Computational Neuroscience: Assignment 1

Henrik Åhl

February 28, 2017

Preface

This is an assignment report in connection to the *Computational Neuroscience* module in the Computational Biology MPhil programme at the University of Cambridge, Lent term 2017. All related code is as of February 28, 2017 available per request by contacting the author at hpa22@cam.ac.uk.

Exercises

Coupled integration of firing neurons

For this assignment we chose to implement a set of numerical solvers in order to analyse the dynamics of our system. These three consist of the following:

1. Euler method
2. Runge-Kutta method of order 4
3. Milstein method for Chemical Langevin equations, under the Stratonovich interpretation of stochastic integrals.

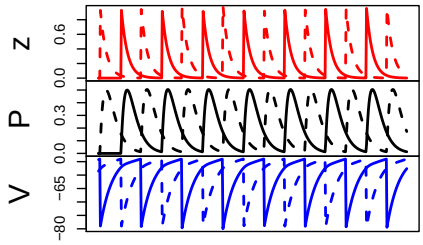
In particular in the third case, the choice of the Stratonovich interpretation over the Itō one is in favour of similarity to the deterministic countervariant under low noise. This is easily seen in the generic (derivative-free) formulation of this, i.e.

$$Y_{n+1} = Y_n + a\left(Y_n\right)\Delta t + b\left(Y_n\right)\Delta W_n + \frac{1}{2\sqrt{(\Delta t)}}\left(b\left(\bar{Y_n}\right) - b\left(Y_n\right)\right)\left(\Delta W_n\right)^2$$

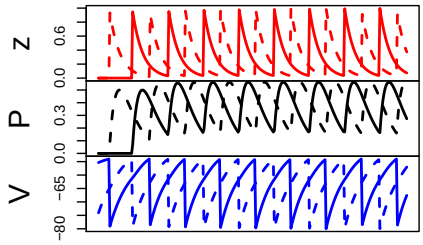
where Y is our generic variable at time n, a is the deterministic propensity function, and b the stochastic counterpart. W are randomly variables from a gaussian distribution, while $(\bar{Y})$ is the predictor step at time n. In our interpretation, we divide the noise terms with a factor σ in order to adjust the amount of noise. Indeed, as stated above, the stochastic terms vanish under $\sigma \gg 1$.

For our deterministic solvers we simply implement our Runge-Kutta 4 (RK4) in order to justify using the Euler approach for our analyses, as it is much less computationally intensive. In all cases, a stepsize of 0.01 mV is used. In the stochastic case, noise is only set to the derivative of z, as it propagates throughout the system, and as our other variables cannot be related to direct physical quantities.

In comparing the Euler method to RK4, it is found that the differences in global error ($\mathcal{O}\left(h\right)$ and $\mathcal{O}\left(h^4\right)$ respectively) does not significantly impact the firing rate and the phase shift (data not shown).



(a) Inhibitory.



(b) Excitatory.

Figure 1: Firing dynamics for a pair of coupled neurons over a range of 500 ms. Note in particular how 1) the neurons appear to lock precisely out of phase from each other, and 2) that the firing rate for the excitatory neurons is higher than for the inhibitory ones. Voltage in mV.

Figures 1a and 1b show the two cases with inhibitory and excitatory neurons respectively. If we compare these rates with the ones shown in the assignment (and after some digging which reveals that these simulations in fact use a P_{max} of 1.0), we indeed achieve the correct firing rates.

When we manipulate the τ factors, which effec-

tively sets the response time for P and z, we investigate proportions between the two constants in order to evaluate how this changes the dynamics of the system. Figure 2 shows this investigation, where the network has been simulated until converged with respect to the phase shift under a precision 10^{-5}. During these simulations, τ_m is held constant at 10 ms, whereafter τ_s takes on values of powers of two of τ_m.

