Table des matières

List of Algorithms

1 Notation

We denote by K the number of intervals (partitions) of the time interval $(0, T^{\max})$, each time interval having a size δ (most of the time we have $K\,\delta = T^{\max}$).

We introduce the following notation for $0 \leqslant k \leqslant K-1$ ($\mathbb{I}_k$ is an open interval on the left and closed on the right) to have a shorter form

$$\begin{aligned} \mathbb{I}_k &= (k\,\delta, (k+1)\,\delta] \\ (t - \mathbb{I}_k) &= [t - (k+1)\,\delta,\, t - k\,\delta) \quad \forall t \end{aligned}$$

We assume that $(0, T^{\max})$ corresponds to the union of the partitions (no overlapping).

$$(0, T^{\max}) = \bigcup_{k=0}^{K-1} \mathbb{I}_k$$

Numbering of partitions starts from 0 ! In Matlab or in R, indices start from 1. As we take into account μ index, index $k+2$ in b corresponds to the k-th partition.

We choose γ equal to 3, in the computation of d

We denote by $\varphi_k(t)$ or by $N_{[t-k\delta,t-(k-1)\delta)}$ (or by $N_{t-\mathbb{I}_{k-1}}$) the number of spikes of the point process in the interval $[t-k\delta, t-(k-1)\delta)$ (or $t-\mathbb{I}_{k-1}$).

For $1 \leqslant k \leqslant K$

$$\begin{array}{rcl} \varphi_k(t) & = & N_{[t-k\delta,T-(k-1)\delta)} = N_{t-\mathbb{I}_{k-1}} \\ \varphi_k(t) & = & \text{number of spikes of the point process in the interval } (t-\mathbb{I}_{k-1}) \end{array} \tag{1}$$

φ_k is a piecewise constant function. Représenter un exemple

2 Matrix computation

We will compute here vectors and matrices

For the sake of simplicity, we denote by T_i the spike times contained in the spike array.

$$A_{ij} = m \text{ iff } (T_j - T_i) \in \mathbb{I}_m \Leftrightarrow m\,\delta < T_j - T_i \leqslant (m+1)\delta$$

$$b_k = \int_0^{Tmax} \psi_t(\varphi_k)\, dN_t \;\; \forall k \in [\![0, K-1[\![$$

Deterministic way.

We have, for $k \geqslant 1$

$$\begin{array}{rcl} b_k & = & \displaystyle\int_0^{Tmax} \varphi_k(t)\, dN_t = \int_0^{Tmax} N_{[t-k\delta,t-(k-1)\delta)}\, dN_t = \int_0^{Tmax} N_{t-\mathbb{I}_{k-1}}\, dN_t \\ b_k & = & \displaystyle\int_0^{Tmax} \varphi_k(t)\, dN_t = \int \sum_{T<T^{\max}} N_{[T-k\delta,T-(k-1)\delta)} = \sum_{T<T^{\max}} N_{T-\mathbb{I}_{k-1}} \end{array} \tag{2}$$

$$d_k = \sqrt{2\gamma \log(T^{\max}) \int_0^{T^{\max}} \psi_t^2(\varphi_k)\, dN_t} + \frac{\gamma \log(T^{\max})}{3} \sup_{t \in [0,T^{\max}]} |\psi_t(\varphi_k)|$$

Deterministic way

$$d_k = \sqrt{2\gamma \log(T^{\max}) \int_0^{T^{\max}} \varphi_k^2(t)\, dN_t} + \frac{\gamma \log(T^{\max})}{3} \sup_{t \in [0,T^{\max}]} |\varphi_k(t))|$$

$$G_{kl} = \int_0^{Tmax} \psi_t(\varphi_k)\, \psi_t(\varphi_l)\; dt \;\;\; \forall k,l \in [\![0, K-1[\![$$

