poisson_processes

1 - Homogeneous Poisson processes

Overview

Let's generate several homogeneous Poisson processes and provide raster and trajectory plots for comparison. Let's further discuss afterwards.

Implementation

Goal

Our goal is to produce several simulations of homogeneous Poisson processes (a special case of continuous time Markov chains) and plot them using raster and trajectory plots, and comment on possible observations.

To create trajectory plots, we will rely on base R while raster plots will be generated with the `STAR` library, imported below:

```
#install.packages("STAR")
library(STAR)
```

Method

We recall the definition of a homogeneous Poisson process with parameter r.

- We yield a homogeneous Poisson process if the independent and identically distributed random variables $\forall k \in \mathbb{N}$, $\tilde{T}^k \sim \mathcal{E}(r)$, the exponential distribution with parameter r.

$$
\begin{aligned}
\forall k \in \mathbb{N},\, \tilde{T}^k &\sim \mathcal{E}(r) \\
\forall j \in \mathbb{N}_+,\, T^j &= \sum_{k=1}^{j} \tilde{T}^k \\
T^0 &= \tilde{T}^0 \\
N_t &= k \qquad (\text{a counting variable s.t. } t \in [T^k, T^{k+1}])
\end{aligned}
$$

Results

To proceed, we start with declaring our functions.

1. Function to generate homogeneous Poisson processes:

```
homogeneous_poisson_process_simulation <- function(
  parameter, process_length, simulation_number
){
  ### Simulates homogeneous Poisson processes simulations with a given
  ### parameter as input and a maximum process length.
  #
  # Samples an exponential distribution with parameter and yiels a matrix
  # of IID exponential random variables. It corresponds to the list of
  # \tilde{T}^k used to build a homogeneous Poisson process.
  simulations = matrix(rexp(process_length*simulation_number, parameter),
                       nrow = simulation_number, ncol = process_length)
  # Adds the 0-column
  simulations = cbind(matrix(0,nrow = simulation_number, ncol=1), simulations)
  # Performs a row-wise cumulative sum to yield the list of T^j corresponding
  # to a homogeneous Poisson process.
  simulations = t(apply(simulations, 1, cumsum))
  # Computes the counting variable.
  counter = c(0:process_length)
  # Returns the counter and simulations
  return(list("counter"=counter, "simulations"=simulations))
}
```

2. Function to generate a trajectory plot:

```
plot_trajectories <- function(poisson_processes, parameter) {
  ### Generates a trajectory plot for the first ten instances of a simulation
  ### of Poisson processes.
  #
  title = paste("Trajectories of the first 10 homogeneous\n",
                "Poisson procs. ~ Exp(",parameter,")",sep="")
  # Generates a sequence of values representing the y-axis
  counter = c(0:(dim(poisson_processes)[2]-1))
  # Plots
  plot(poisson_processes[1,],counter,type="S",col=2,lwd=1,
       xlab="Time", ylab="N (counter)", main= title,
       xlim=c(0, round(max(c(poisson_processes[1:10,])))+1/parameter))
  for (i in 2:10) {
    lines(poisson_processes[i,],counter,type="S",lwd=1, col=i+1)
  }
}
```

3. Function to generate a raster plot:

```
plot_rasters <- function(poisson_processes, parameter,restrict_xlim=F) {
  ### Generates a raster plot for the first 100 instances of a list of
  ### Poisson processes.
  #
  # Transforms the matrix of poisson process into a list
  col_length = dim(poisson_processes)[2]
  sims = t(poisson_processes[,2:col_length])
  trains = split(sims, rep(1:ncol(sims), each = nrow(sims)))
  trains = as.repeatedTrain(trains[1:100])
  # Plots the raster plot
  title = paste0("Raster plot of the first 100 homoge-\nneous",
              " Poisson procs. ~ Exp(", parameter,")")
  if (restrict_xlim){
    plot(trains, colStim = "grey80", main=title,
         ylab="Trials", yaxt="n",
         xlim=c(0,min(round(poisson_processes[,col_length])+1)))
  } else {
    plot(trains, colStim = "grey80", main=title,
         ylab="Trials", yaxt="n")
  }
}
```

Of note, we restrict by default the visualization of a given raster plot to a given x-axis window limited to the interval $[0, T^j_{min}]$ *where* T^j_{min} *is the minimum amount of time that a simulated homogeneous Poisson process reached the maximum amount of jumps allowed in the function* `homogeneous_poisson_process_simulation` *.*

The rationale is that, as Poisson processes go towards this maximum amount of steps (i.e. the parameter `process_length` in `homogeneous_poisson_process_simulation` , the spike train ticks displayed in a raster plot would start thinning out, which might lead users to erroneous conclusions if they perform visual analysis.