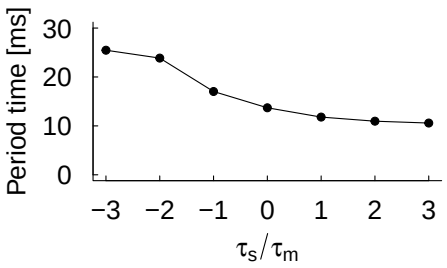
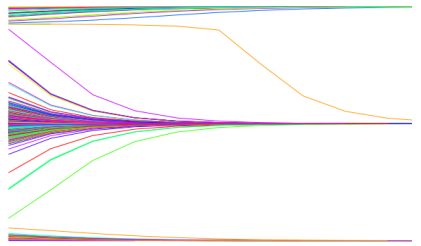
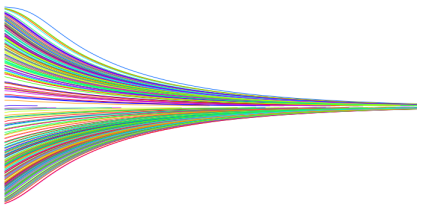


Figure 2: Dependence of firing rate, or period time, on the proportion between the two response factors. The horizontal axis is given in powers of two, i.e. labelling is according to $1 := 2^1, 2 := 2^2$ and so forth.



(a) Inhibitory.



(b) Excitatory.

Figure 3: Phase diagram for the phase shift over the course of 300 simulations. The vertical axis shows the fractional phase shift of the period time for one of the neurons, i.e. a value in the range $[0, 1]$. Note also that this axis is toroidal, such that a value of 1.0 is functionally equivalent to a value of 0. The horizontal axis shows the number of firing events. In other words, the trajectories with a less steep slope converge more slowly. Notably, the excitatory network is more prone to lock completely out of phase, whereas the inhibitory net is more resilient to smaller changes.

As figs. 3a and 3b show, the dynamics of the network highly depened on the interaction between the neurons. If the neurons are excitatory, for example, the neurons are prone to fire completely out of sync after a long enough time as passed. When the interaction instead is inhibitory, the neurons can lock together and fire in synchrony, also if the values are somewhat deviating initially. However, and as not shown in this figure, the excitatory neurons can also fire together given that the initial phase shift is not too large. The boundary for when this locking effect occurs is not easily quantified, as it depends on the time required for a firing event from the initial voltages, and not only on the relative difference between them. Still, at a value of -80 mV for one neuron, a difference of circa 3 mV appears to be the limit for locking in phase. Under excitatory circumstances, the tolerance is instead on the order of 0.1.

Changing the value of the interaction strength appears to in some cases destabilize the system. At some positive values of E_S, the system seemingly breaks and stabilizes at a phase shift fraction in neither the vicinity of 1.0 nor 0.5. In other words, there appears to a change in the reachable attractors given another configuration of parameters.

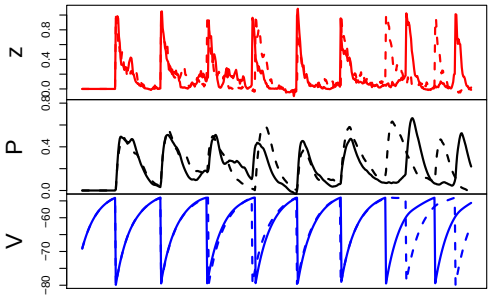


Figure 4: Stochastic simulation of the above dynamics, using a noise reducing factor of $\sigma = 3$. Note how in voltage-space the neurons lock and release due to the noise.

When we simulate the dynamics with noise included, we see the expected behaviour, namely that the neurons are able to lock into phase with each other, and subsequently break this lock due to inherent noise in the gating variable. This can be seen in the latter parts of fig. 4, where the two attractors alternatingly are favoured.

Hopfield network

For our Hopfield network, we investigate various aspects of performance, such as the ability to recall patterns and the effects of weight-loss. We represent our

nodes as vectors of length 100, where each individual node can take values $s \in \{-1, 1\}$. For training, we use two approaches, namely the Hebbian training rule, as well as Storkey training, and assess the features of these two approaches. Whenever the statistical certainty in our results is shown, it is done so as the standard error of the mean, using 10 simulations per such investigation.

We measure storage capacity by inputting the stored patterns and iterating the network until convergence, using an asynchronous and randomly updating rule for the neurons. If the recovered minimum indeed corresponds to the right pattern, we consider it a success.

As a safety-check, we note that the energy is steadily decreasing in all of our implementations, assuring us of a viable implementation. Also visual analysis of the patterns stored and retrieved support this (data not shown in either case).

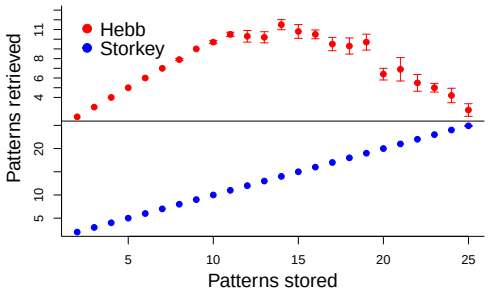


Figure 5: Storage!