Deterministic way

$$G_{kl} = \int_0^{Tmax} \sum_{T_i, T_j < t} \mathbb{1}_{\mathbb{I}_k}(t-T_i)\, \mathbb{1}_{\mathbb{I}_l}(t-T_j) \forall k,l \in [\![0, K-1[\![$$

Algorithm 1 Partial computation of A and b: countpair function

Usage: countpair(spike, del, k, A)

Description: Partial computation of A and computation of b for the k-th partition

Input: spike ; array of spike times

Input: del ; δ, delay

Input: k ; partition index ($k \in [\![0, K-1]\!]$)

Input: A ; N-by-N matrix already initialized # N is the number of spikes

Output: Returns A and count

```
count ← 0                                  # count corresponds to b[k+2]
N ← length(spike)                          # N is the number of spikes
jstart ← which(spike>0)[1]                 # first index such that spike > 0
for j=jstart:N do
    T_low ← spike[j] - (k+1)*del
    T_up ← spike[j] - k*del
    i ← j-1
    while ((i>0) & (i<N)) do
        if spike[i] ∈ [T_low, T_up) then
            count ← count + 1
            A[i,j] ← k
        else
            if (spike[i] < T_low) then
                break
            end if
        end if
    end while
end for
return list(count, A)
```

3 Ordinary Least Squares

4 Lasso shooting algorithm

5 Active set

Algorithm 2 Full computation of A

Usage: computeA(spike, del, K)
Description: Full computation of A using *countpair* function
Input: spike ; array of spike times
Input: del ; δ
Input: K ; number of partitions
Output: A ; N-by-N matrix # N is the number of spikes (K+1) (see 2)

```
N ← length(spike)
A ← matrix(-2, nrow=N, ncol=N)                # initialization
for k=0:(K-1) do
    A ← countpair(spike, del, k, A)[[2]]
end for
return A
```

Algorithm 3 Full computation of b

Usage: computeb_V1(spike, del, Tmax)
Description: Full computation of b using *countpair* function
Input: spike ; array of spike times
Input: del ; δ
Input: K ; number of partitions
Output: b ; (K+1) vector (see **??**)

```
K ← floor(Tmax/del)
N ← length(spike)
b ← vector(mode='double', length=K+1)                # initialization
jstart ← which(spike>0)[1]                            # first index such that spike > 0
b[1] ← N        # A verifier: c'est le nb de spikes dans (0,T^max), donc pas forcément N
if jstart>0 then
    for k=0:(K-1) do
        for j=jstart:N do                            # to improve for jstart
            T_low ← spike[j] - (k+1)*del
            T_up ← spike[j] - k*del
            i ← j-1
            while ((i>0) & (i<N)) do
                if (spike[i] ∈ [T_low, T_up)) then
                    b[k+2] ← b[k+2] + 1
                else
                    if (T_low >spike[i]) then
                        break
                    end if
                end if
                i ← i-1
            end while
        end for
    end for
end if
return b
```

Algorithm 4 Full computation of b

Usage: computeb_V1(spike, del, Tmax)
Description: Full computation of b using *countpair* function
Input: spike ; array of spike times
Input: del ; δ
Input: K ; number of partitions
Output: b ; (K+1) vector (see **??**)

```
N ← length(spike)
b ← matrix(0, nrow=N, ncol=N)                                   # initialization
A ← matrix(-2, nrow=N, ncol=N)         # A is not important for b but needs to be
initialized
b[1] ← N
for k=0:(K-1) do
    b[k+2] ← countpair(spike, del, k, A)[[1]]
end for
return b
```

