**NeoNMM**

*A Python-based environment for simulating intracranial and transcranial electroencephalography signals with a neocortical neural mass model*

Version 1

April 2024

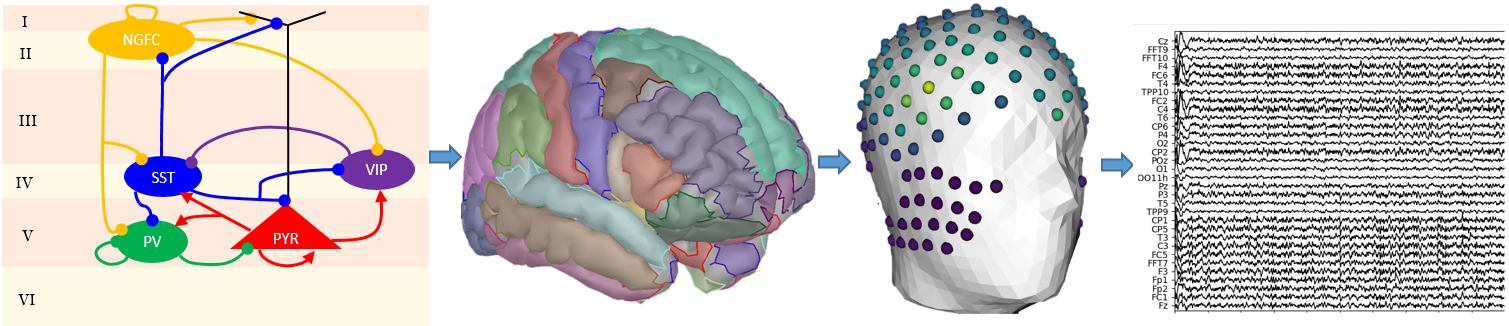


Table of contents

[Introduction 1](#_Toc163038307)

[Installation and Setup 1](#_Toc163038308)

[Getting Started 2](#_Toc163038309)

[Functionality Guide 2](#_Toc163038310)

[Model initialization 2](#_Toc163038311)

[Parameter modifications 2](#_Toc163038312)

[Stimulation 4](#_Toc163038313)

[Save 6](#_Toc163038314)

[Numerical integration 6](#_Toc163038315)

[Signal Filtering 7](#_Toc163038316)

[EEG computation 7](#_Toc163038317)

[Visualization 7](#_Toc163038318)

[Module EEG\_Viewer 8](#_Toc163038319)

[Module Spectrogram 8](#_Toc163038320)

[Module Mesh3DView 9](#_Toc163038321)

[Visualizing parameter evolution and stimulation signal 11](#_Toc163038322)

[Practical example 1: running from a previously saved simulation file 11](#_Toc163038323)

[Practical Example 2: simulating a seizure-like activity 13](#_Toc163038324)

[References 17](#_Toc163038325)

# Introduction

NeoNMM is a software package for simulating mesoscale (neuronal population) and macroscale (connected neuronal populations at the whole brain level). NeoNMM targets computational Neuroscience students, engineers, and researchers, who want to simulate EEG signals with a neurophysiologically grounded neural mass (NMM) as introduced in (1). The software package requires basic coding skills in Python.

# Installation and Setup

As the software package reassembles the Python scripts necessary for simulations and visualization. It does not require installation but a python environment with available modules given in Table 1.

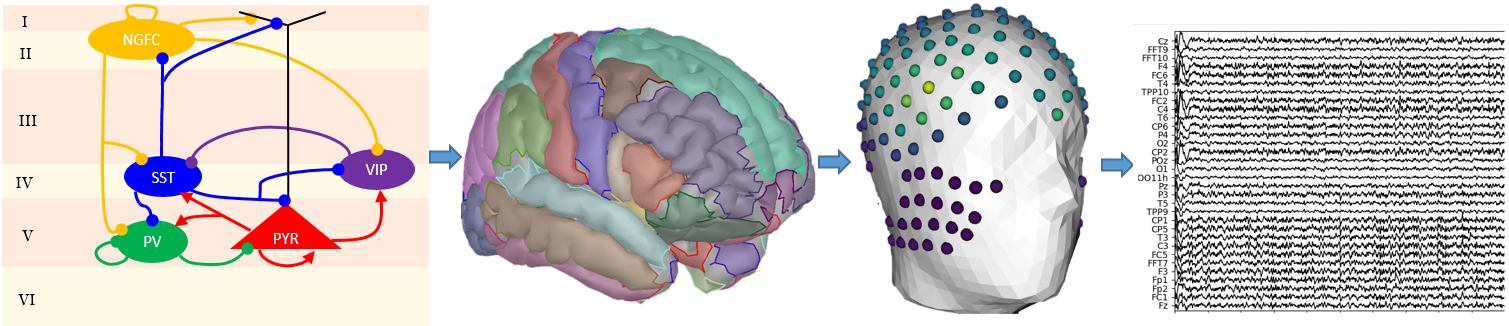
Table 1. Required modules and the least required version

|  |  |
| --- | --- |
| **Module** | **Least required version** |
| PyQt6 | 6.6.1 |
| Numpy | 1.23.5 |
| Matplotlib | 3.8.3 |
| Vtk | 9.2.6 |

# Getting Started

The package can be cloned from the GitLab link : <https://gitlab.univ-rennes1.fr/myochum/neocorticalnmm> or can be installed from the pip command : “pip install neocorticalnmm” A faire

# Functionality Guide



## Model initialization

A model is initialized with the function Cortex\_Model\_NeoNMM.Cortex(Nb\_NMM = number\_of\_NMMs) that creates a model with a number of neuronal populations equals to number\_of\_NMMs based on the model equations, variables, parameters and their default values, output signals defined in the model file Model\_NeoMNN.pywhere.

User can load a previously saved simulation file in txt format using the function LoadSimul() as,

Model, List\_Stim, List\_ParamEvol = LoadSimul(FilePath=SaveFile\_Name, Model=Model)

or create from scratch. For the latter, the model will be initialized with the default parameter values and connectivity matrices as defined in Model\_NeoMNN.py. An example code of the latter case where connectivity matrices are modified using the function set\_connectivityMat() after creating the model follows:

*# create the model*number\_of\_NMM = 2  
Model = Cortex\_Model\_NeoNMM.Cortex(Nb\_NMM=number\_of\_NMM)  
  
List\_Stim = []  
List\_ParamEvol = []  
Cm\_P\_P = np.random.rand(number\_of\_NMM, number\_of\_NMM)  
Cm\_P\_I1 = np.random.rand(number\_of\_NMM, number\_of\_NMM)  
Cm\_P\_I2 = np.random.rand(number\_of\_NMM, number\_of\_NMM)  
Cm\_P\_I3 = np.random.rand(number\_of\_NMM, number\_of\_NMM)  
Cm\_P\_I4 = np.random.rand(number\_of\_NMM, number\_of\_NMM)  
DelayMat = np.random.rand(number\_of\_NMM, number\_of\_NMM)  
Model.set\_connectivityMat([Cm\_P\_P,Cm\_P\_I1,Cm\_P\_I2,Cm\_P\_I3,Cm\_P\_I4,DelayMat])

Eventually, new model classes can be introduced by changing model equations and all dependencies under the condition of respecting the structure of the model file.

