ni Documentation

Release 0.2

Jacob Huth

March 16, 2015

CONTENTS

CHAPTER

ONE

INSTALLATION

To install the toolbox, clone the git repository, or download the zip file from: https://github.com/jahuth/ni

Clone the repository with:

```
git clone https://github.com/jahuth/ni.git
```

In the repository the toolbox, default configuration, the documentation and example files are included. If you have the necessary packages installed, you can start using it right away.

If you lack any of the following packages, you might want to install them (with eg. *pip*)

```
ipython matplotlib scipy numpy pandas scikit-learn statsmodels
```

Contents:

1.1 data Package

Provides easy access to some data.

1.1.1 `data` Module

Using the ni.Data data structures

The *Data* class is supposed to be easily accessible to the *ni.* models. They contain an index that separates the time series into different **cells**, **trials** and **conditions**.

Conditions are mostly for the users, as they are ignored by the model classes. They should be used to separate data before fitting a model on them, such that only data from a certain subset of trials (ie. one or more experimental conditions) are used for the fit. If multiple conditions are contained in a dataset that is passed to a model, the model should treat them as additional trials.

Trials assume a common time frame ie. that bin 0 of each trial corresponds to the same time relative to a stimulus, such that rate fluctuations can be averaged over trials.

Cells signify spike trains that are recorded from different sources (or spike sorted), such that there can be correlations between cells in a certain trail.

The index is hierarchical, as in for each condition there are several trials, which each have several cells. But since modelling is mainly used to distinguish varying behaviour of the same ensemble of cells, the number of cells in a trial and the number of trials pro condition has to be equal.

Storing Spike Data in Python with Pandas

The pandas package allows for easy storage of large data objects in python. The structure that is used by this toolbox is the pandas `pandas.MultiIndexedFrame` which is a `pandas.DataFrame` / pandas.DataFrame with an Index that has multiple levels.

The index contains at least the levels `'Cell'`, `'Trial'` and `'Condition'`. Additional Indizex can be used (eg. `'Bootstrap Sample'` for Bootstrap Samples), but keep in mind that when fitting a model only `'Cell'` and `'Trial'` should remain, all other dimensions will be collapsed as more sets of Trials which may be indistinguishable after the fit.

Condition	Cell	Trial	*t* (Timeseries of specific trial)
0	0	0	0,0,0,0,1,0,0,0,0,1,0...
0	0	1	0,0,0,1,0,0,0,0,1,0,0...
0	0	2	0,0,1,0,1,0,0,1,0,1,0...
0	1	0	0,0,0,1,0,0,0,0,0,0,0...
0	1	1	0,0,0,0,0,1,0,0,0,1,0...
...	...	...	...
1	0	0	0,0,1,0,0,0,0,0,0,0,1...
1	0	1	0,0,0,0,0,1,0,1,0,0,0...
...	...	...	...

To put your own data into a `pandas.DataFrame`, so it can be used by the models in this toolbox create a MultiIndex for example like this:

```
import ni
import pandas as pd
d = []
tuples = []
for con in range(nr_conditions):
        for t in range(nr_trials):
                for c in range(nr_cells):
                                spikes = list(ni.model.pointprocess.getBinary(Spike_times_STC.all_SU
                                if spikes != []:
                                        d.append(spikes)
                                        tuples.append((con,t,c))
index = pd.MultiIndex.from_tuples(tuples, names=['Condition','Trial','Cell'])
data = ni.data.data.Data(pd.DataFrame(d, index = index))
```

If you only have one trial if several cells or one cell with a few trials, it can be indexed like this:

> from ni.data.data import Data import pandas as pd
>
> index = pd.MultiIndex.from_tuples([(0,0,i) for i in range(len(d))], names=['Condition','Cell','Trial']) data = Data(pd.DataFrame(d, index = index))

To use the data you can use `ni.data.data.Data.filter()`:

```
only_first_trials = data.filter(0, level='Trial')

# filter returns a copy of the Data object

only_the_first_trial = data.filter(0, level='Trial').filter(0, level='Cell').filter(0, level='Condit

only_the_first_trial = data.condition(0).cell(0).trial(0) # condition(), cell() and trial() are shor

only_some_trials  = data.trial(range(3,15))
# using slices, ranges or boolean indexing causes the DataFrame to be indexed again from 0 to N, in
```

Also ix and xs pandas operations can be useful:

```
plot(data.data.ix[(0,0,0):(0,3,-1)].transpose().cumsum())
plot(data.data.xs(0,level='Condition').xs(0,level='Cell').ix[:5].transpose().cumsum())
```

class `ni.data.data.`**`Data`**(*matrix*, *dimensions=[ ]*, *key_index='i'*, *resolution=1000*)

Spike data container

Contains a panda Data Frame with MultiIndex. Can save to and load from files.

The Index contains at least Trial, Cell and Condition and can be extended.

`as_list_of_series`(*list_conditions=True*, *list_cells=True*, *list_trials=False*, *list_additional_indizes=True*)

Returns one timeseries, collapsing only certain indizes (on default only trials). All non collapsed indizes

`as_series`()

Returns one timeseries, collapsing all indizes.

The output has dimensions of (N,1) with N being length of one trial x nr_trials x nr_cells x nr_conditions (x additonal indices).

If cells, conditions or trials should be separated, use `as_list_of_series()` instead.

`cell`(*cells=False*)

filters for an array of cells -> see `ni.data.data.Data.filter()`

`condition`(*conditions=False*)

filters for an array of conditions -> see `ni.data.data.Data.filter()`

`filter`(*array=False*, *level='Cell'*)

filters for arbitrary index levels *array* a number, list or numpy array of indizes that are to be filtered *level* the level of index that is to be filtered. Default: 'Cell'

`firing_rate`(*smooth_width=0*, *trials=False*)

computes the firing rate of the data for each cell separately.

`getFlattend`(*all_in_one=True*, *trials=False*)

Deprecated since version 0.1: Use `as_list_of_series()` and `as_series()` instead

Returns one timeseries for all trials.

The *all_in_one* flag determines whether `'Cell'` and `'Condition'` should also be collapsed. If set to *False* and the number of Conditions and/or Cells is greater than 1, a list of timeseries will be returned. If both are greater than 1, then a list containing for each condition a list with a time series for each cell.

`html_view`()

`interspike_intervals`(*smooth_width=0*, *trials=False*)

computes inter spike intervalls in the data for each cell separately.

`read_pickle`(*path*)

Loads a DataFrame from a file

`reduce_resolution`(*factor=2*)

`shape`(*level*)

Returns the shape of the sepcified level:

```
>>> data.shape('Trial')
        100

>>> data.shape('Cell') == data.nr_cells
        True
```

time (*begin=None*, *end=None*)
gives a copy of the data that contains only a part of the timeseries for all trials,cells and conditions.

This resets the indices for the timeseries to 0...(end-begin)

to_pickle (*path*)
Saves the DataFrame to a file

trial (*trials=False*)
filters for an array of trials -> see `ni.data.data.Data.filter()`

`ni.data.data.`**loadFromFile** (*path*)
loads a pandas DataFrame from a file

`ni.data.data.`**matrix_to_dataframe** (*matrix*, *dimensions*)
conerts a trial x cells matrix into a DataFrame

`ni.data.data.`**merge** (*datas*, *dim*, *keys=False*)
merges multiple Data instances into one:

```
data = ni.data.data.merge([ni.data.data.Date(f) for f in ['data1.pkl','data2.pkl','data3.pkl']],
```

`ni.data.data.`**saveToFile** (*path*, *o*)
saves a DataFrame-like to a file

1.1.2 `decoding_data` Module

Loads Data into a Panda Data Frame

class `ni.data.decoding_data.`**Cell** (*data*)

class `ni.data.decoding_data.`**DecodingData**
Loads Data into a Panda Data Frame

class `ni.data.decoding_data.`**Trial**

addCell (*data*)

getMatrix ()

`ni.data.decoding_data.`**get** ()

1.1.3 `monkey` Module

`ni.data.monkey.`**Data** (*file_nr='101a03'*, *resolution=1000*, *trial=*[ ], *condition=*[ ], *cell=*[ ])
Loads Data into a Data Frame

Expects a file number. Available file numbers are in ni.data.monkey.available_files:

```
>>> print ni.data.monkey.available_files
     ['101a03', '104a10', '107a03', '108a08', '112a03', '101a03', '104a11', '107a04', '109a04
```

trial

number of trial to load or list of trials to load. Non-existent trial numbers are ignored.

condition

number of condition to load or list of conditions to load. Non-existent condition numbers are ignored.

cell

number of cell to load or list of cells to load. Non-existent cell numbers are ignored.

Example:

```
data = ni.data.monkey.Data(trial_nr = ni.data.monkey.available_trials[3], trial=range(10), condi
```

1.2 model Package

This package provides two kinds of models:

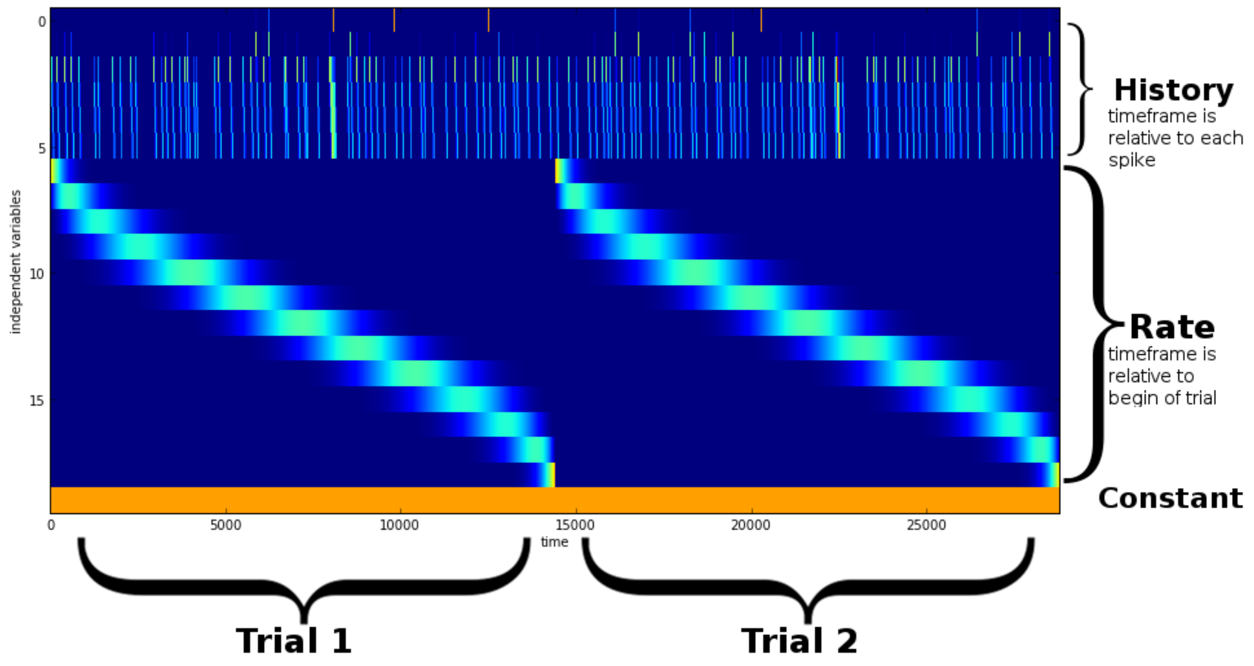
- Generative pointprocess models (see Section `ni.model.pointproccess`)
- GLM based inhomogeneous pointprocess (ip) models

The pointprocess models can instantiate a given firing rate function with a spike train. The ip models use “components” that are combined to fit a spike train using a GLM.