Algorithm 5 Partial computation of d

Usage: compute_d(spike, del, Tmax, A, k, gamma)
Description: Partial computation of A and computation of b for the k-th partition
Input: spike ; array of spike times
Input: del ; δ, delay
Input: $T^{\max}$; maximum time of the experience
Input: A ; N-by-N matrix already initialized # N is the number of spikes
Input: k ; partition index ($k \in [\![0, K-1]\!]$)
Input: γ ; constant used in the definition of d
Output: Returns d

iind ← which(A==k, arr.ind=TRUE) # iind is a matrix
z ← vector(length=length(spike), mode='numeric') # for maximum value
for i=1:N **do** # $N \geqslant 1$!
  $T_1 \leftarrow spike[i] + k * del$; $T_2 = spike[i] + (k+1) * del$
  if $(T_2 \leqslant T^{\max})$ & $(T_2 \geqslant 0)$ **then**
    z[i] ← countspike(spike, del, k, T_2)
  end if
  if $(T_1 < T^{\max})$ & $(T_1 \geqslant 0)$ **then**
    z[i] ← max(z[i], 1)
  end if
  if $(T_1 < 0)$ & $(T_2 > T^{\max})$ **then**
    z[i] ← max(z[i], 1)
  end if
end for
$d \leftarrow \sqrt{2\gamma \log(T^{\max})\, count\_conse(iind[,2])} + \gamma \log(T^{\max})/3 * \max(z)$
return d

Algorithm 6 count_conse

Usage: count_conse(x)
Description: Computation of consecutive ?
Input: x ; array of ?
Output: res ;

```
count ← 0
conse ← 1
i ← 1
n ← length(x)
if ( !n) then
    return 0
end if
compare ← x[1]
x[n+1] ← -1                                   # length of x is increased
while (i ⩽ n) do
    if (x[i+1]==compare) then
        conse ← conse + 1
    else
        count ← count + conse*(conse-1)
        conse ← 1
        compare ← x[i+1]
    end if
    i ← i+1
end while
res ← n+ count
return res
```

Algorithm 7 countspike

Usage: countspike(spike, del, k, t)
Description: Computation of the numbers of spikes such that ?
Input: spike ; array of spike times
Input: del ; δ
Input: k ; partition index
Input: t ; spike time
Output: resuk ;

```
resuk ← 0 ; N ← length(spike) ; ε ← 1e-12
for (i=1:N) do                                # N should be ⩾ 1 !
    if (spike[i]⩾ (t-(k+1)*del)) & (spike[i]< (t-k*del-ε)) then     # ε for
numerical errors
        resuk ← resuk + 1
    end if
end for
return resuk
```

Algorithm 8 Full computation of d

Usage: computed(spike, del, K)

Description: Full computation of d using *count_ conse* and *countspike* functions

Input: spike ; array of spike times

Input: del ; δ

Input: K ; number of partitions

Output: d ; (K+1) vector (see **??**)

N ← length(spike)
d ← vector(length=K+1, mode='numeric') # initialization
if missing(A) **then** # one needs to compute A - see
    A ← matrix(-2, nrow=N, ncol=N)
    for k=0:(K-1) **do**
        A ← countpair(spike, del, k, A)[[2]]
    end for
end if
d[1] ← $\sqrt{2\,\gamma\,\log T^{\max} N} + \gamma\,\log T^{\max}/3$.
for k=0:(K-1) **do**
    d[k+2] ← compute_d(spike, del, $T^{\max}$, A, k, gamma)
end for
return b

