Supplementary Material for: Homogeneity and identity tests for unidimensional Poisson processes for neurophysiological peri-stimulus time histograms

Christophe Pouzat, Antoine Chaffiol, Avner Bar-Hen

Supplementary Material for: Homogeneity and identity tests for unidimensional Poisson processes for neurophysiological peri-stimulus time histograms.

Christophe Pouzat[1],
Antoine Chaffiol[2],
Avner Bar-Hen[1]
[1] MAP5, Paris-Descartes University and CNRS UMR 8145
[2] Pierre and Marie Curie University, CNRS UMR 7210 and INSERM UMR U968

February 4, 2015

1 The analysis with R

1.1 Loading the required libraries and data

The analysis requires a package available on the Comprehensive R Archive Network (CRAN): STAR. The reader should therefore start by installing it if the package is not already installed. The library is then loaded in the session:

```
library(STAR)
```

The three analyzed data sets (e060817terpi, e060817citron and e060817mix) are loaded next:

```
data(e060817terpi)
data(e060817citron)
data(e060817mix)
```

1.2 Raster plots

There is a built-in function creating raster plots in STAR (Pouzat and Chaffiol 2009, the plot method for objects of which the data just loaded are instances), but we need a finer control of the graphical output for our figures and define a mkRaster function:

```
mkRaster <- function (x, stimTimeCourse = NULL, colStim = "grey80", xlim,
                      pch, xlab, ylab, main, ...) {
    if (!is.repeatedTrain(x))
        x <- as.repeatedTrain(x)
    nbTrains <- length(x)
    if (missing(xlim))
        xlim <- c(0, ceiling(max(sapply(x, max))))
    if (missing(xlab))
        xlab <- "Time (s)"
    if (missing(ylab))
        ylab <- "trial"
    if (missing(main))
        main <- paste(deparse(substitute(x)), "raster")
    if (missing(pch))
        pch <- ifelse(nbTrains <= 20, "|", ".")
    acquisitionDuration <- max(xlim)
    plot(c(0, acquisitionDuration), c(0, nbTrains + 1), type = "n",
        xlab = xlab, ylab = ylab, xlim = xlim, ylim = c(1, nbTrains +
            1), bty = "n", main = main, axes = FALSE,...)
    if (!is.null(stimTimeCourse)) {
        rect(stimTimeCourse[1], 0.1, stimTimeCourse[2], nbTrains +
            0.9, col = colStim, lty = 0)
    }
    invisible(sapply(1:nbTrains, function(idx) points(x[[idx]],
        numeric(length(x[[idx]])) + idx, pch = pch)))
    axis(1)
}
```

We can now make Fig. 6:

```
layout(matrix(1:3,nc=3))
par(cex.axis=3,cex.lab=4,cex.main=4,mar=c(5,5,5,1))
mkRaster(e060817citron[[1]],
         stimTimeCourse=attr(e060817citron[["neuron 1"]],"stimTimeCourse"),
         xlab="Time (s)",ylab="",main="Neuron 1",xlim=c(5,10))
mkRaster(e060817citron[[2]],
         stimTimeCourse=attr(e060817citron[["neuron 2"]],"stimTimeCourse"),
         xlab="Time (s)",main="Neuron 2",ylab="",xlim=c(5,10))
mkRaster(e060817citron[[3]],
         stimTimeCourse=attr(e060817citron[["neuron 3"]],"stimTimeCourse"),
         xlab="Time (s)",main="Neuron 3",ylab="",xlim=c(5,10))
```

Fig. 7 is built with:

```
layout(matrix(1:3,nc=3))
par(cex.axis=3,cex.lab=4,cex.main=4,mar=c(5,5,5,1))
mkRaster(e060817citron[[1]],
         stimTimeCourse=attr(e060817citron[["neuron 1"]],"stimTimeCourse"),
         xlab="Time (s)",ylab="",main="Citronellal",xlim=c(5,10))
mkRaster(e060817terpi[[1]],
         stimTimeCourse=attr(e060817terpi[["neuron 1"]],"stimTimeCourse"),
         xlab="Time (s)",main="Terpineol",ylab="",xlim=c(5,10))
mkRaster(e060817mix[[1]],
         stimTimeCourse=attr(e060817mix[["neuron 1"]],"stimTimeCourse"),
         xlab="Time (s)",main="Mixture",ylab="",xlim=c(5,10))
```

1.3 Building the PSTH and stabilizing its variance

We then build a histogram of the data with a 25 ms bin width, keeping only observations in interval [1,14]; we then stabilize the variance (Eq. 3) with:

```
n1citron <- sort(as.vector(unlist(unclass(e060817citron[[1]]))))
n1citron <- n1citron[1 <= n1citron & n1citron <= 14]
n1citron_bin <- seq(1,14.025,0.025)
n1citron_hist <- hist(n1citron,n1citron_bin,plot=FALSE)
n1citron_count <- n1citron_hist$counts
n1citron_y = 2*sqrt((n1citron_count+0.25)/20)
n1citron_x = n1citron_bin[-length(n1citron_bin)]+0.0125
```

1.3.1 PSTH and variance-stabilized-PSTH figure

The `R` commands producing the equivalent of Fig. 1 are:

```
layout(matrix(1:2,nc=2))
par(mar=c(5,5,4,1))
plot(n1citron_bin[-1],n1citron_count,type='s',col='black',
     xlab="Time (s)",ylab=expression("Number of events"~(Y[i])),
     main="Original")
plot(n1citron_x,n1citron_y,type='s',col='black',
     xlab="Time (s)",ylab=expression(2*sqrt((Y[i] + 1/4)/20)),
     main="Variance stabilized",ylim=c(0,3))
```

1.4 Kernel smoothing

1.4.1 The tricube function

We start by defining a `tricube_kernel` function:

```
tricube_kernel <- function(x,bw=1.0) {
    ax <- abs(x/bw)
    result <- numeric(length(x))
    result[ax <= 1] <- 70*(1-ax[ax <= 1]^3)^3/81
    result }
```

1.4.2 The Nadaraya-Watson estimator

We define next a function returning the Nadaraya-Watson estimator at a given point:

```
Nadaraya_Watson_Estimator <- function(x,X,Y,
                                      kernel = function(y) tricube_kernel(y,1.0)) {
    ## Returns the Nadaray-Watson estimator at x, given data X and Y
    ## using kernel.
    ##
    ## Parameters
    ## ----------
    ## x: point at which the estimator is looked for.
    ## X: abscissa of the observations.
    ## Y: ordinates of the observations.
    ## kernel: a univariate 'weight' function.
    ##
    ## Returns
    ## -------
    ## The estimated ordinate at x.
    w <- kernel(X-x)
    sum(w*Y)/sum(w) }
```

