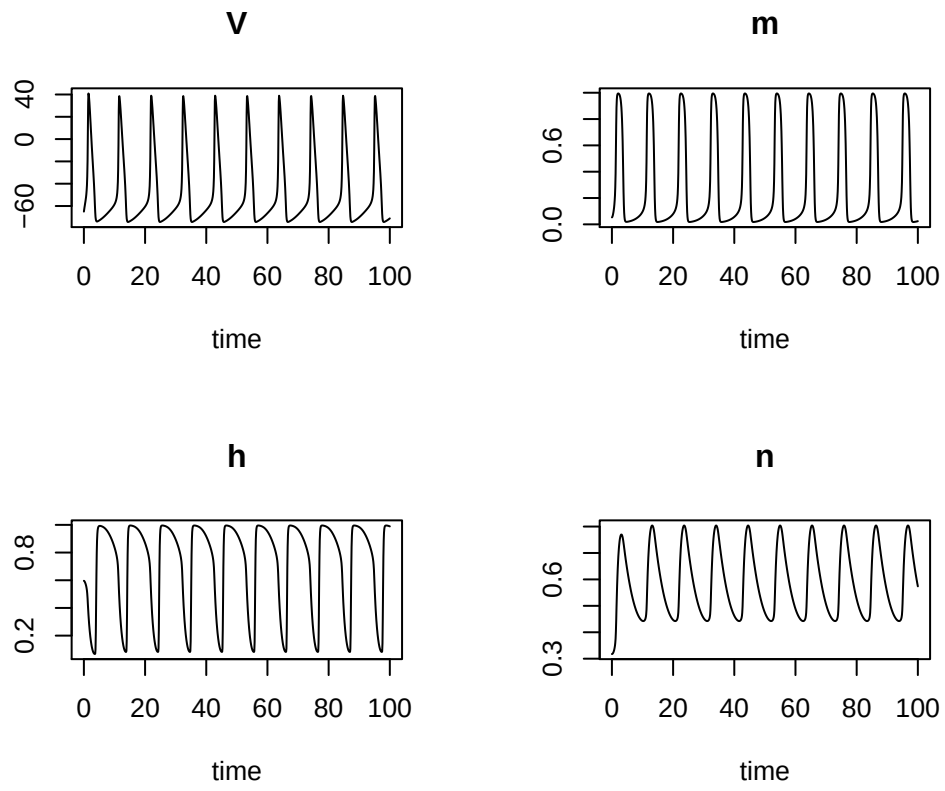
Computational Neuroscience Assignment 1

304365078

Hodgkin Huxley Model

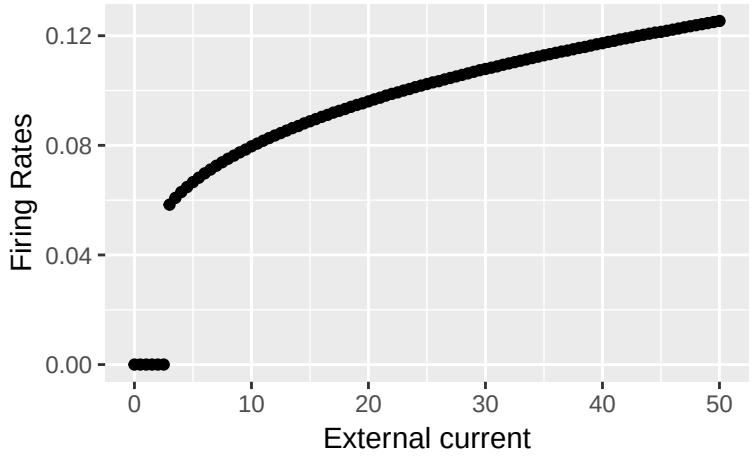
The following code builds a Hodgkin-Huxley model neuron and plots the values over time of V, m, n, h. We then plotted the firing rate of the model as a function of the external current and observed its behavior.

```
hodking_huxley <- function(t, init, parameters) {
  with(as.list(c(init, parameters)),{

    an <- function(V) {0.01*(V+55)/(1-exp(-0.1*(V+55)))}
    bn <- function(V) {0.125*exp(-0.0125*(V+65))}
    am <- function(V) {0.1*(V+40)/(1-exp(-0.1*(V+40)))}
    bm <- function(V) {4*exp(-0.0556*(V+65))}
    ah <- function(V) {0.07*exp(-0.5*(V+65))}
    bh <- function(V) {1/(1+exp(-0.1*(V+35)))}

    dV <- (I - gna*m^3*h*(V-Ena)-gk*n^4*(V-Ek)-gl*(V-El))/C
    dm <- am(V)*(1-m)-bm(V)*m
    dh <- ah(V)*(1-h)-bh(V)*h
    dn <- an(V)*(1-n)-bn(V)*n
    return(list(c(dV, dm, dh, dn)))
  })
}

parameters <- c(Ena=50, Ek=-77, El=-54.387, gna=120, gk=36, gl=0.3, C=1, I=20)
t <- seq(0, 100, by = 0.1)
init <- c(V=-65, m=0.0529, h=0.5961, n=0.3177)
#Numerically integrate
out <- ode(y = init, times = t, func = hodking_huxley, parms = parameters)
plot(out)
```




```r
#Plot v as a function of I over the range from 0 to 500
library(ggplot2)
change_I <- function(max_I) {
  seq_I <- seq(0, max_I, by = 0.5)
  firing_rate <- c()
  t <- seq(0, 100, by = 0.1)
  init <- c(V=-65, m=0.0529, h=0.5961, n=0.3177)
  for (i in 1:length(seq_I)) {
    parameters <- c(Ena=50, Ek=-77, El=-54.387, gna=120, gk=36, gl=0.3, C=1, I=seq_I[i])
    out <- ode(y = init, times = t, func = hodking_huxley, parms = parameters)
    out <- as.data.frame(out)
    found_peaks <- findpeaks(out[,2], threshold = 50)
    if (is.null(found_peaks)) {
      firing_rate[i] <- 0
    } else {
      time_differences <- c()
      for (j in 1:(nrow(found_peaks) -1)) {
        time_differences[j] <- out[found_peaks[j + 1, 2],1] -
          out[found_peaks[j,2],1]}
      firing_rate[i] <- (1 / mean(time_differences))
    }
  }
  return(firing_rate)
}

firing_rates <- change_I(50)
ggplot()+
  geom_point(aes(x = seq(0, 50, by = 0.5), y = firing_rates)) +
  xlab("External current") +
```

```
  ylab("Firing Rates")
```



