Cognitive Neuroscience: Group Project

Final Group Project Code Instructions

Marijn van Wingerden, Department of Cognitive Science and Artificial Intelligence – Tilburg University Academic Year 2020-2021

In this Jupyter Notebook, you will find the programmatic instructions to complete the Group Project.

You will analyse a subset of the trials and conditions from the RITA dataset, as introduced by dr. Roncaglia in the second week of class. Your time-resolved analysis, using the FFT and baseline averaging, will be performed across ALL subjects in a group (Monolinguals, Early Learners, Late Learners) in the dataset, and look for oscillatory activity in relation to the onset of the **critical item** in each sentence (the auxilary verb).

You will inspect activity in the following frequency bands:

- Delta (1-4 Hz)
- Theta (4-8 Hz)
- Alpha (8-12 Hz)
- Beta (15-25 Hz)
- low Gamma (30-60 Hz)
- high Gamma (60-100 Hz) (different cutoffs points can be found in the literature, we are sticking with these for convenience)

All relevant methods have been covered in Worksheets 1-7, with the exception of loading multiple datafiles. Whenever a new method/function is introduced here, it will come with an example.

Datafiles - assignment per group

Each group will analyze one set of filler sentences (NA: Non-Ambiguous sentences) and a set of experimental sentences (AM: Ambiguous sentences). The conditions in the dataset are split over groups in the following way:

- SF: subject-first
- OF: object-first
- IR: irregular rhythm
- RR: regular rhythm

- Monolinguals: RM
    - Groups 01 + 02: SFIR triggers XX1 and XX5
    - Groups 03 + 04: SFRR triggers XX2 and XX6
    - Groups 05 + 06: OFIR triggers XX3 and XX7
    - Groups 07 + 08: OFRR triggers XX4 and XX8
- Early Learners: RB
    - Groups 09 + 10: SFIR triggers XX1 and XX5
    - Groups 11 + 12: SFRR triggers XX2 and XX6
    - Groups 13 + 14: OFIR triggers XX3 and XX7
    - Groups 15 + 16: OFRR triggers XX4 and XX8
- Late Learners: RL
    - Groups 17 + 18: SFIR triggers XX1 and XX5
    - Groups 19 + 20: SFRR triggers XX2 and XX6
    - Groups 21 + 22: OFIR triggers XX3 and XX7
    - Groups 23 + 24: OFRR triggers XX4 and XX8
- Odd groups: analyse the odd trials in the dataset

- Even groups: analyse the even trials in the dataset

You can download the datafiles from: https://surfdrive.surf.nl/files/index.php/s/JcA9speED020q4p

Handing in of your code

You can adapt this script template and hand it in as the code component of your Group Assignment Report.

Whenever you are asked to make a plot, it should be completed with a meaningful plot title, xlabel and ylabel texts. Figures are started with a Matplotlib figure handle: "fig_Q2A, ax = plt.subplots;". This indicates that a link (called handle) to your figure will be saved in the variable, so we can easily check it when checking your scripts. Whenever a naming convention for a variable is given, use it, because it will allow semi-automatic grading of your project script.

Intermediate hand-in

You will be able to hand in a script/notebook with your solutions to Q1-3 after the midterms, and the correct solution will be discussed in class to give you some feedback on how the Group Project is evaluated. Groups that do not hand in their solutions to Q1-3 will receive half of the total points available for these questions by default.

Group members:

Please list the contributors and their U-numbers here in comments:

- Ndivhuwo Nyase u835598
- Pietro Garromi u915216
- Djourdan Gomes-Johnson u
- Siem Houkes u402843
-

Setting up: list your modules to import

For loading/saving puroposes, we will make use of the **os** package. An example worksheet with instructions on how to use the os package will be provided

In [1]:

```
%matplotlib notebook

import os
import numpy as np
from pprint import pprint
import pandas as pd
import matplotlib.pyplot as plt
import scipy.fft as fft
```

Data loading

We will need to load the datafiles from all participants and add them all together so that we end up with a matrix that has nChannels x nTime x nParticipants (instead of trials). You can make your work easier by organising the datafiles in such a way that you put the control.npy files in their own subdirectory, and the experimental.npy files as well.

In order to load the files, we can use the os package.

Adapt the following so that it works on your machine:

In [2]:

```
# Get the current working directory
wd = os.getcwd()
print(wd)
```

```
C:\Users\Admin\Desktop\Tilburg University\Year 2\Semester 2\Cognitive Neuroscience\Group Project
```

In [3]:

```
path_control =  wd + '/Project_Data_Files/' + 'group_15/'
path_experimental = wd + '/Project_Data_Files/' + 'group_15/'
path_base = wd + '/Project_Data_Files/' + 'group_15/'
files = os.listdir(path_base)
control_files = list()
experimental_files = list()

for f in files:
 # check the files that end with specific extention
    if f.rfind("part_10") > -1:
        continue
    elif f.endswith("control.npy"):
        control_files.append(f)
    elif f.endswith("experimental.npy"):
        experimental_files.append(f)


# check that the length of your files list matches the provided datafiles, and contains only .npy datafi

control_files.sort()
pprint(control_files)
print(len(control_files))
experimental_files.sort()
pprint(experimental_files)
print(len(experimental_files))
print(len(experimental_files))
```

```
['group_15_part_01_control.npy',
 'group_15_part_02_control.npy',
 'group_15_part_03_control.npy',
 'group_15_part_04_control.npy',
 'group_15_part_05_control.npy',
 'group_15_part_06_control.npy',
 'group_15_part_07_control.npy',
 'group_15_part_08_control.npy',
 'group_15_part_09_control.npy',
 'group_15_part_11_control.npy',
 'group_15_part_12_control.npy',
 'group_15_part_13_control.npy',
 'group_15_part_14_control.npy',
 'group_15_part_15_control.npy',
 'group_15_part_16_control.npy',
 'group_15_part_17_control.npy',
 'group_15_part_18_control.npy',
 'group_15_part_19_control.npy',
 'group_15_part_20_control.npy',
 'group_15_part_21_control.npy',
 'group_15_part_22_control.npy']
21
['group_15_part_01_experimental.npy',
 'group_15_part_02_experimental.npy',
 'group_15_part_03_experimental.npy',
 'group_15_part_04_experimental.npy',
 'group_15_part_05_experimental.npy',
 'group_15_part_06_experimental.npy',
 'group_15_part_07_experimental.npy',
 'group_15_part_08_experimental.npy',
 'group_15_part_09_experimental.npy',
 'group_15_part_11_experimental.npy',
 'group_15_part_12_experimental.npy',
 'group_15_part_13_experimental.npy',
 'group_15_part_14_experimental.npy',
 'group_15_part_15_experimental.npy',
 'group_15_part_16_experimental.npy',
 'group_15_part_17_experimental.npy',
 'group_15_part_18_experimental.npy',
 'group_15_part_19_experimental.npy',
 'group_15_part_20_experimental.npy',
 'group_15_part_21_experimental.npy',
 'group_15_part_22_experimental.npy']
21
21
```

Combining data and matrix pre-allocation

next, you will need to load these files one by one and extract the data for this participant. The data in the NumPy arrays are stored as Trials x Channels x Time. To aggregate across participants, you will thus need to add a 4th dimension to store the data.

To be able to adequately pre-allocate the data from the different subjects, we will load one trial subject manually to have a look at the shape/dimensionality of the data:

In [4]:

```
EEG = np.load(os.path.join(path_control,control_files[0]))

# control_files is a list of strings, so indexing its first element returns a string
# in this case, we are loading the first entry of control_files, i.e. participant 1

# verify that the number of trials equals 22,
# verify that the number of channels equals 64 or 65
# and verify that there are 751 samples per trace

print("Number of trials = ", EEG.shape[0])
print("Number of channels = ", EEG.shape[1])
print("Number of timepoints = ", EEG.shape[2])
```

```
Number of trials =  22
Number of channels =  64
Number of timepoints =  751
```

Q1 - setting up the data structure and loading data from all participants

The EEG data is currently stored as a 3-dimensional NumPy array. But to run our time-frequency analysis, we need some more information like the sampling rate and the time axis that corresponds to the stimulus-locked analysis window. In order to set up (=pre-allocate) a matrix that will hold all traces for all participants, we need to know the sizes of the dimensions of this 4-dimensional matrix, and fill up this matrix by looping over participants:

In [5]:

```
# There are 64 or 65 channels in the dataset. Only channels up to channel 59 are EEG channels
# the remaining channels are EMG and EOG channels that we will ignore in this analysis
# subset your EEG array so that only the EEG channels remain

EEG = EEG[:,:59,:]
print(EEG.shape)

# Define nTrials, nChans (=channels), nSamples and nParts (=participants). Then, pre-allocate a matrix
# filled with zeros and with size nTrials x nChans x nSamples x nParticipants, one each for the control
# and experimental data. Name them comb_data_control and comb_data_experimental
nTrials = EEG.shape[0]
nChans = EEG.shape[1]
nSamples = EEG.shape[2]
nParts = len(control_files)
comb_data_control = np.zeros((nTrials , nChans, nSamples , nParts))
comb_data_experimental = np.zeros((nTrials , nChans, nSamples , nParts))

# next, we need to loop over all participant datafiles and add them to the appropriate slice in your 4-D
# For this, you need to use specific array indexing to indicate where in comb_data_(control/experimental)
# each participant's data needs to go. You can and should reuse the data-reading code above.

# loop over participants, and wihtin each iteration of the loop, load the
# next datafile and fill comb_data_(control/experimental) with the EEG traces (nTrials x nChans x nSampl
# check the shape of the matrices after filling them

for iPart in range(len(control_files)):
    comb_data_control[:,:,:,iPart] = np.load(os.path.join(path_control,control_files[iPart]))[:,:59,:]
    comb_data_experimental[:,:,:,iPart] = np.load(os.path.join(path_experimental,experimental_files[iPart
print(comb_data_control.shape)
print(comb_data_experimental.shape)
```

```
(22, 59, 751)
(22, 59, 751, 21)
(22, 59, 751, 21)
```

Q2 - explore the data

Let's explore this newly combined dataset a little bit. This collection of EEG traces in your dataset has been taken with a [-0.5s, 1s] window around the relevant event. What's more, each trace has been averaged to its baseline period, so that the mean amplitude should be 0 (with some rounding error).

To verify, first, determine the mean for the time period of -0.5 to 0 seconds. Given that the srate = 500 Hz, the baseline period corresponds to the first 250 samples. We will work with only the control dataset (comb_data_control) in this exercise.

