CorrectCoxVpulse– a python function to correct for space-clamp errors in voltage-clamp test-pulse experiments

Ricardo Murphy (ricardo_murphy@fastmail.fm)

Introduction

Whole-cell voltage-clamp is a widely used method for estimating the voltage (V) and time (t) dependence of membrane currents (Sakmann and Neher 1995). This allows the construction of Hodgkin-Huxley type models of V-dependent membrane conductances, which may then be implemented in compartmental models of cells. Typically rectangular test pulses are applied to various V from some holding potential, and the time course of the injected current, $I(t)$, is recorded. Other protocols are also used such as the V-ramp (e.g. Murphy *et al*., 2024a), but here we will focus on the V-clamp test-pulse protocol. A serious problem when applying the V-clamp technique to neurons is that V is clamped only at the recording site. This will often be the soma, or perhaps an axonal bleb or varicosity, although proximal dendrites can also be patched. Away from the recording site V will not be clamped at the desired values ("space-clamp error"), but the membrane current at those points will contribute to the current measured at the recording site. Hence the interpretation of the recorded values of I and V is not straightforward; rather one must in some way correct for this space-clamp error. Various approaches have been tried (reviewed briefly by Murphy *et al*., 2024a). Here we take as our starting point the method described by Cox (2008), which provides an initial estimate of the membrane current density, $i(t,V)$. This is then improved upon iteratively. Full details of the method are given in Murphy *et al.* (2024a,b). Here I describe the use of a Python function `CorrectCoxVpulse`, which operates in conjunction with the simulator NEURON (Carnevale and Hines, 2009) to correct for space-clamp errors in V-clamp test-pulse data and so return an estimate of $i(t,V)$. Suitable functions are fitted to these data to obtain estimates of the V-dependence of kinetic parameters in a Hodgkin-Huxley style model. At present only depolarising test pulses are supported. To use `CorrectCoxVpulse` you need to install the statistical software *R* and the Python module `rpy2` (see Technical Notes). CorrectCoxVpulse was developed with NEURON Ver. 7.5.

Using the function CorrectCoxVpulse

The folder `CorrectCoxVpulseExample` contains example data files, parameter files, function files and output, as well as a NEURON model (the .hoc and .mod files; remember to compile the latter, perhaps with `Compile_mods.bat`). You will need two text files containing leak-subtracted currents: (1) control; (2) with the conductance of interest blocked. The example data files are `I_control.txt` and `I_blocked.txt`. Your data must conform to this format, specifically a column of sampling times followed by columns of injected current recorded at increasing test-pulse voltages. You will also need two corresponding parameter files (`control_parameters.txt` and `blocked_parameters.txt` are provided as examples). You can use any parameter file names you like, and you can also add your own text

to the parameter files. But parameter names and the colons in the parameter files must not be changed. Also each parameter must be assigned a value. As some parameters are needed by the NEURON model, these can be loaded from the parameter file by your hoc script (see `mossy_fiber_run.hoc`). It can also load the name of the parameter file from the file `parameter_file.txt` (first line). The meaning of the parameters is given in the Table 1. Note that if filenames include a path, it must be specified using "/", not "\". You will also need two function files, specifying the functions that are to be fitted to the estimated $i(t)$ data at each V (described in detail below); `blocked_functions.txt` and `control_functions.txt` are provided as examples. Extra flexibility is provided by files `user_parameters.txt` and `user_functions.py` (also described in detail below). The latter file must be present in the current folder, even if you don't have any use for it.

Put `CorrectCoxVpulse.py` and your own Python script(s) into the same (current) folder as the NEURON model, data, function and parameter files, and import `CorrectCoxVpulse` by putting a line like the following near the start of your script:

```
import CorrectCoxVpulse as cox
```

Calling the function from your script is simple; `result` is a dictionary (see Table 2):

```
result = cox.CorrectCoxVpulse('blocked_parameters.txt')
```

Example scripts are provided (`CorrectCoxVpulseExample...py`). The analysis is done in two sweeps: (1) blocked; (2) control. $i(t,V)$ in the blocked condition is then subtracted from that in control to get the current density of interest. So, in the case of the example data, you would first execute the above line. Final results are copied to the output folder (`outputdir = blocked` in the present case) specified in the parameter file; output file names are given in Table 3. If you wish the temporary folder `coxdir` to be copied to the output folder, add an argument `saveCox = 'yes'` when calling `CorrectCoxVpulse`. The estimated parameters of $i(t,V)$ in the blocked condition are saved by `CorrectCoxVpulse` to file `blocked/cdn_#_paras.txt` (where #is the best iteration; of course you may use a different output folder name). Enter that name (here `blocked/cdn_4_paras.txt`) into the control parameter file (`control_parameters.txt` in this case; see Table 1) and rerun `CorrectCoxVpulse`:

```
result = cox.CorrectCoxVpulse('control_parameters.txt').
```

Again output, including the parameters for the conductance of interest (here in `cdn_5_paras.txt`), is sent to the output folder (`control` in this case).