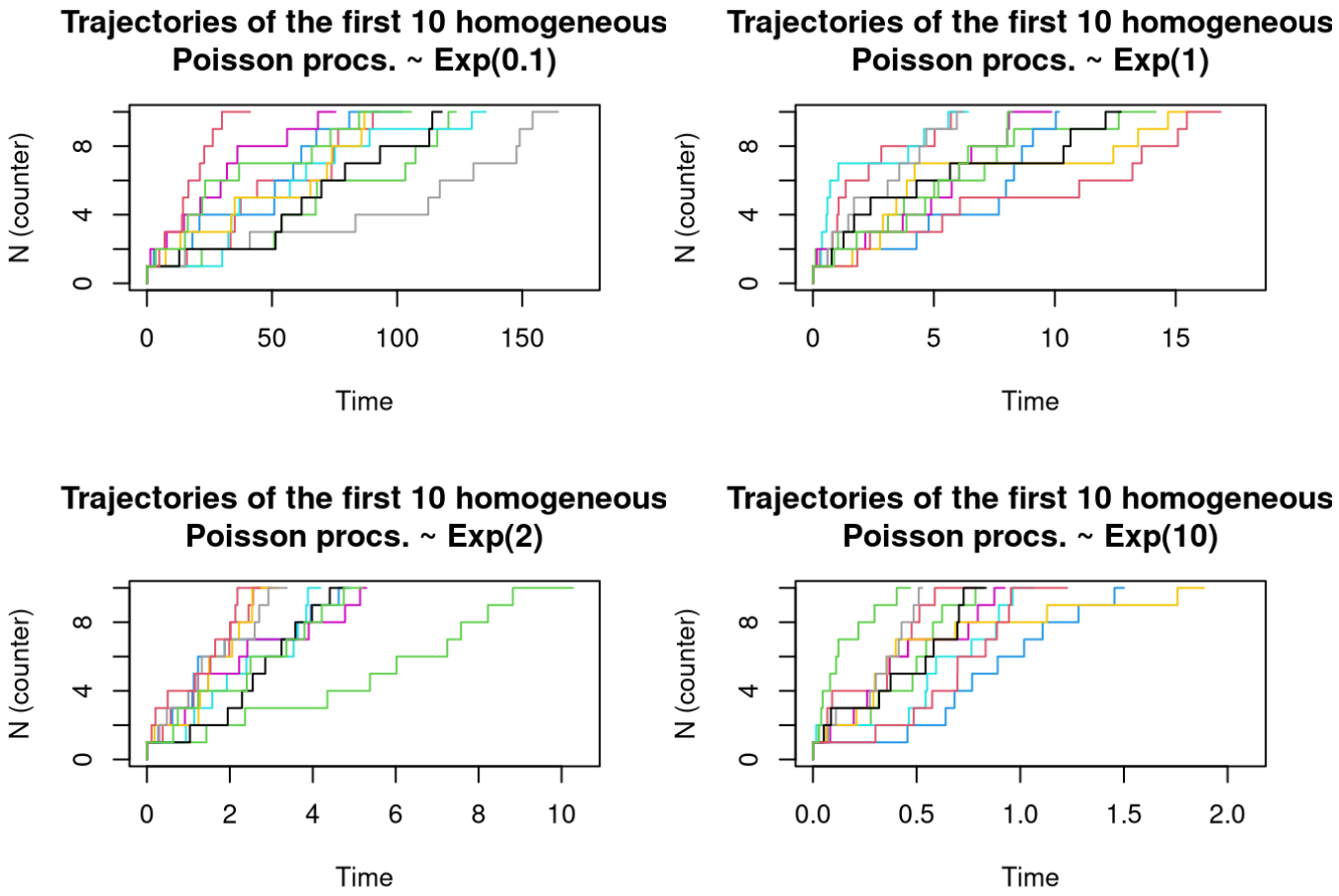
An example without this interval bound will be given in the comment section below.

Finally, we generate a given batch of simulations with a varying parameter for the underlying exponential distribution function used for the simulation process.

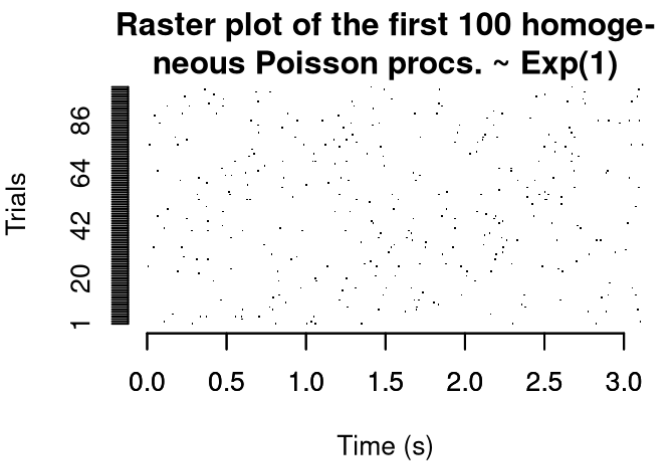
```
# Declares simulation parameters
n_simulations = 1000
simulation_length = 10 # maximum counter/number of jumps
parameters = matrix(c(0.1, 1, 2, 10)) # exp. distribution parameter range

# Simulates for each given exp. distribution parameter
simulations = apply(parameters, 1, function(x) {
  homogeneous_poisson_process_simulation(
    x,simulation_length,n_simulations
  )
})
```

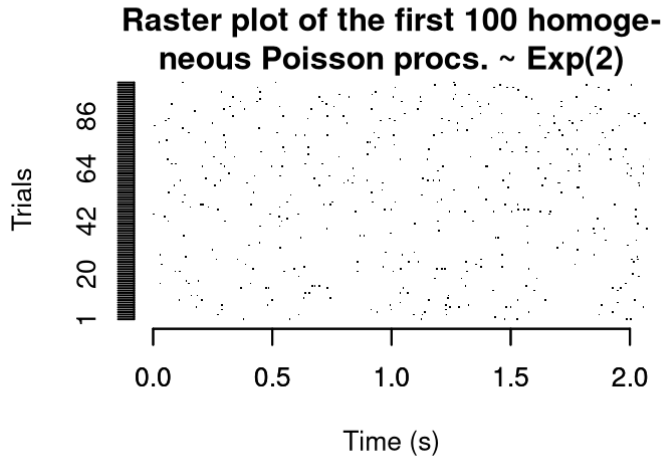
Then we display the corresponding trajectory and raster plots (respectively the first ten processes and first one hundred):

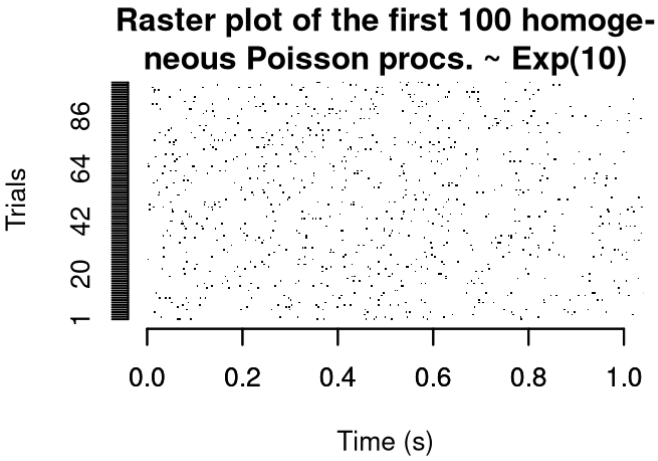
```
# Displays the resulting homogeneous Poisson processes per exp. distribution
# parameter
par(mfrow=c(2,2))
for (i in 1:length(parameters)){
  plot_trajectories(simulations[[i]]$simulations, parameters[i])
}
```




```
par(mfrow=c(2,2))
for (i in 1:length(parameters)){
  plot_rasters(simulations[[i]]$simulations, parameters[i], restrict_xlim = T)
}
```







