MiCM_2021_notebook_full slides

29-37 minutes

---

exercise: create link¶

In [ ]:

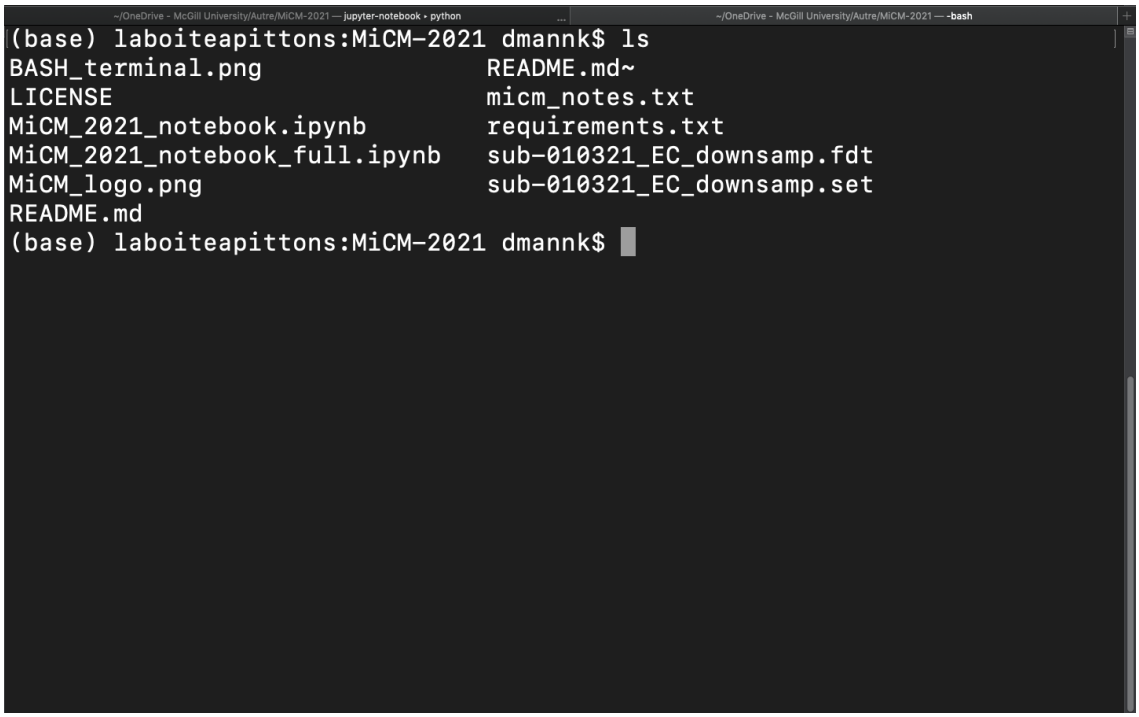
```
from IPython.display import YouTubeVideo
```

In [ ]:

```
YouTubeVideo('owSGVOov9pQ', width=800,
height=300)
```

Jupyter, an interactive Python command shell (i.e. IPython)¶

The IPython shell offers users an interactive environment, enabling easy navigation through *directories* and access to data. We can use many of the **BASH** commands which we'd usually run in a Terminal:



So, rather than listing the content of our directory by executing the *'ls'* BASH command in a Terminal, we can execute *'ls'* directly from a *code cell*:

Another BASH terminal command we could try is the *cat* command to display the content of a file:

We can display messages with *echo*:

We can create textfiles and write to them:

In [ ]:

```
!echo "Hello MiCM !" > my_message.txt
```

Let's now remove the textfile we've just created with *rm*:

We can also verify the path to our current directory using *pwd*:

Create new (sub)directory "data" and list content of current directory. Note that some BASH commands such as *mkdir* must be preceded by *!*:

Move EEG data into "data" subdirectory using *mv* and list content of current directory (can use tab completion). We will be using this EEG data as an exercise later on:

In [ ]:

```
mv sub-010321_EC_downsamp.fdt data/
```

We can also use *wildcards* for referring to files with a known structure to their filename:

Files can be copied into other directories (under new names even):

You can target which directory should have its content listed:

In [ ]:

```
cp README.md data/readme.md
```

Navigate to "data" using *cd* and display new location with *pwd*. Then, list content of current directory (which is now "data"):

Navigate back to previous directory and verify that now in proper

directory:

Whenever you forget where you had moved the data, use the terminal *find* command to retrieve its location:

Warning! Using BASH terminal commands can be tricky at times. For instance, this series of commands should work just fine:

However, running all of the same commands within a common cell usually fails:

Adding ! helps, but may nonetheless not execute the desired commands properly:

...¶

Code cells can also be used to compute mathematical expressions:

In Python, the symbol for calculating powers of a number is ** (double asterisk). We can come up with a 'funky sum':

We may want to automate our funky sum by creating a dedicated function:

In [ ]:

```
def funky_sum(x,y):

    '''
    With inputs x and y, the funky sum is
computed as x**y + y**x.

    Parameters
    ----------
    x : first term
    y : second term
    '''

    z = x**y + y**x
```

```
    return z
```

We can use *input* to query users for the value of different variables:

In [ ]:

```
x = input('1st term of funky sum: ')
y = input('2nd term of funky sum: ')

x = int(x)
y = int(y)

funky_sum(x,y)
```

We had also redacted some documentation for 'funky*sum' which we can display using the* *_doc__ attribute*:

What other attribute was created when compiling our funky_sum function?