1.2.1 What is a “component”?

For the *ni.model* package, a component is a set of time series (or a way to generate these) in a Generalized Linear Model that is used to fit a set of spike trains. When a model is fitted to a set of spike trains, the components are used to compute the designmatrix. Each component provides 1 or more timeseries that represent the influence of a specific event or aspect that is expected to affect the firing rate at that specific point in time to a certain degree.

Eg. a component can be as simple as a constant value over time, or it can be 0 for most of the time, except at those time points where a specific event occurs (eg. a spike of a different neuron). The design matrix shown here is mostly 0 and contains three components (history, rate and a constant):



To model a more or less precise time course of the effect of an event, the component needs to create mutiple time series, each representing a time window relative to the event (usually following the event, if a causal link is assumed). It is convenient to use splines to model these windows, because they will produce a smooth time series when summed. These time windows can be arranged linearly (each spanning the same amount of time) or logarithmically (such that there is a higher resolution close to the event than further away). The length and number of the time windows can be adjusted, since their explanatory value depends on the modeled process.

1.2.2 The Rate component

Given that each spike train is alligned in the same timeframe regarding a certain stimulus (or sequence of stimuli), a rate component models rate fluctuations that occur in every trial. This is done by creating a number of timeseries, each representing a portion of the trial time: eg. early in the trial, in the middle of the trial, at the end of the trial.

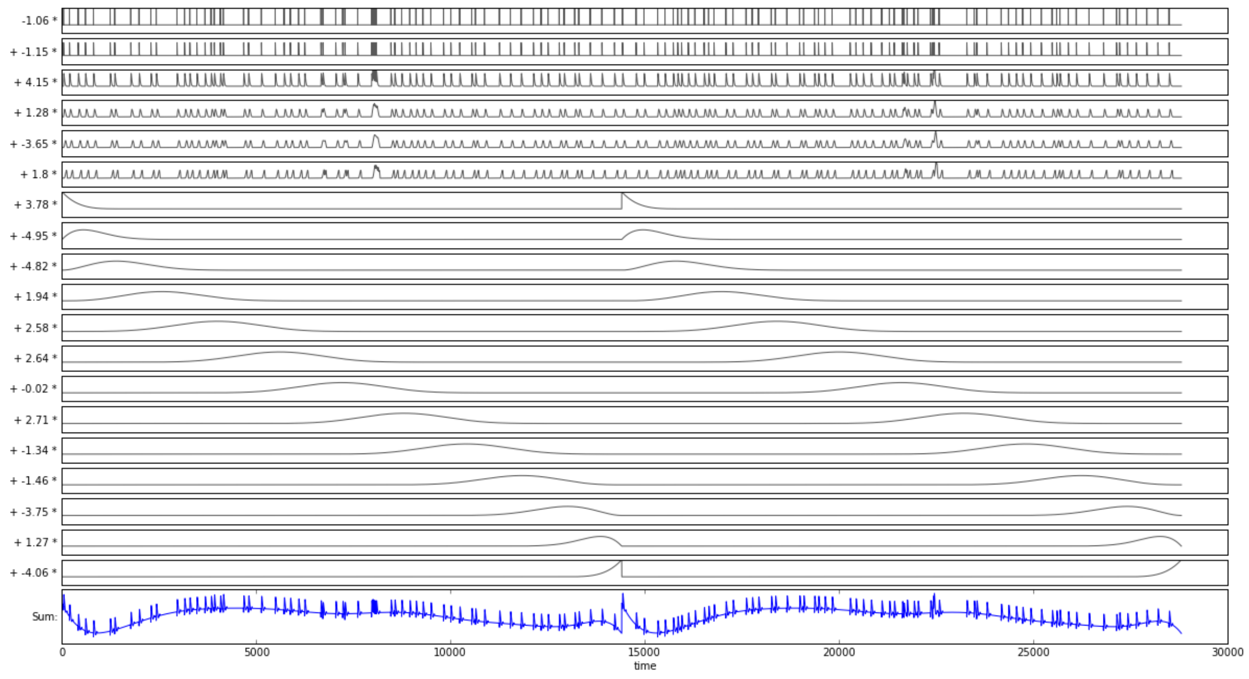
A `ni.model.designmatrix.RateComponent` will span the whole trial with a given number of knots while the class `ni.model.designmatrix.AdaptiveRateComponent` will use the firing rate to space out the knots to cover (on average) an equal amount of spikes.

1.2.3 The history components

Since the spike data containers contain multiple spike trains, a component can access the spiketimes of each of the spike trains contained. The history components use these spike times to model the effect of a certain spike train on the dependent spike train.

- **autohistory:** the effect of the spike train on itself. Eg. refractory period or periodic spiking
- **crosshistory:** models the effect of another spike train on the one modeled
- 2nd order **autohistory_2d:** for bursting neurons, the autohistory alone will not capture the behaviour of the neuron, as it has to either predict periodic activity for the whole trial or no auto effect at all. The 2nd order history takes into account two spikes, such that the end of a burst can be predicted.

Here you can see the slow time frame of a rate fluctuation (fixed to the time frame of each trial) and the fast time frame of a history component (each relative to a spike):



1.2.4 Configuring a Model

The model class has a default behaviour that assumes you want to have one rate component, one autohistory componenet and a crosshistory component for each channel. This behaviour can be changed by providing a different configuration dictionary to the model.

Eg. to create a model without any of the components:

```
model = ni.model.ip.Model({'name':'Model','autohistory':False, 'crosshistory':False, 'rate':False})
```

Or, if additionally the 2nd order autohistory should be enabled:

```
model = ni.model.ip.Model({'name':'Model','autohistory_2d':True })
```

Important for models is also which spike train channel should be modeled by the others:

```
model_0 = ni.mode.ip.Model({'cell':0})
model_1 = ni.mode.ip.Model({'cell':1})
model_2 = ni.mode.ip.Model({'cell':2})
model_3 = ni.mode.ip.Model({'cell':3})
```

Instead of only true or false, crosshistory can also be a list of channels that should be used:

```
model = ni.mode.ip.Model({'cell':1,'crosshistory':[2]}) # cell 1 modeled using activity of cell 2
```

Also there are a number of configuration options that are passed on to a component to change its behaviour:

- **Rate Component:**
  - *knots_rate*: how many knots should be created
  - ***adaptive_rate*: Whether the RateComponent or AdaptiveRateComponent should be used. The AdaptiveRateCon**
    - adaptive_rate_exponent (default = 2)
    - adaptive_rate_smooth_width (default = 20)
- **History Components (autohistory and crosshistory):**
  - *history_length*: total lenght of the time spanned by the history component
  - *history_knots*: number of knots to span

These options are set for the whole model at once:

```
model = ni.mode.ip.Model({'knots_rate':10,'history_length':200}) # sets history_length for autohisto
```

1.2.5 Fitting a Model

An ip.Model can be fitted on a `ni.data.data.Data` object and will produce a `ni.model.ip.FittedModel`. This will contain the fitted coefficients and the design templates, such that the different components can be inspected individually:

```
model = ni.model.ip.Model({'cell':2,'crosshistory':[1,3]})
fm = model.fit(data.trial(range(data.nr_trials/2)))

fm.plot_prototypes() # will plot each fitted component

plot(fm.firing_rate_model().sum(1)) # will plot the firing rate components (rate + constant)

plot(fm.statistics['pvalues']) # plots the pvalues for each coefficient
```

1.2.6 Creating custom components

To be precise, what happens when a model is fitted is the following:

- **The model creates a `ni.model.designmatrix.DesignMatrixTemplate` and adds components (which are subclass**
    - when the Component classes are created they are provided with most of the information they need (eg. trial length)
- **The DesignMatrixTemplate is combined with the data which creates the actual designmatrix by:**
    - For each component the function getSplines is called, providing the component with spikes of other neurons
    - The component returns a 2 dimensional numpy array that has the dimensions of *length of time* **x** *number of time series*
    - if the array does not have the length of the complete time series, it will be repeated until it fits. So a kernel that has the length of exactly one trial will be repeated for each trial (the ni models by default make all trials the same length).
- The design matrix is then passed to the GLM fitting backend

So, if you want to add a component, this can be either implement a function *getSplines* that returns a 2d numpy array that has the correct dimensions (time bins x N), or you can use or inherit from `ni.model.designmatrix.Component` and set the *self.kernel* attribute which will then be returned:

```
my_kernel = ni.model.create_splines.create_splines_linspace(time_length, 6, 0) # creates a 6 knotted
c = Component(header='My own component',kernel=my_kernel)
model = ni.model.ip.Model(custom_components = [c])
```

Applications can be eg. a rate component that is only applied to trials, while a second component is applied to the others. If two kinds of trials are alternated this could be done like this:

```
from ni.model.designmatrix import Component
my_kernel = ni.model.create_splines.create_splines_linspace(time_length, 6, 0) # creates a 6 knotted
is_trial_type_1 = repeat([0,1]*(number_of_trials/2),trial_length)
is_trial_type_2 = 1 - is_trial_type_1
c1 = Component(header='Trial Type 1 Rate',kernel=np.concatenate([my_kernel]*number_of_trials) * is_tr
c2 = Component(header='Trial Type 2 Rate',kernel=np.concatenate([my_kernel]*number_of_trials) * is_tr
model = ni.model.ip.Model({'custom_components': [c1,c2]},trial_length)
```

Or alternatively you can overwrite a kernel of the rate component:

```
kernel = np.concatenate([
            np.concatenate([my_kernel]*number_of_trials) * is_trial_type_1[:,np.newaxis]),
            np.concatenate([my_kernel]*number_of_trials) * is_trial_type_2[:,np.newaxis])
        ])
model = ni.model.RateModel({'rate':True, 'custom_kernels': [{'Name':'rate','Kernel': kernel}]})
```

1.2.7 Classes in the ni.model package

class `ni.model.`**`BareModel`**(*configuration={}*)

Bases: `ni.model.ip.Model`

This is a shorthand class for an Inhomogenous Pointprocess model that contains no Components.

This is completely equivalent to using:

ni.model.ip.Model({'name':'Bare Model','autohistory':False, 'crosshistory':False, 'rate':False})

class `ni.model.`**`RateModel`**`(`*knots_rate=10*`)`

Bases: `ni.model.ip.Model`

This is a shorthand class for an Inhomogenous Pointprocess model that contains only a RateComponent and nothing else.

This is completely equivalent to using:

> ni.model.ip.Model({'name':'Rate Model','autohistory':False, 'crosshistory':False, 'knots_rate':knots_rate})

class `ni.model.`**`RateAndHistoryModel`**`(`*knots_rate=10*, *history_length=100*, *history_knots=4*`)`

Bases: `ni.model.ip.Model`

This is a shorthand class for an Inhomogenous Pointprocess model that contains only a RateComponent, a Autohistory Component and nothing else.