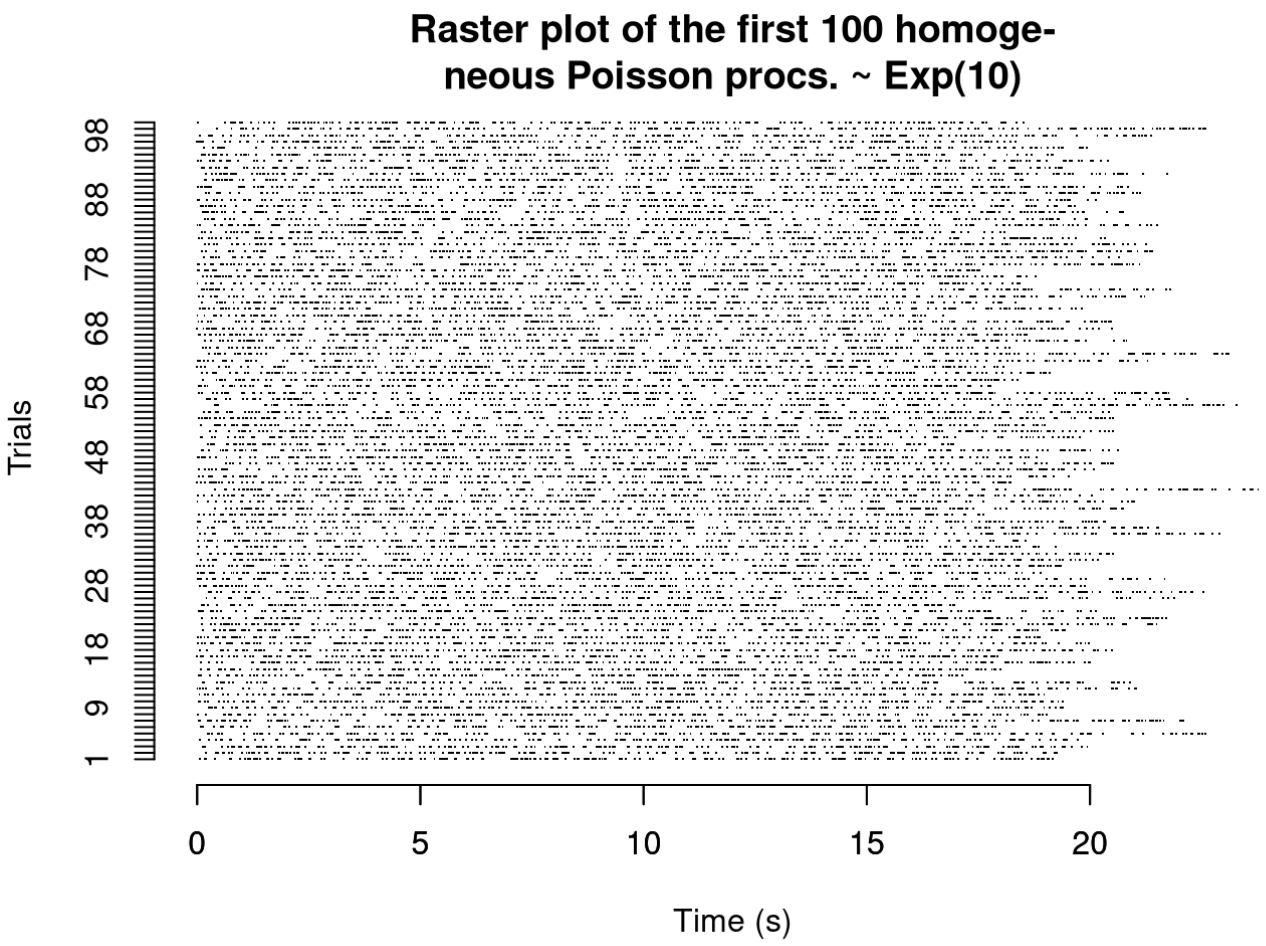
Comments

At first glance, and based on our setup, we seem to find that we do produce homogeneous processes, i.e., we cannot find emerging patterns by visually checking the plotted results.

The only determining observation we can make is that, increasing the exponential distribution's parameter r, we are shortening the length of the simulated homogeneous Poisson processes (in terms of seconds). This is expected as the parameter r is a rate.

We also recall our previous note, that we restricted our plot visualization to a given x-axis window (starting at $x = 0$ and stopping at T^j_{min}, the lowest amount of time a simulation batch needed to reach the last state jump). As previously mentioned, this was to erase the trailing end of the simulated processes where the spike trains thin out – they are reaching their maximum iteration one after the other. We can see this thinning out below as we remove the visualization's window restriction:

```
# Declares parameters
n_simulations = 100
simulation_length = 200 # corresponds to a maximum counter/number of jumps

# Simulates
simulation = homogeneous_poisson_process_simulation(
  10,simulation_length,n_simulations
)

# Plots
plot_rasters(simulation$simulations, 10, restrict_xlim = F)
```



1 - Inhomogeneous Poisson processes

Overview

Let's choose a non-constant rate function $r(t)$ and:

1. **Implement a so-called poor and robust algorithm to simulate an inhomogeneous poisson process with rate $r(t)$**
2. **Implement an algorithm with a rejection procedure to simulate an inhomogeneous poisson process with rate $r(t)$**

Note: Let's compare the two implementations in terms of time complexity afterwards.

Implementation

As a non-constant rate function $r(t)$, we decide on two rate functions:

Of note, each rate function is implemented with a normalization factor parameter that can bound the function's output within the range $[0, 1]$ for instance. This comes handy when building plots and the inhomogeneous Poisson process generation function based on the rejection algorithm.

- a sinusoidal function r such that:

$$r(t) = \frac{\cos(t + \pi) + 1}{\text{normalization\_factor}}$$

- a custom function to create a non-repeating pattern on the first 60 seconds of a given generated inhomogeneous Poisson process:

$$\forall t \in \mathbb{R},\ \text{scale} = \begin{cases} 100, & \text{if } t \in [5, 20] \text{ or } t \in [45, 50] \\ 1, & \text{otherwise} \end{cases}$$

