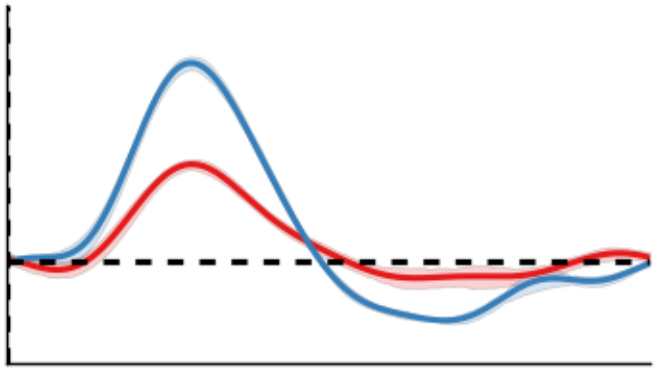
nideconv Documentation

Gilles de Hollander
Tomas Knapen

Jun 09, 2019

Getting started



Nideconv is an easy-to use Python library that can perform automated deconvolution of (primarily) slowly fluctuating (proxies of) neural signals like pupil size and BOLD fMRI. It was developed at the Vrije Universiteit and the Spinoza Centre for Neuroimaging by Gilles de Hollander and Tomas Knapen.

CHAPTER 1

Installing nideconv

1.1 From Github

Right now you can clone the main branch of nideconv using git

```
pip install git+https://github.com/VU-Cog-Sci/nideconv
```

Note: Click *here* to download the full example code

CHAPTER 2

Deconvolution

Neuroscientists (amongst others) are often interested in time series that are derived from neural activity, such as fMRI BOLD and pupil dilation. However, for some classes of data, neural activity gets temporally delayed and dispersed. This means that if the time series is related to some behavioral events that are close together in time, these event-related responses will contaminate each other.

```
# Import libraries and set up plotting
import nideconv
from nideconv import simulate
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('white')
sns.set_context('notebook')
palette = sns.color_palette('Set1')
```

2.1 Simulate data

We simulate fMRI data with a "cue - stimulus" design. There are four cues and corresponding stimulus presentations. The cue is always followed by a stimulus, seperated in time by 1, 2, 3, or 4 seconds. The cue leads to a small de-activation (0.5 % signal change), the stimulus to an activation (1.0 % signal change).

```
cue_onsets = [5, 15, 25, 35]
stim_onsets = [6, 17, 28, 39]

cue_pars = {'name':'cue',
            'mu_group':-.5, # Slight negative response for cue
            'std_group':0,
            'onsets':cue_onsets}

stim_pars = {'name':'stim',
             'mu_group':1, # Positive response for stimulus presentation
             'std_group':0,
```

(continues on next page)

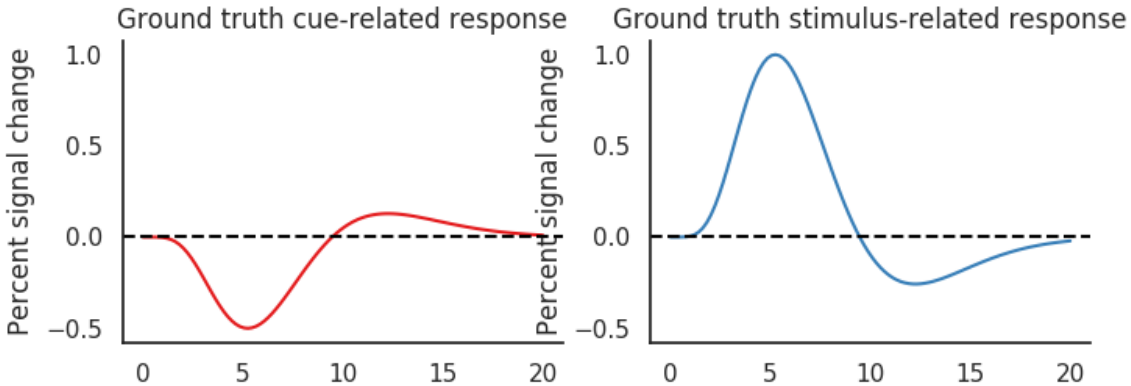
(continued from previous page)

```
             'onsets':stim_onsets}

conditions = [cue_pars,
              stim_pars]

data, onsets, parameters = simulate.simulate_fmri_experiment(conditions,
                                                             run_duration=60,
                                                             noise_level=0.05)
```

2.2 Underlying data-generating model

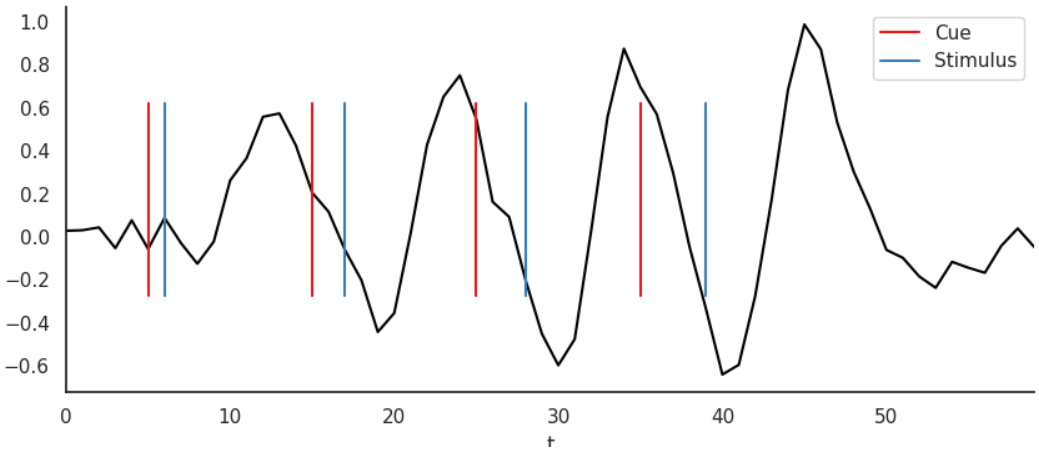
Because we simulated the data, we know that the event-related responses should exactly follow the *canonical Hemodynamic Response Function* [1]_are

```
from nideconv.utils import double_gamma_with_d
import numpy as np

plt.figure(figsize=(8, 2.5))

t = np.linspace(0, 20, 100)
ax1 = plt.subplot(121)
plt.title('Ground truth cue-related response')
plt.plot(t, double_gamma_with_d(t) * -.5,
         color=palette[0])
plt.xlabel('Time since event (s)')
plt.ylabel('Percent signal change')
plt.axhline(0, c='k', ls='--')

plt.subplot(122, sharey=ax1)
plt.title('Ground truth stimulus-related response')
plt.plot(t, double_gamma_with_d(t),
         color=palette[1])
plt.axhline(0, c='k', ls='--')
plt.xlabel('Time since event (s)')
plt.ylabel('Percent signal change')

sns.despine()
```



2.3 Plot simulated data

```
data.plot(c='k')
sns.despine()

for onset in cue_onsets:
    l1 =plt.axvline(onset, c=palette[0], ymin=.25, ymax=.75)

for onset in stim_onsets:
    l2 =plt.axvline(onset, c=palette[1], ymin=.25, ymax=.75)

plt.legend([l1, l2], ['Cue', 'Stimulus'])
plt.gcf().set_size_inches(10, 4)
```



2.4 Naive approach: epoched averaging

A simple approach that is appropriate for fast electrphysiological signals like EEG and MEG, but not necessarily fMRI, would be to select little chunks of the time series, corresponding to the onset of the events-of-interest and the subsequent 20 seconds of signal ("epoching").

We can do such a epoch-analysis using nideconv, by making a ResponseFitter object and using the *get_epochs()*-function:

```
rf =nideconv.ResponseFitter(input_signal=data,
                            sample_rate=1)

# Get all the epochs corresponding to cue-onsets for subject 1,
# run 1.
cue_epochs = rf.get_epochs(onsets=onsets.loc['cue'].onset,
                           interval=[0, 20])
```

Now we have a 4 x 21 DataFrame of epochs, that we can all plot in the same figure:

```
print(cue_epochs)
cue_epochs.T.plot(c=palette[0], alpha=.5, ls='--', legend=False)
cue_epochs.mean().plot(c=palette[0], lw=2, alpha=1.0)
sns.despine()
```

(continues on next page)

(continued from previous page)

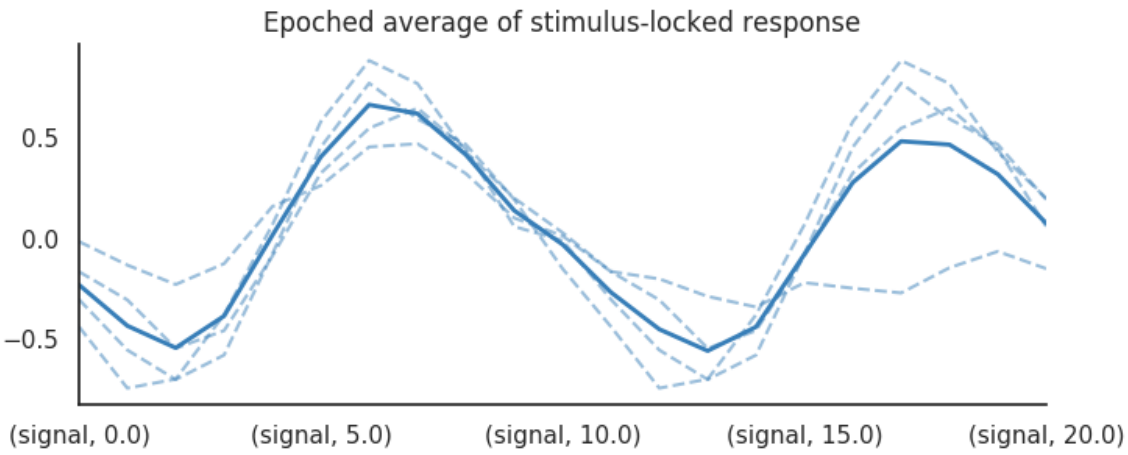
```
plt.xlabel('Time (s)')
plt.title('Epoched average of cue-locked response')
plt.gcf().set_size_inches(8, 3)
```

Out:

```
roi      signal                          ...
time        0.0       1.0       2.0     ...      18.0      19.0      20.0
onset                                     ...
5     -0.158240 -0.013876 -0.130658   ...  0.548503  0.648452  0.446225
15     0.103284  0.014407 -0.163286   ...  0.454592  0.772234  0.594640
25     0.446225  0.060751 -0.008963   ...  0.070136  0.581173  0.884578
35     0.594640  0.467924  0.194969   ... -0.338853 -0.217950 -0.245611

[4 rows x 21 columns]
```

We can do the same for the stimulus-locked responses:

```
stim_epochs = rf.get_epochs(onsets=onsets.loc['stim'].onset,
                            interval=[0, 20])
stim_epochs.T.plot(c=palette[1], alpha=.5, ls='--', legend=False)
stim_epochs.mean().plot(c=palette[1], lw=2, alpha=1.0)
sns.despine()
plt.xlabel('Time (s)')
plt.title('Epoched average of stimulus-locked response')
plt.gcf().set_size_inches(8, 3)
```



2.4.1 Contamination

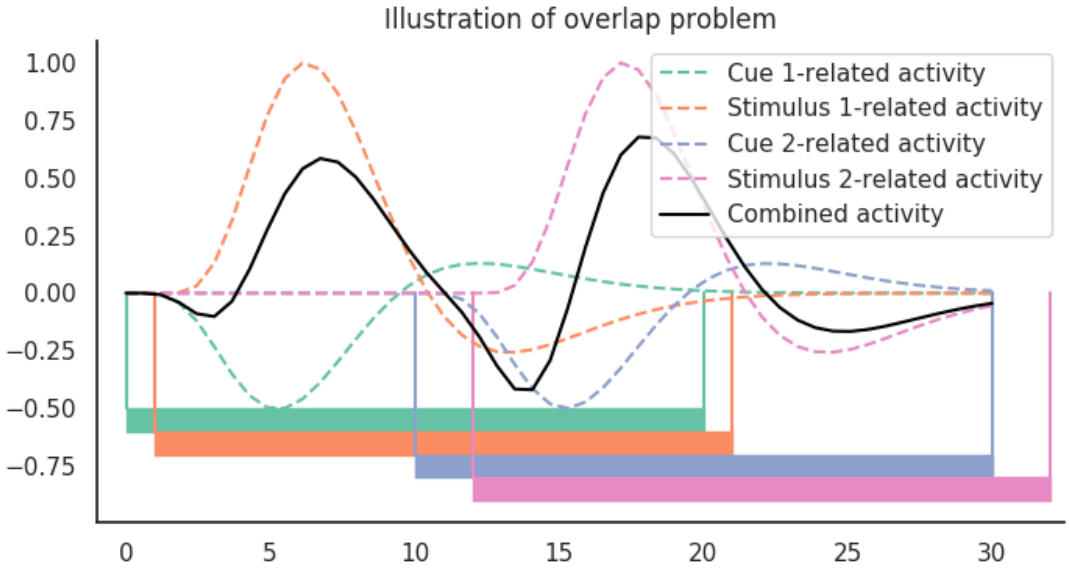
As you can see, when we use epoched averaging, both the cue- and stimulus-related response are *contaminated* by adjacent responses (from both response types). The result is some sinewave-like pattern that has little to do with the data-generating response functions of both cue- and stimulus-related activity

```
# This is because the event-related responses are overlapping in time:
from nideconv.utils import double_gamma_with_d

t = np.linspace(0, 30)

cue1_response = double_gamma_with_d(t) * -.5
stim1_response = double_gamma_with_d(t-1)

cue2_response = double_gamma_with_d(t-10) * -.5
stim2_response = double_gamma_with_d(t-12)

palette2 = sns.color_palette('Set2')

plt.fill_between([0, 20],  -.5, -.6, color=palette2[0])
plt.plot([[0, 20], [0, 20]], [-.5, 0], color=palette2[0])
plt.fill_between([1, 21],  -.6, -.7, color=palette2[1])
plt.plot([[1, 21], [1, 21]], [-.6, 0], color=palette2[1])
plt.fill_between([10, 30],  -.7, -.8, color=palette2[2])
plt.plot([[10, 30], [10, 30]], [-.7, 0], color=palette2[2])
plt.fill_between([12, 32],  -.8, -.9, color=palette2[3])
plt.plot([[12, 32], [12, 32]], [-.8, 0], color=palette2[3])


plt.plot(t, cue1_response, c=palette2[0], ls='--', label='Cue 1-related activity')
plt.plot(t, stim1_response, c=palette2[1], ls='--', label='Stimulus 1-related activity
↪')

plt.plot(t, cue2_response, c=palette2[2], ls='--', label='Cue 2-related activity')
plt.plot(t, stim2_response, c=palette2[3], ls='--', label='Stimulus 2-related activity
↪')

plt.plot(t, cue1_response + \
            stim1_response + \
```

(continues on next page)

(continued from previous page)

```
            cue2_response + \
            stim2_response,
         c='k', label='Combined activity')
plt.legend()
sns.despine()
plt.gcf().set_size_inches(8, 4)
plt.xlim(-1, 32.5)
plt.title('Illustration of overlap problem')
```



2.5 Solution: the Genera Linear Model

An often-used solution to the "overlap problem" is to assume a linear time-invariant system. This means that you assume that overlapping responses influencing time point t add up linearly. Assuming this linearity, the deconvolution boils down to solving a linear sytem: every timepoint y_t from signal Y is a linear combination of the overlapping responses, modeled by corresponding row of matrix X, $X_{t,...}$ (note that the design of matrix X is crucial here but more on that later...). We just need to find the 'weights' of the responses β.:

$$Y = X\beta$$

We can do this using a General Linear Model (GLM) and its closed-form solution ordinary least-squares (OLS).