## Parameter modifications

Model parameters can be modified in a static manner, meaning that they remain constant during all simulation. As an example, the parameter A of the neural population 0 is fixed to 10 below:

*# Modify model parameters in a static manner*

Model.pop.A[0] = 10

Model parameters can vary during a simulation with the class ParamEvolClass(). For each group of parameter variation, a class should be created (e.g. myParamEvol = ParamEvolClass()) and appended to the list of parameter evolutions as List\_ParamEvol.append(myParamEvol)*.*

The software allows to vary several parameters of different neural populations at different time points using different interpolation methods available in *scipy.interpolate.interp1d* class; namely linear, nearest, nearest-up, zero, slinear, quadratic, cubic, previous, or next. For each variation, time points and parameter values at each time point should be defined.

Below, the parameter of the neural populations varies linearly at time points and values , while the parameter of the neural population varies quadratically at time points and values . Parameter variations are visualized in Figure 1 by calling function Plot\_Generate\_ParamEvol(List\_ParamEvol) (see Section Visualizing parameter evolution and stimulation signal).

*## Modify model parameters in a dynamic manner*

*# Modify the parameter A of the populations NMM = [0, 1, 2]*

myParamEvol = ParamEvolClass()

myParamEvol.NMM = [0,1,2]

myParamEvol.Name = 'A'

myParamEvol.time = [0,10,20]

myParamEvol.val  = [0,5,2]

myParamEvol.typeinterp = 'linear'

List\_ParamEvol.append(myParamEvol)

###

# *Modify the parameter G of the population NMM = [5]*

myParamEvol = ParamEvolClass()

myParamEvol.NMM = [5]

myParamEvol.typeinterp = 'quadratic'

myParamEvol.Name = 'G'

myParamEvol.time = [0,6,15]

myParamEvol.val  = [5,25,10]

List\_ParamEvol.append(myParamEvol)

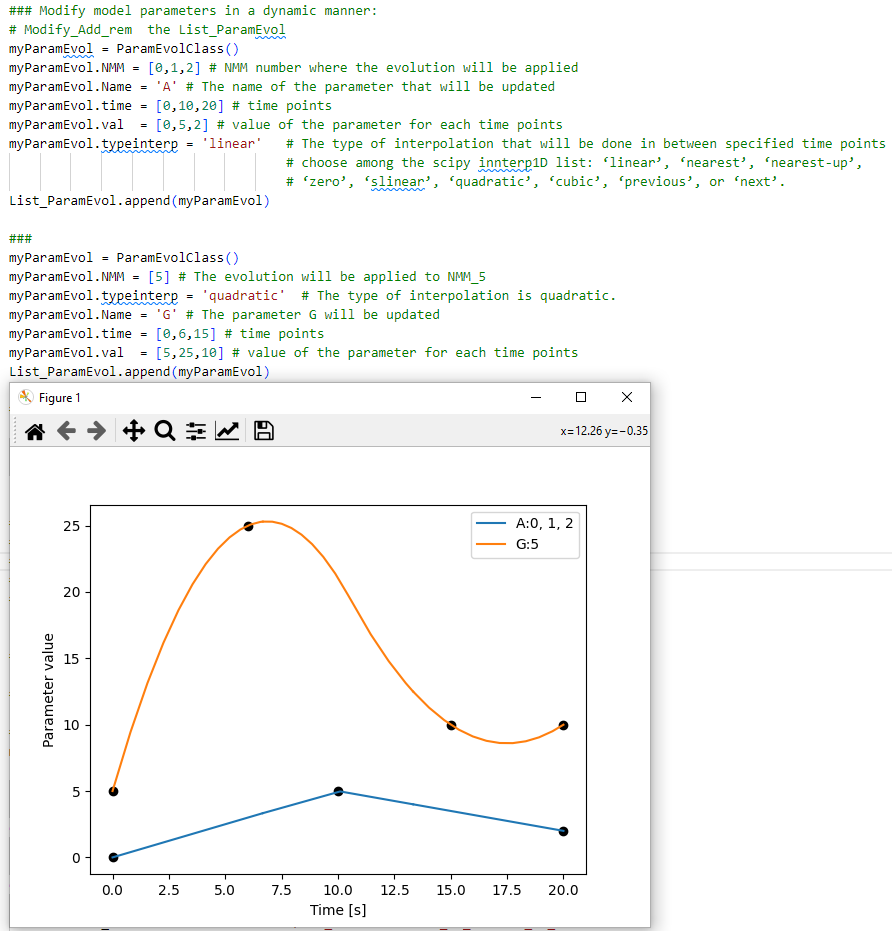


Figure 1. Dynamic parameter modification and visualization.

## Stimulation

A stimulation signal can be introduced with the class stim\_sig(). A class should be created (e.g. myStim = stim\_sig()) and appended to the stimulation signal list (e.g. List\_Stim.append(myStim)for each stimulation type. Stimulation signal can have different forms such as constant, sinusoidal, square wave, sawtooth, etc., and each of them requires a specific parameter set (Table 2). User should define one or several neural populations and the time window during which stimulation will be applied. The stimulation signal can be modulate either the mean membrane voltage (before the transfer function) or to post-synaptic potentials (after the transfer function) by setting Pre\_Post = False or Pre\_Post = True, respectively. The function Plot\_Generate\_Stim\_signal(List\_Stim, model=Model) displays the time evolution of the stimulation signals for each population in the model (see Section Visualizing parameter evolution and stimulation signal).

Table 2. Stimulation types and required parameters

|  |  |
| --- | --- |
| **Type of the stimulation** | **Required parameters and acronyms** |
| For all types | Starting time (t), end time (tend) |
| Constant | Amplitude (A) |
| Sinus, Square, Sawtooth, Triangle | Amplitude (A), frequency (f0), phase (phi) |
| Ramp | Starting amplitude (A), end amplitude (Aend) |
| Sinus Ramp | Starting amplitude (A), end amplitude (Aend), frequency (f0), phase in radian (phi) |
| Square Pulse Biphasic | Amplitude (A), frequency (f0), phase in radian (phi), pulse width (pulsewidth) |
| Chirp | Amplitude (A), starting frequency (f0), end frequency (f1), phase in radian (phi) |
| Chirp Pulse | Amplitude (A), starting frequency (f0), end frequency (f1), phase (phi), pulse width (pulsewidth) |
| Chirp Pulse Biphasic | Amplitude (A), starting frequency (f0), end frequency (f1), phase in radian (phi), pulse width (pulsewidth) |
| Square Pulse Rand | Amplitude (A), frequency (f0), phase in radian (phi), pulse width (pulsewidth) |
| Square Pulse Biphasic Rand | Amplitude (A), frequency (f0), phase in radian (phi), pulse width (pulsewidth) |

The example below demonstrates application of a “Square Pulse Biphasic” signal of amplitude A=1 arbitrary units, frequency f0 = 2Hz, pulse width 0.001 sec is applied to the neural population between , and a “Chirp” signal of amplitude A = 1 arbitrary units, frequency increasing from f0 = 1Hz to f1 = 10Hz, phase phi = 0.1 radius is applied to the neural populations between . The stimulation signals are displayed in Figure 2. Simulation signals are factorized in the model file before begin applied to excitatory and inhibitory subpopulations.