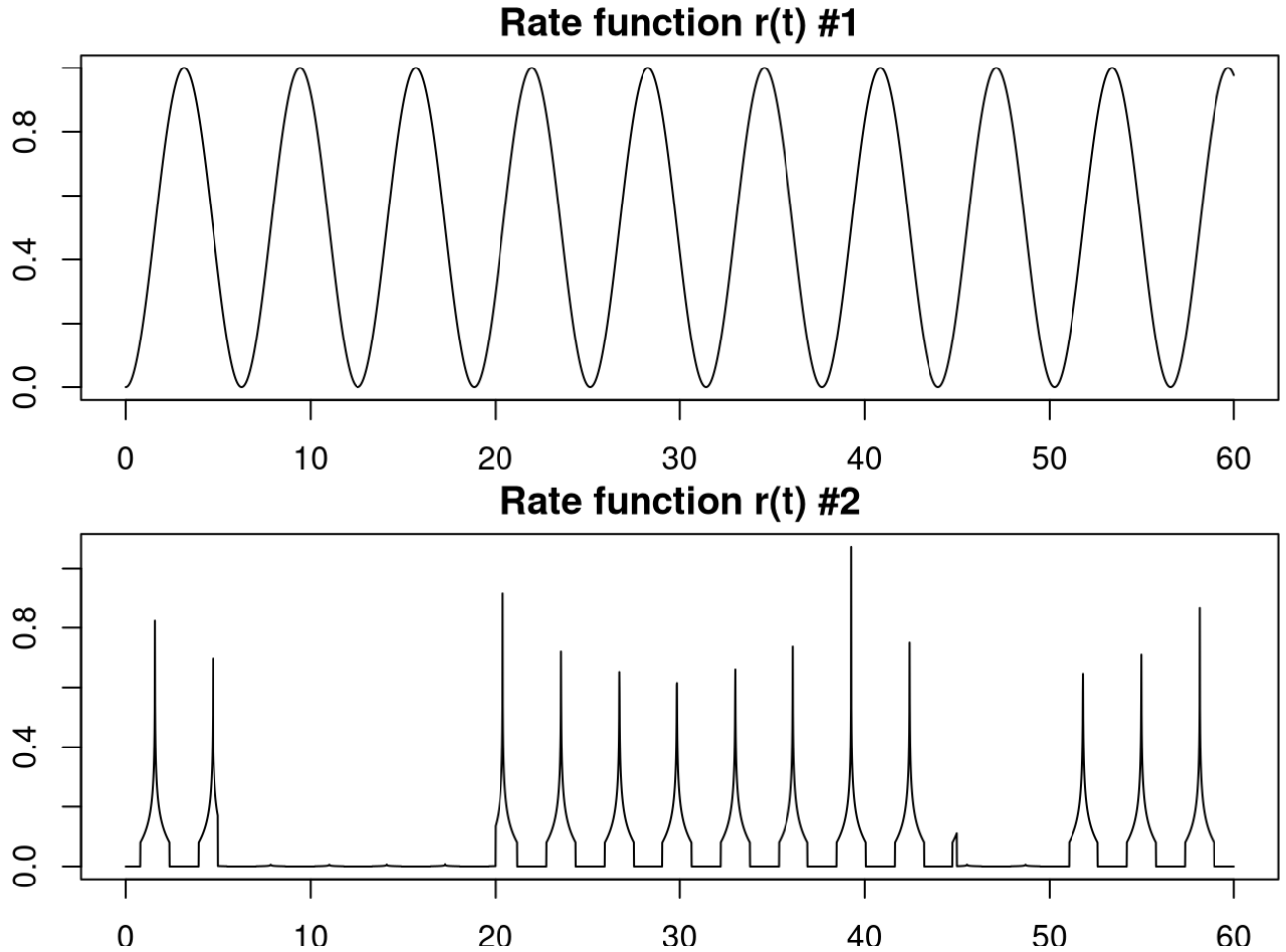
$$r_{\text{intermediary}}(t) = |tan(t)| + 1$$

$$r(t) = \begin{cases} 0, & \text{if } r_{\text{intermediary}}(t) < 2 \\ \frac{log(r_{\text{intermediary}}(t))}{\text{scale} * \text{normalization\_factor}}, & \text{otherwise} \end{cases}$$

The two functions start at 0, i.e. $r(0) = 0$ (this conserves the $T^j = 0$ at $j = 0$ condition proper to Poisson processes).

We now code the two non-constant rate functions that we will use throughout this exercise, and subsequently plot them.

```
rate_function_1 <- function(t, normalization_factor) {
  (cos(t+pi)+1)/normalization_factor
}

rate_function_2 <- function(t, normalization_factor) {
  # This function will produce some NAs when used as input to the mapper in
  # the function inhomogeneous_poor_robust_algorithm below, those will be
  # replaced with 0s.
  if ((t>5 && t<20) || (t>45 && t<50)) {scale = 100} else {scale = 1}
  ret = (abs(tan(t))+1)
  if (ret<2) {0} else ((log(ret)/scale)/normalization_factor)
}

# Declares the plot structure
par(mfrow=c(2,1), mai = c(0.35, 0.55, 0.35, 0.2))
x = matrix(seq(0, 60, 0.01))
# Plots rate function 1
plot(x, apply(x, 1, function(t) {rate_function_1(t, 2)}), type="l",
     ylab="", xlab="Time (s)", main="Rate function r(t) #1")
# Plots rate function 2 (with a normalization factor to increase or decrease
# the wanted rate)
rt2_norm = max(apply(matrix(c(1:60)),1,function(t) {rate_function_2(t, 1)}))*2
plot(x, apply(x, 1, function(t) {rate_function_2(t,rt2_norm)}), type="l",
     ylab="", xlab="Time (s)", main="Rate function r(t) #2")
```



Step 1, poor and robust algorithm implementation

Goal – The poor and robust algorithm is a method to generate inhomogeneous Poisson processes that trades robustness for time complexity. We will implement such an algorithm below and provide some resulting examples based on the previously declared rate functions.

Method – To implement a poor and robust approach, we want to simulate Bernoulli random variables such that, starting at $t = 0$ and $N_0 = 0$, with time discretization parameter δ, we have:

$$\forall j \in \mathbb{N},\ Z_j \sim \mathcal{B}(r(j\delta)\delta)$$
$$\mathbb{P}(Z_j = 1) = r(j\delta)\delta$$
$$\mathbb{P}(Z_j = 0) = 1 - r(j\delta)\delta$$
$$\hat{N}_t^\delta = \sum_{j=0}^{\lfloor \frac{t}{\delta} \rfloor}$$

