Supplementary Material for: Homogeneity and identity tests for unidimensional Poisson processes with an application to neurophysiological peri-stimulus time histograms–R version

Christophe Pouzat, Antoine Chaffiol, Avner Bar-Hen

Supplementary Material for: Homogeneity and identity tests for unidimensional Poisson processes with an application to neurophysiological peri-stimulus time histograms–R version.

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

November 4, 2015

Contents

1 Introduction

The following document presents the analysis with R. The exposition follows roughly the software development approach used in this project. Namely, a single PSTH is analyzed first step by step, requiring the definitions of short functions or the use of a few command lines. Once this prototypical analysis is achieved, one class and its associated methods are defined. The code of the methods being the same (modulo some variable name changes) as the code of the functions previously defined. For clarity of the code presentation—as well as to keep the code length able to fit within a single page—the literate programming paradigm is used throughout this document, implying that the construction of the actual working code often implies sticking together several pieces. Therefore many listings, like Listing 1, will appear like:

```
Some code lines in R or Python
<<a-reference>>
Some more code lines
```

In such cases a "reference" made of a string between "«" and "»" (in the case above "a-reference") refers to a listing whose content should be copied and pasted in place of the reference.

Figures, tables and equations numbers given in this document refer to figures, tables and equations in the companion manuscript.

1.1 Existing tests

Cox and P. A. W. Lewis (1966) present tests for homogeneous Poisson (Sec. 6.3) and renewal (Sec. 6.4) processes. The tests for Poisson processes use the fact that if the observed times: $\{t_1, t_2, \ldots, t_n\}$ are a realization of a homogeneous Poisson process with rate λ on the time interval $[0, t_0]$, then,

conditionally on n, the total number of events observed at the end of the time period, the quantities: $\{u_{(i)} = t_i/t_0\}_{i=1,\ldots,n}$ are observations of the order statistics of n IID draws from a uniform distribution on $(0,1)$. It is then possible to apply a Kolmogorov test or an Anderson-Darling test against this null hypothesis giving a *uniform conditional test for a Poisson process*. Durbin (1961, p. 48) followed by Peter A. W. Lewis (1965) argue further for the use of what Cox and P. A. W. Lewis (1966, p. 154-155) dubbed *Durbin's transformation* of the t_i in order to improve the power of these tests against the uniform null hypothesis. The algorithm producing this transformation follows:

1. Go from the $\{u_{(i)} = t_i/t_0\}_{i=1,\ldots,n}$ discussed in the previous paragraph to the intervals: $c_1 = u_{(1)}$, $c_i = u_{(i)} - u_{(i-1)}$ $(i = 2,\ldots,n)$, $c_{n+1} = 1 - u_{(n)}\}$ (the latter should IID realizations from an exponential distribution with parameter 1).

2. Get the order statistics $\{c_{(1)},\ldots,c_{(n)}\}$ and form the differences $g_i = (n+2-i)\left(c_{(i)} - c_{(i-1)}\right)$ for $i = 1,\ldots,n+1$ with $c_{(0)} = 0$ (they should be independent exponentially distributed random variables with means 1).

3. The observations $u'_{(i)} = \sum_{j=1}^{i} g_j$ for $i = 1,\ldots,n$ should then be observations from the order statistics of n IID draws from a uniform distribution on $(0,1)$.

As pointed out by Cox and P. A. W. Lewis (1966, p. 158) the tests on transformed data are sensitive to discretization: they fail to apply if the latter is too coarse. The data used here where sampled at 12800 Hz with a spike sorting procedure that did not properly cope with sampling jitter (Pouzat and Detorakis 2014). This unaccounted for sampling jitter amounts to a "too coarse" sampling and give rise to a pronounced stair-case aspect of the empirical cumulative distribution function (ECDF) of the $u'_{(i)}$ for small values of i. This leads to spurious positive values when applying the Anderson-Darling test. We therefore decided when working with the transformed data to jitter the original observed times uniformly by plus or minus half a sampling period (in practice plus or minus 40 μs). This destroys the stair-case aspect without touching the overall structure.

In addition to these tests against a uniform distribution on $(0,1)$, the correlation coefficients of the successive inter-event intervals at different lags (the autocorrelation function of the inter-events intervals) is inspected and the log of the survivors function—that should be a straight line under the null hypothesis—is plotted.

1.2 A remark on the pseudo-random number generators used by `R` and `Python`

As most readers know, when a (pseudo) random number is drawn from a continuous distribution (exponential, normal, etc) a function of one or several random numbers with a uniform distribution on [0,1) is used: exponential random numbers are typically generated with the inversion method–this is done in both `R` with `rexp` and in the `numpy.random` module of `Python` with `exponential`–; normal random numbers are generated with the inversion method–used by default in function `rnorm` of `R`–or with the Box-Muller method–used by function `normal` in `numpy.random`–or with the Kinderman and Monahan method, etc. This implies that a crucial role is played by the generator of uniform random numbers on [0,1)–or (0,1) as is the case for `R`–. In principle, when one reads the documentation of the *default* uniform pseudo-random number generators (`PRNG`) implemented in both `R` and `Python`, one gets the impression they are the same since both software used the Mersenne Twister. This PRNG generates in fact discrete number in $\{0, 1, \ldots, 2^{32} - 1\}$ with a period of $2^{19937} - 1$. This feat is achieved by using a tuple with 624 elements, each element being an unsigned integer coded on 32 bit. This means that such a tuple has to be provided in order to initialize the generator. `R` and `Python` do this initialization differently and in order to figure out precisely how they do it, the source codes have to be inspected. It is then possible (but tedious) to use the same tuple in both languages. Then one realizes that the generated sequences of *floating point numbers* (uniform on the unit interval) are different! Inspection of the source codes provides again the explanation: at each call, the Mersenne Twister outputs an unsigned integer coded on 32 bit; `R` divides this number by 2^{32} to get a floating point number $\in [0, 1)$ –then `R` checks if the number is 0 (or negative) and in such a case it returns $1/2 \times 1/(2^{32} - 1)$ –; `Python` draws *two successive* numbers from the Mersenne-Twister and constructs an "intermediate" 53 bit unsigned integer with them–the leftmost 27 bit of first 32 bit unsigned integer provide the leftmost 27 bit of the intermediate number while the leftmost 26 bit of the second 32 bit unsigned integer provide the rightmost 26 bit of the intermediate number; the intermediate number is then divided by 2^{53} to yield a floating point number $\in [0, 1)$ (with the maximal achievable resolution with double precision). `R` generates therefore double precision floating point random numbers with a 32 bit resolution, while `Python` generates numbers with a 53 bit resolution. This (undocumented) difference does not create significant differences in the two versions of our code but it explains why we could not work with the exact same sequences in both versions.

2 The analysis with R

2.1 Implementation of existing tests

We define a function returning the Kolmogorov two sided or one sided statistics against the null hypothesis—uniform distribution on $(0, 1)$:

```
Kolmogorov_D <- function(Up,what=c("D","D+","D-")) {
    stopifnot(all(Up > 0 & Up < 1))
    what = what[1]
    stopifnot(what %in% c("D","D+","D-"))
    n <- length(Up)
    ecdf <- (1:n)/n
    Up = sort(Up)
    Dp <- max(ecdf-Up)*sqrt(n)
    Dm <- max(Up[-n]-ecdf[-n]+1/n)*sqrt(n)
    if (what == "D") return(max(Dp,Dm))
    if (what == "D+") return(Dp)
    if (what == "D-") return(Dm)
}
```

We define next a function returning the Anderson-Darling statistics against the same null hypothesis:

```
AndersonDarling_W2 <- function(Up) {
    stopifnot(all(Up > 0 & Up < 1))
    n <- length(Up)
    -n-sum((2*(1:n)-1)*log(Up)+(2*n-2*(1:n)+1)*log(1-Up))/n
}
```

There are few published tables of the cumulative distribution function of the Anderson-Darling statistics (either for finite sample size or in the asymptotic limit) and there is no R function returning it. The G. Marsaglia and J. Marsaglia (2004, page 3) algorithm returning this function with sixth decimal place (or more) precision is therefore implemented next:

```
pAD_W2 <- function(x) {
  ## Marsaglia and Marsaglia (2004) JSS 9(2):1--5
  res <- numeric(length(x))
  res[x<=0] <- 0
  small <- 0 < x & x < 2
  x_s <- x[small]
  res[small] <- 1/sqrt(x_s)*exp(-1.2337141/x_s)
  res[small] <- res[small]*(2.00012+
                            (.247105-
                             (.0649821-
                              (.0347962-
                               (.011672-.00168691*
                                x_s)*x_s)*x_s)*x_s)*x_s)
  big <- x >= 2
  x_b <- x[big]
  res[big] <- 1.0776-(2.30695-(.43424-
                               (.082433-
                                (.008056-.0003146*
                                 x_b)*x_b)*x_b)*x_b)*x_b
  res[big] <- exp(-exp(res[big]))
  res
}
```

We can test this implementation using the 0.9, 0.95 and 0.99 quantiles given by G. Marsaglia and J. Marsaglia (2004, page 2):

```
c("90%"=pAD_W2(1.9329578327),
  "95%"=pAD_W2(2.492367),
  "99%"=pAD_W2(3.878125))
```

```
      90%       95%       99%
0.8999889 0.9500081 0.9899974
```

The function performing Durbin's transformation is defined next. It takes a series of observed times and an observation interval as arguments:

```
DurbinTransform <- function(observed_times,
                            observation_interval) {
    if (missing(observation_interval))
        observation_interval <- c(floor(min(observed_times)),
                                  ceiling(max(observed_times)))
    stopifnot(all((observation_interval[1] < observed_times) &
                  (observed_times < observation_interval[2])))
    observed_times <- observed_times-observation_interval[1]
    obs_duration <- diff(observation_interval)
    n <- length(observed_times)
    observed_times <- observed_times/obs_duration
    iei <- c(observed_times[1],
             sort(diff(observed_times)),
             1-observed_times[n])
    siei <- c(0,sort(iei))
    g <- (n+2-(1:(n+1)))*diff(siei)
    cumsum(g[1:n])
}
```

2.1.1 An exploration of time discretization and jittering effects on these statistics

As discussed in the first section, the time discretization due to sampling at acquisition time and sub-optimal spike sorting algorithm has consequences on the statistics used to test if an observed (aggregated process) is homogeneous Poisson or not. These consequences are explored here with simulations mimicking the pre-stimulation period of neuron 2 from data set `e070528citronellal` whose analysis is presented in the sequel. This neurons fires 1455 during 6 seconds (and 15 trials) giving an aggregated rate of 242.5 Hz. We perform next a simulation of 10000 homogeneous Poisson processes with the latter rate during 6 s. The two sided Kolmogorov statistics–$D_n\sqrt{n}$ whose 0.95 and 0.99 quantiles are 1.358 and 1.628 respectively–as well as the Anderson-Darling one–W_n^2 whose 0.95 and 0.99 quantiles are 2.492 and 3.878 respectively, the correct value of the latter quantile is from G. Marsaglia and J. Marsaglia (2004, page 2)–are computed on the resulting conditionally uniform process ($\{t_1/6, \ldots, t_n/6\}$) as well as on its discretized version (with a time resolution corresponding to the actual sampling period of our data sets, 1/12800 s) and on a time jittered version of the discretized version (with a uniform jitter between -1/2 and +1/2 the sampling period). The same is done on the data after Durbin's transformation. A function is defined first doing the discretization:

```
discretize_time <- function(observed_times,
                            observation_period=c(0,6),
                            sampling_period=1/12800) {
  dt <- seq(observation_period[1],
            observation_period[2],
            sampling_period)
  (0.5+(findInterval(observed_times,dt)-1))*sampling_period
}
```