*# add a biphasic square pulse stimulation to population NMM = [0]*

myStim = stim\_sig()

myStim.kind = 'Square Pulse Biphasic' *# Kind of stimulation*

myStim.times = 0. *# Time the stimulation will start*

myStim.timee = 10. *# Time the stimulation will end*

myStim.A = 1. *# Magnitude of the stimulation*

myStim.f0 = 2. *# Frequency of the stimulation*

myStim.phi = 0.0 *# Phase of the stimulation in radian*

myStim.pulsewidth = 0.001 *# length of the stimulation pulse*

myStim.pop = [0] *# Label of the NMM where the stimulation will be applied*

myStim.Fs = 1024 *# Frequency for plotting the stimulation signals*

List\_Stim.append(myStim)

*# add a chirp type stimulation to populations NMM = [1, 2]*

myStim = stim\_sig()

myStim.kind = 'Chirp' *# Kind of stimulation*

myStim.times = 0. *# Time the stimulation will start*

myStim.timee = 20. *# Time the stimulation will end*

myStim.A = 1. *# Magnitude of the stimulation*

myStim.f0 = 1. *# Frequency of the stimulation*

myStim.f1 = 10. *# Frequency of the stimulation*

myStim.phi = 0.1 *# Phase of the stimulation in radian*

myStim.pop = [1, 2] *# Label of the NMM where the stimulation will be applied*

myStim.Fs = 1024 *# Frequency for plotting the stimulation signals*

List\_Stim.append(myStim)



Figure 2. Stimulation signals applied to three populations.

## Save

Model, parameter values and evolutions, stimulation signal can be saved in txt format using the function Save\_Simulation() that takes the file name (fileName), list containing the stimulation signal (stim), list containing the parameter evolution (evol) and model (model) as arguments:

*# Save a simulation file under ‘SaveFiles/simulation.txt”*

SaveFile\_Name = r'SaveFiles/simulation.txt'

Save\_Simulation(filename = SaveFile\_Name, stim = List\_Stim, evol = List\_ParamEvol, model = Model)

## Numerical integration

After defining the model, parameters, parameter evolutions, stimulation signals, simulation time T, and numerical integration time step Fs, numerical integration can be launched by calling the function Compute\_Time() of the Model class. The Compute\_Time() function returns local field potentials (LFP), time points (tp), firing rates of the subpopulations (FR), postsynaptic potentials (PSPs), and user-selected extra signals (ESs) as outputs:

*# Numerical integration*

LFPs, tp, Pulses, PSPs, ESs = Model.Compute\_Time(T, Fs, Stim = List\_Stim,

List\_ParamEvol = List\_ParamEvol, Pre\_Post = Pre\_Post)

If a delayed interaction matrix is included in the model, then the function Compute\_Time\_with\_delay(), which takes the same arguments as Compute\_Time(), should be used. The numerical integration follows the Euler-Maruyama method.

## Signal Filtering

NeoNMM calls the scipy.signal module for filtering the simulated signals with the function signalfilter\_EEG(), which takes the signal to be filtered, numerical integration step, filter order, lower and upper bounds as exemplified for a band pass filter below:

*# Fitering the LFP signal after simulation*

LFPs = signalfilter\_EEG(LFPs, Fs, ftype='bandpass', order=3, lowcut=1, highcut=80 )

The other available filters are low pass (‘lowpass’), high pass (‘highpass’) and band stop (‘bandstop’).

## EEG computation

NeoNMM packs 8 different lead field matrices that link the Desikan atlas of 66 cortical regions to number of EEG electrode contacts. The files corresponding to the lead field matrices and EEG electrode names are given in Table 3. Functions LeadField = LoadLeadfield(FileName = FileName\_Leadfield) and EEG\_Names, EEG\_Color = get\_electrode(filename = FileName\_Electrode) load the lead field matrix and electrode names, respectively. The forward problem is solved by multiplying the lead field matrix with the neocortical source activity (excluding the activity of the first NMM (NMM[0]), which corresponds to the thalamus).

Table 3. Correspondence between the lead field and EEG electrode names files.

|  |  |  |
| --- | --- | --- |
| **Number of EEG contacts** | **Lead field name (FileName\_Leadfield)** | **EEG electrode names (FileName\_Electrode)** |
| 21 | "Ressources/LeadField\_66x21\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_21\_DESIKAN\_66\_RL.txt" |
| 32 | "Ressources/LeadField\_66x32\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_32\_DESIKAN\_66\_RL.txt" |
| 65 | "Ressources/LeadField\_66x65\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_65\_DESIKAN\_66\_RL.txt" |
| 110 | "Ressources/LeadField\_66x110\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_110\_DESIKAN\_66\_RL.txt" |
| 131 | "Ressources/LeadField\_66x131\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_131\_DESIKAN\_66\_RL.txt" |
| 200 | "Ressources/LeadField\_66x200\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_200\_DESIKAN\_66\_RL.txt" |
| 256 | "Ressources/LeadField\_66x256\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_256\_DESIKAN\_66\_RL.txt" |
| 257 | "Ressources/LeadField\_66x257\_DESIKAN\_RL.mat" | "Ressources/EEG\_ElectrodeNames\_257\_DESIKAN\_66\_RL.txt" |

# Visualization

NeoNMM uses the PyQt6 module for the visualization. The user needs to call app = QApplication(sys.argv) before calling the visualization functions, and sys.exit(app.exec()) after all visualization function definitions.

NeoNMM incorporates different modules for different visualization purposes as explained in the subsequent subsections.

### Module EEG\_Viewer

An axis created by the module EEG\_Viewer() visualizes the time series of LFP and EEG signals, as exemplified for the LFP below:

*# Display LFP time series*

ex = EEG\_Viewer()

ex.setWindowTitle('LFPs')

ex.update(LFPs, Model.popColor, Model.popName, tp)

ex.showMaximized()

In this example, a class “EEG\_viewer” is instantiated and called “ex”. The window title is set to “LFPs” with the function setWindowTitle(). The signal (LFP), line colors (Model.popColor), signal names (Model.popName) and time points (tp) are passed to the function update(). The function showMaximized() creates a window maximized to the screen size.

The generated window (Figure 3) contains the following options:

* Gain: adjusts signal amplitude
* Window: displayed time window on one page
* Vertical spacing: vertical distance between the signals
* Linewidth: sets the line width in points

Press the Enter Key to apply any modification of the options above.

