

Python code

All the following files can also be downloaded from: <https://github.com/kmemmer/Pythonfiles-Analysing-the-performance-of-KHFAC-nerve-block-stimulation-parameters>

main.py

```
1 #####
2 # This main file contains an overview of functions that were
3 # run to do simulations in the thesis of Koen Emmer
4 # Uncomment functions to run them
5 # For questions, contact me via LinkedIn
6 # https://www.linkedin.com/in/koenemmer/
7 #####
8
9 #####
10 # Import Modules
11 #####
12 # Import Python libraries
13 # Import for BrainFrame
14 # import matplotlib
15
16 from configoptions import *
17 from reportplots import *
18 from datasetproposedmethod import *
19 from datasetsformodeldesign import *
20 from monopolar_simulations import *
21 from bipolar_simulations import *
22
23 #####
24 # Build datasets for
25 #####
26 # runwave("ClassicModel", "square", "complete", "saveall")
27 # runwave("ClassicModel", "sine", "complete", "saveall")
28 # runsquarewaveCB("ClassicModel", "complete", "saveall")
29 # runwave("ProposedModel", "square", "complete", "saveall")
30 # runwave("ProposedModel", "sine", "complete", "saveall")
31 # runsquarewaveCB("ProposedModel", "complete", "saveall")
32
33 #####
34 # Find blocking amplitudes via proposed method
35 #####
36 # squarewaveresults()
37 # sineresults()
38 # asymmetricalresults()
39
40 #####
41 # Plot single KHFAC simulation
42 #####
43 # plot_square_classic(tstop=20)
44 # plot_sine_classic()
45 # plot_asymmetric_classic(f=10, a=0.2, tstop = 50.0, chargebalance=0.1, intrinsic=1)
46 # plot_square_proposed(f=10, a=0.47)
47 # plot_sine_proposed(f=10, a=0.52)
48 # plot_asymmetric_classic(f=10, a=0.55, tstop = 51.0, chargebalance=0.8, intrinsic=1)
49 # plot_asymmetric_classic(f=10, a=0.6, tstop = 51.0, chargebalance=0.9, intrinsic=1)
50 # plot_asymmetric_proposed(f=10, a=0.19, chargebalance=0.1)
51 # plot_asymmetric_proposed(f=10, a=0.2, chargebalance=0.1)
52 # plot_asymmetric_proposed(f=10, a=0.6, chargebalance=0.9)
53 # plot_sine_proposed(10,0.52)
54 # plot_sine_proposed(10,0.45)
55 # plot_sine_proposed(40,0.66)
56 # plot_sine_proposed(40,0.60)
57 # plot_square_proposed_bipolar(10,1.0)
58
59
60 #####
61 # Plot and test waveforms (uncomment one of the lines together with the last two lines to produce a
62 # picture
63 #####
```

```
63 # t_signal, i_signal = sinewave(0, 0.1, 10, 1, 1000)
64 # t_signal, i_signal = chargebalanced_asymmetrical(0, 2.0, 1, 0.06, 0.8, 0.1, 0.1)
65 # t_signal, i_signal = chargebalanced_asymmetrical(0, 0.1, 10, 0.05, 0.9)
66 # t_signal, i_signal = trianglewave(0, 0.1, 10, 1)
67 # t_signal, i_signal = squarewave_ipd(0, 0.2, 10, 1, 0.1, 0.1)
68 # t_signal, i_signal = squarewave(0, 0.2, 10, 1)
69 # t_signal, i_signal = stepwave(0, 0.1, 10, 2, 3)
70 # t_signal, i_signal = stepwave_sine(0, 0.1, 10, 1, 20)
71 # plot_signal(plt, t_signal, i_signal)
72 # plt.show()
73
74 #####
75 # Monopolar simulations -> create result files
76 #####
77 # monopolar_trianglewave_results()
78 # monopolar_sinewave_results()
79 # monopolar_squarewave_results()
80 # monopolar_step("triangular")
81 # monopolar_step("sine")
82 # monopolar_asymmetricalwave_results()
83 # squarewave_ipd_results()
84 # monopolar_squarewave_realdistance_results()
85 # squarewave_ipd_results("validation")
86
87 #####
88 # Bipolar simulations -> create result files
89 #####
90 # bipolar_squarewave_ETAdistance_IECdistance_parallel_results()
91 # bipolar_squarewave_ETAdistance_IECdistance_parallel_results(zoom=True)
92 # bipolar_squarewave_IECdistance_results('parallel')
93 # bipolar_squarewave_IECdistance_results('perpendicular')
94 # bipolar_squarewave_orientation_results('x')
95 # bipolar_squarewave_orientation_results('z')
96
97 #####
98 # Build report plots
99 #####
100 ## Background chapter
101 # plotAP()
102 # plotKHFAcdemogates()
103 # plothminftau()
104
105 ## Method chapter
106 # plot_2D_blockdescription()
107 # plot_3D_blockdescription()
108 # plot_gates_proposedmethodsetup()
109 # plot_asymmetrical_waveform()
110 # justificationplots("sine")
111 # justificationplots("square")
112 # justificationplots("squareCB")
113
114 ## Results Chapter
115 # Monopolar plots
116 # plot_monopolar("monopolar_basic_waveforms", "amplitude")
117 # plot_monopolar("monopolar_basic_waveforms", "cpp")
118 # plot_monopolar("stepfunctions", "amplitude")
119 # plot_monopolar("stepfunctions", "cpp")
120 # plot_monopolar("asymmetrical", "amplitude", "colour")
121 # plot_monopolar("asymmetrical", "cpp", "colour")
122 # plot_monopolar("asymmetrical", "amplitude", "3D")
123 # plot_monopolar("asymmetrical", "cpp", "3D")
124 # plot_monopolar("asymmetrical", "amplitude", "singlefrequency", 10)
125 # plot_monopolar("squarewave_ipd", "amplitude", "lines")
126 # plot_monopolar("squarewave_ipd", "cpp", "lines")
127 # plot_monopolar("squarewave_ipd", "amplitude", "colour")
128 # plot_monopolar("squarewave_ipd", "cpp", "colour")
129 # plot_monopolar("squarewave_ipd", "amplitude", "3D")
130 # plot_monopolar("squarewave_ipd", "cpp", "3D")
131 # plot_monopolar("squarewave_ipd_validation", "amplitude", "lines")
132 # plot_monopolar("squarewave_ipd_validation", "cpp", "lines")
133 # plot_monopolar("squarewave_ipd_validation", "amplitude", "colour")
134 # plot_monopolar("squarewave_ipd_validation", "cpp", "colour")
135 # plot_monopolar("realdistance", "amplitude")
136 # plot_monopolar("realdistance", "cpp")
137 # Bipolar plots
138 # plot_bipolar("ETA_IEC_parallel", "amplitude", "lines")
139 # plot_bipolar("ETA_IEC_parallel", "cpp", "lines")
140 # plot_bipolar("ETA_IEC_parallel", "amplitude", "colour")
```

```

141 # plot_bipolar("ETA_IEC_parallel", "cpp", "colour")
142 # plot_bipolar("ETA_IEC_parallel", "amplitude", "3D")
143 # plot_bipolar("ETA_IEC_parallel", "cpp", "3D")
144 # plot_bipolar("ETA_IEC_parallel", "amplitude", "optimal")
145 # plot_bipolar("ETA_IEC_parallel", "amplitude", "lines", zoom=True)
146 # plot_bipolar("ETA_IEC_parallel", "cpp", "lines", zoom=True)
147 # plot_bipolar("ETA_IEC_parallel", "amplitude", "colour", zoom=True)
148 # plot_bipolar("ETA_IEC_parallel", "cpp", "colour", zoom=True)
149 # plot_bipolar("ETA_IEC_parallel", "amplitude", "3D", zoom=True)
150 # plot_bipolar("ETA_IEC_parallel", "cpp", "3D", zoom=True)
151 # plot_bipolar("ETA_IEC_parallel", "amplitude", "optimal", zoom=True)
152 # plot_bipolar("IEC_parallelperpendicular", "amplitude")
153 # plot_bipolar("IEC_parallelperpendicular", "cpp")
154 # plot_bipolar("orientation", "amplitude")
155
156 ## Discussion plots
157 # plot_asymmetrical_bipolar(0.001)
158 # plot_ipdbasicimplementation(0.25,0.25)
159 # plot_ipdbasicimplementation(0,0.5)
160 # plot_ipdcompleximplementation()
161 # plot_ipd_validation(0.4,0,0)
162
163 #####
164 ## Build presentation plots
165 #####
166 # buildKHFAcGif()
167
168
169 quitNeuron()