A function doing the time jittering is defined next (taking care of the events sitting close to the observation interval boundaries):

```
jitter_time <- function(observed_times,
                        observation_period=c(0,6),
                        sampling_period=1/12800) {
  n <- length(observed_times)
  res <- numeric(n)
  within <- observation_period[1]+sampling_period/2 < observed_times &
  observed_times < observation_period[2]-sampling_period/2
  res[within] <- observed_times[within]+
    (runif(sum(within))-0.5)*sampling_period
  too_small <- observation_period[1]+sampling_period/2 >= observed_times
  if (sum(too_small) > 0)
    res[too_small] <- runif(sum(too_small),
                            observation_period[1],
                            observed_times[too_small]+sampling_period/2)
  too_big <- observed_times >= observation_period[2]-sampling_period/2
  if (sum(too_big)>0)
    res[too_big] <- runif(sum(too_big),
                          observed_times[too_big]-sampling_period/2,
                          observation_period[2])
  sort(res)
}
```

We can now do the simulation with a single realization as follows:

```
set.seed(20061001)
hp1 <- cumsum(rexp(2000,242.5))
hp1 <- hp1[hp1<6]
hp1_d <- discretize_time(hp1)
hp1_dj <- jitter_time(hp1_d)
```

The Kolmogorov and Anderson-Darling statistics are:

```
D_o <- Kolmogorov_D(hp1/6)
D_d <- Kolmogorov_D(hp1_d/6)
D_dj <- Kolmogorov_D(hp1_dj/6)
W2_o <- AndersonDarling_W2(hp1/6)
W2_d <- AndersonDarling_W2(hp1_d/6)
W2_dj <- AndersonDarling_W2(hp1_dj/6)
matrix(c(D_o,D_d,D_dj,W2_o,W2_d,W2_dj),
       nr=2,nc=3,byrow=TRUE,
       dimnames=list(c("D","W2"),
         c("original","discretized","jittered")))
```

```
    original discretized  jittered
D  0.7290281   0.7290172 0.7288731
W2 0.8037918   0.8037792 0.8037821
```

There is no "huge" effect of time discretization here. The same is done after Durbin's transformation. Since intervals of length 0 can be obtained with the discretized data, we set these zero length intervals to five times the smallest floating point number the machine can represent:

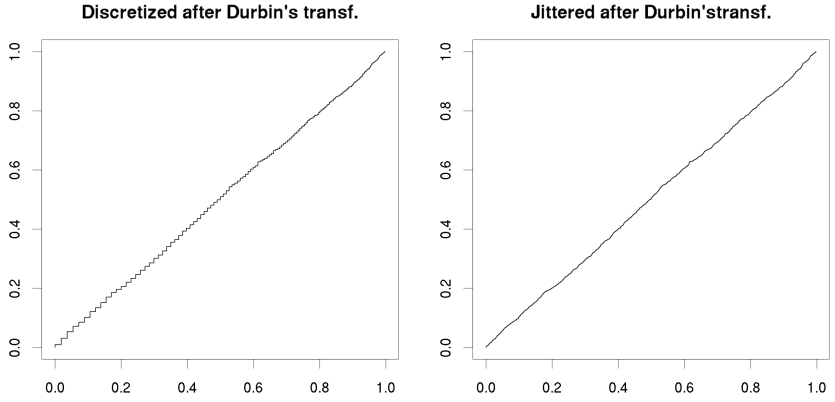
```
hp1_dt <- DurbinTransform(hp1,c(0,6))
hp1_d_dt <- DurbinTransform(hp1_d,c(0,6))
if (any(hp1_d_dt==0)) hp1_d_dt[hp1_d_dt==0] = 5*.Machine$double.eps
hp1_dj_dt <- DurbinTransform(hp1_dj,c(0,6))
```

The Kolmogorov and Anderson-Darling statistics are then:

```
D_o_dt <- Kolmogorov_D(hp1_dt)
D_d_dt <- Kolmogorov_D(hp1_d_dt)
D_dj_dt <- Kolmogorov_D(hp1_dj_dt)
W2_o_dt <- AndersonDarling_W2(hp1_dt)
W2_d_dt <- AndersonDarling_W2(hp1_d_dt)
W2_dj_dt <- AndersonDarling_W2(hp1_dj_dt)
matrix(c(D_o_dt,D_d_dt,D_dj_dt,W2_o_dt,W2_d_dt,W2_dj_dt),
       nr=2,nc=3,byrow=TRUE,
       dimnames=list(c("D","W2"),
         c("original","discretized","jittered")))
```

```
    original discretized  jittered
D  0.5476887   0.6301718 0.5520699
W2 0.4310982   3.8983496 0.4013730
```

There is a large effect of discretization on Anderson-Darling's statistics, effect that seems to be canceled by adding a jitter. Making a figure with the corresponding empirical cumulative distribution functions can help here:



The stair-case pattern is very clear on the ECDF of the discretized data after Durbin's transformation.

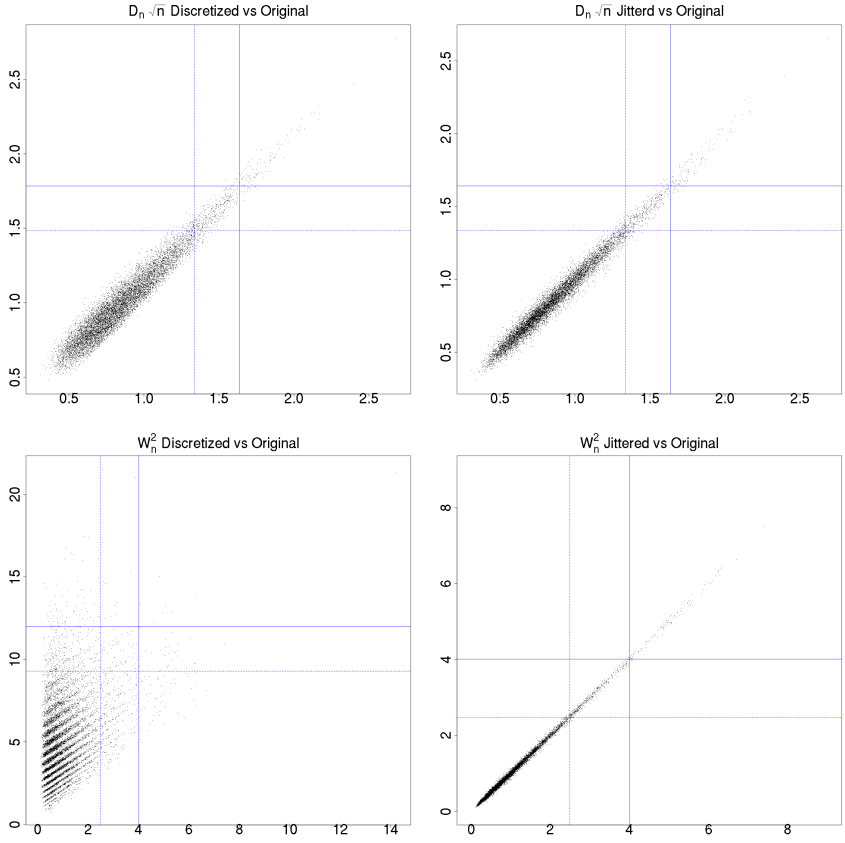
The systematic simulation is done as follows with the empirical 0.95 and 0.99 quantiles for each statistic:

```
set.seed(20061001)
nrep <- 10000
D_W2 <- matrix(0,nr=12,nc=nrep)
rownames(D_W2) <- c("D_o","D_d","D_dj","W2_o","W2_d","W2_dj",
                    "D_o_dt","D_d_dt","D_dj_dt","W2_o_dt","W2_d_dt",
                    "W2_dj_dt")
for (i in 1:nrep) {
  hp <- cumsum(rexp(2000,242.5))
  hp <- hp[hp<6]
  hp_d <- discretize_time(hp)
  hp_dj <- jitter_time(hp_d)
  hp_dt <- DurbinTransform(hp,c(0,6))
  hp_d_dt <- DurbinTransform(hp_d,c(0,6))
  if (any(hp_d_dt==0))
    hp_d_dt[hp_d_dt==0] = 5*.Machine$double.eps
  if (any(hp_d_dt==1))
    hp_d_dt[hp_d_dt==1] = 1-5*.Machine$double.eps
  hp_dj_dt <- DurbinTransform(hp_dj,c(0,6))
  D_W2[,i] <- c(Kolmogorov_D(hp/6),Kolmogorov_D(hp_d/6),
                Kolmogorov_D(hp_dj/6),AndersonDarling_W2(hp/6),
                AndersonDarling_W2(hp_d/6),AndersonDarling_W2(hp_dj/6),
                Kolmogorov_D(hp_dt),Kolmogorov_D(hp_d_dt),
                Kolmogorov_D(hp_dj_dt),AndersonDarling_W2(hp_dt),
                AndersonDarling_W2(hp_d_dt),AndersonDarling_W2(hp_dj_dt))
}
critic_val <- t(apply(D_W2,1,sort))[,c(0.95,0.99)*nrep]
colnames(critic_val) <- c("95%","99%")
critic_val
```

```
             95%       99%
D_o      1.365848  1.622932
D_d      1.365872  1.622980
D_dj     1.366081  1.622934
W2_o     2.552890  3.902664
W2_d     2.552845  3.902644
W2_dj    2.552844  3.902616
D_o_dt   1.337336  1.638557
D_d_dt   1.483138  1.783313
D_dj_dt  1.336199  1.640817
W2_o_dt  2.484505  3.999619
W2_d_dt  9.289051 11.976610
W2_dj_dt 2.463011  4.005226
```

Plotting the statistics for the discretized vs original and the "discretized and then jittered" vs original shows very clearly that the Anderson-Darling

test *should not* be used for discretized data after Durbin's transformation but that jittering the discretized data makes the statistics behave essentially as the ones of the original data (the blue lines show the empirical 0.95 and 0.99 quantiles):



2.2 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 with:

```
library(STAR)
```

2.3 Step by step analysis of the response of neuron 2 from data set `e070528citronellal`

2.3.1 Definition and tests of PSTH construction and stabilization

`make_stabilizedPSTH` definition We define function `make_stabilizedPSTH` that returns a PSTH together with its variance stabilized version, after setting the stimulus onset time at 0, as an object of class `stabilizedPSTH`. The parameters and returned value of this "constructor" are:

```
Parameters
----------
spike_train_list: a list of spike trains (vectors with strictly
  increasing elements), where each element of the list is supposed
  to contain a response and where each list element is assumed
  time locked to a common reference time.
spontaneous_rate: a positive number with the spontaneous rate
  assumed measured separately; if missing, the overall rate obtained
  from spike_train_list is used; the parameter is used to set the
  bin width automatically.
target_mean: a positive number, the desired mean number of events
  per bin under the assumption of homogeneity.
onset: a number giving to the onset time of the stimulus.
region: a two components vector with the number of seconds before
  the onset (a negative number typically) and the number of second
  after the onset one wants to use for the analysis.
stab_method=: a string, either "Freeman-Tukey" (the default,
  x -> sqrt(x)+sqrt(x+1)), "Anscombe" (x -> 2*sqrt(x+3/8)) or "Brown
  et al" (x -> 2*sqrt(x+1/4); the variance stabilizing transformation.

Returns
-------
An object of class (S3) stabilizedPSTH that is fundamentally a list
with the following elements:
st: a vector with the aggregated spike trains (the stimulus comes
    at time 0).
x: a vector with the bins' centers (time starts now at zero).
y: a vector with the stabilized counts.
n: a vector with the actual counts.
n_stim: a scalar, the number of stimulation / trials used to build
  the PSTH.
width: a scalar, the bin width.
stab_method: a string, the variance stabilization method.
spontaneous_rate: a scalar, the spontaneous rate.
support_length: a scalar, the length of the PSTH support.
call: an expression, the matched call.
```