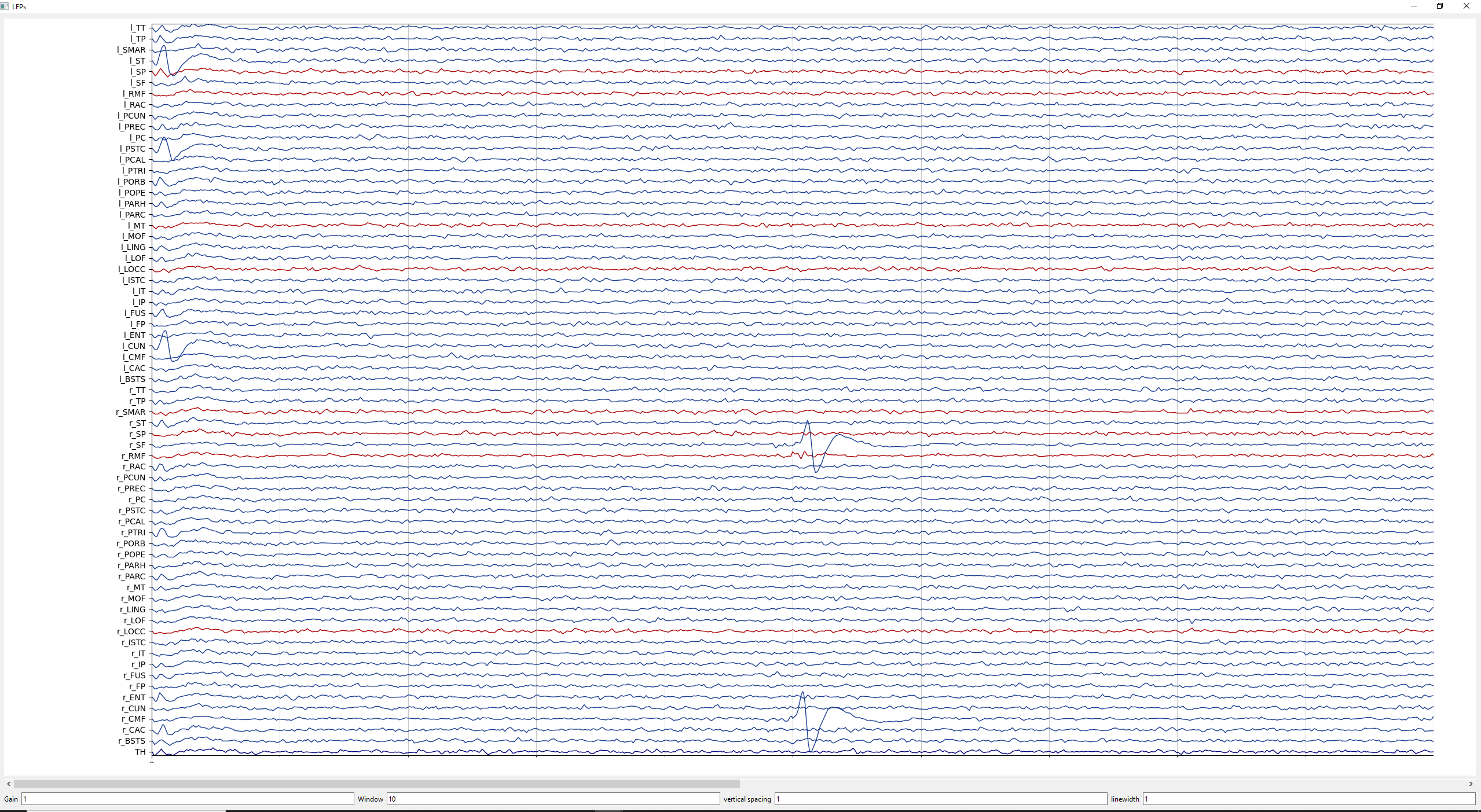


Figure 3. Window generated by EEG\_View().

### Module Spectrogram

An axis created by the module Spectrogram\_Viewer() visualizes the spectrogram of time signals, such as LFP and EEG signals, using the scipy.signal.spectrogram() function as exemplified below:

*# Display spectrogram*

ex3 = Spectrogram\_Viewer( )

ex3.setWindowTitle('Spectrogram\_Viewer')

ex3.update(LFPs=LFPs, Names = Model.popName, Fs=Fs, plot1D2D=False, cut=1 , Fmax=50, Fseg=0.5,  Colors = Model.popColor)

ex3.showMaximized()

In the example above, a class “Spectrogram\_Viewer” is instantiated and called “ex3”. The function update() takes the following arguments:

* LFPs: signals of which the spectrograms will be computed, e.g. LFPs, EEG.
* Names: signals names given as a list. This list is defined in Model.popName for LFPs and EEG\_Names for EEG signals.
* Fs: sampling frequency of the signal, which is equivalent to the time step used for the numerical integration.
* plot1D2D: if True, the spectrograms will be shown in a 1-dimentional grid; if False in a 2-dimentional grid. We recommend plot1D2D = False.
* cut: duration (in seconds) of the initial segment on the time series to be omitted.
* Fmax: maximum frequency on the y-axis.
* Fseg: length of each segment. See scipy.signal.spectrogram() documentation for details.
* Colors: signals colors given as a list. This list is defined in Model.popColor for LFPs and EEG\_Color for EEG signals.

Figure 4 shows the spectrograms of the LFP signals of 67 NMMs. Model names with respective color codes are annotated on the upper left corner of each panel.