```

blockdetection.py

```

1 #####
2 # This file contains the functions that detect a successful block
3 #####
4 import numpy as np
5
6
7 #####
8 # This functions returns values that can be used to determine
9 # the thresholds that categorize a block as successful or
10 # non-successful
11 #####
12 def conductionBlock (rec, par, node):
13     dt = par['dt']
14     tstart = int((par['HFSdelay'] + par['tonset']) / dt) # Start measuring 1 ms after initiating the
15     ↪ KHFAC signal, divide by dt to get listindex (2)
16
17     gatevalues = rec['gates']
18     # Neuron hocObjects need to be converted to arrays via np.array(!)
19     blockingnode_h = np.array(gatevalues['h_node' + str(node)]) # The node where the block occurs (a
20     ↪ virtual anode of the HFS node)
21
22     h_max = np.max(blockingnode_h[tstart:])
23     h_min = np.min(blockingnode_h[tstart:])
24
25     voltages = rec['voltage']
26     blockingnode_voltage = np.array(voltages['v_node' + str(node)])
27     V_max = np.max(blockingnode_voltage[tstart:])
28     V_min = np.min(blockingnode_voltage[tstart:])
29
30     blockingnode_m = np.array(gatevalues['m_node' + str(node)])
31     m_max = np.max(blockingnode_m[tstart:])
32     m_min = np.min(blockingnode_m[tstart:])
33
34     currents = rec['current']
35     blockingnode_h_inf = np.array(currents['h_inf_node' + str(node)])
36
37     blockingnode_h_dist = [a - b for a, b in zip(blockingnode_h_inf, blockingnode_h)]
38     h_dist_max = np.max(blockingnode_h_dist)
39     h_dist_min = np.min(blockingnode_h_dist)
40
41
42     return h_max, h_min, V_max, V_min, m_max, m_min, h_dist_max, h_dist_min

```

```

43
44
45 #####
46 # Proposed block detection method
47 #####
48 def is_blocked(rec, par):
49     dt = par['dt']
50     tstart = int((par['HFSdelay'] + par['tonset']) / dt)
51
52     gatevalues = rec['gates']
53     blocking_anode_hmax = np.max(np.array(gatevalues['h_node' + str(par['HFSreferenceNode']) +
54     ↪ 1]))[tstart:] # The node where the block occurs (a virtual anode of the HFS node)
55
56     voltages = rec['voltage']
57     blocking_node_Vmax = np.max(np.array(voltages['v_node' + str(par['HFSreferenceNode'])])[tstart:])
58     blocking_node_Vmin = np.min(np.array(voltages['v_node' + str(par['HFSreferenceNode'])])[tstart:])
59
60     blocking_anode_Vmax = np.max(np.array(voltages['v_node' + str(par['HFSreferenceNode']+1)])[tstart:])
61     blocking_anode_Vmin = np.min(np.array(voltages['v_node' + str(par['HFSreferenceNode']) +
62     ↪ 1]))[tstart:]
63
64     if blocking_anode_hmax < 0.04 and (blocking_anode_Vmax > -22 or blocking_anode_Vmin > -51.5 or
65     ↪ blocking_node_Vmin < -90):
66         return True
67     else:
68         return False
69
70 #####
71 # Classic block detection method as %
72 #####
73 def bloqueffectiveness(rec, par):
74
75     spikesfirstnode = np.array(rec['spiketimes']['spk' + str(0)])
76     # spikesfirstnode = (par['tstop'] - par['HFSdelay']) / (par['intrinsicTON'] + par['intrinsicTOFF'])
77     spikeslastnode = np.array(rec['spiketimes']['spk' + str(int(par['axonnodes']-1))])
78
79     effectiveness = min(1, max(1.0 - float(len(spikeslastnode[(spikeslastnode > 10.0)])) /
80     ↪ float(len(spikesfirstnode[(spikesfirstnode > 10.0)])), 0))
81     return effectiveness

```

monopolar_simulations.py

```

1 #####
2 # Monopolar simulation functions
3 #####
4 import csv
5 from configoptions import *
6 import time
7 import datetime
8 from MRG_Model import * # Interface with Neuron MRG Model
9 from blockdetection import *
10 from waveforms import * # Waveform functions for Neuron
11 from plottingfunctions_results import *
12 from simulationfunctions_proposedmodel import *
13
14 import os
15 if not os.path.exists('results_monopolar'):
16     os.makedirs('results_monopolar') # Ensure that brainframe makes justification folder
17 if not os.path.exists('reportplots'):
18     os.makedirs('reportplots')
19 if not os.path.exists('reportplots/monopolar_results'):
20     os.makedirs('reportplots/monopolar_results')
21
22 #####
23 # Build dataset for square wave with interphase delay tested with Proposed Model
24 #####
25 def squarewave_ipd_results(type="normal"):
26     print("--- RUN MONOPOLAR SIMULATION SQUAREWAVE WITH INTERPHASE DELAYS ---")
27     config = 'ProposedModel'
28     if type == "validation":
29         config = 'ClassicModel'
30     par, recpar = selectConfig(config)
31     frequency = 10
32     par['HFSfrequency'] = frequency
33     ipd_anodal_list = []