1.4.3 Mallow's Cp score computation

We now need a function returning Mallow's C_p score and define a function, `Cp_score`, doing the job:

```
Cp_score <- function(X,Y,bw = 1.0,
                     kernel = tricube_kernel,
                     sigma2=1/20) {
    ## Computes Mallow's Cp score given data X and Y, a bandwidth bw,
    ## a bivariate function kernel and a variance sigma2.
    ##
    ## Parameters
    ## ----------
    ## X: abscissa of the observations.
    ## Y: ordinates of the observations.
    ## bw: the bandwidth.
    ## kernel: a bivariate function taking an ordinate as first parameter
    ##         and a bandwidth as second parameter.
    ## sigma2: the variance of the ordinates.
    ##
    ## Returns
    ## -------
    ## A tuple with the trace of the smoother and the Cp score.

    L <- matrix(0,nrow=length(X),ncol=length(X))
    ligne <- numeric(length(X))
    for (i in 1:length(X)) {
        ligne <- kernel(X-X[i], bw)
        L[i,] <- ligne/sum(ligne) }
    n <- length(X)
    trace <- sum(diag(L))
    if (trace == n) {
        return(NULL)
    } else {
        Cp = (sum((Y- Y%*%L)^2) + 2*sigma2*trace)/n
        c(trace, Cp) }}
```

We can get the score over a range of bandwidths (from 50 ms to 1 s) with:

```
bw_vector <- seq(0.05,1,0.025)
n1citron_Cp_score <- sapply(bw_vector,
                            function(bw)
                            Cp_score(n1citron_x,n1citron_y,bw))
```

We then extract the bandwidth giving the best (lowest) score and get the corresponding Nadaraya-Watson estimator:

```
bw_best_Cp <- bw_vector[which.min(n1citron_Cp_score[2,])]
n1citron_y_NW_best <- sapply(n1citron_x,
                             function(x)
                             Nadaraya_Watson_Estimator(x,
                                                       n1citron_x,
                                                       n1citron_y,
                                                       kernel = function(y)
                                                       tricube_kernel(y,
                                                                      bw_best_Cp)))
```

1.4.4 Figure with Cp score vs bandwidth and smooth estimator

The equivalent of Fig. 2 in R is built with:

```
layout(matrix(1:2,nc=2))
par(mar=c(5,5,4,1))
plot(bw_vector,n1citron_Cp_score[2,],type="l",col='red',lwd=2,
     xlab='Bandwidth (s)',ylab='Cp Scores',
     main='Score vs bandwidth')
plot(n1citron_x,n1citron_y,type="s",col='black',
     xlab="Time (s)",
     ylab=expression(2*sqrt((Y[i] + 1/4)/20)),
     main="Data and Nadaraya-Watson est.")
lines(n1citron_x,n1citron_y_NW_best,col=2,lwd=2)
```

1.5 Confidence set for the smoother

1.5.1 κ_0

We get the approximate value $\kappa_0 \approx (b-a)/h \left(\int_a^b K'(t)^2 dt\right)^{1/2}$ by computing analytically the integral with the open source computer algebra system (CAS) `maxima` (`http://maxima.sourceforge.net/`):

```
print(float(13*(sqrt(integrate(diff(70*(1-x^3)^3/81,x)^2,x,0,1)*2))/0.225));
```

```
86.58938919551133
```

1.5.2 Getting the constant c of our tube formula

We define next a function, `tube_target` returning the "target", that is:

$$2\,(1-\Phi(c)) + \frac{\kappa_0}{\pi}\exp -\frac{c^2}{2} - \alpha\,,$$

```
tube_target <- function(x,alpha,kappa=86.58938919551133)
    (2*(1-pnorm(x)) + kappa*exp(-x^2/2)/pi - alpha)^2
```

We then get the c values for two α, 0.95 and 0.9 with:

```
c_p95 <- optimize(tube_target,c(3,4),alpha=0.05)$minimum
c_p90 <- optimize(tube_target,c(2,4),alpha=0.1)$minimum
```

1.5.3 Smoothing matrix

We define a function returning the smoothing matrix L—a matrix whose $(L)_{i,j}$ element is given by $l_i(t_j)$, where the $l_i()$ are defined in the text and the t_j are the centers of our PSTH bins—, evaluate the matrix for the data at hand and get the value of $\|l(t)\|$ at each abscissa value:

```
make_L <- function(X,kernel = function(y) tricube_kernel(y,1.0)) {
    result <- matrix(0,nr=length(X),nc=length(X))
    ligne <- numeric(length(X))
    for (i in 1:length(X)) {
        ligne <- kernel(X-X[i])
        result[i,] = ligne/sum(ligne) }
    result }

n1citron_NW_L_best <- make_L(n1citron_x,
                             kernel = function(y)
                             tricube_kernel(y,bw_best_Cp))
n1citron_NW_L_best_norm <- sqrt(apply(n1citron_NW_L_best^2,1,sum))
```

1.5.4 Figure of the smooth estimate with the 0.95 confidence set

The equivalent of Fig. 3 in R is simply obtained with:

```
par(mar=c(5,5,4,1))
plot(n1citron_x,n1citron_y,type="l",col='black',
     xlab="Time (s)",
     ylab=expression(2*sqrt((Y[i] + 1/4)/20)),
     main="Nadaraya-Watson est. with 0.95 conf. bands",
     ylim=c(0,3))
lines(n1citron_x,n1citron_y_NW_best,lwd=2,col='blue')
lines(n1citron_x,
      n1citron_y_NW_best+c_p95*n1citron_NW_L_best_norm/sqrt(20),
      lwd=2,col='red')
lines(n1citron_x,
      n1citron_y_NW_best-c_p95*n1citron_NW_L_best_norm/sqrt(20),
      lwd=2,col='red')
```

1.6 Confidence set for the citronellal response of Neuron 2

We build the PSTH (using a 10 ms bin width since neuron 2 exhibits a higher basal firing rate than neuron 1) and stabilize its variance:

```
n2citron <- sort(as.vector(unlist(unclass(e060817citron[[2]]))))
n2citron <- n2citron[1 <= n2citron & n2citron <= 14]
n2citron_bin <- seq(1,14.01,0.01)
n2citron_hist <- hist(n2citron,n2citron_bin,plot=FALSE)
n2citron_count <- n2citron_hist$counts
n2citron_y = 2*sqrt((n2citron_count+0.25)/20)
n2citron_x = n2citron_bin[-length(n2citron_bin)]+0.005
```

We set the bandwidth at 1 s, and get the new κ_0 value (since the bandwidth changed):

```
print(float(13*(sqrt(integrate(diff(70*(1-x^3)^3/81,x)^2,x,0,1)*2))/1));
```

```
19.48261256899005
```

We then compute the Nadaraya-Watson estimator, the smoothing matrix, the norm of its rows and get and the new c value:

```
n2_citron_bw <- 1
n2citron_y_NW <- sapply(n2citron_x,
                        function(x)
                        Nadaraya_Watson_Estimator(x,
                                                  n2citron_x,
                                                  n2citron_y,
                                                  kernel = function(y)
                                                  tricube_kernel(y,
                                                                 n2_citron_bw)))
n2citron_NW_L <- make_L(n2citron_x,
                        kernel = function(y)
                             tricube_kernel(y,n2_citron_bw))
n2citron_NW_L_norm <- sqrt(apply(n2citron_NW_L^2,1,sum))
c_p95b <- optimize(tube_target,c(3,4),alpha=0.05,kappa=19.48261256899005)$minimum
```

Fig. 4 is built in R with:

```
par(mar=c(5,5,4,1))
plot(n2citron_x,n2citron_y,type="l",col='black',
     xlab="Time (s)",
     ylab=expression(2*sqrt((Y[i] + 1/4)/20)),
     main="1.0 s bandwidth")
lines(n2citron_x,
      n2citron_y_NW+c_p95b*n2citron_NW_L_norm/sqrt(20),
      lwd=2,col='red')
lines(n2citron_x,
      n2citron_y_NW-c_p95b*n2citron_NW_L_norm/sqrt(20),
      lwd=2,col='red')
```

1.7 Testing identity

1.7.1 Boundary crossing probability

The required functions are included in our `STAR` package, they are named: `crossGeneral` and `crossTight`. They return the distribution of the first passage time of a canonical Brownian motion through a "general boundary" (`crossGeneral`) and through a "square root boundary" as considered in this manuscript (`crossTight`). They are fully documented in the package. Tests against the results of Loader and Deely 1987 are included in the example section of the functions' documentation.