During the optimisation process, a number of temporary files are created in the current folder. These files, which are described in Table 4, allow your NEURON program to interact with `CorrectCoxVpulse`. During each iteration, once it has estimated the $i(t,V)$ model parameters, `CorrectCoxVpulse` will run your NEURON model. The first thing this model should do is load the file `parameter_file.txt` to find out what the parameter file name is (first line). The second line in `parameter_file.txt` will be one of the following:

(1) `Blocked: 0`. This is the first iteration under control conditions, and you need to generate a file (`cdn_blocked.txt`) containing $i(t,V)$ data in the blocked state (i.e. with the conductance of interest set to zero in the model) so that it can be subtracted by `CorrectCoxVpulse` from its estimate of $i(t,V)$ in control. Load the relevant kinetic parameters from the file specified by `Model_parameter_file`… `blocked_condition` (Table 1). Also set $R_{access} = 0$, because `CorrectCoxVpulse` always estimates $i(t,V)$ assuming $R_{access} = 0$, even if you have $R_{access} > 0$ in your parameter file.

(2) `Iteration: #`. The iteration number (#). Load the HH model kinetic parameters from `cdn_paras.txt` and the leak parameters from `cdL_paras.txt` (Table 4b). Then run a regular simulation of a *V*-clamp test-pulse experiment, i.e. with R_{access} as set in the parameter file. If this is the blocked condition, then the conductance of interest should be set to zero in the model and `Model_parameter_file`… `blocked_condition` should be `none`.

(3) `Best_iteration: #`. This is the best fit (iteration #). Load model parameters for this iteration and any other relevant data (e.g. for plotting) from the output folder (see `mossy_fiber_run.hoc`). Do a simulation with $R_{access}= 0$ and save the resulting $i(t,V)$ data to `outputdir/cdn_#_Raccess=0.txt` (replace `outputdir` with the your folder name and `#` with the iteration number). This is to allow comparison of $i(t,V)$ recorded in the NEURON model with the final estimates generated by `CorrectCoxVpulse` (see below).

When the current simulation is finished, create a file `flag.txt` with the following hoc code (or similar) and then quit:

```
objref file
//Notify CorrectCoxVpulse.py that NEURON is done.
proc finish() {
   file = new File()
   file.wopen("flag.txt")
   file.printf("%d\r",$1)
   file.close()
   quit()
}
```

The behaviour of `CorrectCoxVpulse` depends on the value of the argument `$1`: -1, terminate; 0, redo the last iteration, having made some adjustments (e.g. to the fit functions, although this is not recommended); 1, proceed to the next iteration.

`CorrectCoxVpulse` produces various plots at the start and end of the optimisation. Hopefully they are self-explanatory. Of note is the plot “Leak-subtracted current densities predicted by the model”, which compares the NEURON model predictions for $i(t,V)$ with the best estimates obtained by `CorrectCoxVpulse` (i.e. those minimising the *I* RMS error). The

NEURON predictions are loaded from your file `outputdir/cdn_#_Raccess=0.txt` (see above) to ensure a valid comparison (since `CorrectCoxVpulse` estimates $i(t,V)$ assuming $R_{access} = 0$). If you wish to see plots for intermediate iterations (e.g. for troubleshooting) add an argument `do_plots = 'yes'` when calling `CorrectCoxVpulse`.

To run a series of optimisations using different parameter values (e.g. different R_{seal} values) put the call to `CorrectCoxVpulse` in a loop, creating a new parameter file (containing a different output folder name) for each turn of the loop. E.g.

```
Rseal = [9, 10, 11]; j = 0
file = open('parameters.txt','r')
paras = file.readlines(); file.close()
while j < len(paras):
    if 'Rseal(GOhm)' in paras[j]: jRs = j
    if 'outputdir' in paras[j]: jout = j
    j = j + 1
i = 0
while i < len(Rseal):
    print('')
    print('Rseal(GOhm): '+str(Rseal[i]))
    outputdir = 'Rseal='+str(Rseal[i])
    paras[jRs] = 'Rseal(GOhm): '+str(Rseal[i])+'\n'
    paras[jout] = 'outputdir: '+outputdir+'\n'
    parafile = 'parameters_Rseal='+str(Rseal[i])+'.txt'
    file = open(parafile,'w')
    file.writelines(paras); file.close()
    result = cox.CorrectCoxVpulse(parafile, saveCox = 'yes')
    i = i + 1
```