This is completely equivalent to using:

> ni.model.ip.Model({'name':'Rate Model with Autohistory Component','autohistory':True, 'crosshistory':False, 'knots_rate':knots_rate, 'history_length':history_length, 'knot_number':history_knots})

`ip` Module

class `ni.model.ip.`**`Configuration`**`(`*c=False*`)`

Bases: `ni.tools.pickler.Picklable`

The following values are the defaults used:

name = "Inhomogeneous Point Process"

cell = 0

Which cell to model in the data (index)

Boolean Flags for Default Model components (True: included, Fals: not included):

```
autohistory = True
crosshistory = True
rate = True
constant = True
autohistory_2d = False
```

backend = "glm"

The backend used. Valid options: "glm" and "elasticnet"

backend_config = False

knots_rate = 10

Number of knots in the rate component

adaptive_rate = False

Whether to use linear spaced knots or an adaptive knot spacing, which depends on the variability of the firing rate. Uses these two options:

```
adaptive_rate_exponent = 2
adaptive_rate_smooth_width = 20
```

history_length = 100

Length of the history kernel

knot_number = 3

Number of knots in the history kernel

custom_kernels = []

List of kernels that overwrite default component behaviour. To overwrite the *rate* component use:

```
[{ 'Name':'rate', 'Kernel': np.ones(100) }]
```

custom_components = []

List of components that get added to the default ones.

self.be_memory_efficient = True

Whether or not to save the data and designmatrix created by the fitting model

Unimportant options no one will want to change:

```
delete_last_spline = True
design = 'new'
mask = np.array([True])
```

Look at the [source] for a full list of defaults.

class `ni.model.ip.`**`FittedModel`**(*model*)

Bases: `ni.tools.pickler.Picklable`

When initialized via Model.fit() it contains a copy of the configuration, a link to the model it was fitted from and fitting parameters:

FittedModel. **fit**

modelFit Output

FittedModel. **design**

The DesignMatrix used. Use *design.matrix* for the actual matrix or design.get('...') to extract only the rows that correspond to a keyword.

`compare`(*data*)

Using the model this will predict a firing probability function according to a design matrix.

Returns:

Deviance_all: dv, **LogLikelihood_all**: ll, **Deviance**: dv/nr_trials, **LogLikelihood**: ll/nr_trials, **llf**: Likelihood function over time **ll**: np.sum(ll)/nr_trials

`complexity`

returns the length of the parameter vector

`dumps`()

see `ni.tools.pickler`

`family_fitted_function`(*p*)

only implemented family: Binomial

`firing_rate_model`()

returns a time series which contains the rate and constant component

generate(*bins=-1*)
Generates new spike trains from the extracted staistics

This function only uses rate model and autohistory. For crosshistory dependence, use `ip_generator`.

bins

How many bins should be generated (should be multiples of trial_length)

getParams()
returns the parameters of each design matrix component as a list

getPvalues()
returns pvalues of each component as a dictionary

html_view()
see `ni.tools.html_view`

plotParams(*x=-1*)
plots the parameters and returns a dictionary of figures

plot_firing_rate_model()
returns a time series which contains the rate and constant component

plot_prototypes()
plots each of the components as a prototype (sum of fitted b-splines) and returns a dictionary of figures

predict(*data*)
Using the model this will predict a firing probability function according to a design matrix.

prototypes()
returns a dictionary with a prototype (numpy.ndarray) per component

pvalues_by_component()
returns pvalues of each component as a dictionary

class `ni.model.ip.`**Model**(*configuration=None*, *nr_bins=0*)
Bases: `ni.tools.pickler.Picklable`

backend

cell

compare(*data*, *p*, *nr_trials=1*)
will compare a timeseries of probabilities *p* to a binary timeseries or Data instance *data*.

Returns:

Deviance_all: dv, **LogLikelihood_all**: ll, **Deviance**: dv/nr_trials, **LogLikelihood**: ll/nr_trials, **llf**: Likelihood function over time **ll**: np.sum(ll)/nr_trials

dm(*in_spikes*, *design=None*)
Creates a design matrix from data and self.design

in_spikes *ni.data.data.Data* instance

design (optional) a different `designmatrix.DesignMatrixTemplate`

fit(*data=None*, *beta=None*, *x=None*, *dm=None*, *nr_trials=None*)
Fits the model

in_spikes *ni.data.data.Data* instance

example:

```
from scipy.ndimage import gaussian_filter
import ni
model = ni.model.ip.Model(ni.model.ip.Configuration({'crosshistory':False}))
data = ni.data.monkey.Data()
data = data.condition(0).trial(range(int(data.nr_trials/2)))
dm = model.dm(data)
x = model.x(data)
from sklearn import linear_model
betas = []
fm = model.fit(data)
betas.append(fm.beta)
print "fitted."
for clf in [linear_model.LinearRegression(), linear_model.RidgeCV(alphas=[0.1, 1.0, 10.0])]:
        clf.fit(dm,x)
        betas.append(clf.coef_)

        figure()
        plot(clf.coef_.transpose(),'.')
        title('coefficients')
        prediction = np.dot(dm,clf.coef_.transpose())
        figure()
        plot(prediction)
        title('prediction')
        ll = x * log(prediction) + (len(x)-x)*log(1-prediction)
        figure()
        plot(ll)
        title('ll')
        print np.sum(ll)
```

fit_with_design_matrix (*fittedmodel*, *spike_train_all_trial*, *dm*)
internal function

generateDesignMatrix (*data*, *trial_length*)
generates a design matrix template. Uses meta data from *data* to determine number of trials and trial length.

html_view ()
see `ni.tools.html_view`

predict (*beta*, *data*)
will predict a firing probability function according to a design matrix.

x (*in_spikes*)
converts data into a dependent variable time series, ie. it chooses the cell that was configured and extracts only this timeseries.

`designmatrix` Module

class `ni.model.designmatrix.`**AdaptiveRateComponent** (*header='rate'*, *rate=False*, *exponent=2*, *knots=10*, *length=1000*, *kernel=False*)

Bases: `ni.model.designmatrix.Component`

Rate Design Matrix Component

header: name of the kernel component rate: a rate function that determines exponent: the rate function will be taken to thins power to have a higher selctivity for high firing rates knots: Number of knots length: length of the component. Will be multiplied kernel: use this kernel instead of a newly created one

getSplines (*data=[ ]*)

class ni.model.designmatrix.**Component** (*header='Undefined', kernel=0*)

Bases: ni.tools.pickler.Picklable

Design Matrix Component

header: name of the kernel component kernel: kernel that will be tiled to fill the design matrix

getSplines (*data=[ ]*)

class ni.model.designmatrix.**DesignMatrix** (*length, width=1*)

Bases: ni.tools.pickler.Picklable

Use DesignMatrixTemplate to create a design matrix.

This class computes an actual matrix, where DesignMatrixTemplate can be saved before the matrix is instanciated.

add (*splines, header*)

addLinSpline (*knots, header, length=0*)

addLogSpline (*knots, header, length=0*)

clip ()

get (*filt*)

getIndex (*filt*)

getMask (*filt*)

plot (*filt='‘*)

setMask (*mask*)

class ni.model.designmatrix.**DesignMatrixTemplate** (*length, trial_length=0*)

Bases: ni.tools.pickler.Picklable

Most important class for Design Matrices

Uses components that are then combined into an actual design matrix:

```
>>> DesignMatrixTemplate(data.nr_trials * data.time_bins)
>>> kernel = cs.create_splines_logspace(self.configuration.history_length, self.configuration.kn
>>> design_template.add(designmatrix.HistoryComponent('autohistory', kernel=kernel))
>>> design_template.add(designmatrix.HistoryComponent('crosshistory'+str(2), channel=2, kernel =
>>> design_template.add(designmatrix.RateComponent('rate',self.configuration.knots_rate,trial_le
>>> design_template.add(designmatrix.Component('constant',np.ones((1,1))))
>>> design_template.combine(data)
```

add (*component*)

Adds a component

combine (*data*)

combines the design matrix template into an actual design matrix.

It needs an ni.Data instance for this to place the history splines.

get (*filt*)

returns the splines of the first component, the header of which matches *filt*

getIndex (*filt*)

returns the index (design matrix rows) of the component matching *filt*

getMask (*filt*)
reurns the mask of the component matching *filt*

get_components (*filt*)
returns all component, the header of which matches *filt*

setMask (*mask*)
sets a mask (list of boolean values), which design matrix rows to use. Default is all True. If *mask* is shorter' than the desgin matrix, all following values are assumed True.

class `ni.model.designmatrix.`**HistoryComponent** (*header='autohistory'*, *channel=0*, *history_length=100*, *knot_number=4*, *order_flag=1*, *kernel=False*, *delete_last_spline=True*)

Bases: `ni.model.designmatrix.Component`

History Design Matrix Component

Will be convolved with spikes before fitting

header: name of the kernel component channel: which channel the kernel should be convolved with (default 0) history_length: length of the kernel knot_number: number of knots (will be logspaced) order_flag: default 0 (no higher order interactions)

kernel: use this kernel instead of a newly created one

Atm only order 1 interactions

getSplines (*channels=*[ ])

class `ni.model.designmatrix.`**HistoryDesignMatrix** (*spikes*, *history_length=100*, *knot_number=5*, *order_flag=1*, *kernel=False*)

Internal helper class

class `ni.model.designmatrix.`**RateComponent** (*header='rate'*, *knots=10*, *length=1000*, *kernel=False*)

Bases: `ni.model.designmatrix.Component`

Rate Design Matrix Component

header: name of the kernel component knots: Number of knots length: length of the component. Will be multiplied

kernel: use this kernel instead of a newly created one

getSplines (*data=*[ ])

class `ni.model.designmatrix.`**SecondOrderHistoryComponent** (*header='autohistory'*, *channel_1=0*, *channel_2=0*, *history_length=100*, *knot_number=4*, *order_flag=1*, *kernel_1=False*, *kernel_2=False*, *delete_last_spline=True*)

Bases: `ni.model.designmatrix.Component`

History Design Matrix Component with Second Order Kernels

Will be convolved with spikes before fitting

header: name of the kernel component channel: which channel the kernel should be convolved with (default 0) history_length: length of the kernel knot_number: number of knots (will be logspaced) order_flag: default 0 (no higher order interactions) kernel: use this kernel instead of a newly created one

Atm only order 1 interactions

getSplines (*channels=[ ]*, *get_1d_splines=False*, *beta=False*)

ni.model.designmatrix.**convolve_spikes** (*spikes*, *kernel*)
Convolves a spike train with a kernel by adding the kernel onto every spiketime.

ni.model.designmatrix.**convolve_spikes_2d** (*spikes_a*, *spikes_b*, *kernel_a*, *kernel_b*)
Does a 2d convolution

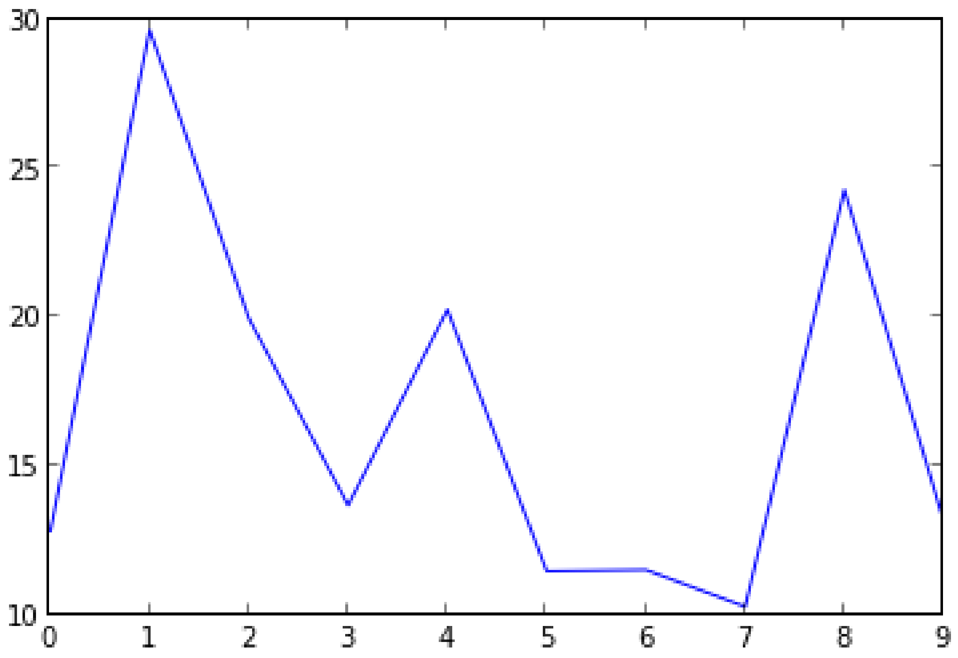
net_sim Module

The Net Simulator is divided into a Configuration, Net and a Result object.