Let's copy funky_sum and include an additional default argument:

In [ ]:

```
def funky_sum_offset(x,y,b=2):

    '''
    With inputs x and y, the funky sum is
computed as x**y + y**x + b.

    Parameters
    ----------
    x : first term
    y : second term
    b : offset parameter (b=2 by default)
    '''

    z = x**y + y**x + b
```

```
    return z
```

Let's now try the *__defaults__* attribute of our new function:

In [ ]:

```
funky_sum_offset.__defaults__
```

In addition to *attributes*, functions also have built-in subfunctions called *methods*:

The *dir* function is also useful when exploring the content of a *class*. Let's create our own class to see how this is useful:

In [ ]:

```
class funky_class():

    '''
    A class for using funky sums.
    '''

    def funky_sum(x,y):

        '''
        With inputs x and y, the funky sum is
computed as x**y + y**x.

        Parameters
        ----------
        x : first term
        y : second term
        '''

        z = x**y + y**x

        return z
```

```
    def funky_sum_offset(x,y,b=2):

        '''
        With inputs x and y, the funky sum is
computed as x**y + y**x + b.

        Parameters
        ----------
        x : first term
        y : second term
        b : offset parameter (b=2 by default)
        '''

        z = x**y + y**x + b

        return z
```

Let's now test our new class and its functions:

In [ ]:

```
print(funky_class.__doc__)
```

In [ ]:

```
funky_class.funky_sum(2,3)
```

In [ ]:

```
funky_class.funky_sum_offset(2,3)
```

Let's create some *numpy arrays* for doing funky sums. First, we'll import the *numpy* package and we'll give it the shorthand name 'np':

In [ ]:

```
x = np.arange(0,3)
```

```
print('The elements of x are:',x)
print('Said otherwise, there are {} elements in
x'.format( len(x) ))
```

In [ ]:

```
y = np.random.uniform(0,1, len(x) )
print('The {} elements of y are {}'.format(
len(y) , y ))
```

In the same way we provided numpy with the shorthand name 'np', we provide our custom 'funky_class.funky_sum_offset' function a shorthand name as well:

In [ ]:

```
fs = funky_class.funky_sum_offset
```

In [ ]:

```
z = fs(x[0],y[0])
print('The funky sum between the first element of
x (i.e. {}) and first element of y (i.e. {:.2})
is {}'.format( x[0] , y[0] , z ))
```

What happens if we input the full 'x' and 'y' arrays rather than just a single of their elements?

In [ ]:

```
z = fs(x,y)
print('The funky sum between \n the elements of x
(i.e. {}) \n and elements of y (i.e. {}) \n\t is
{}'.format( x , y , z ))
```

One important feature of Jupyter notebooks is the ability to generate plots side-by-side with your code.

Let's plot the arrays used for doing funky sums. First we'll need to import *matplotlib.pyplot*:

In [ ]:

```
import matplotlib.pyplot as plt
```

In [ ]:

```
plt.figure()

plt.plot(x, marker='o')
plt.plot(y, marker='o')
plt.plot(z, marker='o')

plt.xticks([0,1,2], ["1st element","2nd
element","3rd element"])
plt.xlabel("Elements")
plt.ylabel("Values")

plt.legend(["x","y","z"])

plt.show()
```

Let's now repeat the process of creating and plotting the x, y, z arrays for different offset values:

In [ ]:

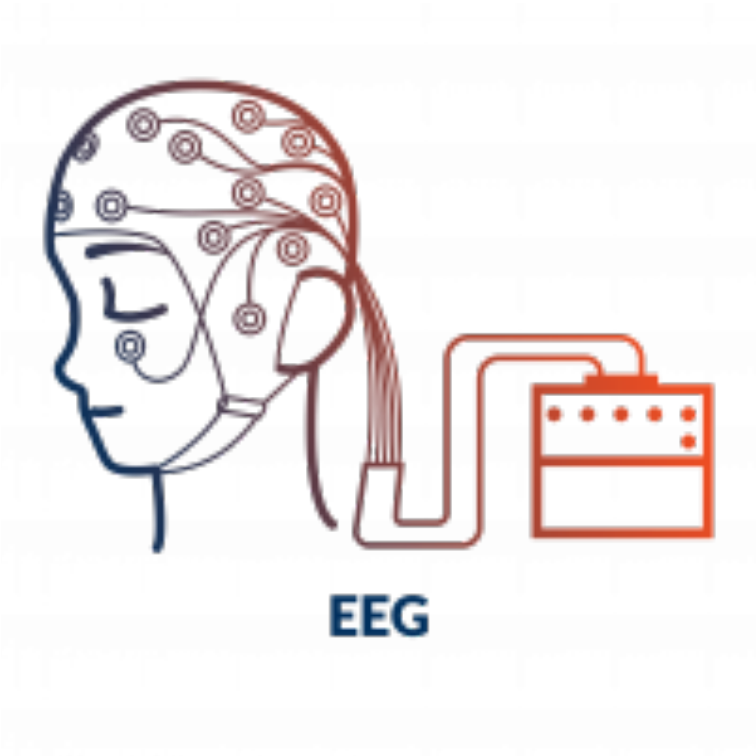
```
plt.figure()

offset = 2
z = fs(x,y, offset)
plt.subplot(2,2,1)
plt.plot(x, marker='o')
plt.plot(y, marker='o')
plt.plot(z, marker='o')
plt.xticks([])
plt.title('z = {}'.format(offset))

offset = 0
z = fs(x,y, offset)
```

```
plt.subplot(2,2,2)
plt.plot(x, marker='o')
plt.plot(y, marker='o')
plt.plot(z, marker='o')
plt.xticks([])
plt.title('z = {}'.format(offset))

offset = -2
z = fs(x,y, offset)
plt.subplot(2,2,3)
plt.plot(x, marker='o')
plt.plot(y, marker='o')
plt.plot(z, marker='o')
plt.xticks([])
plt.title('z = {}'.format(offset))

offset = -1
z = fs(x,y, offset)
plt.subplot(2,2,4)
plt.plot(x, marker='o')
plt.plot(y, marker='o')
plt.plot(z, marker='o')
plt.xticks([])
plt.title('z = {}'.format(offset))

plt.suptitle('Funky sums')

plt.show()
```

EEG data analysis¶

- ## EEG = electroencephalography EEG is an electrophysiology modality where electrodes are placed on the scalp/head to measure fluctuations in bioelectical potentials. Hence, every

electrode measures a time-varying signal.



source: https://brainvision.com/applications/eeg/

Import packages for analysing EEG data

In [ ]:

```
# Must-haves (and already imported)
import numpy as np
import matplotlib.pyplot as plt

# EEG analysis
import mne
from mne.preprocessing import ICA
from sklearn.decomposition import PCA, FastICA
from scipy.stats import zscore
from recombinator.block_bootstrap import
circular_block_bootstrap as cbb
from tensorly.tenalg import khatri_rao as kr
```

Import EEG data using [MNE Python](#) analysis package. We will be using open-source data which has been reported in:

[Babayan, A., Erbey, M., Kumral, D. et al. *A mind-brain-body dataset*](#)

of MRI, EEG, cognition, emotion, and peripheral physiology in young and old adults. Sci Data 6, 180308 (2019).

In [ ]:

```
eeg = mne.io.read_raw_eeglab("data/sub-010321_EC_downsamp.set")

eeg.annotations.delete( np.arange( len(eeg.annotations.description) ) )  # remove annotations, not important
```

Our 'eeg' is an instantiation of the class 'RawEEGLAB', which makes 'eeg' an *object*.

Objects usually already have built-in methods:

We can display useful information about the data using the *info* attribute:

For instance, we can plot the data stored in 'eeg' by calling its *plot* method:

We can also display the documentation of the plot method to see what are the plotting options:

Let's isolate the default value for the 'n_channels' argument:

Let's plot our EEG data again, this time changing the number of EEG channels on display:

At this point, our plots are not interactive. We can change the backend of matplotlib to render interactive plots using a *magic command*.

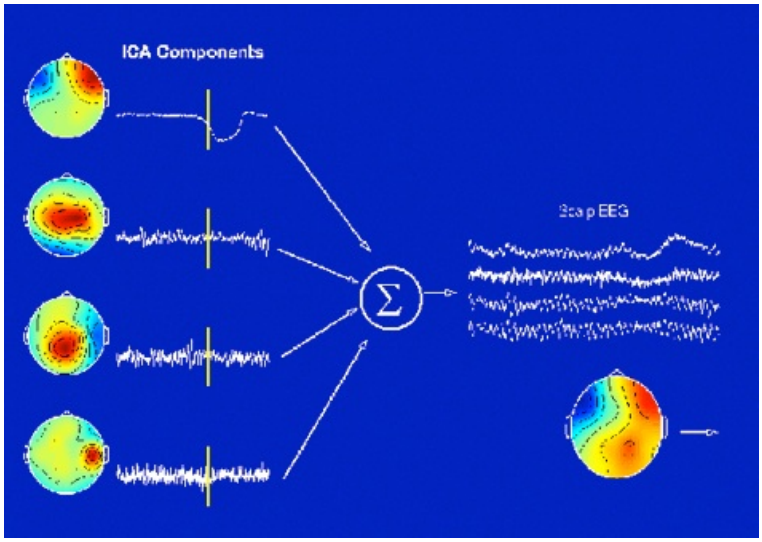
We can also display how are the EEG electrodes positioned on the head:

In [ ]:

```
eeg.plot_sensors(show_names=True)
```

From the time-series plots, it seems that many time-series are

alike. Perhaps there's a way to find patterns and have a more succinct representation of our data. One way to find these patterns is to use **Independent Components Analysis** (ICA) on our EEG data.



source: https://sccn.ucsd.edu/~jung/Site/EEG_artifact_removal.html

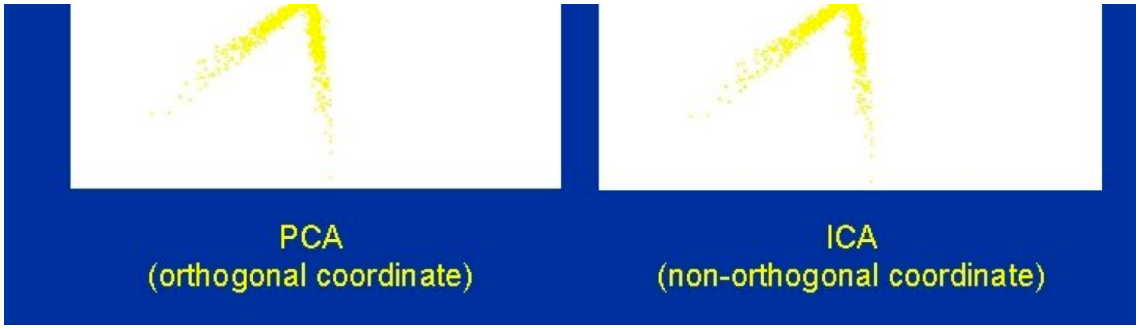
In [ ]:

```
num_comps = 15  # choose number of patterns, or
"components"

ica = ICA(n_components=num_comps,
random_state=97)
ica.fit(eeg)
```

The 'ica' object also contains its own built-in method for plotting the time-series of ICA components:

Let's also plot the *spatial topographies* of the ICA components:

Now, let's also try **Principal Components Analysis** (PCA). We will need to extract the time-series from our 'eeg' object as MNE does not yet support a built-in method for PCA as it does for ICA.



source: https://slidetodoc.com/factor-and-component-analysis-esp-principal-component-analysis/

In [ ]:

```
eeg_ts = eeg.get_data().T
```

Let's display the shape of the EEG data using the built-in *shape* method for Numpy arrays:

In [ ]:

```
print("Shape of EEG data: ", eeg_ts.shape)
```

In [ ]:

```
num_comps = 15  # choose number of patterns, or
"components"

pca = PCA(n_components=num_comps)
pca_data_ts = pca.fit_transform(eeg_ts)
```

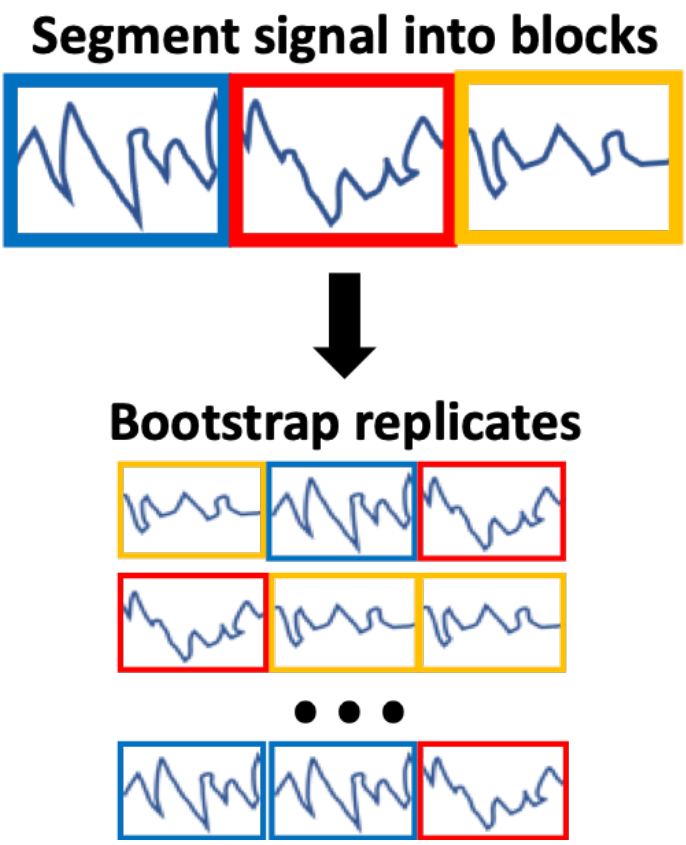
Plot patterns extracted by PCA (i.e. patterns of data):

In [ ]:

```
len_plot = 1000     # choose time-series length
for plotting purposes

num_plots = pca_data_ts.shape[1]
plt.figure(figsize=[9.,9.])

for i in range(num_plots):
    ax = plt.subplot( num_plots , 1, i+1 )
    plt.plot(zscore(pca_data_ts)[:len_plot,i])
```

```
    ax.set_ylabel("PCA {}".format(i), fontsize=8,
rotation=0)
    ax.set_xticks([]); ax.set_yticks([])
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["left"].set_visible(False)
    ax.spines["bottom"].set_visible(False)

plt.show()
plt.tight_layout()
```

It would be nice to evaluate how robust is PCA. In other words, would slight changes to the data translate into slight changes in PCA outputs, or rather lead to quite divergent results? One good way to probe the robustness of an algorithm is through *block-bootstrapping*.



