Analysis script as Jupyter notebook (and PDF)

March 23, 2023

1 Analysis code and resulting output

This Jupyter notebook accompanies the publication:

Keyser J, Medendorp WP, Oostwoud Wijdenes L, Selen LPJ.
Late integration of vision and proprioception during perturbed reaches.
Journal of Neurophysiology, accepted March 2023.

All data underlying this publication are available at the Donders Repository, `https://data.donders.ru.nl`. The DOI of the repository is `https://doi.org/10.34973/14xb-gn60`. This notebook re-creates the main analyses and plots.

For descriptions of methods and results please refer to the publication.

1.1 Usage information

The notebook is organized into multiple code cells, which provides a convenient way to plot and analyze only certain parts of it. Please note that cells may rely on variables and functions defined in previous cells: Some output may only be valid if the cells are run in the order presented here. The output is saved below each code cell, and should be re-created if you re-run the corresponding cell.

The raw data of each subject is saved in a HDF5 file, called e.g. `subject03.hdf5`, see folder `data/` in the repository. You can read a description of their contents in file `data_descriptions.txt`.

Note that for "historical" reasons, the cursor shift is referred to as cursor "jump" in the code. In the paper's data set, the cursor did not jump abruptly but was presented either shifted or veridical.

1.2 Tested software versions

This script was last tested with Python 3.7.4, numpy 1.21.5, scipy 1.7.3, pandas 1.3.5, h5py 3.7.0, rpy2 3.3.6, and matplotlib 3.5.3. Note the external dependency on R, last tested with version 3.6.1.

We used Anaconda to install all software dependencies.