Parameters of the "square root boundary" Following the example of `crossTight` documentation we get the parameters a and b of a "square root boundary" $a + b\sqrt{t}$ giving a 95% coverage probability with:

```
target95 <- mkTightBMtargetFct(ci=0.95)
p95 <- optim(log(c(0.3,2.35)),target95,method="BFGS")
p95$convergence
exp(p95$par)
d95 <- crossTight(a=exp(p95$par[1]),b=exp(p95$par[2]),withBound=TRUE,logScale=FALSE)
summary(d95)
```

```
[1] 0
[1] 0.2999446 2.3479702
Prob. of first passage before 1: 0.025 (bounds: [0.02497,0.02503])
Integration time step used: 0.001.
```

A systematic estimation of the parameters a and b of the square root boundary for coverage probabilities going from 0.9 to 0.99 is carried out as follows (rounding to the third digit):

```
p_vector <- seq(0.1,0.01,-0.01)
get_a_b <- function(p) {
    h_size <- 0.001
    target <- mkTightBMtargetFct(ci=1-p,h=h_size)
    fit <- optim(log(c(0.3,2.35)),
                 target,method="BFGS")
    dom <- crossTight(a=exp(fit$par[1]),
                      b=exp(fit$par[2]),
                      withBound=TRUE,
                      logScale=FALSE)
    within <- dom$Gl[length(dom$Gl)] <= p/2 &
        p/2 <= dom$Gu[length(dom$Gu)]
    while (fit$convergence != 0 || !within) {
        if (fit$convergence != 0) {
            fit <- optim(fit$par,
                         target,
                         method="BFGS")
        } else {
            h_size <- h_size/10
            target <- mkTightBMtargetFct(ci=1-p,h=h_size)
            fit <- optim(fit$par,
                         target,method="BFGS") }
        dom <- crossTight(a=exp(fit$par[1]),
                          b=exp(fit$par[2]),
                          withBound=TRUE,
                          logScale=FALSE)
        within <- dom$Gl[length(dom$Gl)] <= p/2 &
            p/2 <= dom$Gu[length(dom$Gu)] }
    res <- exp(fit$par)
    c(a=res[1],b=res[2])}

sqrt_coef <- t(rbind(p_vector,
                     sapply(p_vector,
                            get_a_b)))
(sqrt_coef <- round(sqrt_coef,digits=3))
```

```
      p_vector     a     b
 [1,]     0.10 0.292 2.077
 [2,]     0.09 0.293 2.120
 [3,]     0.08 0.295 2.167
 [4,]     0.07 0.296 2.220
 [5,]     0.06 0.298 2.279
 [6,]     0.05 0.300 2.348
 [7,]     0.04 0.302 2.430
 [8,]     0.03 0.305 2.531
 [9,]     0.02 0.308 2.668
[10,]     0.01 0.313 2.890
```

Back to the analysis of the data set We build the terpineol PSTH of neuron 1 (we compensate for different onset times, 6.03 s for terpineol and 5.99 for citronellal) and stabilize its variance:

```
n1terpi <- sort(as.vector(unlist(unclass(e060817terpi[[1]]))))-0.04
n1terpi <- n1terpi[1 <= n1terpi & n1terpi <= 14]
n1terpi_bin <- seq(1,14.025,0.025)
n1terpi_hist <- hist(n1terpi,n1terpi_bin,plot=FALSE)
n1terpi_count <- n1terpi_hist$counts
n1terpi_y = 2*sqrt((n1terpi_count+0.25)/20)
n1terpi_x = n1terpi_bin[-length(n1terpi_bin)]+0.0125
```

We do the same for the responses to even and odd numbers stimuli and we build the boundary functions:

```
n1terpiOdd <- sort(as.vector(unlist(unclass(e060817terpi[[1]][(1:10)*2-1]))))-0.04
n1terpiOdd <- n1terpiOdd[1 <= n1terpiOdd & n1terpiOdd <= 14]
n1terpiOdd_bin <- seq(1,14.025,0.025)
n1terpiOdd_hist <- hist(n1terpiOdd,n1terpiOdd_bin,plot=FALSE)
n1terpiOdd_count <- n1terpiOdd_hist$counts
n1terpiOdd_y = 2*sqrt((n1terpiOdd_count+0.25)/10)
n1terpiOdd_x = n1terpiOdd_bin[-length(n1terpiOdd_bin)]+0.0125

n1terpiEven <- sort(as.vector(unlist(unclass(e060817terpi[[1]][(1:10)*2]))))-0.04
n1terpiEven <- n1terpiEven[1 <= n1terpiEven & n1terpiEven <= 14]
n1terpiEven_bin <- seq(1,14.025,0.025)
n1terpiEven_hist <- hist(n1terpiEven,n1terpiEven_bin,plot=FALSE)
n1terpiEven_count <- n1terpiEven_hist$counts
n1terpiEven_y = 2*sqrt((n1terpiEven_count+0.25)/10)
n1terpiEven_x = n1terpiEven_bin[-length(n1terpiEven_bin)]+0.0125

c95 <- function(x)
    sqrt_coef[6,2]+sqrt_coef[6,3]*sqrt(x)

c99 <- function(x)
    sqrt_coef[10,2]+sqrt_coef[10,3]*sqrt(x)
```

The equivalent of Fig. 5 is then obtained in R with:

```
xx <- seq(0,1,len=201)
plot(xx,c95(xx),type="l",col='red',lwd=2,lty='dashed',
     ylim=c(-4,6),xlab="Normalized time",
     ylab=expression(S[k](t)))
lines(xx,-c95(xx),col='red',lwd=2,lty='dashed')
lines(xx,c99(xx),col='red',lwd=2)
lines(xx,-c99(xx),col='red',lwd=2)
lines((n1citron_x-1)/(max(n1citron_x)-1),
      cumsum(sqrt(5)*(n1terpiOdd_y-n1terpiEven_y))/sqrt(length(n1terpi_y)),
      col='blue',lwd=2)
lines((n1citron_x-1)/(max(n1citron_x)-1),
      cumsum(sqrt(10)*(n1terpi_y-n1citron_y))/sqrt(length(n1terpi_y)),
      col='black',lwd=2)
```