Function files

For each iteration, `CorrectCoxVpulse` estimates $i(t,V)$ as described by Murphy et. al (2024a,b) and then fits a function to the $i(t)$ data at each V using the *SciPy* function `optimize.least_squares`.. These functions are selected in function files; examples are `blocked_functions.txt` and `control_functions.txt`. The first few lines list the available functions. Each line consists of a function name, a colon, and the function expression (which, however, is not evaluated; it's just for information). A number of predefined functions are supplied: `A^k…(A^k)IIf`, where `A` and `I` indicate activation and inactivation, respectively, while `II` implies both fast and slow inactivation. Functions `A^k+A`, `A^k+L` and `0+A` have a slowly activating component which is absent in `A^k`. On the other hand, the fast activating component is absent in `0+A`. Every variable in these functions except `k` (a fixed exponent) and `t` (time) is an adjustable parameter that will be estimated by least-squares fitting. Be aware that all these functions assume that the conductance of interest is initially completely deactivated, and

that any inactivation has been completely removed. You can also specify your own functions, which are defined in `user_functions.py` (see below). `myfunc` in `CorrectCoxVpulseExample/control_functions.txt` is an example.

Next come the values of `k` and the model reversal potential (`Erev(mV)`). And after that come a series of lines determining how to extrapolate time constants at low and high *V* when a component of *i* is not present. [Components are numbered 1 and 2; if you want more you'll have to supply them yourself.] These lines come in pairs:

`tau…_extrapolation_(…_V):` *string*
`tau…_(ms):` *value*

string and *value* control the method of extrapolation in the following way (V_0 is the lowest or highest command *V* and τ is the nearest available estimated value):

string	Method of extrapolation
`constant`	`tau…` = *value*.
`last`	`tau…` = τ (*value* is ignored).
`linear`	`tau…` is linearly interpolated between *value* at V_0 and τ.
ΔV (float)	`tau…` is assumed to decay exponentially from a value of τ to a plateau value of *value* with a half *V* of ΔV.

In general one would expect such extrapolation to be needed only at low *V*, but an option for high-*V* extrapolation is provided in case it should ever be needed.

Finally there is a table of voltages and functions to be fitted to *i*(*t*) at those *V*. `CorrectCoxVpulse` will do its best to choose reasonable starting values for the parameters prior to fitting. But its best may not be good enough, resulting in crazy fits or program termination. In that case you can supply your own starting values by placing parameter-name, value pairs after the function name, e.g.

```
-42     (A^k)Ic       tauh1 10
-32     (A^k)IIc      tauh1 10  tauh2 100
```

The `taum…` and `tauh…` are activation and inactivation time constants, respectively. Note that all time constants are in ms (like *t*). `h1inf` and `h2inf` are stationary inactivation variables as $t \rightarrow \infty$, and so ideally should end up in the range (0,1). `i1inf` and `i2inf` are stationary current densities (mA cm^{-2}) as $t \rightarrow \infty$. For inactivating *i*, `i1inf` is the value in the absence of inactivation. The remaining adjustable parameter is the slope `b`, which occurs in functions which approach a straight line for large *t*, presumably because the relevant time constant is so large. For

inactivation (functions `(A^k)Ib`, `(A^k)IIb` and `(A^k)IIe`) inactivation is assumed to be complete (i.e. `h#inf` = 0) so that `b` = –`1/tauh#`. For activation (function `A^k+L`), `b` = `i2inf/taum2`, and so to disentangle these two parameters, `taum2` is set to a multiple (`taum_mult`) of the trace length. The default value of `taum_mult` is 5. You can set your own value by adding an argument "`taum_mult = #`" when calling `CorrectCoxVpulse`. Thus `b` is inversely related to a time constant, and it is the estimate of this time constant that is reported in file `cdn_paras.txt`, together with the other parameter estimates.

The number of parameters reported in `cdn_paras.txt` is determined by the function in the function table with the largest number of parameters (the full model). Any other models fitted are reduced forms of the full model. Extrapolation (as described above) or, if necessary, linear interpolation is used to fill in any missing values, i.e. when fitting a model that is reduced relative to the full model. For example, in the provided file `blocked_functions.txt`, `(A^k)IIb` is the full model, but for $V \leq -32$ mV the reduced models `A^k` and `(A^k)Ic` are fitted. If the full model includes an inactivating component #, but it is absent at some V (i.e. you fit a reduced model lacking component # at that V), `h#inf` is set to unity. If a reduced function contains an inactivating component # but there is no corresponding `h#inf` (e.g. (`A^k)Ia`), inactivation of that component is assumed to be complete (i.e. `h#inf` = 0).