1.3 Copyright information

Copyright 2023 Johannes Keyser <`Johannes.Keyser@sport.uni-giessen.de`>

This script is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the license, or (at your option) any later version, see `https://www.gnu.org/licenses/`.

```
[ ]: # Import modules and define functions that are used later in other notebook␣
      ↪cells.
     # (No output expected if running properly.)
     %matplotlib inline
     import os
     import time
     import h5py
     import seaborn as sns
     import pandas as pd
     import matplotlib
     from rpy2 import robjects
     from numpy import mean, nanmean, square, sqrt, sum, sin, cos, gradient, \
                       any, all, isnan, arctan2, linspace, arange, append, \
                       array, asarray, matrix, nan, zeros, ones, vstack, \
                       tile, repeat, concatenate, absolute, unique, nanpercentile, \
                       mod, min, nanmax, abs, diff, setdiff1d, argmin, squeeze, \
                       log, ceil, floor, sort, nonzero, ravel, round, isfinite, \
                       log10
     from numpy.random import randint, rand, seed
     from matplotlib.pyplot import plot, hist, fill_betweenx, fill_between, \
                                   figure, subplots, subplot, xlim, ylim, axes, \
                                   xlabel, ylabel, legend, title, suptitle, text, \
                                   savefig, xticks, yticks, tight_layout, sca, axis
     from scipy.interpolate import interp1d
     from rpy2.robjects.packages import importr
     from rpy2.robjects import pandas2ri, Formula as rformula, vectors as rvectors
     pandas2ri.activate()
     base = importr('base')
     stats = importr('stats')  # for classical Wilcoxon tests
     BF = importr('BayesFactor')  # for Bayesian t-tests and ANOVAs
     assymbl = base.as_symbol

     matplotlib.rcParams['svg.fonttype'] = 'none'
     matplotlib.rcParams['savefig.transparent'] = True
     matplotlib.rcParams['savefig.format'] = 'pdf'
     matplotlib.rcParams['savefig.edgecolor'] = 'none'
     matplotlib.rcParams['savefig.facecolor'] = 'none'
     matplotlib.rcParams['backend'] = 'Qt5Agg'

     # Constants as defined in experiment program.
     # States of interest.
     STATE_MOVEWAIT = 7  # subject is free to reach
     STATE_MOVING = 8  # subject is currently reaching (maybe w/ perturbation)
     STATE_AT_TRGT = 9  # subject just ended reach, probably at target

     # Types of Muscle vibration.
     VIB_NONE = 0  # No vibration
```

```
VIB_BIC = 1  # Biceps vibration
VIB_TRI = 2  # Triceps vibration

# Types of visual cursor "jumps" (but really cursor shifts!).
JMP_NONE = 0  # no cursor jump
JMP_LEFT = 1  # cursor jump left
JMP_RIGHT = 2  # cursor jump right
JMP_HIDE = 3  # no cursor visible at all

# Types of Reach direction.
DIR_AWAY = 0  # away from subject body
DIR_TWRD = 1  # toward subject body

# Types of force fields.
FLD_NONE = 0  # no channel, i.e. free
FLD_CHAN = 1  # mechanical channel active

# Create mappings from numeric conditions to string labels
VIB_NUM2STR = {VIB_NONE: 'none', VIB_BIC: 'bic', VIB_TRI: 'tri'}
JMP_NUM2STR = {JMP_NONE: 'none', JMP_LEFT: 'left',
               JMP_RIGHT: 'right', JMP_HIDE: 'hidden'}
DIR_NUM2STR = {DIR_AWAY: 'away', DIR_TWRD: 'twrd'}
FLD_NUM2STR = {FLD_NONE: 'none', FLD_CHAN: 'chan'}

# Define custom colors for the main conditions.
BIC_ORANGE = '#AA4400'
TRI_GREEN = '#008800'
LEFT_BLUE = '#5555FF'
RIGHT_RED = '#FF3344'

# Define default colors and alpha values for plotting different trial types.
COLOR = {'bic': BIC_ORANGE, 'tri': TRI_GREEN, 'none': 'k', 'out': 'lightred',
         'left': LEFT_BLUE, 'right': RIGHT_RED, 'hidden': 'orange'}
ALPHA = {'bic': .6, 'tri': .5, 'none': .3, 'out': 1, 'left': .6, 'right': .6,
         'hidden': .4}
STYLE = {'bic': '--', 'tri': '--', 'none': '-', 'out': '-',
         'left': '-', 'right': '-', 'hidden': ':'}

# Define a useful subplot layout to split the recurring conditions DIR and JMP.
#                   JMP/cursor
#             left  none  hidden right
#           +-----+-----+-----+-----+
#      away | 0,0 | 0,1 | 0,2 | 0,3 |     In each panel, the perturbation
#  DIR      +-----+-----+-----+-----+      condition is color-coded.
#      twrd | 1,0 | 1,1 | 1,2 | 1,3 |
#           +-----+-----+-----+-----+
```

```
DIRJMP2ROWCOL = {'away': {'left': (0, 0), 'none': (0, 1),
                          'hidden': (0, 2), 'right': (0, 3)},
                 'twrd': {'left': (1, 0), 'none': (1, 1),
                          'hidden': (1, 2), 'right': (1, 3)}}

# Define useful labels with arrows in perturbation direction.
VIB2ARROW = {'bic': 'biceps $\\rightarrow$',
             'tri': 'triceps $\\leftarrow$',
             'none': 'no vibration'}
JMP2ARROW = {'right': 'right shift $\\rightarrow$',
             'left': 'left shift $\\leftarrow$',
             'none': 'no shift', 'hidden': 'no cursor'}

N_SAMPLES = 700  # number of samples for re-sampling (see below)


# ------------- Define functions used throughout the notebook. -------------- #


def find(condition):
    """
    Return indices where ravel(condition) is true; was deprecated in mlab 2.2.
    """
    res, = nonzero(ravel(condition))
    return res


def wilcox_in_r(data1, data2=None, paired=False, alt_hyp='two.sided',
                do_print=False):
    """
    Run a single Wilcoxon test in R and return the result in a dictionary.

    INPUT
    paired: A Boolean indicating whether you want a paired test.
    alt_hyp: A string specifying the alternative hypothesis; must be
             one of "two.sided" (default), "greater" or "less".
    """
    if isinstance(data1, pd.DataFrame):
        data1 = data1.values

    data1 = robjects.FloatVector(data1)

    if data2 is not None:
        if isinstance(data2, pd.DataFrame):
            data2 = data2.values
        data2 = robjects.FloatVector(data2)
        out_r = stats.wilcox_test(x=data1, y=data2, paired=paired,
```

```
                                   alternative=alt_hyp)
    else:
        out_r = stats.wilcox_test(x=data1, alternative=alt_hyp)

    # NOTE Not great to do a hard (re-)labeling of the output,
    # but we always use paired differences, so it's always V.
    out = {'V': out_r[0][0], 'pval': out_r[2][0]}

    if do_print:
        paired_str = 'Paired' if paired else 'Unpaired'
        print('%s, %s Wilcoxon test, V = %d, p = %.g' %
              (paired_str, alt_hyp, out['V'], out['pval']))

    return out


def ttestInR(dataA, dataB=None, paired=False, altHyp='two.sided',
             doPrint=False, breakIfNonNormal=True, testForNormal=True):
    """
    Run a single t-test in R, and return the result in a dictionary.

    INPUT
    paired: a Boolean indicating whether you want a paired test.
    altHyp: a string specifying the alternative hypothesis; must be
            one of "two.sided" (default), "greater" or "less".
    """
    if isinstance(dataA, pd.DataFrame):
        dataA = dataA.values

    dataA = robjects.FloatVector(dataA)
    if testForNormal:
        is_non_normal(dataA, 'dataA', raise_exception=breakIfNonNormal)

    if dataB is not None:

        if type(dataB) is pd.DataFrame:
            dataB = dataB.values

        dataB = robjects.FloatVector(dataB)

        if testForNormal:
            is_non_normal(dataB, 'dataB', raise_exception=breakIfNonNormal)

        outR = stats.t_test(x=dataA, y=dataB, paired=paired,
                            alternative=altHyp)
    else:
        outR = stats.t_test(x=dataA, paired=False, alternative=altHyp)
```

```
    out = {'df': outR[1][0], 'tval': outR[0][0], 'pval': outR[2][0]}

    if doPrint:
        paired_str = 'Paired' if paired else 'Unpaired'
        print('%s, %s t test: t(%d) = %.2f, p = %g' %
              (paired_str, altHyp, out['df'], out['tval'], out['pval']))

    return out


def is_non_normal(data, data_str="", raise_exception=True, alpha=0.05):
    """Run Shapiro-Wilk's test of normality."""
    outR = stats.shapiro_test(data)
    is_non_normal = outR[1][0] < alpha
    warn = "Non-normal data %s!" % data_str \
           if data_str != "" else "Non-normal data!"

    if is_non_normal:
        print(warn)
        if raise_exception:
            raise ValueError(warn)

    return is_non_normal


def BFttestInR(data1, data2=None, paired=False, do_print=False, alt_hyp=None):
    """
    Run a Bayes Factor t-test in R, and return the result in a dictionary.

    INPUT
    paired: a Boolean indicating whether you want a paired test.
    NOTE: Use of alt_hyp not implemented yet.
    """
    if isinstance(data1, pd.DataFrame):
        data1 = data1.values

    data1 = robjects.FloatVector(data1)

    if data2 is not None:
        if isinstance(data2, pd.DataFrame):
            data2 = data2.values

        data2 = robjects.FloatVector(data2)

    if any(isnan(data1)) or (paired and any(isnan(data2))) or \
       (data2 is not None and len(data1) != len(data2)):
```

```
        return {'bf': nan}

    if data2 is not None:
        out_r = BF.ttestBF(x=data1, y=data2, paired=paired)

    else:
        out_r = BF.ttestBF(x=data1)

    out_r = BF.as_data_frame_BFBayesFactor(out_r)
    out = {'bf': out_r['bf'].values}

    if do_print:
        paired_str = 'Paired' if paired else 'Unpaired'
        print('%s BF t-test, bf = %g' % (paired_str, out['bf']))

    return out


def doTestOnGrps(grp1, grp2=None, col=None, test='ttest', paired=False,
                 altHyp='two.sided', breakIfNonNormal=True, printIdx=False):
    """Run tests along 1 or 2 Pandas groupby-iterators, returns a DataFrame."""
    assert not (grp2 is None and paired), "Without grp2, paired makes no sense"

    if grp2 is None:
        grp2 = grp1
        grp2_was_none = True
    else:
        grp2_was_none = False

    results = []

    for (idx1, dat1), (idx2, dat2) in zip(grp1, grp2):
        if idx1 != idx2:
            print(idx1)
            print(idx2)

        assert idx1 == idx2, "GroupBy's are not aligned!"

        if printIdx:
            print(idx1)

        if grp2_was_none:
            data1 = dat1.values if col is None else dat1[col].values
            data2 = None
        else:
            data1, data2 = (dat1.values, dat2.values) if col is None \
                            else (dat1[col].values, dat2[col].values)
```

```python
        if test == 'ttest':
            outDict = ttestInR(dataA=data1, dataB=data2,
                               paired=paired, altHyp=altHyp,
                               breakIfNonNormal=breakIfNonNormal)
        elif test == 'BF':
            outDict = BFttestInR(data1=data1, data2=data2, paired=paired)

        elif test == 'wilcox':
            if paired and (data1.size != data2.size) or \
               (sum(isfinite(data1)) < 2 or sum(isfinite(data2)) < 2):
                data1, data2 = zeros((2, 1)), zeros((2, 1))

            outDict = wilcox_in_r(data1=data1, data2=data2,
                                  paired=paired, alt_hyp=altHyp)
        else:
            raise ValueError("Specify kind of test: ttest, BF, or wilcox.")
        outDict['idx'] = idx1
        results.append(outDict)

    pd_results = pd.DataFrame(results)
    pd_results.set_index('idx')
    return pd_results


def do_test_on_mats(mat1, mat2=None, test='ttest', paired=False,
                    altHyp='two.sided', break_if_nonnormal=True,
                    testForNormal=True):
    """Run statistical tests on rows of 2 numpy matrices, returns DataFrame."""
    assert not (mat2 is None and paired), "Without mat2, paired makes no sense"

    assert mat1.ndim == 2, "Only for 2D matrices."

    if mat2 is not None:
        assert mat2.ndim == 2, "Only for 2D matrices."
        assert mat1.shape[0] == mat2.shape[0], \
            "Mismatched no. of rows mat1/mat2."

    results = []

    for smpl in arange(mat1.shape[0]):
        data1 = mat1[smpl, :]
        data2 = mat2[smpl, :] if mat2 is not None else None

        if test == 'ttest':
            if sum(isfinite(data1)) < 2 or sum(isfinite(data2)) < 2:
                out_dict = {'df': nan, 'tval': nan, 'pval': nan}
```

```python
            else:
                out_dict = ttestInR(dataA=data1, dataB=data2,
                                    paired=paired, altHyp=altHyp,
                                    breakIfNonNormal=break_if_nonnormal,
                                    testForNormal=testForNormal)

        elif test == 'BF':
            out_dict = BFttestInR(data1=data1, data2=data2, paired=paired)

        else:
            if paired and (data1.size != data2.size) or \
               (sum(isfinite(data1)) < 2 or sum(isfinite(data2)) < 2):
                data1, data2 = zeros((2, 1)), zeros((2, 1))

            out_dict = wilcox_in_r(data1=data1, data2=data2,
                                   paired=paired, alt_hyp=altHyp)
        out_dict['idx'] = smpl
        results.append(out_dict)

    pd_results = pd.DataFrame(results)
    pd_results.set_index('idx')
    return pd_results


def bf2df(bf_obj, meta_dict=None):
    """
    Convert entries from anovaBF into a nice Pandas DataFrame.
    Note that values denote Bayes Factors relative to the "Intercept only"␣
 ↪model.
    To pass a valid and meaningful index, use e.g. meta_dict={'pert_time': 101}.
    """
    # allows adding meta information from a dict
    bf_dct = {} if meta_dict is None else meta_dict
    bf_df = BF.as_data_frame_BFBayesFactor(bf_obj)
    bf_df = bf_df.assign(**bf_dct)
    return bf_df


def cut_chunks(df_tf, min_chunk=50):
    """
    Return a binary mask that is True where at least 'min_chunk' neighboring
    values exist, and False elsewhere.
    """
    assert df_tf.dtype == 'bool', "df_tf may only consist of True/False."
    chunk_mask = ones(df_tf.size).astype('bool')

    # Get all "border" indices where p-vals change between non-significant
```

```python
    # and significant; also add the extremes: 0 and last possible index.
    # Note that unique() returns the *sorted* unique elements of an array.
    sig_idxs = unique(concatenate((find(diff(df_tf))+1, [0, df_tf.size])))

    for idx_a, idx_b in zip(sig_idxs[:-1], sig_idxs[1:]):
        chunk = df_tf[idx_a:idx_b]  # note that sig_idxs must be sorted here
        all_non_sig = not any(chunk)
        assert all(chunk) or all_non_sig, "Mixed values in a chunk!"
        if all_non_sig or (sum(chunk) < min_chunk):
            chunk_mask[idx_a:idx_b] = False

    return chunk_mask


def mask_get_onset(sig_mask):
    """Return first index where sig_mask is true (if any), otherwise NaN."""
    if any(sig_mask):
        return find(sig_mask)[0]
    return nan


def pval_get_onset(test_df, min_chunk=10, alpha=0.05):
    """
    Returns the first index of the first neighbouring running p-values that
    are below 'alpha' for at least 'min_chunk' consecutive samples.
    This method was used e.g. by Reichenbach et al. (2013) with min_chunk=4,
    and Franklin et al. (2016) with min_chunk=20.
    NOTE this assumes that test_df has a column name 'pval'.
    """
    psigs_df = test_df < alpha
    sig_mask = cut_chunks(psigs_df, min_chunk=min_chunk)
    return mask_get_onset(sig_mask)


def ez_result_to_df(ez_result, meta_dict=None):
    # allows adding meta information from a dict:
    content = {} if meta_dict is None else meta_dict
    # collect all the ANOVA effects into a DataFrame
    anovatbl = ez_result[0]
    df_list = []
    for ii in range(len(anovatbl[0])):
        content.update({key: vals[ii] for key, vals in anovatbl.items()})
        df_list.append(pd.DataFrame(content, index=[ii]))

    return pd.concat(df_list)
```

```python
def plotPvalsYorX(testDF, offset=5, vert=False,
                  axs=None, sig_alpha=0.05, min_chunk=50, **kwargs):
    """Plot a line where running tests are significant."""
    if axs is None:
        axs = axes()
    pline = offset*ones(testDF.shape[0])
    psigs = testDF < sig_alpha
    sig_mask = cut_chunks(psigs, min_chunk=min_chunk)
    pline[~sig_mask] = nan

    # vertical or horizontal orientation of axes
    xs, ys = (pline, testDF.idx) if vert else (testDF.idx, pline)
    axs.plot(xs, ys, **kwargs)


def get_onset_offset(data, THRESHOLD=0.75):
    """Get onset, offset (first and last index) when crossing the threshold."""
    perturbed = find(data > THRESHOLD)
    if any(perturbed):
        return (perturbed[0], perturbed[-1])
    return (nan, nan)


def get_subject_data(subject_id=None, verbose=False):
    """
    Read relevant subject data, process them, and put into a pandas DataFrame.
    """
    with h5py.File(DATAFILE_PATH % subject_id, 'r') as DATAFILE:
        Data = DATAFILE['/data/']
        Ntrials = int(array(Data['Trials']))
        VibTypes = Data['VibType'][0]
        JmpTypes = Data['JmpType'][0]
        FldTypes = Data['FieldType'][0]
        DirTypes = Data['DirType'][0]
        TrlNum = Data['TrialNumber'][:]
        TgtPosXY = Data['TargetPosition'][0:2, :]
        Fdata = Data['FrameData']
        PosRawXY = Fdata['RobotPosition'][0:2, :, :]  # X,Y handle position
        VelRawXY = Fdata['RobotVelocity'][0:2, :, :]  # X,Y handle velocity
        FrcRawXY = Fdata['HandleForces'][0:2, :, :]  # X,Y measured forces
        CfrcRaw = Fdata['ChannelForce'][:]  # inferred force into channel wall
        RawPhoto = Fdata['PhotoTransistor'][:]  # light sensor data
        RawAccYY = Fdata['VibsAccelYY'][:, :, :]  # vibrator accelerometers
        TrlTime = Fdata['TrialTime'][:]
        ExpState = Fdata['State'][:]
        # switching times (on/off) of perturbations in ms
        switchtimes_bic = round(Data['SwitchTimesBiceps'][:]*1000)
```

```
    switchtimes_tri = round(Data['SwitchTimesTriceps'][:]*1000)
    switchtimes_jmp = round(Data['SwitchTimesCursor'][:]*1000)

# collect various dataframes per trial, and concatenate them later:
force_dfs = []
trial_dfs = []
dropped_trials = []
for trl in range(Ntrials):
    metadata = {'subject': subject_id,
                'VIB': VIB_NUM2STR[VibTypes[trl]],
                'JMP': JMP_NUM2STR[JmpTypes[trl]],
                'DIR': DIR_NUM2STR[DirTypes[trl]],
                'FLD': FLD_NUM2STR[FldTypes[trl]],
                'trial': int(TrlNum[trl])}

    # get actual movement times, based on online experiment state (5 cm/s)
    targt_pos = TgtPosXY[:, trl]
    exp_state = ExpState[:, trl]
    is_mov = find(exp_state == STATE_MOVING)
    if is_mov.size == 0:  # exclude trials without any (real) movement
        if verbose:
            print("\tdropped trial %03d: no movement... " % trl)
        dropped_trials.append(trl)
        continue

    ttime = TrlTime[is_mov, trl]*1000  # trial time in ms
    time_since_mov_onset = ttime - ttime[0]  # time since reach onset

    # [in ms] ignore trials that took too little or too much time
    # target duration was 700 ms
    if not 700-150 < time_since_mov_onset[-1] < 700+300:
        if verbose:
            print("\tdropped trial %03d: wrong reach timing... " % trl)
        dropped_trials.append(trl)
        continue

    # Resample input traces, and making/leaving time points comparable,
    # i.e. create a sampling index at fixed intervals, in "ms precision".
    # E.g. with even, exact virtual sampling points at 0, 1, 2, ... ms after
    # reach onset (or choose a different factor for down-sampling).
    reach_time = arange(0, N_SAMPLES+1, step=1)  # rigid time samples per ms
    # chan_frc = CfrcRaw[is_mov, trl]
    raw_frc_xy = FrcRawXY[:, is_mov, trl]
    raw_pos_xy = PosRawXY[:, is_mov, trl]
    raw_vel_xy = VelRawXY[:, is_mov, trl]
    exp_state = exp_state[is_mov]
    raw_photo = RawPhoto[is_mov, trl]
```

```
        raw_frc_xy = FrcRawXY[:, is_mov, trl]
        raw_acc_yy = RawAccYY[:, is_mov, trl]
        # fun_intrp_ChFrc = interp1d(x=time_since_mov_onset, y=chan_frc,
        #                             kind='linear')
        fun_intrp_state = interp1d(x=time_since_mov_onset, y=exp_state,
                                   kind='linear')
        fun_intrp_photo = interp1d(x=time_since_mov_onset, y=raw_photo,
                                   kind='linear')
        fun_intrp_frc_xy = interp1d(x=time_since_mov_onset, kind='linear',
                                    y=(raw_frc_xy[0, :], raw_frc_xy[1, :]))
        fun_intrp_pos_xy = interp1d(x=time_since_mov_onset, kind='linear',
                                    y=(raw_pos_xy[0, :], raw_pos_xy[1, :]))
        fun_intrp_vel_xy = interp1d(x=time_since_mov_onset, kind='linear',
                                    y=(raw_vel_xy[0, :], raw_vel_xy[1, :]))
        fun_intrp_acc_yy = interp1d(x=time_since_mov_onset, kind='linear',
                                    y=(raw_acc_yy[0, :], raw_acc_yy[1, :]))

        # interpChFrc = nan * zeros(reach_time.size)
        interp_state = nan * zeros(reach_time.size)
        interp_photo = nan * zeros(reach_time.size)
        interp_frc_xy = nan * zeros((2, reach_time.size))
        interp_pos_xy = nan * zeros((2, reach_time.size))
        interp_vel_xy = nan * zeros((2, reach_time.size))
        interp_acc_yy = nan * zeros((2, reach_time.size))
        # only interpolate the ongoing reach, to preserve time axis
        valid_idx = reach_time < time_since_mov_onset[-1]
        # interpChFrc[valid_idx] = fun_intrp_ChFrc(reach_time[valid_idx])
        interp_state[valid_idx] = fun_intrp_state(reach_time[valid_idx])
        interp_photo[valid_idx] = fun_intrp_photo(reach_time[valid_idx])
        interp_frc_xy[:, valid_idx] = fun_intrp_frc_xy(reach_time[valid_idx])
        interp_pos_xy[:, valid_idx] = fun_intrp_pos_xy(reach_time[valid_idx])
        interp_vel_xy[:, valid_idx] = fun_intrp_vel_xy(reach_time[valid_idx])
        interp_acc_yy[:, valid_idx] = fun_intrp_acc_yy(reach_time[valid_idx])

        # Rotate the measurements by the angle from start to target; for ATI
        # force transducer, this yields the similar values (coordinate frame)
        # as the (inferred) ChannelForce that are relative to reach direction:
        # Positive forces are leftward/ccw into channel from-start-to-target,
        # negative forces are rightward/cw (w.r.t the reach direction).
        # Thus, in plots with force on a Y-axis, only a (mental) 90° rotation
        # is necessary to imagine the actual force direction, but keep in mind
        # that it's with respect to the current reach direction.
        start_pos = PosRawXY[:, is_mov[0], trl]
        targt_pos = TgtPosXY[:, trl]
        chan_angle = arctan2(targt_pos[0] - start_pos[0],
                             targt_pos[1] - start_pos[1])
        rot_mat = matrix([[cos(-chan_angle), sin(-chan_angle)],
```

```
                  [-sin(-chan_angle), cos(-chan_angle)]])
rot_frc_xy = rot_mat * interp_frc_xy

# Rotate positions: X coordinate is perpendicular to channel,
#                   Y coordinate is radially along channel.
half_way = tile(matrix(targt_pos + start_pos).T / 2,
                [1, reach_time.size])
rot_pos_xy = rot_mat * (interp_pos_xy - half_way)

# Lateral force values should indicate (counter-)clock-wise force
# into channel walls *irrespective* of the current reach direction.
# Until here (see above), force is w.r.t reach direction in channel:
# forces <0 are cw, and forces >0 are ccw w.r.t reach direction.
# We want that for direction away, forces <0 indicate ccw instead cw.
mult_sign = -1 if DirTypes[trl] == DIR_AWAY else +1
rot_frc_xy[0] = rot_frc_xy[0]*mult_sign

# Calculate reach speed from x,y velocity.
reach_speed = sqrt(square(interp_vel_xy[0, :]) +
                   square(interp_vel_xy[1, :]))

# Re-express sensor signals w.r.t their baseline and maximum [0, 1].
int0, int1 = 0, 100
baseline0 = nanmean(interp_acc_yy[0, int0:int1])
baseline1 = nanmean(interp_acc_yy[1, int0:int1])
basephoto = nanmean(interp_photo[int0:int1])
proc_acc0 = abs(interp_acc_yy[0, :] - baseline0)
proc_acc1 = abs(interp_acc_yy[1, :] - baseline1)
proc_photo = abs(interp_photo - basephoto)
proc_acc0 = proc_acc0 / nanmax(proc_acc0)
proc_acc1 = proc_acc1 / nanmax(proc_acc1)
proc_photo = proc_photo / nanmax(proc_photo)

# Detect onsets and offsets from the sensors (for cursor/vibration).
on0, off0 = get_onset_offset(proc_acc0, THRESHOLD=0.75)
on1, off1 = get_onset_offset(proc_acc1, THRESHOLD=0.75)
onP, offP = get_onset_offset(proc_photo, THRESHOLD=0.75)
time_since_photo = reach_time - round(onP) if ~isnan(onP) else nan

# Reject trials with out-of-sync combined perturbation ONsets.
CUTOFF = 10  # in ms
if (metadata['JMP'] != 'hidden') and \
   ((metadata['VIB'] == 'tri' and abs(onP - on0) > CUTOFF) or
    (metadata['VIB'] == 'bic' and abs(onP - on1) > CUTOFF)):
    if verbose:
        print("\tdropped trial %03d: onset abs(photo-vib) > %d ms... " %
              (trl, CUTOFF))
```

```
            dropped_trials.append(trl)
            continue

        # Finally, stack the data of interest into DataFrames.
        # create a data frame with all the sample data
        force_dict = metadata.copy()
        force_dict.update({'reach_time': reach_time,
                           'pert_time': time_since_photo,
                           # 'chanForce': interpChFrc,
                           'rotForceX': squeeze(asarray(rot_frc_xy[0, :])),
                           'rotPosX': squeeze(asarray(rot_pos_xy[0, :])),
                           'rotPosY': squeeze(asarray(rot_pos_xy[1, :])),
                           'reach_speed': reach_speed,
                           # 'rotForceY': squeeze(asarray(rot_frc_xy[1, :])),
                           'state': interp_state,
                           'photo': interp_photo,
                           'intAcc0': squeeze(asarray(interp_acc_yy[0, :])),
                           'intAcc1': squeeze(asarray(interp_acc_yy[1, :])),
                           'prcAcc0': proc_acc0,
                           'prcAcc1': proc_acc1})
        force_dfs.append(pd.DataFrame(force_dict))

        # Create a data frame that details on a per-trial basis,
        # mainly switching times and the corresponding detected on- and offsets.
        trial_dict = metadata.copy()
        trial_dict.update({'SW_ON_BIC': int(switchtimes_bic[0, trl]),
                           'SW_OFF_BIC': int(switchtimes_bic[1, trl]),
                           'SW_ON_TRI': int(switchtimes_tri[0, trl]),
                           'SW_OFF_TRI': int(switchtimes_tri[1, trl]),
                           'SW_ON_JMP': int(switchtimes_jmp[0, trl]),
                           'SW_OFF_JMP': int(switchtimes_jmp[1, trl]),
                           'TH_ON_BIC': on1,
                           'TH_OFF_BIC': off1,
                           'TH_ON_TRI': on0,
                           'TH_OFF_TRI': off0,
                           'TH_ON_JMP': onP,
                           'TH_OFF_JMP': offP,
                           'MOV_END': time_since_mov_onset[-1]})
        trial_dict.update({'trial': [trial_dict['trial']]})
        trial_dfs.append(pd.DataFrame(trial_dict))

    print('%d (%.1f%%) trials dropped. ' % (len(dropped_trials),
                                            len(dropped_trials)/640*100))
    # concatenate all the data frames
    df_data = pd.concat(force_dfs)
    idx = ['subject', 'VIB', 'JMP', 'DIR', 'FLD', 'trial', 'pert_time']
    df_data.set_index(idx, inplace=True)
```

```
    df_data.sort_index(inplace=True)

    # concatenate all the trial frames
    df_trial = pd.concat(trial_dfs)
    idx = ['subject', 'VIB', 'JMP', 'DIR', 'FLD', 'trial']
    df_trial.set_index(idx, inplace=True)
    df_trial.sort_index(inplace=True)

    return (df_data, df_trial)
```