- subset your combined data to only the first 250 samples
- select a random participant and subset the data further to only this participant
- select a random EEG channel and subset the data further to only this channel

This should leave you with a nTrials x 250 (samples) matrix. Create a similar evoked matrix with the remainder of the samples. With these matrices, in a 1x3 subplot

- plot the traces for all trials in the **baseline** matrix (use transpose if necessary).
  - the plot should have 250 samples on the x-axis, and nTrial number of lines
- calculate the mean for each trace (i.e., across the samples in a trace):
  - once for the baseline period
  - once for the remainder of the trial
- plot these values (N of these values should be nTrials, check this) in a histogram, each in their own subplot

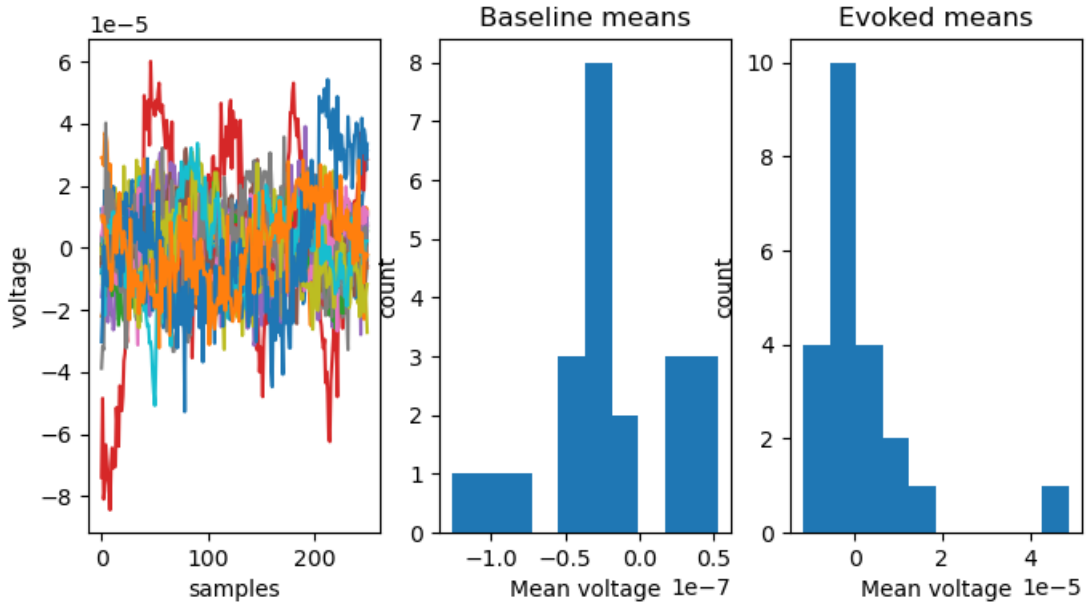
Refer to Worksheet 1 for example uses of **np.mean**. np.std works in a similar way

In [50]:

```
subset_baseline = comb_data_control[:,:,:250,:]
print(subset_baseline.shape) # trials x channels x timepoints x participants
subset_participant_baseline = subset_baseline[:,:,:,1]
print(subset_participant_baseline.shape) # trials x channels x timepoints
subset_channel_baseline = subset_participant_baseline[:,20,:]
print(subset_channel_baseline.shape) # trials x timepoints
```

```
subset_evoked = comb_data_control[:,:,250:,:]
print(subset_evoked.shape) # trials x channels x timepoints x participants
subset_participant_evoked = subset_evoked[:,:,:,1]
print(subset_participant_evoked.shape) # trials x channels x timepoints
subset_channel_evoked = subset_participant_evoked[:,20,:]
print(subset_channel_evoked.shape) # trials x timepoints

baseline_mean = np.mean(subset_channel_baseline, axis = 1) # take mean across samples
print(baseline_mean.shape)
evoked_mean = np.mean(subset_channel_evoked, axis = 1) # take mean across samples
print(evoked_mean.shape)

# now plot these traces and two histograms using the subplot given
fig_Q2A, ax = plt.subplots(figsize=(8,4), nrows=1, ncols=3) # 1x3 graph

# plot the traces in ax[0]
ax[0].plot(np.transpose(subset_channel_baseline))
ax[0].set_xlabel("samples")
ax[0].set_ylabel("voltage")
plt.title("EEG traces in baseline period")

# plot the baseline mean in ax[1] with this code
ax[1].hist(baseline_mean)
plt.axes(ax[1])
plt.title('Baseline means')
ax[1].set_xlabel('Mean voltage')
ax[1].set_ylabel('count')

# plot the evoked mean in ax[2] with this code
ax[2].hist(evoked_mean)
plt.axes(ax[2])
plt.title('Evoked means')
ax[2].set_xlabel('Mean voltage')
ax[2].set_ylabel('count')
plt.show()
```

```
(22, 59, 250, 21)
(22, 59, 250)
(22, 250)
(22, 59, 501, 21)
(22, 59, 501)
(22, 501)
(22,)
(22,)
```



Looking at theses histograms, you should see that the distribution of evoked mean values can vary: while EEG recordings are referenced to a common ground, and usualy then re-referenced to the global average, we are dealing here with specific cutouts of EEG traces around specific events in the dataset. The data has been normalised to the baseline window for each trial, but the mean of the evoked part is not controlled. In addition, there might be differences in the size of the amplitudes between channels and between participants due to differences in conductivity.

Illustrate that this global averaging does not guarantee equal variance. Reuse the baseline and evoked subsets:

- Compute, instead of the mean across samples, the standard deviation (numpy.std or variants)
- calculate the standard deviation for each trace:
  - once for the baseline period
  - once for the remainder of the trial
- Adapt the plotting code for Q2A and plot these distributions of standard deviations in figure Q2B.

In [7]:

```
baseline_std = np.std(subset_channel_baseline, axis = 1) # take std across samples
print(baseline_std.shape)
evoked_std = np.std(subset_channel_evoked, axis = 1) # take std across samples
print(evoked_std.shape)

fig_Q2B, ax = plt.subplots(figsize=(8,4), nrows=1, ncols=3) # don't forget proper plot annotation

# plot the traces in ax[0]
ax[0].plot(np.transpose(subset_channel_baseline))
ax[0].set_xlabel("samples")
ax[0].set_ylabel("voltage")
plt.title("EEG traces in baseline period")

# plot the baseline mean in ax[1] with this code
ax[1].hist(baseline_std)
plt.axes(ax[1])
plt.title('Baseline std')
ax[1].set_xlabel('Std voltage')
ax[1].set_ylabel('count')

# plot the evoked mean in ax[2] with this code
ax[2].hist(evoked_std)
plt.axes(ax[2])
plt.title('Evoked std')
ax[2].set_xlabel('Std voltage')
ax[2].set_ylabel('count')
plt.show()
```

```
(22,)
(22,)
```

Q3 - normalizing the data

So, if there are large differences in mean and/or standard deviation between channels or participants, we can implement some standard scaling. First, let think about why this "standard scaling" (substracting the mean, dividing by std) is important? You will be combining data from different participants for these exercises. They have been possibly recorded at different days, with different gels or electrodes, and thus with different conductivity between participants. We will assume that the recording across trials within one participant remains stable. So, in order to compare and average the recordings from different participants "fairly", we want them to be on more or less the same scale.

We can thus attempt to normalize the signal per participant, by dividing all data per participant by its standard deviation. Let's show the extent of the problem by plotting the participants with the lowest and highest std side by side. Re-create your matrix of std values for the evoked period, but:

- do not subset one participant, but retain all participants (still only selecting 1 channel)
- calculate the standard deviation for each trace (across the samples dimension)
- average the std values for each trace over trials, save in part_std
  - you will retain one value per participant, check this
- use np.argmax (and variants) to create two indexes, min_std and max_std that point to the participants with the lowest and highest standard deviations
- calculate and plot the signal average over trials for these two participants, using different line colors and proper line labeling.
  - Plot them in the first subplot of a 1x2 subplot
- Observe the scaling difference

In [8]:

```
selected_channel = subset_evoked[:,20,:,:]
print(selected_channel.shape)
part_std_all = np.std(selected_channel, axis=1) # we take the std across the second dimension (samples)
print(part_std_all.shape)

part_std = np.mean(part_std_all, axis = 0) # from these, we average across trials
```

```
print(part_std.shape)

min_std = np.argmin(part_std,axis=0)
max_std = np.argmax(part_std,axis=0)
print("Participants with min_std: ", min_std)
print("Participants with max_std: ", max_std)

fig_Q3, ax = plt.subplots(figsize=(10,3), nrows = 1, ncols = 2) # don't forget proper plot annotation
ax[0].plot(np.mean(selected_channel[:,:,min_std], axis = 0), label = "Lowest std")
ax[0].plot(np.mean(selected_channel[:,:,max_std], axis = 0), label = "Highest std")
ax[0].legend(loc = 'upper left')
plt.axes(ax[0])
plt.title('Lowest & Highest std participants', loc = 'right')
ax[0].set_xlabel('samples')
ax[0].set_ylabel('Mean voltage')
plt.show()
```

```
(22, 501, 21)
(22, 21)
(21,)
Participants with min_std:  11
Participants with max_std:  4
```

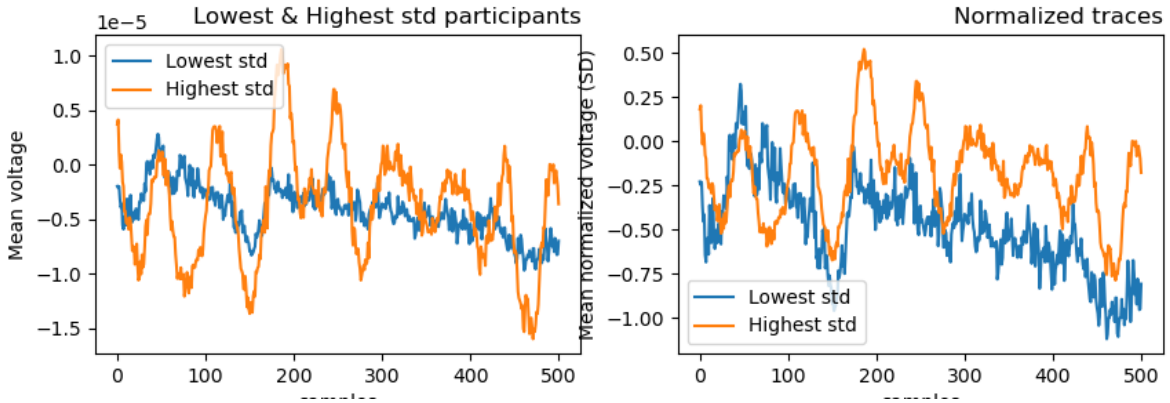
Next, we want to use the standard deviation for these participants to normalize (i.e. divide) the data by this value.

- using the selected channel matrix for these two participants as calculated above
  - this should be a trials x timepoints matrix, per participant
  - store as low_std_participant and high_std_participant
- extract the standard deviation for these participant as calculated above
- Normalize both sets traces (per participant) by the participant std
  - save as low_std_norm and high_std_norm
- In the second subplot, plot the **normalized** average signal over trials for these same two participants, using different line colors and proper line labeling. You can replot the figure if you want.
  - Plot them in the second subplot of fig Q3
  - The data should now be more or less on the same scale

In [47]:

```
low_std_part = selected_channel[:,:,min_std] # select data from participant with lowest std
low_std_norm = low_std_part / part_std[min_std] # divide by the average std for this participant
print(low_std_norm.shape) # trials x timepoints

high_std_part = selected_channel[:,:,max_std] # select data from participant with highest std
high_std_norm = high_std_part / part_std[max_std] # divide by the average std for this participant
print(high_std_norm.shape) # trials x timepoints

fig_Q3, ax = plt.subplots(figsize=(10,3), nrows = 1, ncols = 2) # don't forget proper plot annotation
ax[0].plot(np.mean(selected_channel[:,:,min_std], axis = 0), label = "Lowest std")
ax[0].plot(np.mean(selected_channel[:,:,max_std], axis = 0), label = "Highest std")
ax[0].legend(loc = 'upper left')
plt.axes(ax[0])
plt.title('Lowest & Highest std participants', loc = 'right')
ax[0].set_xlabel('samples')
ax[0].set_ylabel('Mean voltage')

ax[1].plot(np.mean(low_std_norm, axis = 0), label = "Lowest std")
ax[1].plot(np.mean(high_std_norm, axis = 0), label = "Highest std")
ax[1].legend(loc = 'lower left')
plt.axes(ax[1])
plt.title('Normalized traces', loc = 'right')
ax[1].set_xlabel('samples')
ax[1].set_ylabel('Mean normalized voltage (SD)')

plt.show()
```

```
(22, 501)
(22, 501)
```



We now have to apply this normalization across all channels. The fairest way is to calculate the grand standard deviation per participant (over all their channels, so the relative scaling between channels remains intact). This will normalize the range of the signal within one participant, so that they will be comparable between participants. We will have to do this both for the control and the experimental dataset.