We now declare our simulation function:

```
inhomogeneous_poor_robust_algorithm <- function(
  rate_function, normalization_factor,
  delta_parameter, max_seconds=10
) {
  ### Generates a list of Bernouilli random variables so as to generate
  ### a inhomogeneous Poisson process simulation
  #
  # Declares a mapper function to compute the jump location over a fixed
  # list of time steps (over a given range in seconds)
  mapper <- function(t) {
    rbinom(1,1, rate_function(t*delta_parameter,
                              normalization_factor)*delta_parameter)
  }
  # Declares the time steps sequence as a matrix
  time_steps = matrix(c(0:(max_seconds/delta_parameter)))
  # Computes the state jump locations, replacing potential NAs with 0s
  jumps = apply(time_steps, 1, mapper)
  jumps[is.na(jumps)]=0
  # Computes the inhomogeneous Poisson process counter
  N = cumsum(jumps)
  # Returns the output
  return(list("NtDelta"=N, "jumps"=jumps))
}

inhomogeneous_poor_robust_simulate <- function(
  rate_function, normalization_factor,
  delta_parameter, n_simulations, max_seconds=10
){
  ### Generates <n_simulations> of an inhomogeneous Poisson process
  ### given the Poor and Robust algorithm
  #
  # Generates a matrix/list from 1 to <n_simulations> to map over
  n_simulations = matrix(c(1:n_simulations))
  # Generates the simulations
  apply(n_simulations, 1, function(x){
    inhomogeneous_poor_robust_algorithm(
      rate_function, normalization_factor, delta_parameter, max_seconds
    )
  })
}
```

Given the simulation function, we produce a plotting function to display the simulation batch results.

```
plot_inhomogeneous_simulations <- function(
  poisson_processes, delta_parameter, max_seconds,
  rate_function, normalization_factor
) {
  ### Generates a 3-plot visualization for a batch of simulations of an
  ### inhomogeneous Poisson process simulation batch generated with a Poor
  ### and Robust algorithm
  #
  # Retrieves the data to plot and computes the corresponding rate function
  # to be plotted first
  NtDelta = c(); jumps = c()
  for (i in 1:length(poisson_processes)) {
    NtDelta = cbind(NtDelta, poisson_processes[[i]]$NtDelta)
    jumps = cbind(jumps, poisson_processes[[i]]$jumps)
  }
  x = matrix(seq(0, max_seconds, delta_parameter))
  y = apply(x, 1, function(t) {rate_function(t, normalization_factor)})
  # Declares the plot structure
  par(mfrow=c(3,1), mai = c(0.35, 0.55, 0.35, 0.1))
  # Plots the rate function
  plot(x, y, main="Rate function r", type="l", xlab="",ylab="y")
  # Plots the trajectory plot for the first ten generated processes
  counters = c(0:(dim(NtDelta)[1]-1))*delta_parameter
  plot(counters, NtDelta[,1],type="S", col=2, lwd=1,
       xlab="", ylab="N (counter)", xlim = c(0,max_seconds),
       main=paste("Trajectories of the first 10 generated inhomogeneous ",
                  "Poisson processes\n with above rate function and delta ",
                  "parameter ", delta_parameter, sep=""))
  for (i in 2:10) {lines(counters, NtDelta[,i], type="S", lwd=1, col=i+1)}
  # Retrieves the timing where a jump occurs, called an event here and
  # translates those list of times into spike trains
  events = list()
  for (i in 1:dim(jumps)[2]) {
    lst = jumps[,i]*counters
    events = append(events, list(lst[sapply(lst, function(x){x!=0})]))
  }
  events = as.repeatedTrain(events[1:100][!is.na(events[1:100])])
  # Plots the raster plot
  title = paste0("Poisson processes with delta parameter: ", delta_parameter)
  if (length(events) >= 100) {
    title = paste0("Raster plot of the first 100 inhomogeneous\n", title)
  } else {
    title = paste0("Raster plot of the generated inhomogeneous\n", title)
  }
  plot(events, colStim = "grey80", main=title, ylab="Trials", xlab="Time (s)",
       yaxt="n", xlim=c(0,max_seconds))
  # returns
  return(list("counters"=counters, "events"=events,
              "jumps"=jumps, "NtDelta"=NtDelta))
}
```

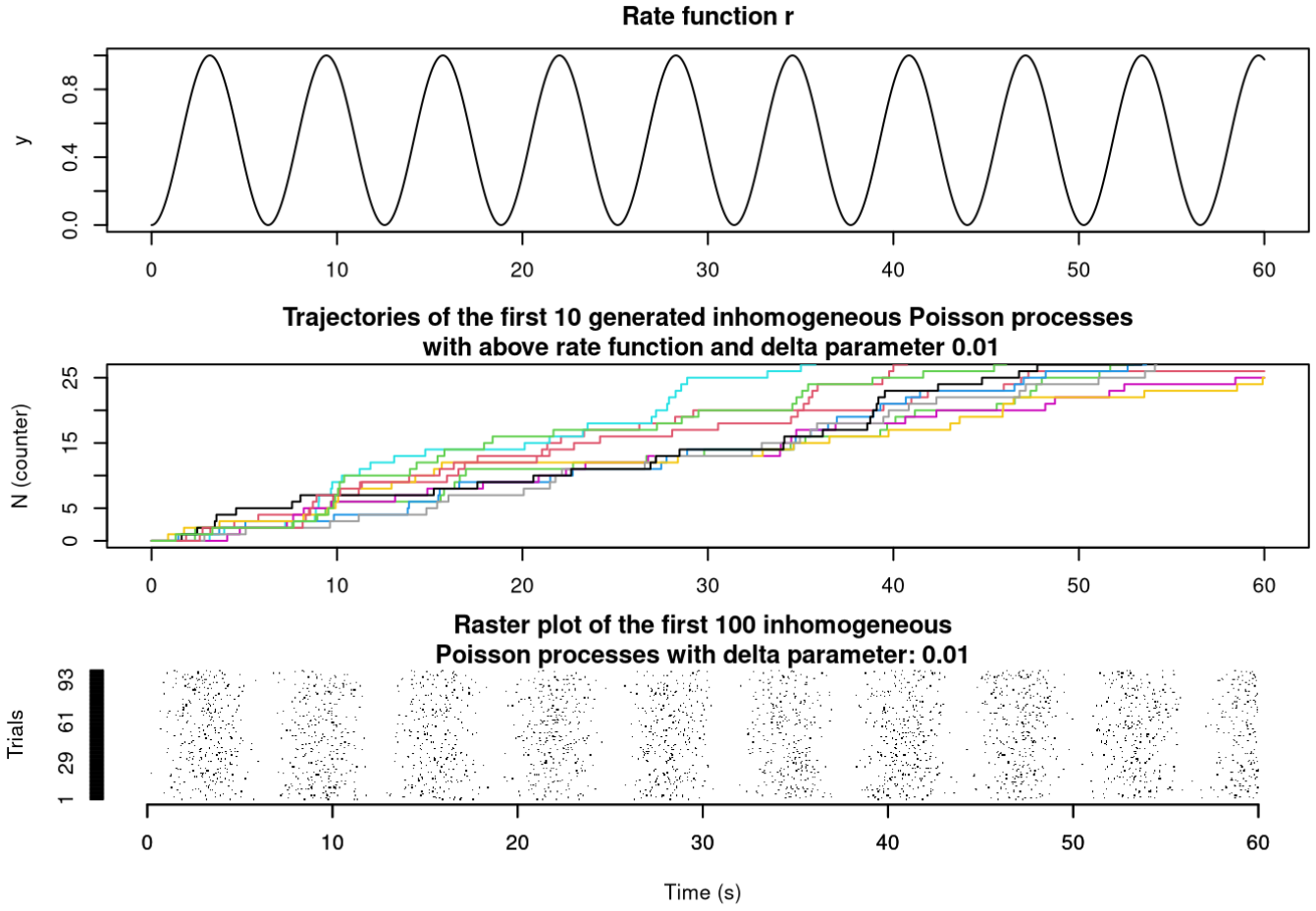
Results – We can now run a batch of simulations over the length of time of a single minute using the poor and robust algorithm.