[2]:
```
# Define path to load subject data from.
DATAFILE_PATH = 'data/%s.hdf5'  # may need adjustment

# Load all subject data into Pandas dataframes.
ALL_SUBJECTS = ['subject%02d' % ss for ss in range(1, 22+1)]
# Load all available data.
ALL_MOVEMENT_DATA, ALL_TRIAL_DETAILS = [], []
for sbj in ALL_SUBJECTS:
    print(f"Loading subject {sbj}...'")
    sbjDF, trlDF = get_subject_data(subject_id=sbj)
    ALL_MOVEMENT_DATA.append(sbjDF)
    ALL_TRIAL_DETAILS.append(trlDF)

ALL_MOVEMENT_DATA = pd.concat(ALL_MOVEMENT_DATA)
ALL_TRIAL_DETAILS = pd.concat(ALL_TRIAL_DETAILS)
ALL_MOVEMENT_DATA.sort_index(inplace=True)
ALL_TRIAL_DETAILS.sort_index(inplace=True)
```

```
Loading subject subject01…'
6 (0.9%) trials dropped.
Loading subject subject02…'
5 (0.8%) trials dropped.
Loading subject subject03…'
6 (0.9%) trials dropped.
Loading subject subject04…'
7 (1.1%) trials dropped.
Loading subject subject05…'
10 (1.6%) trials dropped.
Loading subject subject06…'
26 (4.1%) trials dropped.
Loading subject subject07…'
35 (5.5%) trials dropped.
Loading subject subject08…'
14 (2.2%) trials dropped.
Loading subject subject09…'
1 (0.2%) trials dropped.
Loading subject subject10…'
```