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

This solution is part of the main functionality of nideconv. and can be applied by creating a *ResponseFitter*-object:

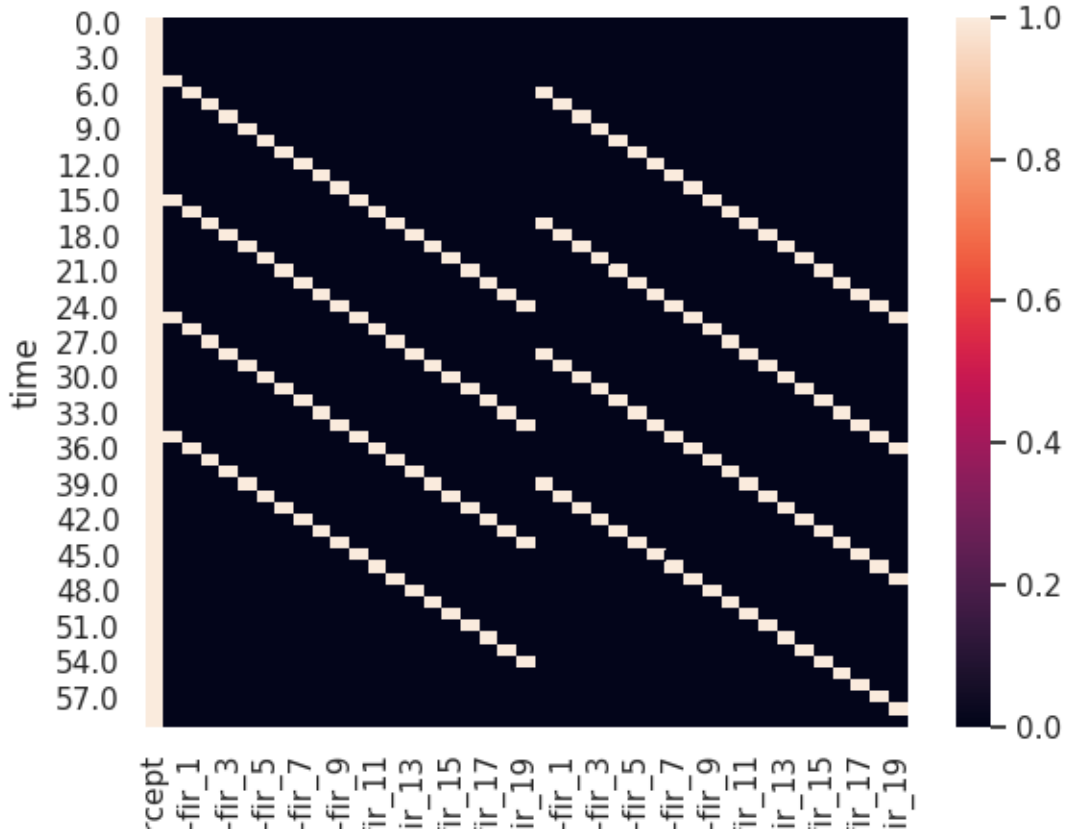
```
rf =nideconv.ResponseFitter(input_signal=data,
                            sample_rate=1)
```

To which the events-of-interest can be added as follows:

```
rf.add_event(event_name='cue',
             onsets=onsets.loc['cue'].onset,
             interval=[0,20])
rf.add_event(event_name='stimulus',
             onsets=onsets.loc['stim'].onset,
             interval=[0,20])
```

Nideconv aumatically creates a design matrix. By default, it does so using 'Finite Impulse Response'-regressors (FIR). Each one of these regressors corresponds to a different event and temporal offset. Such a design matrix looks like this:

```
sns.heatmap(rf.X)
print(rf.X)
```



Out:

```
event type confounds           cue  ...      stimulus
covariate  intercept     intercept  ...     intercept
regressor  intercept         fir_0  ...        fir_18        fir_19
time                                ...
0.0              1.0 -1.776357e-17  ... -8.881784e-18 -1.332268e-17
1.0              1.0  4.826947e-17  ... -2.664535e-17 -4.440892e-17
2.0              1.0 -4.440892e-17  ...  4.440892e-17  7.993606e-17
3.0              1.0 -5.329071e-17  ... -1.776357e-17 -1.199041e-16
4.0              1.0 -2.118095e-17  ... -9.025575e-18  0.000000e+00
5.0              1.0  1.000000e+00  ...  3.093534e-17  1.575989e-16
```

(continues on next page)

(continued from previous page)

```
6.0              1.0  2.232659e-17  ... -5.581465e-17 -1.185217e-16
7.0              1.0  1.585169e-17  ... -4.930796e-18 -1.720337e-17
8.0              1.0  6.778588e-18  ... -7.105427e-17  3.392509e-17
9.0              1.0 -7.105427e-17  ... -5.329071e-17  7.105427e-17
10.0             1.0 -4.859750e-18  ...  2.895930e-17  0.000000e+00
11.0             1.0  5.592731e-17  ... -2.254300e-17 -3.006191e-17
12.0             1.0 -6.844479e-17  ...  2.778522e-17 -7.890584e-17
13.0             1.0  3.397045e-17  ...  4.896324e-17  1.757482e-17
14.0             1.0 -5.028678e-17  ...  8.881784e-17 -2.389496e-17
15.0             1.0  1.000000e+00  ... -3.721182e-18 -5.329071e-17
16.0             1.0 -3.778477e-17  ... -1.334069e-17  9.184751e-17
17.0             1.0  1.011971e-17  ...  9.657810e-18 -3.777648e-17
18.0             1.0 -6.931033e-17  ...  4.387637e-17  7.706233e-17
19.0             1.0 -8.881784e-17  ...  0.000000e+00  9.215158e-17
20.0             1.0 -5.162537e-17  ... -2.221190e-17 -3.552714e-17
21.0             1.0 -8.794843e-17  ...  2.086186e-17 -5.928766e-17
22.0             1.0 -2.662462e-17  ... -5.839173e-17  4.416186e-17
23.0             1.0  3.123730e-17  ...  3.732154e-17  7.286403e-17
24.0             1.0  3.763226e-18  ...  1.000000e+00  4.748292e-17
25.0             1.0  1.000000e+00  ... -5.329071e-17  1.000000e+00
26.0             1.0  5.294499e-18  ...  9.016015e-17  0.000000e+00
27.0             1.0  2.584759e-17  ... -1.163986e-17  7.948316e-17
28.0             1.0  6.476414e-17  ... -3.043355e-17 -5.616264e-17
29.0             1.0 -5.443813e-18  ...  6.971763e-18 -1.101554e-16
30.0             1.0 -4.793710e-18  ... -1.421085e-16 -4.724240e-17
31.0             1.0 -3.378831e-17  ...  1.075351e-17 -1.776357e-17
32.0             1.0  7.324020e-17  ... -5.833184e-17  2.331487e-18
33.0             1.0  1.596911e-17  ...  6.006043e-17  2.334715e-17
34.0             1.0  2.431918e-17  ...  1.658620e-17 -6.672890e-17
35.0             1.0  1.000000e+00  ...  1.000000e+00  1.011500e-16
36.0             1.0  2.285811e-17  ... -1.431587e-17  1.000000e+00
37.0             1.0  7.837668e-17  ...  1.517236e-16  0.000000e+00
38.0             1.0 -1.366774e-17  ...  5.183790e-17  1.113862e-16
39.0             1.0  9.552760e-18  ...  4.957410e-18 -1.893770e-17
40.0             1.0 -2.720046e-17  ... -8.881784e-17 -7.646967e-17
41.0             1.0 -8.881784e-17  ...  7.105427e-17  5.329071e-17
42.0             1.0 -4.466079e-18  ... -9.637756e-18 -7.105427e-17
43.0             1.0  1.705257e-17  ... -3.980469e-17 -3.274346e-17
44.0             1.0 -3.366283e-18  ...  1.384605e-17  1.241116e-16
45.0             1.0 -2.989202e-17  ...  4.025400e-18  3.495476e-17
46.0             1.0  4.431898e-17  ...  1.000000e+00  2.199715e-16
47.0             1.0 -1.274637e-16  ...  1.362504e-17  1.000000e+00
48.0             1.0 -4.777595e-18  ...  4.056828e-17  0.000000e+00
49.0             1.0  4.347886e-17  ... -3.391004e-17 -2.334715e-17
50.0             1.0  1.540136e-17  ... -5.251377e-17 -3.047576e-17
51.0             1.0  0.000000e+00  ... -7.105427e-17 -4.995597e-17
52.0             1.0  2.310311e-17  ... -9.275620e-17 -1.421085e-16
53.0             1.0  9.925901e-17  ...  4.160809e-17  9.673688e-17
54.0             1.0  6.037837e-17  ...  2.200937e-17  2.644305e-17
55.0             1.0 -5.236773e-17  ...  1.389141e-16 -1.256192e-17
56.0             1.0  2.483597e-17  ... -8.881784e-17  4.297320e-17
57.0             1.0 -1.776357e-17  ...  1.000000e+00  5.329071e-17
58.0             1.0  2.733866e-17  ...  1.368792e-18  1.000000e+00
59.0             1.0 -1.968355e-17  ... -2.763133e-17  0.000000e+00

[60 rows x 41 columns]
```

(Note the hierarchical columns (event type / covariate / regressor) on the regressors)

Now we can solve this linear system using ordinary least squares:

```
rf.fit()
print(rf.betas)
```

Out:

```
signal
event type covariate regressor
confounds  intercept intercept  0.023102
cue        intercept fir_0     -0.100649
                     fir_1     -0.063218
                     fir_2      0.060717
                     fir_3     -0.205428
                     fir_4     -0.320793
                     fir_5     -0.500031
                     fir_6     -0.465807
                     fir_7     -0.436244
                     fir_8     -0.217273
                     fir_9     -0.149503
                     fir_10     0.123602
                     fir_11     0.121397
                     fir_12     0.025211
                     fir_13     0.162086
                     fir_14     0.105873
                     fir_15     0.132055
                     fir_16     0.056222
                     fir_17     0.130353
                     fir_18     0.036380
                     fir_19     0.175794
stimulus   intercept fir_0      0.095643
                     fir_1     -0.093245
                     fir_2      0.076270
                     fir_3      0.272010
                     fir_4      0.706539
                     fir_5      0.827825
                     fir_6      0.989700
                     fir_7      0.763656
                     fir_8      0.519356
                     fir_9      0.137110
                     fir_10    -0.000634
                     fir_11    -0.248265
                     fir_12    -0.157455
                     fir_13    -0.318611
                     fir_14    -0.300280
                     fir_15    -0.347443
                     fir_16    -0.147481
                     fir_17    -0.171133
                     fir_18    -0.068836
                     fir_19    -0.016540
```

Importantly, with nideconv it is also very easy to ‘convert‘ these beta-estimates to the found event-related time courses, at a higher temporal resolution:

```
tc =rf.get_timecourses()
print(tc)
```

Out:

```
signal
event type covariate time
cue        intercept 0.00  -0.100649
                     0.05  -0.100649
                     0.10  -0.100649
                     0.15  -0.100649
                     0.20  -0.100649
                     0.25  -0.100649
                     0.30  -0.100649
                     0.35  -0.100649
                     0.40  -0.100649
                     0.45  -0.100649
                     0.50  -0.100649
                     0.55  -0.063218
                     0.60  -0.063218
                     0.65  -0.063218
                     0.70  -0.063218
                     0.75  -0.063218
                     0.80  -0.063218
                     0.85  -0.063218
                     0.90  -0.063218
                     0.95  -0.063218
                     1.00  -0.063218
                     1.05  -0.063218
                     1.10  -0.063218
                     1.15  -0.063218
                     1.20  -0.063218
                     1.25  -0.063218
                     1.30  -0.063218
                     1.35  -0.063218
                     1.40  -0.063218
                     1.45  -0.063218
...                                ...
stimulus   intercept 18.50 -0.068836
                     18.55 -0.016540
                     18.60 -0.016540
                     18.65 -0.016540
                     18.70 -0.016540
                     18.75 -0.016540
                     18.80 -0.016540
                     18.85 -0.016540
                     18.90 -0.016540
                     18.95 -0.016540
                     19.00 -0.016540
                     19.05 -0.016540
                     19.10 -0.016540
                     19.15 -0.016540
                     19.20 -0.016540
                     19.25 -0.016540
                     19.30 -0.016540
                     19.35 -0.016540
                     19.40 -0.016540
                     19.45 -0.016540
                     19.50 -0.016540
                     19.55 -0.016540
                     19.60 -0.016540
```

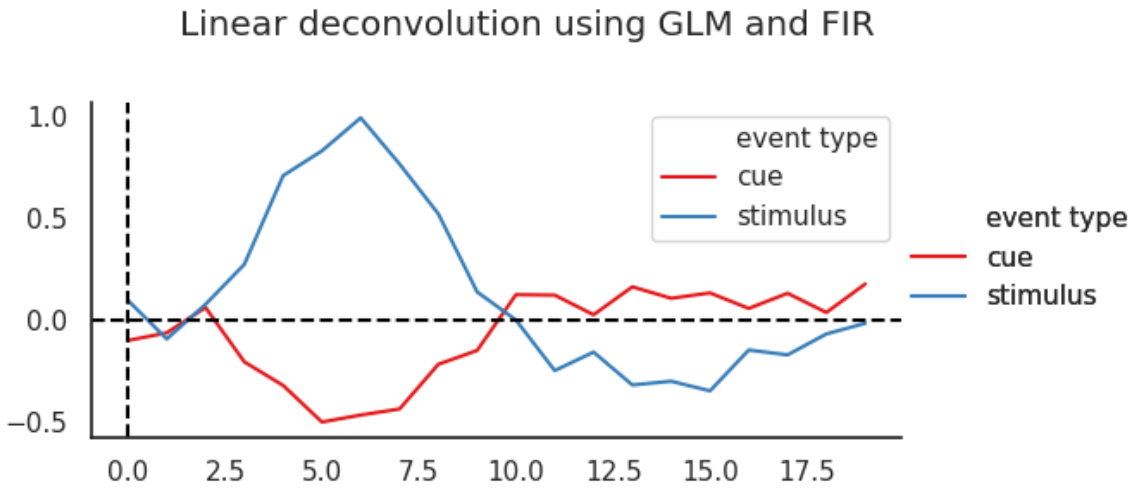
(continues on next page)

(continued from previous page)

```
                    19.65 -0.016540
                    19.70 -0.016540
                    19.75 -0.016540
                    19.80 -0.016540
                    19.85 -0.016540
                    19.90 -0.016540
                    19.95 -0.016540

[800 rows x 1 columns]
```

as well as plot these responses...

```
sns.set_palette(palette)
rf.plot_timecourses()
plt.suptitle('Linear deconvolution using GLM and FIR')
plt.title('')
plt.legend()
```



As you can see, these estimated responses are much closer to the original data-generating functions we were looking for.

Cleary, the linear deconvolution approach allows us to very quickly and effectively 'decontaminate' overlapping responses. Have a look at the next section () for more theory and plots on the selection of appropriate *basis functions*.

2.6 References

event-related BOLD fMRI. NeuroImage, 9(4), 416–429.