The skeleton of the function definition follows:

```
make_stabilizedPSTH <- function(spike_train_list,
                                spontaneous_rate,
                                target_mean = 3,
                                onset,
                                region = c(-2,8),
                                stab_method = c("Freeman-Tukey",
                                  "Anscombe","Brown et al")) {
  <<make_stabilizedPSTH-parameters-check>>
  <<make_stabilizedPSTH-hist-and-stab>>
  <<make_stabilizedPSTH-output>>
}
```

Listing 1: `make_stabilizedPSTH` definition skeleton

The first part checks the parameters and applies some basic processing:

```
stopifnot(is.list(spike_train_list))
n_stim <- length(spike_train_list)
aggregated_train <- sort(as.vector(unlist(spike_train_list)))
if (missing(spontaneous_rate)) {
    time_span <- ceiling(aggregated_train[length(aggregated_train)])-
        floor(aggregated_train[1])
    spontaneous_rate <- length(aggregated_train)/n_stim/time_span
}
stopifnot(spontaneous_rate > 0)
stopifnot(target_mean > 0)
if (missing(onset)) {
    stopifnot(is.repeatedTrain(spike_train_list))
    onset <- attr(spike_train_list,"stimTimeCourse")[1]
}
from = region[1]+onset
to = region[2]+onset
aggregated_train <- aggregated_train[from <= aggregated_train &
                                     aggregated_train <= to]-onset
stopifnot(stab_method %in% c("Freeman-Tukey","Anscombe","Brown et al"))
```

Listing 2: `make_stabilizedPSTH-parameters-check`

The actual histogram computation and its stabilization comes next:

```
bin_width <- ceiling(target_mean/n_stim/spontaneous_rate*1000)/1000
aggregated_bin <- seq(region[1],region[2]+bin_width,bin_width)
aggregated_hist <- hist(aggregated_train,aggregated_bin,plot=FALSE)
aggregated_counts <- aggregated_hist$counts
if (stab_method[1] == "Freeman-Tukey") {
    y <- sqrt(aggregated_counts)+sqrt(aggregated_counts+1)
} else {
    if (stab_method[1] == "Anscombe") {
        y <- 2*sqrt(aggregated_counts+0.375)
    } else {
        y <- 2*sqrt(aggregated_counts+0.25)
    }
}
```

Listing 3: `make_stabilizedPSTH-hist-and-stab`

The output of the function is then defined:

```
res <- list(st = aggregated_train,
            x = aggregated_bin[-length(aggregated_bin)]+bin_width/2,
            y = y,
            n = aggregated_counts,
            n_stim = n_stim,
            width = bin_width,
            stab_method = stab_method[1],
            spontaneous_rate = spontaneous_rate,
            support_length = diff(region),
            call = match.call())
class(res) <- "stabilizedPSTH"
res
```

Listing 4: `make_stabilizedPSTH-output`

Methods for `stabilizedPSTH` objects We define next a print `method` for the `stabilizedPSTH` objects:

```
print.stabilizedPSTH <- function(x,...) {
  cat(paste("A stabilizedPSTH object build from",x$n_stim,
            "trials with a", x$width, "(s) bin width.\n"))
  cat(paste("  The PSTH is defined on a domain", x$support_length,
            "s long.\n"))
  cat(paste0("  The stimulus comes at second 0.\n"))
  cat(paste("  Variance was stabilized with the", x$stab_method,
            "method.\n"))
}
```

Listing 5: `print` method for stabilizedPSTH instances.

```
plot.stabilizedPSTH <- function(x,y,
                                col,lwd,
                                xlab,ylab,
                                what=c("stab","counts"),
                                ...) {
  stopifnot(what[1] %in% c("stab","counts"))
  what <- what[1]
  if (what == "stab") {
    y <- x$y
    if (missing(ylab)) {
      if (x$stab_method == "Freeman-Tukey")
        ylab <- expression(sqrt(n)+sqrt(n+1))
      if (x$stab_method == "Anscombe")
        ylab <- expression(2*sqrt(n+3/8))
      if (x$stab_method == "Brown et al")
        ylab <- expression(2*sqrt(n+1/4))
    }
  } else {
    y <- x$n
    if (missing(ylab)) ylab <- "Counts per bin"
  }
  if (missing(xlab)) xlab <- "Time (s)"
  if (missing(lwd)) lwd <- 1
  if (missing(col)) col <- 1
  plot(x$x,y,col=col,type="l",lwd=lwd,xlab=xlab,ylab=ylab,...)
}
```

Listing 6: `plot` method for stabilizedPSTH instances.

Tests We now test these functions and methods. We use the data recorded in the spontaneous to estimate the spontaneous discharge frequency:

```
data(e070528spont)
(nu_spont_n2 = length(e070528spont[[2]])/60)
```

```
[1] 19.55
```

We then build the instance of our new class `stabilizedPSTH` for neuron 2 of the data set; we also use the newly defined `print` method for this instance:

```
data(e070528citronellal)
(citron_spsth_n2 = make_stabilizedPSTH(e070528citronellal[[2]],
                                       spontaneous_rate=nu_spont_n2,
                                       region=c(-6,6)))
```

```
A stabilizedPSTH object build from 15 trials with a 0.011 (s) bin width.
 The PSTH is defined on a domain 12 s long.
 The stimulus comes at second 0.
```

```
Variance was stabilized with the Freeman-Tukey method.
```

Is the pre-stimulation period compatible with a homogeneous Poisson process As mentioned in the companion manuscript the tests homogeneous / non-homogeneous Poisson we propose are valid only if the convergence to the Poisson process has been reached. This requires that responses to the successive stimulations were uncorrelated and that enough stimulations were aggregated to loose the "memory" exhibited by the individual responses (they are clearly not Poisson). As a first step we can check if the pre-stimulation period is compatible with the realization of a homogeneous Poisson process. We can perform what Cox and P. A. W. Lewis (1966) call a *uniform conditional test for a Poisson process* on the original data computing both the Kolmogorov and the Anderson-Darling statistics:

```
early_train <- citron_spsth_n2$st[citron_spsth_n2$st < 0] + 6
c(D=Kolmogorov_D(early_train/6),
  W2=AndersonDarling_W2(early_train/6))
```

```
       D       W2
0.8279022 0.6255841
```

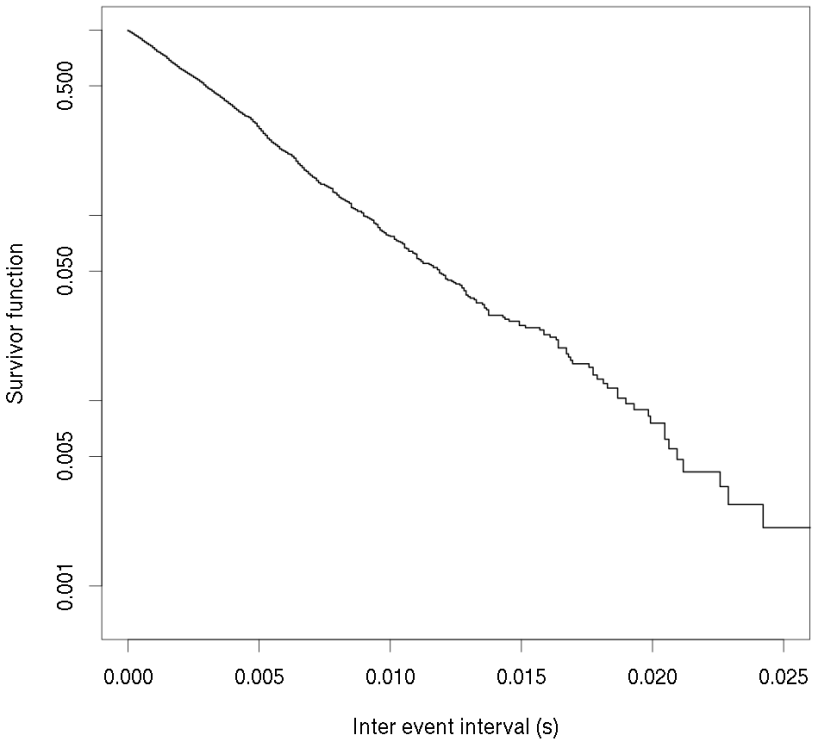
Working Durbin's transformation after jittering the data we get:

```
early_train_j <- jitter_time(early_train,c(0,6))
early_train_t <- DurbinTransform(early_train_j,c(0,6))
c(D=Kolmogorov_D(early_train_t),
  W2=AndersonDarling_W2(early_train_t))
```

```
      D      W2
1.041566 2.161787
```

We can obtain a plot of the log-survivor function of the intervals with:

```
iei_early <- diff(early_train)
par(cex=2)
plot(sort(iei_early),1-seq(along=iei_early)/length(iei_early),
     type="S",log="y",xlab="Inter event interval (s)",
     ylab="Survivor function",xlim=c(0,0.025),lwd=2)
```



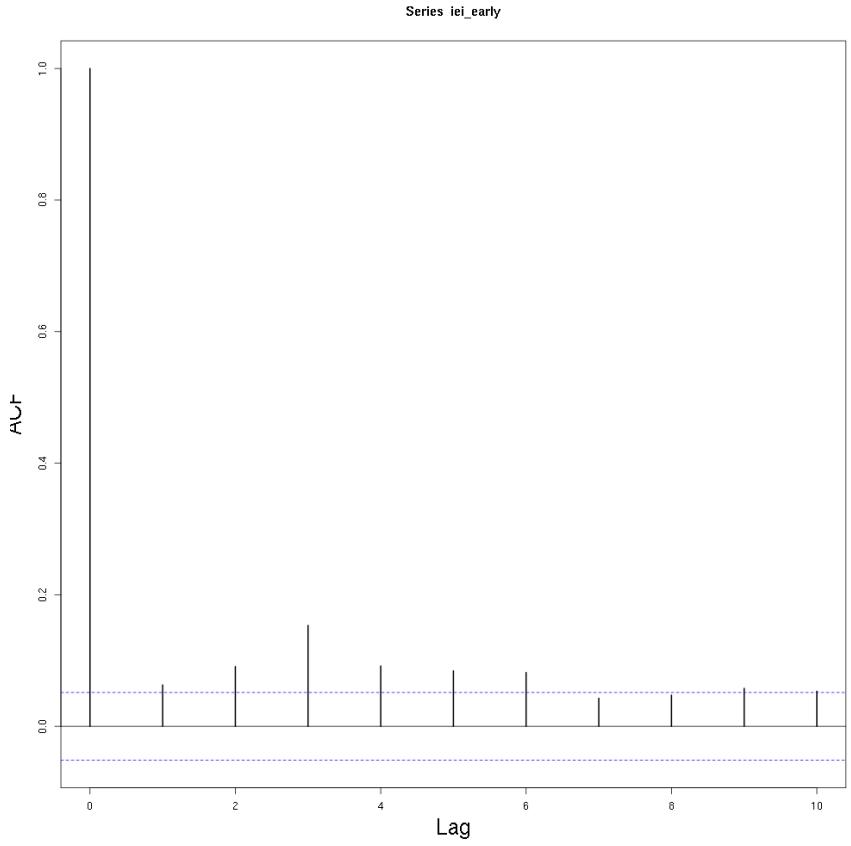
The auto-correlation coefficient of the inter-event interval at lag one is not significantly different from 0 (at the 0.99 level):

```
cor(iei_early[-length(iei_early)],iei_early[-1])*
    sqrt(length(iei_early)-1)
```

```
[1] 2.393833
```