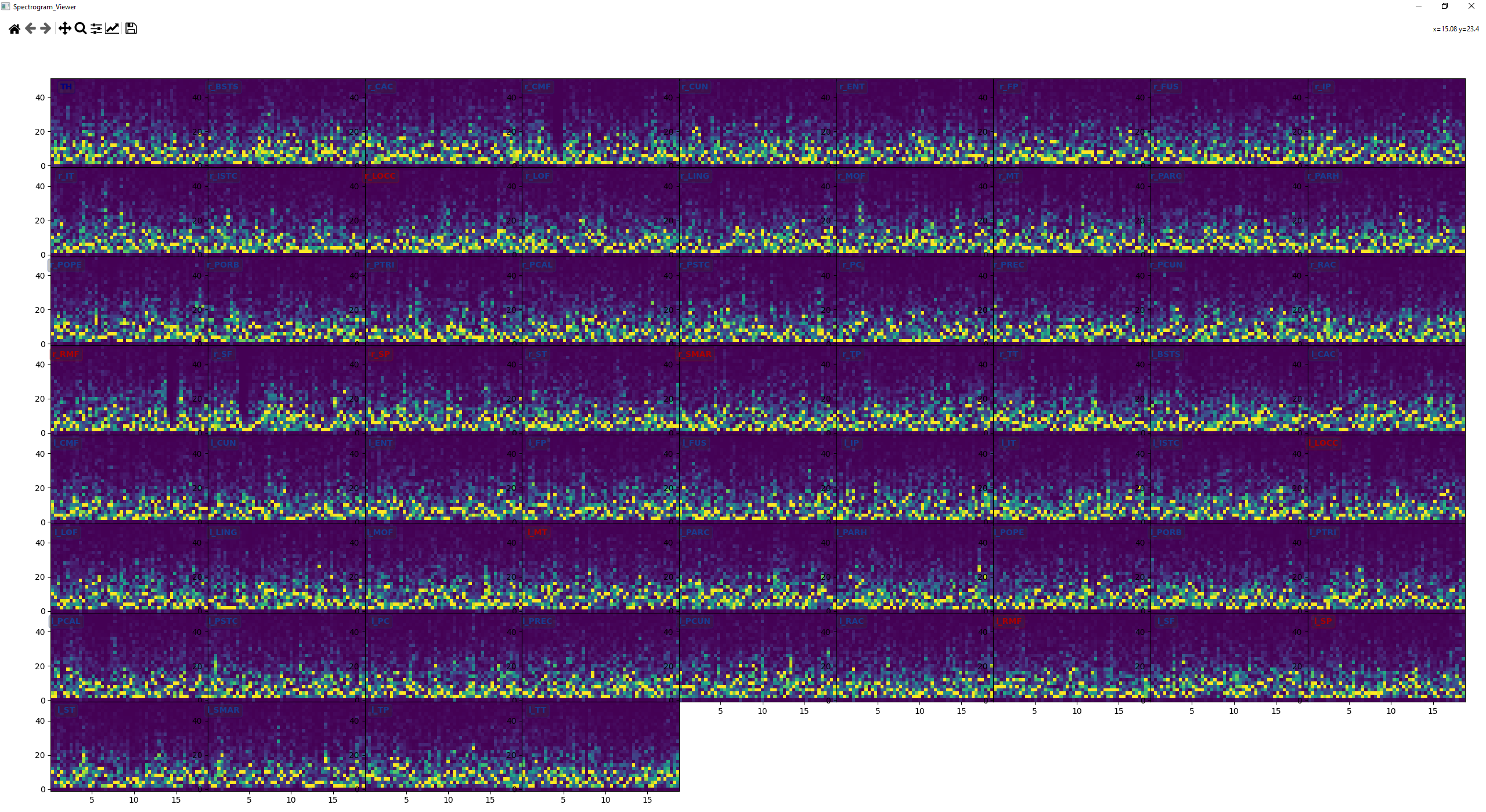


Figure 4. Window generated by Spectrogram\_Viewer() with plot1D2D = False.

### Module Mesh3DView

The module Mesh3DView() creates an interactive axis where the source activity (LFP signals) is projected on an anatomical 3-dimentional cortical model with the EEG electrode contacts are positioned around it (Figure 5). User can manipulate this 3-dimentional object to change to angle, introduce frequency or amplitude based filters, play the temporal evolution of the simulated activity on the cortical surface etc. The available options are listed below:

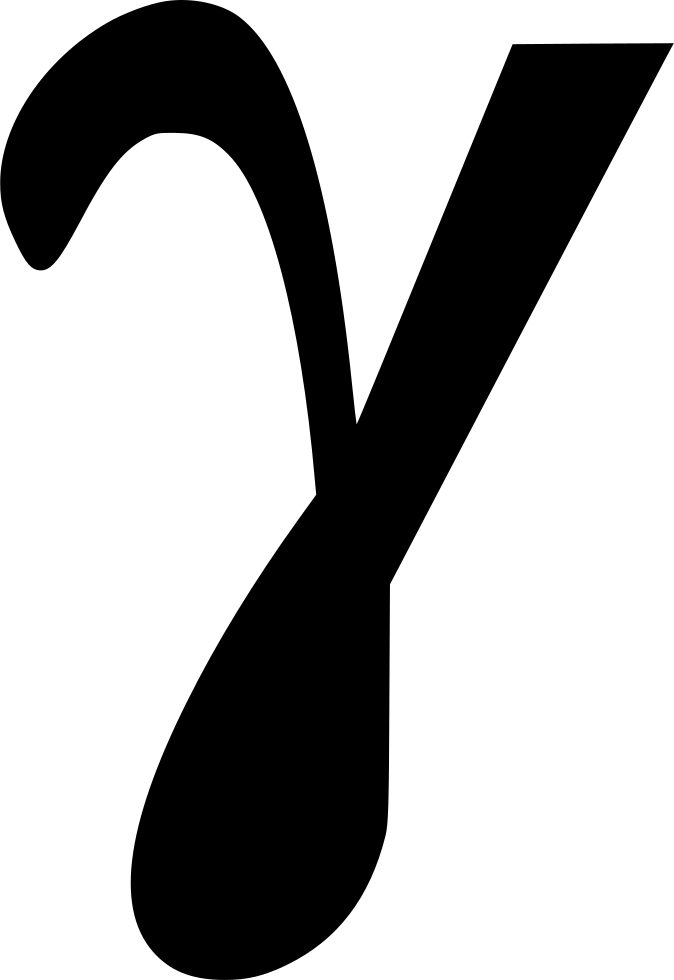
* Display region names: displays electrode names
* Analysis type: The source activity is highlighted with a color code compute from energy either in frequency domain or amplitude domain.
  + Frequency-based analysis: the energy of the LFP time series are computed within the following frequency bands:
    - Delta [0-4Hz[,
    - Theta [4-8Hz[,
    - Alpha [8-12Hz[,
    - Beta [12-30Hz[,
    - Gamma [30Hz-.

The option Without [0Hz- computes the global energy.

* + Amplitude-based analysis: Amplitude\_Mean, Amplitude\_Max, Amplitude\_RMS
* Window (s): time window during which the chosen signal analysis will be applied
* Color maps: ['viridis', 'plasma', 'inferno', 'magma', 'cividis', 'Greys\_r', 'Purples\_r', 'Blues\_r', 'Greens\_r', 'Oranges\_r', 'Reds\_r', 'YlOrBr\_r', 'YlOrRd\_r', 'OrRd\_r', 'PuRd\_r', 'RdPu\_r', 'BuPu\_r', 'GnBu\_r', 'PuBu\_r', 'YlGnBu\_r', 'PuBuGn\_r', 'BuGn\_r', 'YlGn\_r', 'binary\_r', 'gist\_yarg\_r', 'gist\_gray', 'gray', 'bone', 'pink', 'spring', 'summer', 'autumn', 'winter', 'cool', 'Wistia', 'hot', 'afmhot', 'gist\_heat', 'copper', 'PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu', 'RdYlBu', 'RdYlGn', 'Spectral', 'coolwarm', 'bwr', 'seismic']
* Brightness : set the brightness of the mesh



* Gamma: apply a gamma law on the color



* Opacity : set the transparency of the mesh



* Threshold : apply a low and high thresholding on the color



After changing any of the options regarding analysis, window and color map, the user should click on the “Update color” button.

Play/pause button displays the activity in sliding window.



Figure 5. Window generated by Mesh3DView

### Visualizing parameter evolution and stimulation signal

The function Plot\_Generate\_ParamEvol(), which takes the list of parameter evolution (e.g. List\_ParamEvol) as an argument, visualizes the user defined variation of the model parameters (see Parameter modifications) as in Figure 1.

The function Plot\_Generate\_Stim\_signal(), which takes the list of stimulation signals applied to neuronal populations and the model (e.g. List\_Stim, model=Model) as arguments, visualizes the user defined stimulation signals (see Stimulation) by creating a 2-dimentional array of axis panels as in Figure 2.

# Practical example 1: running from a previously saved simulation file

This example shows how to construct a python script for loading and running a previously saved simulation file where only one NMM was simulated and saved. We also show how to display other signals than LFPs, which are also the outputs of the numerical simulation. Since there is only on NMM, the EEG signals will not be computed. This example file does not include any parameter evolution or stimulation signal. Therefore, the functions Plot\_Generate\_ParamEvol() and Plot\_Generate\_Stim\_signal() will not be called.