The graph shows how the current jumps discontinuously from zero to a value when the current passes through the minimum value required to produce sustained firing.

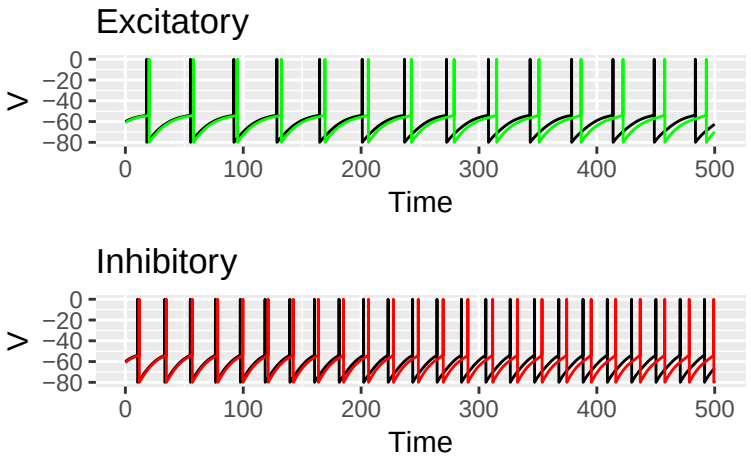
We then analyzed the output of our model when applying a negative current for 5ms followed by no current. Our model generated the graphs presented below, which show that...

Coupled integrate and fire neurons

We constructed a model of two coupled integrate-and-fire neurons given some intial conditions and parameters. Both neurons obey equation 5.43 [of TN] and both synapses are described with an alpha function (equation 5.35 of TN). We, more specifically, considered cases where both synapses were excitatory or inhibitory.

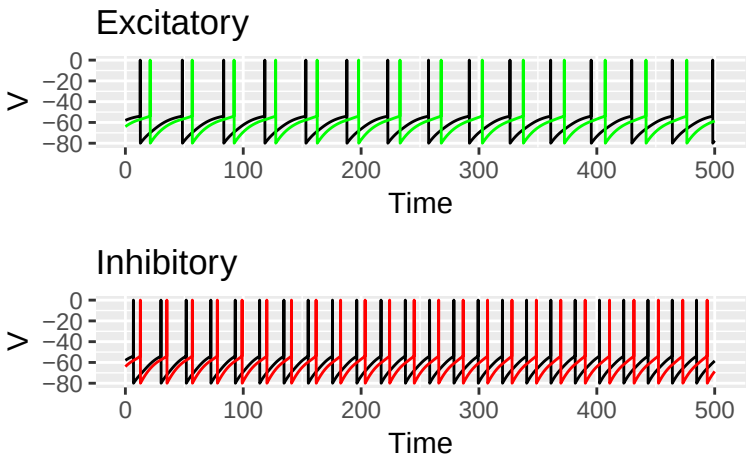
```
El <- -70
Vth <- -54 #threshold
Vres <- -80 #reset
tm <- 20
rmgs <- 0.15
RmIe <- 18
ts <- 10
P_max<- 0.5
t <- seq(0, 500, by = 0.1)

voltage <- matrix(0, nrow = 2, ncol = length(t) + 1)
voltage[,1] <- c(-60, -61)
Ps <- matrix(0, nrow = 2, ncol = length(t) + 1)
Ps[,1] <- c(0.5, 0.5)
z <- matrix(0, nrow = 2, ncol = length(t) + 1)
z[,1] <- c(0.1, 0.1)

synapses <- function(Es, tm = 20, ts = 10, rmgs = 0.15) {
  for (i in 1:(length(t))) {
    for (j in 1:2) {
      if(voltage[-j, i] >= Vth) {
        z[j, i+1] <- 1
        } else {
          dz <- -z[j, i] / ts
          z[j, i+1] <- z[j, i] + dz * (t[i+1] - t[i])
```

```
          }
      dPs <- (exp(1) * P_max * z[j, i] - Ps[j,i]) / ts
      Ps[j, i+1] <- Ps[j, i] + dPs * (t[i+1] - t[i])
      if (voltage[j, i] >= Vth) { #threshold and reset rule
        voltage[j, i+1] <- Vres
        } else {
          dV <- (El - voltage[j, i] -rmgs * Ps[j, i] * (voltage[j, i] - Es) + RmIe)/tm
          voltage[j, i+1] <- voltage[j, i] + dV * (t[i+1] - t[i])
        }
      }
  }
  voltage <- replace(voltage, voltage > Vth, 0)
  return(voltage)
}
```