Total running time of the script: (0 minutes 2.383 seconds)

Note: Click *here* to download the full example code

CHAPTER 3

Basis functions

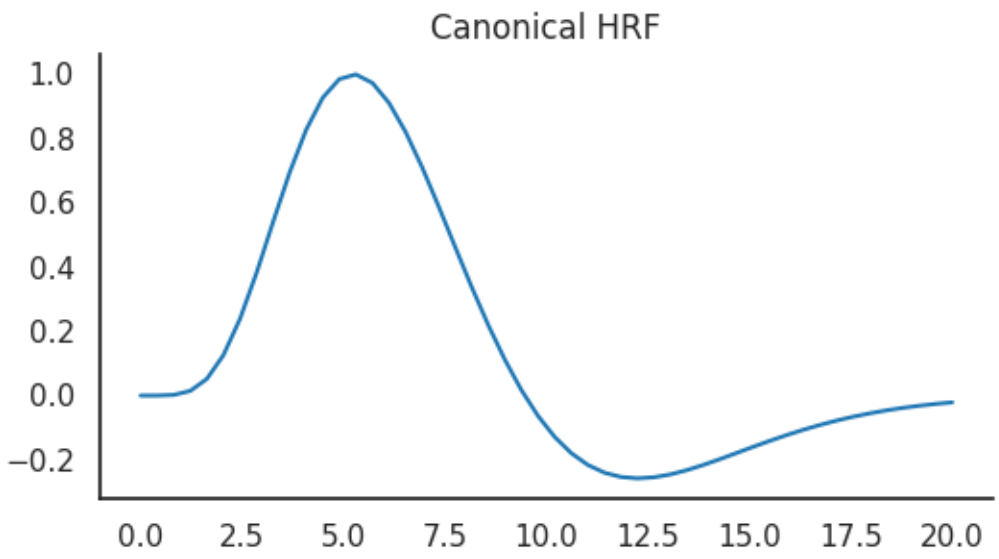
In the previous tutorial we have seen that overlapping even-related time courses can be recovered using the general linear model, as long as we assume they add up linearly and are time-invariant.

```
# Import libraries and setup plotting
from nideconv.utils import double_gamma_with_d
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('white')
sns.set_context('notebook')
```

3.1 Well-specified model

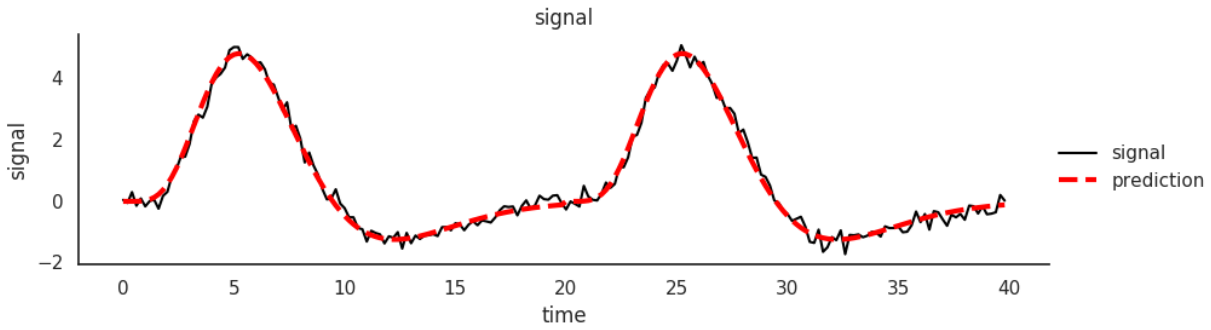
Another important assumption pertains to what we believe the response we are interested in looks like. In the most extreme case, we for example assume that a task-related BOLD fMRI response exactly follows the canonical HRF.

```
plt.figure(figsize=(6,3))
t = np.linspace(0, 20)
plt.plot(t, double_gamma_with_d(t))
plt.title('Canonical HRF')
sns.despine()
```



Let's simulate some data with very little noise and the standard HRF, fit a model, and see how well our model fits the data

```
from nideconv import simulate
from nideconv import ResponseFitter

conditions = [{'name':'Condition A',
               'mu_group':5,
               'std_group':1,
                'onsets':[0, 20]}]

# Simulate data with very short run time and TR for illustrative purposes
data, onsets, pars = simulate.simulate_fmri_experiment(conditions,
                                                       TR=0.2,
                                                       run_duration=40,
                                                       noise_level=.2,
                                                       n_rois=1)

# Make ResponseFitter-object to fit these data
rf = ResponseFitter(input_signal=data,
                    sample_rate=5) # Sample rate is inverse of TR (1/TR)
rf.add_event('Condition A',
             onsets.loc['Condition A'].onset,
             interval=[0, 20],
             basis_set='canonical_hrf')

rf.fit()

rf.plot_model_fit()
```

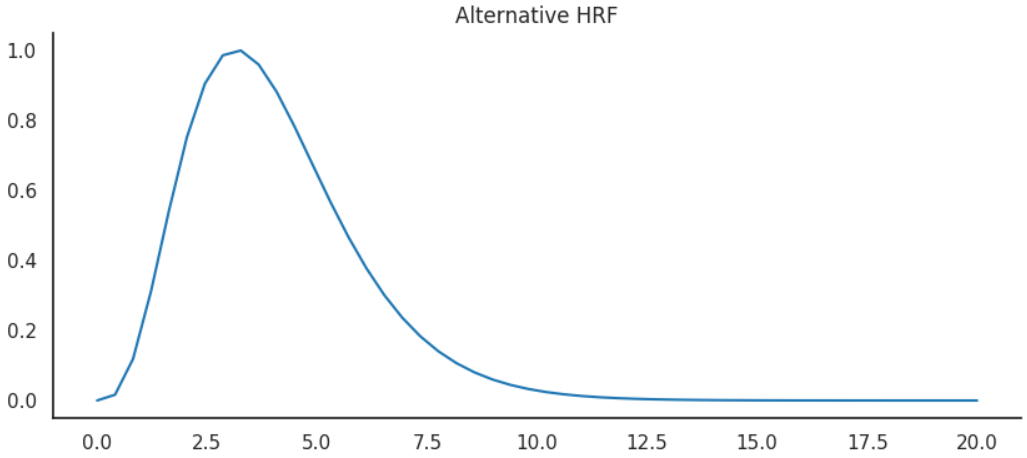


As you can see, the model fits the data well, the model is well-specified.

3.2 Mis-specified model

Now let's what happens with data with different HRF from what the model assumes.

```
# For this HRF the first peak is much earlier (approx 3.5 seconds versus 5.8)
# and the second "dip" is not there (c=0).
kernel_pars = {'a1':3.5,
               'c':0}

plt.plot(t, double_gamma_with_d(t, **kernel_pars))
plt.title('Alternative HRF')
sns.despine()
plt.gcf().set_size_inches(10, 4)
```



Simulate data again

```
data, onsets, pars = simulate.simulate_fmri_experiment(conditions,
                                                       TR=0.2,
                                                       run_duration=50,
                                                       noise_level=.2,
                                                       n_rois=1,
                                                       kernel_pars=kernel_pars)
```

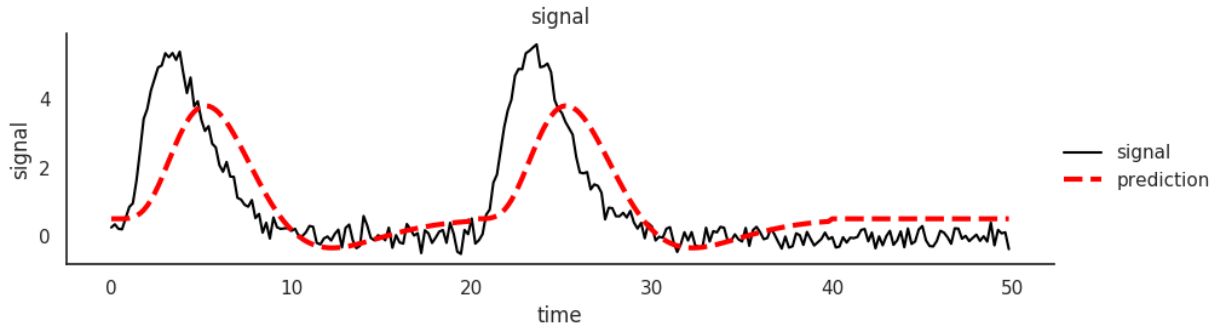
(continues on next page)

(continued from previous page)

```
rf = ResponseFitter(input_signal=data,
                    sample_rate=5)
rf.add_event('Condition A',
             onsets.loc['Condition A'].onset,
             interval=[0, 20],
             basis_set='canonical_hrf')
```

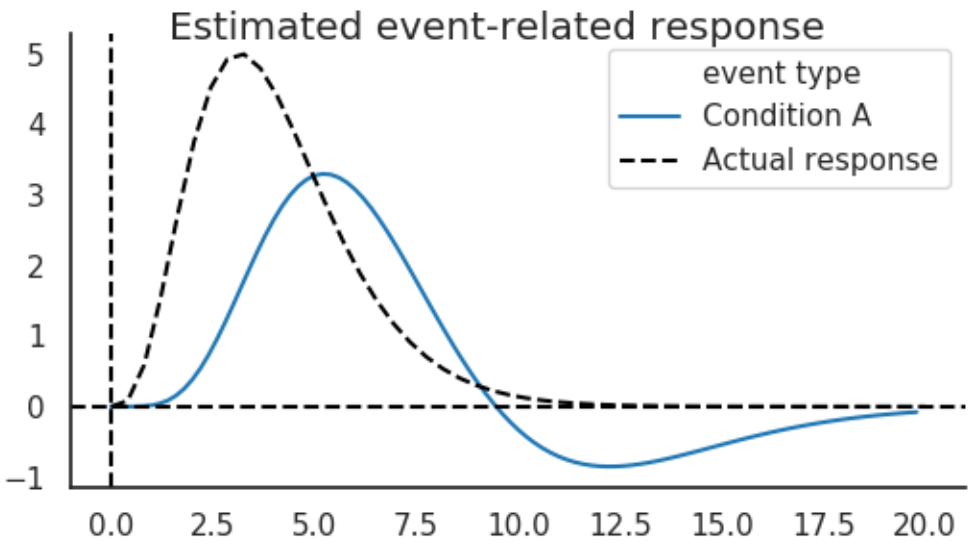
Plot the model fit

```
rf.fit()
rf.plot_model_fit()
```



And the estimated time course

```
def plot_estimated_and_actual_tc(rf,
                                 kernel_pars=kernel_pars,
                                 amplitudes=[5]):
    """
    Plots estimated event-related responses plus the actual underlying
    responses given by kernel_pars and amplitudes
    """
    rf.plot_timecourses(legend=False)
    t = np.linspace(0, 20)

    # Allow for plotting multiple amplitudes
    amplitudes = np.array(amplitudes)
    plt.plot(t,
             double_gamma_with_d(t, **kernel_pars)[:, np.newaxis] * amplitudes,
             label='Actual response',
             ls='--',
             c='k')

    plt.legend()
    plt.suptitle('Estimated event-related response')
    plt.tight_layout()


plot_estimated_and_actual_tc(rf)
```



The estimated time-to-peak is completely off

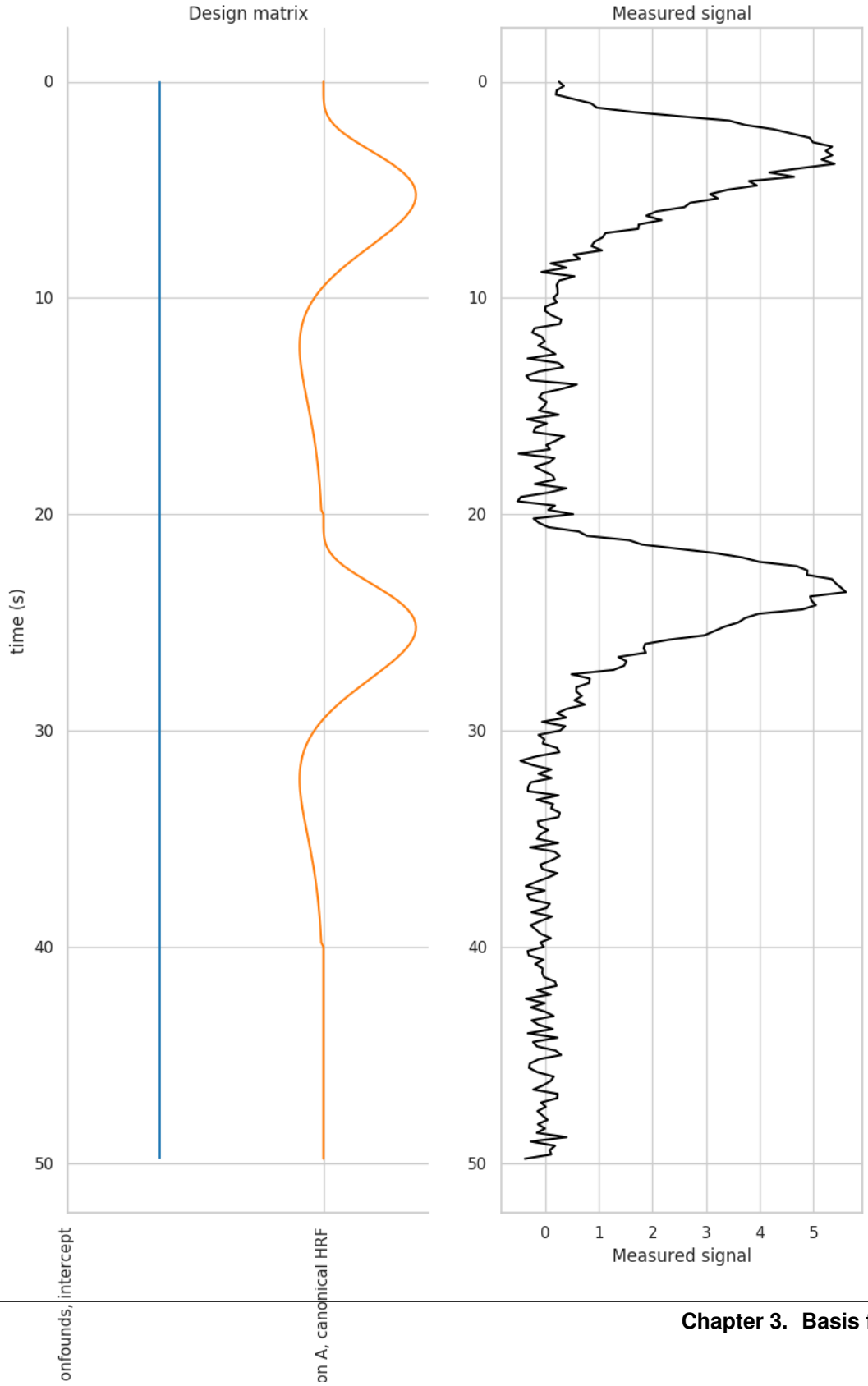
```
print(rf.get_time_to_peak())
```

Out:

```
time to peak
area                                signal
event type  covariate peak
Condition A intercept 1              5.24
```

Clearly, the model now does not fit very well. This is because the design matrix X does not allow for properly modelling the event-related time course. No matter what linear combination β you take of the intercept and canonical HRF (that has been convolved with the event onsets), the data can never be properly explained.