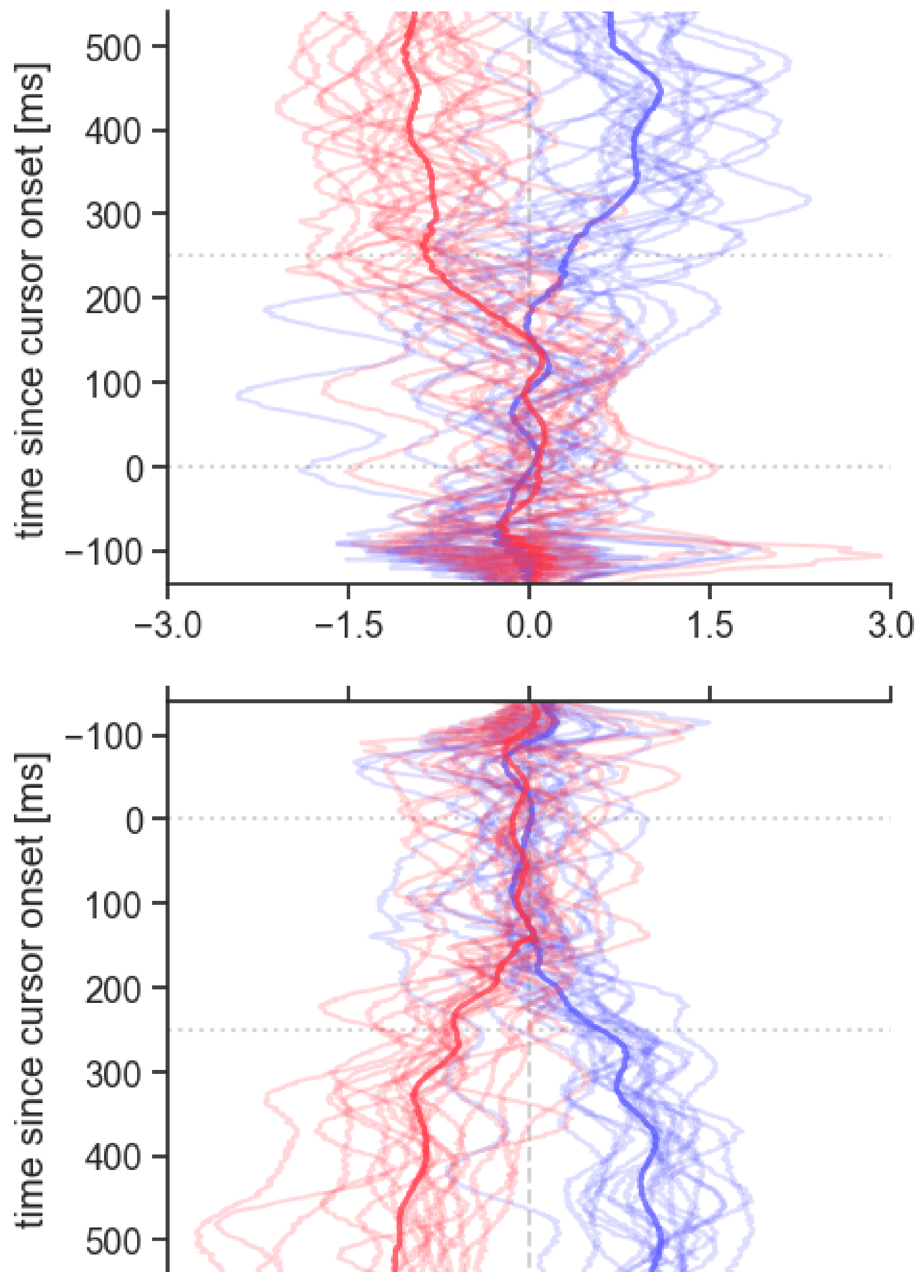
```
8 (1.2%) trials dropped.
Loading subject subject11…'
2 (0.3%) trials dropped.
Loading subject subject12…'
6 (0.9%) trials dropped.
Loading subject subject13…'
13 (2.0%) trials dropped.
Loading subject subject14…'
4 (0.6%) trials dropped.
Loading subject subject15…'
1 (0.2%) trials dropped.
Loading subject subject16…'
25 (3.9%) trials dropped.
Loading subject subject17…'
0 (0.0%) trials dropped.
Loading subject subject18…'
16 (2.5%) trials dropped.
Loading subject subject19…'
11 (1.7%) trials dropped.
Loading subject subject20…'
6 (0.9%) trials dropped.
Loading subject subject21…'
5 (0.8%) trials dropped.
Loading subject subject22…'
6 (0.9%) trials dropped.
```

[33]:

```
# Figure 2: Plot force traces of an example subject.
#           The x-axes are force in [N].
# Select subject:
SUBJECT = 'subject18'  # used example 'subject18' in the paper
# Select condition combination...
# ...for fig. 2 a,c: cursor jumps w/o vibration:
VIB_DIR = (('none', 'twrd'), ('none', 'away'))
# ...for fig. 2 b,d: vibration (and jumps):
# VIB_DIR = (('tri', 'twrd'), ('bic', 'away'))

SBJ_DF = ALL_MOVEMENT_DATA.xs(SUBJECT, level='subject')
FORCE_DV = 'rotForceX'  # dependent force variable
PERT_ONSET = 0
PERT_OFFSET = PERT_ONSET + 250
XLIMS = (-3, 3)
YLIMS = (-140, 540)

sns.set(font_scale=1.2)
with sns.axes_style("ticks"):
    FIG, AXS = subplots(2, 1, figsize=(5, 7))
```

```
    if YLIMS is not None:  # set for same scale across subjects
        ylim(YLIMS)
        sns.despine()

DIR2ROW = {'away': AXS[0], 'twrd': AXS[1]}

for (vib, jmp, _dir, fld), df_cnd in \
    SBJ_DF.groupby(level=['VIB', 'JMP', 'DIR', 'FLD']):

    if jmp in ('none', 'hidden'):
        continue

    if (vib, _dir) not in VIB_DIR or fld != 'chan':
        continue

    clr = COLOR[vib] if vib in ('bic', 'tri') else COLOR[jmp]
    alf = ALPHA[vib] if vib in ('bic', 'tri') else ALPHA[jmp]

    muNul = SBJ_DF.loc[('none', 'none', _dir, 'chan'), :].groupby(
        by='pert_time')[FORCE_DV].mean()
    muCnd = df_cnd.groupby(level='pert_time')[FORCE_DV].mean()
    muDff = muCnd - muNul
    time_dff = muDff.index.get_level_values('pert_time')

    ax = DIR2ROW[_dir]

    # plot individual trials
    for _, df_trl in df_cnd[FORCE_DV].groupby(by='trial'):
        df_dff = df_trl - muNul
        time_trl = df_dff.index.get_level_values('pert_time')
        ax.plot(df_dff, time_trl, '-', color=COLOR[jmp], alpha=alf/3)

    # plot mean over trials
    ax.plot(muDff, time_dff, '-', lw=2, color=COLOR[jmp], alpha=0.75)
    if vib != 'none':
        ax.plot(muDff, time_dff, '--', lw=2, color=COLOR[vib], alpha=0.75)


for ax in AXS.ravel():
    ax.set_xticks((linspace(XLIMS[0], XLIMS[1], 5)))
    ax.set_ylim(YLIMS)
    ax.set_xlim(XLIMS)
    ax.plot((0, 0), YLIMS, '--', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_ONSET,)*2, ':', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_OFFSET,)*2, ':', color='lightgrey', zorder=0)
    ax.set_ylabel('time since cursor onset [ms]')
```