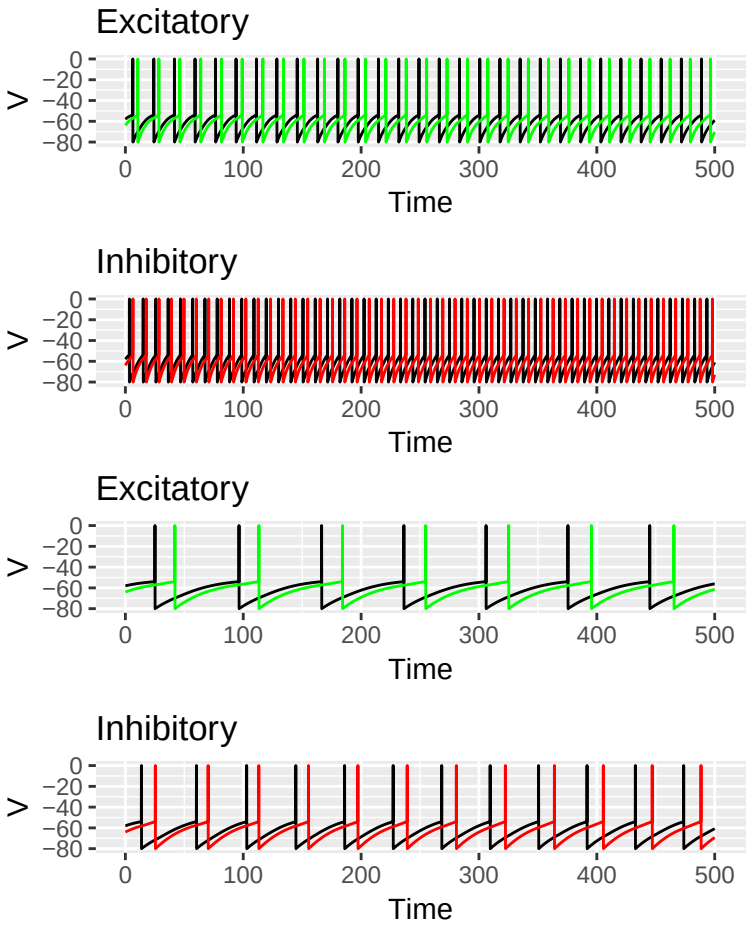
Next, we observed what happened to our model when modifying the inital conditions. We then investigated our model changing the time constants and strengths previously given. The graphs below show how the bigger the difference between the initial voltages, the more asynchronous the two neurons are.

```
#Changing the initial conditions
```



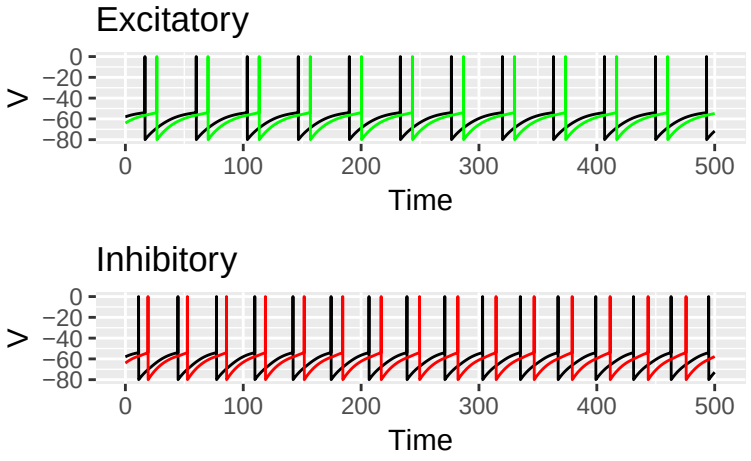
As we can see below, changing the time constants leads to a change in the spiking rate. When we lower the time constants we see an increase in spiking rate, but when we increase it the spiking rate gets lower.