```

```

34     for i in range(20):
35         ipd_anodal_list.append(round(0.0 + 0.05 * i, 2))
36     if type == "validation":
37         ipd_anodal_list = []
38         for i in range(5):
39             ipd_anodal_list.append(round(0.0 + 0.20 * i, 2))
40     ipd_cathodal_list = ipd_anodal_list
41     starttime = time.time()
42     amplitudelist = []
43     for i in range(100):
44         amplitudelist.append(round(0.1 + 0.01 * i, 2))
45     if type == "validation":
46         amplitudelist = []
47         for i in range(80):
48             amplitudelist.append(round(0.35 + 0.01 * i, 2))
49     createMRGaxon(par)
50     result = {}
51     for ipd1 in ipd_anodal_list:
52         result[ipd1] = {}
53         for ipd2 in ipd_cathodal_list:
54             result[ipd1][ipd2] = {}
55             for a in amplitudelist:
56                 rec = recordMRGaxon(recpar)
57                 par['HFSamp'] = a
58                 if ipd1+ipd2 >= 1.0:
59                     break
60                 t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
61                 ↪ par['HFSamp'], ipd1, ipd2)
62                 updateMRGaxon(par)
63                 print("--- RUN KHfAC-SIGNAL (%s kHz, %s mA, %s ipd1, %s ipd2, type = %s) ---" %
64                 ↪ (frequency, a, ipd1, ipd2, type))
65                 runMRGaxon(rec, t_signal, i_signal)
66                 block = False
67                 if type != "validation":
68                     block = is_blocked(rec,par)
69                 else:
70                     if blockeffectiveness(rec,par) >= 0.9:
71                         block = True
72                 if block:
73                     cpp = (1.0-ipd1-ipd2)/(2*(1000*frequency))*a
74                     result[ipd1][ipd2]["frequency"] = frequency
75                     result[ipd1][ipd2]["ipd1"] = ipd1
76                     result[ipd1][ipd2]["ipd2"] = ipd2
77                     result[ipd1][ipd2]["amplitude"] = a
78                     result[ipd1][ipd2]["chargeperphase"] = cpp
79                     break
80     totaltime = time.time() - starttime
81     print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
82     filename = 'results_monopolar/results_squarewave_ipd_proposedmethod'
83     if type == "validation":
84         filename = 'results_monopolar/results_squarewave_ipd_validation'
85     np.save(filename, result)
86
87 #####
88 # Build dataset for step wave triangular shape
89 #####
90 def monopolar_step(shape):
91     print("--- RUN MONOPOLAR SIMULATION STEPWAVE (" + shape + ") WITH INCREASING STEPSIZES---")
92     config = 'ProposedModel'
93     par, recpar = selectConfig(config)
94     frequency = 10
95     par['HFSfrequency'] = frequency
96     step_list = []
97     for i in range(1,51):
98         step_list.append(i)
99     starttime = time.time()
100    amplitudelist = []
101    for i in range(200):
102        amplitudelist.append(round(0.1 + 0.01 * i, 2))
103    createMRGaxon(par)
104    result = {}
105    for steps in step_list:
106        result[steps] = {}
107        for a in amplitudelist:
108            rec = recordMRGaxon(recpar)
109            par['HFSamp'] = a
110            if shape == "triangular":

```

```

110         t_signal, i_signal = stepwave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
111         ↪ par['HFSamp'], steps)
112     elif shape == "sine":
113         t_signal, i_signal = stepwave_sine(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
114         ↪ par['HFSamp'], steps)
115     else:
116         print "Wrong shape selected!"
117         break
118     updateMRGaxon(par)
119     print("--- RUN KHFAc-SIGNAL (%s kHz, %s mA, %s steps (triangular)) ---" % (frequency, a,
120     ↪ steps))
121     runMRGaxon(rec, t_signal, i_signal)
122     if is_blocked(rec, par):
123         cpp = 0.0
124         if shape == "triangular":
125             stepsize = a / steps
126             for i in range(steps+1):
127                 cpp += (i * stepsize / (4 * frequency * steps))
128             for i in range(steps):
129                 cpp += (i * stepsize / (4 * frequency * steps))
130         elif shape == "sine":
131             for i in range(steps*2):
132                 cpp += (a * np.sin(2 * np.pi * i / (steps * 4)) / (4 * frequency * steps))
133         result[steps]["frequency"] = frequency
134         result[steps]["numberofsteps"] = steps
135         result[steps]["amplitude"] = a
136         result[steps]["chargeperphase"] = cpp
137         break
138     totaltime = time.time() - starttime
139     print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
140     filename = 'results_monopolar/results_monopolar_stepwave_' + shape
141     np.save(filename, result)
142
143     #####
144     # Build dataset for triangular wave tested with Proposed Model
145     #####
146     def monopolar_triangularwave_results():
147         frequencylist = [3]
148         for i in range(19):
149             frequencylist.append((i + 2) * 2)
150         result = find_blockingamplitudes_Freq(frequencylist, "triangular")
151         filename = 'results_monopolar/results_monopolar_triangularwave'
152         np.save(filename, result)
153
154     #####
155     # Build dataset for square wave tested with Proposed Model
156     #####
157     def monopolar_squarewave_results():
158         frequencylist = [3]
159         for i in range(19):
160             frequencylist.append((i + 2) * 2)
161         result = find_blockingamplitudes_Freq(frequencylist, "square")
162         filename = 'results_monopolar/results_monopolar_squarewave'
163         np.save(filename, result)
164
165     #####
166     # Build dataset for sine wave tested with Proposed Model
167     #####
168     def monopolar_sinewave_results():
169         frequencylist = [3]
170         for i in range(19):
171             frequencylist.append((i + 2) * 2)
172         result = find_blockingamplitudes_Freq(frequencylist, "sine")
173         filename = 'results_monopolar/results_monopolar_sinewave'
174         np.save(filename, result)
175
176     #####
177     # Build dataset for assymmetricalwave tested with Proposed Model
178     #####
179     def monopolar_assymmetricalwave_results():
180         frequencylist = [3]
181         for i in range(19):
182             frequencylist.append((i + 2) * 2)
183         chargebalancelist = []

```

```

185     for i in range(18):
186         chargebalancelist.append(round(0.1 + 0.05*i, 2))
187     result = find_blockingamplitudes_CB(frequencylist, chargebalancelist)
188     filename = 'results_monopolar/results_monopolar_assymmetricalwave_allfrequencies'
189     np.save(filename, result)
190
191
192 #####
193 # Build dataset for assymmetricalwave tested with Proposed Model
194 #####
195 def monopolar_squarewave_realdistance_results():
196     print("--- RUN MONOPOLAR SIMULATION SQUAREWAVE WITH CHANGE OVER REAL PERPENDICULAR DISTANCE ---")
197     config = 'ProposedModel'
198     par, recpar = selectConfig(config)
199     frequency = 10
200     par['HFSfrequency'] = frequency
201     perpdistance_list = []
202     # Max nervediometer is 4.67+1.17 = 5,84. Axondiameter is 10micrometre, range is 10,585. Steps of 10
203     ~ micrometre
204     for i in range(100, 5850, 10):
205         perpdistance_list.append(i)
206     starttime = time.time()
207     amplitudelist = []
208     for i in range(200):
209         amplitudelist.append(round(0.01 * i, 2))
210     for i in range(280):
211         amplitudelist.append(round(2.0 + 0.1 * i, 2))
212     createMRGaxon(par)
213     result = {}
214     count = 0
215     for dist in perpdistance_list:
216         result[dist] = {}
217         par['HFSz'] = dist
218         for a in amplitudelist[count:]:
219             rec = recordMRGaxon(recpar)
220             par['HFSamp'] = a
221             t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
222             ~ par['HFSamp'])
223             updateMRGaxon(par)
224             print("--- RUN KHFAc-SIGNAL (%s kHz, %s mA, %s perpdistance) ---" % (frequency, a, dist))
225             runMRGaxon(rec, t_signal, i_signal)
226             if is_blocked(rec, par):
227                 cpp = a / (2*frequency)
228                 result[dist]["frequency"] = frequency
229                 result[dist]["perpendiculardistance"] = dist
230                 result[dist]["amplitude"] = a
231                 result[dist]["chargeperphase"] = cpp
232                 break
233             count += 1
234     totaltime = time.time() - starttime
235     print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
236     filename = 'results_monopolar/results_monopolar_squarewave_realdistance'
237     np.save(filename, result)
238
239 #####
240 # Plot results of square wave with interphase delay
241 #####
242 def plot_monopolar(type, restype="amplitude", plottype="lines", plotfreq = 10):
243     if type == "squarewave_ipd":
244         result = np.load("results_monopolar/results_squarewave_ipd_proposedmethod.npy",
245         ~ encoding="ASCII").item()
246         plot_results_monopolar_ipd(result, restype, plottype)
247         filename = "results_squarewave_ipd_" + plottype + "_" + restype
248     if type == "squarewave_ipd_validation":
249         result = np.load("results_monopolar/results_squarewave_ipd_validation.npy",
250         ~ encoding="ASCII").item()
251         plot_results_monopolar_ipd(result, restype, plottype, validation=True)
252         filename = "results_squarewave_ipd_validation_" + plottype + "_" + restype
253     if type == "monopolar_basic_waveforms":
254         result_square = np.load("results_monopolar/results_monopolar_squarewave.npy").item()
255         result_sine = np.load("results_monopolar/results_monopolar_sinewave.npy").item()
256         result_triangular = np.load("results_monopolar/results_monopolar_triangularwave.npy").item()
257         plot_results_monopolar_basicwaveforms(result_square, result_sine, result_triangular,
258         ~ type=restype)
259         filename = "results_basicwaveforms_" + restype
260     if type == "stepfunctions":
261         result_stepsine = np.load("results_monopolar/results_monopolar_stepwave_sine.npy").item()