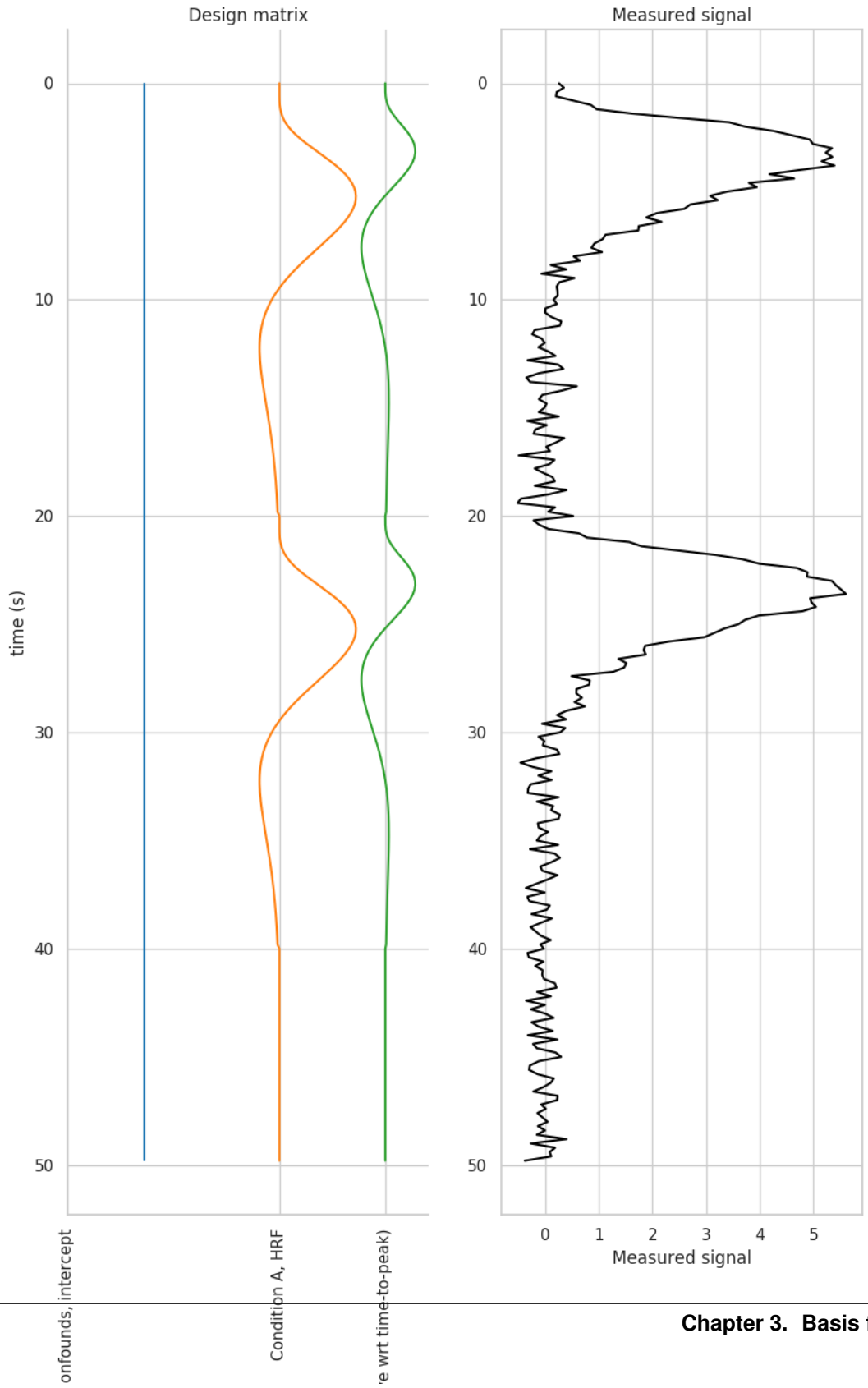
```
def plot_design_matrix(rf):
    """
    Plots the design matrix of rf in left plot, plus the signal
    it should explain in the right plot.
    Time is in the vertical dimension. Each regressor in X is plotted
    as a seperate line.
    """
    sns.set_style('whitegrid')
    ax = plt.subplot(121)
    rf.plot_design_matrix()

    plt.title('Design matrix')

    plt.subplot(122, sharey=ax)
    plt.plot(rf.input_signal, rf.input_signal.index, c='k')
    plt.title('Measured signal')
    plt.xlabel('Measured signal')

    plt.gcf().set_size_inches(10,15)
    sns.set_style('white')

plot_design_matrix(rf)
```


Design matrix
Measured signal
time (s)
0
10
20
30
40
50
0
1
2
3
4
5
Measured signal
onfounds, intercept
on A, canonical HRF

3.3 The derivative of the cHRF with respect to time

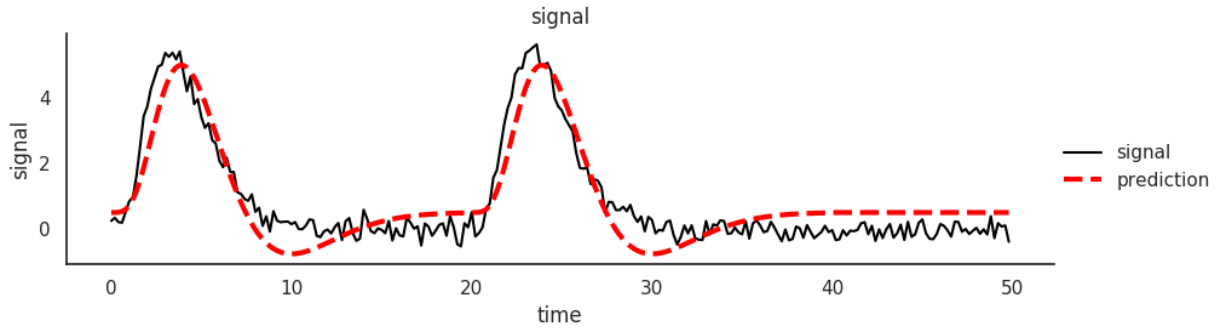
The solution to this mis-specification is to increase the model complexity by adding extra regressors that increase the flexibility of the model. A very standard approach in BOLD fMRI is to include the derivative of the HRF with respect to time for dt=0.1. Then the design matrix looks like this

```
rf = ResponseFitter(input_signal=data,
                    sample_rate=5)
rf.add_event('Condition A',
             onsets.loc['Condition A'].onset,
             interval=[0, 20],
             basis_set='canonical_hrf_with_time_derivative') # note the more complex
                                                             # basis function set

plot_design_matrix(rf)
```

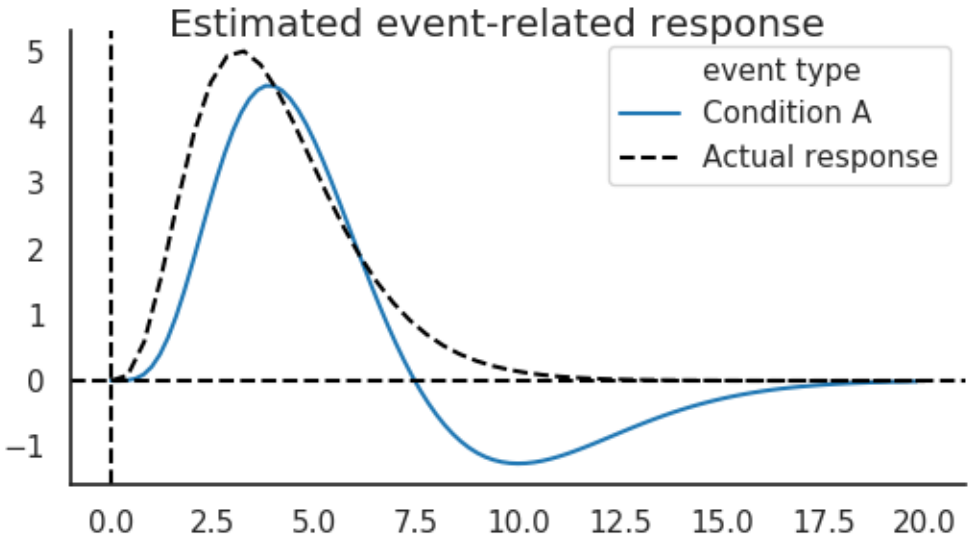

Design matrix
Measured signal
time (s)
0
10
20
30
40
50
onfounds, intercept
Condition A, HRF
e wrt time-to-peak)
0
1
2
3
4
5
Measured signal

The GLM can now "use the new, red regressor" to somewhat shift the original HRF earlier in time.

```
rf.fit()
rf.plot_model_fit()
```




```
plot_estimated_and_actual_tc(rf)
```



The estimated time-to-peak is more in the range of what it should be

```
print(rf.get_time_to_peak())
```

Out:

```
time to peak
area                              signal
event type  covariate peak
Condition A intercept 1             3.89
```

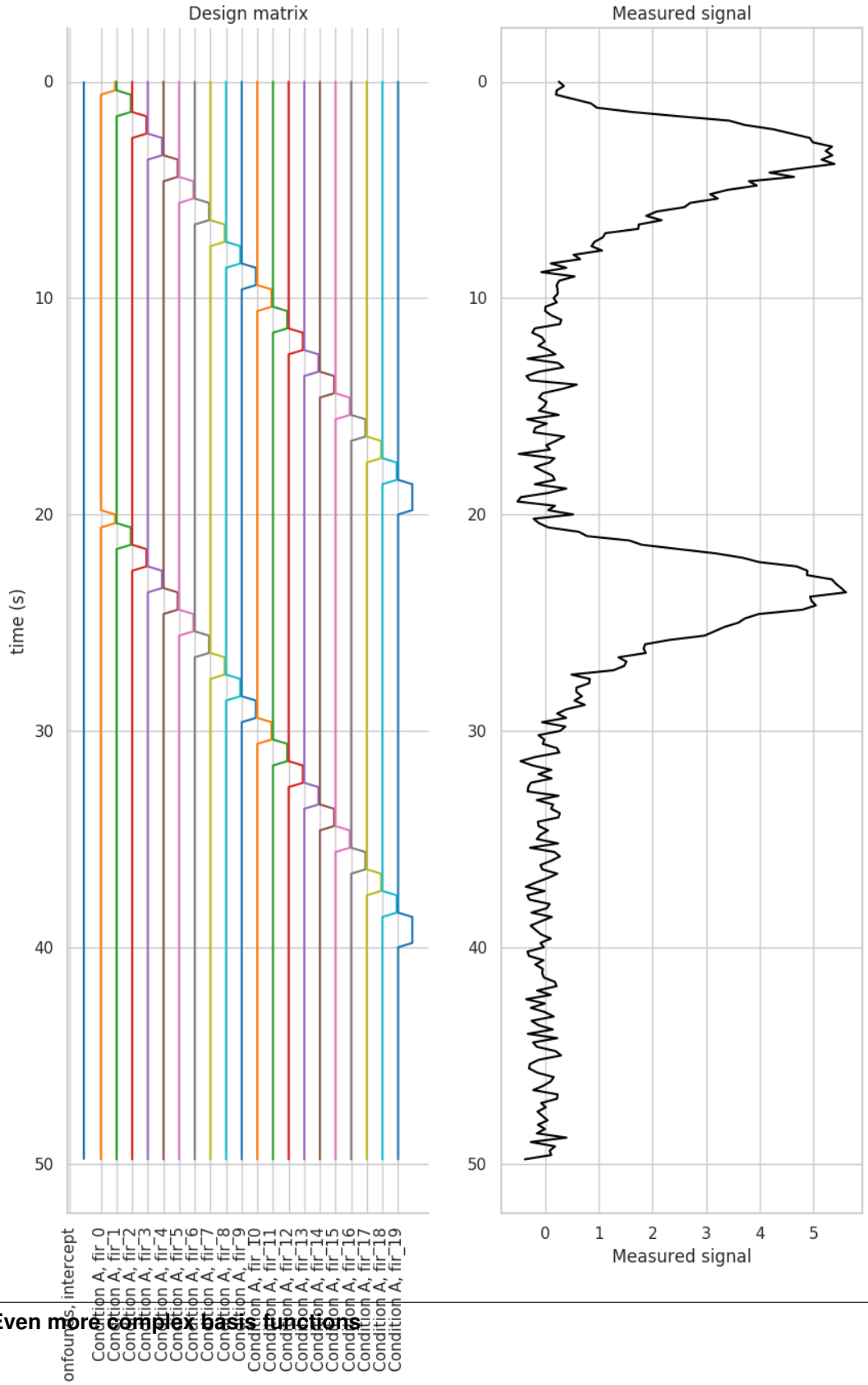
3.4 Even more complex basis functions

Note that the canonical HRF model-with-derivative still does not fit perfectly: the Measured signal is still peaking earlier in time than the model. And the model still assumes as post-peak dip that is not there in the data. Therefore, it

also underestimates the height of the first peak.

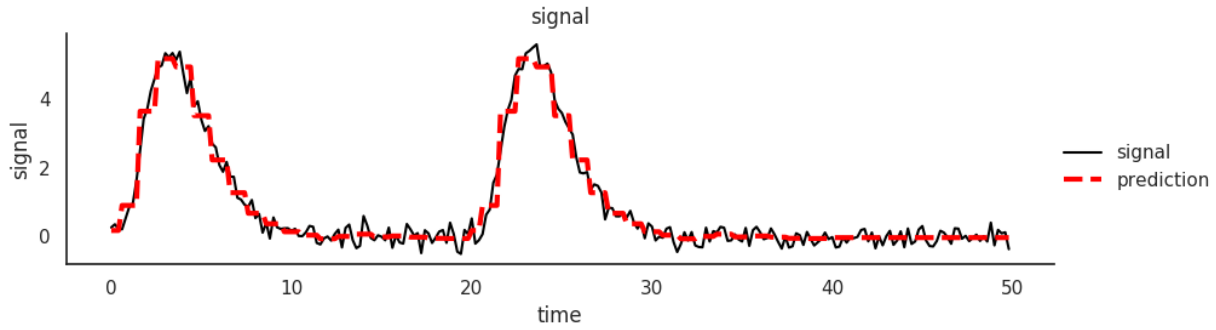
One solution is to use yet more complex basis functions, such as the Finite Impulse Response functions we used in the previous tutorial. This basis functions consists of one regressor per time-bin (as in time offset since event).

```
rf = ResponseFitter(input_signal=data,
                    sample_rate=5)
rf.add_event('Condition A',
             onsets.loc['Condition A'].onset,
             interval=[0, 20],
             basis_set='fir',
             n_regressors=20) # One regressor per second

plot_design_matrix(rf)
```


Design matrix
Measured signal
time (s)
0
10
20
30
40
50
onfounds, intercept
Condition A, fir_0
Condition A, fir_1
Condition A, fir_2
Condition A, fir_3
Condition A, fir_4
Condition A, fir_5
Condition A, fir_6
Condition A, fir_7
Condition A, fir_8
Condition A, fir_9
Condition A, fir_10
Condition A, fir_11
Condition A, fir_12
Condition A, fir_13
Condition A, fir_14
Condition A, fir_15
Condition A, fir_16
Condition A, fir_17
Condition A, fir_18
Condition A, fir_19
0
1
2
3
4
5
Measured signal

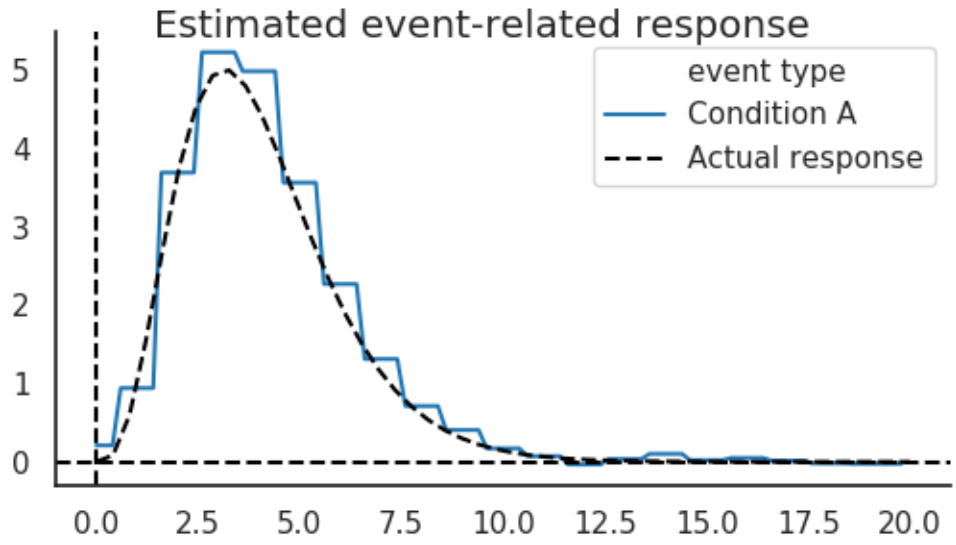
Clearly, this model is much more flexible, and, hence, it fits better:

```
rf.fit()
rf.plot_model_fit()
```



Clearly, this model is much more flexible, and, hence, it fits better:

```
plot_estimated_and_actual_tc(rf)
```



Also, the estimated time-to-peak is more in the range of where it should be

```
print(rf.get_time_to_peak())
```

Out:

```
time to peak
area                              signal
event type  covariate peak
Condition A intercept 1             3.0
```

3.5 No Free Lunch (Bias-Variance tradeoff)

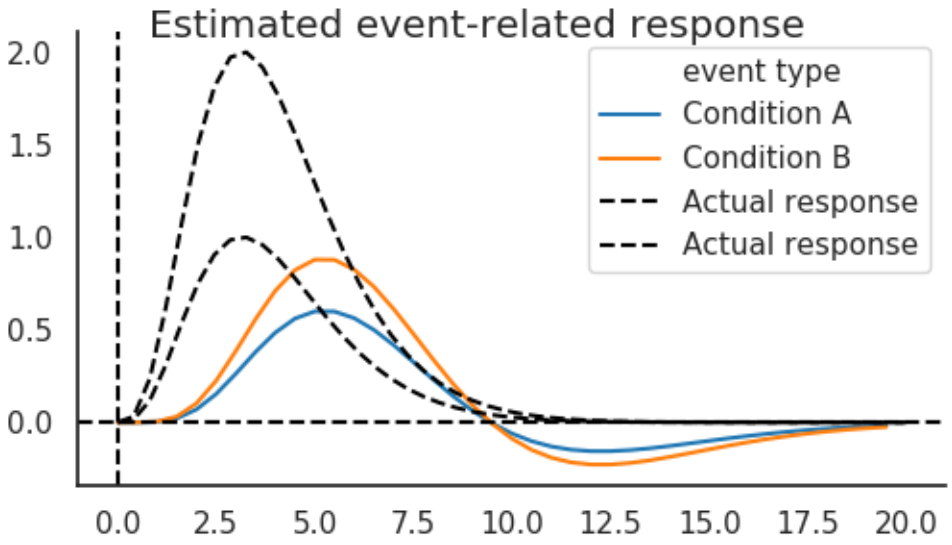
The higher flexibility of the FIR model is due to its higher *degrees-of-freedom*, roughly the number of regressors. It is important to note that a higher number of degrees-of-freedom also mean a higher *variance* of the model. A higher variance means that smaller fluctuations in the data will lead to larger differences in parameter estimates. This is especially problematic in high-noise regimes. The following simulation will show this.