Load the parameters in `cdn_paras.txt` to set up Hodgkin-Huxley type channel mechanisms (.mod) files. Example .mod files are provided. Use of the time constants (`tau…`) and stationary inactivation variables (`h#inf`) should be straightforward. Setting up the stationary activation variables (`m1inf` and, where appropriate, `m2inf`) is a little more tricky. Perhaps there is more than one way to do this, but the provided .mod files employ the method described by Murphy *et al.* (2024b):

$$\mathrm{m1inf}(V) = \left(\frac{(V_{\max} - E_{\mathrm{rev}}) \times \mathrm{i1inf}(V)}{(V - E_{\mathrm{rev}}) \times \mathrm{i1inf}(V_{\max})} \right)^{1/k}, \quad \mathrm{m2inf}(V) = \frac{(V_{\max} - E_{\mathrm{rev}}) \times \mathrm{i2inf}(V)}{(V - E_{\mathrm{rev}}) \times \mathrm{i2inf}(V_{\max})}$$

where $V_{\max}$ is the maximum command V. The full models associated with the provided .mod files are as follows:

MFB_KA.mod	A^k
MFB_KAA.mod	A^k+A
MFB_KAAadhoc.mod	A^k+A + *ad hoc* function (see below).
MFB_KAI.mod	(A^k)Ic
MFB_KAII.mod	(A^k)IIc
MFB_KAIIadhoc.mod	(A^k)IIc + *ad hoc* function (see below).

user_parameters.txt and user_functions.py

The second of these files must be present in the current directory even if you don't use it. As the name suggests, you can place your own parameters in `user_parameters.txt`; see `CorrectCoxVpulseExample/user_parameters.txt` for an example. In fact you can place any text lines you like; they will not be altered but just passed as a list (`user_paras`) to the user-defined functions in `user_functions.py`. Another parameter passed to these functions is `blockedfile`, which is the name of the model parameter file in the blocked condition (e.g. `blocked/cdn_4_paras.txt`). Of course if you're currently analysing the blocked condition then `blockedfile` = `none`. Or in other words if `blockedfile` ≠ `none`, you must be analysing data from the control condition. In `user_functions.py` you can define your own functions to fit to $i(t)$, and adjust the parameter table (`paratable`) saved to `cdn_paras.txt`. It is important to ensure that a parameter estimate ends up in the correct place in the table. For example, in the supplied function `0+A`, component `1` is absent but, after fitting, `paratable[i,0]` and `paratable[i,1]` contain the estimates of `i2inf` and `taum2` (here `i` indicates the command voltage). But from the structure of the table (see the column names) these should be in `paratable[i,2]` and `paratable[i,3]`, `paratable[i,0]` should be zero, and `paratable[i,1]` should have an extrapolated value. `CorrectCoxVpulse`.makes the necessary adjustments, but you will need to take care of `paratable` for your own functions. Note also that if you want one of your functions to be the full model, you should include # in the function name. You can also define an *ad hoc* function with fixed parameters that will be added to the fit functions in order to fine tune the fit. (Murphy *et al.*, 2024b). See `CorrectCoxVpulseExample/user_functions.py` for further information on defining your own functions and adjusting `paratable`.

Table 1. Definition of parameters in the parameters file

Parameter	Example	Description
neuron_path	C:/nrn/bin	Omit this line if you want your system to find neuron.exe.
neuron_file	model.hoc	NEURON model file.
maxit	9	Maximum number of iterations.
|DV|(mV)	0.1	See δ in Eq. (2) of Murphy *et al*. (2024b).
Iunit	nA	Unit of current in the data file (pA or nA; the latter will be converted to the former).
tunit	ms	Unit of time in the data file (ms or s; the latter will be converted to the former).
Data_delimeter	tab	In the data file (space or tab).
Number_of_header_lines_to_skip…	10	In the data file (excluding column names).
I*(V)_interpolation	modified	Type of interpolation (see Technical Notes).
Stop_on_minimum_RMS_error	no	If yes, iterations will stop at the first minimum RMS error encountered.
Decimate_data_by_a_factor_of	10	Decimate the data in the data file by this factor.
Moving_average_window_(ms)	1	The data will be smoothed with this moving average (or not if 0).
Fraction_of_trace_for_estimating_SD…	0.25	This terminal fraction of the $I(t)$ trace will be used to estimate SD(I)[1].
V_LJP_corrected	yes	If no, a correction will be applied (see LJP).
Leak-subtracted_currents_in_file	I_control.txt	The input data file (columns: t $I(V_1)$ $I(V_2)$ $I(V_3)$…, where V_i increases monotonically).
Model_parameter_file…blocked_condition	none	Or give a file name such as blocked/cdn_4_paras.txt.