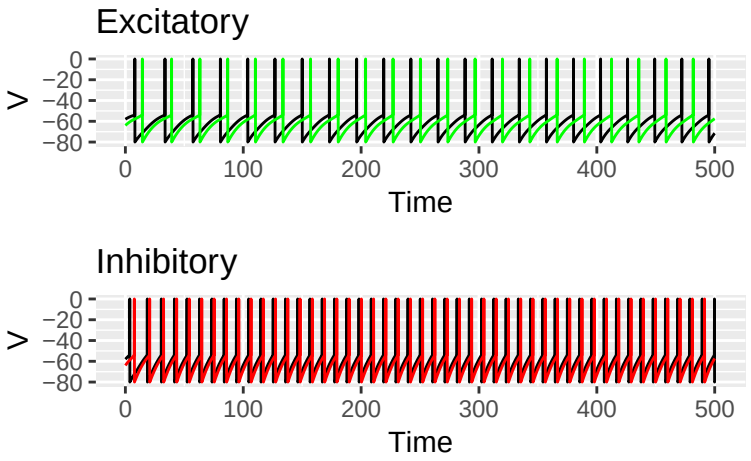
```
#Changing the time constants
```



On the other hand, an increase of strength leads us to a reduction of the spiking rate, and a reduction of the strength shows a higher spiking rate.

```
#Changing the strengths
```

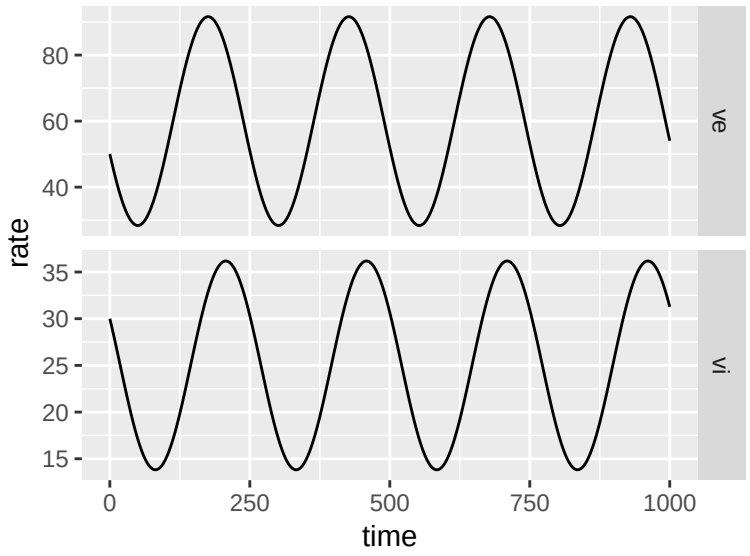




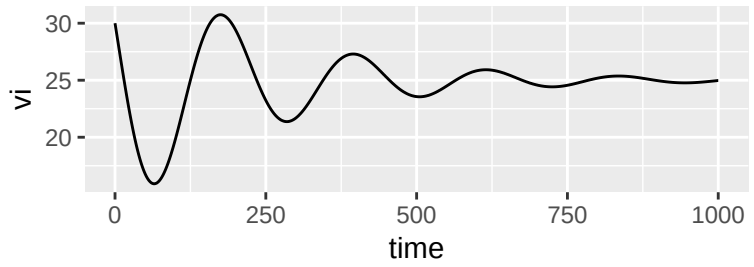
Networks

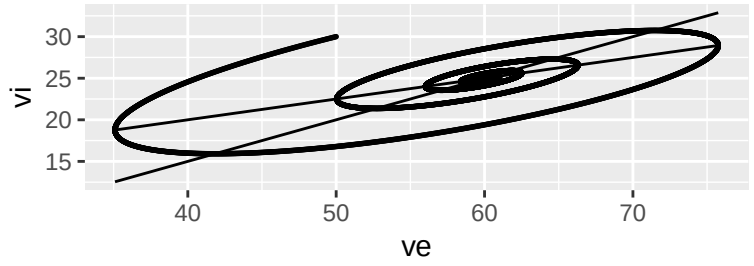
Thridly, we built and studied a simple model of oscillations arising from the interaction of excitatory and inhibitory populations of neurons. Below, our model will simulate different scenarios.

```
# Networks
networks <- function(t, init, parameters) {
  with(as.list(c(init, parameters)),{
    dve <- (-ve + (Mee * ve + Mei * vi - gamma_e))/te
    dvi <- (-vi + (Mii * vi + Mie * ve - gamma_i))/ti
    return(list(c(dve, dvi)))
  })
}

#Set parameters, initial values and time steps
parameters <- c(Mee = 1.25, Mie = 1, Mii = -1, Mei = -1, gamma_e = -10, gamma_i = 10,
                te = 10, ti = 80)
t <- seq(0, 1000, by = 0.1)
init <- c(ve = 50, vi = 30)

#Numerically integrate
networks_out <- ode(y = init, times = t, func = networks, parms = parameters)
networks_out <- as.data.frame(networks_out)
networks_clean <- networks_out %>% gather("v", "rate", 2:3)

ggplot(networks_clean, aes(fill=v, x=time, y=rate)) +
  facet_grid(v ~ ., scales = "free") +
  geom_line()
```