3.5.1 Simulation

```
# Set a random seed so output will always be the same
np.random.seed(666)

# Simulate data
TR = 0.5
sample_rate = 1./TR
data, onsets, pars = simulate.simulate_fmri_experiment(noise_level=2.5,
                                                       TR=TR,
                                                       run_duration=1000,
                                                       n_trials=100,
                                                       kernel_pars=kernel_pars)
# cHRF model
hrf_model = ResponseFitter(data, sample_rate)
hrf_model.add_event('Condition A', onsets.loc['Condition A'].onset, interval=[0, 20],␣
↪basis_set='canonical_hrf')
hrf_model.add_event('Condition B', onsets.loc['Condition B'].onset, interval=[0, 20],␣
↪basis_set='canonical_hrf')

# cHRF model with derivative wrt time-to-peak
hrf_dt_model = ResponseFitter(data, sample_rate)
hrf_dt_model.add_event('Condition A', onsets.loc['Condition A'].onset, interval=[0,␣
↪20], basis_set='canonical_hrf_with_time_derivative')
hrf_dt_model.add_event('Condition B', onsets.loc['Condition B'].onset, interval=[0,␣
↪20], basis_set='canonical_hrf_with_time_derivative')

# FIR_model
fir_model = ResponseFitter(data, sample_rate)
fir_model.add_event('Condition A', onsets.loc['Condition A'].onset, interval=[0, 20])
fir_model.add_event('Condition B', onsets.loc['Condition B'].onset, interval=[0, 20])
```

3.5.2 Simplest model (cHRF)

```
hrf_model.fit()
plot_estimated_and_actual_tc(hrf_model,
                             amplitudes=pars.amplitude.tolist())
```



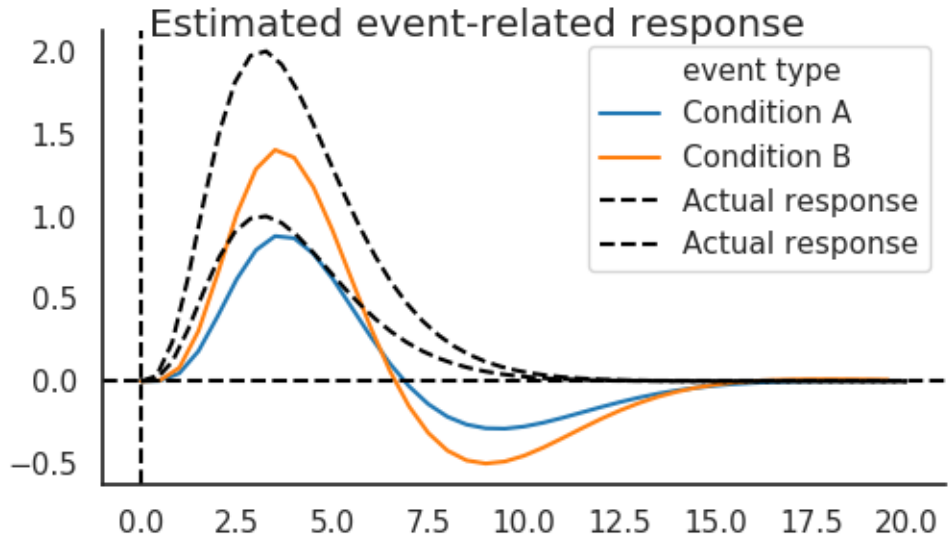

```
print(hrf_model.get_time_to_peak())
```

Out:

```
time to peak
area                                signal
event type  covariate peak
Condition A intercept 1              5.25
Condition B intercept 1              5.25
```

3.5.3 Extend model (cHRF + deriative wrt time)

```
hrf_dt_model.fit()
plot_estimated_and_actual_tc(hrf_dt_model,
                             amplitudes=pars.amplitude.tolist())
```



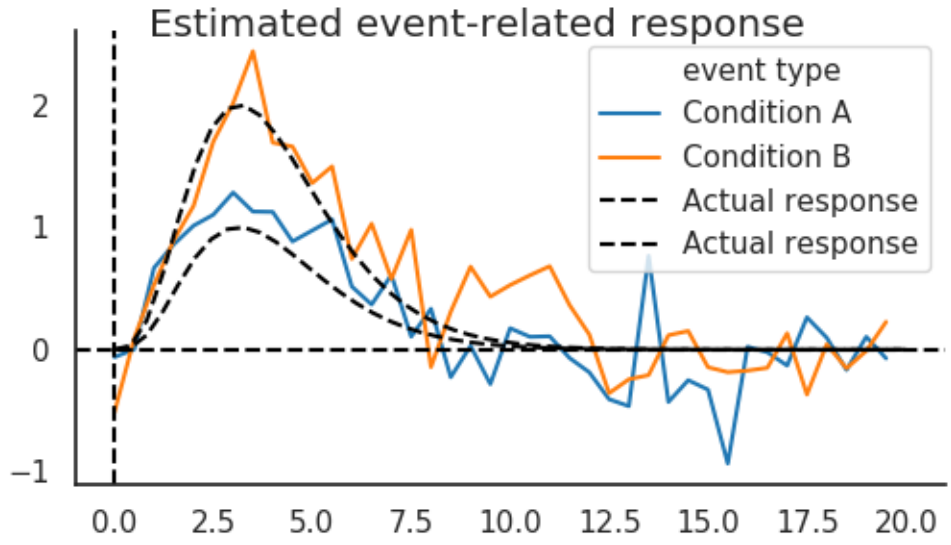
```
print(hrf_dt_model.get_time_to_peak())
```

Out:

```
time to peak
area                                signal
event type  covariate peak
Condition A intercept 1              3.75
Condition B intercept 1              3.65
```

3.5.4 Most complex model (FIR)

```
fir_model.fit()
plot_estimated_and_actual_tc(fir_model,
                             amplitudes=pars.amplitude.tolist())
```




```
print(fir_model.get_time_to_peak())
```

Out:

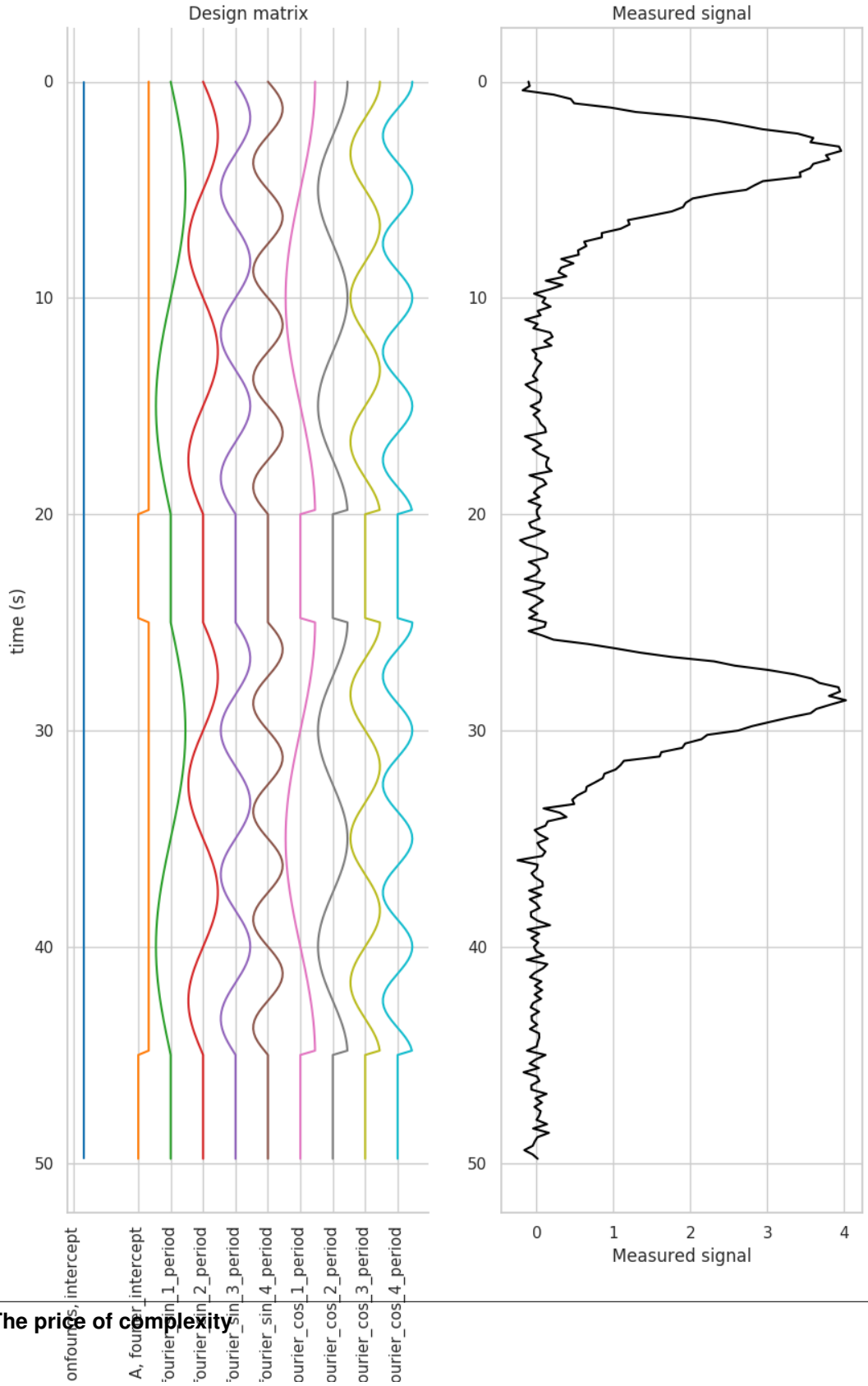
```
time to peak
area                                signal
event type  covariate peak
Condition A intercept 1              3.0
Condition B intercept 1              3.5
```

3.6 The price of complexity

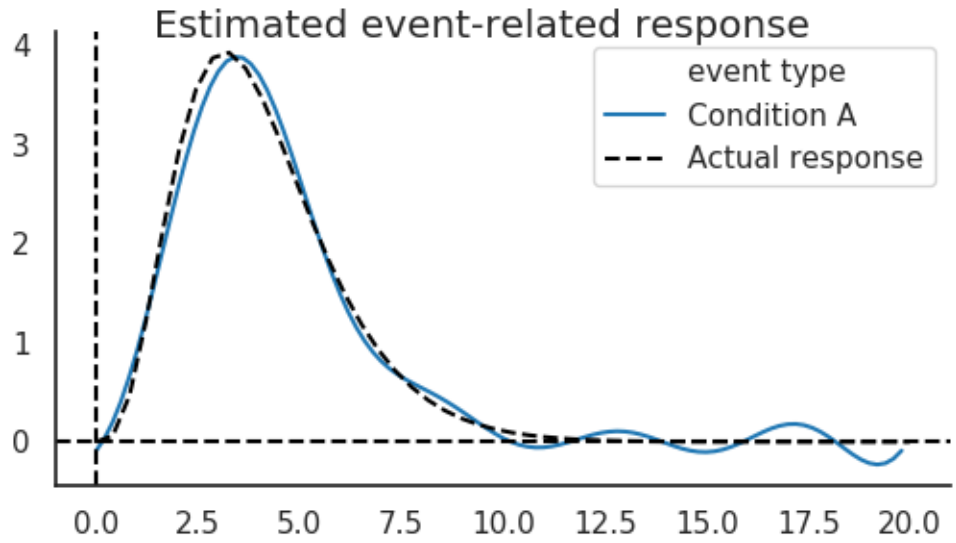
As you can see, the simplest model does not perform very well, because it is mis-specified to such a large degree. However, the most complex model (FIR) also does not perform very well: the estimated event-related time course is extremely noisy and it looks very "spiky". The cHRF that includes a derivative wrt to time also doesn't perform perfectly, because it assume as post-peak undershoot.

Another basis function set that is quite useful for slow, smooth time courses like fMRI BOLD and the pupil is the *Fourier set*. It consists of an intercept and sine-cosine pairs of increasing frequency.

```
conditions = [{'name':'Condition A',
               'mu_group':5,
               'std_group':1,
                'onsets':[0, 25]}]

TR = 0.2
sample_rate = 1./TR

data, onsets, pars = simulate.simulate_fmri_experiment(conditions,
                                                       TR=TR,
                                                       run_duration=50,
                                                       noise_level=.1,
                                                       n_rois=1,
                                                       kernel_pars=kernel_pars)

fourier_model = ResponseFitter(data, sample_rate)
fourier_model.add_event('Condition A',
                        onsets.loc['Condition A'].onset,
                        basis_set='fourier',
                        n_regressors=9,
                        interval=[0, 20])
plot_design_matrix(fourier_model)
```


Design matrix
Measured signal
time (s)
0
10
20
30
40
50
Measured signal
0
1
2
3
4
onfounds, intercept
A, fourier_intercept
ourier_sin_1_period
ourier_sin_2_period
ourier_sin_3_period
ourier_sin_4_period
ourier_cos_1_period
ourier_cos_2_period
ourier_cos_3_period
ourier_cos_4_period

```
fourier_model.fit()
plot_estimated_and_actual_tc(fourier_model,
                             amplitudes=pars.amplitude.tolist())
```




```
print(rf.get_time_to_peak())
```

Out:

```
time to peak
area                              signal
event type  covariate peak
Condition A intercept 1              3.0
```

3.6.1 Smoothness constraint

The Fourier model combines the flexibility of the FIR model, with a lower number of degrees of freedom. It can do so because the number of time courses it can explain is reduced. It can only account for *smooth* time courses with lower temporal frequencies. This is a good thing for many applications, like BOLD fMRI, where we know that the time course *has to be* smooth, since this is the nature of the neurovascular response (a similar argument can be made for pupil dilation time courses).

3.7 Conclusion

This tutorial showed how we can use different basis functions in our GLM to deconvolve event-related responses. We can very constrained basis functions, like the canonical HRF, our very flexible basis functions, like the FIR basis set. In general, a balance should be struck between flexibility and degrees of freedom, which can in part be acheived by using basis functions that are targeted towards the kind of responses that are to be expected, notably responses that are temporally smooth. The Fourier basis set is a good example of a generic basis set that allows for relative flexibility with relatively low number of degrees of freedom.