Method_for…the_leak_current…iL_(0-3)	1	Method used to estimate the leak current density i_L[2].
Fit_functions_for_i(t,V)_in_file	control_funcs.txt	These will be fitted to the estimated current density $i(t)$ at each V.
Print_parameter_estimates_(yes/no)	no	If yes, print the initial and final parameter estimates for the fitted functions.
Print_warnings…	no	Do (yes) or don't (no).
Add_this_ad_hoc_function_to_i(t,V)	none	See CorrectCoxVpulseExample/user_functions.py.
Exclude_this_initial_interval_(ms)…	5	If you want to avoid an initial artifact in $i(t)$.
Constrain_i(t,V)_to_be_positive	no	Do (yes) or don't (no).
coxdir	tempCox	Temporary folder for intermediate data.
outputdir	output	Output folder (absolute or relative); **will be deleted if it already exists**.
Area(um^2)	100	Membrane area of isopotential compartment (e.g. soma, bleb, dendritic section); nseg = 1.
diam(um)	0.4 0.4	List of neurite diameters (minimum 2).
Ra(Ohm.cm)	100	Axial resistivity.
Raccess(MOhm)	10	Patch access resistance.
Rseal(GOhm)	Inf	Patch seal resistance.
Cm(uF/cm^2)	1.0	Membrane specific capacitance.
LJP(mV)	2.0	Liquid junction potential[3].
tstart(ms)	100	Pulse on time.
tstop(ms)	400	Pulse off time (includes tstart).
Ihold(pA)	-30	Holding current.

`Vhold(mV)`	`-90`	Holding voltage.
`Number_of_test_pulses`	`10`	What it says.
`Vcmnd(mV)`	`-80 -70 …`	Test-pulse voltages.
`Leak-subtraction_DV(mV)`	`-5`	*V* pulse used for leak subtraction.
`Leak-subtraction_DI(pA)`	`-3`	Stationary change in current observed on applying the leak-subtraction pulse.
`poly`	`gcv`	Method for smoothing $I(V)$ and $I^*(V)$[4].
`Plot_I(V)…at_these_times_(ms;_max=12)`	`0 2 4 10…`	To see how the estimates of $I(V)$ ($k = 0$) and $I^*(V)$ ($k > 0$) look for each iteration.

[1]The minimum RMS error will inevitably be larger than SD(*I*) because of lack of fit (also SD(*I*) may be underestimated by the fitted polynomials, which ignore autocorrelation). You can see just how much larger by inspecting `IholdVholdVcrctVrestSDI.txt`. Note that SD(*I*) is a pooled value from all the *I* traces taken together. SD(*I*) values for the individual traces (i.e. for each *V*) are saved to `SDI.txt`.

[2]Methods for estimating $i_L = g_L(V–E_L)$ (g_L = leak conductance, E_L = reversal potential). For methods 1 and 2, $g_L$and E_L are saved to `cdL_paras.txt`; for method 0, the user must supply this file. For methods 0-2, $i_L(V)$ is saved to `cdL.txt`. For method 3, the user must supply `cdL.txt` [two columns: *V* (mV) and i_L (mA cm^{-2}), with names]; E_L and g_L are not defined.

`0`	Load user-specified values of g_L and E_L from `cdL_paras.txt`. Calculate $i_L(V)$.
`1`	Set $i_L(V) = i(V,0)$, i.e. the current density immediately following the *V*-pulse onset. Also estimate g_L and E_L from a regression line fitted to $i(V,0)$.
`2`	Estimate g_L and E_L from a regression line fitted to $i(V,0)$. Calculate $i_L(V)$.
`3`	Load a user-specified $i_L(V)$ from `cdL.txt`.

[3]Used to calculate the leak current through the seal resistance, $(V + \mathrm{LJP})/R_{seal}$. Also to correct *V* values if they aren't already corrected.

[4]Smoothing polynomial(s) for to $I(V)$ and $I^*(V)$ (see also Technical Notes):

`3`	Fit a polynomial (cubic in this case). Fitted with *NUMPY* function `polyfit`.
`4 4.0`	Find the 'best-fit' polynomial of degree ≤ 4 using forward selection with an *F*-to-enter value (float) of 4.0 (Kutner *et al.*, 2004).
`2 3 -80`	Splice together polynomials of degrees 2 and 3 at $V \approx -80$ mV (initial guess). Fitted with *SciPy* function `optimize.least_squares`.
`cv`	Use cross validation (requires *R* and Python module `rpy2`).
`gcv`	Use generalised cross validation (requires *R* and Python module `rpy2`).