Let's now create our bootstrap replicates:

In [ ]:

```
%%time

block_len = 10000  # choose size of bootstrap
blocks
num_boots = 50     # choose number of bootstrap
replicates

eeg_ts_boot = cbb(eeg_ts, block_len, num_boots,
replace=True).transpose(1,2,0)
```

Note: There are other useful magic commands for profiling time and memory usage. Unfortunately, installing the dependencies is not straightforward and therefore outside the scope of this worksop. You may consult [this link]() for more information.

Let's plot PCA time-series before and after bootstrapping:

In [ ]:

```
cmp = 0    # select which PCA component to plot
boot = 40  # select which bootstrap iteration to
plot
len_plot = 1000

t = np.arange(len_plot) / eeg.info['sfreq']

plt.figure(figsize=[9.,3.])
plt.subplot(2,1,1); plt.plot(t,
eeg_ts[:len_plot,cmp]); plt.title('Before
bootstrapping'); plt.yticks([])
plt.subplot(2,1,2); plt.plot(t,
eeg_ts_boot[:len_plot,cmp,boot]);
plt.title('After bootstrapping'); plt.yticks([])
# can write commands side-by-side with ";" semi-
colon seperator
plt.xlabel('Time (seconds)')
```

```
plt.show(); plt.tight_layout()
```

The fun part: repeat PCA on the bootstrap EEG data and apply learned PCA model on original EEG data:

In [ ]:

```
pca_boot_ts = []

for boot in range(num_boots):

    pca_boot = PCA(n_components=num_comps)
    pca_boot.fit(eeg_ts_boot[:,:,boot])
# learn PCA model on bootstrap EEG data
    pca_boot_ts.append(
pca_boot.transform(eeg_ts.squeeze()) )  # apply
PCA model on original EEG data

pca_boot_ts =
np.array(pca_boot_ts).transpose(1,2,0)
```

What does applying different bootstrap-based PCA models to the original EEG data look like?

In [ ]:

```
cmp = 1    # select which PCA component to plot

plt.figure(figsize=[9.,3.])
plt.plot(zscore(pca_boot_ts[:len_plot,cmp,:]));
plt.xticks([]); plt.yticks([])
plt.show()

# save image
```

Seems like we should ensure that time-series are not flipped:

In [ ]:

```
pca_boot_ts_signcorr = pca_boot_ts.copy()

for i in range(pca_data_ts.shape[-1]):
    signs = np.corrcoef( pca_data_ts[:,i] ,
pca_boot_ts[:,i,:].squeeze() , rowvar=False
)[0,1:]
    pca_boot_ts_signcorr[:,i,:] = kr([
signs[None] , pca_boot_ts[:,i,:].squeeze() ])
```

In [ ]:

```
cmp = 1    # select which PCA component to plot

plt.figure(figsize=[9.,3.])
plt.plot(zscore(pca_boot_ts_signcorr[:len_plot,cmp,:]));
plt.xticks([]); plt.yticks([])
plt.show()
```

Let's now compute the mean and standard deviation across bootstrap iterations for each PCA component:

In [ ]:

```
mean_boot_ts = np.mean(pca_boot_ts_signcorr,
axis=2)
std_boot_ts = np.std(pca_boot_ts_signcorr,
axis=2)
```

And now plot PCA time-series with bootstrap-based *confidence intervals*:

In [ ]:

```
len_plot = 1000  # choose time-series length for
plotting purposes

t = np.arange(len_plot)
plt.figure(figsize=[9.,9.])
```

```
for i in range(num_comps):

    y = mean_boot_ts[:len_plot,i]
    ci = std_boot_ts[:len_plot,i]

    ax = plt.subplot( num_plots , 1, i+1 )
    plt.fill_between(t, (y-ci), (y+ci),
color='orange', alpha=0.75)
    plt.plot(t, y)

    ax.set_ylabel("PCA {}".format(i), fontsize=8,
rotation=0)
    ax.set_xticks([]); ax.set_yticks([])
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["left"].set_visible(False)
    ax.spines["bottom"].set_visible(False)

plt.show()
plt.tight_layout()
```

exercise: create link In [ ]: from IPython.display import YouTubeVideo In [ ]: YouTubeVideo('owSGVOov9pQ', width=800, height=300) Jupyter, an interactive Python command shell (i.e. IPython)The IPython shell offers users an interactive environment, enabling easy navigation through directories and access to data. We can use many of the BASH commands which we'd usually run in a Terminal: So, rather than listing the content of our directory by executing the 'ls' BASH command in a Terminal, we can execute 'ls' directly from a code cell: In [ ]: ls Another BASH terminal command we could try is the cat command to display the content of a file: In [ ]: cat README.md We can display messages with echo: In [ ]: !echo "Hello MiCM !" We can create textfiles and write to them: In [ ]: !echo "Hello MiCM !" > my_message.txt In [ ]: ls In [ ]: cat