After configuration of the network it can be instantiated by calling *Net(conf)* with a valid configuration *conf*. This creates eg. random connectivity so that the simulation with the same network can be repeated multiple times.

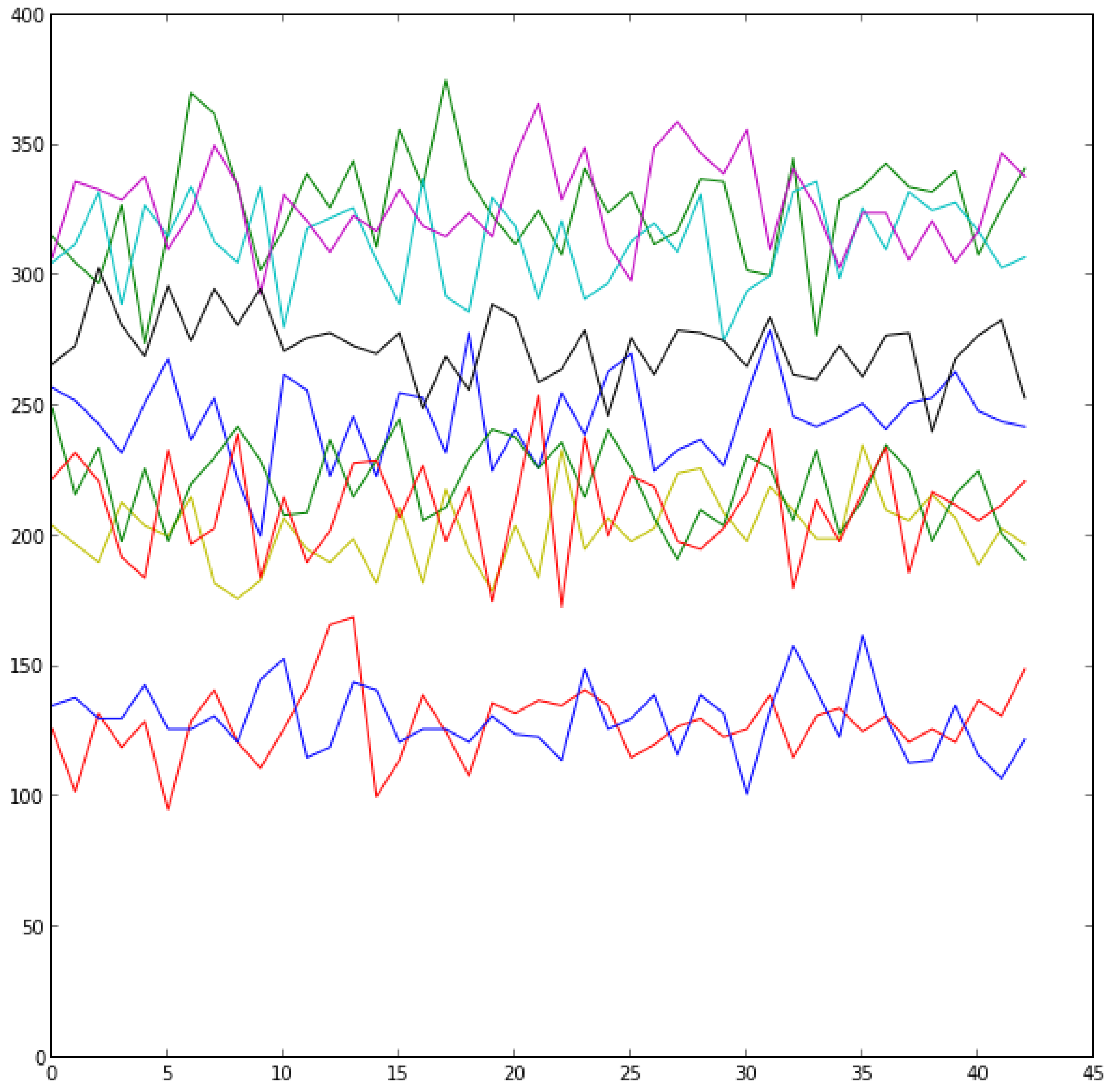
```
c = ni.model.net_sim.SimulationConfiguration()
c.Nneur = 10
net = ni.model.net_sim.Net(c)
print net
net.plot_firing_rates()

'ni.model.net_sim' Simulation Setup
Timerange: (250, 10250)
 10 channels with firing rates:
        [12.815928361, 29.6328550796, 19.9415819867, 13.6710936491, 20.242131795, 11.4661487294, 11.5
Firing Rates plot
```



```
for i in range(1,11):
    print i
    res1 = net.simulate()
    res1.plot_firing_rates()
```

```
plot(numpy.array([r.num_spikes_per_channel for r in net.results]))
plot([0]*len(net.results))
```



class `ni.model.net_sim.`**`Net`**(*config*)
The Net Simulator class. Use with an Configuration instance.

`load`(*filename*)
loads itself from a file

`plot_firing_rates`()
plots the intended firing rates

`plot_interaction`()
plots the interactions of the network

`save`(*filename*)
saves itself to a file

simulate()
Simulates the network

class ni.model.net_sim.**SimulationConfiguration**
Configures the simulation. The default values are:

> **Nneur** = 100 **sparse_coeff** = 0.1 **Trial_Time** = 1000 **prior_epoch** = 250 **Ntrials** = 10 **Nsec** = Ntrials*Trial_Time/1000 **Ntime** = Nsec*1000 **eps** =0.1 **frate_mu** = 1.0/25.0 **Nhist** = 50 **output** = False **rate_function** = False

Nsec

Ntime

class ni.model.net_sim.**SimulationResult**
Holds the results of a simulation

data

plot()
plots the resulting spike train

plot_firing_rates()
plots the resulting firing rate

plot_firing_rates_per_channel()
plots the firing rate for each channel

stopTimer()
stops the internal timer

store(*data*)
stores data in the container

ni.model.net_sim.**simulate**(*config*)
creates a network and simulates it.

create_design_matrix_vk Module

ni.model.create_design_matrix_vk.**computeCovariate**(*index*, *o*, *C*, *V1*)
Computes a row of the designMatrix corresponding to a certain covariate.

ni.model.create_design_matrix_vk.**create_design_matrix_vk**(*V1*, *o*)
Fills free rows in the current design matrix, deduced from size(mD) and len(freeCov), corresponding to a single covariate according to the spline bases of Volterra kernels. The current kernel(s) and the respective numbers of covariates that will be computed for each kernel is deduced from masterIndex by determining the position in hypothetical upper triangular part of hypercube with number of dimensions corresponding to current kernel order. Using only the 'upper triangular part' of the hypercube reflects the symmetry of the kernels which stems from the fact that only a single spline is used as basis function.

saves covariate information in cell array 'covariates', format is {kernelOrder relativePositionInKernel productTermsOfV1}

Anpassung für Gordon: masterIndex, log, C, mD, freeCov werden berechnet statt übergeben.

ni.model.create_design_matrix_vk.**detKernels**(*freeCov*, *masterIndex*, *oCov*, *mOrder*, *C*)
Determines from the number of free slots in Designmatrix len(freeCov) and the current masterIndex how many covariates for which Volterra coefficient can be computed. Updates model order mOrder.

ni.model.create_design_matrix_vk.**detModelOrder**(*masterIndex*, *C*)
Determines model order and corresponding number of covariates.

`ni.model.create_design_matrix_vk.`**`numCov`**`(`*C*, *complexity*`)`
Computes number of covariates in a model for which len(complexity) symmetric kernels are assumed.

`ni.model.create_design_matrix_vk.`**`upTriHalf`**`(`*C*, *cDim*`)`
Computes number of elements in upper triangular half of hybercube.

`create_splines` Module

`ni.model.create_splines.`**`N`**`(`*u*, *i*, *p*, *knots*`)`
Compute Spline Basis

Evaluates the spline basis of order p defined by knots at knot i and point u.

`ni.model.create_splines.`**`augknt`**`(`*knots*, *order*`)`
Augment knot sequence such that some boundary conditions are met.

`ni.model.create_splines.`**`create_splines`**`(`*length*, *nr_knots*, *remove_last_spline*, *fn_knots*`)`
Generates B-spline basis functions based on the length and number of knots of the ongoing iteration. fn_knots is a function that computes the knots.

`ni.model.create_splines.`**`create_splines_linspace`**`(`*length*, *nr_knots*, *remove_last_spline*`)`
Generates B-spline basis functions based on the length and number of knots of the ongoing iteration

`ni.model.create_splines.`**`create_splines_logspace`**`(`*length*, *nr_knots*, *remove_last_spline*`)`
Generates B-spline basis functions based on the length and number of knots of the ongoing iteration

`ni.model.create_splines.`**`spcol`**`(`*x*, *knots*, *spline_order*`)`
Computes the spline colocation matrix for knots in x.

The spline collocation matrix contains all m-p-1 bases defined by knots. Specifically it contains the ith basis in the ith column.

Input: x: vector to evaluate the bases on knots: vector of knots spline_order: order of the spline

Output:

colmat: m x m-p matrix The colocation matrix has size m x m-p where m denotes the number of points the basis is evaluated on and p is the spline order. The colums contain the ith basis of knots evaluated on x.

`ni.model.create_splines.`**`spline`**`(`*x*, *knots*, *p*, *i=0.0*`)`
Evaluates the ith spline basis given by knots on points in x

`backend_elasticnet` Module

This module provides a backend to the .ip model. It wraps the sklearn.linear_model.ElasticNet / ElasticNetCV objects.

class `ni.model.backend_elasticnet.`**`Configuration`**
Default Values:

crossvalidation = True

If true, alpha and l1_ratio will be calculated by crossvalidation.

alpha = 0.5

l1_ratio = 1

be_memory_efficient = True Does not keep the data with which it is fitted.

class `ni.model.backend_elasticnet.`**`Fit`**(*f*, *m*)

> **`predict`**(*X=False*)

class `ni.model.backend_elasticnet.`**`Model`**(*c=False*)

> **`fit`**(*x*, *dm*)

`ni.model.backend_elasticnet.`**`compare`**(*x*, *p*, *nr_trials=1*)

`ni.model.backend_elasticnet.`**`predict`**(*x*, *dm*)

`backend_glm` Module

This module provides a backend to the .ip model. It wraps the statsmodels.api.GLM object.

class `ni.model.backend_glm.`**`Configuration`**

Default Values:

> **be_memory_efficient = True** Does not keep the data with which it is fitted.

class `ni.model.backend_glm.`**`Fit`**(*f*, *m*)

> **`predict`**(*X=False*)

class `ni.model.backend_glm.`**`Model`**(*c=False*)

> **`fit`**(*y*, *X*)

`ni.model.backend_glm.`**`compare`**(*x*, *p*, *nr_trials=1*)

`ni.model.backend_glm.`**`predict`**(*x*, *dm*)

1.3 The `ni.model.pointprocess` Module

class `ni.model.pointprocess.`**`PointProcess`**(*dimensionality*)

A Point Process container.