Table 2. Definition of entries in the `result` dictionary.

Key	**Description**
`Residuals(pA)`	An $n \times n_V$ Numpy array of residual errors (observed I – NEURON-model-predicted I), where n is the number of samples and n_V the number of V pulses.
`output_folder`	As specified in the parameter file (`outputdir`).
`Best iteration`	Iteration (k) with the smallest RMS error.

Table 3. Output file names. # is the best iteration number.
Files saved by `CorrectCoxVpulse` to the output folder (as specified by `outputdir`):

File name	Description
`cd_#.txt`	$i(t,V)$ estimated by `CorrectCoxVpulse`.
`cd_fit_#.txt`	Fitted $i(t,V)$ function values.
`cdL_#.txt`	Estimated $i_L(V)$.
`cdL_#_paras.txt`	Estimated g_L and E_L.
`cdn_#_paras.txt`	Current density, $i(t,V)$, parameters.
`Ierror_#.txt`	ε = observed I – model-predicted I (total I, including I_{hold}, but corrected for R_{seal}). Used by `CorrectCoxVpulse` to estimate $i(t,V)$.
`Iresids_#.txt`	Observed I – model-predicted I (leak-subtracted; used to estimate the RMS error).
`Raw_tDI.txt`	Raw observed leak-subtracted currents.
`Crct_tDI.txt`	Leak-subtracted currents corrected for R_{seal}.
`Crct_tV.txt`	$V(t)$ corrected for R_{access}.
`IholdVholdVcrctVrestSDI.txt`	I_{hold} corrected for R_{seal}; V_{hold}, V_{rest} and mean-pulse V corrected for R_{access}; estimated SD(I).
`printout.txt`	The run time and RMS error for each iteration.
`cdn_#.txt` (optional)	$i(t,V)$ predicted by the NEURON-model.
`cdn_#_Raccess=0.txt`	Predicted $i(t,V)$ when $R_{access} = 0$.
`tDIpred_#.txt`	Predicted leak-subtracted currents.
`Ihold_DI_#.txt`	Predicted values of I_{hold} and the stationary change in I during the leak-subtraction pulse.
`cdn_blocked.txt`	Predicted $i(t,V)$ in the blocked condition when $R_{access} = 0$ (subtracted from $i(t,V)$ in control).

Table 4. Temporary files in the current folder.

Table 4a. NEURON-model output (input for `CorrectCoxVpulse`).

File name	**Description**
`tDIpred.txt`	Save your NEURON-model predictions for the leak-subtracted currents to this file. The format must conform to that of the supplied file `tDIpred.txt` in folder `CorrectCoxVpulseExample`.
`cdn.txt` (optional)	Save your NEURON-model predictions for the current density of interest to this file. The format must conform to that of supplied file `cdn.txt` in folder `CorrectCoxVpulseExample`.
`Ihold_DI.txt`	Estimates of the holding current (`Ihold(pA)`) and the stationary change in current observed on applying the leak-subtraction pulse (`DI(pA)`); see `Ihold_DI.txt` in folder `CorrectCoxVpulseExample`.
`flag.txt`	See main text.

Table 4b. `CorrectCoxVpulse` output (input for your NEURON model).

`parameter_file.txt`	See main text.
`cdn_paras.txt`	The current best estimate of the current-density parameters. Pass these data as a FUNCTION_TABLE to your .mod file (see the .hoc and .mod files in folder `CorrectCoxVpulseExample`).
`cdL_paras.txt`	E_L and g_L(passive leak: $i_L = g_L(V - E_L)$. For i_L methods `1` and `2` (see Table 1, footnote 2); otherwise i_L is user-specified (methods `0` and `3` in Table 1, footnote 2). Or load $i_L(V)$ from `cdL.txt`.
`cd.txt`	$i(t,V)$ estimated by `CorrectCoxVpulse` (for plotting)[1].
`cd_fit.txt`	Fitted $i(t,V)$ function values vs t at different V (for plotting)[1].