```

```

258     result_steptriangular =
259         ↪ np.load("results_monopolar/results_monopolar_stepwave_triangular.npy").item()
260     plt_results_monopolar_stepwaveforms(result_stepsine, result_steptriangular, type=restype)
261     filename = "results_stepfunctions_"+restype
262     if type == "realdistance":
263         result = np.load("results_monopolar/results_monopolar_squarewave_realdistance.npy").item()
264         plt_results_monopolar_realdistance(result, type=restype)
265         filename = "results_monopolar_squarewave_realdistance_"+restype
266     if type == "asymmetrical":
267         result =
268             ↪ np.load("results_monopolar/results_monopolar_asymmetricalwave_allfrequencies.npy").item()
269         plot_results_asymmetrical(result, restype, plottype, plotfreq)
270         filename_plottype = plottype
271         if plottype == "singlefrequency":
272             filename_plottype = plottype + "_" + str(plotfreq) + "kHz"
273         filename = "results_monopolar_asymmetricalwaves_"+filename_plottype+"_"+restype
274     plt.savefig("reportplots/monopolar_results/" + filename + ".pdf", format='pdf', dpi=1200)
275     plt.show()

```

bipolar_simulations.py

```

1  #####
2  # Bipolar simulation functions
3  #####
4  import csv
5  from configoptions import *
6  import time
7  import datetime
8  from MRG_Model import *           # Interface with Neuron MRG Model
9  from blockdetection import *
10 from waveforms import *           # Waveform functions for Neuron
11 from plottingfunctions_results import *
12 # from simulationfunctions_proposedmodel import *
13
14
15 import os
16 if not os.path.exists('results_bipolar'):
17     os.makedirs('results_bipolar')           # Ensure that brainframe makes justification folder
18 if not os.path.exists('reportplots'):
19     os.makedirs('reportplots')
20 if not os.path.exists('reportplots/bipolar_results'):
21     os.makedirs('reportplots/bipolar_results')
22
23 #####
24 # Build dataset bipolar square wave simulation, changing electrode to axon distance and inter electrode
25 ↪ contact distance
26 #####
27 def bipolar_squarewave_ETAdistance_IECdistance_parallel_results(zoom=False): #ETA = Electrode To Axon,
28     ↪ IEC = Inter Electrode Contact
29     print("--- RUN BIPOLAR SIMULATION SQUAREWAVE WITH CHANGING ELECTRODE TO AXON DISTANCE AND INTER
30         ↪ ELECTRODE CONTACT DISTANCE, ELECTRODE PARALLEL TO AXON ---")
31     config = 'ProposedModel_Bipolar'
32     par, recpar = selectConfig(config)
33     frequency = 10
34     par['HFSfrequency'] = frequency
35     perpdistance_list = []
36     for i in range(1000, 6001, 500):
37         perpdistance_list.append(i)
38
39     poledistance_list = []
40     count = {}
41     if zoom:
42         for i in range(1000, 10000, 100):
43             poledistance_list.append(i)
44     else:
45         for i in range(1000, 60001, 1000):
46             poledistance_list.append(i)
47     for i in poledistance_list:
48         count[i] = 0
49     amplitudelist = []
50     for i in range(200):
51         amplitudelist.append(round(0.01 * i, 2))
52     for i in range(280):
53         amplitudelist.append(round(2.0 + 0.1 * i, 2))
54     starttime = time.time()
55     createMRGaxon(par)

```



```

53 result = {}
54 for ETAdist in perpdistance_list:
55     result[ETAdist] = {}
56     par['HFSz'] = ETAdist
57     par['HFSz2'] = ETAdist
58     for IECdist in poledistance_list:
59         result[ETAdist][IECdist] = {}
60         par['HFSy2'] = -IECdist
61         for a in amplitudelist[count[IECdist]:]:
62             rec = recordMRGaxon(recpar)
63             par['HFSamp'] = a
64             t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
65             ↪ par['HFSamp'])
66             updateMRGaxon(par)
67             print("--- RUN KHFac-SIGNAL (%s kHz, %s mA, %s um el-to-ax distance, %s um
68             ↪ interelectrodecontact distance) ---" % (frequency, a, ETAdist, IECdist))
69             runMRGaxon(rec, t_signal, i_signal)
70             if is_blocked(rec, par):
71                 cpp = (1.0)/(2*(1000*frequency))*a
72                 result[ETAdist][IECdist]["frequency"] = frequency
73                 result[ETAdist][IECdist]["electrodetoaxondistance"] = ETAdist
74                 result[ETAdist][IECdist]["interelectrodecontactdistance"] = IECdist
75                 result[ETAdist][IECdist]["amplitude"] = a
76                 result[ETAdist][IECdist]["chargeperphase"] = cpp
77                 count[IECdist] = max(0, count[IECdist] - 10)
78                 break
79             count[IECdist] += 1
80 totaltime = time.time() - starttime
81 print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
82 filename_end = ""
83 if zoom:
84     filename_end = "_zoom"
85 filename = 'results_bipolar/results_squarewave_ETAdist_IECdist_parallel'+filename_end
86 np.save(filename, result)
87
88 #####
89 # Build dataset for square wave with interphase delay tested with Proposed Model
90 #####
91 def bipolar_squarewave_IECdistance_results(mode): #ETA = Electrode To Axon, IEC = Inter Electrode Contact
92     print("--- RUN BIPOLAR SIMULATION SQUAREWAVE WITH CHANGING ELECTRODE TO AXON DISTANCE AND INTER
93     ↪ ELECTRODE CONTACT DISTANCE, ELECTRODE %s TO AXON ---" % mode)
94     config = 'ProposedModel_Bipolar'
95     par, recpar = selectConfig(config)
96     frequency = 10
97     par['HFSfrequency'] = frequency
98     poledistance_list = []
99     count = {}
100     for i in range(1000, 60001, 500):
101         poledistance_list.append(i)
102     amplitudelist = []
103     for i in range(200):
104         amplitudelist.append(round(0.01 * i, 2))
105     starttime = time.time()
106     createMRGaxon(par)
107     result = {}
108     for dist in poledistance_list:
109         result[dist] = {}
110         if mode == 'parallel':
111             par['HFSy2'] = -dist
112         elif mode == 'perpendicular':
113             par['HFSx2'] = -dist
114         for a in amplitudelist:
115             rec = recordMRGaxon(recpar)
116             par['HFSamp'] = a
117             t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
118             ↪ par['HFSamp'])
119             updateMRGaxon(par)
120             print("--- RUN KHFac-SIGNAL (%s kHz, %s mA, %s um interelectrodecontact distance) ---" %
121             ↪ (frequency, a, dist))
122             runMRGaxon(rec, t_signal, i_signal)
123             if is_blocked(rec, par):
124                 cpp = (1.0)/(2*(1000*frequency))*a
125                 result[dist]["frequency"] = frequency
126                 result[dist]["interelectrodecontactdistance"] = dist
127                 result[dist]["amplitude"] = a
128                 result[dist]["chargeperphase"] = cpp
129                 break