Using our definition of capacity, the Hebbian net is able to quite successfully store circa 12.5 patterns in a 100 node network on average, as fig. 5 shows, as well as 13 patterns in maximum. This corresponds well to the theoretical maximum, which would be 13.8 patterns for 100 nodes. However, this assumes a completely uniform distribution of state values, which is not the case in our randomly generated patterns. In the Storkey version, we are able to retrieve far more patterns due to the locality adjustments in the algorithm. Due to limitation in time, we are however not able to retrieve a maximum number of stored patterns with our way of measuring this.

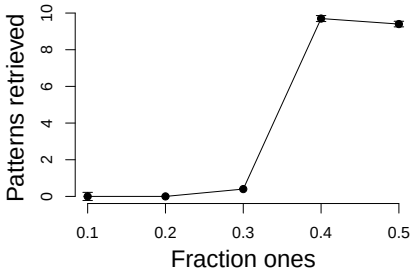


Figure 6: A network with 10 patterns stored, and denoted number of bits in the patterns set to ones. For skewed sparseness, the network drastically becomes unable to retrieve the stored patterns.

When we change the sparseness in our patterns, the ability to retrieve patterns clearly changes, which fig. 6 indicates. Some variations in the fraction bits set to either value appears to be managable for the net, but larger variations struggle significantly, with good results only within the range of 40-60 % sparseness. This ought to be due to the high overlap in the number of bits between the patterns, i.e. high correlation, which makes the network unable to converge towards the correct one.

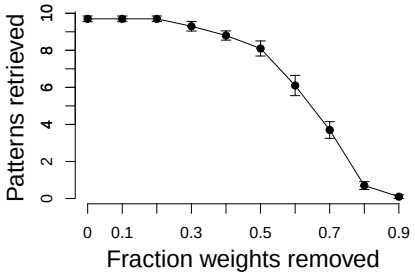


Figure 7: Network with 10 patterns stored, exposed to removal of a denoted fraction of weights. Also with a fairly significant loss of weights, the network is still able to retrieve most or all patterns.

In removing weights from the network, we see in fig. 7 that for up to 30 % weights removed, most patterns are still found. It is likely that also the Stokey net would perform at least similarly, and presumably better, due to the neighbourhood aspect of the weights, i.e. such that a neighbourhood can replace the functionality of a single weight in the Hebbian countervariant. However, due to time constraints, we are unable to show this.