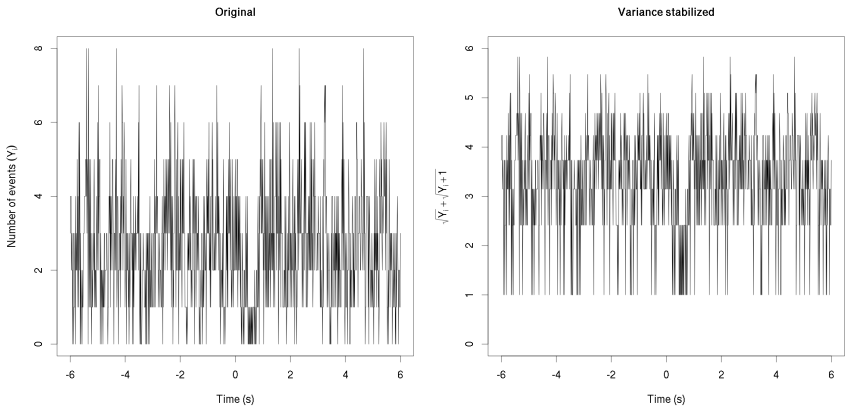
But a plot of the auto-correlation function up to lag 10 does show some signs of correlations:

```
acf(iei_early,lag.max=10,lwd=2,cex.lab=2)
```



PSTH and variance-stabilized-PSTH figure A figure showing the "counts" and the "stabilized counts" is produced by the following commands:

```
layout(matrix(1:2,nc=2))
par(mar=c(5,5,4,1),cex=2)
plot(citron_spsth_n2,what="counts",
     ylab=expression("Number of events"~(Y[i])),
     main="Original")
plot(citron_spsth_n2,
     ylab=expression(sqrt(Y[i])+sqrt(Y[i] + 1)),
     main="Variance stabilized",ylim=c(0,6),xlim=c(-6,6))
```



2.3.2 Kernel smoothing

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 }
```

Listing 7: `tricube-kernel-definition-with-R`

The Nadaraya-Watson estimator We define next a function, `NW_Estimator`, returning the Nadaraya-Watson estimator at a given point. Its parameters and returned value are:

```
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.
```

```
NW_Estimator <- function(x,X,Y,
                         kernel = function(y) tricube_kernel(y,1.0)) {
  w <- kernel(X-x)
  sum(w*Y)/sum(w) }
```

Listing 8: `Nadaraya-Watson-estimator-definition-with-R`

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. Its parameters and returned value are:

```
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 vector with the bandwidth, trace of the smoother and the Cp score.
```

```
Cp_score <- function(X,Y,bw = 1.0,
                     kernel = tricube_kernel,
                     sigma2=1) {
    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(bw,trace, Cp) }}
```

Listing 9: `Cp-score-definition-with-R`

In an actual test setting we would use a few kernel bandwidths (1 to 10) in order to have a moderate Bonferroni correction (giving tighter confidence bands); typically we would used multiples of the initial bin width like: 5, 10, 50, 100, 500 giving:

```
bw_multiplicator <- c(5,10,50,100,500)
bw_vector <- citron_spsth_n2$width*bw_multiplicator
citron_Cp_n2 <- sapply(bw_vector,
                       function(bw) Cp_score(citron_spsth_n2$x,
                                             citron_spsth_n2$y,
                                             bw))
```

Here, for the sake of illustration, a denser set of bandwidth will also be used:

```
bw_multiplicatorDense <- seq(5,101,1)
bw_vectorDense <- citron_spsth_n2$width*bw_multiplicatorDense
citron_CpDense_n2 <- sapply(bw_vectorDense,
                            function(bw) Cp_score(citron_spsth_n2$x,
                                                  citron_spsth_n2$y,
                                                  bw))
```

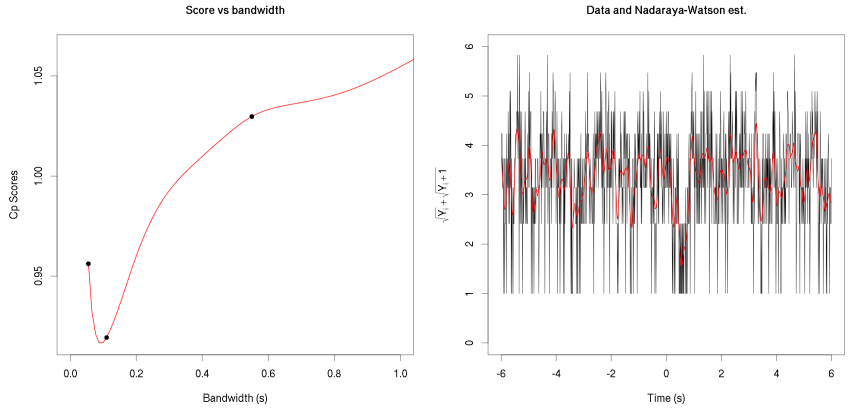
We then extract the bandwidth giving the best (lowest) score and get the corresponding Nadaraya-Watson estimator:

```
min_pos <- which.min(citron_Cp_n2[3,])
bw_best_Cp <- citron_Cp_n2[1,min_pos]
citron_NW_n2 <- sapply(citron_spsth_n2$x,
                       function(x)
                       NW_Estimator(x,
                                    citron_spsth_n2$x,
                                    citron_spsth_n2$y,
                                    kernel = function(y)
                                             tricube_kernel(y,
                                                            bw_best_Cp)))
```

2.3.3 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),cex=2)
plot(bw_vectorDense,citron_CpDense_n2[3,],type="l",col='red',lwd=2,
     xlab='Bandwidth (s)',ylab='Cp Scores',
     main='Score vs bandwidth',xlim=c(0,1))
points(bw_vector,citron_Cp_n2[3,],pch=16)
plot(citron_spsth_n2,
     ylab=expression(sqrt(Y[i])+sqrt(Y[i] + 1)),
     main="Data and Nadaraya-Watson est.",ylim=c(0,6))
lines(citron_spsth_n2$x,citron_NW_n2,col=2,lwd=2)
```



2.3.4 Confidence set for the smoother

κ_0 We get the value of the integral $IK = \left(\int_a^b K'(t)^2 dt\right)^{1/2}$ appearing in $\kappa_0 \approx (b-a)/h\, IK$ with the open source computer algebra system (CAS) `maxima` (`http://maxima.sourceforge.net/`):

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

```
1.498662505306927
```

We then get the κ_0 for neuron 2:

```
(kappa_0_n2 <- citron_spsth_n2$support_length*1.498662505306927/bw_best_Cp)
```

```
[1] 163.4905
```

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)
    (2*(1-pnorm(x)) + kappa*exp(-x^2/2)/pi - alpha)^2
```

Listing 10: `define-tube-target-with-R`

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

```
c_p95 <- optimize(tube_target,c(3,5),
                  alpha=0.05/length(bw_multiplicator),
                  kappa=kappa_0_n2)$minimum
c_p90 <- optimize(tube_target,c(2,5),
                  alpha=0.1/length(bw_multiplicator),
                  kappa=kappa_0_n2)$minimum
```

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–:

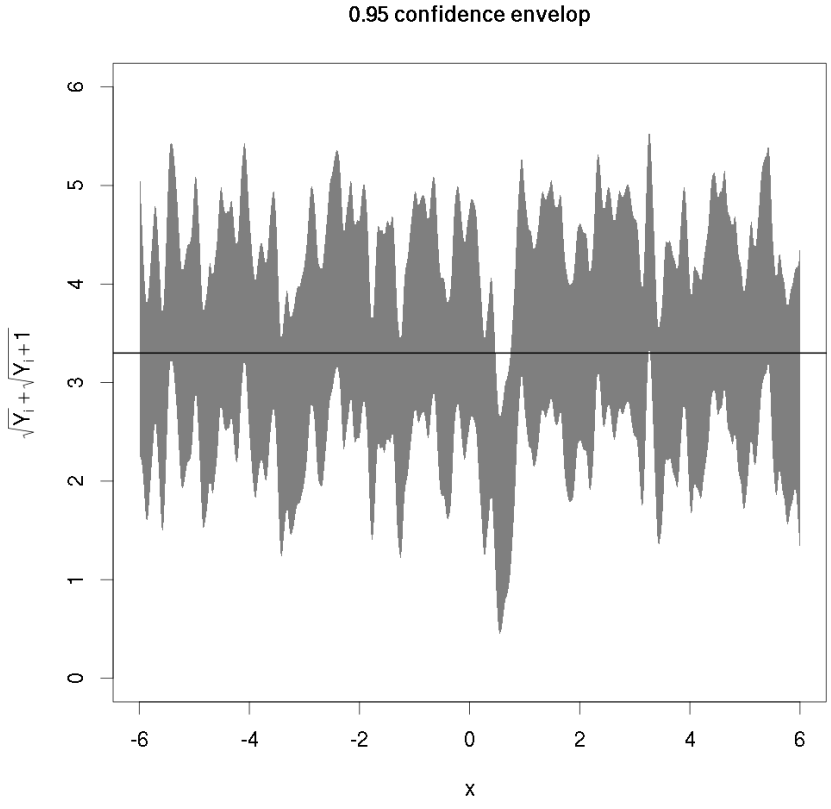
```
make_L <- function(X,kernel = function(y) tricube_kernel(y,1.0)) {
    result <- matrix(0,nr=length(X),nc=length(X))
    ligne <- numeric(length(X))
    for (i in 1:length(X)) {
        ligne <- kernel(X-X[i])
        result[i,] = ligne/sum(ligne) }
    result }
```

Listing 11: `make_L-definition-with-R`

```
n2citron_NW_L_best <- make_L(citron_spsth_n2$x,
                             kernel = function(y)
                             tricube_kernel(y,bw_best_Cp))
n2citron_NW_L_best_norm <- sqrt(apply(n2citron_NW_L_best^2,1,sum))
```

Figure of the smooth estimate with the 0.95 confidence set The equivalent of Fig. 2 in R is simply obtained with:

```
par(mar=c(5,5,4,1),cex=2)
u = citron_NW_n2+c_p95*n2citron_NW_L_best_norm
l = citron_NW_n2-c_p95*n2citron_NW_L_best_norm
x = citron_spsth_n2$x
plot(x,citron_spsth_n2$y,main="0.95 confidence envelop",
     ylab=expression(sqrt(Y[i])+sqrt(Y[i] + 1)),type="n",
     ylim=c(0,6))
polygon(c(x,rev(x)),c(u,rev(l)),border=NA,col="grey50")
abline(h=3.3,lwd=2)
```



Results of the existings tests Applying the Kolmogorov test and the Anderson-Darling test on the data gives:

```
n2_train <- citron_spsth_n2$st + 6
c(D=Kolmogorov_D(n2_train/12),
  W2=AndersonDarling_W2(n2_train/12))
```

```
      D      W2
1.278224 1.039232
```

After jittering and Durbin's transformation we get:

```
n2_train_j <- jitter_time(n2_train,c(0,12))
n2_train_t <- DurbinTransform(n2_train_j,c(0,12))
c(D=Kolmogorov_D(n2_train_t),
  W2=AndersonDarling_W2(n2_train_t))
```

```
      D      W2
1.039290 2.519261
```

So the critical 0.95 quantile of the Anderson-Darling distribution (2.492) is exceeded but not the 0.99 quantile (3.857).

2.3.5 Define class and methods doing the same job

`make_smoothStabilizedPSTH` definition We can now define a new class, `smoothStabilizedPSTH`, whose instances contain all the results linked to the kernel smoothing procedure. The arguments and returned value of the constructor, `make_smoothStabilizedPSTH`, are:

```
Parameters
----------
stabilizedPSTH: an instance of stabilizedPSTH.
bandWidthMultipliers: a vector (but it can also be a scalar) of numbers (> 1)
  by which the initial binwidth is going to be multiplied, defining thereby
  the set of bandwidths that is going to be explored. The longer the vector
  the stronger the Bonferroni correction for the confidence band calculation.
sigma2: a strictly positive number (default 1), the value at which the variance
  has been stabilized.

