An Example of RAVE Module

Zhengjia Wang

February 21, 2018

# An Example of Rave Module

This file is an example of RAVE module. It is also a standalone file that you can use to generate formal report.

To hide code blocks, you can set knitr::opts\_chunk$set(echo = FALSE)

Let’s start to draw averaged power over frequency range.

## Step 1: Initialization.

The first step is to initialize the module. During the initialization, you should load libraries by (library(...)), source your own functions (source(...)), or load data (load(...)). You might also want to use function rave\_prepare to load some test data for easy-debug use.

The chunk should be given name (unnamed chunks will be ignored). During the run-time, initialization will be executed only once, and rave\_prepare will be ignored

library(shiny)  
library(fields)  
library(magrittr)  
library(rave)  
rave\_prepare(  
 subject = 'YAB\_Congruency\_test',  
 electrodes = 14:16,  
 epoch = 'YAB',  
 time\_range = c(1,2),  
 quiet = T # quiet = FALSE will tell you which variables are attached.  
)

## Step 3: Define inputs, and refine

Inputs are wrapped within function rave\_inputs. They should be shiny input types such as textInput, sliderInput, etc. If you are not sure about how to write these functions, for example, selectInput type ?shiny::selectInput.

**IMPORTANT**: It’s highly recommended that you enter the parameter names explicitly inside of calls. For example, func(a=b, c=d)

Here we define two inputs:

* electrode: which electrode to calculate the collapsed power
* freq\_range: which frequency range to collapse

rave\_inputs(  
 selectInput(inputId = 'electrode', label = 'Electrode',   
 choices = '', selected = NULL), # To make the call legit  
 sliderInput(inputId = 'freq\_range', label = 'Frequency Range',  
 min = 0, max = 1, value = c(0,1), step = 1, post = 'Hz')  
)

When you run the block above, you will notice that if you type ls(), there will be two variables with exactly the same name as your inputIds. Yes, you can use these variables as if they are local variables.

ls()

## [1] "electrode" "freq\_range"

However, this is not enough. For instance, your frequency range will vary if you choose different pre-processing parameters, or your electrode could be different according to data imported. rave\_inputs will only run once to set up UI elements. However, such reactive information need to be reset when new data is loaded. Therefore we need one more function rave\_updates, and this function is a little bit tricky.

We want variable electrode to be chosen from all available electrodes. For example, if we load electrodes for subject *YAB\_Congruency\_test*, then we might want variable electrode to be chosen from these three values, then we have three choices (choices=electrodes)

rave\_updates(  
 electrode = list(  
 choices = electrodes  
 ),  
 freq\_range = list(  
 min = min(frequencies),  
 max = max(frequencies),  
 value = range(frequencies)  
 )  
)

Usually this is enough for RAVE module. However, it might not be enough for you, since you might notice that the variable electrode is still NULL. Let’s make some minor changes so that we can continue the module. You can use unnamed chunks to modify the variables. These chunks will be ignored and considered as debugging code.

electrode = 14

## Step 4: Define outputs

Similar to inputs, outputs are shiny output types such as plotOutput, tableOutput, or verbatimTextOutput, etc. Other output types are also supported (such as plotly::plotlyOutPut)

rave\_outputs(  
 plotOutput(outputId = 'power\_plot\_func')  
)

The outputIds are function names that RAVE will look for during the running process. Therefore we need to define those functions.

## Step 5: Execute - Core of the module

rave\_execute({  
 power$subset(  
 Frequency = (Frequency >= freq\_range[1] & Frequency <= freq\_range[2]),  
 Electrode = (Electrode == electrode)  
 ) ->  
 dat  
})

## [ INFO ]: [32m> [39m[32m{  
## dat <- power$subset(Frequency = (Frequency >= freq\_range[1] & Frequency <= freq\_range[2]), Electrode = (Electrode == electrode))  
## }[39m

## Step 6: Register output functions

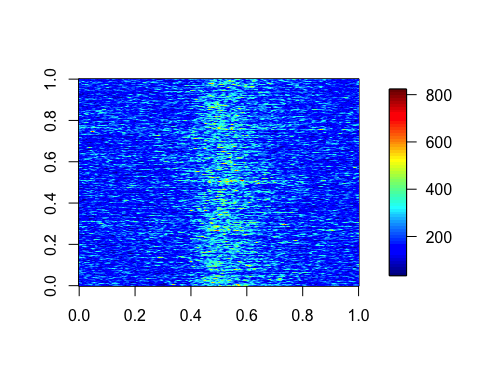
In order to have these functions called by module scheduler, you MUST have there names identical to outputIds. Also, the function usually takes no arguments (some exceptions will be mentioned in the next article).

Let’s write power\_plot\_func. Let’s assume that we have a four-mode tensor (ECoGTensor object) dat: Trial X Frequency X Time X Electrode and in which there’s only one electrode and frequency is already subset. Then we only need to collapse over frequencies.

power\_plot\_func <- function(){  
 # assume dat exists, check ?ECoGTensor for details  
 apply(dat$data, 3, rowMeans) %>%   
 t %>%   
 image.plot  
}

## Step 7: Test if your module is running properly

power\_plot\_func()



## Step 8: Add this line to make module exportable

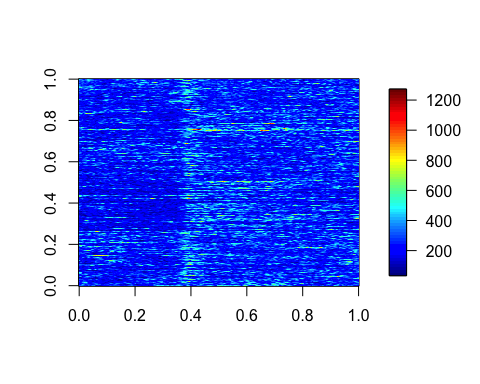
module = make\_module('CE', 'main.Rmd')  
iter\_results(  
 module, inputId = 'electrode', valueList = 52:55, outputs = 'power\_plot\_func',# 'over\_time\_plot  
 param = NULL  
) ->  
 funcs

## Electrode-52

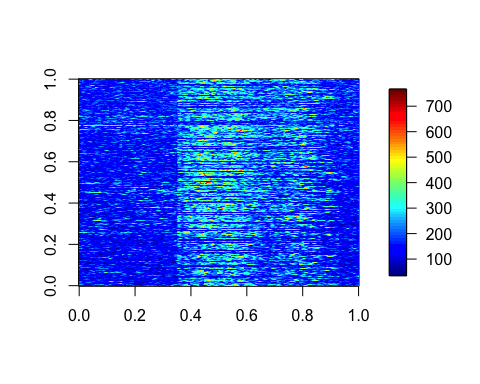
##

## Electrode-53

##



## Electrode-54



## Electrode-55

