Hands-on instructions

First, log in to CIBR server using your favourite method; either with terminal command *ssh -Y user_name@cibr1.psy.jyu.fi*, or NoMachine. Remember that you need to be in the JYU network, either physically or via VPN. You can always find help in the CIBR support pages: https://cibr.jyu.fi/en/intranet

You can also use your own setup of MNE-Python, but then things might behave differently.

About MNE, Python and scripting

MNE-Python is a package, or toolbox, built on top of the modern Python programming language. MNE-Python internally utilizes also many other Python modules for signal processing, visualization, machine learning etc. The main idea in our approach to using MNE-Python is to make (short) code snippets or scripts, which take advantage of the high-level functions and data structures offered by the software, to carry out your data analysis tasks. Here, “high-level“ refers to the grade of abstraction, and it works to help users in performing the tasks in a clean, robust and standardised manner. Data objects, on the other hand, are strongly typed containers and, therefore, are able to contain functions (methods) needed to process those data. For example, an evoked data object is able to visualize the evoked responses it contains, without user's attention to low-level details.

MNE-Python is a continuously evolving community effort, so there are several versions available. On CIBR servers, different versions are kept in their own virtual environments. To use e.g. version 0.19, command first “conda activate mne_19”. After that, you can start the interactive Python interpreter with the command “ipython”. There are many ways to Python, you could try for example Jupyter Notebooks or your favourite IDE. Search yourself for more info to find your favorite approach. Finally, start, or *import*, the MNE module, by commanding “import mne” to the Python command prompt. If there are no errors, all is ok.

Once you get your Python interpreter started and MNE imported, it’s good to know that you can always inquire information about an object or function by adding a question mark in the end; try *e.g.* “mne?” or “mne.Evoked.save?”. Python also understands tab-completion, so just by writing “mne.viz.” and hitting the tab-key should show you some available visualization functions. Note that in Python, the module name needs to be mentioned and dot-separated to avoid ambiguous names!

During the exercises, copy the commands that you use in a separate text file. That way, you will soon have your own script to perform what you accomplish during the training.

For future reference, remember that MNE-Python supports an extensive library of tutorials and indexed help on-line at mne.tools.

About doing the exercises

In this hands-on, we will use data in directory */projects/training/data/*. In the training folder, you can find raw MEG data, averaged MEG data, pre-processed anatomical MRIs, and more. Averaged evoked data are by convention named with post-fix *-ave*, and raw data with *-raw*. There are many other extensions as well, for example *-fwd* and *-inv* for forward and inverse solutions, respectively. When producing new files during training, put them in your own directory so they can be easily removed later, and to avoid mixing with others'.

Capabilities of MNE-Python

MNE-Python provides a large selection of functions for MEG data processing. You can perform almost the entire data analysis pipeline with MNE-Python:

- browse and pre-process raw data (SSS, filtering, bad channels, ICA)
- epoching and averaging (find and process events, visualize)
- perform time-frequency analyses (including phase-locking)
- connectivity analysis
- various tools for multivariate analysis (decoding, RSA)
- some statistical tools, *e.g.* permutation cluster statistics
- integrate with FreeSurfer for forward model anatomy
- compute and visualize source models (MNE, beamforming, dipoles)

In upcoming Methods Clinics, we will touch on source modeling and decoding, at least.

Exercise 1: Load and plot MEG raw data

Let's first load some raw MEG data. We'll use the data in '/projects/training/data/multimodal_01_raw_tsss.fif'. The function used to read MEG data in FIFF format is *mne.io.read_raw_fif*. Check how this function is used, *i.e.* which arguments it needs to operate correctly, by using the question mark. It is common in MNE examples to name the raw data object as *raw*. This can be done by starting the read command with '*raw*=', which assigns the output of the reader to that variable name. The whole command is thus

raw = mne.io.read_raw_fif('/projects/training/data/multimodal_01_raw_tsss.fif')

Then the *raw structure*, however, includes not only the data itself as numbers, but a lot of other information, or metadata. You can check these by asking for the data field called *info*:

raw.info

Note that there are no parentheses in the end, because this is not a method (function operating on the data), just an attribute data field. Next, check how to plot the raw signals by writing "raw.plot?". You will see that there is a large number of arguments to the command, but all of them have a default value. So, we'll trust the default values for and just plot the raw data without any arguments.

raw.plot()

Here we have added the parentheses to indicate that we want to execute this method. Note that the *plot* part here is not a generic Python thing, but a method of the raw data class itself, programmed by some extremely kind person for this specific use case.