* General structure of the script:

from PyQt6.QtWidgets import QApplication

from PackageSources.Model import Cortex\_Model\_NeoNMM

from PackageSources.Computation.Loading import LoadSimul, Save\_Simulation

from PackageSources.Computation.Generate\_Signal import Plot\_Generate\_ParamEvol, Plot\_Generate\_Stim\_signal

from PackageSources.Computation.Filter import signalfilter\_EEG

from PackageSources.Display.EEG\_Viewer import EEG\_Viewer

from PackageSources.Display.Spectrogram import Spectrogram\_Viewer

from PackageSources.Computation.Classes import stim\_sig, ParamEvolClass

import sys

import numpy as np

def main():

…

if \_\_name\_\_ == '\_\_main\_\_':

    main()

* Function main()
* Create the model with the number of NMMs:

Model = Cortex\_Model\_NeoNMM.Cortex(Nb\_NMM=1)

* Load the saved simulation file under « SaveFiles » directory:

SaveFile\_Name = r'SaveFiles/1NMM\_alpha.txt'

Model, List\_Stim, List\_ParamEvol = LoadSimul(FilePath=SaveFile\_Name, Model=Model)

* Define numerical integration parameters:

Fs = 1024

T = 10

* Run numerical integration:

LFPs, tp, Pulses, PSPs, ESs = Model.Compute\_Time(T, Fs, Stim = List\_Stim, List\_ParamEvol = List\_ParamEvol, Pre\_Post = False)

* If necessary, filter LFP signal for a clearer visualization:

LFPs = signalfilter\_EEG(LFPs, Fs, ftype='bandpass', order=3, lowcut=1, highcut=80 )

* Create a QT application for displaying the results:

app = QApplication(sys.argv)

* Call EEG\_Viewer() for displaying LFP:

ex0 = EEG\_Viewer()

ex0.setWindowTitle('LFPs')

ex0.update(LFPs, Model.popColor, Model.popName, tp)

ex0.showMaximized()

* Display the firing rates of each neuronal subpopulation recorded under the variable “Pulses”. Notice that signal names are exported from Model.get\_Pulse\_Names(), and color map from matplotlib.colors module.

import matplotlib.colors as colors

colors\_list = list(colors.\_colors\_full\_map.values())

ex1 = EEG\_Viewer( )

ex1.setWindowTitle('Pulses')

ex1.update(Pulses[0], colors\_list[0:Pulses.shape[1]], Model.get\_Pulse\_Names(), tp )

ex1.showMaximized()

* Sum of the post-synaptic potentials received by the subpopulation of pyramidal cells (also referred as membrane potential) is saved as an extra signal (ESs). The function EEG\_View() is called for visualization.

ex2 = EEG\_Viewer( )

ex2.setWindowTitle('Vm(t)')

ex2.update(ESs[0], Model.popColor, Model.popName, tp)

ex2.showMaximized()

* Spectrogram of LFP is computed and display using the function Spectrogram\_Viewer():

ex3 = Spectrogram\_Viewer( )

ex3.setWindowTitle('Spectrogram\_Viewer')

ex3.update(LFPs=LFPs,Names=Model.popName, Fs=Fs, plot1D2D=False, cut=1 , Fmax=50, Fseg=0.5,  Colors=Model.popColor)

ex3.showMaximized()

* The function main() ends by executing Qt application:

sys.exit(app.exec())

Running this scripts yields four windows as in Figure 6.

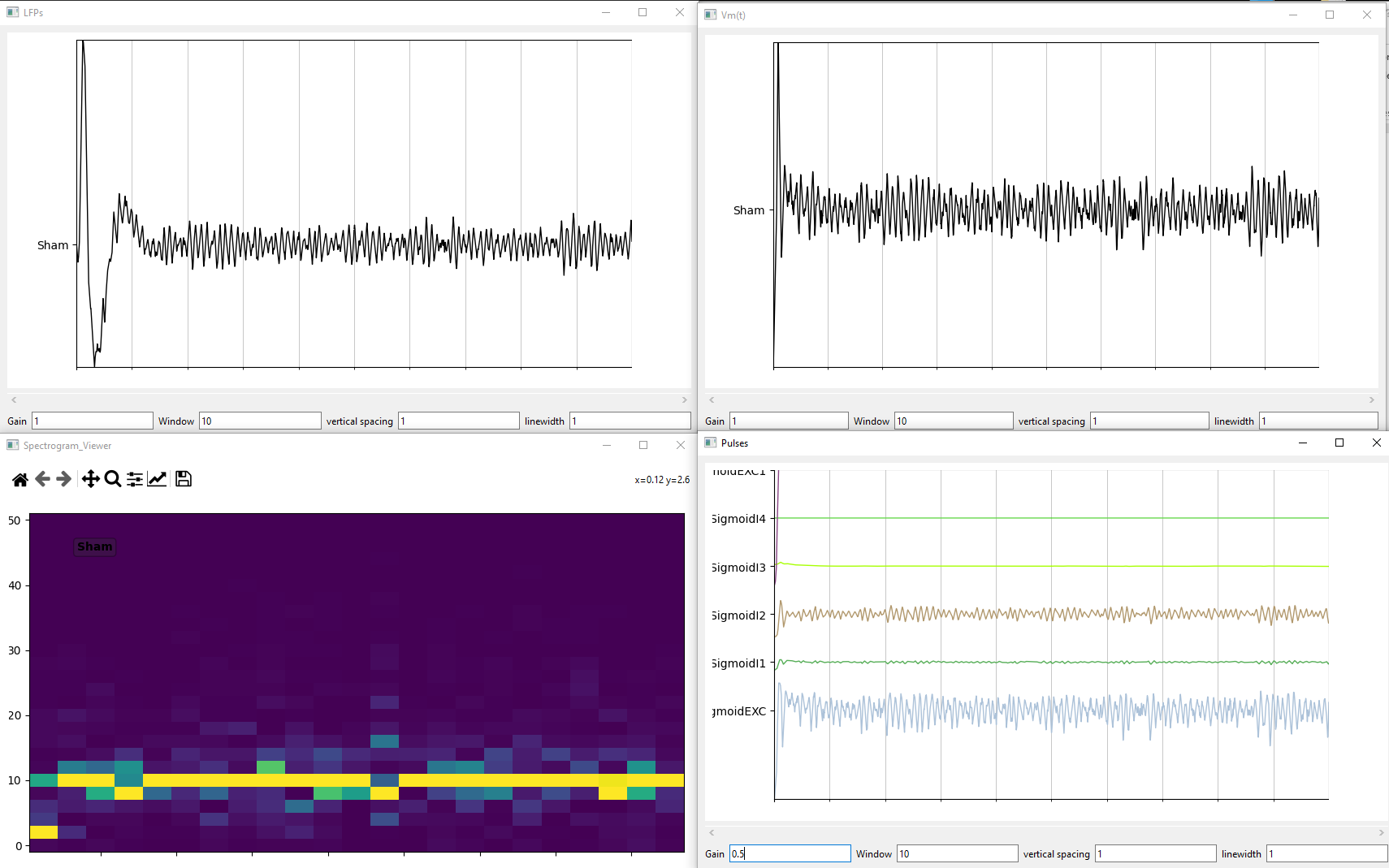


Figure 6. Displayed signals after running Practical example 1.