Usually generated by loading from a file or via `ni.model.pointprocess.createPoisson()`

`addSpike`(*t*)

adds a spike to the point process, if it falls in the allowed range.

`getCounts`()

Gives a (in most cases binary) time series of the point process.

`getProbability`(*t_from*, *t_to*)

Undocumented

`plot`(*y=0*, *marker='|'*)

Plots the pointprocess as points at line *y*.

marker determines the color and shape of the marker. Default is a vertical line '|'

`plotGaussed`(*width*)

Plots the pointprocess as a smoothed time series

`ni.model.pointprocess.`**`PointProcessFromSpikeTimes`**(*times*)

class `ni.model.pointprocess.`**`SimpleFiringRateModel`**

Uses just the firing rate as a predictor

`compare`(*Data*, *Prediction*)

`fit`(*data*)

`loglikelihood`(*Data*, *Prediction*)

`predict`(*Data*)

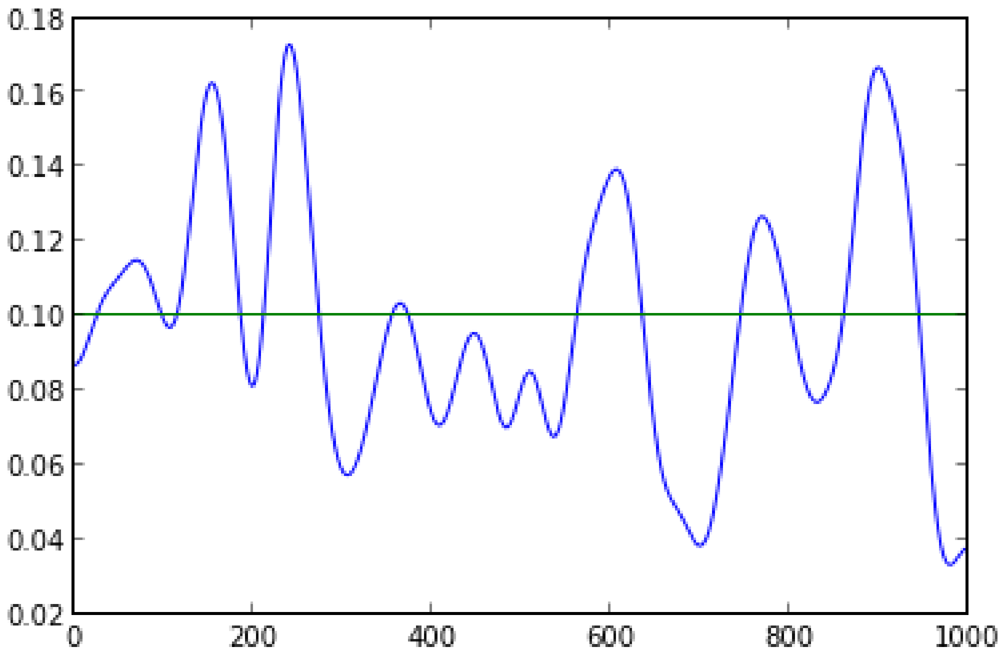
`ni.model.pointprocess.`**`createPoisson`**(*p*, *l*)

This generates a spike sequence of length *l* according to either a fixed firing rate *p*, or a repeated sequence of firing rates if *type(p) == np.ndarray*.

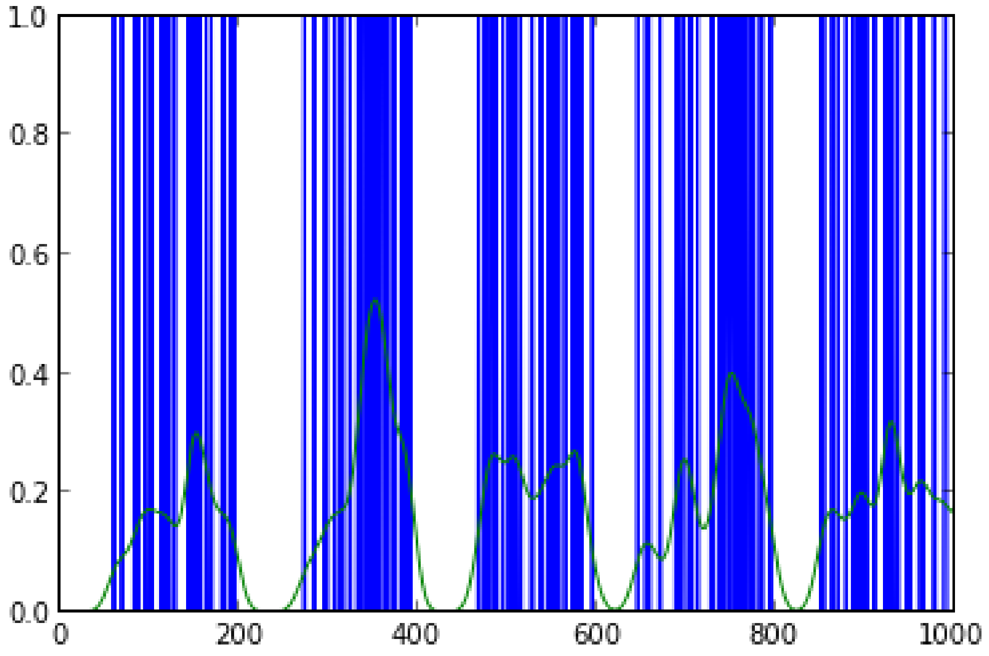
It creates a `ni.model.pointprocess.PointProcess`

Example 1:

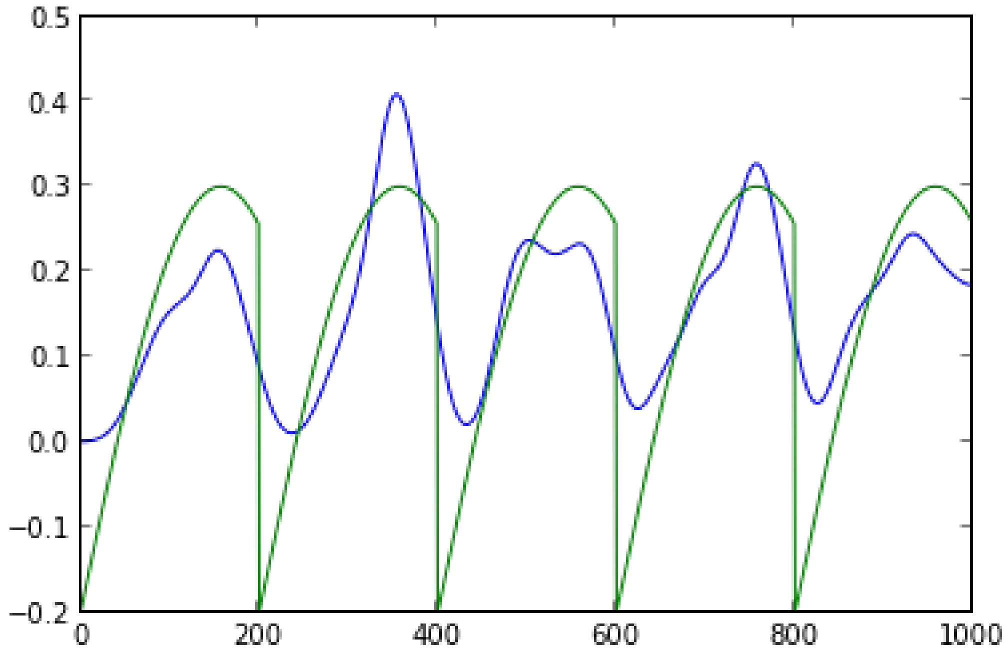
```
p1 = ni.model.pointprocess.createPoisson(0.1,1000)
p1.plotGaussed(20)
plot(p1.frate)
```



```
p2 = ni.model.pointprocess.createPoisson(sin(numpy.array(range(0,200))*0.01)*0.5- 0.2,1000)
p2.plot()
p2.plotGaussed(10)
```



```
p2.plotGaussed(20)
plot(p2.frate)
```



Example with multiple channels:

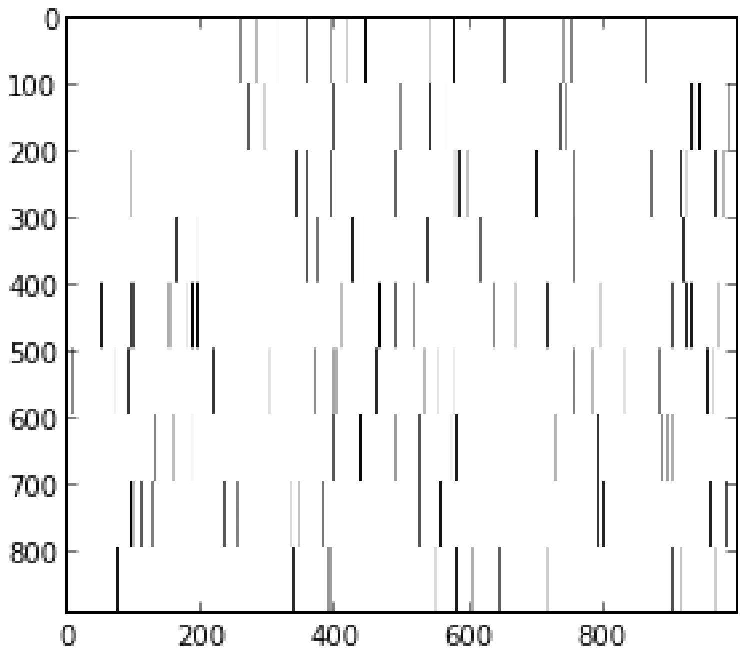
```
frate = (numpy.array(range(0,200))*0.001)*0.2+0.01
channels = 9

dists = [ni.model.pointprocess.createPoisson(frate,1000) for i in range(0,channels)]
#for i in range(0,9): dists[i].plotGaussed(10)
import itertools
```

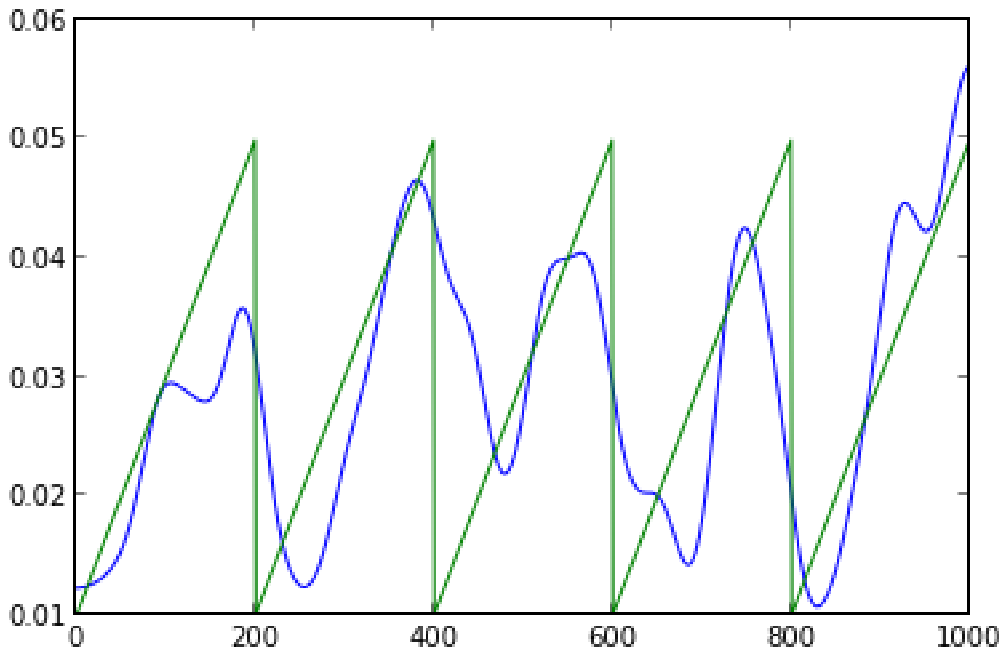
```
spks = np.array([dists[i].getCounts() for i in range(0,channels) for j in range(0,99) ])
imshow(-1*spks)
set_cmap('gray')
```

Will generate:

```
(A plot of spikes)
```



```
ni.model.pointprocess.plotGaussed(np.array([dists[i].getCounts() for i in range(0,channels)]).me
plot(dists[0].frate)
```



`ni.model.pointprocess.`**`getBinary`**`(`*spikes*, *min_length=1*`)`

Gives back a binary array from an array of spike times. The maximum for each bin is 1.