Returns
-------
An instance of class smoothStabilizedPSTH that is essentially a list with the
following elements:
all the elements of the list making argument stabilizedPSTH, except the call
  element that is changed.
bandWidthMultipliers: a copy of the argument with the same name.
bw_values: a vector, the set of kernel bandwidth that has been explored.
trace_values: a vector, the set of traces of the corresponding smoothing matrix.
```

```
Cp_values: a vector, the set of Mallow's Cp scores obtained.
bw_best_Cp: a scalar, the bandwidth giving the best Cp.
NW: a vector, the Nadaraya-Watson estimator obtained with bw_best_Cp.
L_best_norm: a vector, the corresponding "norm" of the smoothing matrix.
kappa_0: a scalar, the value of kappa_0.
get_c: a function that returns the critical value used to compute the confidence
  band and that takes three arguments:
  alpha: the level of the test (after Bonferroni correction).
  lower: a scalar, the left bound for the optimization (see the built-in
    optimize function).
  upper: a scalar, the right bound for the optimization.
```

The following function definition refers to listings of the previous section. The constructor definition skeleton is:

```
make_smoothStabilizedPSTH <- function(stabilizedPSTH,
                                      bandWidthMultipliers = c(5,10,50,
                                                               100,500),
                                      sigma2=1
                                      ) {

  stopifnot("stabilizedPSTH" %in% class(stabilizedPSTH))
  <<tricube-kernel-definition-with-R>>
  <<Nadaraya-Watson-estimator-definition-with-R>>
  <<Cp-score-definition-with-R>>
  <<make_L-definition-with-R>>
  <<define-tube-target-with-R>>
  stopifnot(all(bandWidthMultipliers > 1))
  stopifnot(sigma2 > 0)
  <<get-Cp-values-and-kappa_0>>
  <<Nadaraya-Watson-and-smoothing-matrix>>
  get_c <- function(alpha=0.05,lower=2,upper=5)
    optimize(tube_target,c(lower,upper),
             alpha=alpha/length(bw_vector),kappa=kappa_0)$minimum
  <<make_smoothStabilizedPSTH-output>>
}
```

`get-Cp-values-and-kappa_0` computes the Cp values at different bandwidth and gets the best before obtaining the `kappa_0`, just like we did before:

```
bw_vector <- stabilizedPSTH$width*bandWidthMultipliers
Cp_values <- sapply(bw_vector,
                    function(bw) Cp_score(stabilizedPSTH$x,
                                          stabilizedPSTH$y,
                                          bw,sigma2=sigma2))
if (length(bw_vector)==1) Cp_values <- matrix(Cp_values,3,1)
bw_best_Cp <- Cp_values[1,which.min(Cp_values[3,])]
if (bw_best_Cp == bw_vector[1] ||
    bw_best_Cp == bw_vector[length(bw_vector)])
    warning("Best Cp value reached at bandwidth extremum.")
kappa_0 <- stabilizedPSTH$support_length*1.498662505306927/bw_best_Cp
```

Listing 12: `get-Cp-values-and-kappa_0`

The Nadaraya-Watson and the smoothing matrix together with the norm of its rows are then obtained:

```
NW <- sapply(stabilizedPSTH$x,
             function(x) NW_Estimator(x,stabilizedPSTH$x,
                                      stabilizedPSTH$y,
                                      kernel = function(y)
                                          tricube_kernel(y,bw_best_Cp)))
L_best <- make_L(stabilizedPSTH$x,
                 kernel = function(y) tricube_kernel(y,bw_best_Cp))
L_best_norm <- sqrt(apply(L_best^2,1,sum))
```

Listing 13: `Nadaraya-Watson-and-smoothing-matrix`

The output is prepared next:

```
res <- list(x=stabilizedPSTH$x,
            y=stabilizedPSTH$y,
            n=stabilizedPSTH$n,
            n_stim=stabilizedPSTH$n_stim,
            width=stabilizedPSTH$width,
            onset=stabilizedPSTH$onset,
            stab_method=stabilizedPSTH$stab_method,
            spontaneous_rate=stabilizedPSTH$spontaneous_rate,
            support_length=stabilizedPSTH$support_length,
            bandWidthMultipliers=bandWidthMultipliers,
            bw_values = Cp_values[1,],
            trace_values=Cp_values[2,],
            Cp_values=Cp_values[3,],
            bw_best_Cp=bw_best_Cp,
            NW=NW,
            L_best_norm=L_best_norm,
            kappa_0=kappa_0,
            get_c = get_c,
            call = match.call())
class(res) <- "smoothStabilizedPSTH"
res
```

Listing 14: `make_smoothStabilizedPSTH-output`

`plot` method for `smoothStabilizedPSTH` instances We define next a `plot` method for `smoothStabilizedPSTH` objects. When considering such an object there are several "facets" one might want to look at:

- The data used to build the smooth estimate, that is, the stabilized PSTH. This can be obtained by setting argument `what` to "stab".
- The smooth estimate itself; this can be obtained by setting argument `what` to "smooth" (default value).

- A confidence band corresponding to a specific level. This can be obtained by setting argument `what` to "band"; then additional arguments: alpha (setting the level of the test *after Bonferroni correction*) as well as lower and upper that are used by function `get_c` of `smoothStabilizedPSTH` objects (these must be specified in order to get the value, through numerical optimization, of parameter c).

If one does not want to do testing but display an estimator of the inhomogeneous Poisson process, argument `scale` can be set to "Hz" instead of it's default value "natural". But one might want to see the evolution of Mallow's Cp score with the bandwidth or or with the smoother's trace; the former is obtained by setting argument `what` to "Cp vs bandwidth" while the latter is obtained by setting `what` to "Cp vs trace".

All argument to both of these methods that are not described here have their classical meaning for `plot` methods.

In order to fit the `plot.smoothStabilizedPSTH` method definition on single pages we define it by pieces. The outline of the method is:

```
plot.smoothStabilizedPSTH <- function(x,y,
                                      col,lwd,
                                      xlab,ylab,
                                      type,
                                      what=c("smooth",
                                             "band",
                                             "stab",
                                             "Cp vs bandwidth",
                                             "Cp vs trace"),
                                      scale=c("natural","Hz"),
                                      alpha=0.05,
                                      lower=2,upper=5,
                                      ...) {
  <<check-par-smoothStabilizedPSTH>>
  if (what %in% c("smooth","band","stab")) {
      <<case-smooth-band-stab-smoothStabilizedPSTH>>
  } else {
      <<case-Cp-smoothStabilizedPSTH>>
  }
}
```

Listing 15: `plot-method-for-smoothStabilizedPSTH-defintion`

Where «check-par-smoothStabilizedPSTH» checks the validity of the parameters passed to the constructor:

```
what <- what[1]
stopifnot(what %in% c("smooth","band","stab","Cp vs bandwidth",
                      "Cp vs trace"))
scale <- scale[1]
stopifnot(scale %in% c("natural","Hz"))
if (missing(type)) type <- "l"
```

Listing 16: `check-par-smoothStabilizedPSTH`

Piece «`case-smooth-band-stab-smoothStabilizedPSTH`» deals with the constructions of plots showing a smooth, a confidence band or the raw data used to get the smooth:

```
if (missing(ylab)) {
    if (scale == "natural") {
        if (x$stab_method == "Freeman-Tukey")
            ylab <- expression(sqrt(n)+sqrt(n+1))
        if (x$stab_method == "Anscombe")
            ylab <- expression(2*sqrt(n+3/8))
        if (x$stab_method == "Brown et al")
            ylab <- expression(2*sqrt(n+1/4))
    } else {
        ylab <- "Frequency (Hz)"
    }
}
if (what == "stab") y <- x$y
if (what == "smooth") y <- x$NW
if (what == "band") {
    y <- x$NW
    c <- x$get_c(alpha,lower,upper)
    u <- y+c*x$L_best_norm
    l <- y-c*x$L_best_norm
}
if (scale == "Hz") {
    denom <- x$n_stim*x$width
    if (x$stab_method == "Freeman-Tukey")
        InvFct <- function(y) {
            y <- pmax(y,1)
            ((y^2-1)/2/y)^2/denom }
    if (x$stab_method == "Anscombe")
        InvFct <- function(y) {
            y <- pmax(y,2*sqrt(3/8))
            (y^2/4 + sqrt(1.5)/4/y - 11/8/y^2 - 1/8)/denom }
    if (x$stab_method == "Brown et al")
        InvFct <- function(y) {
            y <- pmax(y,1)
            (y^2/4-0.25)/denom }
    y <- InvFct(y)
    if (what == "band") {
        u <- InvFct(u)
        l <- InvFct(l)
    }
}
if (missing(xlab)) xlab <- "Time (s)"
if (missing(lwd)) lwd <- 1
if (missing(col)) col <- 1
if (what %in% c("stab","smooth")) {
    plot(x$x,y,col=col,type=type,lwd=lwd,xlab=xlab,ylab=ylab,...)
} else {
    plot(c(x$x,x$x),c(u,l),type="n",lwd=lwd,xlab=xlab,ylab=ylab,...)
    polygon(c(x$x,rev(x$x)),
            c(u,rev(l)),col=col,border=NA)
}
```

Listing 17: `case-smooth-band-stab-smoothStabilizedPSTH`

Piece «case-Cp-smoothStabilizedPSTH» plots the Cp against the bandwidth or against the smoothing matrix trace:

```
y <- x$Cp_values
if (missing(ylab)) ylab <- "Cp"
if (what == "Cp vs bandwidth") {
    X <- x$bw_values
    if (missing(xlab)) xlab <- "Bandwidth (s)"
} else {
    X <- x$trace_values
    if (missing(xlab)) xlab <- "Smoother trace"
}
if (missing(lwd)) lwd <- 1
if (missing(col)) col <- 1
plot(X,y,col=col,type=type,lwd=lwd,xlab=xlab,ylab=ylab,...)
```

Listing 18: `case-Cp-smoothStabilizedPSTH`

`lines` method for `smoothStabilizedPSTH` instances The `lines` works in the same way as the `plot` one except that it can't generate Cp values displays.