# Practical Example 2: simulating a seizure-like activity

In this example, we modify one of the already available scripts on the GitHub page, which is “67\_spikewave\_2.txt”. The model contains 67 regions, of which the first one corresponds to thalamus and the rest to neocortical regions associated to the Desikan atlas. The region rPREC undergoes interictal spike-wave discharges. We change three parameters of the corresponding NMM (NMM[24]) to get a transition from interictal phase with spike-wave discharges to an ictal phase followed by a seizure termination. This transition is obtained by fixing the parameter CI1CI1 to 0 during whole simulation, and by varying the parameters PCI2 and Ba dynamically.

The general structure of the script follows the one of Practical example 1: running from a previously saved simulation file. Below we detail parameter variations and forward solution. For the illustrative purposes, numerical integration accounts for the neocortical structural delay matrix. Since the subcortical regions as thalamus do not contribute to the EEG signals, the lead field matrix projects the 66 neocortical regions to EGG electrodes.

* Function main():
* Create a model with 67 regions, (1 for thalamus, 66 for neocortex):

Model = Cortex\_Model\_NeoNMM.Cortex(Nb\_NMM=67)

* Load the saved simulation file under « SaveFiles » directory:

SaveFile\_Name = r'SaveFiles/67NMM\_spikewave\_2.txt'

Model,List\_Stim, List\_ParamEvol = LoadSimul(FilePath = SaveFile\_Name, Model=Model)

* Define numerical integration parameters:

   Fs = 1024

   T = 40

* Change the value of CI1CI1 of NMM[24]:

Model.pop.CI1I1[24] = 0

* Define how the parameter PCI2 and Ba change during stimulation:

    myParamEvol = ParamEvolClass()

    myParamEvol.NMM = [24]

    myParamEvol.Name = 'CPI2'

    myParamEvol.time = [0.0, 10.0, 11.0, 15.0, 30.0]

    myParamEvol.val  = [55.0, 35.0, 10.0, 10.0, 55.0]

    myParamEvol.typeinterp = 'linear'

    List\_ParamEvol.append(myParamEvol)

    myParamEvol = ParamEvolClass()

    myParamEvol.NMM = [24]

    myParamEvol.Name = 'Ba'

    myParamEvol.time = [0.0, 10.0, 11.0, 25.0, 30.0]

    myParamEvol.val  = [19.0, 19.0, 2.0, 5.0, 19.0]

    myParamEvol.typeinterp = 'linear'

    List\_ParamEvol.append(myParamEvol)

* Save the model, parameter evolution, stimulation to a file for future use:

SaveFile\_Name = r'SaveFiles/67NMM\_interictal\_to\_ictal.txt'

Save\_Simulation(fileName=SaveFile\_Name, stim=List\_Stim, evol=List\_ParamEvol, model=Model)

* Run numerical integration by takin into account the cortical delay matrix using the function Compute\_Time\_with\_delay():

LFPs, tp, Pulses, PPSs, ESs = Model.Compute\_Time\_with\_delay(T, Fs, Stim = List\_Stim, List\_ParamEvol = List\_ParamEvol, Pre\_Post = False)

* Choose the montage (number of EEG contacts) using the variable montage, load the corresponding lead field and electrode name files.

montage = 32

if montage == 21:

FileName\_Leadfield = r"Ressources/LeadField\_66x21\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_21\_DESIKAN\_66\_RL.txt"

elif montage ==32:

FileName\_Leadfield = r"Ressources/LeadField\_66x32\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_32\_DESIKAN\_66\_RL.txt"

elif montage ==65:

        FileName\_Leadfield = r"Ressources/LeadField\_66x65\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_65\_DESIKAN\_66\_RL.txt"

elif montage ==110:

        FileName\_Leadfield = r"Ressources/LeadField\_66x110\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_110\_DESIKAN\_66\_RL.txt"

elif montage ==131:

        FileName\_Leadfield = r"Ressources/LeadField\_66x131\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_131\_DESIKAN\_66\_RL.txt"

elif montage ==200:

        FileName\_Leadfield = r"Ressources/LeadField\_66x200\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_200\_DESIKAN\_66\_RL.txt"

elif montage ==256:

        FileName\_Leadfield = r"Ressources/LeadField\_66x256\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_256\_DESIKAN\_66\_RL.txt"

elif montage ==257:

        FileName\_Leadfield = r"Ressources/LeadField\_66x257\_DESIKAN\_RL.mat"

        FileName\_Electrode = r"Ressources/EEG\_ElectrodeNames\_257\_DESIKAN\_66\_RL.txt"

LeadField = LoadLeadfield(FileName = FileName\_Leadfield)

EEG\_Names, EEG\_Color = get\_electrode(filename=FileName\_Electrode)

* Apply the lead field to the LFP signals of the neocortical sources (exclude thalamus):

EEG = np.dot(LeadField,  LFPs[1:,:])

* Create a QT application and call functions for displaying LFP, LFP spectrogram, 3D mesh, EEG and parameter evolution:

    app = QApplication(sys.argv)

*# Display LFP signals*

    ex1 = EEG\_Viewer( )

    ex1.setWindowTitle('LFPs')

    ex1.update(LFPs, Model.popColor, Model.popName, tp)

    ex1.showMaximized()

*# Display LFP spectrogramme*

    ex2 = Spectrogram\_Viewer( )

    ex2.setWindowTitle('Spectrogram\_Viewer')

    ex2.update(LFPs=LFPs, Names=Model.popName, Fs=Fs, plot1D2D=False, cut=1 , Fmax=50, Fseg=0.5, Colors=Model.popColor)

    ex2.showMaximized()

*# Display the 3D anatomical model with the source activity and electrodes*

    ex3 = Mesh\_SimpleView(parent=app, LFPs= LFPs[1:,:] , Fs=Fs, Names=Model.popName,Colors=Model.popColor, FileName = FileName\_Electrode)

    ex3.showMaximized()

*# Display EEG signals*

    ex4 = EEG\_Viewer( )

    ex4.setWindowTitle('EEG')

    ex4.update(EEG, EEG\_Color, EEG\_Names, tp)

    ex4.showMaximized()