Total running time of the script: (0 minutes 11.525 seconds)

Note: Click *here* to download the full example code

CHAPTER 4

Deconvolution on group of subjects

The *GroupResponseFitter*-object of *Nideconv* offers an easy way to fit the data of many subjects together well-specified model *nideconv.simulate* can simulate data from multiple subjects.

```
from nideconv import simulate

data, onsets, pars = simulate.simulate_fmri_experiment(n_subjects=8,
                                                       n_rois=2,
                                                       n_runs=2)
```

Now we have an indexed Dataframe, *data*, that contains time series for every subject, run, and ROI:

```
print(data.head())
```

Out:

```
area 1    area 2
subject run t
1       1   0.0  0.108141  0.206549
            1.0 -0.199233  0.100078
            2.0  0.152216  1.322024
            3.0 -0.209591  0.410515
            4.0 -1.206617 -0.040518
```

```
print(data.tail())
```

Out:

```
area 1    area 2
subject run t
8       2   295.0  2.797666  2.103899
            296.0 -0.308147  0.574413
            297.0  0.122700  1.787650
            298.0  1.305447  2.825086
            299.0  1.271533  1.641312
```

We also have onsets for every subject and run:

```
print(onsets.head())
```

Out:

```
                                onset
subject run trial_type
1       1   Condition A  219.820716
            Condition A  103.107395
            Condition A  266.262272
            Condition A    7.918901
            Condition A   89.896628
```

```
print(onsets.tail())
```

Out:

```
                                onset
subject run trial_type
8       2   Condition B  182.382405
            Condition B  237.654933
            Condition B  258.877295
            Condition B   63.358865
            Condition B   37.917893
```

We can now use *GroupResponseFitter* to fit all the subjects with one object:

```
from nideconv import GroupResponseFitter


# `concatenate_runs` means that we concatenate all the runs of a single subject
# together, so that we have to fit only a single GLM per subject.
# A potential advantage is that the GLM has less degrees-of-freedom
# compared to the amount of data, so that the esitmate are potentially more
# stable.
# A potential downside is that different runs might have different intercepts
# and/or correlation structure.
# Therefore, by default, the `GroupResponseFitter` does _not_ concatenate
# runs.
g_model = GroupResponseFitter(data,
                              onsets,
                              input_sample_rate=1.0,
                              concatenate_runs=False)
```

We use the *add_event*-method to add the events we are interested in. The *GroupResponseFitter* then automatically collects the right onset times from the *onsets*-object.

We choose here to use the *Fourier*-basis set, with 9 regressors.

```
g_model.add_event('Condition A',
                  basis_set='fourier',
                  n_regressors=9,
                  interval=[0, 20])

g_model.add_event('Condition B',
```

(continues on next page)

(continued from previous page)

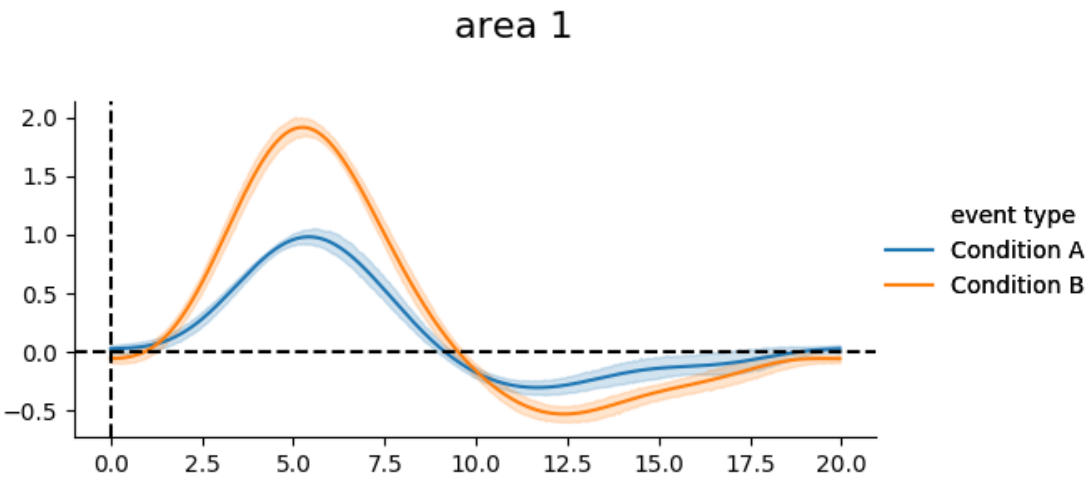
```
                  basis_set='fourier',
                  n_regressors=9,
                  interval=[0, 20])
```

We can fit all the subjects at once using the *fit*-method

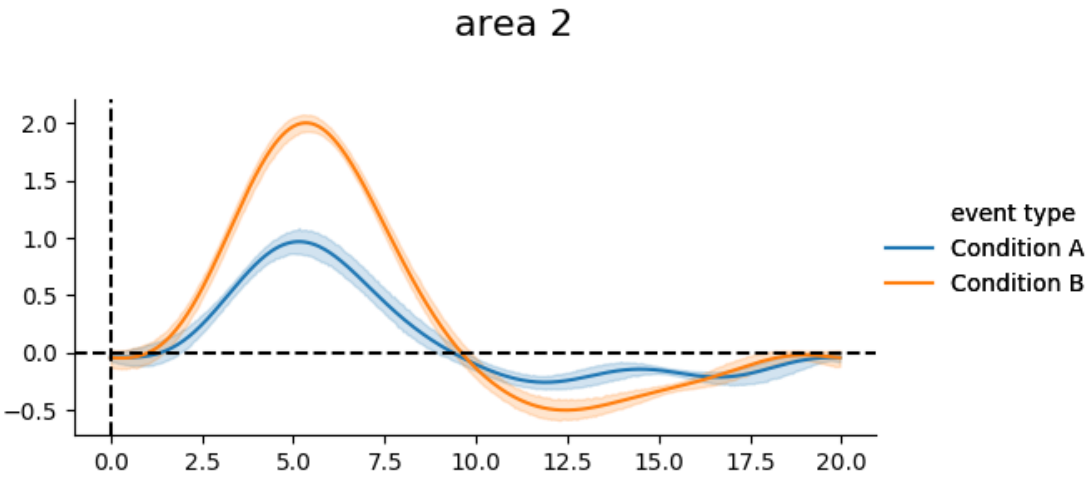
```
g_model.fit()
```

We can plot the mean timecourse across subjects

```
print(g_model.get_subjectwise_timecourses().head())
g_model.plot_groupwise_timecourses()
```



-



-

Out:

```
area 1    area 2
subject event type  covariate time
1       Condition A intercept 0.00  0.056636  0.047815
                              0.05  0.056482  0.047954
                              0.10  0.056218  0.047809
                              0.15  0.055854  0.047394
```

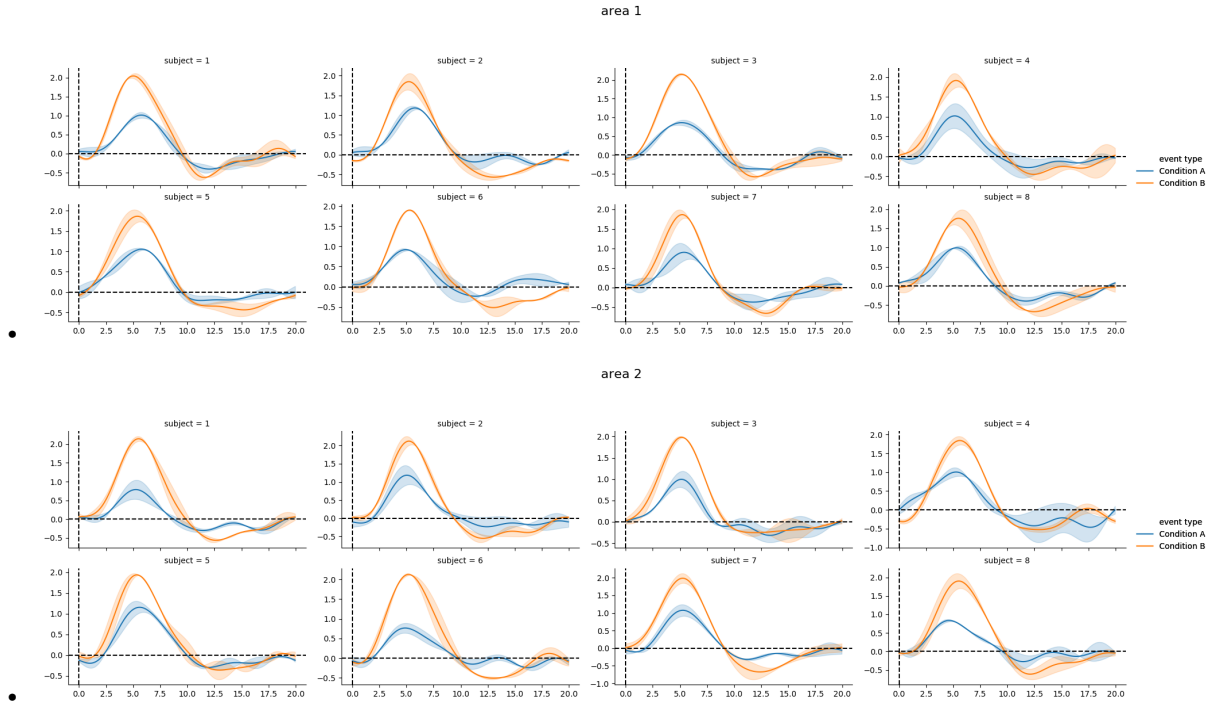
(continues on next page)

(continued from previous page)

```
                                0.20   0.055398   0.046727
```

As well as individual time courses

```
print(g_model.get_conditionwise_timecourses())
g_model.plot_subject_timecourses()
```



Out:

```
area 1    area 2
event type  covariate time
Condition A intercept 0.00   0.027545 -0.039614
                      0.05   0.028095 -0.039715
                      0.10   0.028649 -0.039852
                      0.15   0.029217 -0.040009
                      0.20   0.029811 -0.040169
                      0.25   0.030440 -0.040316
                      0.30   0.031117 -0.040429
                      0.35   0.031852 -0.040491
                      0.40   0.032659 -0.040482
                      0.45   0.033549 -0.040383
                      0.50   0.034535 -0.040174
                      0.55   0.035629 -0.039835
                      0.60   0.036845 -0.039346
                      0.65   0.038195 -0.038685
                      0.70   0.039693 -0.037834
                      0.75   0.041352 -0.036772
                      0.80   0.043184 -0.035478
                      0.85   0.045204 -0.033932
                      0.90   0.047423 -0.032116
                      0.95   0.049855 -0.030010
                      1.00   0.052513 -0.027594
```

(continues on next page)

(continued from previous page)

```
                       1.05   0.055408 -0.024851
                       1.10   0.058554 -0.021763
                       1.15   0.061962 -0.018312
                       1.20   0.065644 -0.014483
                       1.25   0.069611 -0.010259
                       1.30   0.073873 -0.005627
                       1.35   0.078442 -0.000571
                       1.40   0.083326  0.004920
                       1.45   0.088535  0.010859
...                                ...       ...
Condition B intercept 18.50 -0.074890 -0.028460
                      18.55 -0.072558 -0.026545
                      18.60 -0.070388 -0.024846
                      18.65 -0.068382 -0.023366
                      18.70 -0.066540 -0.022104
                      18.75 -0.064863 -0.021059
                      18.80 -0.063349 -0.020230
                      18.85 -0.061996 -0.019614
                      18.90 -0.060803 -0.019208
                      18.95 -0.059764 -0.019008
                      19.00 -0.058876 -0.019006
                      19.05 -0.058132 -0.019198
                      19.10 -0.057526 -0.019574
                      19.15 -0.057051 -0.020126
                      19.20 -0.056697 -0.020843
                      19.25 -0.056454 -0.021716
                      19.30 -0.056312 -0.022730
                      19.35 -0.056259 -0.023874
                      19.40 -0.056281 -0.025132
                      19.45 -0.056366 -0.026488
                      19.50 -0.056498 -0.027927
                      19.55 -0.056661 -0.029431
                      19.60 -0.056838 -0.030981
                      19.65 -0.057011 -0.032557
                      19.70 -0.057162 -0.034138
                      19.75 -0.057272 -0.035704
                      19.80 -0.057319 -0.037232
                      19.85 -0.057283 -0.038699
                      19.90 -0.057142 -0.040080
                      19.95 -0.056873 -0.041351

[800 rows x 2 columns]
```

Total running time of the script: (3 minutes 28.376 seconds)

Note: Click *here* to download the full example code

CHAPTER 5

Extract timeseries from ROIs using fmriprep data

The first step in most deconvolution analyes is the extraction of the signal from different regions-of-interest.

To do this, *nideconv* contains an easy-to-use module (*nideconv.utils.roi*) that leverages the functionality of nilearn. It can extract a time series for every ROI in an atlas. Standard atlases included in nilearn can be found in the nilearn manual.

Using *nilearn*, the module can also temporally filter the voxelwise signals as well as, clean them from any confounds.

This module is especially useful for preprocessed in the BIDS format, as for example the output of fmriprep.

5.1 Extracting a time series from a single functional run

Here we extract some signals from a single functional run from a import *Stroop* dataset we got from a open data repository on Openneuro. The data has been preprocessed using the fmriprep software.

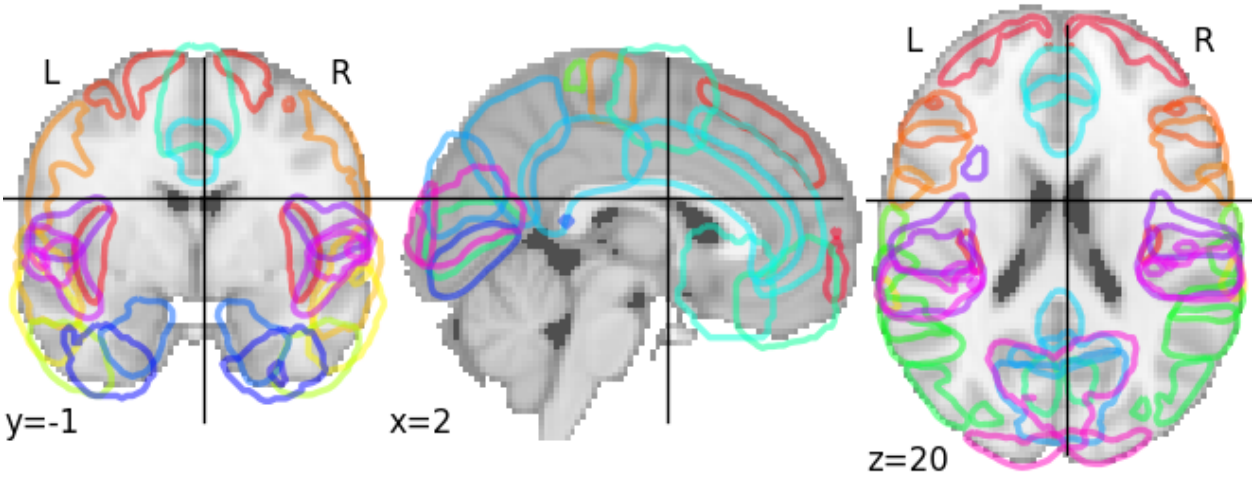
- The raw data was put into */data/openfmri/stroop/sourcedata*
- and the data preprocessed with fmriprep in */data/openfmri/stroop/derivatives*

```
# Libraries
from nilearn import datasets
from nideconv.utils import roi
import pandas as pd
from nilearn import plotting

# Locate the data of the first subject
func = '/data/openfmri/ds000164/derivatives/fmriprep/sub-001/func/sub-001_task-stroop_
↪bold_space-MNI152NLin2009cAsym_preproc.nii.gz'
# ... and confounds extracted by fmriprep
confounds_fn = '/data/openfmri/ds000164/derivatives/fmriprep/sub-001/func/sub-001_
↪task-stroop_bold_confounds.tsv'
# We need to load the confounds and fill nas
confounds = pd.read_table(confounds_fn).fillna(method='bfill')
```

(continues on next page)

(continued from previous page)

```
# We only want to include a subset of confounds
confounds_to_include = ['FramewiseDisplacement', 'aCompCor00',
                        'aCompCor01', 'aCompCor02', 'aCompCor03',
                        'aCompCor04', 'aCompCor05', 'X', 'Y', 'Z',
                        'RotX', 'RotY', 'RotZ']
confounds = confounds[confounds_to_include]


# Use the cortical Harvard-Oxford atlas
atlas_harvard_oxford = datasets.fetch_atlas_harvard_oxford('cort-prob-2mm')
plotting.plot_prob_atlas(atlas_harvard_oxford.maps)
```




```
ts = roi.extract_timecourse_from_nii(atlas_harvard_oxford,
                                     func,
                                     confounds=confounds.values,
                                     t_r=1.5,
                                     high_pass=1./128,
                                     )
```

Now we have a dataframe with a time series for every roi in *atlas_harvard_oxford*

```
print(ts.head())
```