In order to do this:

- first preallocate two normalized matrices with the same size as comb*data*(control/experimental), called comb_data*norm*(control/experimental).
- Next, create a loop to go over Participants, and inside the loop:
  - select the data for the current participant (trials x channels x timepoints)
    - separately for the control and experimental datasets.
  - calculate the grand standard deviation per participant
    - also separately for the control and experimental datasets
  - normalization all values by this grand standard deviation
    - Examine the numpy.std documentation to get a single std value across a 3D matrix
  - save the normalized data per participant in the pre-allocatede normalized matrix
    - separately for control/experimental datasets

In [10]:

```
comb_data_norm_control = np.zeros(comb_data_control.shape)
comb_data_norm_experimental = np.zeros(comb_data_experimental.shape)

for iPart in range(len(control_files)):
    # control trials
    curr_dat = comb_data_control[:,:,:,iPart]
    curr_std = np.std(curr_dat)
    comb_data_norm_control[:,:,:,iPart] = comb_data_control[:,:,:,iPart]/curr_std # element-wise divisio
    # experimental trials
    curr_dat = comb_data_experimental[:,:,:,iPart]
    curr_std = np.std(curr_dat)
    comb_data_norm_experimental[:,:,:,iPart] = comb_data_experimental[:,:,:,iPart]/curr_std # element-wi
    print("processing participant: ", iPart+1, "of ", len(control_files))
```

```
processing participant:  1 of  21
processing participant:  2 of  21
processing participant:  3 of  21
processing participant:  4 of  21
processing participant:  5 of  21
processing participant:  6 of  21
processing participant:  7 of  21
processing participant:  8 of  21
processing participant:  9 of  21
processing participant:  10 of  21
processing participant:  11 of  21
processing participant:  12 of  21
processing participant:  13 of  21
processing participant:  14 of  21
processing participant:  15 of  21
processing participant:  16 of  21
processing participant:  17 of  21
processing participant:  18 of  21
processing participant:  19 of  21
processing participant:  20 of  21
processing participant:  21 of  21
```

Finally, we will be plotting the raw and normalised datasets side by side to observe the differences. Before you starts, make sure you have two 4-dimensional matrices, one for the control data and one for the experimental data. Both should contain the full number of samples (e.g., baseline and evoked period).

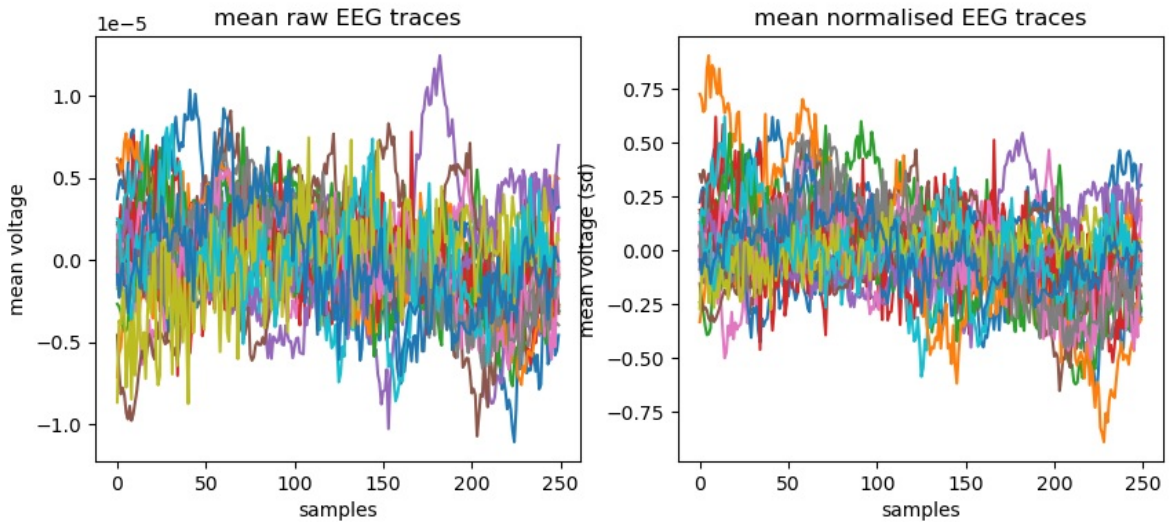
Create a 1x2 subplot, with on the left the raw data and on the right the normalised data

- re-use part of the code in Q2 to make a subset of the baseline data
- from this subset, select a random channel
- for both subsetted matrices, calculate the average signal over trials
  - this should result in a nTimepoints x nParticipants 2D matrix
  - store as mean_baseline_raw and mean_baseline_norm
- Then, plot this average signal over time (as in Fig Q2A) in a 1x2 subplot
  - for the raw dataset in panel 0
  - for the normalised dataset in panel 1
- Save this figure as Figure1

In [48]:

```
# non-normalised data
subset_baseline_raw = comb_data_control[:,:,:250,:]
print(subset_baseline_raw.shape) # trials x channels x timepoints x participants
subset_channel_baseline_raw = subset_baseline_raw[:,20,:,:]
mean_baseline_raw = np.mean(subset_channel_baseline_raw, axis = 0) # across trials
print(mean_baseline_raw.shape) # timepoints x participants

# normalised data
subset_baseline_norm = comb_data_norm_control[:,:,:250,:]
print(subset_baseline_norm.shape) # trials x channels x timepoints x participants
subset_channel_baseline_norm = subset_baseline_norm[:,20,:,:]
mean_baseline_norm = np.mean(subset_channel_baseline_norm, axis = 0) # across trials
print(mean_baseline_norm.shape) # timepoints x participants

# now plot the raw average traces and normalised average traces
fig_Q3C, ax = plt.subplots(figsize=(10,4), nrows=1, ncols=2) # 1x2 graph

# plot the mean raw traces in ax[0]
ax[0].plot(mean_baseline_raw)
ax[0].set_xlabel("samples")
ax[0].set_ylabel("mean voltage")
plt.axes(ax[0])
plt.title("mean raw EEG traces")

# plot the mean normalised traces in ax[1]
ax[1].plot(mean_baseline_norm)
plt.axes(ax[1])
plt.title("mean normalised EEG traces")
ax[1].set_xlabel('samples')
ax[1].set_ylabel('mean voltage (sd)')
plt.show()

# save Figure Q3C as your Figure 1 for your report
Figure1 = fig_Q3C
```

```
(22, 59, 250, 21)
(250, 21)
(22, 59, 250, 21)
(250, 21)
```



You should see that the scaling **within** a participant (line of across-trial average signal) is not really affected, but that potential outliers **between** participants have been brought into the same scale range.

We are now done with pre-processing of the data and ready to commence the spectral analysis

Q4 - set up the data structure for Time Frequency Analysis

Now that we have explored and normalized our data, we can proceed with the Time-Frequency Analysis.

We will need some basic information about the sampling rate and thus the frequency resolution. In addition, we need to define our time axis. You can refer to the earlier notebooks for information on how to do this. The EEG data was sampled at 500 Hz, the first sample was taken at t = -0.5s before the critical time, and the last sample was taken at t = +1.0s after the critical item

- Define the sampling rate (save as srate)
- Define the Nyquist frequency (save as nyquist_freq)
- Define the time step and time axis (save as sample_step and time)

In [12]:

```
srate = 500 # define a sampling rate - the number of timesteps per second (in Hz)

nyquist_freq = srate/2 #Nyquist frequency -- the highest frequency you can measure in the data
sample_step = 1/srate;

# define time. We want a 1D array that holds a sequence that goes from -0.5
# (seconds) to +1 in steps of the sampling frequency.
time = np.arange(start = -0.5, stop = 1, step = sample_step) # in seconds
print("shape of the time axis:",time.shape)
```

```
shape of the time axis: (750,)
```

Q5 - preallocating the TFR 5D array

Before we can preallocate our TFR matrix with a time-resolved frequency decomposition (per trial, channel, participants, and per analysis window), we need to figure out how many analysis windows should be in the final array, based on the window length, the overlap and the number of samples in each EEG trace in the dataset. We also need to know how many fourier coefficients we need to store per analysis window.

For the Frequency decomposition to have an optimal temporal resolution, we need to choose an analysis window length that is as short as possible, while still allowing a frequency resolution of 2Hz with the current sampling rate. Notebook 7 explained the relationship between the number of samples and the frequency resolution.

- Define the number of samples that will yield a 2Hz frequency resolution with the current sampling rate
- Calculate spect_freqs as this index of frequencies that can be tested with a 2Hz frequency resolution
- Define nFreq as the number of frequencies that will be analysed

Next, now that we know the size of our analysis window, we can define the timestep between adjacent windows. Remember that we want adjacent analysis windows to overlap, so the move across the time axis is always with a percentage of the length of the analysis window. Our first analysis window start at time index 0 (the first sample). The last window starts at some point on the time axis so that in ends exactly at the second-to-last sample.

- Set window_step to 4% of the length of the analysis window to allow sufficient overlap between adjacent analysis windows.
- Define nWindows as the number of analysis windows you will be able to fit in the time axis, considering the timestep between adjacent windows.
- create a vector called window_starts that has the starting sample (not time!) of each analysis window.
  - check your work: as the windows shift by 4% of the window length, the 26th window should start exactly 1 window length after the 1st.

Now, we can pre-allocate our final array to hold the Fourier Spectrum for each window analysed.

- Pre-allocate a zeros-filled TFR matrix with 5 dimensions:
  - nTrials x nChans x nWindows x nFreq x nParticipants
  - store as TFR*5D*(control/experimental)
  - add np.nan to the entire matrix. This will fill all positions with NaN

In [13]:

```
# Create a default signal

amplit = np.array([1])
thetas = np.array([0*np.pi])# zero phase
freqs = np.array([2])
freq_mat = np.zeros((len(freqs),len(time)))
for iMat in range(freq_mat.shape[0]):
    freq_mat[iMat,:] = amplit[iMat]*np.sin(2*np.pi*freqs[iMat]*time + thetas[iMat]) # one line per signal
print("shape of the freq_mat:",freq_mat.shape)
signal = np.sum(freq_mat, axis=0)
```

```
# we will first pre-allocate a zeros-filled matrix to hold the data
N_window_samples = int(nyquist_freq)
print("n_window_samples: ",N_window_samples)

N = len(time) # Number of samples
print("N :", N)
spect_freqs = fft.rfftfreq(N, sample_step)
#print("Spect_freqs", spect_freqs)
nFreq = fft.rfftfreq(int(N_window_samples), 1/srate)
print("Nfreq", nFreq)
```

```
shape of the freq_mat: (1, 750)
n_window_samples:  250
N : 750
Nfreq [  0.   2.   4.   6.   8.  10.  12.  14.  16.  18.  20.  22.  24.  26.
  28.  30.  32.  34.  36.  38.  40.  42.  44.  46.  48.  50.  52.  54.
  56.  58.  60.  62.  64.  66.  68.  70.  72.  74.  76.  78.  80.  82.
  84.  86.  88.  90.  92.  94.  96.  98. 100. 102. 104. 106. 108. 110.
 112. 114. 116. 118. 120. 122. 124. 126. 128. 130. 132. 134. 136. 138.
 140. 142. 144. 146. 148. 150. 152. 154. 156. 158. 160. 162. 164. 166.
 168. 170. 172. 174. 176. 178. 180. 182. 184. 186. 188. 190. 192. 194.
 196. 198. 200. 202. 204. 206. 208. 210. 212. 214. 216. 218. 220. 222.
 224. 226. 228. 230. 232. 234. 236. 238. 240. 242. 244. 246. 248. 250.]
```