Of note, using the rate_function_2 function outputs warnings due to the creation of NAs – those are replaced by 0 in the code.

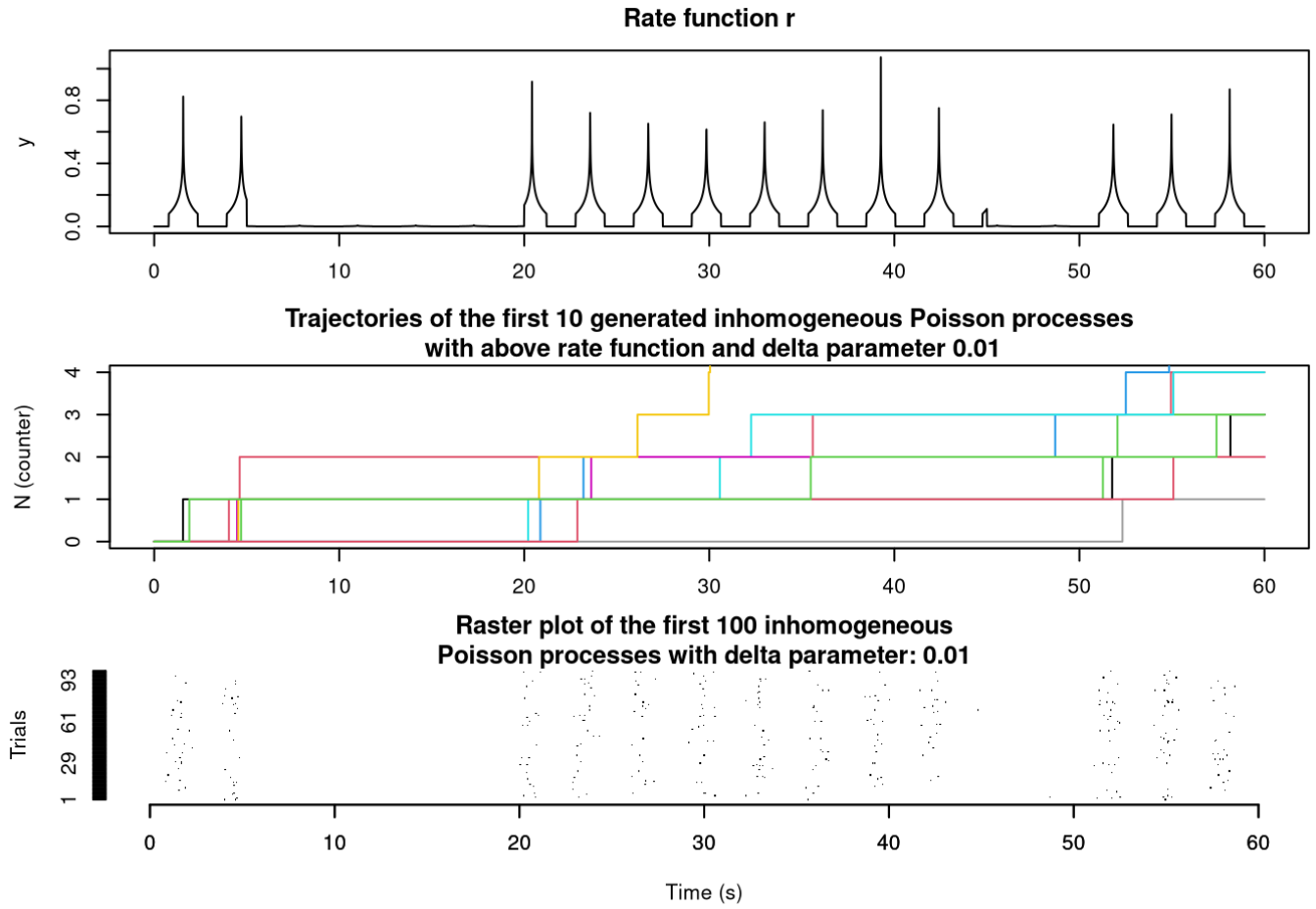
```
max_seconds = 60
delta_parameter = 0.01
n_simulations = 100

# Computes the Poor and Robust simulations with the first rate function
simulations_rf1_poor_robust = inhomogeneous_poor_robust_simulate(
  rate_function_1, 2, delta_parameter, n_simulations, max_seconds
)

# Computes the Poor and Robust simulations with the second rate function
simulations_rf2_poor_robust = inhomogeneous_poor_robust_simulate(
  rate_function_2, rt2_norm, delta_parameter, n_simulations, max_seconds
)
```

We now display the results:

```
# Prints the plots visualizing the results of the simulations generated
# with the Poor and Robust algorithm with the first rate function
sim_rf1_results = plot_inhomogeneous_simulations(
  simulations_rf1_poor_robust, delta_parameter, max_seconds, rate_function_1, 2
)
```




```
# Prints the plots visualizing the results of the simulations generated
# with the Poor and Robust algorithm with the second rate function
sim_rf2_results = plot_inhomogeneous_simulations(
  simulations_rf2_poor_robust, delta_parameter, max_seconds, rate_function_2, rt2_norm
)
```



Comments – We have based our inhomogeneous Poisson process modeling on the poor and robust algorithm Moreover, for simulation purposes, we set our generation process on a 60-second period with a delta parameter $\delta = 0.01$. As such, we can visually attest that we have an inhomogeneous generation process.

Step 2, algorithm with rejection procedure implementation

Goal – We are now looking to implement the rejection algorithm to generate inhomogeneous Poisson processes. Once generated, we will compare these processes with the ones generated by the previously declared poor and robust algorithm.

We will implement the rejection algorithm below and provide some examples based on the previously declared rate functions.