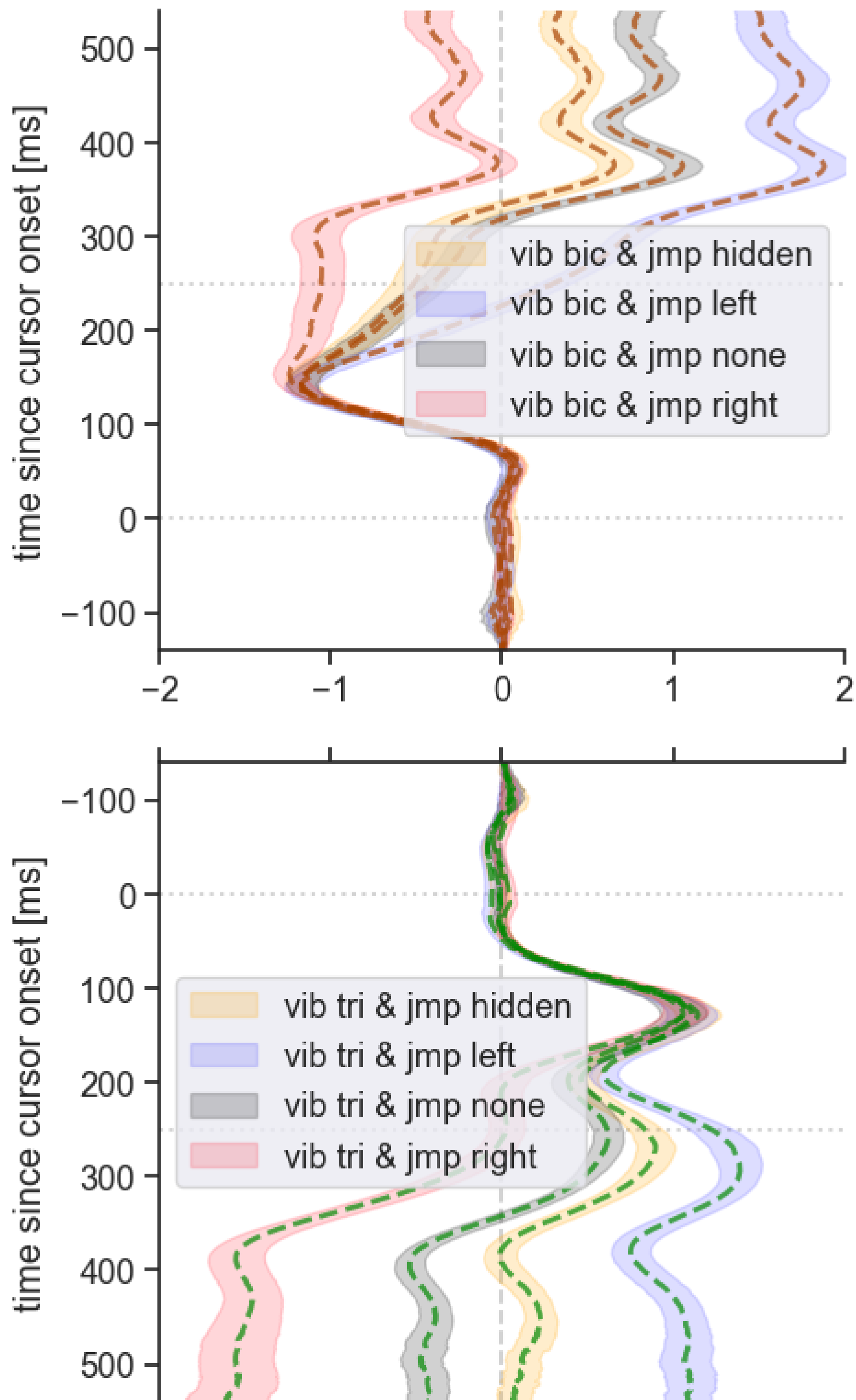
```
for ax in [AXS[1]]:
    sns.despine(ax=ax, top=False, bottom=True)
    ax.invert_yaxis()
    ax.xaxis.tick_top()
    ax.set_xticklabels(())

tight_layout()
```



[28]:

```
# Plot figure 3 a,d and b,e: force means (+/- sem) over subjects.
# Panels are defined by which conditions to exclude:
# EXCLUDE_VIB_DIR = [('tri', 'twrd'), ('bic', 'away')]  # fig 3 a,d
EXCLUDE_VIB_DIR = [('none', 'twrd'), ('none', 'away')]  # fig 3 b,e

# Normalize perturbation effects relative to trials without perturbation
# but with visible, veridical cursor.
GRP_VAR = ['subject', 'DIR']
GRP_NRM = ALL_MOVEMENT_DATA.xs('chan', level='FLD',
                               drop_level=False).groupby(
                                   level=GRP_VAR)['rotForceX']
DIFF_DFS = []
for (sbj, _dir), df in GRP_NRM:
    # get trials without vibration or jump as 'null' reference
    dfNul = df.xs(('none', 'none'), level=('VIB', 'JMP'), drop_level=False)
    muNul = dfNul.groupby(level='pert_time').mean()
    # subtract null reference from each perturbation combination
    grpPert = df.groupby(level=['VIB', 'JMP', 'trial'])
    for (vib, jmp, trl), dfCnd in grpPert:
        muCnd = dfCnd.groupby(level='pert_time').mean()
        muDff = muCnd - muNul
        dfDff = pd.DataFrame({'norm_force': muDff, 'subject': sbj,
                              'VIB': vib, 'JMP': jmp, 'DIR': _dir,
                              'trial': trl}, index=muCnd.index)
        DIFF_DFS.append(dfDff)

# Compute aggregates within subjects.
NORM_FORCE = pd.concat(DIFF_DFS)
NORM_FORCE.reset_index(inplace=True)
NORM_FORCE.set_index(['subject', 'VIB', 'JMP',
                      'DIR', 'trial', 'pert_time'], inplace=True)
# mean over trials
MEAN_FORCE = NORM_FORCE.groupby(level=['subject', 'VIB', 'JMP',
                                       'DIR', 'pert_time']).mean()
# sem over trials
SEMS_FORCE = NORM_FORCE.groupby(level=['subject', 'VIB', 'JMP',
                                       'DIR', 'pert_time']).sem()

PERT_ONSET = 0
PERT_OFFSET = PERT_ONSET + 250
XLIMS = (-2, 2)
YLIMS = (-140, 540)

sns.set(font_scale=1.2)
with sns.axes_style("ticks"):
    FIG, AXS = subplots(2, 1, figsize=(5, 8))
    sns.despine()
```

```python
DIR2AXS = {'away': AXS[0], 'twrd': AXS[1]}

MEAN_CND = MEAN_FORCE.groupby(['VIB', 'JMP', 'DIR', 'pert_time']).mean()
SEM_CND = MEAN_FORCE.groupby(['VIB', 'JMP', 'DIR', 'pert_time']).sem()
GRP_CND = MEAN_CND.groupby(['VIB', 'JMP', 'DIR'])


# Plot combined perturbation conditions; mean (+/- sem) over subjects.
for (vib, jmp, _dir), df in GRP_CND:

    if (vib, _dir) in EXCLUDE_VIB_DIR:
        continue

    ax = DIR2AXS[_dir]
    lbl = f"vib {vib} & jmp {jmp}"
    time_axis = df.index.get_level_values('pert_time')
    mean_plot = df.values
    sem_plot = SEM_CND.xs((vib, jmp, _dir), level=('VIB', 'JMP', 'DIR'))
    sem_x1 = (mean_plot - sem_plot).squeeze()
    sem_x2 = (mean_plot + sem_plot).squeeze()
    ax.fill_betweenx(time_axis, sem_x1, sem_x2, color=COLOR[jmp], alpha=.2,␣
↪label=lbl)
    ax.plot(mean_plot, time_axis, linestyle=STYLE[vib], color=COLOR[vib],
            alpha=.7, lw=2)
    ax.legend(loc=0)

for ax in AXS.ravel():
    ax.set_xticks((linspace(XLIMS[0], XLIMS[1], 5)))
    ax.set_ylim(YLIMS)
    ax.set_xlim(XLIMS)
    ax.plot((0, 0), YLIMS, '--', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_ONSET,)*2, ':', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_OFFSET,)*2, ':', color='lightgrey', zorder=0)
    ax.set_ylabel('time since cursor onset [ms]')

for ax in [AXS[1]]:
    sns.despine(ax=ax, top=False, bottom=True)
    ax.invert_yaxis()
    ax.xaxis.tick_top()
    ax.set_xticklabels(())

tight_layout()
```




```
[ ]: # Compute group-level Bayesian ANOVA to check for VIB, JMP and interaction.
     # This will be used below to plot fig. 3 c,f.
     # NOTE that this may take several minutes to compute.
```

