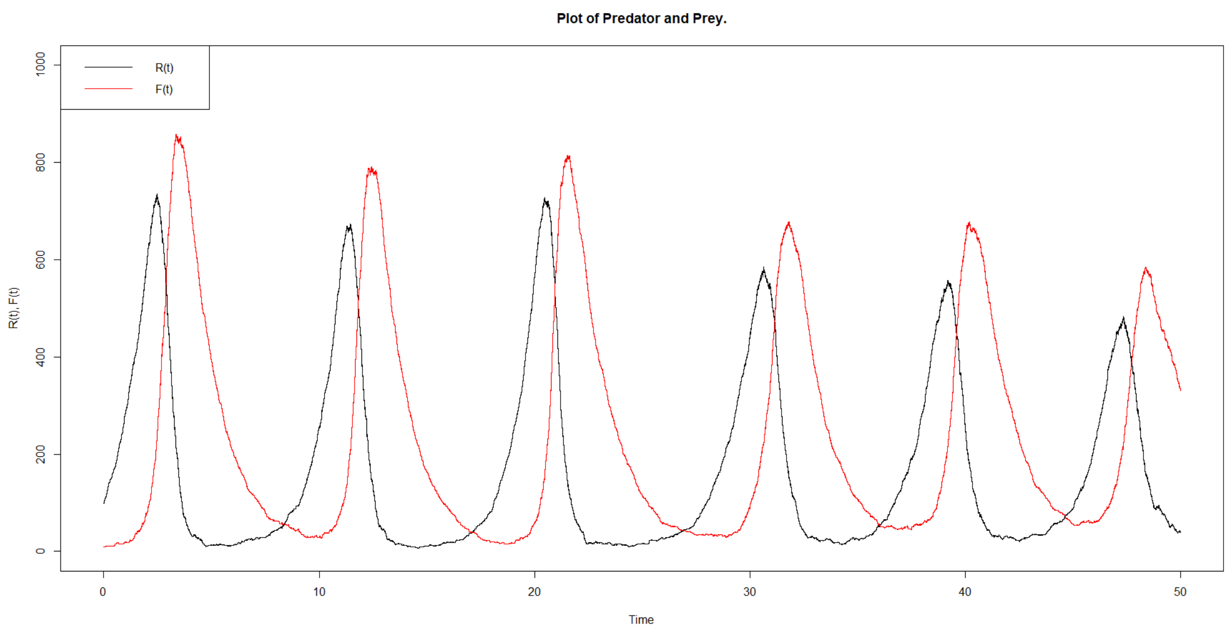


Figure 13: Plot similar to that in notes.