Method – To implement a rejection approach, we assume that the rate function $r(t)$ is upper-bounded by some value K such that:

$$\begin{aligned} &\forall t \geq 0 \\ &t \in [0, T_{max}] \\ &0 \leq r(t) \leq K \end{aligned}$$

From these assumptions, the rejection algorithm simulates homogeneous Poisson processes with a parameter K bounding the corresponding rate function $r(t)$ where the homogeneously-generated Poisson processes' spikes $\bar{T}$ are rejected with corresponding probability $\frac{r(\bar{T})}{K}$:

$$\mathbb{P}(\text{spike is kept, i.e. } Z = 1) = \frac{r(\bar{T})}{K}$$
$$\mathbb{P}(\text{spike is rejected, i.e. } Z = 0) = 1 - \frac{r(\bar{T})}{K}$$

We implement the simulation based on the rejection algorithm as such:

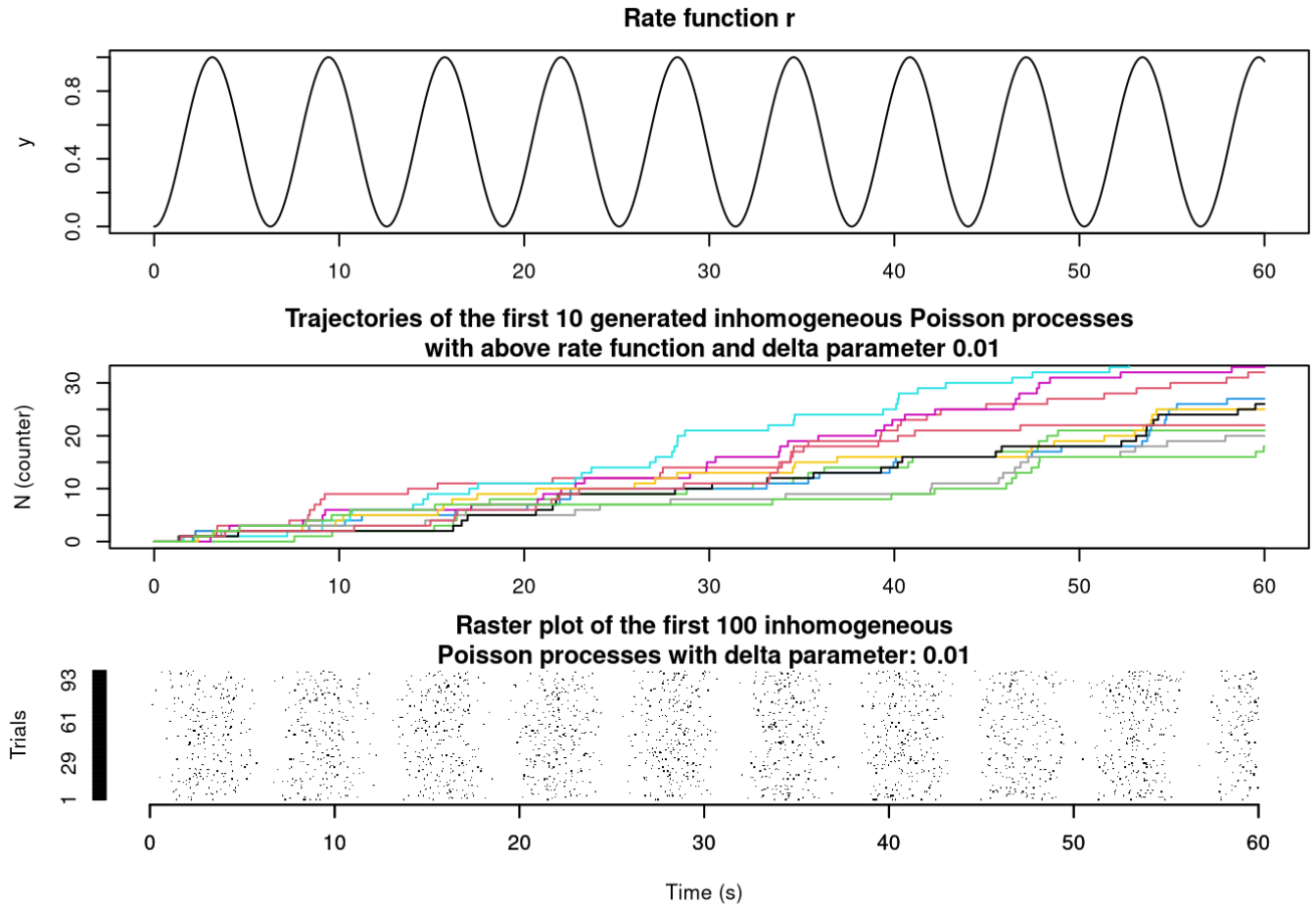
```
inhomogeneous_rejection_simulate <- function(
  rate_function, normalization_factor,
  delta_parameter, n_simulations, max_seconds=10
){
  ### Generates <n_simulations> of an inhomogeneous Poisson process
  ### given the rejection algorithm
  #
  # Generates a homogeneous Poisson process matrix over a scaled range of
  # seconds in order to capture enough spikes to be comparable to the previous
  # simulation function (poor and robust)
  scaled_max_seconds = 1.2*max_seconds
  simulations = homogeneous_poisson_process_simulation(
    1, scaled_max_seconds, n_simulations
  )
  simulations=simulations$simulations
  # Computes which simulated element to remove from the generated homogeneous
  # Poisson processes given a probability distribution derived from the rate
  # function -- a boolean mask is generated
  boolean_mask = apply(
    simulations, c(1,2),
    function(t){as.logical(rbinom(1,1,rate_function(t, normalization_factor)))}
  )
  ### Generates <n_simulations> of an inhomogeneous Poisson process
  ### given the Poor and Robust algorithm
  #
  # Generates a matrix/list from 1 to <n_simulations> to map over
  n_simulations = matrix(c(1:n_simulations))
  # Generates the simulations
  apply(n_simulations, 1, function(x){
    filtered_simulations = simulations[x,][boolean_mask[x,]]
    counters = seq(0, max_seconds, delta_parameter)
    jumps = as.numeric(counters %in% round(filtered_simulations,
                                           nchar(0.1/delta_parameter)))
    NtDelta = cumsum(jumps)
    list("NtDelta"=NtDelta, "jumps"=jumps)
  })
}
```

Results – We can now run a set of simulations over the length of time of a single minute with the rejection algorithm. Using the rate_function_2 outputs warnings due to the creation of NAs – those are replaced by 0 in the code instead.