```

```

126     totaltime = time.time() - starttime
127     print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
128     filename = 'results_bipolar/results_squarewave_IECdist_' + mode
129     np.save(filename, result)
130
131
132     #####
133     # Build dataset for square wave with bipolar electrode of changing orientation
134     #####
135     def bipolar_squarewave_orientation_results(mode = 'x'): #ETA = Electrode To Axon, IEC = Inter Electrode
136         ← Contact
137         print("--- RUN BIPOLAR SIMULATION SQUAREWAVE WITH CHANGING ELECTRODE ORIENTATION OVER %s-AXIS---" %
138             ← mode)
139         config = 'ProposedModel_Bipolar'
140         par, recpar = selectConfig(config)
141         frequency = 10
142         par['HFSfrequency'] = frequency
143         orientation_list = []
144         count = {}
145         poledistance_list = []
146         for i in range(45, 91, 5):
147             orientation_list.append(i)
148         for i in range(1000, 5001, 200):
149             poledistance_list.append(i)
150         amplitudelist = []
151         for i in range(20, 200):
152             amplitudelist.append(round(0.01 * i, 2))
153         starttime = time.time()
154         createMRGaxon(par)
155         result = {}
156         for angle in orientation_list:
157             result[angle] = {}
158             for dist in poledistance_list:
159                 result[angle][dist] = {}
160                 par['HFSy2'] = -np.cos(np.deg2rad(angle))*dist
161                 if mode == 'x':
162                     par['HFSx2'] = -np.sin(np.deg2rad(angle))*dist
163                 if mode == 'z':
164                     par['HFSz2'] = par['HFSz'] + np.sin(np.deg2rad(angle))*dist
165                     print par['HFSz2']
166                     print par['HFSy2']
167                     print par['HFSx2']
168                 for a in amplitudelist:
169                     rec = recordMRGaxon(recpar)
170                     par['HFSamp'] = a
171                     t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
172                         ← par['HFSamp'])
173                     updateMRGaxon(par)
174                     print("--- RUN KHfAC-SIGNAL (%s kHz, %s mA, %s degree angle, %s um interphase distance)
175                         ← ---" % (frequency, a, angle, dist))
176                     runMRGaxon(rec, t_signal, i_signal)
177                     if is_blocked(rec, par):
178                         cpp = (1.0)/(2*(1000*frequency))*a
179                         result[angle][dist]["frequency"] = frequency
180                         result[angle][dist]["electrodeangle"] = angle
181                         result[angle][dist]["interelectrodecontactdistance"] = dist
182                         result[angle][dist]["amplitude"] = a
183                         result[angle][dist]["chargeperphase"] = cpp
184                         break
185
186         totaltime = time.time() - starttime
187         print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
188         filename = 'results_bipolar/results_squarewave_orientation_' + mode
189         np.save(filename, result)
190
191     #####
192     # Plot results of square wave with interphase delay
193     #####
194     def plot_bipolar(type, restype="amplitude", plottype="lines", zoom=False):
195         if type == "ETA_IEC_parallel":
196             zoomtext = ""
197             if zoom:
198                 zoomtext = "_zoom"
199             result = np.load("results_bipolar/results_squarewave_ETAdist_IECdist_parallel" + zoomtext +
200                 ← ".npy", encoding="ASCII").item()
201             plot_results_bipolar ETA_IEC_parallel(result, restype, plottype)
202             filename = "results_bipolar ETA_IEC_parallel" + zoomtext + "_" + plottype + "_" + restype
203         elif type == "IEC_parallelperpendicular":

```

```

199     result_parallel = np.load("results_bipolar/results_squarewave_IECdist_parallel.npy",
200     ↪ encoding="ASCII").item()
201     result_perpendicular = np.load("results_bipolar/results_squarewave_IECdist_perpendicular.npy",
202     ↪ encoding="ASCII").item()
203     plot_results_bipolar_IEC_parallelperp(result_parallel, result_perpendicular, restype)
204     filename = "results_bipolar_IEC_parvsperp_"+restype
205     elif type == "orientation":
206         result_x = np.load("results_bipolar/results_squarewave_orientation_x.npy",
207         ↪ encoding="ASCII").item()
208         result_z = np.load("results_bipolar/results_squarewave_orientation_z.npy",
209         ↪ encoding="ASCII").item()
210         filename = "results_bipolar_orientation_"+restype
211         plot_results_bipolar_orientation(restype, result_x, result_z)
212     plt.savefig("reportplots/bipolar_results/" + filename + ".pdf", format='pdf', dpi=1200)
213     plt.show()

```

waveforms.py

```

1  #####
2  #####
3  # Pythoncode containing Neuron-vector descriptions of waveforms
4  #####
5  #####
6  import numpy as np
7  import matplotlib.pyplot as plt
8
9  #####
10 # Transforms a standard waveform into a signal that is ready to be used by Neuron
11 # t_start is when the signal should start
12 # t_stop is when the signal should end
13 # t_period is the time vector of the signal
14 # i_period is the current amplitude vector of the signal
15 # f is the frequency of the signal
16 # amplitude is the amplitude of the signal
17 # The outputs are:
18 # t_signal: the time vector of the signal
19 # i_signal: the amplitude vector of the signal
20 # Neuron combines the two to construct a signal,
21 # build up by having amplitude i_signal[x] at time t_signal[x]
22 #####
23 def KHfACblocksignal(t_start, t_stop, t_period, i_period, f, amplitude):
24     f = 1000*f # Translate from KHz to Hz
25     period = 1000.0/f #in milliseconds
26     t_period = [x * period/max(t_period) for x in t_period] #Neuron t is in ms
27     i_period = [x * amplitude for x in i_period]
28     timestep = max(t_period)
29     t_signal = []
30     i_signal = []
31     for i in range(0, int(t_stop/period)):
32         t_signal = np.concatenate((t_signal, t_period))
33         t_period = [x + timestep for x in t_period]
34         if i < int(t_start/period):
35             i_signal = np.concatenate((i_signal, [x * 0 for x in i_period]))
36         else:
37             i_signal = np.concatenate((i_signal, i_period))
38     return t_signal, i_signal
39
40
41 #####
42 # Creates a sampled sinewave-period for Neuron, consisting of:
43 # t_period: the time vector of one period
44 # i_period: the amplitude vector of one period
45 #####
46 def sinewaveperiod(samplesperperiod):
47     t_period = []
48     i_period = []
49     for i in range(0, samplesperperiod+1):
50         t_period = np.concatenate((t_period, i), axis=None)
51         i_period = np.concatenate((i_period, np.sin(2*np.pi*i/samplesperperiod)), axis=None)
52     return t_period, i_period
53
54
55 #####
56 # Creates a sampled sinewave signal for Neuron
57 #####
58 def sinewave(t_start, t_stop, f, amplitude, samplesperperiod):