1.8 Simulation study

We want to estimate the coverage probability of our "Brownian domains" as a function of the sample size. We are going to use a Monte Carlo simulation to do that for each of our nine sets of square root boundary coefficients. To that end we define first a function carrying out the simulations for a given

sample size:

```
inside_domain <- function(sample_size,
                          n_rep=100000,
                          coeff_list=sqrt_coef) {
    ## Computes a 95% confidence interval for the 'coverage
    ## probability' of each square-root boundary defined in the list
    ## coeff_list for a given sample size using n_rep Monte Carlo
    ## replicates.
    ##
    ## Parameters
    ## ----------
    ## sample_size: an integer, the sample size.
    ## n_rep: an integer, the number of MC replicates.
    ## coeff_list: a matrix. Each row should contain the
    ##             coefficient a and b in its second and third elements,
    ##             the boundary being defined by: a + b*sqrt(t).
    ##
    ## Returns
    ## -------
    ## A matrix, each row contains the extremes of an
    ## Agresti-Coull 95% CI as defined by Brown et al (2001) Statistical
    ## Science 16:101-117. There is one row for each row of
    ## coeff_list.
    st_v <- sqrt(seq(1,(sample_size))/sample_size)
    b_matrix <- apply(coeff_list,1, function(coeff) coeff[2]+coeff[3]*st_v)
    total_v <- numeric(dim(coeff_list)[1])
    for (i in 1:n_rep) {
        sim <- cumsum(rnorm(sample_size))/sqrt(sample_size)
        within <- apply(b_matrix,2,
                        function(B) all(-B <= sim & sim <= B))
        total_v <- total_v + within }
    proba <- sapply(total_v, function(T) (T+2)/(n_rep+4))
    t(sapply(proba,
             function(p)
             c(p - 2*sqrt(p*(1-p)/(n_rep+4)),
               p + 2*sqrt(p*(1-p)/(n_rep+4)))))
}
```

We then use this function to get the empirical coverage probabilities in a range of sample sizes:

```
set.seed(20110928)
samp_size_v <- c(25,50,75,100,250,500,750,1000,2500,5000,7500,10000)
empirical_CP <- sapply(samp_size_v,
                       function(n) t(inside_domain(n)))
```

The results obtained with `R` can be compared with the ones reported in Table 2 obtained with `Python`:

	25	50	75	100	250	500	750	1000	2500	5000	7500	10000
0.99 up	0.995	0.994	0.994	0.994	0.993	0.992	0.992	0.992	0.992	0.991	0.991	0.992
0.99 low	0.993	0.992	0.992	0.991	0.99	0.99	0.99	0.99	0.99	0.989	0.989	0.989
0.98 up	0.989	0.988	0.986	0.986	0.984	0.983	0.983	0.983	0.983	0.982	0.981	0.982
0.98 low	0.986	0.985	0.984	0.984	0.981	0.981	0.98	0.981	0.98	0.979	0.979	0.98
0.97 up	0.982	0.981	0.978	0.978	0.975	0.974	0.974	0.974	0.973	0.973	0.971	0.972
0.97 low	0.98	0.978	0.976	0.975	0.972	0.971	0.971	0.971	0.97	0.969	0.968	0.969
0.96 up	0.976	0.974	0.971	0.971	0.967	0.965	0.965	0.965	0.964	0.963	0.962	0.963
0.96 low	0.973	0.971	0.968	0.968	0.963	0.962	0.962	0.962	0.96	0.96	0.958	0.959
0.95 up	0.97	0.967	0.964	0.963	0.958	0.956	0.956	0.956	0.955	0.954	0.952	0.953
0.95 low	0.966	0.964	0.96	0.959	0.955	0.953	0.953	0.952	0.951	0.95	0.948	0.95
0.94 up	0.963	0.96	0.956	0.955	0.95	0.947	0.947	0.946	0.944	0.944	0.942	0.943
0.94 low	0.96	0.956	0.952	0.951	0.947	0.943	0.943	0.942	0.94	0.94	0.938	0.939
0.93 up	0.956	0.953	0.948	0.947	0.942	0.938	0.938	0.937	0.935	0.935	0.933	0.934
0.93 low	0.953	0.949	0.944	0.943	0.938	0.934	0.934	0.932	0.93	0.931	0.929	0.929
0.92 up	0.95	0.945	0.941	0.939	0.934	0.929	0.929	0.927	0.925	0.925	0.924	0.924
0.92 low	0.946	0.941	0.936	0.935	0.929	0.925	0.924	0.923	0.921	0.921	0.919	0.92
0.91 up	0.943	0.938	0.933	0.931	0.924	0.92	0.919	0.917	0.915	0.916	0.914	0.915
0.91 low	0.939	0.934	0.928	0.926	0.92	0.916	0.915	0.913	0.91	0.911	0.909	0.91
0.90 up	0.936	0.931	0.924	0.923	0.916	0.911	0.91	0.909	0.906	0.906	0.904	0.905
0.90 low	0.932	0.926	0.92	0.918	0.911	0.907	0.905	0.904	0.901	0.901	0.899	0.9

2 The analysis with `Python`

2.1 Setting up `Python`

The analysis presented in the manuscript and detailed next is carried out with `Python 3` (the following code runs and gives identical results with `Python 2`). We are going to use the 3 classical modules of Python's scientific ecosystem: `numpy`, `scipy` and `matplotlib`. We are also going to use a fourth module of this ecosystem: `sympy` as well as the `h5py` module. We start by importing these modules:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
import sympy as sy
import h5py
```

2.2 Getting the data

Our data (Pouzat and Chaffiol 2015) are stored in HDF5 format on the zenodo server (`DOI:10.5281/zenodo.1428145`). They are all contained in a file named `CockroachDataJNM_2009_181_119.h5`. The data within this file have an hierarchical organization similar to the one of a file system (one of the main ideas of the HDF5 format). The first organization level is the exper-

iment; there are 4 experiments in the file: `e060517`, `e060817`, `e060824` and `e070528`. Each experiment is organized by neurons, `Neuron1`, `Neuron2`, etc, (with a number of recorded neurons depending on the experiment). Each neuron contains a `dataset` (in the HDF5 terminology) named `spont` containing the spike train of that neuron recorded during a period of spontaneous activity. Each neuron also contains one or several further sub-levels named after the odor used for stimulation `citronellal`, `terpineol`, `mixture`, etc. Each a these sub-levels contains as many `datasets`: `stim1`, `stim2`, etc, as stimulations were applied; and each of these data sets contains the spike train of that neuron for the corresponding stimulation. Another `dataset`, named `stimOnset` containing the onset time of the stimulus (for each of the stimulations). All these times are measured in seconds.

2.3 Loading neuron 1 citronellal responses

We get the responses to the 20 stimulations with citronellal of neuron 1 in experiment e060817 with (assuming that the data file `CockroachDataJNM_2009_181_119.h5` has been downloaded into the current working directory of the Python session):

```
f = h5py.File("CockroachDataJNM_2009_181_119.h5","r")
citron_onset = f["e060817/Neuron1/citronellal/stimOnset"][...][0]
train_list = [f[y][...] for y in
              ["e060817/Neuron1/citronellal/stim"+str(x)
               for x in range(1,21)]]
n1citron = np.sort(np.concatenate(train_list))
f.close()
```

2.4 Building the PSTH and stabilizing its variance

We then build a histogram of the data with a 25 ms bin width, keeping only observations in interval [1,14], with:

```
n1citron = n1citron[np.logical_and(1 <= n1citron , n1citron <= 14)]
n1citron_bin = np.arange(1,14.025,0.025)
n1citron_count,n1citron_bin = np.histogram(n1citron,n1citron_bin)
n1citron_y = 2*np.sqrt((n1citron_count+0.25)/20)
n1citron_x = n1citron_bin[:-1]+0.0125
```

2.4.1 PSTH and variance-stabilized-PSTH figure

Fig. 1 is produced by the following commands:

```
fig = plt.figure()
plt.subplot(121)
plt.plot(n1citron_bin[1:],n1citron_count,ls='steps',color='black')
plt.xlabel("Time (s)")
plt.ylabel("Number of events ($Y_i$)")
plt.title("Original")
plt.subplot(122)
plt.plot(n1citron_x,n1citron_y,ls='steps',color='black')
plt.xlabel("Time (s)")
plt.ylabel("$2 \sqrt{(Y_i + 1/4)/20}$")
plt.title("Variance stabilized")
plt.subplots_adjust(wspace=0.4)
plt.savefig('figs/make-n1citron-histos-figure.png')
plt.close()
```

2.5 Kernel smoothing

2.5.1 The tricube function

We start by defining a `tricube_kernel` function:

```
def tricube_kernel(x,bw=1.0):
    ax = np.absolute(x/bw)
    result = np.zeros(x.shape)
    result[ax <= 1] = 70*(1-ax[ax <= 1]**3)**3/81.
    return result
```

2.5.2 The Nadaraya-Watson estimator

We define next a function returning the Nadaraya-Watson estimator at a given point:

```
def Nadaraya_Watson_Estimator(x,X,Y,
                              kernel = lambda y:
                              tricube_kernel(y,1.0)):
    """Returns the Nadaray-Watson estimator at x, given data X and Y
    using kernel.

    Parameters
    ----------
    x: point at which the estimator is looked for.
    X: abscissa of the observations.
    Y: ordinates of the observations.
    kernel: a univariate 'weight' function.

    Returns
    -------
    The estimated ordinate at x.
    """
    w = kernel(X-x)
    return np.sum(w*Y)/np.sum(w)
```

2.5.3 Mallow's Cp score computation

We now need a function returning Mallow's C_p score and define a function, `Cp_score`, doing the job:

```
def Cp_score(X,Y,bw = 1.0, kernel = tricube_kernel,sigma2=1/20.):
    """Computes Mallow's Cp score given data X and Y, a bandwidth bw,
    a bivariate function kernel and a variance sigma2.

    Parameters
    ----------
    X: abscissa of the observations.
    Y: ordinates of the observations.
    bw: the bandwidth.
    kernel: a bivariate function taking an ordinate as first parameter
            and a bandwidth as second parameter.
    sigma2: the variance of the ordinates.

    Returns
    -------
    A tuple with the trace of the smoother and the Cp score.
    """
    from numpy.matlib import identity
    L = np.zeros((len(X),len(X)))
    ligne = np.zeros(len(X))
    for i in range(len(X)):
        ligne = kernel(X-X[i], bw)
        L[i,:] = ligne/np.sum(ligne)
    n = len(X)
    trace = np.trace(L)
    if trace == n: return None
    Cp = np.dot(np.dot(Y,(identity(n)-L)),
                np.dot((identity(n)-L),Y).T)[0,0]/n + 2*sigma2*trace/n
    return [trace, Cp]
```

We can get the score over a range of bandwidths (from 50 ms to 1 s) with:

```
bw_vector = np.arange(0.05,1,0.025)
n1citron_Cp_score = np.array([Cp_score(n1citron_x,n1citron_y,bw)
                              for bw in bw_vector])
```

We then extract the bandwidth giving the best (lowest) score and get the corresponding Nadaraya-Watson estimator:

```
bw_best_Cp = bw_vector[np.argmin(n1citron_Cp_score[:,1])]
n1citron_y_NW_best = np.array([Nadaraya_Watson_Estimator(x,n1citron_x,
                                                         n1citron_y,
                                                         kernel = lambda y:
                                                         tricube_kernel(y,
                                                                        bw_best_Cp))
                                                         for x in n1citron_x])
```

2.5.4 Figure with Cp score vs bandwidth and smooth estimator

Fig. 2 is built with:

```
fig = plt.figure(figsize=(10,5))
plt.subplot(121)
plt.plot(bw_vector,n1citron_Cp_score[:,1],color='red',lw=2)
plt.xlabel('Bandwidth (s)')
plt.ylabel('Cp Scores')
plt.title('Score vs bandwidth')
plt.subplot(122)
plt.plot(n1citron_x,n1citron_y,ls='steps',color='black')
plt.plot(n1citron_x,n1citron_y_NW_best,lw=2,color='red')
plt.xlabel("Time (s)")
plt.ylabel("$2 \sqrt{(Y_i + 1/4)/20}$")
plt.title("Data and Nadaraya-Watson est.")
plt.subplots_adjust(wspace=0.4)
plt.savefig('figs/n1citron-Nadaraya-Watson-estimator.png')
plt.close()
```

2.6 Confidence set for the smoother

2.6.1 κ_0

We get the approximate value $\kappa_0 \approx (b-a)/h \left(\int_a^b K'(t)^2 dt\right)^{1/2}$ by computing analytically the integral with `sympy`:

```
sx = sy.symbols('sx')
K = 70*(1-sx**3)**3/81 ## symbolic version of the tricube kernel
## Integration is carried out next, remember that the data cover
## a 13 s range, explaining the pre factor.
kappa0 = 13*(sy.sqrt(sy.integrate(sy.diff(K,sx)**2,
                                  (sx,0,1))*2)).evalf()/bw_best_Cp
```

2.6.2 Getting the constant c of our tube formula

We define next a function, `tube_target` returning the "target", that is:

$$2\,(1-\Phi(c)) + \frac{\kappa_0}{\pi} \exp -\frac{c^2}{2} - \alpha\,,$$

```
def tube_target(x,alpha,kappa=kappa0):
    from scipy.stats import norm
    return 2*(1-norm.cdf(x)) + kappa*np.exp(-x**2/2)/np.pi - alpha
```

We then get the c values for two α, 0.95 and 0.9 with:

```
from scipy.optimize import brentq
c_p95 = brentq(tube_target,a=3,b=4,args=(0.05,))
c_p90 = brentq(tube_target,a=2,b=4,args=(0.1,))
```

2.6.3 Smoothing matrix

We define a function returning the smoothing matrix L—a matrix whose $(L)_{i,j}$ element is given by $l_i(t_j)$, where the $l_i()$ are defined in the text and

the t_j are the centers of our PSTH bins—, evaluate the matrix for the data at hand and get the value of $\|l(t)\|$ at each abscissa value:

```
def make_L(X,kernel = lambda y: tricube_kernel(y,1.0)):
    result = np.zeros((len(X),len(X)))
    ligne = np.zeros(len(X))
    for i in range(len(X)):
        ligne = kernel(X-X[i])
        result[i,:] = ligne/np.sum(ligne)
    return result

n1citron_NW_L_best = make_L(n1citron_x,
                            kernel = lambda y:
                            tricube_kernel(y,bw_best_Cp))
n1citron_NW_L_best_norm = np.sqrt(np.sum(n1citron_NW_L_best**2,axis=1))
```

2.6.4 Figure of the smooth estimate with the 0.95 confidence set

Fig. 3 is simply obtained with:

```
plt.figure()
plt.plot(n1citron_x,n1citron_y,color='black')
plt.plot(n1citron_x,n1citron_y_NW_best,lw=2,color='blue')
plt.plot(n1citron_x,
         n1citron_y_NW_best+c_p95*n1citron_NW_L_best_norm/np.sqrt(20),
         lw=2,color='red')
plt.plot(n1citron_x,
         n1citron_y_NW_best-c_p95*n1citron_NW_L_best_norm/np.sqrt(20),
         lw=2,color='red')
plt.xlabel("Time (s)")
plt.ylabel("$2 \sqrt{(Y_i + 1/4)/20}$")
plt.title("Nadaraya-Watson est. with 0.95 conf. bands")
plt.savefig('figs/n1citron-Nadaraya-Watson-Confidence-Bands.png')
plt.close()
```

2.7 Confidence set for the citronellal response of Neuron 2

We load the data with:

```
f = h5py.File("CockroachDataJNM_2009_181_119.h5","r")
train_list = [f[y][...] for y in
              ["e060817/Neuron2/citronellal/stim"+str(x)
               for x in range(1,21)]]
n2citron = np.sort(np.concatenate(train_list))
f.close()
```

We build the PSTH (using a 10 ms bin width since neuron 2 exhibits a higher basal firing rate than neuron 1) and stabilize its variance:

```
n2citron = n2citron[np.logical_and(1 <= n2citron , n2citron <= 14)]
n2citron_bin = np.arange(1,14.01,0.01)
n2citron_count,n2citron_bin = np.histogram(n2citron,n2citron_bin)
n2citron_y = 2*np.sqrt((n2citron_count+0.25)/20)
n2citron_x = n2citron_bin[:-1]+0.005
```

We set the bandwidth at 1 s, compute the Nadaraya-Watson estimator, the

smoothing matrix, the norm of its rows. We get the new κ_0 value (since the bandwidth changed) and the new c value:

```
n2_citron_bw = 1
n2citron_y_NW = np.array([Nadaraya_Watson_Estimator(x,
                                                    n2citron_x,
                                                    n2citron_y,
                                                    kernel = lambda y:
                                                    tricube_kernel(y,n2_citron_bw))
                                                    for x in n2citron_x])
n2citron_NW_L = make_L(n2citron_x,kernel = lambda y:
                       tricube_kernel(y,n2_citron_bw))
n2citron_NW_L_norm = np.sqrt(np.sum(n2citron_NW_L**2,axis=1))
kappa0b = 13*(sy.sqrt(sy.integrate(sy.diff(K,sx)**2,
                                   (sx,0,1))*2)).evalf()/n2_citron_bw
c_p95b = brentq(tube_target,a=3,b=4,args=(0.05,kappa0b))
```

Fig. 4 is then built with:

```
plt.figure()
plt.plot(n2citron_x,n2citron_y,color='black')
plt.plot(n2citron_x,
         n2citron_y_NW+c_p95b*n2citron_NW_L_norm/np.sqrt(20),
         lw=2,color='red')
plt.plot(n2citron_x,
         n2citron_y_NW-c_p95b*n2citron_NW_L_norm/np.sqrt(20),
         lw=2,color='red')
plt.xlabel("Time (s)")
plt.ylabel("$2 \sqrt{(Y_i + 1/4)/20}$")
plt.title("1.0 s bandwidth")
plt.savefig('figs/n2citron-figure.png')
plt.close()
```

2.8 Testing identity

2.8.1 Boundary crossing probability

Background We are going to need the probability for a canonical Brownian motion to cross a boundary whose equation is $a+b\sqrt{t}$ between time 0 and time 1. To this end we use the results of Loader and Deely 1987 that can be summarized as follows, writing $G(t)$ the CDF of the first passage time, $g(t)$ the corresponding density and $c(t)$ a *continuous* boundary. We can choose a function $b(t)$, then G is solution of the following Volterra integral equation:

$$F(t) = \int_0^t K(t,u)dG(u)\,,$$

where

$$F(t) = \Phi\left(-\frac{c(t)}{\sqrt{t}}\right) + \exp\left(-2b(t)\left(c(t)-tb(t)\right)\right)\Phi\left(\frac{-c(t)+2tb(t)}{\sqrt{t}}\right)$$

and

$$\begin{aligned} K(t,u) &= \Phi\left(-\tfrac{c(u)-c(t)}{\sqrt{t-u}}\right) + \\ &\quad \exp\left(-2b(t)\left(c(t)-c(u)-(t-u)b(t)\right)\right)\Phi\left(\tfrac{c(u)-c(t)+2(t-u)b(t)}{\sqrt{t-u}}\right). \end{aligned}$$

We now take $0 = t_0 < t_1 < \cdots < t_n = t$ with $t_j = jh$ for some $h > 0$ and we set $t_{j-1/2} = (t_j + t_{j-1})/2$ a discretized version of our Volterra equation is then given by the *mid-point method*:

$$F(t_j) = \sum_{i=1}^{j} K(t_j, t_{i-1/2})\Delta_i \quad j = 1, \ldots, n\,,$$

where $\Delta_i = G(t_i) - G(t_{i-1})$ and since this linear system is lower triangular we get:

$$\Delta_j = \left(F(t_j) - \sum_{i=1}^{j-1} K(t_j, t_{i-1/2})\Delta_i\right)/K(t_j, t_{j-1/2}) \quad j = 1, \ldots, n\,.$$

Assuming that $c'(t)$ exists for all $t > 0$ and setting $L(t,u) = \partial K(t,u)/\partial u$,

$$\begin{aligned} G_L(t_1) &= F(t_1) \\ G_L(t_n) &= F(t_n) + \textstyle\sum_{j=1}^{n-1} G_L(t_j)\left[K(t_n, t_{j+1} - K(t_n, t_j)\right] \quad n = 2, \ldots \end{aligned}$$

and

$$\begin{aligned} G_U(t_1) &= F(t_1)/K(t_1, t_0) \\ G_U(t_n) &= \left\{F(t_n) + \textstyle\sum_{j=1}^{n-1} G_U(t_j)\left[K(t_n, t_j - K(t_n, t_{j-1})\right]\right\}/K(t_n, t_{n-1}) \quad n = 2, \ldots \end{aligned}$$

Loader and Deely 1987 show that if $L(t,u) \geq 0$ for $u < t$ then

$$G_L(t_n) \leq G(t_n) \leq G_U(t_n) \quad n = 1, 2, \ldots$$

`Python` code We present next a direct implementation of this algorithm in `Python`. Since the function `G_at_1_with_bounds` is a bit long, we defining it using the literate programming paradigm. We start with the `docstring` (user documentation of the function):

```
"""Probabilty for a canonical Brownian motion to cross a boundary
defined by the continous function c_fct beween 0 and 1.