```
lines.smoothStabilizedPSTH <- function(x,
                                       what=c("smooth","band","stab"),
                                       scale=c("natural","Hz"),
                                       alpha=0.05,
                                       lower=2,upper=5,
                                       ...) {
  what <- what[1]
  stopifnot(what %in% c("smooth","band","stab"))
  scale <- scale[1]
  stopifnot(scale %in% c("natural","Hz"))
  if (what == "stab") y <- x$y
  if (what == "smooth") y <- x$NW
  if (what == "band") {
    y <- x$NW
    c <- x$get_c(alpha,lower,upper)
    u <- y+c*x$L_best_norm
    l <- y-c*x$L_best_norm
  }
  if (scale == "Hz") {
    if (x$stab_method == "Freeman-Tukey")
        InvFct <- function(y) {
            y <- pmax(y,1)
            ((y^2-1)/2/y)^2/x$n_stim/x$width }
    if (x$stab_method == "Anscombe")
        InvFct <- function(y) {
            y <- pmax(y,2*sqrt(3/8))
            (y^2/4 + sqrt(1.5)/4/y -
               11/8/y^2 - 1/8)/x$n_stim/x$width }
    if (x$stab_method == "Brown et al")
        InvFct <- function(y) {
            y <- pmax(y,1)
            (y^2/4-0.25)/x$n_stim/x$width }
    y <- InvFct(y)
    if (what == "band") {
      u <- InvFct(u)
      l <- InvFct(l)
    }
  }
  if (what %in% c("stab","smooth")) {
    lines(x$x,y,...)
  } else {
    polygon(c(x$x,rev(x$x)),
            c(u,rev(l)),border=NA,...)
  }
}
```

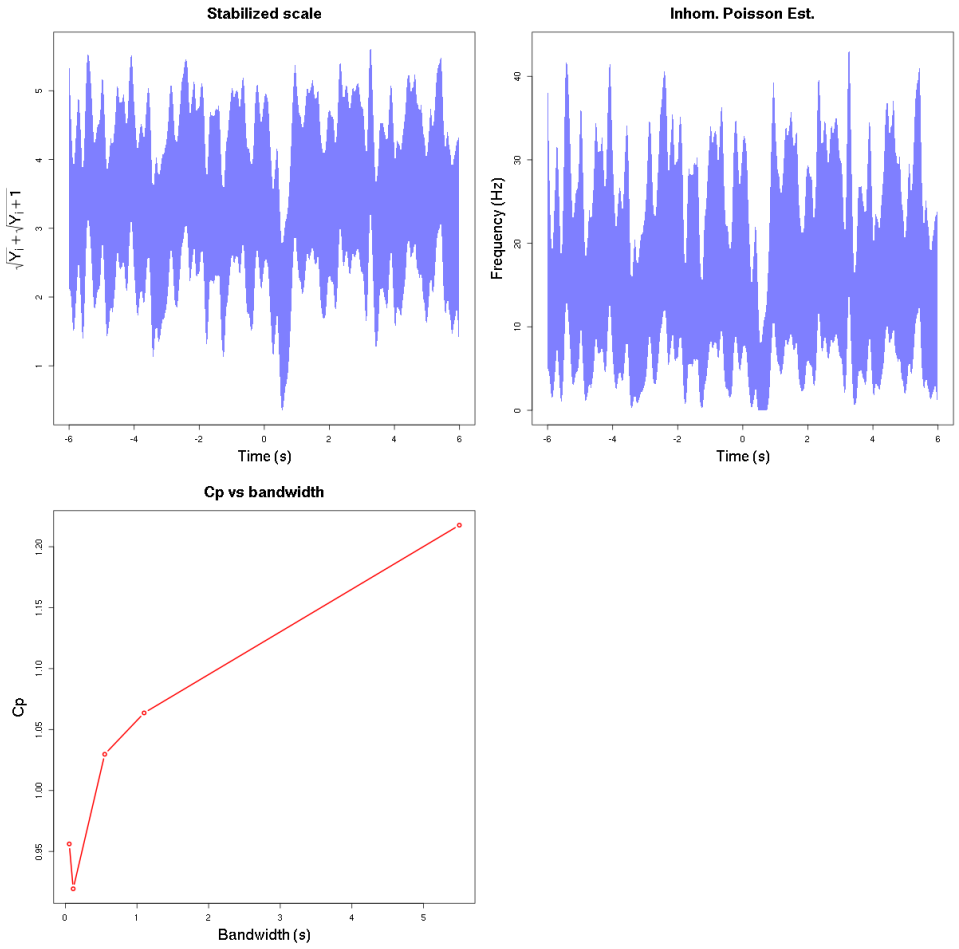
Listing 19: `lines-method-for-smoothStabilizedPSTH-defintion`

Tests So let us make an example of use by first getting the `smoothStabilizedPSTH` of neuron 2:

```
citron_sspsth_n2 = make_smoothStabilizedPSTH(citron_spsth_n2)
```

The next figure shows the 99% confidence bands (left), the Cp values as a function of the bandwidth (middle) and the estimated inhomogeneous Poisson intensity with 95% confidence bands (right):

```
layout(matrix(1:4,2,2))
par(mar=c(5,5,4,1),cex.lab=2,cex.main=2)
plot(citron_sspsth_n2,what="band",alpha=0.01,
     col=rgb(0,0,1,0.5),main=paste("Stabilized scale"),
     ylab=expression(sqrt(Y[i])+sqrt(Y[i] + 1)))
plot(citron_sspsth_n2,what="Cp vs bandwidth",type="b",
       col=2,lwd=2,main="Cp vs bandwidth")
plot(citron_sspsth_n2,what="band",alpha=0.05,
     col=rgb(0,0,1,0.5),main=paste("Inhom. Poisson Est."),scale="Hz")
```



2.4 Systematic analysis

We can now analyze all the odor responses of the data set in the same way, building 99% confidence bands using 5 seconds before the stimulus onset and 6 seconds after it (the longest compromise among our data sets).

2.4.1 Experiment e060817

We get the spontaneous discharge rates of the three neurons of experiment e060817:

```
data(e060817spont)
(e060817_spont_nu = sapply(e060817spont,length)/60)
```

```
neuron 1  neuron 2  neuron 3
8.816667 20.483333 13.016667
```

We create next `stabilizedPSTH` instances corresponding to the citronellal responses of each neuron as well as the `smoothStabilizedPSTH`:

```
data(e060817citron)
e060817citron_spsth = lapply(1:3,
  function(idx) make_stabilizedPSTH(e060817citron[[idx]],
                                    e060817_spont_nu[idx],
                                    region = c(-5,6)))
e060817citron_sspsth = lapply(e060817citron_spsth,
                              make_smoothStabilizedPSTH)
```

The terpineol and mixture responses are processed with:

```
data(e060817terpi)
e060817terpi_spsth = lapply(1:3,
  function(idx) make_stabilizedPSTH(e060817terpi[[idx]],
                                    e060817_spont_nu[idx],
                                    region = c(-5,6)))
e060817terpi_sspsth = lapply(e060817terpi_spsth,
                             make_smoothStabilizedPSTH)
data(e060817mix)
e060817mix_spsth = lapply(1:3,
  function(idx) make_stabilizedPSTH(e060817mix[[idx]],
                                    e060817_spont_nu[idx],
                                    region = c(-5,6)))
e060817mix_sspsth = lapply(e060817mix_spsth,
                           make_smoothStabilizedPSTH)
```

2.4.2 Experiment `e060824`

This data set contains only two neurons and a single odor response (to citral). The analysis is done with:

```
data(e060824spont)
(e060824_spont_nu = sapply(e060824spont,length)/59)
data(e060824citral)
e060824citral_spsth = lapply(1:2,
  function(idx) make_stabilizedPSTH(e060824citral[[idx]],
                                    e060824_spont_nu[idx],
                                    region = c(-5,6)))
e060824citral_sspsth = lapply(e060824citral_spsth,
                              make_smoothStabilizedPSTH)
```

2.4.3 Experiment `e060517`

This data set contains the responses of three neurons to ionon:

```
data(e060517spont)
(e060517_spont_nu = sapply(e060517spont,length)/61)
data(e060517ionon)
e060517ionon_spsth = lapply(1:3,
  function(idx) make_stabilizedPSTH(e060517ionon[[idx]],
                                    e060517_spont_nu[idx],
                                    region = c(-5,6)))
e060517ionon_sspsth = lapply(e060517ionon_spsth,
                             make_smoothStabilizedPSTH)
```

2.4.4 Experiment e070528

This data set contains the responses of four neurons to citronellal:

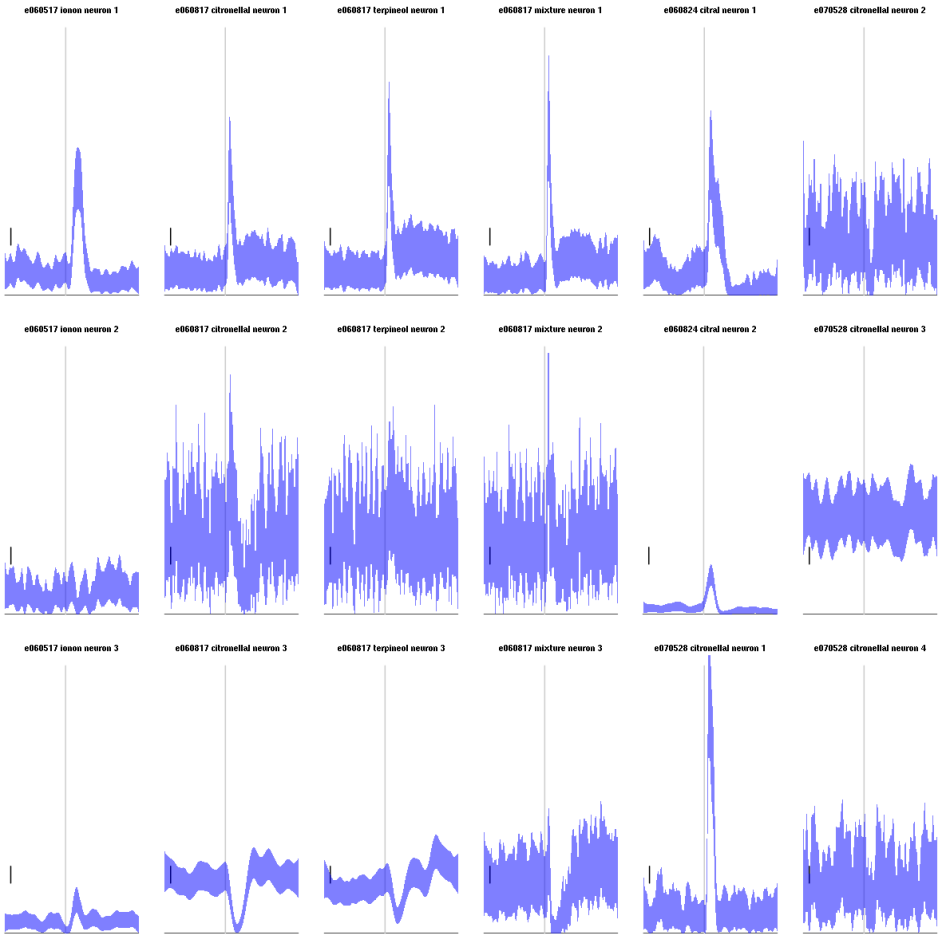
```
data(e070528spont)
(e070528_spont_nu = sapply(e070528spont,length)/60)
data(e070528citronellal)
e070528citronellal_spsth = lapply(1:4,
  function(idx) make_stabilizedPSTH(e070528citronellal[[idx]],
                                    e070528_spont_nu[idx],
                                    region = c(-5,6)))
e070528citronellal_sspsth = lapply(e070528citronellal_spsth,
                                   make_smoothStabilizedPSTH)
```

2.4.5 A new version of Fig. 8 of Pouzat and Chaffiol (2009)

We can now make a new version of Fig. 8 of Pouzat and Chaffiol (2009) with 99% confidence bands instead of 95% pointwise confidence intervals using the "natural" scale, the one on which the variance has been stabilized:

```
ylim <- c(0,80)
opar <- par(mar=c(2,1,2,1))
on.exit(par(opar))
layout(matrix(1:18,nrow=3))
plotList <- function(list,start,middle) {
    n <- length(list)
    sapply(1:n,
           function(idx) {
               sspsth <- list[[idx]]
               plot(sspsth,what="smooth",scale="Hz",type="n",xlab="",
                    ylab="",xaxt="n",yaxt="n",bty="n",ylim=ylim,
                    main=paste(start,middle,"neuron",idx))
               segments(-4.5,15,-4.5,20,lwd=2)
               segments(-5,0,6,0)
               segments(0,0,0,ylim[2],col="grey80",lwd=2)
               lines(sspsth,what="band",scale="Hz",col=rgb(0,0,1,0.5),
                     alpha=0.01)
           })
}

plotList(e060517ionon_sspsth,"e060517","ionon")
plotList(e060817citron_sspsth,"e060817","citronellal")
plotList(e060817terpi_sspsth,"e060817","terpineol")
plotList(e060817mix_sspsth,"e060817","mixture")
plotList(e060824citral_sspsth,"e060824","citral")
plotList(e070528citronellal_sspsth,"e070528","citronellal")
```



2.5 Testing identity

2.5.1 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(1-p_vector,
                     sapply(p_vector,
                            get_a_b)))
(sqrt_coef <- round(sqrt_coef,digits=3))
```

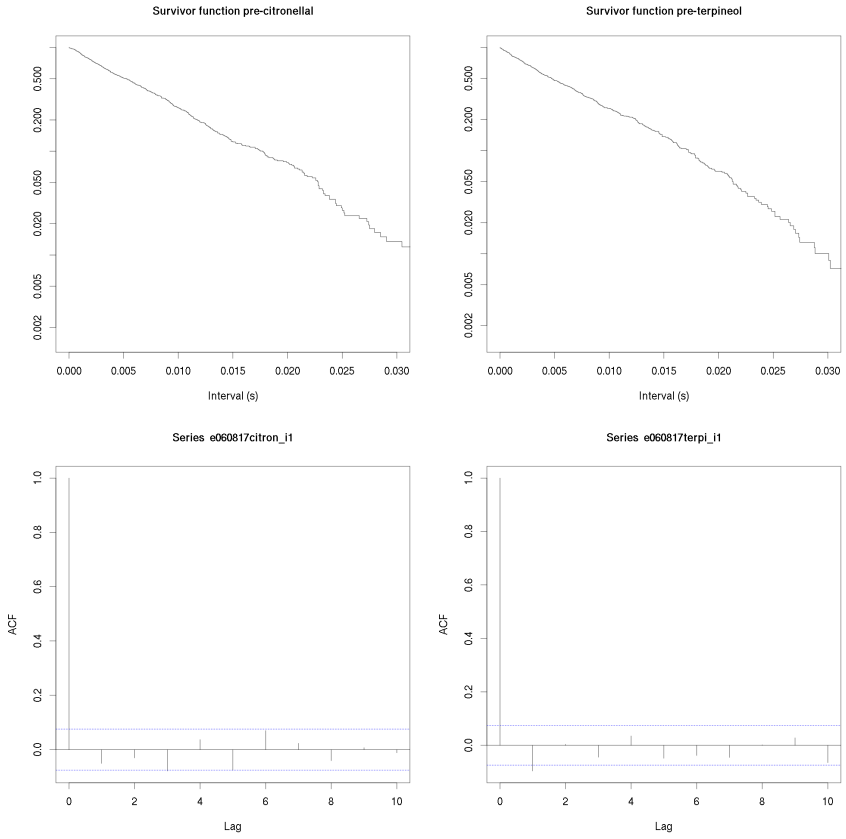
```
              a     b
 [1,] 0.90 0.292 2.077
 [2,] 0.91 0.293 2.120
 [3,] 0.92 0.295 2.167
 [4,] 0.93 0.296 2.220
 [5,] 0.94 0.298 2.279
 [6,] 0.95 0.300 2.348
 [7,] 0.96 0.302 2.430
 [8,] 0.97 0.305 2.531
 [9,] 0.98 0.308 2.668
[10,] 0.99 0.313 2.890
```

Back to the analysis of the data set We have already built the citronellal and terpineol PSTHs of neuron 1. We start by checking that during the pre-stimulation period the aggregated processes have the properties of an homogeneous Poisson process.

```
e060817citron_n1 <- e060817citron_spsth[[1]]$st
e060817citron_e1 <- e060817citron_n1[e060817citron_n1<0]+5
e060817citron_j1 <- jitter_time(e060817citron_e1,c(0,5))
e060817citron_t1 <- DurbinTransform(e060817citron_j1,c(0,5))
e060817citron_test <- c(D_o=Kolmogorov_D(e060817citron_e1/5),
                        W2_o=AndersonDarling_W2(e060817citron_e1/5),
                        D_t=Kolmogorov_D(e060817citron_t1),
                        W2_t=AndersonDarling_W2(e060817citron_t1))
e060817terpi_n1 <- e060817terpi_spsth[[1]]$st
e060817terpi_e1 <- e060817terpi_n1[e060817terpi_n1<0]+5
e060817terpi_j1 <- jitter_time(e060817terpi_e1,c(0,5))
e060817terpi_t1 <- DurbinTransform(e060817terpi_j1,c(0,5))
e060817terpi_test <- c(D_o=Kolmogorov_D(e060817terpi_e1/5),
                        W2_o=AndersonDarling_W2(e060817terpi_e1/5),
                        D_t=Kolmogorov_D(e060817terpi_t1),
                        W2_t=AndersonDarling_W2(e060817terpi_t1))
matrix(c(e060817citron_test,e060817terpi_test),
       nr=2,byrow=TRUE,
       dimnames=list(c("citronellal","terpineol"),
         c("D original","W2 original",
           "D transformed","W2 transformed")))
```

```
           D original W2 original D transformed W2 transformed
citronellal  0.8559287   0.8342023     0.7424727      0.6534344
terpineol    0.5494214   0.3413150     1.0418106      1.3340762
```

The log-survivor function as well as the auto-correlation function of the inter event intervals with the two stimulations are:



We now want to build `stabilizedPSTH` instances corresponding to the even and odd terpineol stimulations:

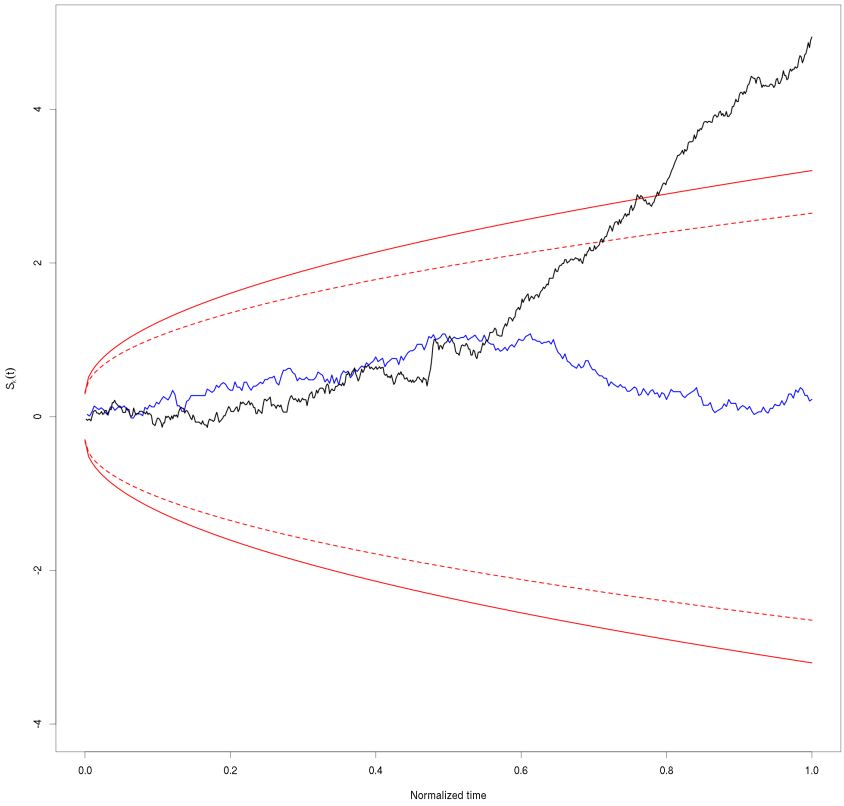
```
e060817terpiOdd_n1 <- e060817terpi[[1]]
e060817terpiOdd_n1[(1:10)*2] <- NULL
terpiOdd_n1_spsth <- make_stabilizedPSTH(e060817terpiOdd_n1,
                                         e060817_spont_nu[1],
                                         region = c(-5,6))
e060817terpiEven_n1 <- e060817terpi[[1]]
e060817terpiEven_n1[(1:10)*2-1] <- NULL
terpiEven_n1_spsth <- make_stabilizedPSTH(e060817terpiEven_n1,
                                          e060817_spont_nu[1],
                                          region = c(-5,6))
```

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

```
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)

xx <- seq(0,1,len=201)
par(cex=2)
plot(xx,c95(xx),type="l",col='red',lwd=3,lty='dashed',
     ylim=c(-4,5),xlab="Normalized time",
     ylab=expression(S[k](t)))
lines(xx,-c95(xx),col='red',lwd=3,lty='dashed')
lines(xx,c99(xx),col='red',lwd=3)
lines(xx,-c99(xx),col='red',lwd=3)
X <- seq(along=terpiOdd_n1_spsth$y)/length(terpiOdd_n1_spsth$y)
Yp <- terpiOdd_n1_spsth$y
Ym <- terpiEven_n1_spsth$y
Y <- cumsum(Yp-Ym)/sqrt(length(Yp))/sqrt(2)
lines(X,Y,col='blue',lwd=3)
X <- seq(along=e060817citron_spsth[[1]]$x)/
    length(e060817citron_spsth[[1]]$x)
Yp <- e060817terpi_spsth[[1]]$y
Ym <- e060817citron_spsth[[1]]$y
Y <- cumsum(Yp-Ym)/sqrt(length(Yp))/sqrt(2)
lines(X,Y,col='black',lwd=3)
```

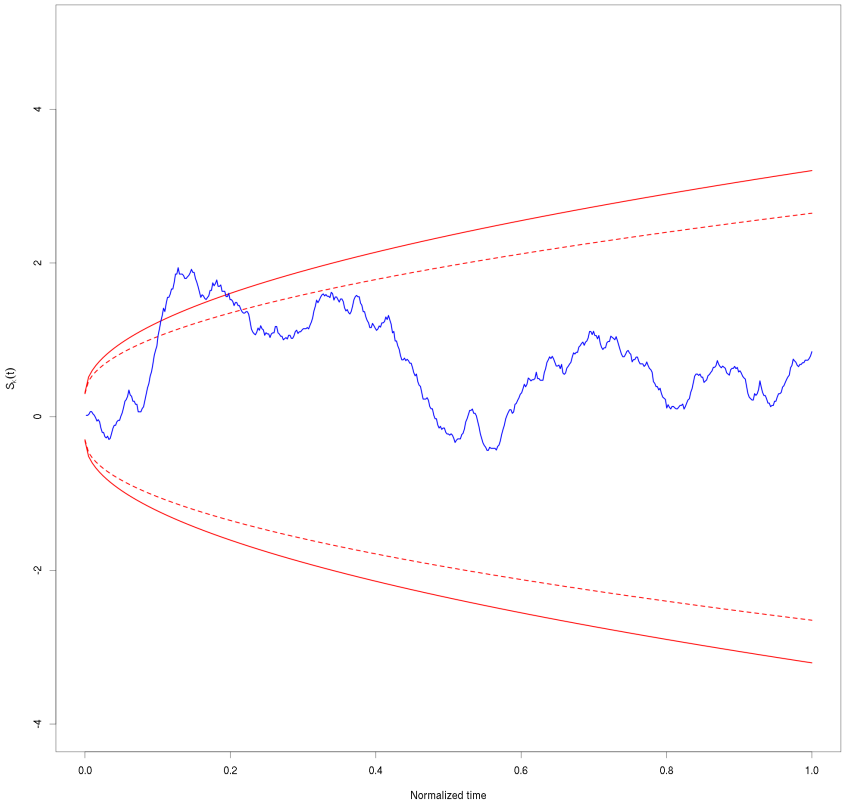


We now consider the citronellal response of neuron 2 from data set `e070528`. The idea here is to compare the 6 seconds prior to stimulus presentation with the 6 seconds after. So we start by building 2 `stabilizedPSTH` instance corresponding to the two parts:

```
citron_spsth_n2_before = make_stabilizedPSTH(e070528citronellal[[2]],
  spontaneous_rate=e070528_spont_nu[2],
  region = c(-6,0))
citron_spsth_n2_after = make_stabilizedPSTH(e070528citronellal[[2]],
  spontaneous_rate=e070528_spont_nu[2],
  region = c(0,6))
Y_before = citron_spsth_n2_before$y
Y_after = citron_spsth_n2_after$y
Y_diff = (Y_before-Y_after)/sqrt(2)
Y_NCM = cumsum(Y_diff)/sqrt(length(Y_diff))
X_NCM = (1:length(Y_diff))/length(Y_diff)
```