```

```

59     t_period, i_period = sinwaveperiod(samplesperperiod)
60     return KHFAcblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)
61
62
63 #####
64 # Creates a sampled triangle wave signal for Neuron
65 #####
66 def trianglewave(t_start, t_stop, f, amplitude):
67     t_period = [0, 1, 2, 3, 4]
68     i_period = [0, 1, 0, -1, 0]
69     return KHFAcblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)
70
71
72 #####
73 # Creates a sampled square wave signal for Neuron
74 # ipd1 and ipd2 represent interphase delays, as a fraction of the waveform period (e.g. ipd1=0.1 will
75   ↳ create an interphase delay of one tenth of the signal period after the anodal pulse
76 #####
77 def squarewave(t_start, t_stop, f, amplitude, ipd1 = 0, ipd2 = 0):
78     period_length = 1.0
79     no_ipd_length = period_length - ipd1 - ipd2
80     t_period = [0, no_ipd_length / 2, no_ipd_length / 2, no_ipd_length / 2 + ipd1, no_ipd_length / 2 +
81   ↳ ipd1, no_ipd_length + ipd1, no_ipd_length + ipd1, period_length]
82     i_period = [1, 1, 0, 0, -1, -1, 0, 0]
83     return KHFAcblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)
84
85 #####
86 # Creates a sampled charge balanced asymmetrical square wave for Neuron
87 # chargeperphase in uC
88 # f in kHz
89 # t_start & t_stop in ms
90 #####
91 def chargebalanced_asymmetrical(t_start, t_stop, f, chargeperphase, ratioHi, ipd1 = 0, ipd2 = 0):
92     ratioLo = 1.0-ratioHi
93     period = 1.0
94     no_ipd_length = period - ipd1 - ipd2
95     T_anodal = no_ipd_length * ratioHi # period of anodal (positive) stimulation phase
96     T_cathodal = no_ipd_length * ratioLo # period of cathodal (negative) stimulation phase
97     A_anodal = 1.0 # Anodal pulse is reference amplitude
98     A_cathodal = -1* A_anodal*T_anodal/T_cathodal # Cathodal pulse scaled to match reference
99     ↳ amplitude
100     amplitude = f * chargeperphase / ratioHi # Real amplitude of anodal pulse is calculated as
101     ↳ multiplication
102     t_period = [0, T_anodal, T_anodal, T_anodal + ipd1, T_anodal + ipd1, T_anodal + ipd1 + T_cathodal,
103   ↳ T_anodal + ipd1 + T_cathodal, T_anodal + ipd1 + T_cathodal + ipd2]
104     i_period = [A_anodal, A_anodal, 0, 0, A_cathodal, A_cathodal, 0, 0]
105     return KHFAcblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)
106
107 #####
108 # Creates a sampled square wave signal with interphase delays for Neuron
109 # #####
110 # def squarewave_ip(t_start, t_stop, f, amplitude, ipd1, ipd2):
111 #     period_length = 1.0
112 #     no_ipd_length = period_length-ipd1-ipd2
113 #     t_period = [0, no_ipd_length/2, no_ipd_length/2, no_ipd_length/2+ipd1, no_ipd_length/2+ipd1,
114   ↳ no_ipd_length+ipd1, no_ipd_length+ipd1, period_length]
115 #     i_period = [1, 1, 0, 0, -1, -1, 0, 0]
116 #     return KHFAcblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)
117
118 #####
119 # Creates a stepwave for Neuron, consisting of:
120 # t_period: the time vector of one period
121 # i_period: the amplitude vector of one period
122 #####
123 def stepwave(t_start, t_stop, f, amplitude, steps_per_quartercycle): #Trianglewave
124     t_period = [0]
125     i_period = [0]
126     step_increment = 1.0/steps_per_quartercycle
127     for i in range(0, steps_per_quartercycle*4):
128         t_period = np.concatenate((t_period, [i,i+1]), axis=None)
129         i_period = np.concatenate((i_period, [i_period[-1] + step_increment, i_period[-1] +
130   ↳ step_increment]), axis = None)
131         if abs(abs(i_period[-1]) - 1.0) <= 1.0/100000000:
132             step_increment *= -1
133     return KHFAcblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)

```

```

130
131 def stepwave_sine(t_start, t_stop, f, amplitude, steps_per_quartercycle): #Sineshape
132     t_period = [0]
133     i_period = [0]
134     for i in range(0, steps_per_quartercycle*4):
135         t_period = np.concatenate((t_period, [i,i+1]), axis=None)
136         i_period = np.concatenate((i_period, [np.sin(2*np.pi*i/(steps_per_quartercycle*4)),
137         ↪ np.sin(2*np.pi*i/(steps_per_quartercycle*4))]), axis = None)
138     return KHFAChlocksignal(t_start, t_stop, t_period, i_period, f, amplitude)

```

configoptions.py

```

1  # This file stores different standard configurations
2  # By Koen Emmer
3  import numpy as np
4  import math
5
6  def selectConfig(config):
7      # STANDARDIZED PARAMETERS
8      par = {
9          # [MRGnode]
10         # ENVIRONMENTAL PARAMETERS
11         'celsius'          : 37.0,    # Temperature
12         'v_init'           : -80.0,   # Initial membrane voltage
13         'axonnodes'        : 51,      # Amount of Nodes of Ranvier on the axon
14         'fiberD'           : 10.0,    # Diameter of the axonfiber, choose from 5.7, 7.3,
15         ↪ 8.7, 10.0, 11.5, 12.8, 14.0, 15.0, 16.0
16         # SIMULATION PARAMETERS
17         'dt'               : 0.005,   # Simulation timestep (in ms)
18         'tstop'            : 10,      # Total simulation time (in ms)
19         # HFS PARAMETERS
20         'HFSreferenceNode' : 2,       # Node of Ranvier where KHFAc stimulation is applied
21         'HFSamp'           : 0.0,     # Amplitude of KHFAc waveform (mA)
22         'HFSfrequency'     : 10.0,    # Frequency of KHFAc waveform (kHz)
23         'HFSdelay'         : 5.0,     # Time when KHFAc waveform starts (ms)
24         'HFSx'             : 0.0,     # Location of the KHFAc electrode on the x-axis
25         'HFSy'             : 0.0,     # Location of the KHFAc electrode on the y-axis
26         'HFSz'             : 1000.0,  # Location of the KHFAc electrode on the z-axis
27         'tonset'           : 10,      # Duration of onset response
28         'bipolar'          : 0,       # Set to 1 for bipolar electrode
29         # PARAMETERS FOR INTRACELLULAR STIMULUS
30         'intrinsicStim'    : 0,       # Change to 1 to add intrinsic pulse(train)
31     }
32
33     if config=='singleAPdemo':
34         # Create single AP
35         par.update({
36             # ENVIRONMENTAL PARAMETERS
37             'celsius'          : 37.0,    # Temperature
38             'v_init'           : -80.0,   # Initial membrane voltage
39             'axonnodes'        : 21,     # Amount of Nodes of Ranvier on the axon
40             # SIMULATION PARAMETERS
41             'tstop'            : 10,      # Total simulation time (in ms)
42             # PARAMETERS FOR INTRACELLULAR STIMULUS
43             'intrinsicStim'    : 1,      # Change to 1 to add intrinsic pulse(train)
44             'intrinsicType'    : 0,      # 0 is single pulse, 1 is pulsetrain
45             'intrinsicNode'    : 10.0,   # Node where intracellular stimulus starts
46             'intrinsicAmp'     : 1.0,    # Amplitude of intracellular stimulus (nA)
47             'intrinsicDel'     : 3.0,    # Time when intracellular stimulus starts (ms)
48             'intrinsicDur'     : 1.0,    # Duration of intracellular stimulus (ms)
49             ↪ # ONLY SINGLE PULSE
50             'intrinsicTON'     : 1.0,    # Duration of ON phase intracellular stimulus (ms)
51             ↪ # ONLY PULSE TRAIN
52             'intrinsicTOFF'    : 1.0,    # Duration of OFF phase intracellular stimulus
53             ↪ (ms) # ONLY PULSE TRAIN
54             'intrinsicNum'     : None    # Number of intrinsic pulses (when doing a
55             ↪ pulsetrain)
56         })
57
58     elif config=='HFSdemo':
59         par.update({
60             # [MRGnode]
61             # ENVIRONMENTAL PARAMETERS
62             'axonnodes'        : 51.0,    # Amount of Nodes of Ranvier on the axon
63             # SIMULATION PARAMETERS
64             'tstop'            : 20,     # Total simulation time (in ms)