Out:

```
roi   Frontal Pole  Insular Cortex  Superior Frontal Gyrus        ...         Planum␣
↪Temporale  Supracalcarine Cortex  Occipital Pole
time                                                               ...
0.0       0.000682       -0.000467              -0.000190        ...              -0.
↪000555              -0.000355       -0.000059
1.5      -0.010150       -0.001433              -0.017112        ...              -0.
↪023171              -0.000483       -0.023559
3.0      -0.010898       -0.006012              -0.018928        ...              -0.
↪015287               0.002714       -0.016452
4.5      -0.009095       -0.008516              -0.017839        ...              -0.
↪026472               0.002852       -0.021837
6.0      -0.006933       -0.006477              -0.014109        ...              -0.
↪023391              -0.006571       -0.008051
```

(continues on next page)

(continued from previous page)

```
[5 rows x 48 columns]
```

An easy way to save these time series is to use the *to_pickle* functionality of *DataFrame*

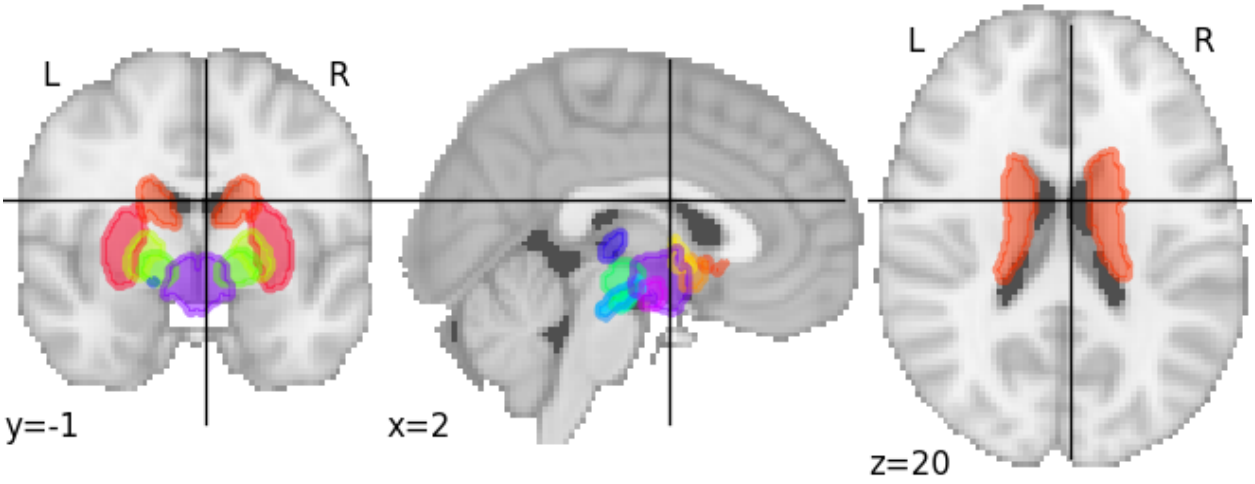
```
ts.to_pickle('/data/openfmri/stroop_task/derivatives/timeseries/sub-001_task-stroop_
↪harvard_oxford.pkl')
```

5.2 Extract time series for all subjects for complete fmriprep'd dataset

nideconv also contains a method to convert an entire fmriprep'd data set to a set of timeseries. This method only needs:

- An atlas in the right format (as supplied with nilearn)
- A BIDS folder containing preprocessed data (e.g., output of fmriprep)
- A BIDS folder containing the raw data.

```
from nideconv.utils import roi
from nilearn import datasets

# Here we use a subcortical atlas
atlas_pauli = datasets.fetch_atlas_pauli_2017()
plotting.plot_prob_atlas(atlas_pauli)
```




```
ts = roi.get_fmriprep_timeseries(fmriprep_folder='/data/openfmri/stroop_task/
↪derivatives/fmriprep/',
                                 sourcedata_folder='/data/openfmri/stroop_task/
↪sourcedata/',
                                 atlas=atlas_pauli)
```

Out:

```
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-001/func/
↪sub-001_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-002/func/
↪sub-002_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
```

(continues on next page)

(continued from previous page)

```
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-003/func/
↪sub-003_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-004/func/
↪sub-004_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-005/func/
↪sub-005_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-006/func/
↪sub-006_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-007/func/
↪sub-007_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-008/func/
↪sub-008_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-009/func/
↪sub-009_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-010/func/
↪sub-010_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-011/func/
↪sub-011_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-012/func/
↪sub-012_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-013/func/
↪sub-013_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-014/func/
↪sub-014_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-015/func/
↪sub-015_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-016/func/
↪sub-016_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-017/func/
↪sub-017_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-018/func/
↪sub-018_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-019/func/
↪sub-019_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-020/func/
↪sub-020_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-021/func/
↪sub-021_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-022/func/
↪sub-022_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-023/func/
↪sub-023_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-024/func/
↪sub-024_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-025/func/
↪sub-025_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-026/func/
↪sub-026_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-027/func/
↪sub-027_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
Extracting signal from /data/openfmri/stroop_task/derivatives/fmriprep/sub-028/func/
↪sub-028_task-stroop_bold_space-MNI152NLin2009cAsym_preproc.nii.gz...
```

We now have a very large dataframe containing time series for every subject and run and for every ROI. The data are indexed with subject and, if applicable session, task, and run.

```
print(ts)
```

Out:

```
roi                           Pu        Ca       NAC       EXA       GPe     ...
↪ VeP       HN       HTH       MN       STH
subject task   time                                                        ...
001     stroop 0.0   -0.024824  0.005317  0.098381  0.125982  0.091782     ...     0.
↪202659  0.158002  0.023573  0.211872  0.033134
               1.5   -0.306827 -0.300921 -0.691543 -1.249696 -0.276813     ...    -2.
↪935104  3.526281  0.326159 -0.957822  0.337725
               3.0    0.126746 -0.263373  0.828887 -1.982217 -0.148301     ...    -1.
↪997062  1.060147  0.368259  0.684619 -0.530833
               4.5   -0.058993 -0.064497  0.071342  1.246516  0.630554     ...     1.
↪453879  3.973592 -0.438184 -0.448751 -0.673395
               6.0   -0.400270 -0.131084 -0.625979 -0.240803 -0.262864     ...     3.
↪694245 -0.485297 -0.227601  0.494319 -1.236625
               7.5   -0.594582 -0.513957 -0.995756  1.024497 -0.293712     ...     0.
↪847354 -4.501187 -0.225672  3.020581 -1.995670
               9.0   -0.449662 -0.591477 -1.091159  0.880994  0.410390     ...     0.
↪563932 -2.134154 -0.126763 -0.774114 -1.798392
              10.5   -0.208486 -0.381288 -0.112447  0.431814  0.024794     ...    -1.
↪370045 -1.974902 -0.281998  1.564937 -0.466028
              12.0   -0.343081 -0.225314 -0.189731 -0.972417 -0.743845     ...    -2.
↪359034  0.857672 -0.424203 -1.777042 -2.303664
              13.5   -0.459105 -0.598026 -0.533962 -1.010868 -0.251734     ...     1.
↪629834 -6.938855  0.692716  0.741312 -1.470699
              15.0   -0.491484 -0.327372  0.607643  0.843527 -0.615236     ...     3.
↪607578  1.481565  0.065486 -0.950975 -2.257750
              16.5   -0.146404 -0.651250 -0.097022  0.017546 -0.249387     ...     0.
↪728848 -1.740604 -0.143918 -0.533464 -0.344056
              18.0   -0.373896 -0.179628  0.044084  0.328200 -0.447820     ...     0.
↪546706 -1.253267 -0.435520  0.604706 -0.854877
              19.5   -0.099276 -0.452165  0.160602 -0.710290 -0.554776     ...     0.
↪615395  0.335330  0.658555 -0.805163 -0.434580
              21.0    0.045956 -0.176265 -0.144740 -0.156893  0.137828     ...    -1.
↪213612 -0.427435 -0.013467 -0.668370 -0.341432
              22.5   -0.081434 -0.363184 -0.214064 -0.216374 -0.051227     ...    -2.
↪587318 -3.152864 -0.183157  0.876120 -0.876575
              24.0   -0.307041 -0.105083  0.264052 -0.353192 -0.095140     ...    -1.
↪014375 -0.148635 -0.235113 -2.459859 -0.780580
              25.5   -0.383516 -0.345184  0.509282 -1.531761  0.028877     ...     1.
↪643677 -5.526221 -0.104641  0.168232 -0.013299
              27.0   -0.267569 -0.117790  0.755515  0.803180  0.629724     ...     1.
↪262726  0.924462  0.100649  1.496751 -0.278082
              28.5   -0.361883 -0.334188 -0.116929  1.694318 -0.393364     ...    -0.
↪108977  0.734137 -0.267002 -0.631139 -0.790549
              30.0   -0.071437 -0.105663 -0.340938 -1.097817 -0.318331     ...     1.
↪783942 -4.189012  0.047490 -0.593999  0.282345
              31.5    0.144421  0.009748 -0.505244 -0.636580 -0.118336     ...     2.
↪621462  1.910466 -0.506244 -0.109885 -0.616836
              33.0   -0.062435  0.101496  0.533229 -0.499491  0.184077     ...    -1.
↪031575 -2.678909 -0.272494  2.096710 -0.417437
              34.5    0.070220 -0.026479 -0.251159  0.556795 -0.838313     ...    -0.
↪098461 -2.043544  0.267077 -0.305758 -0.157330
              36.0    0.080010 -0.178705 -0.350321 -0.854944  0.503817     ...     1.
↪129987  1.555083 -0.257059 -0.804126  1.013317
```

(continues on next page)

(continued from previous page)

```
               37.5  -0.072593  0.045305 -0.010711  0.140783  0.209213    ...      3.
↪374004  0.396039  0.258189 -2.795393 -0.161647
               39.0   0.156933 -0.269197 -0.585724 -0.596147 -0.137166    ...     -1.
↪715686 -4.348575 -0.047123  0.929340  0.358612
               40.5   0.171676 -0.118252  0.627424 -1.188610 -0.020654    ...     -0.
↪497419  0.466622 -0.168192 -1.940330 -0.409833
               42.0  -0.090199  0.020260  0.224642 -0.204530 -0.367158    ...     -2.
↪483469 -3.125973  0.709150  0.275251 -0.056099
               43.5  -0.006724  0.092758  0.290414 -0.494094 -0.220415    ...     -1.
↪992957  2.744478 -0.153984 -3.685024 -1.789079
...                        ...       ...       ...       ...       ...    ...
↪ ...       ...       ...       ...       ...
028     stroop 510.0 -0.000659 -0.100086 -0.662823 -0.322409  0.244794    ...     -1.
↪750508  0.651356 -0.188932  0.636989  0.341632
               511.5 -0.069122 -0.035227 -0.455429 -0.169374 -0.290590    ...      2.
↪424526 -0.472725 -0.763888  0.687077 -0.905188
               513.0 -0.092176 -0.056032 -0.029846 -0.178926  0.465211    ...     -0.
↪925284 -1.346232  0.049967  0.748831  1.101769
               514.5 -0.001127 -0.167024 -0.535128  0.036451 -0.252076    ...      0.
↪153997  0.369322 -0.571099  0.028998 -0.333085
               516.0 -0.039429 -0.040291 -0.106873 -0.646278 -0.834980    ...     -1.
↪088412 -0.603679 -0.542698 -1.538116  2.716484
               517.5  0.042434 -0.238461 -0.047380 -0.029248  0.296041    ...     -1.
↪627384 -3.870078  0.494523 -0.962271  0.790728
               519.0  0.048889 -0.088462 -0.111574 -0.615242 -0.192059    ...      0.
↪380134  1.347656 -0.713702 -0.380996  0.982982
               520.5  0.056310 -0.005925  0.416474 -1.303348  0.051203    ...      0.
↪231380 -1.882325 -0.163210  0.227902  0.992329
               522.0  0.041298  0.022731 -0.033397 -0.009638  0.164090    ...      0.
↪664663  3.253373 -0.069452  1.688823  0.655181
               523.5 -0.138554  0.114313  0.185558  0.647344  0.365004    ...     -0.
↪366816  2.949501 -0.004114 -0.022473 -0.814738
               525.0 -0.077593  0.153181 -0.478577 -0.591277  0.324527    ...     -0.
↪650740  1.498044  0.834692 -0.076548  1.264570
               526.5 -0.113477 -0.135194 -0.503829 -0.718250 -0.469233    ...      1.
↪008015 -0.894386 -0.502698  1.269312 -0.052719
               528.0 -0.033154  0.121374 -0.647306 -0.200863  0.123628    ...     -0.
↪752909 -0.989394  0.154674 -0.399817  0.177255
               529.5  0.460705  0.050999 -0.372112  0.434588 -0.430714    ...      1.
↪226812  1.699830 -0.474807  0.072520 -0.050545
               531.0  0.283997 -0.009623 -0.022878 -0.061585 -0.032400    ...     -1.
↪989329 -0.946382  0.485436  2.038023 -1.482713
               532.5  0.102716 -0.066149  0.152076  0.228064 -0.855818    ...      3.
↪143413 -2.950538  0.037868  0.464920  0.072040
               534.0 -0.318551 -0.158901 -0.327959  0.401055  0.247583    ...     -1.
↪053805 -3.719073 -0.159115  0.454866 -0.161271
               535.5 -0.662931 -0.265144  0.131707 -0.945303 -0.440436    ...     -1.
↪041556 -0.768201  0.373429 -2.610995 -2.657022
               537.0 -0.351289 -0.295574  0.455739 -0.098979 -0.371119    ...      2.
↪282096 -0.211732 -0.817694 -2.343147 -1.606882
               538.5  0.293919  0.030136 -0.139833  1.120084  0.207005    ...      0.
↪165288  1.321914  0.165129 -0.099461 -1.265270
               540.0  0.553214  0.238073  0.364720  1.413141  0.246156    ...     -0.
↪215140 -0.939121  0.541443  1.507328 -0.194477
               541.5 -0.037640 -0.018405  0.254527 -0.403528 -0.385405    ...      0.
↪118627 -0.955283  0.135546 -0.639177 -0.116389
               543.0 -0.303300 -0.109989  0.391246 -0.653175 -0.032351    ...     -2.
↪226433  2.809995  0.310023  0.257117  0.573156
```

(continues on next page)

(continued from previous page)

```
              544.5 -0.620516 -0.087524  0.217855 -0.070457 -0.086806    ...    -0.
↪331080  0.902673  0.287605 -1.478492 -0.899452
              546.0 -0.547904 -0.283596  0.100694  0.351159 -0.177932    ...    -1.
↪710300 -1.809199  0.504293  1.410871 -0.961112
              547.5 -0.397426 -0.314690  0.310524 -0.682668 -0.580231    ...    -1.
↪656477 -2.397482 -0.235764  1.354550 -0.387289
              549.0 -0.148036 -0.194763 -0.604797 -0.638916 -0.164227    ...    -2.
↪277775 -2.042818 -0.085919  0.934715 -0.902066
              550.5 -0.112926 -0.329659 -0.119895 -0.432620 -0.303010    ...    -2.
↪143460 -6.103886 -0.434857  1.696892 -2.466397
              552.0  0.103200 -0.087416 -0.148366  0.119024 -1.550297    ...    -1.
↪571636 -1.190581 -0.096551  0.300252 -2.362502
              553.5  0.021301  0.021162  0.137266  0.237211  0.057831    ...    -0.
↪315415 -0.099618  0.164036 -0.357988 -0.261514

[10360 rows x 16 columns]
(370, 48)
```

Now we can save these time series for later use.

```
ts.to_pickle('/data/openfmri/stroop/derivatives/timeseries/pauli_2017.pkl')
```

5.3 Harvard-Oxford atlas

For later use, we also extract data using the Harvard-Oxford cortical atlas

```
atlas_harvard_oxford = datasets.fetch_atlas_harvard_oxford('cort-prob-2mm')
ts = roi.get_fmriprep_timeseries(fmriprep_folder='/data/openfmri/stroop_task/
↪derivatives/fmriprep/',
                                 sourcedata_folder='/data/openfmri/stroop_task/
↪sourcedata/',
                                 atlas=atlas_harvard_oxford)

ts.to_pickle('/data/openfmri/stroop/derivatives/timeseries/harvard_oxford_2017.pkl')
```

Total running time of the script: (0 minutes 0.000 seconds)

Note: Click *here* to download the full example code

CHAPTER 6

Bayesian hierachical deconvolution of neural signals

So far we have used _frequentist_ methods to estimate the GLMs and deconvolve neural signals.