[1]For evaluating the fits of the functions specified in the function file to the estimates of $i(t,V)$. This seems difficult to achieve with `CorrectCoxVpulse` while the optimisation is in progress, so I leave it to the user to do the plotting if they wish in their own scripts.

no_space_clamp_errors()

Use this function to assess the effect of not correcting for space-clamp errors on the estimates of model kinetic parameters (see folder `NoSpaceClampErrorExample`):

```
result = cox.no_space_clamp_errors(string1)
```

where `result` returns the output directory and *`string1`* should be the name of a parameter file. This file can be a virtual copy of the one used by `CorrectCoxVpulse`, although most of the entries will not be used and one line is different: instead of "`Model_parameter_file_for_blocked_condition…`" we have

```
Subtract_from_I(t,V)_the_currents_in_file: sring2
```

where *`sring2`* should be `none` in the blocked condition. When analysing control data, *`sring2`* is the name of the file containing the leak-subtracted currents in the blocked condition (which will be subtracted from those in control). In either case $i(t,V)$ is estimated as $I(t,V)/A$, where I is the current (or difference thereof) and A is the membrane area of the patched compartment. Functions are fitted and parameters saved as usual, but no simulations are run.

fit_neuron()

The effect of not correcting for space-clamp errors can be further assessed with this function. Call the function with:

```
p = {'scale_cd': 0.5, 'scale_cd_blocked': 0.5}
result = cox.fit_neuron(string, p, fmt='%12.9f',
         weighted_fit = 'no')
```

where `string` is a parameter file name, `p` is a dictionary of starting values for the indicated adjustable parameters, `fmt` is a format string for output (default = `'%15.5e'`) and by default `weighted_fit = 'yes'` (with weights = 1/SD(I) for the individual traces; see Table 1).. The parameter file is just like the control parameter file used by `CorrectCoxVpulse` (e.g. `NoSpaceClampErrorExample/control_parameters_fit_neuron.txt`). `fit_neuron()` will then attempt to optimise the scaling factors for the current densities by running successive simulations of the NEURON model (for an example see folder `NoSpaceClampErrorExample`). For each simulation, as when using `CorrectCoxVpulse`, the NEURON model must generate files `flag.txt` and `tDIpred.txt`, where the latter contains predictions for the leak-subtracted currents in control. If the kinetic parameters used by the NEURON model are those determined with `no_space_clamp_errors()`, `fit_neuron()` can be used to assess the effects of ignoring space-clamp errors on NEURON-model predictions. At the end of the optimisation, a plot of predicted and observed leak-subtracted currents is produced. In addition a file `fit_neuron.txt` will be created in the output folder containing the estimated values of `scale_cd` and `scale_cd_blocked`. If you only wish to optimise one of these parameters

then the other should be omitted. In fact you can optimise arbitrary parameters just by creating an appropriate dictionary `p`. For each iteration, the current estimates of those parameters will be written to `parameter_file.txt` as name-value pairs (the names being equal to the `p` keys). As usual the first line of `parameter_file.txt` contains the parameter file name, but the second line is: `No_space-clamp_errors: #`, where `#` is the iteration number. If you wish to inspect the residuals they may be found in the `result` dictionary, together with other output from the *SciPy* function `optimize.least_squares` and the output folder name. **NOTE:** It is important that data are saved to `tDIpred.txt` with sufficient numerical precision (e.g. with a format specifier of `"%25.16e"`), otherwise the optimiser may not be able to detect changes in the error sum of squares (which will result in parameters not being altered and termination of the fit). You might also need to change some of the arguments of `optimize.least_squares`. This can be done when calling `fit_neuron`, e.g.

```
result = cox.fit_neuron(string, p, jac='2-point')
```

which changes the value of `jac` from `'3-point'`(the `fit_neuron` default) to `'2-point'` (the `least_squares` default). Other `least_squares` defaults are retained by `fit_neuron`. See the `optimize.least_squares` documentation for further details.

Before using least-squares optimisation (which can take a long time) it might be an idea to test some trial values of a single parameter while holding any others constant. This can be done by providing a list `pv` = [*maxval*, *nvals*], where *maxval* is the maximum trial value (the minimum being taken as `p[0]`, which should be the only element of `p`) and *nvals* is the number of trial values. For example,

```
p = {paraname: 0.4}; pv = [0.6,3]
result = cox.fit_neuron(string, p, pv = pv)
```

will try values of 0.4, 0.5 and 0.6 of the parameter *paraname*. SSE values for each trial parameter value are saved to the file *paraname*`_SSE.txt`.