```

```

60     # HFS PARAMETERS
61     'HFSreferenceNode' : 25,      # Node of Ranvier where KHfAC stimulation is applied
        ↪ (First node is 0)
62     'HFSamp'          : 1.0,    # Amplitude of KHfAC waveform (mA)
63     'HFSfrequency'   : 10.0,   # Frequency of KHfAC waveform (kHz)
64     'HFSdelay'       : 2.0,    # Time when KHfAC waveform starts (ms)
65     'HFSx'           : 0.0,    # Location of the KHfAC electrode on the
        ↪ x-axis ()
66     'HFSy'           : 0.0,    # Location of the KHfAC electrode on the
        ↪ y-axis ()
67     'HFSz'           : 1000.0, # Location of the KHfAC electrode on the
        ↪ z-axis ()
68 })
69
70 elif config=="ClassicModel":
71     par.update({
72         # [MRGnode]
73         # ENVIRONMENTAL PARAMETERS
74         'axonnodes' : 51.0,    # Amount of Nodes of Ranvier on the axon
75         # SIMULATION PARAMETERS
76         'tstop'     : 51,      # Total simulation time (in ms)
77         'dt'        : 0.001,   # Simulation timestep (in ms)
78         # HFS PARAMETERS
79         'HFSreferenceNode' : 25, # Node of Ranvier where KHfAC stimulation is applied
        ↪ (First node is 0)
80         # 'HFSamp'          : 1.5, # Amplitude of KHfAC waveform (mA)
81         # 'HFSfrequency'   : 10.0, # Frequency of KHfAC waveform (kHz)
82         'HFSdelay'       : 0.0, # Time when KHfAC waveform starts (ms)
83         'HFSx'           : 0.0, # Location of the KHfAC electrode on the
        ↪ x-axis (um)
84         'HFSy'           : 0.0, # Location of the KHfAC electrode on the
        ↪ y-axis (um)
85         'HFSz'           : 1000.0, # Location of the KHfAC electrode on the
        ↪ z-axis (um)
86         # PARAMETERS FOR INTRACELLULAR STIMULUS
87         'intrinsicStim' : 1,    # Change to 1 to add intrinsic pulse(train)
88         'intrinsicType' : 1,    # 0 is single pulse, 1 is pulsetrain
89         'intrinsicNode' : 0,    # Node where intracellular stimulus starts
90         'intrinsicAmp'  : 2.0,  # Amplitude of intracellular stimulus (nA)
91         'intrinsicDel'  : 10.0, # Time when intracellular stimulus starts (ms)
92         'intrinsicDur'  : 0.1,  # Duration of intracellular stimulus (ms)
        ↪ # ONLY SINGLE PULSE
93         'intrinsicTON'  : 0.5,  # Duration of ON phase intracellular stimulus (ms)
        ↪ # ONLY PULSE TRAIN
94         'intrinsicTOFF' : 1.5,  # Duration of OFF phase intracellular stimulus
        ↪ (ms) # ONLY PULSE TRAIN
95         'intrinsicNum' : None   # Number of intrinsic pulses (when doing a
        ↪ pulsetrain)
96     })
97 elif config=="ClassicModel_Bipolar":
98     par.update({
99         # [MRGnode]
100        # ENVIRONMENTAL PARAMETERS
101        'axonnodes' : 51.0,    # Amount of Nodes of Ranvier on the axon
102        # SIMULATION PARAMETERS
103        'tstop'     : 51,      # Total simulation time (in ms)
104        'dt'        : 0.001,   # Simulation timestep (in ms)
105        # HFS PARAMETERS
106        'HFSreferenceNode' : 25, # Node of Ranvier where KHfAC stimulation is applied
        ↪ (First node is 0)
107        # 'HFSamp'          : 1.5, # Amplitude of KHfAC waveform (mA)
108        # 'HFSfrequency'   : 10.0, # Frequency of KHfAC waveform (kHz)
109        'HFSdelay'       : 0.0, # Time when KHfAC waveform starts (ms)
110        'HFSx'           : 0.0, # Location of the KHfAC electrode on the
        ↪ x-axis (um)
111        'HFSy'           : 0.0, # Location of the KHfAC electrode on the
        ↪ y-axis (um)
112        'HFSz'           : 1000.0, # Location of the KHfAC electrode on the
        ↪ z-axis (um)
113        'bipolar'       : 1,    # Set to 1 for bipolar stimulation
114        'HFSx2'         : 0.0,  # Location of the KHfAC electrode on the
        ↪ x-axis (um)
115        'HFSy2'         : 0.0,  # Location of the KHfAC electrode on the
        ↪ y-axis (um)
116        'HFSz2'         : 1000.0, # Location of the KHfAC electrode on the
        ↪ z-axis (um)
117        # PARAMETERS FOR INTRACELLULAR STIMULUS
118        'intrinsicStim' : 1,    # Change to 1 to add intrinsic pulse(train)

```

```

119     'intrinsicType'           : 1,          # 0 is single pulse, 1 is pulsetrain
120     'intrinsicNode'         : 0,          # Node where intracellular stimulus starts
121     'intrinsicAmp'          : 2.0,        # Amplitude of intracellular stimulus (nA)
122     'intrinsicDel'          : 10.0,       # Time when intracellular stimulus starts (ms)
123     'intrinsicDur'          : 0.1,        # Duration of intracellular stimulus (ms)
124     ↪ # ONLY SINGLE PULSE
125     'intrinsicTON'          : 0.5,        # Duration of ON phase intracellular stimulus (ms)
126     ↪ # ONLY PULSE TRAIN
127     'intrinsicTOFF'         : 1.5,        # Duration of OFF phase intracellular stimulus
128     ↪ (ms) # ONLY PULSE TRAIN
129     'intrinsicNum'          : None        # Number of intrinsic pulses (when doing a
130     ↪ pulsetrain)
131 })
132 elif config=="ProposedModel":
133     # Create simplified Model
134     par.update({
135         # [MRGnode]
136         # ENVIRONMENTAL PARAMETERS
137         'axonnodes'          : 5,          # Amount of Nodes of Ranvier on the axon
138         ↪ (standard 5)
139         # SIMULATION PARAMETERS
140         'dt'                  : 0.001,     # Simulation timestep (in ms)
141         'tstop'               : 20.0,     # Total simulation time (in ms) (standard
142         ↪ 2)
143         # HFS PARAMETERS
144         'HFSamp'              : 0.5,       # Amplitude of KHFAC waveform (mA)
145         'HFSfrequency'        : 10.0,     # Frequency of KHFAC waveform (kHz)
146         'HFSdelay'            : 0.0,       # Time when KHFAC waveform starts (ms)
147         'HFSx'                : 0.0,       # Location of the KHFAC electrode on the
148         ↪ x-axis ()
149         'HFSy'                : 0.0,       # Location of the KHFAC electrode on the
150         ↪ y-axis ()
151         'HFSz'                : 1000.0,    # Location of the KHFAC electrode on the
152         ↪ z-axis ()
153         'tonset'              : 18,        # Duration of onset response (standard
154         ↪ 2)
155     })
156 elif config == "ProposedModel_Bipolar":
157     # Create simplified Model
158     par.update({
159         # [MRGnode]
160         # ENVIRONMENTAL PARAMETERS
161         'axonnodes'          : 5,          # Amount of Nodes of Ranvier on the axon (standard 5)
162         # SIMULATION PARAMETERS
163         'dt'                  : 0.001,     # Simulation timestep (in ms)
164         'tstop'               : 20.0,     # Total simulation time (in ms) (standard
165         ↪ 2)
166         # HFS PARAMETERS
167         'HFSamp'              : 0.5,       # Amplitude of KHFAC waveform (mA)
168         'HFSfrequency'        : 10.0,     # Frequency of KHFAC waveform (kHz)
169         'HFSdelay'            : 0.0,       # Time when KHFAC waveform starts (ms)
170         'HFSx'                : 0.0,       # Location of the KHFAC electrode on the
171         ↪ x-axis ()
172         'HFSy'                : 0.0,       # Location of the KHFAC electrode on the
173         ↪ y-axis ()
174         'HFSz'                : 1000.0,    # Location of the KHFAC electrode on the
175         ↪ z-axis ()
176         'tonset'              : 18,        # Duration of onset response (standard
177         ↪ 2)
178         'bipolar'             : 1,         # Set to 1 for bipolar stimulation
179         'HFSx2'               : 0.0,       # Location of the KHFAC electrode on the
180         ↪ x-axis
181         'HFSy2'               : 0.0,       # Location of the KHFAC electrode on the
182         ↪ y-axis
183         'HFSz2'               : 1000.0,    # Location of the KHFAC electrode on the
184         ↪ z-axis ()
185     })
186 par['HFSreferenceNode'] = int(math.floor(par['axonnodes']/2)) # Node of Ranvier where KHFAC
187 ↪ stimulation is applied (first node = 0)
188
189 if par['intrinsicStim'] == 1:
190     par['intrinsicNum'] = int(par['tstop'] / (par['intrinsicTON'] + par['intrinsicTOFF']))
191
192 recpar = { # Parameters used for recording
193     # 'record'           : True,
194     # 'plot'            : False,
195     'nodes'            : np.array(range(0, int(par['axonnodes']))) , # Nodes that
196     ↪ are to be recorded