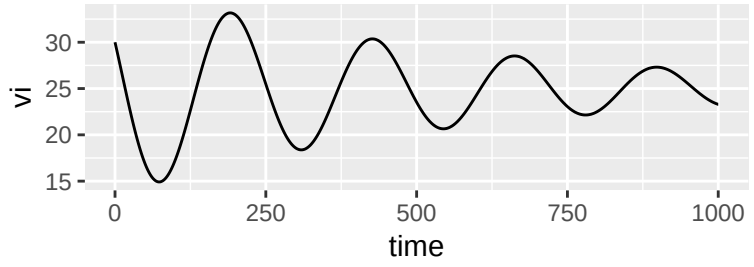


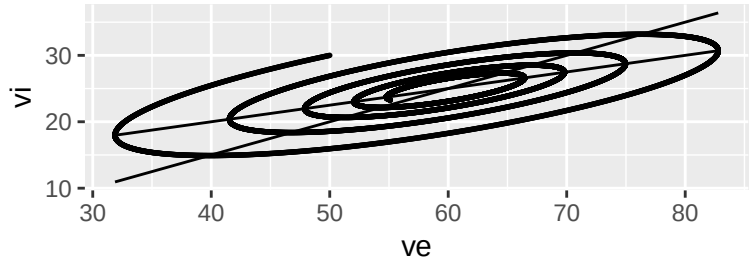

```
#Varying ti
var_ti <- function(ti) {
  parameters <- c(Mee = 1.25, Mie = 1, Mii = -1, Mei = -1, gamma_e = -10, gamma_i = 10,
                  te = 10, ti = ti)
  networks_out <- ode(y = init, times = t, func = networks, parms = parameters)
  networks_out <- as.data.frame(networks_out)

  vi_fun1 <- function(ve) {
    (gamma_e - ve*(Mee - 1))/Mei}

  vi_fun2 <- function(ve) {
    (gamma_i - (Mie * ve))/(Mii - 1)}

  require(gridExtra)
  p1 <- ggplot(networks_out, aes(time, vi)) + geom_line()
  gamma_e <- -10
  gamma_i <- 10
  Mee <- 1.25
  Mie <- 1
  Mii <- -1
  Mei <- -1

  p2 <- ggplot(networks_out, aes(ve, vi)) + geom_point(size = 0.3) +
    stat_function(fun = vi_fun1) +
    stat_function(fun = vi_fun2)
  grid.arrange(p1,p2, nrow = 2)
}

var_ti(60)
```