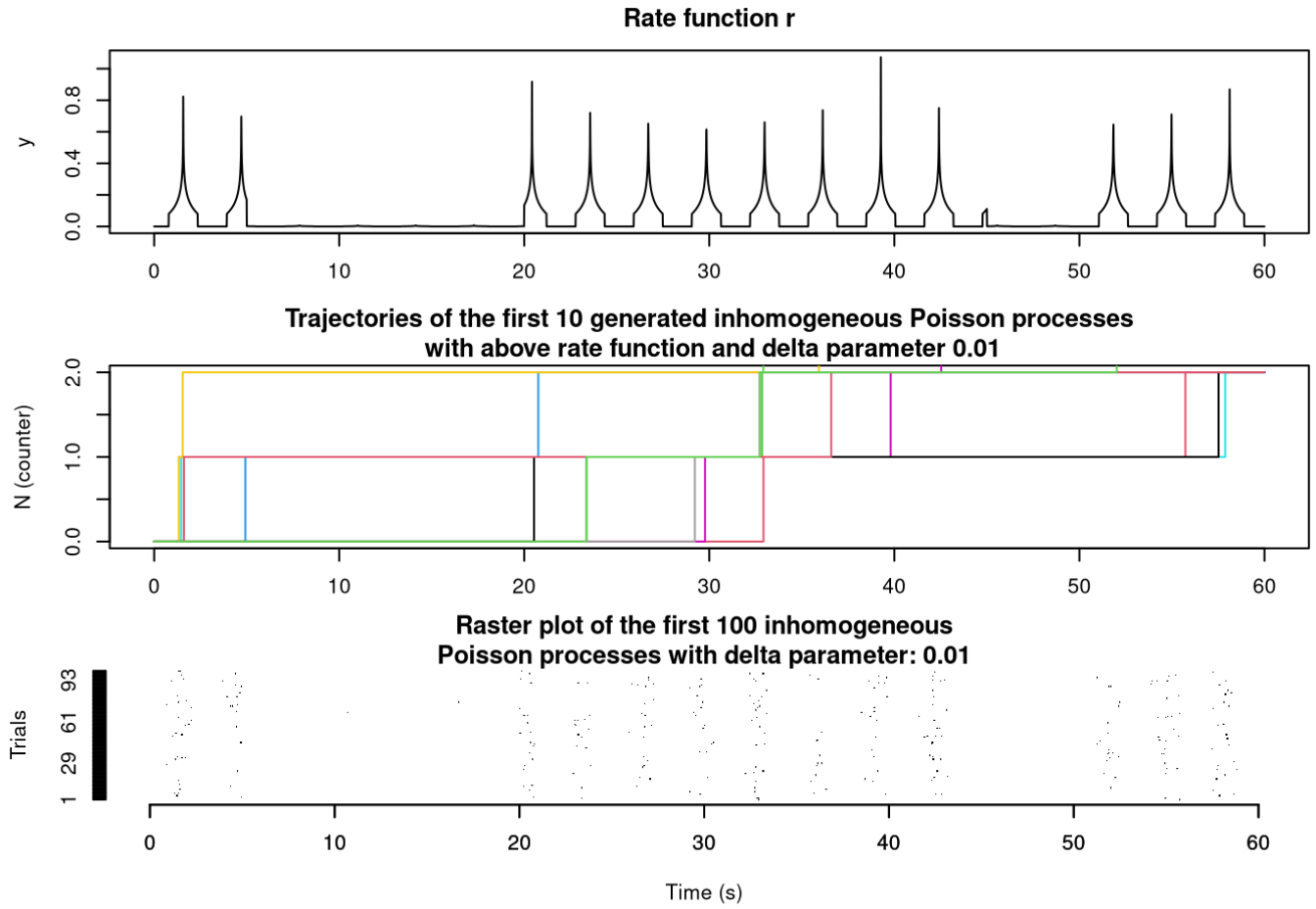
```
max_seconds = 60
delta_parameter = 0.01
n_simulations = 100

# Computes the Poor and Robust simulations with the first rate function
simulations_rf1_rejection = inhomogeneous_rejection_simulate(
  rate_function_1, 2, delta_parameter, n_simulations, max_seconds
)

# Computes the Poor and Robust simulations with the second rate function
simulations_rf2_rejection = inhomogeneous_rejection_simulate(
  rate_function_2, rt2_norm, delta_parameter, n_simulations, max_seconds
)
```

We now can display the results:

```
# Prints the plots visualizing the results of the simulations generated
# with the Rejection algorithm with the first rate function
sim_rf1_results = plot_inhomogeneous_simulations(
  simulations_rf1_rejection, delta_parameter, max_seconds, rate_function_1, 2
)
```




```
# Prints the plots visualizing the results of the simulations generated
# with the Rejection algorithm with the second rate function
sim_rf2_results = plot_inhomogeneous_simulations(
  simulations_rf2_rejection, delta_parameter, max_seconds,
  rate_function_2, rt2_norm
)
```



We also want to compare the speed of each inhomogeneous generation functions with both rate functions.

```
max_seconds = 60
delta_parameter = 0.01
n_simulations = 1000

system.time(inhomogeneous_poor_robust_simulate(
  rate_function_1, 2, delta_parameter, n_simulations, max_seconds
))
```

```
##    user  system elapsed
##  27.910   0.049  28.169
```

```
system.time(inhomogeneous_rejection_simulate(
  rate_function_1, 2, delta_parameter, n_simulations, max_seconds
))
```

```
##    user  system elapsed
##   0.569   0.017   0.590
```

```
system.time(inhomogeneous_poor_robust_simulate(
  rate_function_2, rt2_norm, delta_parameter, n_simulations, max_seconds
))
```

```
##    user  system elapsed
##  30.939   0.190  31.778
```

```
system.time(inhomogeneous_rejection_simulate(
  rate_function_2, rt2_norm, delta_parameter, n_simulations, max_seconds
))
```

```
##    user  system elapsed
##   0.682   0.023   0.726
```

Comments – We have based our inhomogeneous Poisson process modeling on the rejection algorithm. Moreover, for simulation purposes, we set our generation process on a 60-second period with a delta parameter $\delta = 0.01$. As such, we can visually attest that we have an inhomogeneous generation process.

We see that the rejection algorithm is about two order of magnitude faster than the poor and robust algorithm – which was expected given the name.

Visually, we don't see much difference except that the rejection algorithm seems to yield a lower number of jumps overall compared to the other method. As such, we might have a hint as to why it is less robust (it might exclude more points than needed to generate an fully proper inhomogeneous Poisson process).