my_message.txt Let's now remove the textfile we've just created with rm: In [ ]: rm my_message.txt We can also verify the path to our current directory using pwd: In [ ]: pwd Create new (sub)directory "data" and list content of current directory. Note that some BASH commands such as mkdir must be preceded by !: In [ ]: mkdir data In [ ]: ls Move EEG data into "data" subdirectory using mv and list content of current directory (can use tab completion). We will be using this EEG data as an exercise later on: In [ ]: mv sub-010321_EC_downsamp.fdt data/ In [ ]: ls We can also use wildcards for referring to files with a known structure to their filename: In [ ]: mv *set data In [ ]: ls Files can be copied into other directories (under new names even): In [ ]: cp README.md data In [ ]: ls You can target which directory should have its content listed: In [ ]: ls data In [ ]: rm data/README.md In [ ]: cp README.md data/readme.md In [ ]: ls data Navigate to "data" using cd and display new location with pwd. Then, list content of current directory (which is now "data"): In [ ]: cd data/ In [ ]: pwd In [ ]: ls Navigate back to previous directory and verify that now in proper directory: In [ ]: cd .. In [ ]: pwd Whenever you forget where you had moved the data, use the terminal find command to retrieve its location: In [ ]: !find . -name sub* Warning! Using BASH terminal commands can be tricky at times. For instance, this series of commands should work just fine: In [ ]: ls In [ ]: cd data In [ ]: ls In [ ]: cd .. However, running all of the same commands within a common cell usually fails: In [ ]: ls cd data ls cd .. Adding ! helps, but may nonetheless not execute the desired commands properly: In [ ]: !ls !cd data !ls !cd .. ... Code cells can also be used to compute mathematical expressions: In [ ]: 2+3 In Python, the symbol for calculating powers of a number is ** (double asterisk). We can come up with a 'funky sum': In [ ]: 2**3 + 3**2 We may want to automate our funky sum by creating a dedicated function: In [ ]: def funky_sum(x,y): ''' With inputs x and y, the funky sum is computed as x**y + y**x. Parameters ---------- x : first term y : second term ''' z = x**y + y**x return z In [ ]: funky_sum(2,3) We can

use input to query users for the value of different variables: In [ ]: x = input('1st term of funky sum: ') y = input('2nd term of funky sum: ') x = int(x) y = int(y) funky_sum(x,y) We had also redacted some documentation for 'funkysum' which we can display using the *_doc__ attribute*: In [ ]: print(funky_sum.__doc__) What other attribute was created when compiling our funky_sum function? In [ ]: dir(funky_sum) Let's copy funky_sum and include an additional default argument: In [ ]: def funky_sum_offset(x,y,b=2): ''' With inputs x and y, the funky sum is computed as x**y + y**x + b. Parameters ---------- x : first term y : second term b : offset parameter (b=2 by default) ''' z = x**y + y**x + b return z In [ ]: funky_sum_offset(2,3) Let's now try the __defaults__ attribute of our new function: In [ ]: funky_sum_offset.__defaults__ In [ ]: funky_sum.__defaults__ In addition to attributes, functions also have built-in subfunctions called methods: In [ ]: funky_sum.__dir__() The dir function is also useful when exploring the content of a class. Let's create our own class to see how this is useful: In [ ]: class funky_class(): ''' A class for using funky sums. ''' def funky_sum(x,y): ''' With inputs x and y, the funky sum is computed as x**y + y**x. Parameters ---------- x : first term y : second term ''' z = x**y + y**x return z def funky_sum_offset(x,y,b=2): ''' With inputs x and y, the funky sum is computed as x**y + y**x + b. Parameters ---------- x : first term y : second term b : offset parameter (b=2 by default) ''' z = x**y + y**x + b return z Let's now test our new class and its functions: In [ ]: print(funky_class.__doc__) In [ ]: dir(funky_class) In [ ]: funky_class.funky_sum(2,3) In [ ]: funky_class.funky_sum_offset(2,3) Let's create some numpy arrays for doing funky sums. First, we'll import the numpy package and we'll give it the shorthand name 'np': In [ ]: import numpy as np In [ ]: x = np.arange(0,3) print('The elements of x are:',x) print('Said otherwise, there are {} elements in x'.format(len(x))) In [ ]: y = np.random.uniform(0,1, len(x)) print('The {} elements of y are {}'.format(len(y) , y)) In the same way we provided numpy with the