Sometimes it is instructive to check the data after frequency-domain filtering. Try filter the raw data utilizing the *filter*-method of the raw data object – you will need arguments *l_freq* and *h_freq* to define the high- and low-pass frequencies, as stated in the help. In the *read_raw* command, the file path was the only argument (a *positional* one). In Python, you can use also use *keyword* arguments, like *path=/myproject/data.fif*, where "path" is now the keyword to identify the argument. We now use keyword arguments for filtering. In addition, we take advantage of the handy chaining property of Python, where the output of the previous method is used as the input of the next method:

raw.copy().load_data().filter(l_freq=1, h_freq=40).plot()

Here we needed a small trick to actually load the data into memory - we also first take a copy of the original raw data to leave that unchanged. Finally, check all the methods available to the raw object by writing "raw." and hitting tab. You can also find these in the on-line reference: https://mne.tools/stable/generated/mne.io.Raw.html

Exercise 2: Epoching and averaging

Next, we'll check how the evoked responses look like. Once you have the raw data, you need to first look for the trigger-marked events and form the data epochs based on these event codes. For this, there is a ready-made function *mne.find_events()* – ask Python or the on-line API reference how to use it. Name the output as *events*, or anything else if you like. Check the program listing in the end of this exercise if you need help. The event codes are defined when the experiment is performed, here we will use event ID 2 ().

We will use *mne.Epochs* to construct the data object which would include the epoched data - the Raw data object cannot be used to store epoch data. For the object constructor command, you should specify the raw data, the events and the time limits for the epochs. Check in which order these are given or if you will use the named keywords to specify the arguments. Would you be able to add the baseline period as well? At this point, it could also be useful to plot the epochs, either as continuous data or as a raster image. Check the plotting methods offered by your Epochs object! Then, we should average the epoched data to get the evoked data, yet another type of data object:

```
evoked = epochs.average()
```

This will average over all epochs present in *epochs* data – investigate how to average over a given condition only. Hint: when selecting the epochs, what could be the difference between the statements *epochs[2]* and *epochs["2"]* ?

There are some additional things that you will need or will prove useful in many tasks, for example *reject* can be used to exclude noisy epochs, and *mne.pick_types()* can be used to pick specific channel types. Check the MNE-Python on-line examples for more info! You can try the different plotting methods available for evoked data. What can you do with *mne.viz.plot_compare_evokeds*? If you are unsure which objects you have, ask "whos" to show the workspace contents.

Here are commands that can be used in investigating evoked responses. What do they do? Extra: which stimulus modalities are the responses associated with?

```
events = mne.find_events(raw, min_duration=0.003)
reject=dict(grad=4e-10, mag=4e-12)
epochs = mne.Epochs(raw, events, event_id=[2, 5, 16], tmin=-0.2, tmax=0.5,
baseline=(None,0), reject=reject)
evoked2 = epochs["2"].average()
evoked2.plot_joint()
evoked5 = epochs["5"].average()
evoked5.plot_joint()
evoked16 = epochs["16"].average()
evoked16.plot_joint()
mne.viz.plot_compare_evokeds([evoked2, evoked5, evoked16])
```

Exercise 3: Induced responses

Next, we will use MNE-Python to study neuronal oscillations with TFR analysis. The aim is to inspect the difference between TFR of the evoked response and the TFR of the induced response in your favourite stimulus category. Basically, the difference stems from the order

of executing averaging and time-frequency decomposition. Take a moment to consider how and why does this matter? Which one is more important or interesting to you? Then, try understand the commands below, execute them, and check the outcomes. In the end, you will compute a TFR constructed after subtracting the average evoked response from each epoch prior to computing the TFR. Would this make sense? Note that the channel selection here (ch# 167) is entirely arbitrary and you can use any channels you like.

```
import numpy as np
events = mne.find_events(raw, min_duration=0.003)
raw.info["ch_names"][167]
epochs=mne.Epochs(raw, events, event_id=16, tmin=-1, tmax=2, baseline=(None,0), picks=[167])
evoked=epochs.average()
ind_tfr=mne.time_frequency.tfr_morlet(epochs, np.arange(4,40), n_cycles=4, return_itc=False,
zero_mean=True, average=True, verbose=None)
evo_tfr=mne.time_frequency.tfr_morlet(evoked, np.arange(4,40), n_cycles=4, return_itc=False,
zero_mean=True, average=True, verbose=None)
ind_tfr.plot()
evo_tfr.plot()
diff_tfr = ind_tfr - evo_tfr # How nice!
diff_tfr.plot()
```

Can you think why there are responses about 500 ms before and after the stimulus?

Exercise 4: Running scripts

There are several ways to run Python scripts made by yourself or someone else.

In ipython, you can use the "magic" command "%run":

%run -i script.py

In Linux terminal, you can just command "python" to run the script:

python script.py

Note that if you are not saving anything, you will immediately lose all results.

Some environments offer their own ways to run scripts.

You can practice this now: take the text file, where you have copied the working commands from this session. Edit it to do the parts that you would like the script to do. Then save the file in text format to your home directory - give it a name that ends in suffix ".py". Then run the script, *e.g.* in ipython. Can you see if it works?