*# Display dynamic parameters*

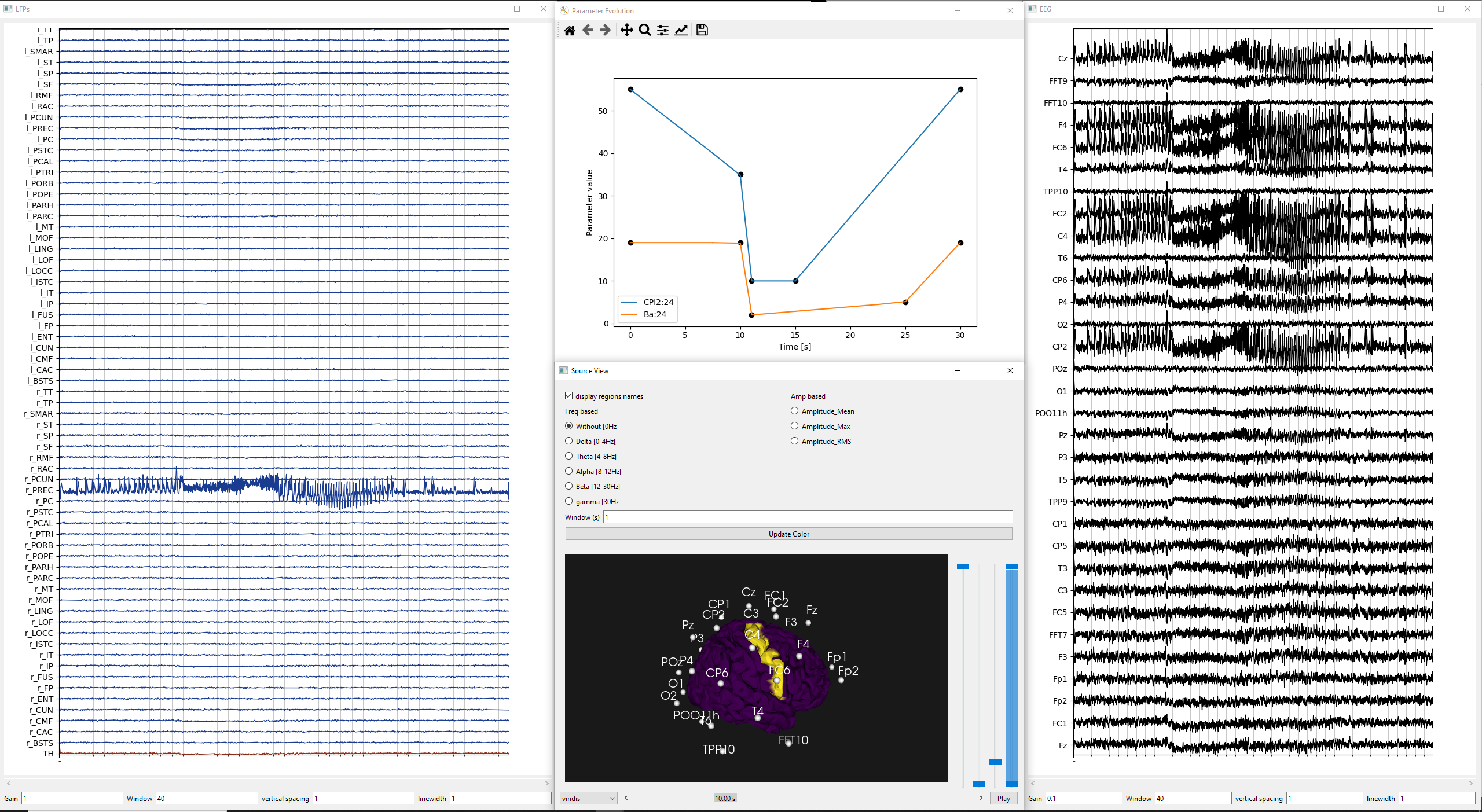
    Plot\_Generate\_ParamEvol(List\_ParamEvol)

*# Display stimulation signal if any*

    Plot\_Generate\_Stim\_signal(List\_Stim, model=Model)

    sys.exit(app.exec())

Results are displayed in Figure 7.



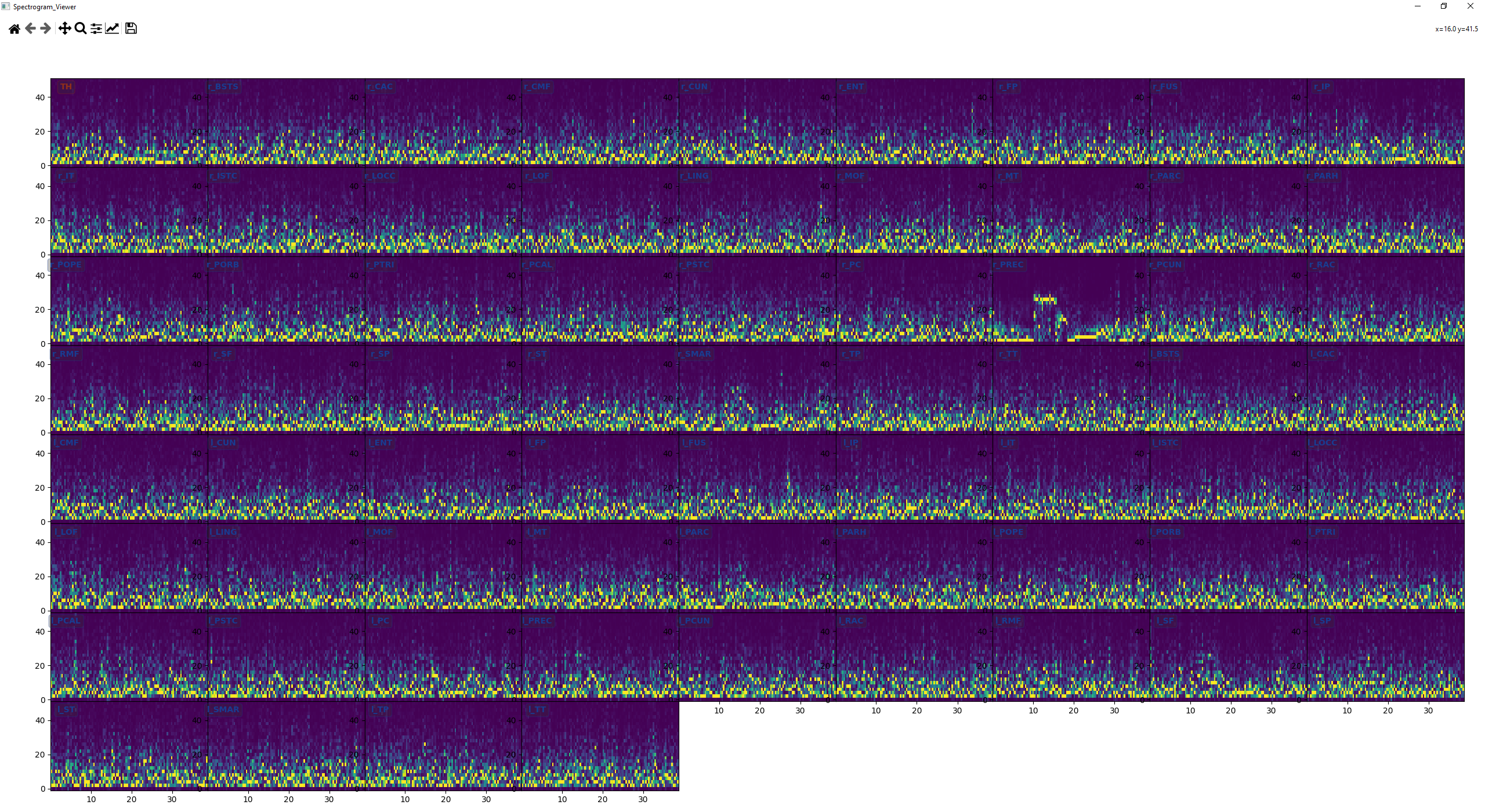


Figure 7. Displayed signals after running Practical example 2. Notice the difference in the spectrogram in the panel r-PREC.

# References

1. Köksal-Ersöz E, Lazazzera R, Yochum M, Merlet I, Makhalova J, Mercadal B, et al. Signal processing and computational modeling for interpretation of SEEG-recorded interictal epileptiform discharges in epileptogenic and non-epileptogenic zones. J Neural Eng. 2022 Sep;19(5):055005.