shorthand name 'np', we provide our custom 'funky_class.funky_sum_offset' function a shorthand name as well: In [ ]: fs = funky_class.funky_sum_offset In [ ]: z = fs(x[0],y[0]) print('The funky sum between the first element of x (i.e. {}) and first element of y (i.e. {:.2}) is {}'.format(x[0] , y[0] , z)) What happens if we input the full 'x' and 'y' arrays rather than just a single of their elements? In [ ]: z = fs(x,y) print('The funky sum between \n the elements of x (i.e. {}) \n and elements of y (i.e. {}) \n\t is {}'.format(x , y , z)) One important feature of Jupyter notebooks is the ability to generate plots side-by-side with your code. Let's plot the arrays used for doing funky sums. First we'll need to import matplotlib.pyplot: In [ ]: import matplotlib.pyplot as plt In [ ]: plt.figure() plt.plot(x, marker='o') plt.plot(y, marker='o') plt.plot(z, marker='o') plt.xticks([0,1,2], ["1st element","2nd element","3rd element"]) plt.xlabel("Elements") plt.ylabel("Values") plt.legend(["x","y","z"]) plt.show() Let's now repeat the process of creating and plotting the x, y, z arrays for different offset values: In [ ]: plt.figure() offset = 2 z = fs(x,y, offset) plt.subplot(2,2,1) plt.plot(x, marker='o') plt.plot(y, marker='o') plt.plot(z, marker='o') plt.xticks([]) plt.title('z = {}'.format(offset)) offset = 0 z = fs(x,y, offset) plt.subplot(2,2,2) plt.plot(x, marker='o') plt.plot(y, marker='o') plt.plot(z, marker='o') plt.xticks([]) plt.title('z = {}'.format(offset)) offset = -2 z = fs(x,y, offset) plt.subplot(2,2,3) plt.plot(x, marker='o') plt.plot(y, marker='o') plt.plot(z, marker='o') plt.xticks([]) plt.title('z = {}'.format(offset)) offset = -1 z = fs(x,y, offset) plt.subplot(2,2,4) plt.plot(x, marker='o') plt.plot(y, marker='o') plt.plot(z, marker='o') plt.xticks([]) plt.title('z = {}'.format(offset)) plt.suptitle('Funky sums') plt.show() EEG data analysis ## EEG = electroencephalography EEG is an electrophysiology modality where electrodes are placed on the scalp/head to measure fluctuations in bioelectical potentials. Hence, every electrode measures a time-varying signal. source: https://brainvision.com/applications/eeg/ Import packages for analysing EEG data In [ ]: # Must-haves (and already imported) import numpy as np import matplotlib.pyplot as plt # EEG analysis

import mne from mne.preprocessing import ICA from sklearn.decomposition import PCA, FastICA from scipy.stats import zscore from recombinator.block_bootstrap import circular_block_bootstrap as cbb from tensorly.tenalg import khatri_rao as kr Import EEG data using MNE Python analysis package. We will be using open-source data which has been reported in: Babayan, A., Erbey, M., Kumral, D. et al. A mind-brain-body dataset of MRI, EEG, cognition, emotion, and peripheral physiology in young and old adults. Sci Data 6, 180308 (2019). In [ ]: eeg = mne.io.read_raw_eeglab("data/sub-010321_EC_downsamp.set") eeg.annotations.delete(np.arange(len(eeg.annotations.description))) # remove annotations, not important Our 'eeg' is an instantiation of the class 'RawEEGLAB', which makes 'eeg' an object. In [ ]: eeg Objects usually already have built-in methods: In [ ]: dir(eeg) We can display useful information about the data using the info attribute: In [ ]: eeg.info For instance, we can plot the data stored in 'eeg' by calling its plot method: In [ ]: eeg.plot() We can also display the documentation of the plot method to see what are the plotting options: In [ ]: print(eeg.plot.__doc__) In [ ]: eeg.plot.__defaults__ Let's isolate the default value for the 'n_channels' argument: In [ ]: eeg.plot.__defaults__[3] Let's plot our EEG data again, this time changing the number of EEG channels on display: In [ ]: eeg.plot(n_channels=10) At this point, our plots are not interactive. We can change the backend of matplotlib to render interactive plots using a magic command. In [ ]: %matplotlib notebook In [ ]: eeg.plot() We can also display how are the EEG electrodes positioned on the head: In [ ]: eeg.plot_sensors(show_names=True) From the time-series plots, it seems that many time-series are alike. Perhaps there's a way to find patterns and have a more succinct representation of our data. One way to find these patterns is to use Independent Components Analysis (ICA) on our EEG data. source: https://sccn.ucsd.edu/~jung /Site/EEG_artifact_removal.html In [ ]: num_comps = 15 # choose