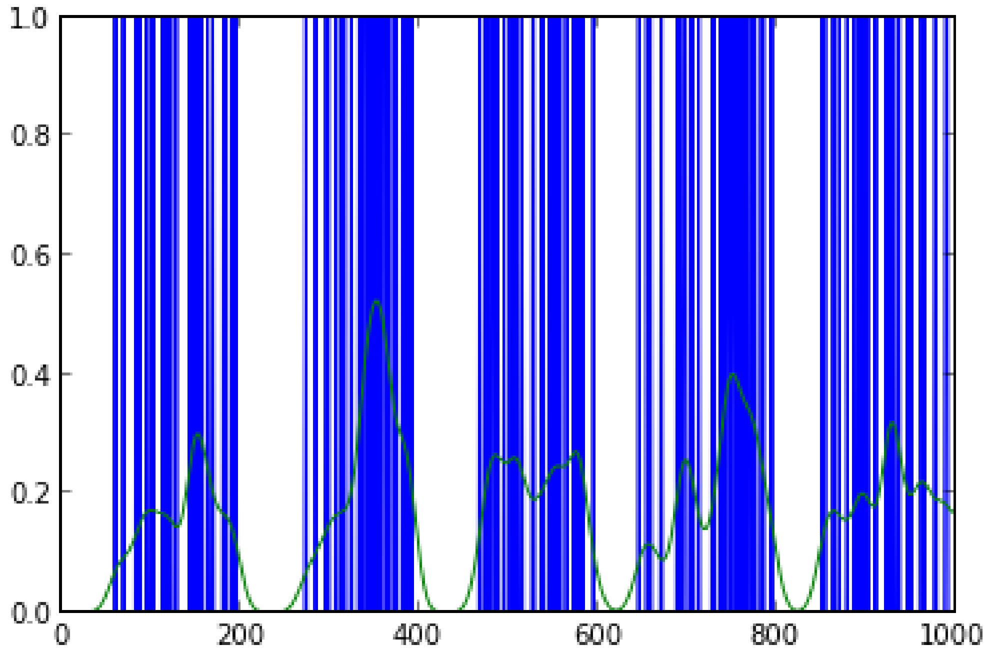
`ni.model.pointprocess.`**`getCounts`**`(`*spikes*`)`

Gives back an array of spike counts from an array of spike times. If the output is suppsed to be a Binomial, use *getBinary* instead.

`ni.model.pointprocess.`**`interspike_interval`**`(`*spikes_a*, *spikes_b=False*`)`

`ni.model.pointprocess.`**`plotGaussed`**`(`*data*, *width*`)`

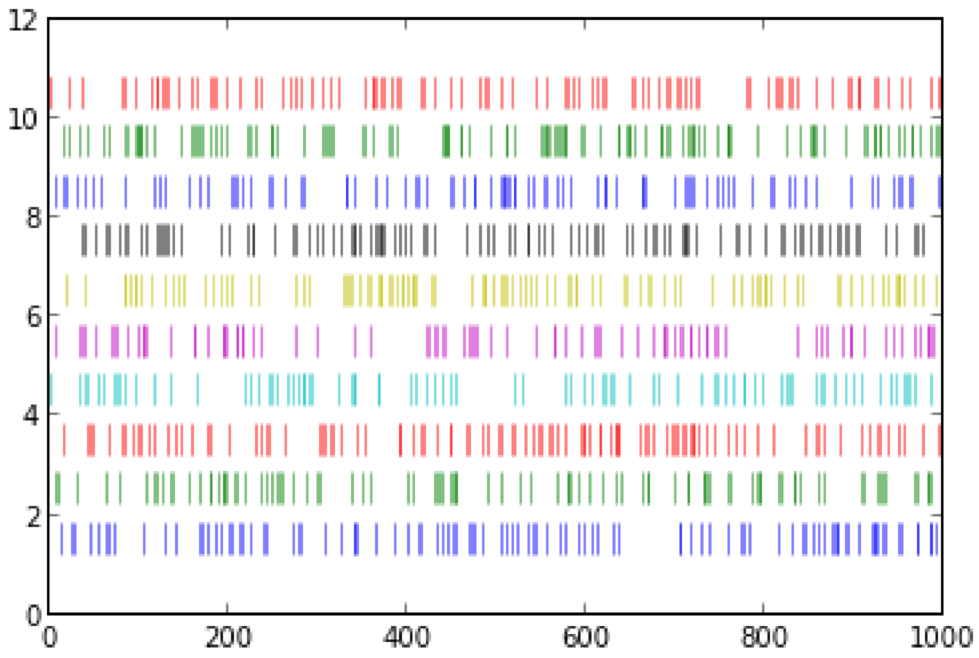
```
p2 = ni.model.pointprocess.createPoisson(sin(numpy.array(range(0,200))*0.01)*0.5- 0.2,1000)
p2.plot()
p2.plotGaussed(10)
```



`ni.model.pointprocess.`**`plotMultiSpikes`**`(`*spikes*`)`

- *spikes* is a binary 2d matrix

Generates something like:



`ni.model.pointprocess.`**`reverse_correlation`**(*spikes_a*, *spikes_b=False*)

1.4 tools Package

1.4.1 `bootstrap` Module

`ni.tools.bootstrap.`**`bootstrap`**(*bootstrap_repetitions*, *model*, *data*, *test_data=[ ]*, *shuffle=True*, *prefix=''*, *bootstrap_data=[ ]*)

A helper function that performs bootstrap evaluation of models.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_repetitions

model

data

'test_data'=[]

'shuffle'=True

'prefix'=' String that is prefixed to the results.

'bootstrap_data'=[] If new data instead of trial shuffling is to be used as bootstrap data, this data should be passed here. The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample*

`ni.tools.bootstrap.`**`bootstrap_samples`**(*bootstrap_data*, *model*, *data*, *test_data=[ ]*, *shuffle=False*, *prefix=''*, *boot_dim='Bootstrap Sample'*)

Performs bootstrap evaluation with bootstrap data.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_data If new data instead of trial shuffling is to be used as bootstrap data, this data should be passed here. The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample*

model

Model to be evaluated. It needs to provide an *x()*, *dm()* and *fit(x=, dm=)*/*fit(data)* method.

data

'test_data'=[]

'shuffle'=True

'prefix'='' String that is prefixed to the results.

boot_dim The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample* or be a list. If some other index should be used as bootstrap samples, *boot_dim* can be set to that.

`ni.tools.bootstrap.`**`bootstrap_time`**(*bootstrap_repetitions*, *model*, *data*, *test_data*=[ ], *prefix*='')

Performs bootstrap evaluation of models.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_repetitions

model

Model to be evaluated. It needs to provide an *x()*, *dm()* and *fit(x=, dm=)* method.

data

'test_data'=[]

'prefix'='' String that is prefixed to the results.

`ni.tools.bootstrap.`**`bootstrap_trials`**(*bootstrap_repetitions*, *model*, *data*, *test_data*=[ ], *shuffle=True*, *prefix*='', *bootstrap_data*=[ ])

Performs bootstrap evaluation by trial shuffling.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_repetitions

Number of bootstrap repetitions

model

Model to be evaluated. It needs to provide an *x()*, *dm()* and *fit(x=, dm=)*/*fit(data)* method.

data

'test_data'=[]

'shuffle'=True

'prefix'='' String that is prefixed to the results.

'bootstrap_data'=[] If new data instead of trial shuffling is to be used as bootstrap data, this data should be passed here. The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample*

`ni.tools.bootstrap.`**`description`**(*prefix=''*, *additional_information=''*)
Describes the common bootstrap output variables as a dictionary. *additional_information* will be appended to each entry, *prefix* will be prepended to each key.

`ni.tools.bootstrap.`**`generate`**(*model*, *bootstrap_repetitions*)

`ni.tools.bootstrap.`**`merge`**(*stats*)

`ni.tools.bootstrap.`**`plotBootstrap`**(*res*, *path*)
Deprecated since version 0.1: use the plot capabilities of the `ni.tools.statcollector.StatCollector`.

`ni.tools.bootstrap.`**`plotCompareBootstrap`**(*reses*, *path*)
Deprecated since version 0.1: use the plot capabilities of the `ni.tools.statcollector.StatCollector`.

1.4.2 `progressbar` Module

`ni.tools.progressbar.`**`progress`**(*a*, *b*)
Undocumented

`ni.tools.progressbar.`**`progress_end`**()
Undocumented

`ni.tools.progressbar.`**`progress_init`**()
Undocumented

1.4.3 `project` Module

NI Project Management

- All steps in a configuration / simulation process will be logged to some folder structure
- after the simulation and even after changing the original code, the results should still be viewable / interpretable with a project viewer
- batches of runs should be easy to batch interpret (characteristic plots etc.)
- **metadata should contain among others:** date software versions configuration options manual comments
- saving of plots/data should be done by the project manager

class `ni.tools.project.`**`Figure`**(*path*, *display=False*, *close=True*)
Figure Context Manager

Can be used with the **with** statement:

```
import ni
x = np.arange(0,10,0.1)
with ni.figure("some_test.png"):
    plot(cos(x))        # plots to a first plot
    with ni.figure("some_other_test.png"):
        plot(-1*np.array(x)) # plots to a second plot
    plot(sin(x))        # plots to the first plot again
```

Or if they are to be used in an interactive console:

```
import ni
x = np.arange(0,10,0.1)
with ni.figure("some_test.png",close=False):
        plot(cos(x))    # plots to a first plot
                with ni.figure("some_other_test.png",close=False):
                        plot(-1*np.array(x)) # plots to a second plot
        plot(sin(x))    # plots to the first plot again
```

Both figures will be displayed, but the second one will remain available after the code. (But keep in mind that in the iPython pylab console, after every input, all figures will be closed)

close()

show(*close=True*)

class ni.tools.project.**Job**(*project*, *session*, *path*, *job_name='‘*, *job_number='‘*, *file='‘*, *status='initializing...'*, *dependencies=[ ]*)

can_run()

get_status()

html_view()

run(*parameters=[ ]*)

save()

set_activity(*msg='‘*)

set_status(*msg='‘*)

update()

class ni.tools.project.**ListContainer**

append(*msg_type*, *priority*, *date*, *job*, *txt*)

clear()

class ni.tools.project.**LogContainer**(*f*)

append(*msg_type*, *priority*, *date*, *job*, *txt*)

clear()

class ni.tools.project.**PickleContainer**(*f*)

append(*msg_type*, *priority*, *date*, *job*, *txt*)

clear()

class ni.tools.project.**Project**(*folder='unsaved_project'*, *name='‘*)