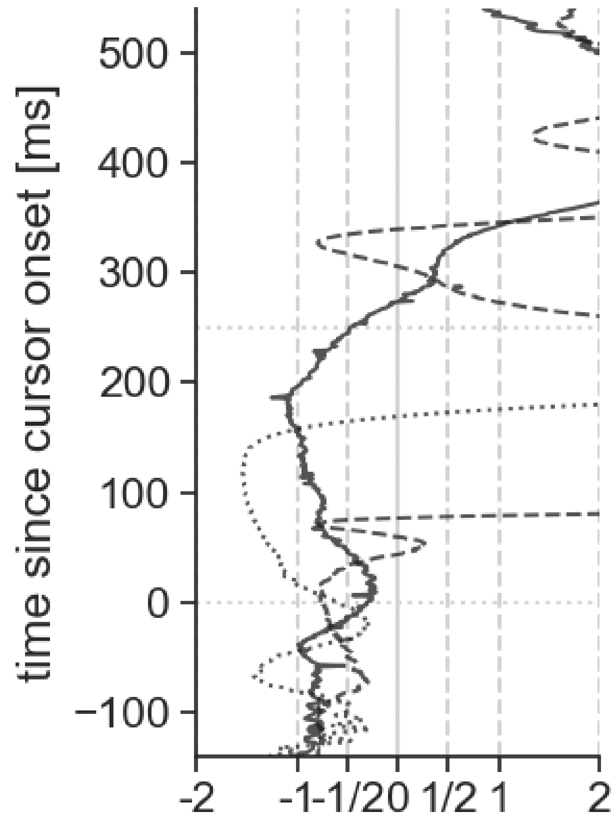
```
GRP_VAR = ['subject', 'DIR']

ALL_CHAN = ALL_MOVEMENT_DATA.drop('none', level='FLD')
SBJ_MEANS = ALL_CHAN.groupby(
    level=['subject', 'DIR', 'VIB', 'JMP', 'pert_time'])['rotForceX'].mean()

list_banova_dfs = []
for (dir_, tt), df in SBJ_MEANS.groupby(['DIR', 'pert_time']):
    print((dir_, tt))  # print some indication of progress...

    df4banova = df.reset_index()

    if any(df4banova.isna()):  # don't calculate with incomplete datasets
        continue

    for col in ('VIB', 'JMP', 'subject'):  # convert into factors
        df4banova[col] = df4banova[col].astype('category')

    # set up mixed-effects BANOVA; subjects are a (repeated) random factor
    bfres = BF.anovaBF(rformula('rotForceX ~ VIB + JMP + VIB:JMP + subject'),
                       whichRandom='subject', data=df4banova, progress=False,
                       iterations=20000)
    list_banova_dfs.append(bf2df(bfres,
                                 meta_dict={'pert_time': tt, 'DIR': dir_}))

banovas_df = pd.concat(list_banova_dfs)
banovas_df.reset_index(inplace=True)
banovas_df.set_index(['DIR', 'pert_time', 'index'], inplace=True)
banovas_df.index.names = ['DIR', 'pert_time', 'effect']
```

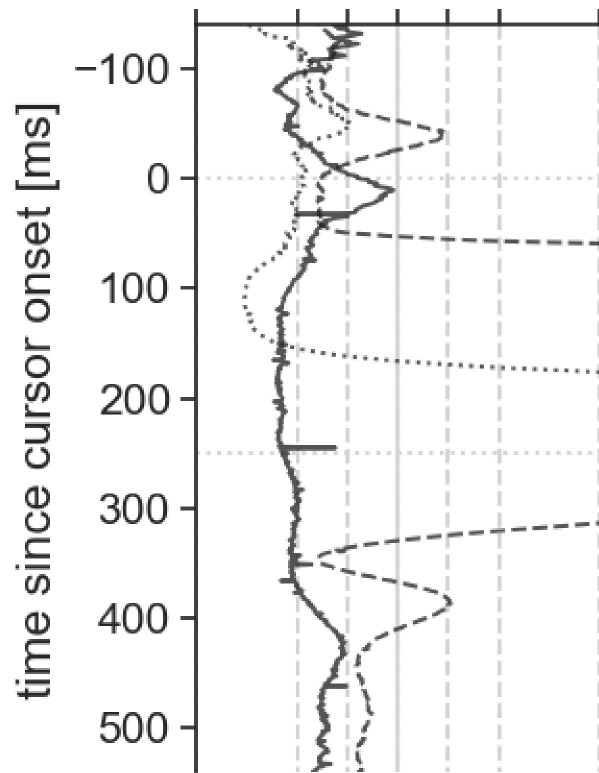
[12]:
```
# Fig. 3 c,f: Plot group-level BANOVA effects from cell above.
PERT_ONSET = 0
PERT_OFFSET = PERT_ONSET + 250
YLIMS = (-140, 540)
MIN_CHUNK = 10

# Use some labeling thresholds for BF as suggested by Jeffreys (1961)
BF_LABELS = ['-2', '-1', '-1/2', '0', '1/2', '1', '2']
BF_LEVELS = log10(array([10**eval(ll) for ll in BF_LABELS]))
XLIMS = (BF_LEVELS.min(), BF_LEVELS.max())

sns.set(font_scale=1.5)
with sns.axes_style("ticks"):
    FIG, AXS = subplots(2, 1, figsize=(3.5, 9))
    sns.despine()

DIREFF2AXS = {'away': {'VIB': AXS[0], 'JMP': AXS[0], 'VIB:JMP': AXS[0]},
```

```
             'twrd': {'VIB': AXS[1], 'JMP': AXS[1], 'VIB:JMP': AXS[1]}}

EFF2LST = {'VIB': '--', 'JMP': ':', 'VIB:JMP': '-'}

print(f"Group-level effect onsets with min_chunk = {MIN_CHUNK}:")
for dir_ in ('away', 'twrd'):
    dir_data = banovas_df.xs(dir_, level='DIR')['bf']
    for eff in ('VIB', 'JMP', 'VIB:JMP'):
        if eff == 'VIB:JMP':
            # get BF favoring interaction relative to model with main effects
            bf_intr = dir_data.xs('VIB + JMP + VIB:JMP + subject',␣
 ↪level='effect')
            bf_base = dir_data.xs('VIB + JMP + subject', level='effect')
            bflogs = log10(bf_intr / bf_base)
        else:
            # get BF favoring main effect over model with just intercept
            bflogs = log10(dir_data.xs(eff + ' + subject', level='effect'))

        ax = DIREFF2AXS[dir_][eff]
        ls = EFF2LST[eff]
        dir_time = bflogs.index
        ax.plot(bflogs, dir_time, color='k', alpha=.75, ls=ls)

        # print 1st crossing into substantial evidence for min_chunk samples
        # cut off at 'substantial' evidence 10**(1/2)
        bfsigs = bflogs > log10(10**(1/2))
        bfmask = cut_chunks(bfsigs, min_chunk=MIN_CHUNK)
        on_idx = find(bfmask)[0] if any(bfmask) else nan
        onset = nan if isnan(on_idx) else bfsigs.index[on_idx]
        print('%s, %s at %g ms' % (dir_, eff, onset))


for ax in AXS.ravel():
    ax.set_ylim(YLIMS)
    ax.set_xlim(XLIMS)
    ax.plot(XLIMS, (PERT_ONSET,)*2, ':', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_OFFSET,)*2, ':', color='lightgrey', zorder=0)
    ax.set_ylabel('time since cursor onset [ms]')
    ax.set_xticks(BF_LEVELS)
    ax.set_xticklabels(BF_LABELS)
    for bb in BF_LEVELS:
        ax.plot((bb,)*2, YLIMS, '--', color='lightgrey', zorder=0)
    ax.plot((log10(1),)*2, YLIMS, '-', color='lightgrey', zorder=0)

for ax in [AXS[1]]:
    sns.despine(ax=ax, top=False, bottom=True)
    ax.invert_yaxis()
```

```
    ax.xaxis.tick_top()
    ax.set_xticklabels(())

tight_layout()
```