In [14]:

```
total_samples = len(signal)
print(total_samples)
```

```
750
```

In [15]:

```
#define the length, the start and the end of the windows
window_step = 0.04*N_window_samples
last_window_start = total_samples - N_window_samples
window_starts = np.arange(start = 0, stop =last_window_start+1, step = window_step)
print("last start: ", last_window_start)
print("starts :", window_starts)
nWindows = len(window_starts)
print("number of windows: ", nWindows)
window_ends = window_starts + int(nyquist_freq)
print("ends: ", window_ends)

print("Window step:", window_step)

for start, end in zip(window_starts, window_ends):
    print("Starting at sample", start, "ending at", end,)
```

```
last start:  500
starts : [  0.  10.  20.  30.  40.  50.  60.  70.  80.  90. 100. 110. 120. 130.
 140. 150. 160. 170. 180. 190. 200. 210. 220. 230. 240. 250. 260. 270.
 280. 290. 300. 310. 320. 330. 340. 350. 360. 370. 380. 390. 400. 410.
 420. 430. 440. 450. 460. 470. 480. 490. 500.]
number of windows:  51
ends:  [250. 260. 270. 280. 290. 300. 310. 320. 330. 340. 350. 360. 370. 380.
 390. 400. 410. 420. 430. 440. 450. 460. 470. 480. 490. 500. 510. 520.
 530. 540. 550. 560. 570. 580. 590. 600. 610. 620. 630. 640. 650. 660.
 670. 680. 690. 700. 710. 720. 730. 740. 750.]
Window step: 10.0
Starting at sample 0.0 ending at 250.0
Starting at sample 10.0 ending at 260.0
Starting at sample 20.0 ending at 270.0
Starting at sample 30.0 ending at 280.0
Starting at sample 40.0 ending at 290.0
Starting at sample 50.0 ending at 300.0
Starting at sample 60.0 ending at 310.0
Starting at sample 70.0 ending at 320.0
Starting at sample 80.0 ending at 330.0
Starting at sample 90.0 ending at 340.0
Starting at sample 100.0 ending at 350.0
Starting at sample 110.0 ending at 360.0
Starting at sample 120.0 ending at 370.0
Starting at sample 130.0 ending at 380.0
Starting at sample 140.0 ending at 390.0
Starting at sample 150.0 ending at 400.0
Starting at sample 160.0 ending at 410.0
Starting at sample 170.0 ending at 420.0
Starting at sample 180.0 ending at 430.0
Starting at sample 190.0 ending at 440.0
Starting at sample 200.0 ending at 450.0
Starting at sample 210.0 ending at 460.0
Starting at sample 220.0 ending at 470.0
Starting at sample 230.0 ending at 480.0
Starting at sample 240.0 ending at 490.0
Starting at sample 250.0 ending at 500.0
Starting at sample 260.0 ending at 510.0
Starting at sample 270.0 ending at 520.0
Starting at sample 280.0 ending at 530.0
Starting at sample 290.0 ending at 540.0
Starting at sample 300.0 ending at 550.0
Starting at sample 310.0 ending at 560.0
Starting at sample 320.0 ending at 570.0
Starting at sample 330.0 ending at 580.0
Starting at sample 340.0 ending at 590.0
Starting at sample 350.0 ending at 600.0
Starting at sample 360.0 ending at 610.0
Starting at sample 370.0 ending at 620.0
Starting at sample 380.0 ending at 630.0
Starting at sample 390.0 ending at 640.0
Starting at sample 400.0 ending at 650.0
Starting at sample 410.0 ending at 660.0
Starting at sample 420.0 ending at 670.0
Starting at sample 430.0 ending at 680.0
Starting at sample 440.0 ending at 690.0
Starting at sample 450.0 ending at 700.0
Starting at sample 460.0 ending at 710.0
Starting at sample 470.0 ending at 720.0
Starting at sample 480.0 ending at 730.0
Starting at sample 490.0 ending at 740.0
Starting at sample 500.0 ending at 750.0
```

In [16]:

```
# create 5D zero matrices for both experiment and control
print(nTrials)
print(nChans)
print(nWindows)
print(len(nFreq))
print(nParts)

TFR_5D_control = np.zeros((nTrials, nChans, nWindows, len(nFreq), nParts))
TFR_5D_exp =  np.zeros((nTrials, nChans, nWindows, len(nFreq), nParts))
print(TFR_5D_control.shape)
print(TFR_5D_exp.shape)
```

```
22
59
51
126
21
(22, 59, 51, 126, 21)
(22, 59, 51, 126, 21)
```

In [17]:

```
# replacing the 0s with nans
TFR_5D_control[:] = np.nan
TFR_5D_exp[:] = np.nan
#print(TFR_5D_control)
print(TFR_5D_control.shape)
print(TFR_5D_exp.shape)
```

```
(22, 59, 51, 126, 21)
(22, 59, 51, 126, 21)
```

Q6 - performing a time-resolved frequency analysis

Now that we have the TFR_5D array pre-allocated, we can start to fill this 5D TFR matrix in a quadruple loop. Perform the following steps for your control and experimental TFR_5D arrays:

- Loop over nTrials, nChans and nParticipants.
- Inside this nested loop, extract the current full EEG trace as curr_signal from comb_data_norm

important: check if signal is present

For some participants and some trials, the signal you want to extract is actually not present. Check if the current trace is not equal to all zeros (implement a check to see if there is at least one sample that is not zero). If the current trial **is** all zeros, skip the FFT step for this trial. The powerspectrum should remain NaN for this trial (and all windows in it)

if there is a signal present

This is the signal we will analyse in consecutive windows inside the fourth loop.

- Loop over nWindows
- Define curr_start and curr_stop as the sample indices for the current window
- Extract the data from the current window as curr_window
  - check the length: is it indeed as long as N?

Finally, still inside this third nested loop, adapt and perform the FFT analysis from the examples shown in the notebooks.

- run the FFT
- store the powerspectrum with the correct transformation from the Fourier Coefficients as tmp_ps

This gives you a tmp_ps variable (the current power spectrum) with size 1 x nFreq. Save this powerspectrum in its appropriate place in the TFR_5D matrix.

In [18]:

```
print(nTrials)
print(nChans)
print(nParts)

# calculate the baseline spectrum
curr_signal_base = signal[:N_window_samples] # extract the current signal
spectrum_base = fft.rfft(curr_signal_base) # the :N is not needed here, but included for clarity
spect_freqs_base = fft.rfftfreq(int(N_window_samples), 1/srate) # N_window_samples is the length normalis
amps_base = np.abs(spectrum_base) / N_window_samples *2
print(comb_data_norm_control.shape)
print(N_window_samples)
```

```
22
59
21
(22, 59, 751, 21)
250
```

In [19]:

```
for trials in range(nTrials):
    for chan in range(nChans):
        for participants in range(nParts):
            curr_signal=comb_data_norm_control[trials, chan,:,participants]
            for window in range(nWindows):
                curr_start = window_starts[window]
                curr_stop = window_ends[window]
            #    curr_signal_slice = curr_signal[int(curr_start):int(curr_stop)]
```

```
            curr_window = comb_data_norm_control[trials, chan, int(curr_start):int(curr_stop), partic

            # create fourier spectrum
            spectrum_slice = fft.rfft(curr_window) # the :N is not needed here, but included for clar
            spect_freqs_slice = fft.rfftfreq(int(N_window_samples), 1/srate) # N_window_samples is th
            amps_slice = np.abs(spectrum_slice) / N_window_samples *2

            # in our case there is no missing signal, so we commented out the if statement to make ti
            #if not np.all(curr_signal_slice==0):
            #   powerspectrum=np.zeros((nTrials, nChans, nWindows, len(nFreq), nParts))

            tmp_ps_control = amps_slice / amps_base
            TFR_5D_control[trials,chan,window,:,participants] = tmp_ps_control
            tmp_ps = amps_slice

            TFR_5D_control[trials,chan,window,:,participants] = tmp_ps
```

In [20]:

```
print("Shape of the control powerspectrum:",tmp_ps_control.shape)
print("Shape of the TFR control:",TFR_5D_control.shape)
```

```
Shape of the control powerspectrum: (126,)
Shape of the TFR control: (22, 59, 51, 126, 21)
```

In [21]:

```
for trials in range(nTrials):
    for chan in range(nChans):
        for participants in range(nParts):
            curr_signal=comb_data_norm_experimental[trials, chan,:,participants]
            for window in range(nWindows):
                curr_start = window_starts[window]
                curr_stop = window_ends[window]
                curr_signal_slice = curr_signal[int(curr_start):int(curr_stop)]

                spectrum_slice = fft.rfft(curr_signal_slice) # the :N is not needed here, but included fo
                spect_freqs_slice = fft.rfftfreq(int(N_window_samples), 1/srate) # N_window_samples is th
                amps_slice = np.abs(spectrum_slice) / N_window_samples *2

                #if not np.all(curr_signal_slice==0):
                #    powerspectrum=np.zeros((nTrials, nChans, nWindows, len(nFreq), nParts))

                tmp_ps_exp = amps_slice / amps_base
                TFR_5D_exp[trials,chan,window,:,participants] = tmp_ps_exp

                #print("start at sample", iStart,"running through", iEnd, "; current signal length:", le
```

In [22]:

```
print("Shape of the powerspectrum:",tmp_ps_exp.shape)
print("Shape of the TFR experimental:",TFR_5D_exp.shape)
```

```
Shape of the powerspectrum: (126,)
Shape of the TFR experimental: (22, 59, 51, 126, 21)
```

Great job! You have now constructed the full time-resolved powerspectrum, across participants, trials and channels for your datasets! In the next part, we will explore these powerspectra and apply one more normalization step before making those nice time-resolved powerspectra or TFRs that go on the fridge.

Q7 - Extracting the averaged baseline powerspectrum

In this step, you will extract the baseline powerspectrum. In order to do this, you must select the baseline period as the set of one or more of those windows that start from -0.5s and run up to the window that ends at 0s (stimulus presentation), and average the fourier spectra across these windows (e.g. in the time domain). By doing this, we reduce the time-resolved powerspectrum for the baseline period to a single (average) baseline powerspectrum. We retain the dimensions trial, channel and participant, but just collapse (=average) over time. This means your baseline powerspectrum is defined as: nTrials x nChans x nFreq x nPart

- Set base_window_start to 0, as we start from the first window.
- Using code, set base_window_end as the window that ends just before or at t=0s in the time domain

- Extract the set of windows base_window_start:base_window_end (end-inclusive) from the TFR_5D arrays

- Average across the time dimension if needed
- Store the resulting arrays as TFR*base*(control/experimental)
  - this should yield a 4D-matrix with size nTrials x nChans x nFreq x nParticipants)
  - check the shapes

In [23]:

```
base_window_start = 0
base_window_end = np.arange(start = base_window_start, stop = TFR_5D_control.shape[2]/3, step = 1)
```

```
# Change the value types to be used in np.matrix

base_window_end=np.asarray(base_window_end, dtype = int)
#print(base_window_end)

# Subset the pre-stimulus time for preperation of averaging the powerspectrum for both control and exper

TFR_control_baseline = TFR_5D_control[:,:,base_window_end,:,:,]
TFR_exp_baseline = TFR_5D_exp[:,:,base_window_end,:,:,]

print("Shape of pre-stimuli control: ",TFR_control_baseline.shape)
print("Shape of pre-stimuli control: ",TFR_exp_baseline.shape)

# average the baseline period, this will reduce the shape of the TFR_5D array to nTrials x nChans x nFre

TFR_control_base = np.mean(TFR_control_baseline,axis =2)
TFR_exp_base = np.mean(TFR_exp_baseline,axis = 2)

# Print shape of the TFR for control and experiment
print("Shape of TFR_control_base: ",TFR_control_base.shape)
print("Shape of TFR_exp_base: ",TFR_exp_base.shape)
```

```
Shape of pre-stimuli control:  (22, 59, 17, 126, 21)
Shape of pre-stimuli control:  (22, 59, 17, 126, 21)
Shape of TFR_control_base:  (22, 59, 126, 21)
Shape of TFR_exp_base:  (22, 59, 126, 21)
```

Q8 - visualize the baseline powerspecrtum

To visualizing the average baseline powerspectrum, you will start with a single channel and single participant, and plot the powerspectrum for all trials. Let's take channel Fz as an example. Here is the list of channels (run it from the next code cell).