An alternative statistical paradigm is _Bayesian_ estimation. For a solid, readable introduction into Bayesian statistics, please see the puppy book by John K. Kruschke or the **'Bayesian Methods for Hackers<https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers>'_** -book.

With *nideconv*, you can estimate _Hierachical_ GLMs using the Bayesian framework. This is possible by using Markov Chain Monte Calro sampling using the NUTS (no U-turn sampler) implemented in *STAN <https://mc-stan.org/>*.

First we again simulate som data. To make it interesting, we have a 'Correct' and 'Error'-condition. Importantly, every run has only 1-6 Error trials

```
from nideconv import simulate
conditions = [{'name':'Correct', 'mu_group':.25, 'std_group':.1, 'n_trials':16},
              {'name':'Error', 'mu_group':.5, 'std_group':.1, 'n_trials':(1,6)}]

data, onsets, pars = simulate.simulate_fmri_experiment(conditions,
                                                       n_subjects=9,
                                                       n_runs=1,
                                                       TR=1.5)
```

First, we fit this data using the traditional frequentist GLM with Fourier basis functions:

```
from nideconv import GroupResponseFitter
gmodel = GroupResponseFitter(data, onsets, input_sample_rate=1/1.5, concatenate_
↪runs=False)
gmodel.add_event('Correct', basis_set='fourier', n_regressors=9, interval=[0, 21])
gmodel.add_event('Error', basis_set='fourier', n_regressors=9, interval=[0, 21])
```

We fit the model using ridge regression

```
gmodel.fit(type='ridge', alphas=[1.0])
```

The response we estimate for the group looks pretty good.

```
gmodel.plot_groupwise_timecourses()
```

However, the response estimates for different indivdiuals (solid lines) are quite off from the ground truth (dotted lines)

```
# Now we plot for every subject the estimated HRF
fac = gmodel.plot_subject_timecourses(ci=95, col_wrap=3, size=10, legend=False, n_
↪boot=100)

# ...and the underlying ground truth
import seaborn as sns
from nideconv.utils import convolve_with_function
import numpy as np

t = np.linspace(0, 21, 21*20)
hrf = np.zeros_like(t)
hrf[0] = 1
hrf = convolve_with_function(hrf, 'double_hrf', 20)

for subject, ax in enumerate(fac[0].axes.ravel()):
    subject += 1
    ax.plot(t, hrf * pars.loc[subject, 'Correct'].amplitude, ls='--', lw=1.5, c=sns.
↪color_palette()[0])
    ax.plot(t, hrf * pars.loc[subject, 'Error'].amplitude, ls='--', lw=1.5, c=sns.
↪color_palette()[1])
```

We convert the *GroupResponseModel* to a *HierarchicalBayesianModel* and estimate posterior distributions by MCMC sampling:

```
from nideconv import HierarchicalBayesianModel
model = HierarchicalBayesianModel.from_groupresponsefitter(gmodel)
model.build_model()
model.sample()
```

Plot the individual subject time courses and their Bayesian credible interval (CI)

```
fac = model.plot_subject_timecourses(col_wrap=3, legend=False)


# plot ground truth
for subject, ax in enumerate(fac.axes.ravel()):
    subject += 1
    ax.plot(t, hrf*pars.loc[subject, 'Correct'].amplitude, ls='--', lw=1.5, color=sns.
↪color_palette()[0])
    ax.plot(t, hrf*pars.loc[subject, 'Error'].amplitude, ls='--', lw=1.5, color=sns.
↪color_palette()[1])
```

The Hierarchical Bayesian model performs better in estimating individuals HRFs because it # is regularized: it shrinks the estimates for subjects with little errors towards the group mean.

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER 7

ResponseFitter

class nideconv.**ResponseFitter**(*input_signal*, *sample_rate*, *oversample_design_matrix=20*, *add_intercept=True*, ***kwargs*)

ResponseFitter takes an input signal and performs deconvolution on it. To do this, it requires event times, and possible covariates. ResponseFitter can, for each event type, use different basis function sets, see Event.

Methods

add_confounds(self, name, confound)	Add a timeseries or set of timeseries to the general design matrix as a confound
add_event(self, event_name[, onsets, ...])	create design matrix for a given event_type.
fit(self[, type, cv, alphas, store_residuals])	Regress a created design matrix on the input_data.
get_epochs(self, onsets, interval[, ...])	Return a matrix corresponding to specific onsets, within a given interval.
get_rsq(self)	calculate the rsq of a given fit.
predict_from_design_matrix(self[, X, melt])	predict a signal given a design matrix.
ridge_regress(self[, cv, alphas, ...])	run CV ridge regression instead of ols fit.

add_intercept	
get_basis_functions	
get_original_signal	
get_residuals	
get_standard_errors_timecourse	
get_t_value_timecourses	
get_time_to_peak	
get_timecourses	
plot_design_matrix	
plot_model_fit	
plot_timecourses	

add_confounds(*self*, *name*, *confound*)
Add a timeseries or set of timeseries to the general design matrix as a confound

Parameters

confound [array] Confound of (n_timepoints) or (n_timepoints, n_confounds)

add_event(*self, event_name, onsets=None, basis_set='fir', interval=[0, 10], n_regressors=None, durations=None, covariates=None, **kwargs*)
create design matrix for a given event_type.

Parameters

event_name [string] Name of the event_type, used as key to lookup this event_type's characteristics

****kwargs** [dict] keyward arguments to be internalized by the generated and internalized Event object. Needs to consist of the necessary arguments to create an Event object, see Event constructor method.

fit(*self*, *type='ols'*, *cv=20*, *alphas=None*, *store_residuals=False*)
Regress a created design matrix on the input_data.

Creates internal variables betas, residuals, rank and s. The beta values are then injected into the event_type objects the ResponseFitter contains.

Parameters

type [string, optional] the type of fit to be done. Options are 'ols' for np.linalg.lstsq, 'ridge' for CV ridge regression.

get_epochs(*self*, *onsets*, *interval*, *remove_incomplete_epochs=True*)
Return a matrix corresponding to specific onsets, within a given interval. Matrix size is (n_onsets, n_timepoints_within_interval).

Note that any events that are in the ResponseFitter-object will be regressed out before calculating the epochs.

get_rsq(*self*)
calculate the rsq of a given fit. calls predict_from_design_matrix to predict the signal that has been fit

predict_from_design_matrix(*self*, *X=None*, *melt=False*)
predict a signal given a design matrix. Requires regression to have been run.

Parameters

X [np.array, (timepoints, n_regressors)] the design matrix for which to predict data.

ridge_regress(*self*, *cv=20*, *alphas=None*, *store_residuals=False*)
run CV ridge regression instead of ols fit. Uses sklearn's RidgeCV class

Parameters

cv [int] number of cross-validation folds

alphas [np.array] the alpha/lambda values to try out in the CV ridge regression

CHAPTER 8

GroupResponseFitter

class nideconv.**GroupResponseFitter**(*timeseries*, *onsets*, *input_sample_rate*, *oversample_design_matrix=20*, *confounds=None*, *concatenate_runs=True*, **args*, ***kwargs*)

Can fit a group of individual subjects and/or runs using a high-level interface.

Methods

add_event	
fit	
get_conditionwise_timecourses	
get_rsq	
get_subjectwise_timecourses	
get_t_value_timecourses	
get_time_to_peak	
get_timecourses	
plot_groupwise_timecourses	
plot_subject_timecourses	

CHAPTER 9

NiftiResponseFytter

class nideconv.nifti.**NiftiResponseFitter**(*func_img*, *sample_rate*, *mask=None*, *oversample_design_matrix=20*, *add_intercept=True*, *detrend=False*, *standardize=False*, *confounds_for_extraction=None*, *memory=None*, ***kwargs*)

Methods

add_confounds(self, name, confound)	Add a timeseries or set of timeseries to the general design matrix as a confound
add_event(self, event_name[, onsets, ...])	create design matrix for a given event_type.
fit(self[, type, cv, alphas, store_residuals])	Regress a created design matrix on the input_data.
get_epochs(self, onsets, interval[, ...])	Return a matrix corresponding to specific onsets, within a given interval.
get_rsq(self)	calculate the rsq of a given fit.
predict_from_design_matrix(self[, X])	predict a signal given a design matrix.
ridge_regress(self, *args, **kwargs)	run CV ridge regression instead of ols fit.

add_intercept	
get_basis_functions	
get_original_signal	
get_residuals	
get_standard_errors_timecourse	
get_t_value_timecourses	
get_time_to_peak	
get_timecourses	
plot_design_matrix	
plot_model_fit	
plot_timecourses	

add_confounds(*self*, *name*, *confound*)
Add a timeseries or set of timeseries to the general design matrix as a confound

Parameters

confound [array] Confound of (n_timepoints) or (n_timepoints, n_confounds)

add_event(*self, event_name, onsets=None, basis_set='fir', interval=[0, 10], n_regressors=None, durations=None, covariates=None, **kwargs*)
create design matrix for a given event_type.

Parameters

event_name [string] Name of the event_type, used as key to lookup this event_type's characteristics

****kwargs** [dict] keyward arguments to be internalized by the generated and internalized Event object. Needs to consist of the necessary arguments to create an Event object, see Event constructor method.

fit(*self*, *type='ols'*, *cv=20*, *alphas=None*, *store_residuals=False*)
Regress a created design matrix on the input_data.

Creates internal variables betas, residuals, rank and s. The beta values are then injected into the event_type objects the ResponseFitter contains.

Parameters

type [string, optional] the type of fit to be done. Options are 'ols' for np.linalg.lstsq, 'ridge' for CV ridge regression.

get_epochs(*self*, *onsets*, *interval*, *remove_incomplete_epochs=True*)
Return a matrix corresponding to specific onsets, within a given interval. Matrix size is (n_onsets, n_timepoints_within_interval).

Note that any events that are in the ResponseFitter-object will be regressed out before calculating the epochs.

get_rsq(*self*)
calculate the rsq of a given fit. calls predict_from_design_matrix to predict the signal that has been fit

predict_from_design_matrix(*self*, *X=None*)
predict a signal given a design matrix. Requires regression to have been run.

Parameters

X [np.array, (timepoints, n_regressors)] the design matrix for which to predict data.

ridge_regress(*self*, **args*, ***kwargs*)
run CV ridge regression instead of ols fit. Uses sklearn's RidgeCV class

Parameters

cv [int] number of cross-validation folds

alphas [np.array] the alpha/lambda values to try out in the CV ridge regression

CHAPTER 10

simulate_fmri_experiment

nideconv.simulate.**simulate_fmri_experiment**(*conditions=None*, *TR=1.0*, *n_subjects=1*, *n_runs=1*, *n_trials=40*, *run_duration=300*, *oversample=20*, *noise_level=1.0*, *n_rois=1*, *kernel='double_gamma'*, *kernel_pars={}*)

Simulates an fMRI experiment and returns a pandas DataFrame with the resulting time series in an analysis-ready format.

By default a single run of a single subject is simulated, but a larger number of subjects, runs, and ROIs can also be simulated.

Parameters

conditions [list of dictionaries or *None*] Can be used to customize different conditions. Every conditions is represented as a dictionary in this list and has the following form:

```
[{'name':'Condition A',
  'mu_group':1,
  'std_group':0.1},
  {'name':'Condition B',
  'mu_group':1,
  'std_group':0.1}]
```

mu_group indicates the mean amplitude of the response to this condition across subjects. *std_group* indicates the standard deviation of this amplitude across subjects.

Potentially, customized onsets can also be used as follows:

```
{'name':'Condition A',
 'mu_group':1,
 'std_group':0.1
 'onsets':[10, 20, 30]}
```

TR [float] Indicates the time between volume acquistisions in seconds (Inverse of the sample rate).

n_subjects [int] Number of subjects.

n_runs [int] Number of runs *per subject*.

n_trials [int] Number of trials *per condition per run*. Only used when no custom onsets are provided (see *conditions*).

run_duration [float] Duration of a single run in seconds.

noise_level [float] Standard deviation of Gaussian noise added to time series.

n_rois [int] Number of regions-of-interest. Determines the number of columns of *data*.

Returns

data [DataFrame] Contains simulated time series with subj_idx, run and time (s) as index. Columns correspond to different ROIs

onsets [DataFrame] Contains used event onsets with subj_idx, run and trial type as index.

parameters [DataFrame] Contains parameters (amplitude) of the different event type.

Other Parameters

oversample [int] Determines how many times the kernel is oversampled before convolution. Should usually not be changed.

kernel [str] Sets which kernel to use for response function. Currently only '*double_hrf*' can be used.

Examples

By default, *simulate_fmri_experiment* simulates a 5 minute run with 40 trials for one subject

```
>>> data, onsets, params = simulate_fmri_experiment()
>>> print(data.head())
                     area 1
subj_idx run t
1        1   0.0 -1.280023
             1.0  0.908086
             2.0  0.850847
             3.0 -1.010475
             4.0 -0.299650
>>> print(data.onsets)
                                onset
subj_idx run trial_type
1        1   A             94.317361
             A            106.547084
             A            198.175115
             A             34.941112
             A             31.323272
>>> print(params)
                       amplitude
subj_idx trial_type
1        A                   1.0
         B                   2.0
```

With n_subjects we can increase the number of subjects

```
>>> data, onsets, params = simulate_fmri_experiment(n_subjects=20)
>>> data.index.get_level_values('subj_idx').unique()
Int64Index([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
```

(continues on next page)

(continued from previous page)

```
            20],
           dtype='int64', name='subj_idx')
```

CHAPTER 11

get_fmriprep_timeseries

nideconv.utils.roi.**get_fmriprep_timeseries**(*fmriprep_folder*, *sourcedata_folder*, *atlas*, *atlas_type=None*, *low_pass=None*, *high_pass=0.0078125*, *confounds_to_include=None*, **args*, ***kwargs*)

Extract time series for each subject, task and run in a preprocessed dataset in BIDS format, given all the ROIs in *atlas*.

Currently only *fmriprep* outputs are supported. The *sourcedata_folder* is necessary to look up the TRs of the functional runs.

Parameters

fmriprep_folder: string Path to the folder that contains fmriprep'ed functional MRI data.

sourcedata_folder: string Path to BIDS folder that has been used as input for fmriprep

atlas: sklearn.datasets.base.Bunch This Bunch should contain at least a *maps*-attribute containing a label (3D) or probabilistic atlas (4D), as well as an *label* attribute, with one label for every ROI in the atlas. The function automatically detects which of the two is provided. It extracts a (weighted) time course per ROI. In the case of the probabilistic atlas, the voxels are weighted by their probability (see also the Mappers in nilearn).

atlas_type: str, optional Can be 'labels' or 'probabilistic'. A label atlas should be 3D and contains one unique number per ROI. A Probabilistic atlas contains as many volume as ROIs. Usually, *atlas_type* can be detected automatically.

low_pass: None or float, optional This parameter is passed to signal.clean. Please see the related documentation for details

high_pass: None or float, optional This parameter is passed to signal.clean. Please see the related documentation for details

confounds_to_include: list of strings List of confounds that should be regressed out. By default a limited list of confounds is regressed out: Namely, FramewiseDisplacement, aCompCor00, aCompCor01, aCompCor02, aCompCor03, aCompCor04, aCompCor05, X, Y, Z, RotX, RotY, and RotZ

Examples

```
>>> source_data = '/data/ds001/sourcedata'
>>> fmriprep_data = '/data/ds001/derivatives/fmriprep'
>>> from nilearn import datasets
>>> atlas = datasets.fetch_atlas_pauli_2017()
>>> from nideconv.utils.roi import get_fmriprep_timeseries
>>> ts = get_fmriprep_timeseries(fmriprep_data,
                                 source_data,
                                 atlas)
>>> ts.head()
roi                         Pu        Ca
subject task   time
001     stroop 0.0  -0.023651 -0.000767
               1.5  -0.362429 -0.012455
               3.0   0.087955 -0.062127
               4.5  -0.099711  0.146744
               6.0  -0.443499  0.093190
```

Index