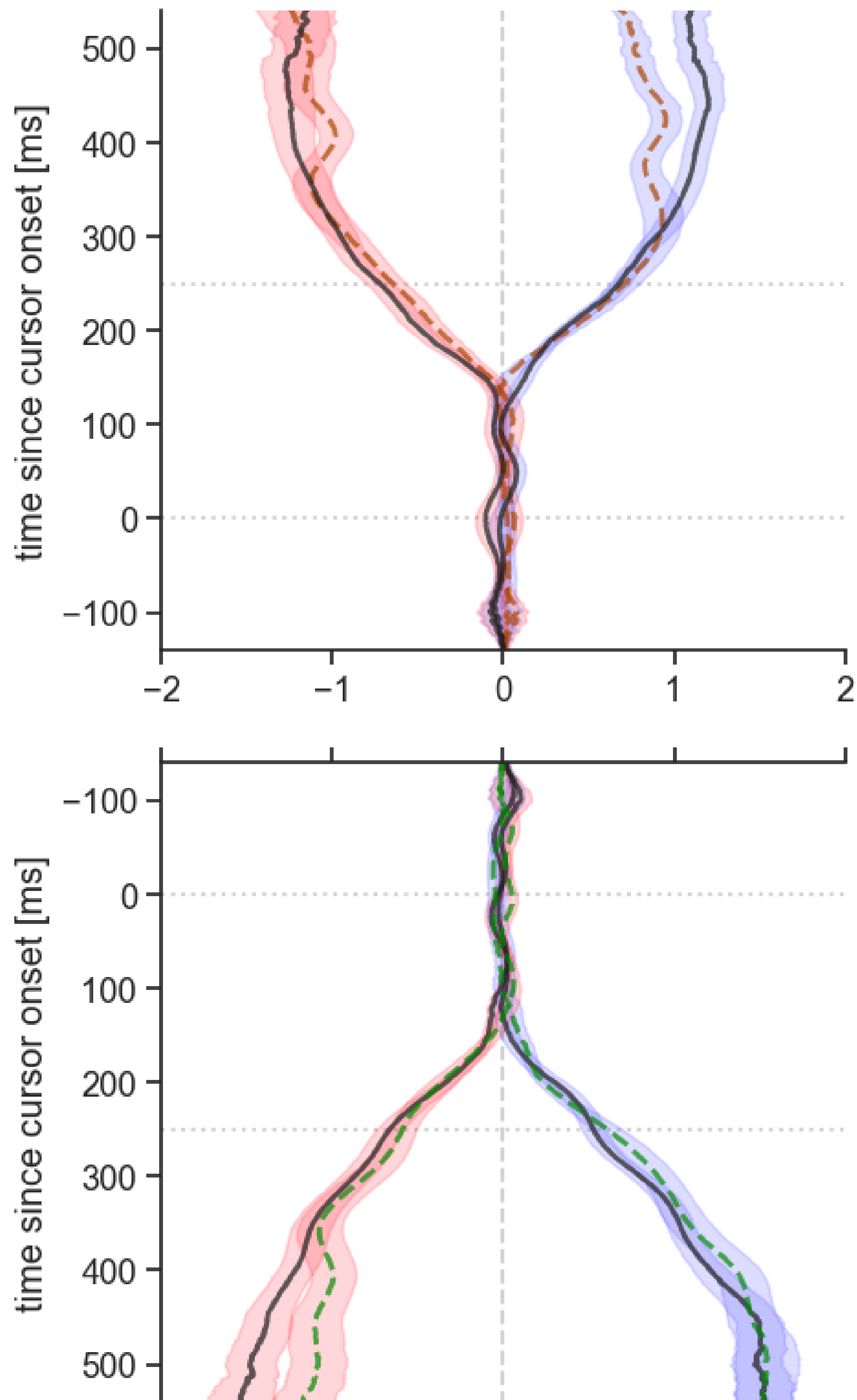
```
Group-level effect onsets with min_chunk = 10:
away, VIB at 78 ms
away, JMP at 173 ms
away, VIB:JMP at 322 ms
twrd, VIB at 55 ms
twrd, JMP at 170 ms
twrd, VIB:JMP at nan ms
```


time since cursor onset [ms]
500
400
300
200
100
0
−100
-2
-1
-1/2
0
1/2
1
2


time since cursor onset [ms]
−100
0
100
200
300
400
500

[34]:
```
# Plot figure 4 a,c: Effect of cursor left/right shifts with/without vibration.

# Normalize cursor shifts L/R relative to matching trials without cursor shift.
# This entails that there are 2 null conditions: With or without vibration.
GRP_VAR = ['subject', 'DIR']
GRP_NRM = ALL_MOVEMENT_DATA.xs('chan', level='FLD',
                              drop_level=False).groupby(
                                  level=GRP_VAR)['rotForceX']
DIFF_DFS = []  # for force differences per condition
for (sbj, _dir), df1 in GRP_NRM:
    # subtract null reference from each perturbation combination
    for (vib, jmp, trl), df2 in df1.groupby(level=['VIB', 'JMP', 'trial']):
        # get trials without jump as 'null' reference
        dfNul = df1.xs((sbj, vib, 'none', _dir),
                       level=('subject', 'VIB', 'JMP', 'DIR'))
        muNul = dfNul.groupby(level='pert_time').mean()
        muCnd = df2.groupby(level='pert_time').mean()
        muDff = muCnd - muNul
        dfDff = pd.DataFrame({'norm_force': muDff, 'subject': sbj,
                              'VIB': vib, 'JMP': jmp, 'DIR': _dir,
                              'trial': trl}, index=muCnd.index)
        DIFF_DFS.append(dfDff)

# Compute aggregates within subjects.
NORM_FORCE = pd.concat(DIFF_DFS)
NORM_FORCE.reset_index(inplace=True)
NORM_FORCE.set_index(['subject', 'VIB', 'JMP',
                      'DIR', 'trial', 'pert_time'], inplace=True)
# mean over trials
MEAN_FORCE = NORM_FORCE.groupby(level=['subject', 'VIB', 'JMP',
                                       'DIR', 'pert_time']).mean()
# sem over trials
SEMS_FORCE = NORM_FORCE.groupby(level=['subject', 'VIB', 'JMP',
                                       'DIR', 'pert_time']).sem()

PERT_ONSET = 0
PERT_OFFSET = PERT_ONSET + 250
XLIMS = (-2, 2)
YLIMS = (-140, 540)

sns.set(font_scale=1.2)
with sns.axes_style("ticks"):
    FIG, AXS = subplots(2, 1, figsize=(5, 8))
    sns.despine()

DIR2AXS = {'away': AXS[0], 'twrd': AXS[1]}
```

```
MEAN_CND = MEAN_FORCE.groupby(['VIB', 'JMP', 'DIR', 'pert_time']).mean()
SEM_CND = MEAN_FORCE.groupby(['VIB', 'JMP', 'DIR', 'pert_time']).sem()
GRP_CND = MEAN_CND.groupby(['VIB', 'JMP', 'DIR'])

# Plot combined perturbation conditions; mean (+/- sem) over subjects.
EXCLUDE_JMP = ['hidden', 'none']
for (vib, jmp, _dir), df in GRP_CND:

    if jmp in EXCLUDE_JMP:
        continue

    ax = DIR2AXS[_dir]
    lbl = f"vib {vib} & jmp {jmp}"
    time_axis = df.index.get_level_values('pert_time')
    mean_plot = df.values
    sem_plot = SEM_CND.xs((vib, jmp, _dir), level=('VIB', 'JMP', 'DIR'))
    sem_x1 = (mean_plot - sem_plot).squeeze()
    sem_x2 = (mean_plot + sem_plot).squeeze()
    ax.fill_betweenx(time_axis, sem_x1, sem_x2, color=COLOR[jmp], alpha=.2)
    ax.plot(mean_plot, time_axis, linestyle=STYLE[vib], color=COLOR[vib],
            alpha=.7, lw=2)
    # ax.legend(loc=0)  # legend must be done manually to make sense :/

for ax in AXS.ravel():
    ax.set_xticks((linspace(XLIMS[0], XLIMS[1], 5)))
    ax.set_ylim(YLIMS)
    ax.set_xlim(XLIMS)
    ax.plot((0, 0), YLIMS, '--', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_ONSET,)*2, ':', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_OFFSET,)*2, ':', color='lightgrey', zorder=0)
    ax.set_ylabel('time since cursor onset [ms]')

for ax in [AXS[1]]:
    sns.despine(ax=ax, top=False, bottom=True)
    ax.invert_yaxis()
    ax.xaxis.tick_top()
    ax.set_xticklabels(())

tight_layout()
```