- construct an index variable "Fz_idx" that holds the index number for the "Fz" channel
- subset TFR_control_base to extract the nTrial x nFreq x nParticipants array for channel Fz. Save as Fz_ps
- select a random participant, subset to retain the nTrial x nFreq powerspectrum array for this person in this channel

In [25]:

```
import random as rd
ch_names = ['Fpz', 'Fp1', 'Fp2', 'AFz', 'AF3', 'AF4', 'AF7', 'AF8', 'Fz', 'F1', 'F2', 'F3', 'F4', 'F5', 'F
print(len(ch_names))

# Get Fz channel
Fz_idx = ch_names.index('Fz')

# We are only looking at the Fz channel

Fz_ps = TFR_control_base[:,Fz_idx,:,:]
print("Shape of Fz_ps: ",Fz_ps.shape)

#we are only looking for one participant
randpar=rd.randint(0,20)
Fz_ps_rand = Fz_ps[:,:,randpar]
print("Which participant: ",randpar)
print("Shape of Fz ps random subject: ",Fz_ps_rand.shape)
```

```
59
Shape of Fz_ps:  (22, 126, 21)
Which participant:  16
Shape of Fz ps random subject:  (22, 126)
```

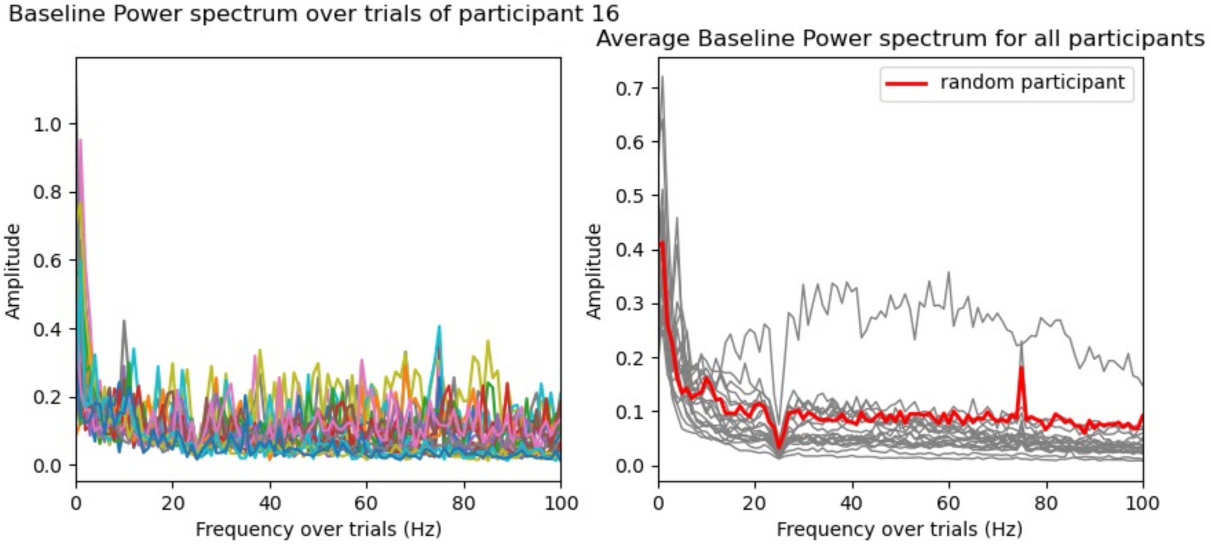
Next, you will plot the baseline powerspectrum

- Create a 1x2 subplot called fig_Q8A, with on the left the baseline power spectrum for each trials as a separate line
- On the right, plot the average baseline power spectrum for this participant and all participants:
  - calculate the average (use nanmean!) over trials per participant, for all participants. Save as Fz_ps_av
  - plot the averaged baseline power spectra (over trials) for each participant, in grey
  - plot the average baseline power spectrum for your participant againg in color
  - limit the frequency axis for both plots to 100Hz
  - include a legend label that highlights which participant is in red
  - add proper plot annotations

In [51]:

```
fig_Q8A, ax = plt.subplots(figsize=(10,4), nrows=1, ncols=2) # 1x2 graph

# Graph showing the baseline powerspectrum for participant 12
for i in range(nTrials-1):
    ax[0].plot(Fz_ps_rand[i,:])
```

```python
ax[0].set_xlim([0, 100])
ax[0].set_xlabel("Frequency over trials (Hz)")
ax[0].set_ylabel("Amplitude")
ax[0].set_title("Baseline Power spectrum over trials of participant {} \n".format(randpar))

# Average the trials for all participants
Fz_ps_av = np.nanmean(Fz_ps,axis =0)
print(Fz_ps_av.shape)

#Graph showing the average baseline power spectrum for all the participants
ax[1].set_xlim([0, 100])
for i in range(nTrials-1):
    ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
ax[1].plot(Fz_ps_av[:, randpar], color = "red", LineWidth = 2, label = "random participant")
ax[1].set_xlabel("Frequency over trials (Hz)")
ax[1].set_ylabel("Amplitude")
ax[1].legend()
ax[1].set_title("Average Baseline Power spectrum for all participants")
plt.show()
```




```
(126, 21)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
```

```
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:19: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:,i],color = "grey", LineWidth = 1)
<ipython-input-51-e2d0e89d745e>:20: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(Fz_ps_av[:, randpar], color = "red", LineWidth = 2, label = "random participant")
```

As you can see from fig Q8A, there are still substantial differences in the overall powerspectrum power between the lines representing different trials, and in lines from different participants. This is because we have normalized the EEG traces within participants by their standard deviation, but not necessarily between participants. Some participants might just show more beta power (for example) than others. Rather, we are interested in power *changes* in comparison to a baseline. We will consider, for each trial, the extracted baseline powerspectrum that you just extracted, and normalize the power spectrum of all windows that fall into the evoked period by this baseline spectrum. Any frequencies with *more* power than the baseline will have values above 1, frequencies with *less* power will have values lower than 1.

But first, just to visualize the dynamics of the powerspectra over all participants, we will peak-normalize them. This should bring all powerspectra into the same range ([0,1]).

- Reuse the Fz_ps_av array (average Fz powerspectrum, per participant).
- use np.nanmax to get the maximum value per participant.
  - Think about the correct axis
  - save as max_pow (with a shape of (nPart,))
- peak-normalize (=divide) the powerspectrum for each participant by their max value using this line:
  - Fz_ps_av_peaknorm = Fz_ps_av / np.tile(max_pow, [nFreq,1])

np.tile expands the max. power value for each participant power spectrum out to an array with nFreq x nPart size

- Create a 1x2 subplot called fig_Q8B
- Now plot the peak-normalized Powerspectra again in figure Q8B. We can see the familiar 1/f pink noise pattern appear

Finally, we will plot the **average** peak-normalised spectrum with its 90% confidence interval. With about 20+ participants (the number varies between datasets), the 90% confidence interval of the power in each frequency is approximated by taking the 3rd lowest and 3rd highest values as the confidence bands. This excludes the two highest and two lowest values per frequency.

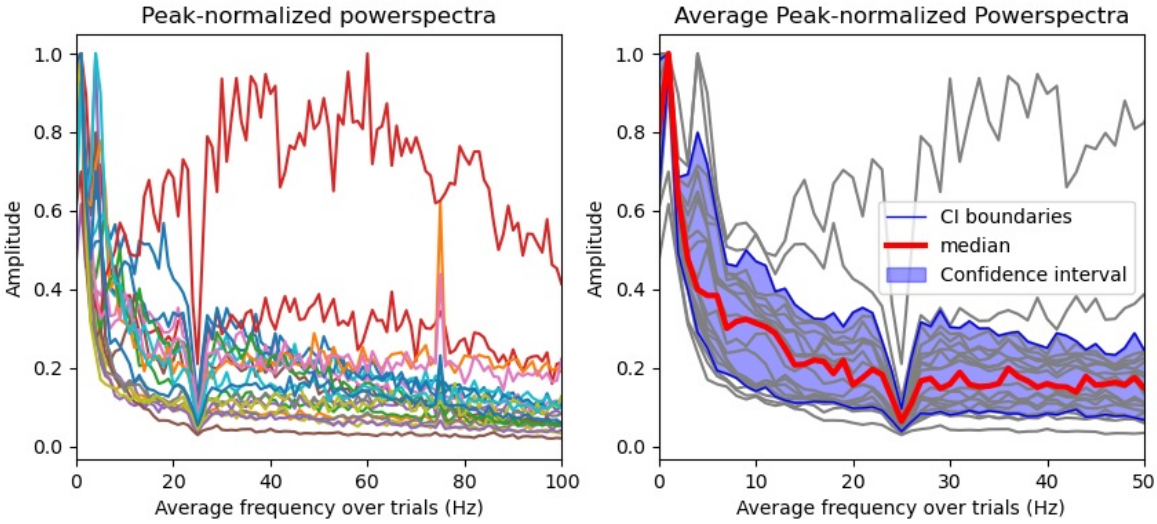
- sort the datapoints of Fz_ps_av_peaknorm per frequency.
  - NB this breaks the connection between normalised power values for different frequencies within a participant
- save the 3rd lowest values as ci_low (size: nFreq x 1)
- save the 3rd highest values as ci_high (size: nFreq x 1)
- use fill_between from matplotlib (ax.fill_between) to plot the confidence band of the peak-normalised powerspectrum
  - plot in the right panel of Q8B
  - choose your own color for the fill
  - add a thicker line for the **median** value, with a plot label
  - limit to 50Hz
- complete plot with proper annotation
- save as Figure2 for your report

In [52]:

```
print(Fz_ps_av.shape)

#  get the maximum power spectrum value per participant
max_pow = np.nanmax(Fz_ps_av,axis = 0)
print(max_pow.shape)
```

```
#Subset the prestimuli period
nFreq = TFR_5D_control.shape[2]/3
nFreq = int(nFreq)
print(nFreq)


# Brings powerspectra to range [0,1]
Fz_ps_av_peaknorm = Fz_ps_av / np.tile(max_pow, [126,1])
print("Fz peaknorm",Fz_ps_av_peaknorm.shape)

Fz_ps_av_peaknorm_sort = np.sort(Fz_ps_av_peaknorm, axis = 1)

print("After sort",Fz_ps_av_peaknorm_sort.shape)

#print(Fz_ps_av_peaknorm_sort[0,:])

ci_low = Fz_ps_av_peaknorm_sort[:,3]
print(ci_low.shape)
ci_high = Fz_ps_av_peaknorm_sort[:,nParts-3]
print(ci_high.shape)

median = np.zeros(Fz_ps_av_peaknorm.shape[0])

print("Fz_ps_av_peaknorm shape", Fz_ps_av_peaknorm.shape)
for x in range(Fz_ps_av_peaknorm.shape[0]):
    median[x]= np.median(Fz_ps_av_peaknorm_sort[x, :])

fig_Q8B, ax = plt.subplots(figsize=(10,4), nrows=1, ncols=2)

ax[0].plot(Fz_ps_av_peaknorm)
ax[0].set_xlim([0, 100])
ax[0].set_xlabel("Average frequency over trials (Hz)")
ax[0].set_ylabel("Amplitude")
ax[0].set_title("Peak-normalized powerspectra")

ax[1].plot(Fz_ps_av_peaknorm_sort, color = 'grey')
ax[1].set_xlim([0, 50])
ax[1].plot(ci_low, linewidth = 1, color = 'blue', label="CI boundaries")
ax[1].plot(ci_high, linewidth = 1, color = 'blue')
ax[1].fill_between(range(Fz_ps_av_peaknorm_sort.shape[0]),ci_low, ci_high, alpha = 0.4,color = 'blue', lab
ax[1].plot(median, linewidth = 3, color = 'red', label = 'median')
ax[1].legend()
ax[1].set_xlabel("Average frequency over trials (Hz)")
ax[1].set_ylabel("Amplitude")
ax[1].set_title("Average Peak-normalized Powerspectra")



plt.show()

Figure2 = fig_Q8B
```

```
(126, 21)
(21,)
17
Fz peaknorm (126, 21)
After sort (126, 21)
(126,)
(126,)
Fz_ps_av_peaknorm shape (126, 21)
```



Q9 - applying baseline normalization

To properly implement baseline normalization of the powerspectra, we use our calculated powerspectra over the baseline period as a normalization factor for each individual combination of channel and participants (as power can vary between channels, and between individuals, as we have seen). So, basically we normalise powerspectra for the windows that correspond to the evoked period (for each of these combinations separately) all by their own baseline powerspectrum. So you will be normalising only *within* a trial.