Project Class

loads a Project folder (containing eg. a main.py file)

abandon()

autorun()

clear()

dbg(*txt*, *priority=-1*)

do_log (*b*)

dumpheap ()

err (*txt*, *priority=0*)

execute (*code*, *local_vars={}*, *session=False*)

find_sessions ()

get_parameters_from_job_file ()

get_session_status (*r=False*)

html_view ()

job (*j*)
TODO: rename to something else

job_activate (*j*, *msg='running...'*)

job_done (*j*)

last_run ()

less_running_than (*N*)

log (*txt*, *priority=0*)

msg (*msg_type*, *txt*, *priority=0*)

next ()

next_job (*ignore_dependencies=False*)

parse_job_file (*filename*, *session*)

print_job_status ()

print_long_job_status ()

report (*silent=False*)

reportHTML ()

require_job (*j*)

reset_failed_jobs ()

run (*parameters=*[ ], *job=False*)

save (*name*, *val*)

save_html (*path='project.html'*)

select_session (*path*)

set_session_status (*msg='running...'*)

setup_jobs (*parameter_string=''*)

sibjob (*j*)
Sibling Job

Is on the same level as the previous job (ie. a child of its parent)

subjob (*j*)

superjob ()

update_job_status ()

class `ni.tools.project.`**`Session`**(*project*, *path=''*, *parameter_string=''*)

> **`abandon`**()
>
> **`add_job`**(*job_name=''*, *job_number=''*, ***kwargs*)
>
> **`execute`**(*code*, *local_vars={}*)
>
> **`find_jobs`**()
>
> **`get_status`**()
>
> **`html_view`**()
>
> **`next_job`**(*retry_failed=False*, *ignore_dependencies=False*)
>
> **`parse_job_file`**(*filename*, *parameter_string=''*)
>
> **`print_job_status`**()
>
> **`print_long_job_status`**()
>
> **`reset_failed_jobs`**(*which='failed.'*, *to='pending'*)
>
> **`save_html`**(*path='session.html'*)
>
> **`set_status`**(*msg=''*)
>
> **`setup_jobs`**(*source_file*, *parameter_string=''*)
>
> **`update_job_files`**(*source_file=''*, *parameter_string=''*)
>
> **`update_jobs`**()

class `ni.tools.project.`**`TemporaryJob`**(*project*, *session*, *job_name*)

> Bases: `ni.tools.project.Job`

class `ni.tools.project.`**`TemporarySession`**(*project*)

> Bases: `ni.tools.project.Session`

class `ni.tools.project.`**`VariableContainer`**

`ni.tools.project.`**`atoi`**(*text*)

> converts text containing numbers into ints / used by `natural_keys()`

`ni.tools.project.`**`dbg`**(*txt*, *priority=-1*)

`ni.tools.project.`**`do_log`**(*b*)

`ni.tools.project.`**`dumpheap`**()

`ni.tools.project.`**`err`**(*txt*, *priority=0*)

`ni.tools.project.`**`figure`**(*path*, *display=False*, *close=True*)

> Can be used with the **with** statement:

```
import ni
x = np.arange(0,10,0.1)
with ni.figure("some_test.png"):
    plot(cos(x))          # plots to a first plot
    with ni.figure("some_other_test.png"):
        plot(-1*np.array(x)) # plots to a second plot
    plot(sin(x))          # plots to the first plot again
```

> Or if they are to be used in an interactive console:

```
import ni
x = np.arange(0,10,0.1)
with ni.figure("some_test.png",display=True):
    plot(cos(x))         # plots to a first plot
    with ni.figure("some_other_test.png",close=False):
        plot(-1*np.array(x)) # plots to a second plot
    plot(sin(x))         # plots to the first plot again
```

Both of these figures will be displayed, but the second one will remain open and can be activated again.

`ni.tools.project.`**`job`**`(`*j*`)`

`ni.tools.project.`**`load`**`(`*path*`)`

`ni.tools.project.`**`log`**`(`*txt*, *priority=0*`)`

`ni.tools.project.`**`natural_keys`**`(`*text*`)`
alist.sort(key=natural_keys) sorts in human order (See Toothy's implementation in the comments of http://nedbatchelder.com/blog/200712/human_sorting.html)

`ni.tools.project.`**`natural_sorted`**`(`*l*`)`
sorts a sortable in human order (0 < 20 < 100)

`ni.tools.project.`**`report`**`(`*silent=False*`)`

`ni.tools.project.`**`require_job`**`(`*j*`)`

`ni.tools.project.`**`run`**`()`

`ni.tools.project.`**`save`**`(`*name*, *val*`)`

`ni.tools.project.`**`sibjob`**`(`*j*`)`

`ni.tools.project.`**`subjob`**`(`*j*`)`

`ni.tools.project.`**`superjob`**`()`

1.4.4 `html_view` Module

This module can generate HTML output from text or objects that provide a .html_view() function:

```
import ni
view = ni.View()        # this is a shortcut for ni.tools.html_view.View
view.add("#1/title","This is a test")
view.add("#2/Some Example Models/tabs/",ni.model.ip.Model({'name': 'Basic Model'}))
view.add("#2/Some Example Models/tabs/",ni.model.ip.Model({'autohistory_2d':True, 'name': 'Model wit
view.add("#2/Some Example Models/tabs/",ni.model.ip.Model({'rate':False, 'name': 'Model without Rate
view.add("#3/Some Example Data/tabs/1",ni.data.monkey.Data())
view.render("this_is_a_test.html")
```

class `ni.tools.html_view.`**`Figure`**`(`*view*, *path*, *close=True*, *figsize=False*`)`
Figure Context Manager

Can be used with the **with** statement:

```
import ni
v = ni.View()
x = np.arange(0,10,0.1)
with ni.tools.html_view.Figure(v,"some test"):
    plot(cos(x))         # plots to a first plot
    with ni.tools.html_view.Figure(v,"some other test"):
        plot(-1*np.array(x)) # plots to a second plot
    plot(sin(x))          # plots to the first plot again
v.render("context_manager_test.html")
```

class `ni.tools.html_view.`**`View`**(*path='‘*)

`add`(*path*, *obj*)

`figure`(*path='‘*, *close=True*, *figsize=False*)

Provides a Context Manager for figure management

Should be used if plots are to be used in

Example:

```
import ni
v = ni.View()
x = np.arange(0,10,0.1)
with v.figure("some test"):
    plot(cos(x))                # plot to a first plot
    with v.figure("some other test"):
        plot(-1*np.array(x))    # plot to a second plot
    plot(sin(x))                # plot to the first plot again
v.render("context_manager_test.html")
```

`has`(*path*)

`html_view`()

`load`(*filename*)

`loadList`(*filenames*)

`load_glob`(*filename_template*)

`load_list`(*filenames*)

`parse`(*tree*)

`process`(*obj*, *mode='text'*)

`render`(*path*, *include_files=True*)

`save`(*filename*)

`savefig`(*p='‘*, *fig='‘*, *close=True*)

`ni.tools.html_view.`**`atoi`**(*text*)

`ni.tools.html_view.`**`natural_keys`**(*text*)

alist.sort(key=natural_keys) sorts in human order http://nedbatchelder.com/blog/200712/human_sorting.html (See Toothy’s implementation in the comments)

`ni.tools.html_view.`**`natural_sorted`**(*l*)

sorts a sortable in human order (0 < 20 < 100)

1.4.5 `strap` Module

`ni.tools.strap.`**`bootstrap`**(*bootstrap_repetitions*, *model*, *data*, *test_data=[ ]*, *shuffle=True*, *prefix='‘*, *bootstrap_data=[ ]*)

A helper function that performs bootstrap evaluation of models.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_repetitions

model

data

'test_data'=[]

'shuffle'=True

'prefix'=' String that is prefixed to the results.

'bootstrap_data'=[] If new data instead of trial shuffling is to be used as bootstrap data, this data should be passed here. The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample*

`ni.tools.strap.`**`bootstrap_samples`**(*bootstrap_data*, *model*, *data*, *test_data=[ ]*, *shuffle=False*, *prefix=''*, *boot_dim='Bootstrap Sample'*)

Performs bootstrap evaluation with bootstrap data.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_data If new data instead of trial shuffling is to be used as bootstrap data, this data should be passed here. The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample*

model

Model to be evaluated. It needs to provide an *x()*, *dm()* and *fit(x=, dm=)/fit(data)* method.

data

'test_data'=[]

'shuffle'=True

'prefix'=' String that is prefixed to the results.

boot_dim The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample* or be a list. If some other index should be used as bootstrap samples, *boot_dim* can be set to that.

`ni.tools.strap.`**`bootstrap_time`**(*bootstrap_repetitions*, *model*, *data*, *test_data=[ ]*, *prefix=''*)

Performs bootstrap evaluation of models.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_repetitions

model

Model to be evaluated. It needs to provide an *x()*, *dm()* and *fit(x=, dm=)* method.

data

'test_data'=[]

'prefix'='' String that is prefixed to the results.

`ni.tools.strap.`**`bootstrap_trials`**(*bootstrap_repetitions*, *model*, *data*, *test_data=[ ]*, *shuffle=True*, *prefix=''*, *bootstrap_data=[ ]*)

Performs bootstrap evaluation by trial shuffling.

A Model *model* is fitted with some data *data*, called "actual data" or "D" and subsequently on all of a number of bootstrap samples "D*_n" for n in range(*bootstrap_repetitions*). This yields an *actual fit* and *bootstrap_repetitions* times *boot fit* (or *fit**) for each sample.

Use bootstrap_results for explanations on the dimensions of the result.

bootstrap_repetitions

Number of bootstrap repetitions

model

Model to be evaluated. It needs to provide an *x()*, *dm()* and *fit(x=, dm=)*/*fit(data)* method.

data

'test_data'=[]

'shuffle'=True

'prefix'='' String that is prefixed to the results.

'bootstrap_data'=[] If new data instead of trial shuffling is to be used as bootstrap data, this data should be passed here. The *ni.data.data.Data* Instance should contain an additional index *Bootstrap Sample*

`ni.tools.strap.`**`description`**(*prefix=''*, *additional_information=''*)

Describes the common bootstrap output variables as a dictionary. *additional_information* will be appended to each entry, *prefix* will be prepended to each key.

`ni.tools.strap.`**`generate`**(*model*, *bootstrap_repetitions*)

`ni.tools.strap.`**`merge`**(*stats*)

`ni.tools.strap.`**`plotBootstrap`**(*res*, *path*)

Deprecated since version 0.1: use the plot capabilities of the `ni.tools.statcollector.StatCollector`.

`ni.tools.strap.`**`plotCompareBootstrap`**(*reses*, *path*)

Deprecated since version 0.1: use the plot capabilities of the `ni.tools.statcollector.StatCollector`.

1.4.6 `statcollector` Module

class `ni.tools.statcollector.`**`StatCollector`**(*stat_init={}*)

A class to collect statistics about models. It can be used to analyse nested models, as slashes in the name are interpreted as submodels.