Technical notes

As discussed in Murphy *et al*. (2024a,b), the *Cox method with correction* for correcting space-clamp errors involves the solution of a nonlinear ODE in the current density, *i* [Eq. (1) in Murphy *et al*. (2024b)]. Specification of the ODE requires estimation of a fictitious current (*I**), which is determined by linear interpolation (or extrapolation if this s not possible). The interpolation method is set by the parameter `I*(V)_interpolation` in the parameter file. The available choices are `simple`, `standard` and `modified`. The algorithms are taken from Gerald and Wheatly (1984). In principle `standard` and `modified` should be more robust than `simple`. `modified` is faster than `standard` and so is the recommended method. The initial value $i = 0$ is taken at $V = V_{rest}$. Eq. (1) is solved for $i(V)$ using function `scipy.integrate.solve_ivp` from the *SciPy* Python package. To facilitate the numerical solution, the $I(V)$ and $I^*(V)$ data are smoothed according to the setting of the parameter `poly` in the parameter file (Table 1, footnote 4). Polynomial smoothing worked well for *V* ramps

(Murphy *et al*., 2024a) but seems less suited to the analysis of *V* test-pulse data (Murphy *et al*., 2024b). So either cross validation (`cv`) or generalised cross validation (`gcv`) are the recommended methods, although splicing together two polynomials might be useful in difficult cases. Values of V_{rest} are estimated as the roots of the fitted smoothing functions, obtained with the function `numpy.roots` for the polynomials or method `root` for `cv` and `gcv`. To use `cv` or `gcv` you must install the statistical software *R* and the Python module `rpy2`, since smoothing is effected with the *R* function `smooth.spline`. I have used `rpy2` successfully with Python 3.7.6. For Linux: https://rpy2.github.io/doc/latest/html/index.html. For Windows: https://anaconda.org/r/rpy2. *Anaconda* offers the following advice in the latter case:

Setting up rpy2 on Windows

Problem
If you install rpy2 on Windows and try to use R objects or R magic commands you may see this error message:

The program can't start because libbz2-1.dll is missing from your computer. Try reinstalling the program to fix this problem.

Solution
To resolve this issue, follow these steps:

1. If you have not installed rpy2, install from the R channel on Anaconda.org:

conda install -c r rpy2

2. After the installation is complete, add the following directory paths to your PATH environment variable. This section can be found here:
Computer -> Properties -> Advanced System Settings -> Environment Variables

If Anaconda was installed for just you, edit the PATH variable for your user. If you have installed for all users, edit the system PATH which may require admin privileges.
Add the two following lines to the end of what is already there. Be sure the full path should matches your installation path and Anaconda directory name:

C:\Users\username\Anaconda2\Library\mingw-w64\lib;
C:\Users\username\Anaconda2\Library\mingw-w64\bin

Your PATH should now have the following line for Anaconda and rpy2:

C:\Users\username\Anaconda2;C:\Users\username\Anaconda2\Scripts;C:\Users\username\Anaconda2\Library\bin;C:\Users\username\Anaconda2\Library\mingw-w64\lib;C:\Users\username\Anaconda2\Library\mingw-w64\bin

Close any open command prompts or python consoles, and relaunch them for the PATH to take effect.

For alternative Windows installations check out https://jianghaochu.github.io/how-to-install-rpy2-in-windows-10.html and http://joonro.github.io/blog/posts/install-rpy2-windows-10/.

Acknowledgement

The morphology of the neuron model in mossy_fiber_morphology.hoc was based on *hoc* code written by Konstantin Ostroumov (https://prabook.com/web/konstantin.ostroumov/476287). This software was developed in the laboratory of Prof. Johan F. Storm at the University of Oslo with financial support from the European Union's Horizon 2020 research and innovation programme under grant agreement 7202070 (Human Brain Project; HBP, SGA3) and the Norwegian Research Council (NFR/SFF).

References

Carnevale, N. T. and Hines M. L. (2009). *The NEURON Book*, Cambridge University Press.

Cox S. J. (2008). Direct correction of non-space-clamped currents via Cole's theorem. *Journal of Neuroscience Methods* **169**(2): 366-373.

Gerald C.F. and Wheatly P.O. (1984). *Applied Numerical Analysis*. Addison-Wesley.

Kutner M.H., Nachtsheim C.J., Newman E.L. and Li W. (2004). *Applied Linear Statistical Models*. McGraw-Hill/Irwin

Murphy, R., H. Alle, J. R. P. Geiger and J. F. Storm (2024a). Estimation of persistent sodium-current density in hippocampal mossy fiber axons: correction of space-clamp errors. *Journal of Physiology*: http://doi.org/10.1113/JP284657.

Murphy, R., H. Alle, J. R. P. Geiger and J. F. Storm (2024b). The Kv7/KCNQ/M current in hippocampal mossy fiber axons: I correction of space-clamp errors in voltage-clamp experiments. *Journal of Physiology*: submitted.

Sakmann, B. and E. Neher, Eds. (1995). *Single-Channel Recording*. New York, NY, Springer.