First, you need to figure out the window *index* of the the first window that starts at t=0s. Starting from this window and including all following windows will be the included in the normalisation.

- subset TFR*5D*(control/experimental) to TFR*evoked*(control/experimental)
- pre-allocate an array called TFR_evoked_norm with the same size
  - first with zeros, then add np.nan

To normalise the evoked powerspectra by the baseline powerspectra, you can either:

- run a loop across participants, channels, trials and windows
  - and do the normalization per window OR:
- use np.tile and np.newaxis to expand the baseline powerspectrum out to the same size as the evoked powerspectrum array
  - and then do element-wise division

In [28]:

```
print("Orginal shape of TFR:",TFR_5D_control.shape)
nWindow_baseline = TFR_5D_control.shape[2]/3
nWindow_baseline = int(nWindow_baseline)

#Get the envoked windows

nWindow_evoked = TFR_5D_control.shape[2]-nWindow_baseline
nWindow_evoked = int(nWindow_evoked)

TFR_evoked_norm = np.zeros((TFR_5D_control.shape[0],TFR_5D_control.shape[1],nWindow_evoked,TFR_5D_control
TFR_evoked_norm[:] = np.nan
print("Shape of TFR_evoked_norm:", TFR_evoked_norm.shape)


# Subset the evoked windows into the new matrixes

TFR_evoked_control = TFR_5D_control[:,:,nWindow_baseline:,:,:,]
TFR_evoked_exp = TFR_5D_exp[:,:,nWindow_baseline:,:,:,]
print("Shape of TFR_evoked_control:",TFR_evoked_control.shape)
print("Shape of TFR_evoked_exp:",TFR_evoked_exp.shape)

TFR_evoked_control_norm = np.zeros((TFR_5D_control.shape[0],TFR_5D_control.shape[1],nWindow_evoked,TFR_5D
TFR_evoked_control_norm[:] = np.nan
print("Shape of TFR_evoked_norm:", TFR_evoked_control_norm.shape)
```

```
TFR_evoked_exp_norm = np.zeros((TFR_5D_control.shape[0],TFR_5D_control.shape[1],nWindow_evoked,TFR_5D_con
TFR_evoked_exp_norm[:] = np.nan
print("Shape of TFR_evoked_norm:", TFR_evoked_exp_norm.shape)
```

```
Orginal shape of TFR: (22, 59, 51, 126, 21)
Shape of TFR_evoked_norm: (22, 59, 34, 126, 21)
Shape of TFR_evoked_control: (22, 59, 34, 126, 21)
Shape of TFR_evoked_exp: (22, 59, 34, 126, 21)
Shape of TFR_evoked_norm: (22, 59, 34, 126, 21)
Shape of TFR_evoked_norm: (22, 59, 34, 126, 21)
```

In [29]:

```
# Loop through participants, channels, trials and windows and normalize the new matrix by dividing each .

for part in range(nParts):
    for chan in range(nChans):
        for trial in range(nTrials):
            for wind in range(nWindow_evoked):
                TFR_evoked_control_norm[trial, chan, wind, :, part] = TFR_evoked_control[trial, chan, win
```

```
<ipython-input-29-131394c8d1b2>:7: RuntimeWarning: divide by zero encountered in true_divide
  TFR_evoked_control_norm[trial, chan, wind, :, part] = TFR_evoked_control[trial, chan, wind, :, part] /
TFR_5D_control[trial, chan, 0, :, part]
```

In [30]:

```
print(TFR_evoked_control_norm.shape)
```

```
(22, 59, 34, 126, 21)
```

In [31]:

```
# Loop through participants, channels, trials and windows and normalize the new matrix by dividing each .

for part in range(nParts):
    for chan in range(nChans):
        for trial in range(nTrials):
            for wind in range(nWindow_evoked):
                TFR_evoked_exp_norm[trial, chan, wind, :, part] = TFR_evoked_exp[trial, chan, wind, :, pa
```

```
<ipython-input-31-f57c24d3398f>:7: RuntimeWarning: divide by zero encountered in true_divide
  TFR_evoked_exp_norm[trial, chan, wind, :, part] = TFR_evoked_exp[trial, chan, wind, :, part] / TFR_5D_e
xp[trial, chan, 0, :, part]
```

Q10 Visualize the time-resolved, normalized TFR for a single channel

Now that both TFR_evoked_control and TFR_evoked_exp have been normalized to TFR*evoked*(control/exp_)norm, we can once again plot from the randomly selected channel Fz.

- From the normalised TFRs, extract the time-resolved powerspectrum for chan Fz
  - store into Fz_evoked_exp and Fz_evoked_control
  - verify that the shape is nTrials x nWindows x nFreq x nParts

The values in these arrays represent normalised power values (ratios) for each time x frequency pair relative to their own baseline. As such, they can be larger than 1 (more power than in baseline) or smaller than 1 (less power than in baseline). To make that scale symmetric and linear, we can apply the log10 function to the averaged powerspectrum.

- use np.log10 to convert the power ratios to a logarithmic scale
  - this can be applied to the entire array at once
  - store as Fz_evoked_exp_log and Fz_evoked_control_log
- take the average across the trial dimension
  - be careful of missing values (NaN) -> use np.nanmean
  - store as Fz_TFR_exp, Fz_TFR_control

To visualize this normalized Fz-powerspectrum for a single participant, we first need the appropriate time axis, and next some index into the evoked part of the TFR. Let's take the first participant as an example. Construct the time axis as a vector that has the starting points of the windows in the evoked period in seconds, not samples.

- Extract the window start samples correspondingt to the evoked period as vector of 26 samples
- Index into the time array to find the starting points of theses windows in seconds
  - check: the first evoked windows starts at 0s.
  - store as time_evoked

Now you can extract the normalized Fz-powerspectrum for the first participant (this should be an nWindow_evoked x nFreq array) **as a matrix** (worksheet 7), and plot against time_evoked. You can use the function **pcolormesh** from matplotlib for this. The first argument to this function is the time (x-axis), the second the frequencies (y-axis) and the third is the TFR (z-axis: color scale). We need handles to these to plot items for the colorbars, that is what the 'ctrl' and 'exp' stand for. Do not remove them!

- Plot the normalised TFRs for the evoked period
  - plot in a 1x2 subplot with the control data (left) and experimental data (right)
  - Apply appropriate annotation and formatting
  - Store as fig_Q10A
  - Evaluate the colormap and colorbar functions as indicated.
  - adjust vmin and vmax to the power in the frequencies above 0 - the DC offset can vary quite a bit
    - ensure that they are symmetric around zero - the green colors will indicate 0 (i.e. no change)

Depending on your particular task conditions, you might see some postive (i.e. larger than baseline) activity in the Beta or low Gamma Bands (around 30Hz).

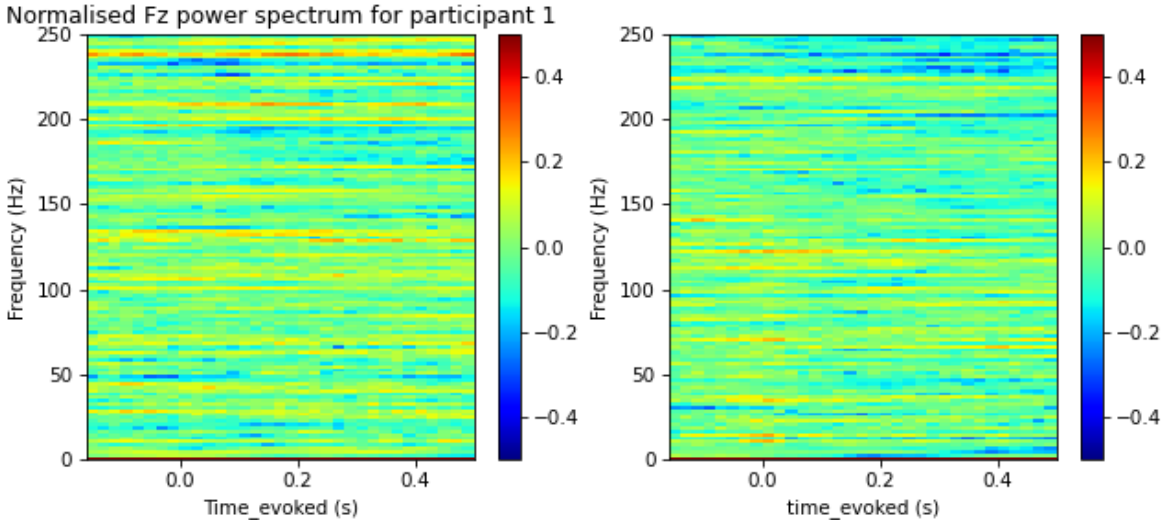
In [41]:

```
# extract Fz channel
Fz_idx = ch_names.index('Fz')
Fz_evoked_exp = TFR_evoked_exp_norm[:, Fz_idx, :, :, :]
Fz_evoked_control = TFR_evoked_control_norm[:, Fz_idx, :, :, :]

print("Shape of Fz_evoked_exp", Fz_evoked_exp.shape)
print("Shape of Fz_evoked_control", Fz_evoked_control.shape)

# log10 normalisation
Fz_evoked_exp_log = np.log10(Fz_evoked_exp)
Fz_evoked_control_log = np.log10(Fz_evoked_control)

# average across trials, select first participant
#for participant in range(Fz_evoked_exp_log.shape[3])


Fz_TFR_exp = np.nanmean(Fz_evoked_exp_log, axis=0)
Fz_TFR_control = np.nanmean(Fz_evoked_control_log, axis=0)

print("Shape of average exp", Fz_TFR_exp.shape)
print("Shape of average control", Fz_TFR_control.shape)

part1_exp = Fz_TFR_exp[:,:,1]
part1_control = Fz_TFR_control[:,:,1]

print("Shape of participant 1", part1_exp.shape)
print("Shape of participant 1",part1_control.shape)
# set up time_evoked
```

```
'''
srate = 500
num_samples = part1_exp.shape[0]
num_seconds = 1
samples_milliseconds = num_samples / (srate * 1000)
samples_time = (srate * 1000) * num_seconds
print(samples_milliseconds)

#print(window_starts)
window_starts_post = window_starts[17:]       # Sample Indices Vector
#print(window_starts_post)
print(len(window_starts_post))
Fs = 500;             # Sampling Frequency (Hz)
t = window_starts_post/srate;         # Time Vector (seconds)
print(type(window_starts_post))
window_starts_post=np.asarray(window_starts_post, dtype = int)

print(window_starts_post)
window_start = []
for i in range(len(window_starts_post)):
    window_start.append(window_starts_post[i]-170)
print(window_start)
#print(t)
window_start = np.array(window_start)
t = window_start/srate;
print(t)
'''

#Convert the samples to time_evoked

starting_windows = (window_starts[51-34:])
print(starting_windows)

time_evoked = np.zeros((0))
times = ()
for x in starting_windows:
    print(time[int(x)])
    time_evoked = np.append(time_evoked, time[int(x)])

#time_evoked = ...

spect_freqs_base = fft.rfftfreq(int(N_window_samples), 1/srate)

# Plot grpah showing the normalized Fz-powerspectrum for the first participant
fig_Q10, ax = plt.subplots(figsize=(10,4), nrows=1, ncols=2) # 1x2 graph

ctrl = ax[0].pcolormesh(time_evoked, spect_freqs_base,  part1_control.transpose(), vmin=-0.5, vmax=0.5, c
exp = ax[1].pcolormesh(time_evoked, spect_freqs_base, part1_exp.transpose(), vmin=-0.5, vmax=0.5, cmap='j

# plot the colorbar items linked to the pcolormesh objects next to their plots
fig_Q10.colorbar(ctrl, ax=ax[0])
fig_Q10.colorbar(exp, ax=ax[1])

ax[0].set_title("Normalised Fz power spectrum for participant 1")
ax[0].set_xlabel("Time_evoked (s)")
ax[0].set_ylabel("Frequency (Hz)")
ax[1].set_xlabel("time_evoked (s)")
ax[1].set_ylabel("Frequency (Hz)")
plt.show()
```

```
Shape of Fz_evoked_exp (22, 34, 126, 21)
Shape of Fz_evoked_control (22, 34, 126, 21)
Shape of average exp (34, 126, 21)
Shape of average control (34, 126, 21)
Shape of participant 1 (34, 126)
Shape of participant 1 (34, 126)
[170. 180. 190. 200. 210. 220. 230. 240. 250. 260. 270. 280. 290. 300.
 310. 320. 330. 340. 350. 360. 370. 380. 390. 400. 410. 420. 430. 440.
 450. 460. 470. 480. 490. 500.]
-0.1599999999999997
-0.13999999999999968
-0.11999999999999966
-0.09999999999999964
-0.07999999999999963
-0.05999999999999961
-0.03999999999999959
-0.019999999999999574
```

```
4.440892098500626e-16
0.020000000000000462
0.040000000000000048
0.060000000000000005
0.080000000000000052
0.10000000000000053
0.12000000000000055
0.14000000000000057
0.16000000000000006
0.18000000000000006
0.20000000000000062
0.22000000000000064
0.24000000000000066
0.26000000000000007
0.28000000000000007
0.30000000000000007
0.32000000000000073
0.34000000000000075
0.36000000000000076
0.38000000000000008
0.40000000000000008
0.42000000000000008
0.44000000000000083
0.46000000000000085
0.48000000000000087
0.50000000000000009
<ipython-input-41-73797c4b68e4>:10: RuntimeWarning: divide by zero encountered in log10
  Fz_evoked_exp_log = np.log10(Fz_evoked_exp)
<ipython-input-41-73797c4b68e4>:11: RuntimeWarning: divide by zero encountered in log10
  Fz_evoked_control_log = np.log10(Fz_evoked_control)
```




```
<ipython-input-41-73797c4b68e4>:76: MatplotlibDeprecationWarning: shading='flat' when X and Y have the s
ame dimensions as C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X
and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will beco
me an error two minor releases later.
  ctrl = ax[0].pcolormesh(time_evoked, spect_freqs_base,  part1_control.transpose(), vmin=-0.5,
vmax=0.5, cmap='jet')
<ipython-input-41-73797c4b68e4>:77: MatplotlibDeprecationWarning: shading='flat' when X and Y have the s
ame dimensions as C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X
and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will beco
me an error two minor releases later.
  exp = ax[1].pcolormesh(time_evoked, spect_freqs_base, part1_exp.transpose(), vmin=-0.5, vmax=0.5,
cmap='jet')
```