Parameters
----------
c_fct: a continuous function of a single variable defining the
       boundary.
b_fct: an accessory function helping the convergence, the
       derivative of c_fct is a good default choice.
bounds: a Boolean variable, if True (default) lower and upper
        bounds for the probability are returned.

Returns
-------
The probability if bounds is False or a tuple with the lower bound
the probability and the upper bound.

Details
-------
Bounds calculation uses Eq. 3.6 and 3.7 p 102 of Loader and Deely
(1987) J Statist Comput Simul 27: 95-105, and some conditions on
the partial derivative of the Kernel appearing in the Volterra
integral equation are supposed to be met."""
```

In the actual `G_at_1_with_bounds` definition below, `«G_at_1_with_bounds-docstring»` should be replaced by the code above. We then define a univariate function `F` corresponding the function F above. This function needs to have access to the `norm` class of `scipy.stats` and to have access to two functions `c_fct` and `b_fct` corresponding respectively to c and b:

```
def F(t):
    c_t = c_fct(t)
    b_t = b_fct(t)
    term1 = norm.cdf(-c_t/np.sqrt(t))
    factorA =  np.exp(-2*b_t*(c_t-t*b_t))
    factorB = norm.cdf((-c_t+2*t*b_t)/np.sqrt(t))
    return term1 + factorA*factorB
```

In the actual `G_at_1_with_bounds` definition below, `«F-definition»` is meant to be replaced by the above code. We define next a bivariate function `K` corresponding to K above and requiring the same functions `c_fct` and `b_fct` as `F`. This function implicitely assumes that $c(u) - c(t)$ falls to 0 faster than $\sqrt{t-u}$ when $t > 0$ and $u \to t$:

```
def K(t,u):
    if t == u:
        return 1.0
    c_t = c_fct(t)
    c_u = c_fct(u)
    b_t = b_fct(t)
    term1 = norm.cdf((c_u-c_t)/np.sqrt(t-u))
    factorA =  np.exp(-2*b_t*(c_t-c_u-(t-u)*b_t))
    factorB = norm.cdf((c_u-c_t+2*(t-u)*b_t)/np.sqrt(t-u))
    return term1 + factorA*factorB
```

We now define the user function `G_at_1_with_bounds`:

```
def G_at_1_with_bounds(c_fct,b_fct,n,bounds=True):
    <<G_at_1_with_bounds-docstring>>
    from scipy.stats import norm
    <<F-definition>>
    <<K-definition>>
    t_v = np.linspace(0,1,n+1)
    t_v_half = (t_v[1:]+t_v[:-1])*0.5
    Delta = np.zeros((n))
    if bounds:
        G_L = np.zeros((n))
        G_U = np.zeros((n))
        G_L[0] = F(t_v[1])
        G_U[0] = F(t_v[1])/K(t_v[1],t_v[0])
    Delta[0] = F(t_v[1])/K(t_v[1],t_v_half[0])
    for j in range(1,n):
        term1 = F(t_v[j+1])
        factor1 = Delta[:j]
        factor2 = [K(t_v[j+1],t) for t in t_v_half[:j]]
        term2 = np.sum(factor1*np.array(factor2))
        divisor = K(t_v[j+1],t_v_half[j])
        Delta[j] = (term1-term2)/divisor
        if bounds:
            factor2 = np.diff(np.array([K(t_v[j+1],t)
                                        for t in t_v[:(j+2)]]))
            G_L[j] = term1 + np.sum(G_L[:j]*factor2[1:])
            G_U[j] = (term1 +
                      np.sum(G_U[:j]*factor2[:-1]))/K(t_v[j+1],t_v[j])
    if bounds:
        return (G_L[n-1],np.sum(Delta),G_U[n-1])
    else:
        return np.sum(Delta)
```

Test against Loader and Deely reported results We can check our code against the results reported in Table II p 104 of Loader and Deely 1987, starting with the first "column" of the upper part of the table:

```
LD87tableIIaa = [G_at_1_with_bounds(lambda x: np.sqrt(1+x),
                                    lambda x: 0.5/np.sqrt(1+x),n)
                                    for n in [8,16,32,64,128]]

[[str(round(x[0],5)),str(round(x[2],5))] for x in LD87tableIIaa]
```

```
[['0.19524', '0.1969'],
 ['0.1956', '0.19643'],
 ['0.1958', '0.19621'],
 ['0.1959', '0.1961'],
 ['0.19595', '0.19605']]
```

We stop here in the reproduction of the table, but the reader can easily check for himself that the we reproduce the whole table with our code.

Parameters of the "square root boundary" We are going to consider a simple boundary leading to an almost minimal surface (Kendall, Marin, and Robert 2007), that is a "square root boundary" $a+b\sqrt{t}$. Kendall, Marin, and Robert 2007 report in their Table 1 that $a = 0.3$ and $b = 2.35$ give a 0.95 "coverage probability". Partitioning [0,1] in 256 equal parts we get:

```
G_at_1_with_bounds(lambda x: 0.3+2.35*np.sqrt(x),lambda x: 0.5*2.35/np.sqrt(x),256)
```

```
(0.024756138795870526, 0.024863677999752844, 0.024975076286891391)
```

We can refine these values by defining first a function returning a target function (to optimize later) with:

```
def mk_boundary_target(alpha=0.05,n=128):
    target = 0.5*alpha
    def b_target(log_x):
        a = np.exp(log_x[0])
        b = np.exp(log_x[1])
        return (target -
                G_at_1_with_bounds(lambda y: a+b*np.sqrt(y),
                                   lambda y: 0.5*b/np.sqrt(y),n,
                                   False))**2
    return b_target
```

We use our "target making" function:

```
b95target = mk_boundary_target(alpha=0.05,n=128)
```

And we refine our parameters:

```
from scipy.optimize import minimize
b95 = minimize(b95target,[np.log(0.3),np.log(2.35)],
               method='BFGS',options={'disp': True})
```

```
... Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 2
         Function evaluations: 16
         Gradient evaluations: 4
```

The first coefficient is:

```
a_95,b_95 = (np.exp(x) for x in b95.x)
a_95
```

```
0.29995665705124541
```

The second coefficient is:

```
b_95
```

```
2.3484037518980978
```

We check the bounds:

```
G_at_1_with_bounds(lambda x: a_95+b_95*np.sqrt(x),
                   lambda x: 0.5*b_95/np.sqrt(x),512)
```

```
(0.02491617879464314, 0.024970600466047332, 0.025025989749594357)
```

We made a systematic estimation of the parameters a and b of the square root boundary for coverage probabilities going from 0.9 to 0.99. To that end we defined a "square root boundary tailored version" of `G_at_1_with_bounds` that makes a much better use of the vectorization allowed (en encouraged) by `Python`. We do not give the code in this document but it is fully disclosed in its source file.

We end up with the following coefficient table where the first row contains the probability of exit, the second the coefficient a and the third the coeeficient b:

```
sqrt_coef
```

```
[[0.1, 0.29180955432863043, 2.0771977869954412],
 [0.09, 0.29323505286797247, 2.1203442183163022],
 [0.08, 0.29473127117408465, 2.1674353022357664],
 [0.07, 0.29633188549204681, 2.2200098585866801],
 [0.06, 0.29805778404512068, 2.2794451106566656],
 [0.05, 0.29995772183498814, 2.34844328179922],
 [0.04, 0.30212398911444788, 2.4293475497024737],
 [0.03, 0.30467964750693033, 2.5312658394604974],
 [0.02, 0.30784648015962873, 2.668232689515055],
 [0.01, 0.3124559676910898, 2.8906058429411168]]
```

Back to the analysis of the data set We load the data as usual and we are careful in aligning the stimuli onset times:

```
f = h5py.File("CockroachDataJNM_2009_181_119.h5","r")
terpi_onset = f["e060817/Neuron1/terpineol/stimOnset"][...][0]
train_list = [f[y][...]-terpi_onset+citron_onset for y in
              ["e060817/Neuron1/terpineol/stim"+str(x)
               for x in range(1,21)]]
n1terpi = np.sort(np.concatenate(train_list))
f.close()
```

We build the PSTH and stabilize its variance:

```
n1terpi = n1terpi[np.logical_and(1 <= n1terpi , n1terpi <= 14)]
n1terpi_bin = np.arange(1,14.025,0.025)
n1terpi_count,n1terpi_bin = np.histogram(n1terpi,n1terpi_bin)
n1terpi_y = 2*np.sqrt((n1terpi_count+0.25)/20)
n1terpi_x = n1terpi_bin[:-1]+0.0125
```

We get the responses to odd and even stimulations (don't forget that Python starts indexing arrays at 0) separately:

```
n1terpiOdd = np.sort(np.concatenate([train_list[i]
                                     for i in range(0,20,2)]))
n1terpiEven = np.sort(np.concatenate([train_list[i]
                                      for i in range(1,20,2)]))
```

We build the PSTH, stabilize the variances and define the boundary functions:

```
n1terpiOdd = n1terpiOdd[np.logical_and(1 <= n1terpiOdd,
                                       n1terpiOdd <= 14)]
n1terpiOdd_bin = np.arange(1,14.025,0.025)
n1terpiOdd_count,n1terpiOdd_bin = np.histogram(n1terpiOdd,
                                               n1terpiOdd_bin)
n1terpiOdd_y = 2*np.sqrt((n1terpiOdd_count+0.25)/10)
n1terpiOdd_x = n1terpiOdd_bin[:-1]+0.0125

n1terpiEven = n1terpiEven[np.logical_and(1 <= n1terpiEven,
                                         n1terpiEven <= 14)]
n1terpiEven_bin = np.arange(1,14.025,0.025)
n1terpiEven_count,n1terpiEven_bin = np.histogram(n1terpiEven,
                                                 n1terpiEven_bin)
n1terpiEven_y = 2*np.sqrt((n1terpiEven_count+0.25)/10)
n1terpiEven_x = n1terpiEven_bin[:-1]+0.0125

def c95(x): return sqrt_coef[5][1]+sqrt_coef[5][2]*np.sqrt(x)

def c99(x): return sqrt_coef[9][1]+sqrt_coef[9][2]*np.sqrt(x)
```

We can now make Fig. 5 with:

```
xx = np.linspace(0,1,201)
plt.figure()
plt.plot(xx,c95(xx),color='red',lw=2,linestyle='dashed')
plt.plot(xx,-c95(xx),color='red',lw=2,linestyle='dashed')
plt.plot(xx,c99(xx),color='red',lw=2)
plt.plot(xx,-c99(xx),color='red',lw=2)
plt.plot((n1citron_x-1)/(np.max(n1citron_x)-1),
         np.cumsum(np.sqrt(5)*(n1terpiOdd_y-n1terpiEven_y))/\
         np.sqrt(len(n1terpi_y)),color='blue',lw=2)
plt.plot((n1citron_x-1)/(np.max(n1citron_x)-1),
         np.cumsum(np.sqrt(10)*(n1terpi_y-n1citron_y))/\
         np.sqrt(len(n1terpi_y)),color='black',lw=2)
plt.xlabel("Normalized time")
plt.ylabel("$S_k(t)$")
plt.savefig('figs/n1-citron-terpi-comp.png')
plt.close()
```

2.9 Simulation study

We want to estimate the coverage probability of our "Brownian domains" as a function of the sample size. We are going to use a Monte Carlo simulation to do that for each of our nine sets of square root boundary coefficients. To that end we define first a function carrying out the simulations at a given sample size:

```
def inside_domain(sample_size,
                  n_rep=100000,
                  coeff_list=sqrt_coef):
    """Computes a 95% confidence interval for the 'coverage
    probability' of each square-root boundary defined in the list
    coeff_list for a given sample size using n_rep Monte Carlo
    replicates.

    Parameters
    ----------
    sample_size: an integer, the sample size.
    n_rep: an integer, the number of MC replicates.
    coeff_list: a list of lists. Each sub list should contain the
                coefficient a and b in its second and third elements,
                the boundary being defined by: a + b*sqrt(t).

    Returns
    -------
    A list of tuple, each subtuple contains the extremes of an
    Agresti-Coull 95% CI as defined by Brown et al (2001) Statistical
    Science 16:101-117. There is on list element for each element of
    coeff_list."""
    from scipy.stats import norm
    t_v = np.arange(1,(sample_size+1))/float(sample_size)
    b_list = [coeff[1]+coeff[2]*np.sqrt(t_v) for coeff in coeff_list]
    total_v = np.zeros((len(coeff_list)))
    for i in range(n_rep):
        s = np.cumsum(norm.rvs(size=sample_size))/np.sqrt(sample_size)
        inside = [np.all(s.__le__(B)) and np.all(s.__ge__(-B))
                  for B in b_list]
        total_v += np.array(inside,dtype=int)
    proba = [(T+2)/(n_rep+4.) for T in total_v]
    res = [(p - 2*np.sqrt(p*(1-p)/(n_rep+4.)),
            p + 2*np.sqrt(p*(1-p)/(n_rep+4.)))
            for p in proba]
    return res
```

We then use this function to get the empirical coverage probabilities in a range of sample sizes:

```
np.random.seed(20110928)
samp_size_list = [25,50,75,100,250,500,750,1000,2500,5000,7500,10000]
empirical_CP = [inside_domain(samp_size)
                for samp_size in samp_size_list]
```

We get the results shown on Table 2 (rounding upward the third decimal for the upper bound and downward for the lower bound).

References

Kendall, W.S., J.M. Marin, and C.P. Robert (2007). "Brownian Confidence Bands on Monte Carlo Output". In: *Statistics and Computing* 17.1, pp. 1–10.

Loader, C. R. and J. J. Deely (1987). "Computations of boundary crossing probabilities for the Wiener process". In: *Journal of Statistical Computation and Simulation* 27.2, pp. 95–105.

Pouzat, Christophe and Antoine Chaffiol (2009). "Automatic Spike Train Analysis and Report Generation. An Implementation with R, R2HTML and STAR". In: *J Neurosci Methods* 181, pp. 119–144.

— (2015). *Data set from Pouzat and Chaffiol (2009) Journal of Neuroscience Methods 181:119.* DOI:10.5281/zenodo.14281.