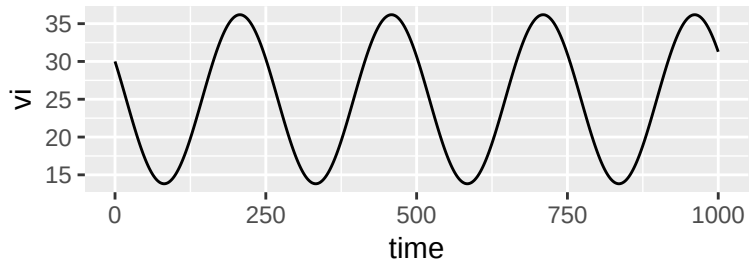


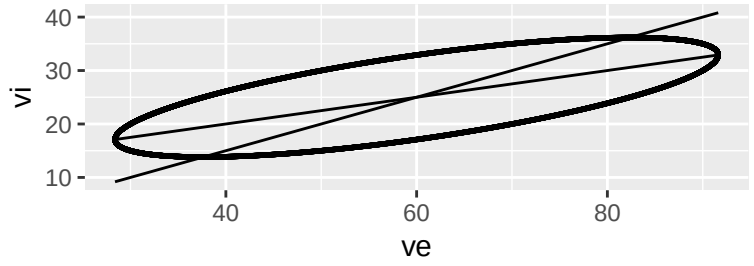

```
var_ti(70)
```



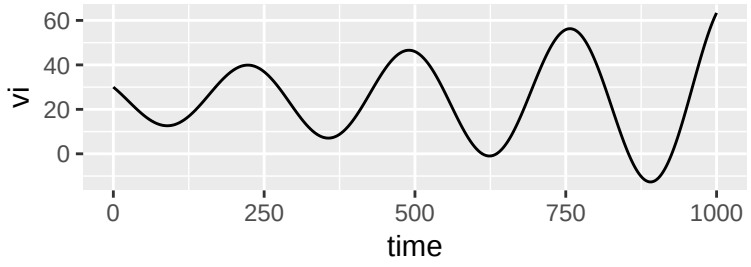


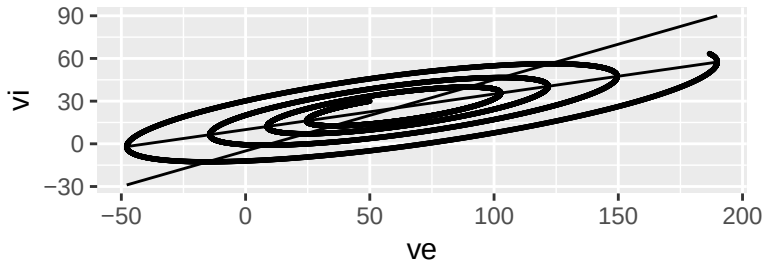

```
var_ti(80)
```



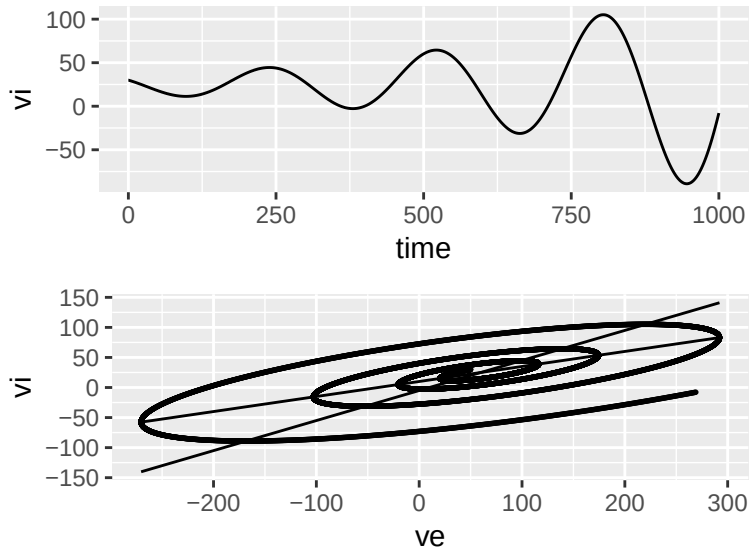



```
var_ti(90)
```






```
var_ti(100)
```

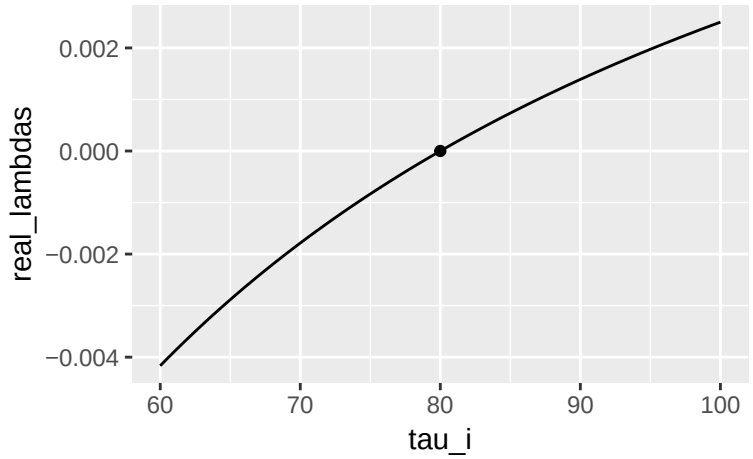


As the stability of the fixed point is determined by the real parts of the eigenvalues of the matrix found in TN equation 7.53, the following model predicts the value of τ. If the real parts of both eigenvalues are less than 0, the fixed point is stable, whereas if either is greater than 0, the fixed point is unstable.

```
#Predicting ti for stability
lambda <- function(ti) { #evaluate real and imaginary
  lambda1 <- (1/2) * (((Mee - 1) / te) + ((Mii - 1) / ti) +
                        sqrt(((((Mee - 1) / te) - ((Mii - 1)/ti))^2) +
                               ((4 * Mei * Mie) / (te * ti))))
  lambda2 <- (1/2) * (((Mee - 1) / te) + ((Mii - 1) / ti) -
                        sqrt(((((Mee - 1) / te) - ((Mii - 1)/ti))^2) +
                               ((4 * Mei * Mie) / (te * ti))))
  return(c(lambda1, lambda2))
}
te = 10
ti = 80
Mee <- 1.25
Mie <- 1
Mii <- -1
Mei <- -1

real_lambda <- function(ti) {
  return((1/2) * (((Mee - 1) / te) + ((Mii - 1) / ti)))
}

#Test real part with various ti
tau_i <- seq(60, 100, by = 0.5)
real_lambdas <- rep(0, length(tau_i))

for (i in 1:length(tau_i)) {
  real_lambdas[i] <- real_lambda(tau_i[i])
}

tau_i[which(real_lambdas == 0)]
```