Acknowledgements

Many thanks to my very good friend Julian Melgar for no particular reason. Also to Iñigo, since he asked for it.

```
#!/usr/bin/env Rscript
allow.inverse = FALSE
threshold = 0
pattern.length = 100
no.iterations = 5000
technical.replicates = 1
no.nodes = pattern.length
no.patterns.stored = seq(10, 10)
distortion.fracs = seq(0, 0, by = 0.1)
fraction.ones = seq(0.5, 0.5, by = 0.1)
weights.removed = seq(0, .9, by = 0.1)
training.method = "hebb"

################################################
# Functions we use
################################################
calc.energy = function(input) {
  energy = 0
  for (ii in 1:ncol(weights)) {
    for (jj in 1:nrow(weights)) {
      energy = energy - 0.5 * weights[ii, jj] * input[ii] * input[jj]
    }
    energy = energy + threshold * input[ii]
  }
  return(energy)
}

hebb.learn.patterns = function(patterns) {
  inp.w = matrix(0, no.nodes, no.nodes)
  for (ii in 1:ncol(inp.w))
    for (jj in 1:nrow(inp.w)) {
      inp.w[ii, jj] =
        1.0 / no.nodes * sum(patterns[ii,] * patterns[jj,])
    }
  diag(inp.w) = 0
  return(inp.w)
}

get.h = function(ii, jj, nu, prev.w, patterns) {
  h.sum = 0
  for (ll in (1:ncol(prev.w))[-c(ii, jj)])
    h.sum = h.sum + prev.w[ii, ll] * patterns[ll, nu]
  return(h.sum)
}

storkey.learn.patterns = function(patterns) {
  inp.w = matrix(0, no.nodes, no.nodes)
  prev.w = matrix(0, ncol(inp.w), nrow(inp.w))
  for (nu in 1:ncol(patterns)) {
    for (ii in 1:ncol(inp.w)) {
      for (jj in 1:nrow(inp.w)) {
        inp.w[ii, jj] = prev.w[ii, jj] + 1.0 / no.nodes * (
          patterns[ii, nu] * patterns[jj, nu] - patterns[ii, nu] * get.h(jj, ii, nu, prev.w,
              patterns) - patterns[jj, nu] *
            get.h(ii, jj, nu, prev.w, patterns)
        )
      }
    }
    prev.w = inp.w
  }
  diag(inp.w) = 0
  return(inp.w)
}

update.async.random = function(input) {
  rand = sample(1:no.nodes, 1)
  input[rand] = ifelse(sum(weights[rand, ] * input) > threshold, 1,-1)
```

```
67    return(input)
68  }
69
70  update.async.ordered = function(input, index) {
71    input[index] = ifelse(sum(weights[index, ] * input) > threshold, 1,-1)
72    return(input)
73  }
74
75  output.pattern = function(output, pattern) {
76    all(output == pattern)
77  }
78
79  ###############################################
80  # Actually run stuff
81  ###############################################
82  patterns.stored = vector("list", length(fraction.ones))
83  names(patterns.stored) = fraction.ones
84  for (ii in 1:length(patterns.stored)) {
85    patterns.stored[[ii]] = vector("list", length(weights.removed))
86    names(patterns.stored[[ii]]) = weights.removed
87    for (jj in 1:length(patterns.stored[[ii]])) {
88      patterns.stored[[ii]][[jj]] =
89        vector("list", length(distortion.fracs))
90      names(patterns.stored[[ii]][[jj]]) = distortion.fracs
91      for (kk in 1:length(patterns.stored[[ii]][[jj]])) {
92        patterns.stored[[ii]][[jj]][[kk]] =
93          vector("list", length(no.patterns.stored))
94        names(patterns.stored[[ii]][[jj]][[kk]]) = no.patterns.stored
95      }
96    }
97  }
98
99  # Set training method
100 if (tolower(training.method) == "hebb") {
101   learn.patterns = hebb.learn.patterns
102 } else if (tolower(training.method) == "storkey") {
103   learn.patterns = storkey.learn.patterns
104 }
105
106 data = list() # This we'll treat separately
107 for (iter in 1:technical.replicates) {
108   for (this.no.patterns.stored in no.patterns.stored) {
109     for (this.fraction.ones in fraction.ones) {
110       # Generate patterns
111       patterns = replicate(this.no.patterns.stored,
112                            sample(c(
113                              rep(1, pattern.length * this.fraction.ones),
114                              rep(-1, pattern.length * (1 - this.fraction.ones))
115                            ), pattern.length, replace = FALSE))
116