number of patterns, or "components" ica = ICA(n_components=num_comps, random_state=97) ica.fit(eeg) The 'ica' object also contains its own built-in method for plotting the time-series of ICA components: In [ ]: ica.plot_sources(eeg) Let's also plot the spatial topographies of the ICA components: In [ ]: ica.plot_components() Now, let's also try Principal Components Analysis (PCA). We will need to extract the time-series from our 'eeg' object as MNE does not yet support a built-in method for PCA as it does for ICA. source: https://slidetodoc.com/factor-and-component-analysis-esp-principal-component-analysis/ In [ ]: eeg_ts = eeg.get_data().T Let's display the shape of the EEG data using the built-in shape method for Numpy arrays: In [ ]: print("Shape of EEG data: ", eeg_ts.shape) Run PCA: In [ ]: num_comps = 15 # choose number of patterns, or "components" pca = PCA(n_components=num_comps) pca_data_ts = pca.fit_transform(eeg_ts) Plot patterns extracted by PCA (i.e. patterns of data): In [ ]: len_plot = 1000 # choose time-series length for plotting purposes num_plots = pca_data_ts.shape[1] plt.figure(figsize=[9.,9.]) for i in range(num_plots): ax = plt.subplot(num_plots , 1, i+1) plt.plot(zscore(pca_data_ts)[:len_plot,i]) ax.set_ylabel("PCA {}".format(i), fontsize=8, rotation=0) ax.set_xticks([]); ax.set_yticks([]) ax.spines["top"].set_visible(False) ax.spines["right"].set_visible(False) ax.spines["left"].set_visible(False) ax.spines["bottom"].set_visible(False) plt.show() plt.tight_layout() It would be nice to evaluate how robust is PCA. In other words, would slight changes to the data translate into slight changes in PCA outputs, or rather lead to quite divergent results? One good way to probe the robustness of an algorithm is through block-bootstrapping. Let's now create our bootstrap replicates: In [ ]: %%time block_len = 10000 # choose size of bootstrap blocks num_boots = 50 # choose number of bootstrap replicates eeg_ts_boot = cbb(eeg_ts, block_len, num_boots, replace=True).transpose(1,2,0) Note: There are other useful magic

commands for profiling time and memory usage. Unfortunately, installing the dependencies is not straightforward and therefore outside the scope of this worksop. You may consult this link for more information. Let's plot PCA time-series before and after bootstrapping: In [ ]: cmp = 0 # select which PCA component to plot boot = 40 # select which bootstrap iteration to plot len_plot = 1000 t = np.arange(len_plot) / eeg.info['sfreq'] plt.figure(figsize=[9.,3.]) plt.subplot(2,1,1); plt.plot(t, eeg_ts[:len_plot,cmp]); plt.title('Before bootstrapping'); plt.yticks([]) plt.subplot(2,1,2); plt.plot(t, eeg_ts_boot[:len_plot,cmp,boot]); plt.title('After bootstrapping'); plt.yticks([]) # can write commands side-by-side with ";" semi-colon seperator plt.xlabel('Time (seconds)') plt.show(); plt.tight_layout() The fun part: repeat PCA on the bootstrap EEG data and apply learned PCA model on original EEG data: In [ ]: pca_boot_ts = [] for boot in range(num_boots): pca_boot = PCA(n_components=num_comps) pca_boot.fit(eeg_ts_boot[:,:,boot]) # learn PCA model on bootstrap EEG data pca_boot_ts.append(pca_boot.transform(eeg_ts.squeeze())) # apply PCA model on original EEG data pca_boot_ts = np.array(pca_boot_ts).transpose(1,2,0) What does applying different bootstrap-based PCA models to the original EEG data look like? In [ ]: cmp = 1 # select which PCA component to plot plt.figure(figsize=[9.,3.]) plt.plot(zscore(pca_boot_ts[:len_plot,cmp,:])); plt.xticks([]); plt.yticks([]) plt.show() # save image Seems like we should ensure that time-series are not flipped: In [ ]: pca_boot_ts_signcorr = pca_boot_ts.copy() for i in range(pca_data_ts.shape[-1]): signs = np.corrcoef(pca_data_ts[:,i] , pca_boot_ts[:,i,:].squeeze() , rowvar=False)[0,1:] pca_boot_ts_signcorr[:,i,:] = kr([signs[None] , pca_boot_ts[:,i,:].squeeze()]) Plot again: In [ ]: cmp = 1 # select which PCA component to plot plt.figure(figsize=[9.,3.]) plt.plot(zscore(pca_boot_ts_signcorr[:len_plot,cmp,:])); plt.xticks([]); plt.yticks([]) plt.show() Let's now compute the mean and standard

deviation across bootstrap iterations for each PCA component:

In [ ]:
```
mean_boot_ts = np.mean(pca_boot_ts_signcorr, axis=2)
std_boot_ts = np.std(pca_boot_ts_signcorr, axis=2)
```
And now plot PCA time-series with bootstrap-based confidence intervals: In [ ]:
```
len_plot = 1000 # choose time-series length for plotting purposes
t = np.arange(len_plot)
plt.figure(figsize=[9.,9.])
for i in range(num_comps):
    y = mean_boot_ts[:len_plot,i]
    ci = std_boot_ts[:len_plot,i]
    ax = plt.subplot( num_plots , 1, i+1 )
    plt.fill_between(t, (y-ci), (y+ci), color='orange', alpha=0.75)
    plt.plot(t, y)
    ax.set_ylabel("PCA {}".format(i), fontsize=8, rotation=0)
    ax.set_xticks([]); ax.set_yticks([])
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["left"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
plt.show()
plt.tight_layout()
```