import matplotlib.pyplot as plt

import numpy as np

import numpy.fft as fft

import math

# y, t, optw, W, C, y95b, y95u, yb = KDE(spike\_times)

# Function KDE returns an optimized kernel density estimate using a Gauss kernel function.

# Input arguments:

# spike\_times: sample data list or array.

# Output arguments:

# y: Estimated density

# t: Points at which estimation was computed.

# The same as tin if tin is provided.

# (If the sampling resolution of tin is smaller than the sampling

# resolution of the data, spike\_times, the estimation was done at

# smaller number of points than t. The results, t and y, are obtained

# by interpolating the low resolution sampling points.)

# tw: Optimal kernel bandwidth.

# Kernel bandwidths examined.

# Cost function of W.

# y95b, y95u:

# Bootstrap confidence intervals.

# yb: Bootstrap samples.

# Optimization principle:

# The optimal bandwidth is obtained as a minimizer of the formula,

# sum\_{i, j} \int k(x - x\_i) k(x - x\_j) dx - 2 sum\_{i~=j} k(x\_i - x\_j),

# where k(x) is the kernel function, according to

# Hideaki Shimazaki and Shigeru Shinomoto

# Kernel Bandwidth Optimization in Spike Rate Estimation

# Journal of Computational Neuroscience 2010

# http://dx.doi.org/10.1007/s10827-009-0180-4

# The above optimization is based on a principle of minimizing

# expected L2 loss function between the kernel estimate and an

# unknown underlying density function. An assumption is merely

# that samples are drawn from the density independently each other.

# For more information, please visit

# http://2000.jukuin.keio.ac.jp/shimazaki/res/kernel.html

# See also SSVKERNEL, SSHIST

# Hideaki Shimazaki

# http://2000.jukuin.keio.ac.jp/Shimazaki

# (New correction in version 1)

# y-axis was multiplied by the number of data, so that

# y is a time hisogram representing the density of spikes.

def KDE(spike\_times) :

spike\_times = np.array(sorted(spike\_times))

max\_value = max(spike\_times)

min\_value = min(spike\_times)

T = max\_value - min\_value

diff\_spike = np.array(sorted(np.diff(spike\_times)))

dt\_samp = diff\_spike[np.nonzero(diff\_spike)][0]

tin = np.linspace(min\_value, max\_value, min(math.ceil(T / dt\_samp), 1e3))

spike\_ab = spike\_times[np.nonzero((spike\_times >= min(tin)) \* (spike\_times <= max(tin)))]

dt = min(np.diff(tin))

y\_hist = np.histogram(spike\_ab, np.append(tin, max\_value) - dt / 2)[0]

L = len(y\_hist)

N = sum(y\_hist)

y\_hist = y\_hist / (N \* dt)

Wmin = 2 \* dt

Wmax = 1 \* (max\_value - min\_value)

tol = 1e-5

phi = (math.sqrt(5) + 1) / 2

a = ilogexp(Wmin)

b = ilogexp(Wmax)

c1 = (phi - 1) \* a + (2 - phi) \* b

c2 = (2 - phi) \* a + (phi - 1) \* b

f1 = CostFunction(y\_hist, N, logexp(c1), dt)[0]

f2 = CostFunction(y\_hist, N, logexp(c2), dt)[0]

k = 0

W = [0] \* 20

C = [0] \* 20

while(abs(b - a) > tol \* (abs(c1) + abs(c2)) and k < 20) :

if(f1 < f2) :

b = c2

c2 = c1

c1 = (phi - 1) \* a + (2 - phi) \* b

f2 = f1

f1, yh1 = CostFunction(y\_hist, N, logexp(c1), dt)

W[k] = logexp(c1)

C[k] = f1

optw = logexp(c1)

y = yh1 / sum(yh1 \* dt)

else :

a = c1

c1 = c2

c2 = (2 - phi) \* a + (phi - 1) \* b

f1 = f2

f2, yh2 = CostFunction(y\_hist, N, logexp(c2), dt)

W[k] = logexp(c2)

C[k] = f2

optw = logexp(c2)

y = yh2 / sum(yh2 \* dt)

k += 1

nbs = int(1e3)

yb = np.zeros([nbs, len(tin)])

for i in range(0, nbs) :

idx = [math.ceil(np.random.random() \* N) for i in range(0, N)]

xb = spike\_ab[idx]

y\_histb = np.histogram(xb, np.append(tin, max\_value) - dt / 2)[0] / (dt \* N)

yb\_buf = fftkernel(y\_histb, optw / dt)

yb\_buf = yb\_buf / sum(yb\_buf \* dt)

yb[i] = yb\_buf # linear extrapolation in the MATLAB version is omitted here, because we adopt only the case of taking one argument.

ybsort = sort(yb)

y95b = ybsort[math.floor(0.05 \* nbs), :]

y95u = ybsort[math.floor(0.95 \* nbs), :]

y = y \* len(spike\_times)

drawKDE(y, tin, y95b, y95u)

return y95b, y95u

def sort(mat) :

N = len(mat[0])

for i in range(0, N) :

mat[:, i] = sorted(mat[:, i])

return mat

# def logexp(x) :

# return math.log(1 + math.exp(x))

def logexp(x) :

if x < 1e2 :

return math.log(1 + math.exp(x))

if x >= 1e2 :

return x

# def ilogexp(x) :

# return math.log(math.exp(x) - 1)

def ilogexp(x) :

if x < 1e2 :

return math.log(math.exp(x) - 1)

if x >= 1e2 :

return x

def CostFunction(y\_hist, N, w, dt) :

yh = fftkernel(y\_hist, w / dt) # density

# formula for density

C = sum(yh \* yh) \* dt - 2 \* sum(yh \* y\_hist) \* dt + 2 \* 1 / (math.sqrt(2 \* math.pi) \* w \* N)

C \*= N \* N

return C, yh

def fftkernel(x, w) :

# y = fftkernel(x, w)

#

# Function `fftkernel' applies the Gauss kernel smoother to an input signal using FFT algorithm.

#

# Input argument

# x : Sample signal vector

# w : Kernel bandwidth (the standard deviation) in unit of the sampling resolution of x.

# Output argument

# y : Smoothed signal.

#

# MAY 5 / 23, 2012 Author Hideaki Shimazaki

# RIKEN Brain Science Institute

# http://2000.jukuin.keio.ac.jp/shimazaki

#

# (New correction in version 1)

# y-axis was multiplied by the number of data, so that

# y is a time histogram representing the density of spikes.

L = len(x)

Lmax = max(1,0, math.floor(L + 3.0 \* w))

n = int(2 \*\* (nextpow2(Lmax)))

X = fft.fft(x, n)

f = (np.array(range(0, n)) + 0.0) / n

f = np.r\_[-f[range(0, int(n / 2) + 1)], f[range(int(n / 2), 1, -1)]]

K = [math.exp(-0.5 \* ((w \* 2 \* math.pi \* f\_i) \*\* 2)) for f\_i in f]

y = fft.ifft(X \* K, n)

y = y[0:L]

return y

def nextpow2(n) :

if (n < 0) :

return 0

else :

m = int(math.ceil(math.log2(n)))

return m

def drawKDE(y, t, y95b, y95u) :

plt.stackplot(t, y)

plt.ylim(ymin = 0)

plt.show()