The test figure is obtained with:

```
par(cex=2)
plot(xx,c95(xx),type="l",col='red',lwd=3,lty='dashed',
     ylim=c(-4,5),xlab="Normalized time",
     ylab=expression(S[k](t)))
lines(xx,-c95(xx),col='red',lwd=3,lty='dashed')
lines(xx,c99(xx),col='red',lwd=3)
lines(xx,-c99(xx),col='red',lwd=3)
lines(X_NCM,Y_NCM,col='blue',lwd=3)
```



2.6 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

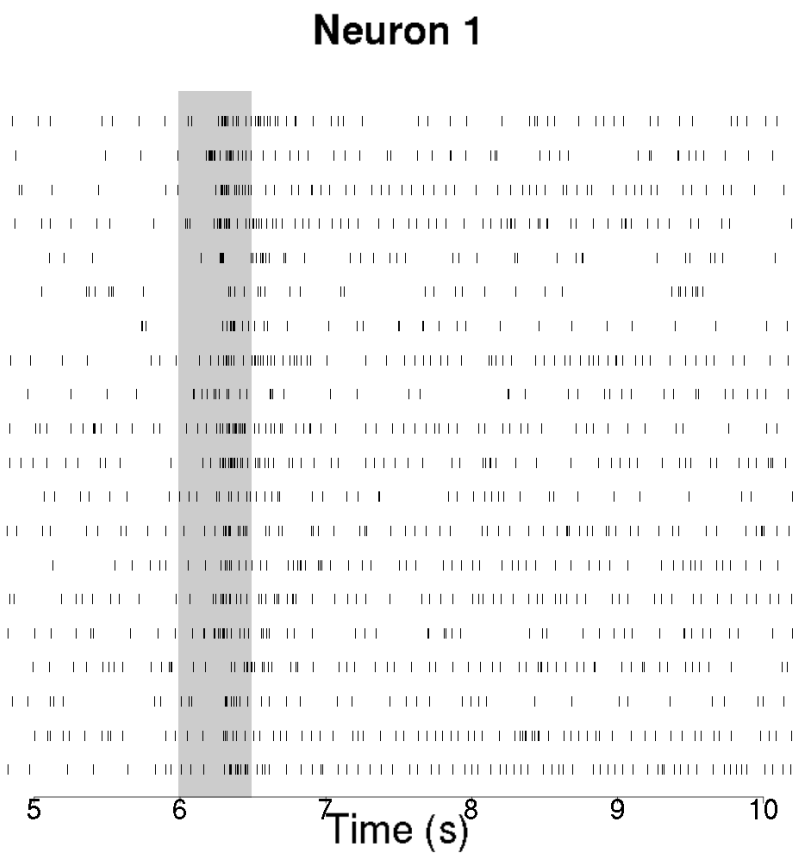
2.7 Raster plots

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)
}
```

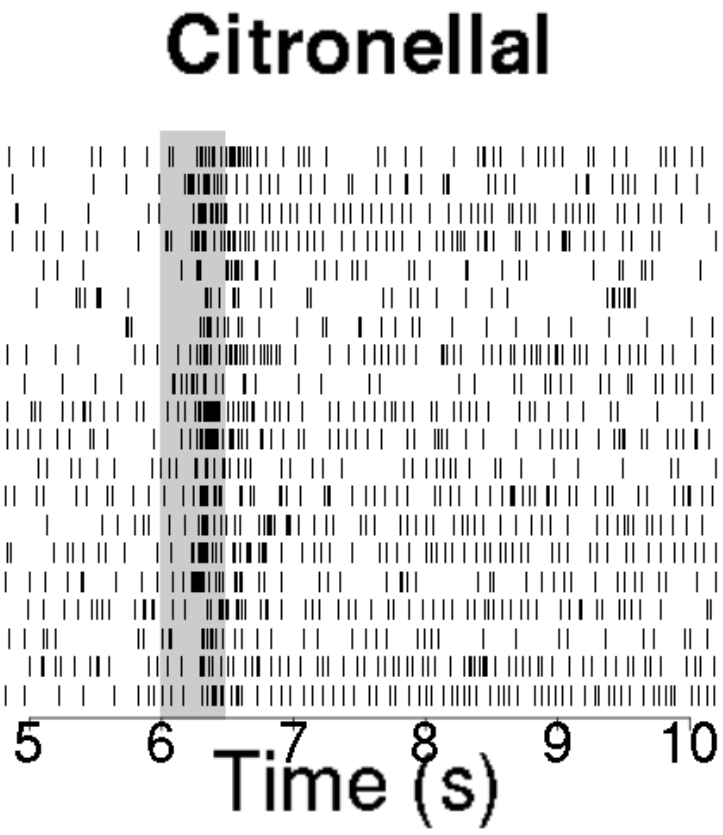
The first raster plot is then obtained with (compared to the `Python` version, the stimulus onset time is not set to zero and the stimulus on period is represented by the grey background):

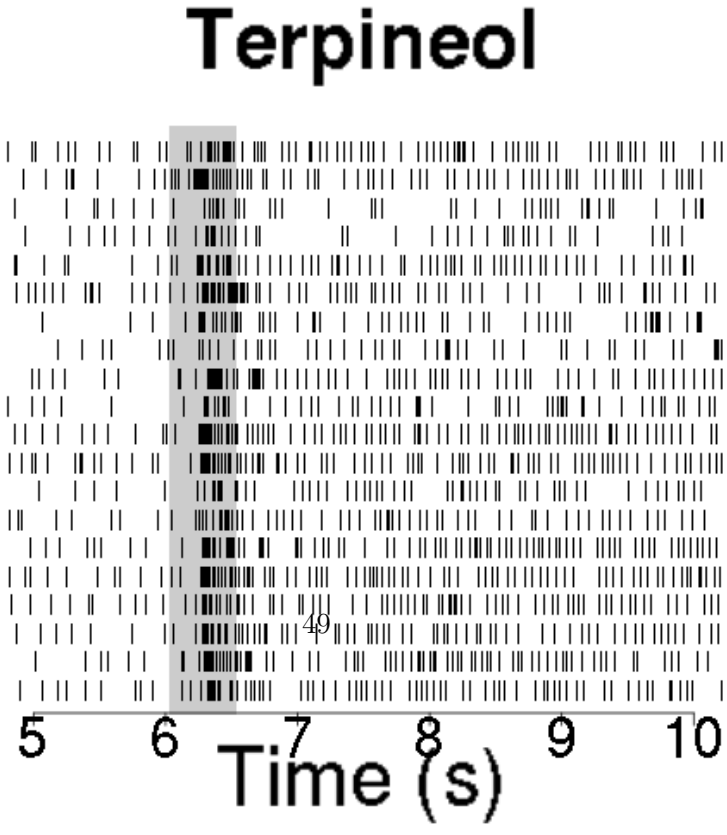
```
par(cex.axis=3,cex.lab=4,cex.main=4,mar=c(5,5,5,1))
mkRaster(e060817citron[[1]],
         stimTimeCourse=attr(e060817citron[["neuron 1"]],"stimTimeCourse"),
         xlab="Time (s)",ylab="",main="Neuron 1",xlim=c(5,10))
```



The second raster is built with:

```
layout(matrix(1:2,nr=2))
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))
```

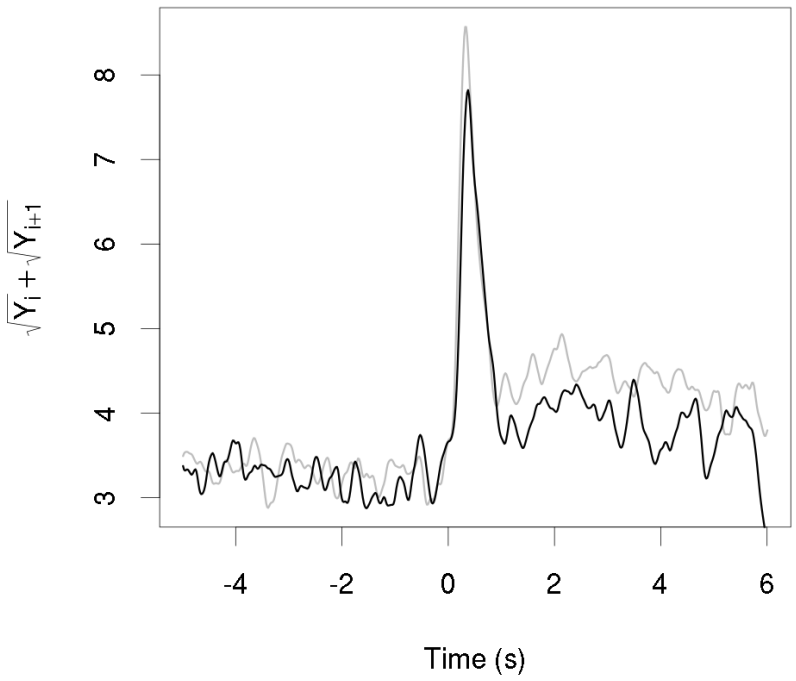

Citronellal
5
6
7
8
9
10
Time (s)


Terpineol
5
6
7
8
9
10
Time (s)

2.8 Terpineol and citronellal responses of neuron 1 from e060817

We create a figure showing the smoothStabilizedPSTH instances:

```
par(cex=3)
plot(e060817terpi_sspsth[[1]],what="smooth",
     ylab=expression(sqrt(Y[i])+sqrt(Y[i+1])),lwd=3,col='grey')
lines(e060817citron_sspsth[[1]],what="smooth",lwd=3)
```



References

Cox, D. R. and P. A. W. Lewis (1966). *The Statistical Analysis of Series of Events*. John Wiley & Sons.

Durbin, J. (1961). "Some Methods of Constructing Exact Tests". In: *Biometrika* 48.1/2, pp. 41–55.

Lewis, Peter A. W. (1965). "Some Results on Tests for Poisson Processes". In: *Biometrika* 52.1/2, pp. 67–77.

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.

Marsaglia, George and John Marsaglia (2004). "Evaluating the Anderson-Darling Distribution". In: *Journal of Statistical Software* 9.2, pp. 1–5.

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.

Pouzat, Christophe and Georgios Is. Detorakis (2014). "SPySort: Neuronal Spike Sorting with Python". In: *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014)*. Ed. by Pierre de Buyl and Nelle Varoquaux, pp. 27–34.