117       for (this.weights.removed in weights.removed) {
118         # Set the weights
119         weights = learn.patterns(patterns)
120         to.remove = sample(1:no.nodes, this.weights.removed * no.nodes, replace = FALSE)
121         weights[to.remove, to.remove] = 0
122
123         for (this.distortion.frac in distortion.fracs) {
124           patterns.found = 0 # This is what we measure.
125           # Simulate over the patterns
126           for (this.pattern in 1:ncol(patterns)) {
127             # Do some replicates for data
128             for (rep in 1:technical.replicates) {
129               output = patterns[, this.pattern]
130
131               # Distort the given fraction of bits.
132               flipbits = sample(1:pattern.length,
133                                 this.distortion.frac * pattern.length,
```

```
                                    replace = FALSE)
                if (length(flipbits) > 0)
                  output[flipbits] = -1 * output[flipbits]

                # See if convergence.
                for (ii in 1:no.iterations) {
                  #print(calc.energy(output))
                  output = update.async.random(output) #update.async.ordered(output, ii %% no.
                      nodes + 1)
                }
                # Have we converged? To the right one?
                if (any(apply(patterns, 2, function(x)
                  output.pattern(output, x))) &
                  output == patterns[, this.pattern]) {
                  patterns.found = patterns.found + 1
                }
              }
            }
            patterns.stored[[as.character(this.fraction.ones)]][[as.character(this.weights.
                removed)]][[as.character(this.distortion.frac)]][[as.character(this.no.patterns
                .stored)]] = patterns.found / technical.replicates
          }
        }
      }
    }
    data = append(data, patterns.stored)
}
```

```
#!/usr/bin/env Rscript
# Set stuff
inhibitory = F
method = "milstein"
EL = -70.0 #mV
ES = ifelse(inhibitory, -80.0, 0.0) #mV
Vth = -54.0 #mV
Vreset = -80.0 # mV
tm = 20.0 # ms
ts = 10.0 # ms
rmgs = 0.15
RmIe = 18.0 # mV
Pmax = 0.5

# Init
start.p = c(0, 0)
start.v = runif(2, min = Vreset, max = Vth)
start.z = c(0, 0)
start.t = 0.0
end.t = 500.0
h = 0.01
no.iterations = (end.t - start.t) / h
vol = c(3, 999999999, 9999999999)

# Init
p = start.p
z = start.z
t = start.t
v = start.v
p.outs = data.frame(x = rep(NA, end.t), y = rep(NA, end.t))
z.outs = data.frame(x = rep(NA, end.t), y = rep(NA, end.t))
v.outs = data.frame(x = rep(NA, end.t), y = rep(NA, end.t))
prev = -1
firing.coord = c()
phase.shift = c()
firing.rate = c()

# Define derivatives
z.derivs = function(t, z) {
```

```
40    -z / ts
41  }
42
43  p.derivs = function(t, p, z) {
44    (exp(1) * Pmax * z - p) / ts
45  }
46
47  v.derivs = function(t, v, p) {
48    (EL - v - rmgs * p * (v - ES) + RmIe) / tm
49  }
50
51  euler = function(t, v, p, z, h, vol) {
52    z <<- z + z.derivs(t, z) * h
53    p <<- p + p.derivs(t, p, z) * h
54    v <<- v + v.derivs(t, v, p) * h
55    t <<- t + h
56  }
57
58  milstein = function(t, v, p, z, h, vol) {
59    for (ii in 1:length(v)) {
60      z.pred.first = ifelse(z.derivs(t, z[ii]) > 0, sqrt(z.derivs(t, z[ii])),-sqrt(-z.derivs(t,
            z[ii])))
61      z.pred = z[ii] + z.derivs(t, z[ii]) * h + z.pred.first * sqrt(h)
62      z.first = ifelse(z.derivs(t, z[ii]) > 0, sqrt(z.derivs(t, z[ii])),-sqrt(-z.derivs(t, z[ii
            ])))
63      z.second = ifelse(z.derivs(t, z.pred) > 0,
64                        sqrt(z.derivs(t, z.pred)), -sqrt(-z.derivs(t, z.pred)))
65      z[ii] <<-
66        z[ii] + z.derivs(t, z[ii]) * h +        z.first * rnorm(1) / vol[1] * sqrt(h) + 1.0 /
              (2.0 * sqrt(h)) *
67        (z.second - z.first) * (rnorm(1) ** 2 / (vol[1] ** 2)) * h
68
69      p.pred.first = ifelse(p.derivs(t, p[ii], z[ii]) > 0,
70                            sqrt(p.derivs(t, p[ii], z[ii])), -sqrt(-p.derivs(t, p[ii], z[ii])))