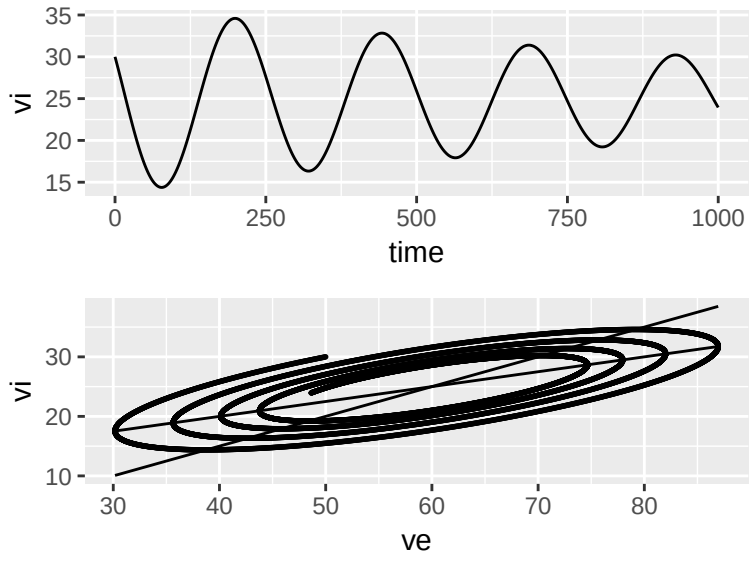
```
## [1] 80
```

```
ggplot() +
  geom_line(aes(tau_i, real_lambdas)) +
  geom_point(aes(tau_i[which(real_lambdas == 0)],
                 real_lambdas[which(real_lambdas == 0)]))
```

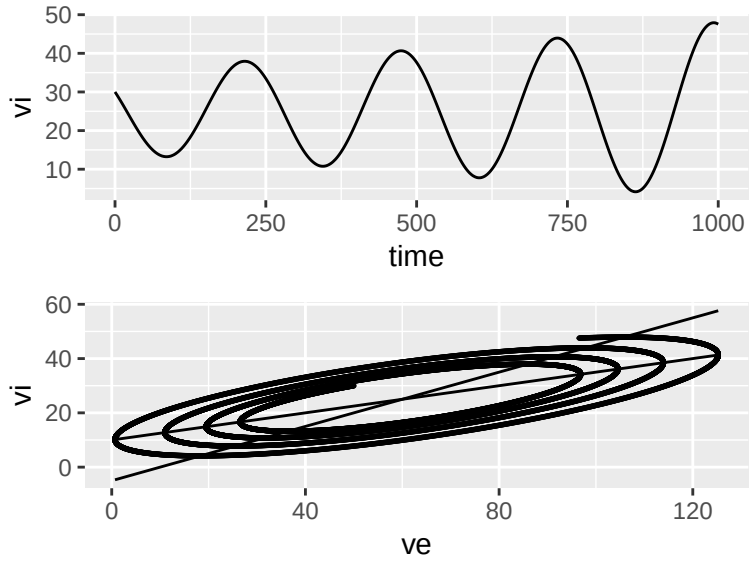


Thus we observe that a τ value of 80 gives us a real part of 0, leading us to state that the fixed point will be stable for any values of τ under 80, having the real parts of both eigenvalues being less than 0. This is also shown in the presented graphs, where we can see how 80 represents the critical value. Also, we observed what happened to our system for two values of τ around this critical value.

```
var_ti(75)
```




```
var_ti(85)
```