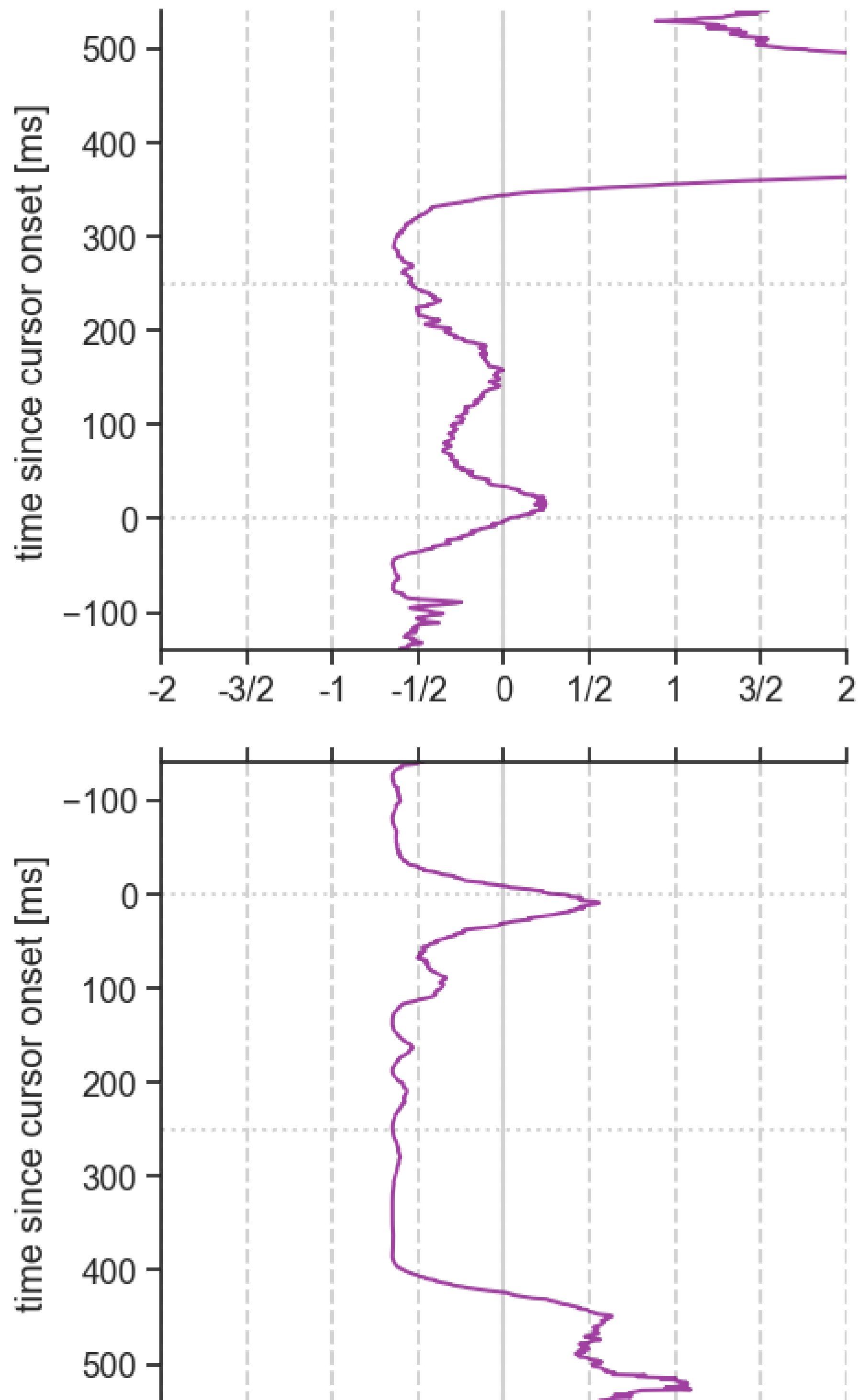



```
[39]: # Plot figure 4 b,d: Bayes factor of "JZS" t-test for the planned 2x3 contrast
      # between congruent versus incongruent perturbation directions, with the null
      # model in the denominator.
```

```
PERT_ONSET = 0
PERT_OFFSET = PERT_ONSET + 250
YLIMS = (-140, 540)

# Use the labeling thresholds for BF as suggested by Jeffreys (1961)
BF_LABELS = ['-2', '-1', '-3/2', '-1/2', '0', '1/2', '3/2', '1', '2']
BF_LEVELS = log10(array([10**eval(ll) for ll in BF_LABELS]))
XLIMS = (BF_LEVELS.min(), BF_LEVELS.max())

sns.set(font_scale=1.2)
with sns.axes_style("ticks"):
    FIG, AXS = subplots(2, 1, figsize=(5, 8), sharex=False)
    sns.despine()

DIR2AXS = {'away': AXS[0], 'twrd': AXS[1]}

#  add axis annotations
for ax in AXS.ravel():
    ax.set_ylim(YLIMS)
    ax.set_xlim(XLIMS)
    ax.plot(XLIMS, (PERT_ONSET,)*2, ':', color='lightgrey', zorder=0)
    ax.plot(XLIMS, (PERT_OFFSET,)*2, ':', color='lightgrey', zorder=0)
    ax.set_ylabel('time since cursor onset [ms]')
    ax.set_xticks(BF_LEVELS)
    ax.set_xticklabels(BF_LABELS)
    for bb in BF_LEVELS:
        ax.plot((bb,)*2, YLIMS, '--', color='lightgrey', zorder=0)
    ax.plot((log10(1),)*2, YLIMS, '-', color='lightgrey', zorder=0)

for ax in [AXS[1]]:
    sns.despine(ax=ax, top=False, bottom=True)
    ax.invert_yaxis()
    ax.xaxis.tick_top()
    ax.set_xticklabels(())

# Compare 2x3 interaction for each vibrated muscle (biceps or triceps) vs 0:
#    (shiftL & vib - no shift & vib) - (shiftL & no vib - no shift & no vib)
# - ((shiftR & vib - no shift & vib) - (shiftR & no vib - no shift & no vib)).
# NOTE: If the forces are already normalized relative to the no-shift condition,
#       this reduces to a 2x2 interaction, assumed below:
#       (shiftL & vib - shiftL & no vib) - (shiftR & vib - shiftR & no vib)
LEVELS = ('VIB', 'JMP', 'DIR')
MIN_CHUNK = 10
print("Interaction onsets found at:")
for (vib, _dir) in (('bic', 'away'), ('tri', 'twrd')):
    ax = DIR2AXS[_dir]
    contrast = \
```

```
        (MEAN_FORCE.xs((vib, 'left', _dir), level=LEVELS)
         - MEAN_FORCE.xs(('none', 'left', _dir), level=LEVELS)) \
      - (MEAN_FORCE.xs((vib, 'right', _dir), level=LEVELS)
         - MEAN_FORCE.xs(('none', 'right', _dir), level=LEVELS))
    c_grp = contrast.groupby('pert_time')
    tstBF = doTestOnGrps(c_grp, test='BF', paired=False)
    bftrns = log10(tstBF['bf'].astype('float'))
    ax.plot(bftrns, tstBF.idx, color='purple', alpha=.75)

    # print 1st crossing into substantial evidence for min_chunk samples
    # cut off at 'substantial' evidence, i.e. 10**(1/2)
    bfsigs = bftrns > log10(10**(1/2))
    bfmask = cut_chunks(bfsigs, min_chunk=MIN_CHUNK)
    on_idx = find(bfmask)[0] if any(bfmask) else nan
    onset = nan if isnan(on_idx) else tstBF.idx[on_idx]
    print(f"{_dir}, {vib} at {onset:g} ms")

tight_layout()
```

```
Interaction onsets found at:
away, bic at 351 ms
twrd, tri at 445 ms
```




```
[ ]: # Onset detection based on Bayesian ANOVA per subject.
     # NOTE run over all 22 subjects may take tens of minutes (~40 min)!
     seed(42)  # set seed for reproducibility (seems to also apply to BF package)
```

```python
grp = ALL_MOVEMENT_DATA.groupby(level=['subject', 'DIR', 'FLD'])['rotForceX']
onsets = pd.DataFrame()
START_TIME = time.time()
print("BANOVA-based onsets between all channel trials...")
CUT_A, CUT_B = 0, 550  # inclusive boundaries of trial times after perturbation
MIN_CHUNK = 10  # continuous chunk of
NUM_COLS = CUT_B - CUT_A + 1  # number of columns

for (sbj, _dir, fld), df in grp:
    if fld != 'chan':
        continue

    print((sbj, _dir))  # show some indication of progress...

    list_banova_dfs = []
    for tt in range(CUT_A, CUT_B):
        df4banova = df.xs(tt, level='pert_time').reset_index()

        if any(df4banova.isna()):  # don't calculate with incomplete datasets
            continue

        for col in ('VIB', 'JMP'):  # convert independent variables to factors
            df4banova[col] = df4banova[col].astype('category')

        # set up fixed-effects BANOVA; no factor is random, none is repeated
        bfres = BF.anovaBF(rformula('rotForceX ~ VIB + JMP + VIB:JMP'),
                           data=df4banova, progress=False)
        bfdf = bf2df(bfres, {'pert_time': tt})
        # set sensible indices and index names
        bfdf.reset_index(inplace=True)
        bfdf.set_index(['pert_time', 'index'], inplace=True)
        bfdf.index.names = ['pert_time', 'effect']
        list_banova_dfs.append(bfdf)

    banovas_df = pd.concat(list_banova_dfs)

    for eff in ('VIB', 'JMP', 'VIB:JMP'):
        if eff == 'VIB:JMP':
            # get BF favoring interaction relative to model with main effects
            bf_full = banovas_df.xs('VIB + JMP + VIB:JMP', level='effect')['bf']
            bf_main = banovas_df.xs('VIB + JMP', level='effect')['bf']
            bflogs = log10(bf_full / bf_main)
        else:
            # get BF favoring main effect over model with just intercept
            bflogs = log10(banovas_df.xs(eff, level='effect')['bf'])

        bfsigs = bflogs > 1/2  # cut off at 'substantial' evidence 10^{1/2}
```

```
        bfmask = cut_chunks(bfsigs, min_chunk=MIN_CHUNK)
        on_idx = find(bfmask)[0] if any(bfmask) else nan
        onset = nan if isnan(on_idx) else bfsigs.index[on_idx]
        content = {'subject': [sbj],
                   'DIR': _dir,
                   'condVS': eff,
                   'onset': onset}
        onsets = onsets.append(pd.DataFrame(data=content), ignore_index=True)

onsets.set_index(['subject', 'DIR', 'condVS'], inplace=True)
onsets = onsets['onset']
duration = time.time() - START_TIME
print(f"Done. Total time: {duration:.1f} s wall time.")
```

[21]:

```
## Print table of onsets computed in cell above, as mean and std over subjects.
GRP = onsets.groupby(level=['DIR', 'condVS'])
table_absolute = pd.DataFrame({'mean': GRP.mean(), 'std': GRP.std()})

print()
print(table_absolute)

# Relative onset comparisons between all 3 ANOVA effects
LEVELS = ('condVS', 'DIR')
COMPARISONS = ((('away', 'VIB'), ('away', 'JMP')),
               (('twrd', 'VIB'), ('twrd', 'JMP')),
               (('away', 'JMP'), ('away', 'VIB:JMP')),
               (('twrd', 'JMP'), ('twrd', 'VIB:JMP')))
print('\nPaired onset comparisons between (B)ANOVA effects:')
for ((dir1, eff1), (dir2, eff2)) in COMPARISONS:
    onsets_eff1 = onsets.xs((eff1, dir1), level=LEVELS)
    onsets_eff2 = onsets.xs((eff2, dir2), level=LEVELS)
    diff_eff2_1 = onsets_eff2 - onsets_eff1
    wlcx = wilcox_in_r(diff_eff2_1)
    print(f"{dir2}:{eff2} - {dir1}:{eff1}\t= ", end='')
    print(f"{diff_eff2_1.mean():.1f} ms "
          f"({diff_eff2_1.std():.1f} SD), "
          f"NA = {sum(diff_eff2_1.isna()):2d}", end='')
    print(f"; Wilcox V = {wlcx['V']:.0f}, p = {wlcx['pval']:.6f}")
```

```
                   mean         std
DIR  condVS
away JMP      191.045455   23.881607
     VIB       93.000000   10.605479
     VIB:JMP  291.571429  116.486853
twrd JMP      197.454545   47.174173
     VIB       74.818182   17.418207
```

```
     VIB:JMP  339.000000   53.756395

Paired onset comparisons between (B)ANOVA effects:
away:JMP - away:VIB     = 98.0 ms (22.1 SD), NA =  0; Wilcox V = 253, p =
0.000043
twrd:JMP - twrd:VIB     = 122.6 ms (45.8 SD), NA =  0; Wilcox V = 252, p =
0.000049
away:VIB:JMP - away:JMP = 103.8 ms (108.2 SD), NA =  8; Wilcox V = 96, p =
0.004028
twrd:VIB:JMP - twrd:JMP = 136.3 ms (59.9 SD), NA = 13; Wilcox V = 45, p =
0.003906
```