Q11 Averaged Power Spectrum Contrast across participants.

Next, we will average the normalized powerspectrum across participants, to look for consistent frequency responses between individuals. We will zoom in to the contrast between control and experimental conditions in this exercise. Based on the qualitatively interesting parts of this contrast, you will extract the power constrast in a certain spectral band (delta, theta, alpha, beta, gamma (low/high) of your choice and visualise the spectral band values.

- Repeat the steps for Q10, first averaging within a participant across trials
  - separately per condition
- Next, subtract the TFR for the control condition from the experimental condition
  - subtract within each participant
- Then, average these condition contrasts across participants
  - This should also yield an nWindow_evoked * nFreq array.
  - Store as Fz_TFR_contrast_av
- Finally, create a graph that shows the contrast averaged across participants
  - Plot into the first subplot of Fig_Q11
- Extract the average power contrast in a frequency band of your choice
  - see top of the notebook for the frequency cutoffs.
  - it is up to you to include/exclude a certain frequency in the band (experiment!)
  - average the spectral power contrast across the included frequencies
  - delineate a number of time windows where your effect of interest shows up
    - average the spectral band power contrast over these time windows
  - plot a histogram, boxplot or violin plot showing the distribution of these values in the second subplot
    - These data are, eventually, 1xnParts
    - store as band_power
  - add proper annotation and labelling to both subplots
- Save this Figure as your Figure3 for your report

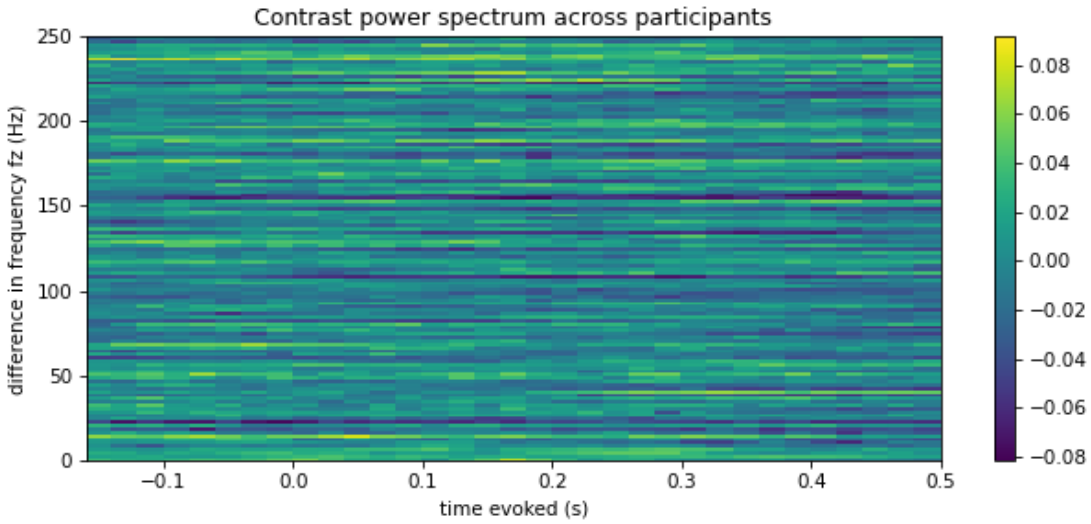
optional

You can assess the significance of this contrast distribution by comparing it to a population mean of zero. Use Scipy.Stats to apply the appropriate one-sample test for this hypothesis.

In [42]:

```
#Get tht mean of the trials per participant

Fz_TFR_exp = np.nanmean(Fz_evoked_exp_log, axis=0)
Fz_TFR_control = np.nanmean(Fz_evoked_control_log, axis=0)
print("Shape of averaged trial per participant(Control)", Fz_TFR_control.shape)
print("Shape of averaged trial per participant(exp) ",Fz_TFR_exp.shape)

Fz_TFR_contrast = np.zeros((Fz_TFR_control.shape[0], Fz_TFR_control.shape[1], Fz_TFR_control.shape[2]))

#Loop through window, frequency and participant and get the difference between the control and the exper

for window in range (Fz_TFR_control.shape[0]):
    for freq in range (Fz_TFR_control.shape[1]):
        for part in range (Fz_TFR_control.shape[2]):
            Fz_TFR_contrast[window, :, part] = Fz_TFR_control[window, :, part] - Fz_TFR_exp[window, :, pa

#Get the average of al the participants

print(Fz_TFR_contrast.shape)
Fz_TFR_contrast_av = np.nanmean(Fz_TFR_contrast, axis = 2)



fig_Q11, ax = plt.subplots(figsize=(10,4))
contrast_plot = ax.pcolormesh(time_evoked, spect_freqs_base, np.transpose(np.asmatrix(Fz_TFR_contrast_av)


fig_Q11.colorbar(contrast_plot, ax=ax)

plt.title("Contrast power spectrum across participants")
plt.xlabel("time evoked (s)")
plt.ylabel("difference in frequency fz (Hz)")
# optional
import scipy.stats as stats
```

```
Shape of averaged trial per participant(Control) (34, 126, 21)
Shape of averaged trial per participant(exp)  (34, 126, 21)
<ipython-input-42-8bd9d86db3f1>:15: RuntimeWarning: invalid value encountered in subtract
  Fz_TFR_contrast[window, :, part] = Fz_TFR_control[window, :, part] - Fz_TFR_exp[window, :, part]
(34, 126, 21)
```




```
<ipython-input-42-8bd9d86db3f1>:25: MatplotlibDeprecationWarning: shading='flat' when X and Y have the s
ame dimensions as C is deprecated since 3.3.  Either specify the corners of the quadrilaterals with X
and Y, or pass shading='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].  This will beco
me an error two minor releases later.
  contrast_plot = ax.pcolormesh(time_evoked, spect_freqs_base,
np.transpose(np.asmatrix(Fz_TFR_contrast_av)))
```

Q12 - open figure creation with frequency band subsetting & confidence interval

For this last exercise, you should combine the different steps taken before to create a new plot or series of plots. You are free to explore different channels, create TFRs and contrast TFRs to find an interesting combination of channel and frequency band. Your final plot should show:

- the timecourse of normalised power for the control and experimental condition
  - for a given spectral power band OR
- the timecourse of the experimental-control contrast
  - for a given spectral power band

In both cases, plot:

- the mean across participants
- the upper and lower confidence interval on these values
  - Reuse the gist of the code introduced before

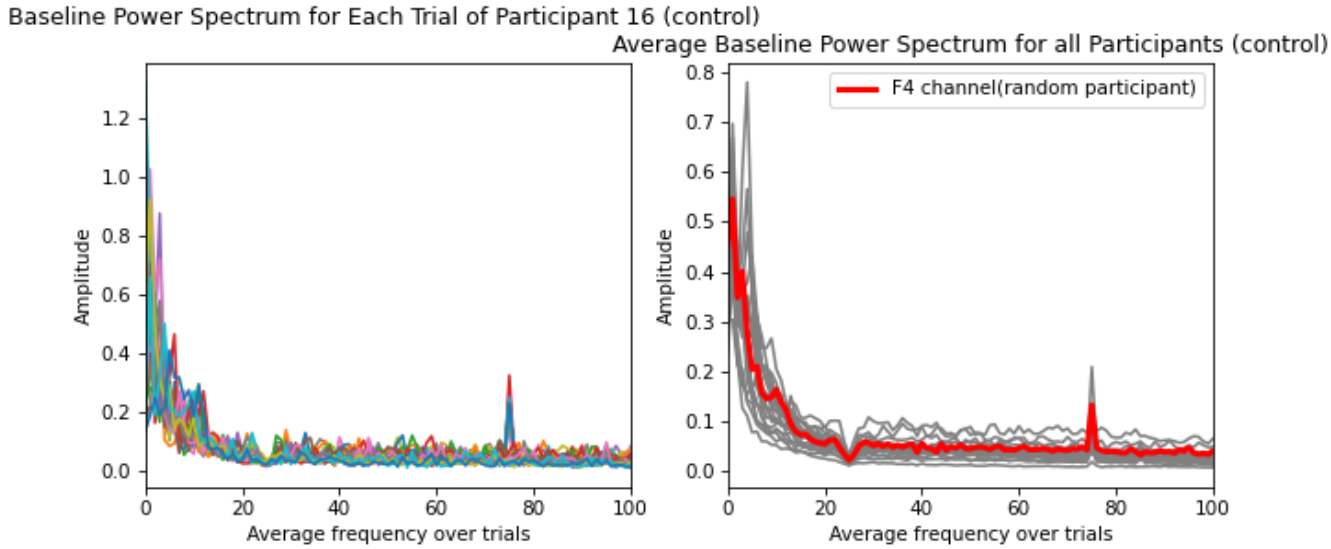
This should result in a timeseries with its confidence interval, tracking the evolution of average spectral power in a given band over the evoked period of the trial. Add appropriate annotation and labeling and save fig_Q12 as Figure4 for you report.