Algorithm 9 Full computation of G

Usage: computeG_V1(spike, delta, Tmax, A)
Description: Full computation of G
Input: spike ; array of spike times
Input: delta ; δ
Input: $T^{\max}$; final time
Input: A ; matrix
Output: G ; (K+1)-by-(K+1) matrix (see **??**)

```
N ← length(spike)
K ← floor(Tmax/delta)
if missing(A) then                                   # one needs to compute A - see
    A ← computeA(spike, delta, K)
end if
G ← matrix(0, nrow=K+1, ncol=K+1)                    # initialization
if K> 2 then
    for l=0:(K-2) do                                 # computation of the lower part of G
        for k=(l+1):(K-1) do
            rescase1 ← 0 ; rescase2 ← 0              # 1st case
            iind ← which(A==(k-l), arr.ind=TRUE)     # matrix indices
            lidex ← nrow(iind)
            for idex=1:lidex do                      # lidex needs to be > 0
                inter_fi ← min(spike[iind[idex, 1]] + (k + 1) ∗ δ, T^max)
                inter_de ← max(spike[iind[idex, 2]] + l ∗ δ, 0)
                length_inter ← inter_fi - inter_de
                if length_inter>0 then
                    rescase1 ← rescase1 + length_inter
                end if
            end for
            iind ← which(A==(k-l-1), arr.ind=TRUE)   # matrix indices
            lidex ← nrow(iind)
            for idex=1:lidex do                      # lidex needs to be > 0
                inter_fi ← min(spike[iind[idex, 1]] + (k + 1) ∗ δ, T^max)
                inter_de ← max(spike[iind[idex, 2]] + l ∗ δ, 0)
                length_inter ← inter_fi - inter_de
                if length_inter>0 then
                    rescase1 ← rescase1 + length_inter
                end if
            end for
```

```
            iind ← which(A==(k-l-1), arr.ind=TRUE) # 2nd case, iind contains matrix indices
            lidex ← nrow(iind)
            for idex=1:lidex do                                  # lidex needs to be > 0
                inter_fi ← min(spike[iind[idex, 2]] + (l + 1) ∗ δ, T^max)
                inter_de ← max(spike[iind[idex, 1]] + k ∗ δ, 0)
                length_inter ← inter_fi - inter_de
                if length_inter>0 then
                    rescase2 ← rescase2 + length_inter
                end if
            end for
            G[k+2,l+2] ← rescase1 + rescase2
        end for
    end for
    for k=0:(K-1) do                                             # k = l
        rescase1 ← 0
        iind ← which(A==0, arr.ind=TRUE)                         # matrix indices
        lidex ← nrow(iind)
        for idex=1:lidex do                                      # lidex needs to be > 0
            inter_fi ← min(spike[iind[idex, 1]] + (k + 1) ∗ δ, T^max)
            inter_de ← max(spike[iind[idex, 2]] + k ∗ δ, 0)
            length_inter ← inter_fi - inter_de
            if length_inter>0 then
                rescase1 ← rescase1 + length_inter
            end if
        end for
        rescase3 ← 0
        for idex=1:N do                                          # N should be > 0 !
            inter_fi ← min(spike[i] + (k + 1) ∗ δ, T^max)
            inter_de ← max(spike[i] + k ∗ δ, 0)
            length_inter ← inter_fi - inter_de
            if length_inter>0 then
                rescase3 ← rescase3 + length_inter
            end if
        end for
        G[k+2,k+2] ← 2*rescase1 + rescase3
    end for                                                      # 1st column of G
```

```
    G[1,1] ← T^max
    for k=0:(K-1) do
        rescase ← 0
        for i=1:N do                                             # N should be > 0 !
            inter_fi ← min(spike[i] + (k + 1) ∗ δ, T^max)
            inter_de ← max(spike[i] + k ∗ δ, 0)
            length_inter ← inter_fi - inter_de
            if length_inter>0 then
                rescase ← rescase + length_inter
            end if
        end for
        G[k+2,1] ← rescase
    end for
end if
lowG ← lower.tri(G)                          # to get the strictly lower triangular part
G[upper.tri(G)] ← G[lowG][order(row(G)[lowG], col(G)[lowG])]     # to make G symmetric
return G
```

Algorithm 10 Lasso shooting

Usage: countpair(T, δ, k, A)
Description: Computation of A
Input: T ; array of spike times
Input: δ ; delay
Input: k ; number of partitions
Input: A ; N-by-N matrix   # N is the number of spikes
Output: Returns A after counting

```
initialization of a
m ← 0
a^old ← a
while (|F(a) − F(a^old)| > ε) do
    for i=0,K do
        a_i ←
        m ← m + 1
    end for
end while
return a
```