71      p.pred = p[ii] + p.derivs(t, p[ii], z[ii]) * h + p.pred.first * sqrt(h)
72      p.first = ifelse(p.derivs(t, p[ii], z[ii]) > 0,
73                       sqrt(p.derivs(t, p[ii], z[ii])), -sqrt(-p.derivs(t, p[ii], z[ii])))
74      p.second = ifelse(p.derivs(t, p.pred, z[ii]) > 0,
75                        sqrt(p.derivs(t, p.pred, z[ii])),-sqrt(-p.derivs(t, p.pred, z[ii])))
76      p[ii] <<-
77        p[ii] + p.derivs(t, p[ii], z[ii]) * h + p.first * rnorm(1) / vol[2] * sqrt(h) + 1.0 /
              (2.0 * sqrt(h)) *
78        (p.second - p.first) * (rnorm(1) ** 2 / (vol[2] ** 2)) * h
79
80      v.pred.first = ifelse(v.derivs(t, v[ii], p[ii]) > 0,
81                            sqrt(v.derivs(t, v[ii], p[ii])), -sqrt(-v.derivs(t, v[ii], p[ii])))
82      v.pred = v[ii] + v.derivs(t, v[ii], p[ii]) * h + v.pred.first * sqrt(h)
83      v.first = ifelse(v.derivs(t, v[ii], p[ii]) > 0,
84                       sqrt(v.derivs(t, v[ii], p[ii])), -sqrt(-v.derivs(t, v[ii], p[ii])))
85      v.second = ifelse(v.derivs(t, v.pred, p[ii]) > 0,
86                        sqrt(v.derivs(t, v.pred, p[ii])),-sqrt(-v.derivs(t, v.pred, p[ii])))
87      v[ii] <<-
88        v[ii] + v.derivs(t, v[ii], p[ii]) * h + v.first * rnorm(1) / vol[3] * sqrt(h) + 1.0 /
              (2.0 * sqrt(h)) *
89        (v.second - v.first) * (rnorm(1) ** 2 / (vol[3] ** 2)) * h
90    }
91    t <<- t + h
92  }
93
94  rk4 = function(t, v, p, z, h, vol) {
95    z.k1 = z.derivs(t, z)
96    z.k2 = z.derivs(t + h / 2.0, z + h / 2.0 * z.k1)
97    z.k3 = z.derivs(t + h / 2.0, z + h / 2.0 * z.k2)
98    z.k4 = z.derivs(t + h, z + h * z.k3)
99    z <<- z + h / 6.0 * (z.k1 + 2 * z.k2 + 2 * z.k3 + z.k4)
100
101   p.k1 = p.derivs(t, p, z)
```

```
102   p.k2 = p.derivs(t + h / 2.0, p + h / 2.0 * p.k1, z + h / 2.0 * z.k1)
103   p.k3 = p.derivs(t + h / 2.0, p + h / 2.0 * p.k2, z + h / 2.0 * z.k2)
104   p.k4 = p.derivs(t + h, p + h * p.k3, z + h * z.k3)
105   p <<- p + h / 6.0 * (p.k1 + 2 * p.k2 + 2 * p.k3 + p.k4)
106
107   v.k1 = v.derivs(t, v, p)
108   v.k2 = v.derivs(t + h / 2.0, v + h / 2.0 * v.k1, p + h / 2.0 * p.k1)
109   v.k3 = v.derivs(t + h / 2.0, v + h / 2.0 * v.k2, p + h / 2.0 * p.k2)
110   v.k4 = v.derivs(t + h, v + h * v.k3, p + h * p.k3)
111   v <<- v + h / 6.0 * (v.k1 + 2 * v.k2 + 2 * v.k3 + v.k4)
112   t <<- t + h
113 }
114
115 # Set numerical method
116 if (tolower(method) == "milstein") {
117   fct = milstein
118 } else if (tolower(method) == "euler") {
119   fct = euler
120 } else if (tolower(method) == "rk4") {
121   fct = rk4
122 } else {
123   stop("Error: Method not found.")
124 }
125
126 # RUN!
127 for (x in 1:no.iterations) {
128   converging.limit = 10 ** -5
129   phase.shift = c(9999999, 999999999)
130   # while (abs(phase.shift[1] - phase.shift[2]) > converging.limit) {
131   fct(t, v, p, z, h, vol)
132   firing = which(v > Vth)
133   for (ii in firing) {
134     v[ii] = Vreset
135     z[ii %% length(z) + 1] = 1.0
136     firing.coord = append(firing.coord, t)
137     if (ii == 2) {
138       phase.shift = append(((firing.coord[length(firing.coord)] - firing.coord[length(firing.
              coord) - 1]) / (firing.coord[length(firing.coord)] - firing.coord[length(firing.
              coord) - 2])
139       ), phase.shift)
140       if (length(firing.coord) > 2)
141         firing.rate = append(firing.rate, firing.coord[length(firing.coord)] - firing.coord[
                length(firing.coord) -
142                                                                                              2])
143     }
144
145     if (phase.shift[1] == 0 & phase.shift[2] > .5)
146       phase.shift[1] = 1
147     if (phase.shift[1] == 1 & phase.shift[2] < .5)
148       phase.shift[1] = 0
149   }
150
151   #cat(c(t, v[1], v[2], p[1], p[2], z[1], z[2], "\n"), sep = "\t")
152   if (floor(t) > prev) {
153     prev = floor(t)
154     p.outs[t, ] = p
155     z.outs[t, ] = z
156     v.outs[t, ] = v
157   }
158 }
```