In [44]:

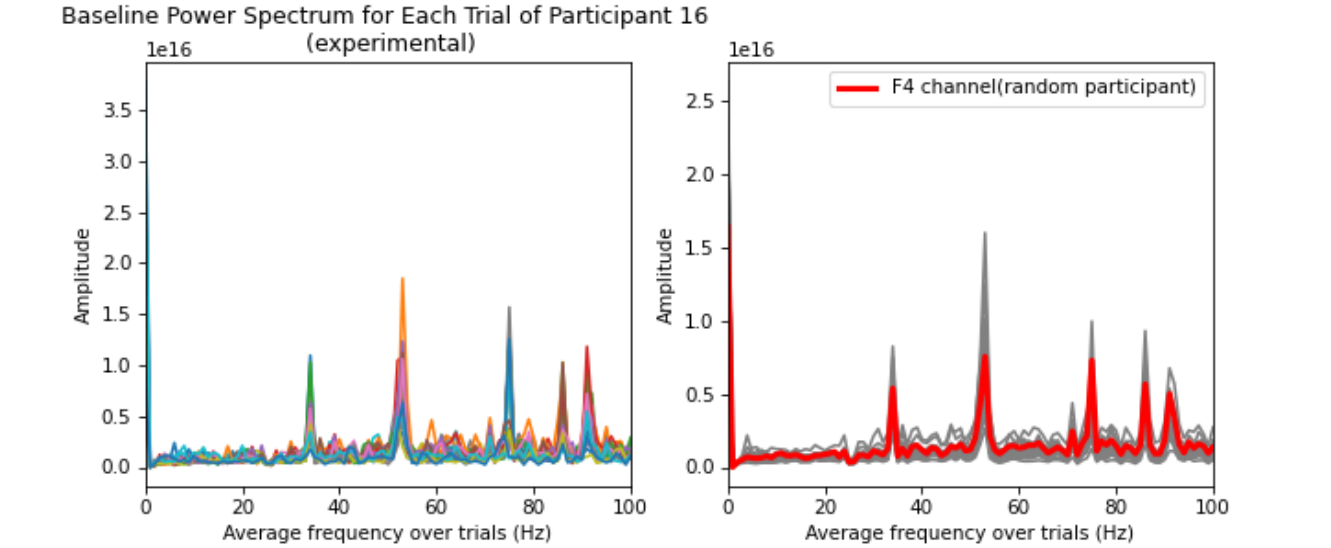
```
import random as rd
ch_names = ['Fpz', 'Fp1', 'Fp2', 'AFz', 'AF3', 'AF4', 'AF7', 'AF8', 'Fz', 'F1', 'F2', 'F3', 'F4', 'F5', 'F
print(len(ch_names))

# Get F4 channel
F4_idx = ch_names.index('F4')
print(F4_idx)

# We are only looking at the Fz channel

F4_ps = TFR_control_base[:,F4_idx,:,:]
print("Shape of F4_ps: ",F4_ps.shape)

F4_ps_rand = F4_ps[:,:,randpar]
print("Which participant: ",randpar)
print("Shape of F4 ps random subject: ",F4_ps_rand.shape)

fig_12_control, ax = plt.subplots(figsize=(10,4), nrows=1, ncols=2) # 1x2 graph

for i in range(nTrials-1):
    ax[0].plot(F4_ps_rand[i,:])
```

```
ax[0].set_xlim([0, 100])
ax[0].set_xlabel("Average frequency over trials")
ax[0].set_ylabel("Amplitude")
ax[0].set_title("Baseline Power Spectrum for Each Trial of Participant {} (control) \n".format(randpar))

F4_ps_av = np.nanmean(F4_ps,axis =0)
print(F4_ps_av.shape)

ax[1].set_xlim([0, 100])
for i in range(nTrials-1):
    ax[1].plot(F4_ps_av[:,i],color = "grey")
ax[1].plot(F4_ps_av[:, randpar], color = "red", LineWidth = 3, label = "F4 channel(random participant)")
ax[1].legend()
ax[1].set_xlabel("Average frequency over trials")
ax[1].set_ylabel("Amplitude")
ax[1].set_title("Average Baseline Power Spectrum for all Participants (control)")



plt.show()

fig_12_exp, ax = plt.subplots(figsize=(10,4), nrows=1, ncols=2) # 1x2 graph
F4_ps_exp = TFR_exp_base[:,F4_idx,:,:]
print("Shape of F4_ps: ",F4_ps_exp.shape)


F4_ps_rand_exp = F4_ps_exp[:,:,randpar]

print("Shape of F4 ps random subject: ",F4_ps_rand_exp.shape)


for i in range(nTrials-1):
    ax[0].plot(F4_ps_rand_exp[i,:])

ax[0].set_xlim([0, 100])
ax[0].set_xlabel("Average frequency over trials (Hz)")
ax[0].set_ylabel("Amplitude")
ax[0].set_title("Baseline Power Spectrum for Each Trial of Participant {} \n (experimental)".format(randp

F4_ps_av_exp = np.nanmean(F4_ps_exp,axis =0)
print(F4_ps_av_exp.shape)

ax[1].set_xlim([0, 100])
for i in range(nTrials-1):
    ax[1].plot(F4_ps_av_exp[:,i],color = "grey")
ax[1].plot(F4_ps_av_exp[:, randpar], color = "red", LineWidth = 3, label = "F4 channel(random participant
ax[1].legend()
ax[1].set_xlabel("Average frequency over trials (Hz)")
ax[1].set_ylabel("Amplitude")
plt.show()
```

```
59
12
Shape of F4_ps:  (22, 126, 21)
Which participant:  16
Shape of F4 ps random subject:  (22, 126)
```




```
(126, 21)
<ipython-input-44-a124f2ad9c05>:34: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(F4_ps_av[:, randpar], color = "red", LineWidth = 3, label = "F4 channel(random
participant)")
```



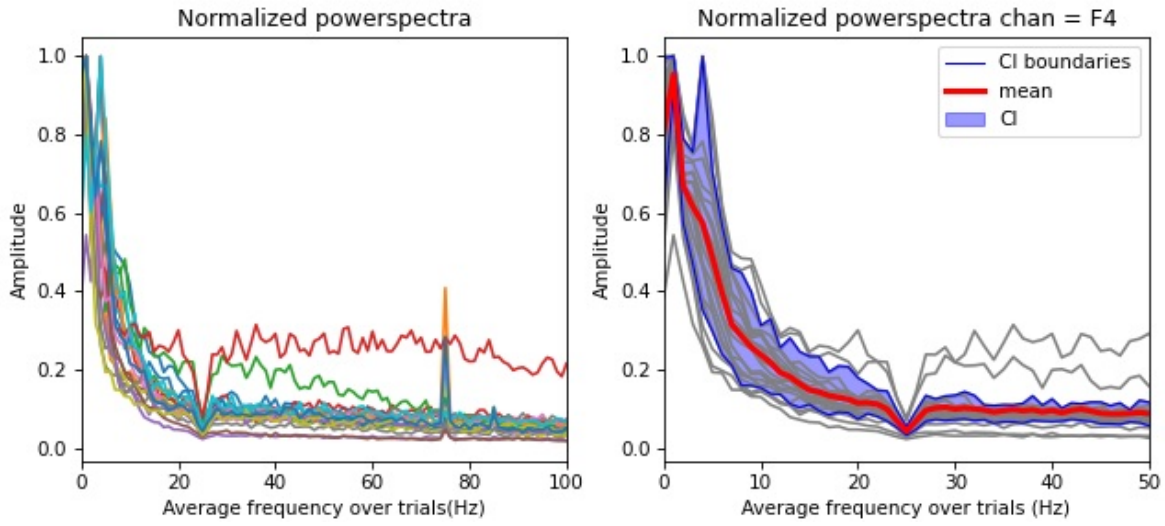

```
Shape of F4_ps:  (22, 126, 21)
Shape of F4 ps random subject:  (22, 126)
(126, 21)
<ipython-input-44-a124f2ad9c05>:68: MatplotlibDeprecationWarning: Case-insensitive properties were
deprecated in 3.3 and support will be removed two minor releases later
  ax[1].plot(F4_ps_av_exp[:, randpar], color = "red", LineWidth = 3, label = "F4 channel(random
participant)")
```

In [46]:

```
print(F4_ps_av.shape)
max_pow = np.nanmax(F4_ps_av,axis = 0)
print(max_pow.shape)

nFreq = TFR_5D_control.shape[2]/3
nFreq = int(nFreq)
print(nFreq)
# Brings powerspectra to range [0,1]
print(F4_ps_av.shape)
max_pow = np.nanmax(F4_ps_av,axis = 0)
print(max_pow.shape)

nFreq = TFR_5D_control.shape[2]/3
nFreq = int(nFreq)
print(nFreq)
# Brings powerspectra to range [0,1]
F4_ps_av_peaknorm = F4_ps_av / np.tile(max_pow, [126,1])
```

```
print("F4 peaknorm",F4_ps_av_peaknorm.shape)

F4_ps_av_peaknorm_sort = np.sort(F4_ps_av_peaknorm, axis = 1)

print("After sort",F4_ps_av_peaknorm_sort.shape)

#print(F4_ps_av_peaknorm_sort[0,:])

ci_low = F4_ps_av_peaknorm_sort[:,3]
print(ci_low.shape)
ci_high = F4_ps_av_peaknorm_sort[:,nParts-3]
print(ci_high.shape)

mean = np.zeros(F4_ps_av_peaknorm.shape[0])

print("F4_ps_av_peaknorm.shape shape", F4_ps_av_peaknorm.shape)
for x in range(F4_ps_av_peaknorm.shape[0]):
    mean[x]= np.mean(F4_ps_av_peaknorm_sort[x, :])

fig_Q12C, ax = plt.subplots(figsize=(10,4), nrows=1, ncols=2)

ax[0].plot(F4_ps_av_peaknorm)
ax[0].set_xlim([0, 100])
ax[0].set_xlabel("Average frequency over trials(Hz)")
ax[0].set_ylabel("Amplitude")
ax[0].set_title("Normalized powerspectra")

ax[1].plot(F4_ps_av_peaknorm_sort, color = 'grey')
ax[1].set_xlim([0, 50])
ax[1].plot(ci_low, linewidth = 1, color = 'blue', label="CI boundaries")
ax[1].plot(ci_high, linewidth = 1, color = 'blue')
ax[1].fill_between(range(F4_ps_av_peaknorm_sort.shape[0]),ci_low, ci_high, alpha = 0.4,color = 'blue', lab
ax[1].plot(mean, linewidth = 3, color = 'red', label = 'mean')

ax[1].set_xlabel("Average frequency over trials (Hz)")
ax[1].set_ylabel("Amplitude")

ax[1].set_title("Normalized powerspectra chan = F4")
ax[1].legend()
plt.show()
```

```
(126, 21)
(21,)
17
(126, 21)
(21,)
17
F4 peaknorm (126, 21)
After sort (126, 21)
(126,)
(126,)
F4_ps_av_peaknorm.shape shape (126, 21)
```



Congratulations on completing the assignment! Please check the instructions for submission of your final report in the Canvas Assignment.