Hopfield network

Finally, we constructed a Hopfield network with binary units and tested its ability to recall different patterns.

```
#Hopfield Network
hopfield_weights <- function(image_patterns) {
  N <- ncol(image_patterns)
  weights <- matrix(0, nrow = N, ncol = N)
  weights <- t(image_patterns) %*% image_patterns
  for (i in 1:N) {
    for (j in 1:N) {
      if (i == j) {
        weights[i, j] <- 0
      }
    }
  }
  return(weights)
}

hopfield <- function(patterns, image_patterns, N, threshold = 10000) {
  #Generate random patterns
  X <- matrix(0, nrow = patterns, ncol = N)
  for (i in 1:patterns) {
    X[i,] <- sample(c(-1,1), N, replace = TRUE)
  }
  original_X <- X

  #Get the weights from the image_patterns
  W <- hopfield_weights(image_patterns)

  #Perform the correction of the random patterns
  count <- 0
  flag <- TRUE

  while (flag && count <= threshold) {
```

```
    #calculate the action components
    a <- X %*% W
    for (i in 1:patterns) {
      for (j in 1:N) {
        if (a[i,j] >= 0) {
          a[i,j] <- 1
          } else {a[i,j] <- -1}
        }
      }
    #Compare activation to current pattern
    difference <- ((a - X))
    if (all(difference == 0)) {
      flag <- FALSE
    } else {
      X <- a
      count <- count + 1
    }
  }
  return(list(original = original_X, new = X))
}
```

```
#Test
D <- matrix(c(-1,1,1,1,-1,-1,1,-1,-1,1,-1,1,-1,-1,1,-1,1,-1,1,-1,-1,1,-1,1,1,1,-1),
            nrow=5,ncol = 5)
J <- matrix(c(1,1,1,-1,-1,1,-1,-1,1,-1,-1,-1,-1,1,-1,-1,-1,-1,1,-1,1,1,1,1,1),
            nrow = 5,ncol = 5)
C <- matrix(c(-1,1,1,1,1,1,-1,-1,-1,-1,1,-1,-1,-1,-1,1,-1,-1,-1,-1,-1,1,1,1,1),
            nrow = 5,ncol = 5)
M <- matrix(c(1,-1,-1,-1,1,1,-1,-1,-1,1,1,-1,1,-1,1,1,1,-1,1,1,1,-1,-1,-1,1),
            nrow = 5,ncol = 5)
image_patterns <- matrix(c(C,D,J,M), nrow = 4, byrow = T)

m4 <- hopfield(4, image_patterns, 25)
m3 <- hopfield(3, image_patterns, 25)
m2 <- hopfield(2, image_patterns, 25)
m1 <- hopfield(1, image_patterns, 25)
```

According to Dayan and Abbott in *Theoretical Neuroscience*, the choice of which units are active in each pattern is random. We denote by alpha the sparseness of the memory patterns. As alpha decreases, making the patterns more sparse, more of them can be stored but each contains less information.

We were able to observe that our model could correctly store up to two patterns for the tested case. The sparseness of the pattern did affect this result with a higher sparseness meaning a lower performance. Also, a reduced number of neurons

In order to analyzed the capacity of our network, we created a function that would score the network, looking at the amount of pixels that were the same than the image pattern.

```
#Sparseness
sparseness <- function(hopfield_output) {
  difference <- hopfield_output$original - hopfield_output$new
  # print(difference)
  return(sum(abs(difference)))
}

#Test sparseness
```

```r
sparseness(m4)
```

```
## [1] 82
```

```r
sparseness(m3)
```

```
## [1] 54
```

```r
sparseness(m2)
```

```
## [1] 38
```

```r
sparseness(m1)
```

```
## [1] 26
```

When analyzing the robustness of our Hopfield Network, we looked at pattern recalls after implementing weight losses. Below are our results.

```r
#Robustness

#Random loss of weights
hopfield_weights_drop <- function(image_patterns, deletion) {
  N <- ncol(image_patterns)
  weights <- matrix(0, nrow = N, ncol = N)
  weights <- t(image_patterns) %*% image_patterns
  for (i in 1:N) {
    for (j in 1:N) {
      if (i == j) {
        weights[i, j] <- 0
      }
    }
  }
  for (d in 1:deletion) {
    i <- sample(c(1:25), 1)
    j <- sample(c(1:25), 1)
    weights[i, j] <- 0
    weights[j, i] <- 0
  }
  return(weights)
}

hopfield_drop <- function(patterns, image_patterns, N, threshold = 10000) {

  #Generate random patterns
  X <- matrix(0, nrow = patterns, ncol = N)
  for (i in 1:patterns) {
    X[i,] <- sample(c(-1,1), N, replace = TRUE)
  }
  original_X <- X

  #Get the weights from the image_patterns
  W <- hopfield_weights_drop(image_patterns, 5)

  #Perform the correction of the random patterns
  count <- 0
  flag <- TRUE
```

```
  while (flag && count <= threshold) {
    #calculate the action components
    a <- X %*% W
    for (i in 1:patterns) {
      for (j in 1:N) {
        if (a[i,j] >= 0) {
          a[i,j] <- 1
          } else {a[i,j] <- -1}
        }
      }

    #Compare activation to current pattern
    difference <- ((a - X))
    if (all(difference == 0)) {
      flag <- FALSE
    } else {
      X <- a
      count <- count + 1
    }
  }
  return(list(original = original_X, new = X))
}

#Test after random weight loss
m4_d <- hopfield_drop(4, image_patterns, 25)
m3_d <- hopfield_drop(3, image_patterns, 25)
m2_d <- hopfield_drop(2, image_patterns, 25)
m1_d <- hopfield_drop(1, image_patterns, 25)

#Test sparseness
sparseness(m4)
```

```
## [1] 82
```

```
sparseness(m3)
```

```
## [1] 54
```

```
sparseness(m2)
```

```
## [1] 38
```

```
sparseness(m1)
```

```
## [1] 26
```