Example:

```
>>> rate model/0
>>> rate model/1
>>> rate model/2
>>> rate model/3
>>> nested model/0
>>> nested model/1
>>> nested model/2
>>> nested model/3
>>> nested model/0/1
>>> nested model/0/2
>>> nested model/0/3
>>> nested model/0/2/1
>>> nested model/0/2/3
>>> nested model/0/2/3/1
```

This example could be generated by fitting a model with a certain number of crosshistory cells. In each iteration the best model is extended by another cell. The nested model can then be evaluated whether it has increasing likelihood and/or eic, aic or other statistics:

```
>>> stats.getModelsOnPath(['nested model',0,2,3,1]).get('eic')
[ -1023, -1020, -900, -950 ]
```

self.stats: a dict of lists, each list containing dicts with: name llf eic, aic, eice, complexity (optional) additional - a dict with more information (ignored for now)

addNode (*name*, *data={}*)
adds the node *name* with the attributes in the dictionary *dic*. If *name* exists, it wll be overwritten.

addToNode (*name*, *dic*)
adds all attributes in the dictionary *dic* to the node *name*

filter (*name*)
returns a StatCollector Object with only a subset of models

get (*dim*)
returns a numpy ndarray with the *dim* attributes of each node that contains *dim*

getChildren ()

getDimensions ()
returns which dimensions are availble for the contained nodes

getList (*dim*)
returns a list with the *dim* attributes of each node that contains *dim*

getModelsOnPath (*name*)
returns models that lead to the node *name*

getNode (*name*)
returns a dictionary with all attributes of the node *name*

getTree (*substitution_patterns=[ ]*)
returns a tuple used by plotTree to plot a tree representation of the nodes.

html_view ()
Generates an html_view of this object.

Example:

```
stats.html_view().render('stats_file.html')
```

keys ()
returns the name of all nodes. Synonym of `StatCollector.nodes()`

load (*filename*)
Loads the StatCollector saved to *filename*.

This file should be a pickled dictionary.

loadList (*filenames*)
loads a list of files. (Alias of `StatCollector.load_list()`)

load_glob (*filename_template*)

load_list (*filenames*)
    loads a list of files

nodes ()
    returns the name of all nodes. Synonym of `StatCollector.keys()`

plotHist (*path*, *width*, *dims*)
    plots a histogram of each dimension in *dims*

plotTree (*dim*, *substitution_patterns*=[ ], *line_kwargs={}*, *marker_kwargs={}*, *right_to_left=False*)
    Plots a tree of the nodes, using *dim* as the height, if the node contains *dim*.

Slashes in the model names will be used as the different levels in the tree. The order of the parts between the slashes is ignored for now (3/4 and 4/3 are the same).

substitution_patterns may contain substitution patterns (used to connect nodes) for re.sub as a three tuple (pattern, substitute, color), where color is the color that will be assigned to this connection

line_kwargs and *marker_kwargs* can contain arguments in a dictionary to alter the options to set lines or markers. The dictionary will be passed on to the plot function.

right_to_left determines, whether the plot is plotted from left to right (default) or the other way around (*right_to_left* = True).

prefix (*prefix='/'*)
    Makes the last node a part of the property name

pull_from_inner_dict (*from_dictionary='statistics'*, *from_key='bic'*, *to_key='BIC'*)
    If a dictionary is added as a dimension, this function can pull values from that dictionary and add them to each node that has the specific dictionary. As the bootstrap functions add the statistics dictionary of the model fit to the node, this function has to be used to eg. access the BIC criterion (which is why this is the default from and to keys).

re (*regex*)
    returns a StatCollector Object with only a subset of models

rename (*pattern*, *substitution*)
    Renames nodes with the regex pattern *pattern* just like `re.sub()`.

Example to rename different number of knots to a tree, where each the increase in knots is counted as a submodel:

```
statsr = stats.rename(r'50','30/50').rename(r'30','20/30').rename(r'20','10/20').rename(r'10
```

rename_value_to_tree (*value=-1*)
    Renames nodes, such that increases of one value are counted as submodels

value is the index (of the slash splitted node name) of the value that is to be replaced. The default is -1, ie. the last value.

Example:

```
import ni
stats = ni.StatCollector()
stats.addNode('Model 0/10',{'a':100})
stats.addNode('Model 0/20',{'a':100})
stats.addNode('Model 0/30',{'a':100})
stats.addNode('Model 0/50',{'a':100})
stats.addNode('Model 0/1000',{'a':100})
stats.rename_value_to_tree().plotTree('a')
```

The example will rename the last node to 'Model 0/10/20/30/50/1000'

save (*filename*)
Saves the StatCollector to *filename*.

This file will be a pickled dictionary.

set (*name*, *key*, *value*)
sets one attribute for node *name*

split (*keys*)
Takes a portion of the property names and makes it a node

`ni.tools.statcollector.`**atoi**(*text*)
converts text containing numbers into ints / used by `natural_keys()`

`ni.tools.statcollector.`**listToPath**(*name*)
Creates a string that joins a list together with slashes. The list can contain strings and numbers.

`ni.tools.statcollector.`**natural_keys**(*text*)
alist.sort(key=natural_keys) sorts in human order (See Toothy's implementation in the comments of http://nedbatchelder.com/blog/200712/human_sorting.html)

`ni.tools.statcollector.`**natural_sorted**(*l*)
sorts a sortable in human order (0 < 20 < 100)

CHAPTER TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

THREE

SOME EXAMPLES

This code can be executed in an ipython notebook (to be used in your browser) or qtconsole.

To start a notebook:

```
ipython notebook --pylab=inline
```

To start the qtconsole:

```
ipython qtconsole --pylab=inline
```

This will load matplotlib functions for easy plotting and even provide many aliases that use names similar to the ones used in matlab.

3.1 Basic Example

The most basic example (if a path to the data is set and accessible) of how to use the toolbox is:

```
import ni
data = ni.data.monkey.Data(condition = 0)
model = ni.model.ip.Model({'cell':4,'crosshistory':[6,7]})
fm = model.fit(data.trial(range(data.nr_trials/2)))
plot(fm.prototypes()['rate'],'k')
```

The first line loads data from the monkey module with the default filename. The second line creates a model for cell 4, using 6 and 7 as crosshistory dependencies. Then the model is fitted on half of the data and the rate component is displayed.

3.2 Generating example data

If no data is available, some can be generated with the net_sim module:

```
import ni
conf = ni.model.net_sim.SimulationConfiguration()
conf.Nneur = 10
conf.Ntrials = 100
r = ni.model.net_sim.simulate(conf)
print r.data
```

Will output:

```
Spike data: 1 Condition(s) 100 Trial(s) of 10 Cell(s) in 1000 Time step(s). No other data.
```

The data can then be used with a model:

```
model = ni.model.ip.Model({'cell':4,'crosshistory':[6,7]})
fm = model.fit(r.data.trial(range(r.data.nr_trials/2)))
fm.plot_prototypes()
```

3.3 Using other sources of data

If the **nltk** (http://nltk.org/) is installed, you can also generate data like this:

```
import ni
import numpy as np
import pandas
from nltk.corpus import genesis
from ni.model.pointprocess import getBinary
trial_length = 1000 # use blocks of this many characters as a trial
d = []
index_tuples = []
for (condition, txt_file) in enumerate(['english-kjv.txt','finnish.txt','german.txt','french.txt']):
        s = genesis.raw(txt_file).replace('.\n',' ').replace('\n',' ').replace('.',' ') # to make the
        for t in range(len(s)/trial_length):
                for (cell, letter) in enumerate([' ', 'a', 'e', 'i']):
                        d.append(list(getBinary( np.cumsum([len(w)+1 for w in s[(t*trial_length):((t
                        index_tuples.append((condition,t,cell))
index = pandas.MultiIndex.from_tuples(index_tuples, names=['Condition','Trial','Cell'])
data = ni.Data(pandas.DataFrame(d, index = index))

model = ni.model.ip.Model({'history_length':10, 'rate':False})
for condition in range(4):
        fm = model.fit(data.condition(condition).trial(range(50)))
        print str(condition)+': ' + ' '.join([str(fm.compare(data.condition(i).trial(range(50,100)))
```

3.4 Adding a custom kernel

To add a designmatrix component to the default model, include it in the *custom_components* list of the configuration.:

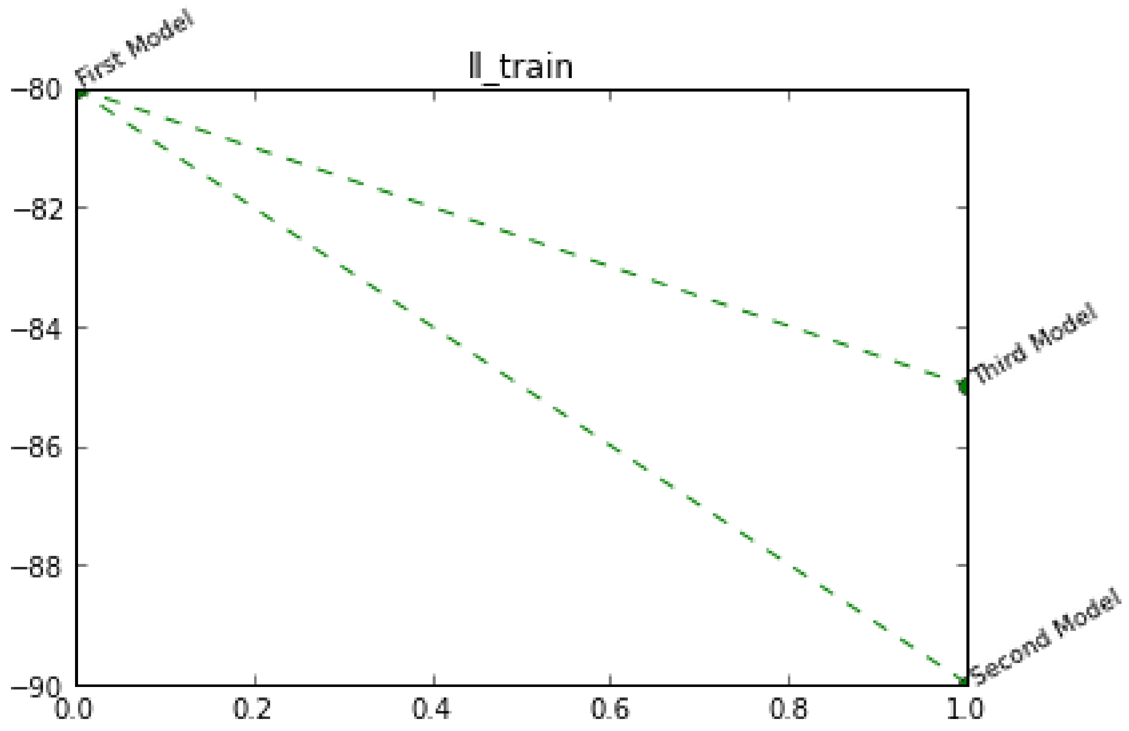
```
import ni
data = ni.data.monkey.Data(condition=0)
long_kernel = ni.model.create_splines.create_splines_linspace(data.nr_trials * data.trial_length,5,Fa
ni.model.ip.Model({'custom_components': [ ni.model.designmatrix.Component(header='trend', kernel=long
```

3.5 StatCollectors

An example of adding something to a StatCollector:

```
import ni
stat = ni.StatCollector()
stat.addNode("Model 0", {'name': 'First Model',
                         'll_test': -240,
                         'll_train': -80,
                         'complexity':10})
stat.addNode("Model 0/1", {'name': 'Second Model',
                           'll_test': -100,
                           'll_train': -90,
                           'complexity':14})
stat.addNode("Model 0/2", {'name': 'Third Model',
                           'll_test': -130,
                           'll_train': -85,
                           'complexity':14})
stat.plotTree('ll_train')
```

Will output:



Mostly, the output of `ni.tools.bootstrap` functions will be added to the StatCollector, containing all the important information.

PYTHON MODULE INDEX