```

```

178     'recordVoltage'           : True,      # Switch to allow recording of voltages
179 }
180
181 return par, recpar

```

MRG_Model.py

```

1 #####
2 # INTERFACE FUNCTIONS FOR PYTHON<->NEURON
3 # This is an adopted version from the work of Baquer Gomez (not published in thesis):
4 # https://repository.tudelft.nl/islandora/object/uuid%3A5c875b14-08a3-4ef0-8e90-4fc997dc19d3
5 # Extended by Koen Emmer to record gate variables, all other credits go to Baquer Gomez and the
6 # names mentioned in the code
7 #####
8 import sys
9 sys.path.append("<path>")
10 from neuron import h
11 import numpy as np
12
13
14 #####
15 # Functions that load variables into Neuron
16 #####
17 def pass_parameters_to_nrn(parameters):#, verb=False): # USED
18     '''
19     Passes parameters from a dictionary to NEURON.
20     If the element is a vector it assumes that the a vector
21     has been created as objref and new Vector() in the hoc code.
22     Set 'verb' to True for verbose.
23     '''
24     for k, v in parameters.iteritems():
25         if type(v) is not type(np.array([])):
26             h("{k} = " + str(v) + ";")
27             # if verb:
28             #     print(k + " = " + str(v))
29
30
31 #####
32 # Functions that initialize the axon in Neuron
33 #####
34 def createMRGaxon(par):#, verbose): # USED
35     '''
36     Initializes the model.
37     Creates the axon and stimulation according to the parameters.
38     '''
39     h('{load_file("stdgui.hoc")}') # Starts the Neuron session
40     pass_parameters_to_nrn(par)#, verb=verbose) # Passes the configuration file
41     h('{load_file("MRG_MODEL_edit.hoc")}') # Loads the hoc file of the model
42     h('{buildModel()}') # Runs the buildmodel function of the hoc file,
43     ↪ creating the axon
44
45 #####
46 # Functions that set up recording process in Neuron
47 #####
48 def recordMRGaxon(recpar):#, verbose):
49     '''
50     Inserts the recorders as specified in recpar.
51     '''
52     k = recpar['nodes'] # List with all the nodes [0, 1, 2, ...,
53     ↪ N]
54     rec = {}
55
56     if recpar['recordVoltage']:
57         rec['voltage'] = record_node_voltage(k) # Maybe not used when looking for gating
58         ↪ variables
59     rec['gates'] = record_node_gates(k)
60     rec['current'] = record_node_current(k)
61     rec['spiketimes'], rec['apcount'] = record_node_spikes(k)
62     # if verbose:
63     #     print('Now recording from ' + str(k))
64     return rec
65
66 def record_node_voltage(nodenum, rec=None):
67     '''

```



```

67     Records the membrane potential of a particular set of nodes.
68     '''
69     segments = []
70     for n in nodenumber:
71         segments.append(h.node[n](0.5))
72     for seg, n in zip(segments, nodenumber):
73         rec = insert_nrn_recorders(seg, {'v_node' + str(n): '_ref_v'}, rec) # Another function layer;
74         ↪ necessary?
75     return rec
76
77 def record_node_gates(nodenumber, rec=None):
78     segments = []
79     for n in nodenumber:
80         segments.append(h.node[n](0.5))
81     for seg, n in zip(segments, nodenumber):
82         rec = insert_nrn_recorders(seg, {'h_node' + str(n): '_ref_h_axnode'}, rec) # Nonlinear fast
83         ↪ sodium inactivation gate
84         rec = insert_nrn_recorders(seg, {'m_node' + str(n): '_ref_m_axnode'}, rec) # Nonlinear fast
85         ↪ sodium activation gate
86         rec = insert_nrn_recorders(seg, {'mp_node' + str(n): '_ref_mp_axnode'}, rec) # Persistent
87         ↪ sodium activation gate
88         rec = insert_nrn_recorders(seg, {'s_node' + str(n): '_ref_s_axnode'}, rec) # Slow potassium
89         ↪ activation gate
90     return rec
91
92 def record_node_current(nodenumber, rec=None):
93     segments = []
94     for n in nodenumber:
95         segments.append(h.node[n](0.5))
96     for seg, n in zip(segments, nodenumber):
97         rec = insert_nrn_recorders(seg, {'i_Na_node' + str(n): '_ref_ina_axnode'}, rec)
98         rec = insert_nrn_recorders(seg, {'i_Nap_node' + str(n): '_ref_inap_axnode'}, rec)
99         rec = insert_nrn_recorders(seg, {'i_K_node' + str(n): '_ref_ik_axnode'}, rec)
100        rec = insert_nrn_recorders(seg, {'i_l_node' + str(n): '_ref_il_axnode'}, rec)
101
102        rec = insert_nrn_recorders(seg, {'m_inf_node' + str(n): '_ref_m_inf_axnode'}, rec)
103        rec = insert_nrn_recorders(seg, {'tau_m_node' + str(n): '_ref_tau_m_axnode'}, rec)
104        rec = insert_nrn_recorders(seg, {'h_inf_node' + str(n): '_ref_h_inf_axnode'}, rec)
105        rec = insert_nrn_recorders(seg, {'tau_h_node' + str(n): '_ref_tau_h_axnode'}, rec)
106    return rec
107
108 def record_node_spikes(nodenumber, rec=None,
109                       apc=None, threshold = -15):
110     '''
111     Records the action potentials of a particular set of nodes.
112     Returns a "rec" dictionary.
113     '''
114     if rec is None:
115         rec = {}
116     if apc is None:
117         apc = {}
118     for n in nodenumber:
119         apc['apc'+str(n)] = h.APCount(h.node[int(n)](0.5))
120         apc['apc'+str(n)].thresh = threshold
121         rec['spk'+str(n)] = h.Vector()
122         apc['apc'+str(n)].record(rec['spk'+str(n)])
123     return rec, apc
124
125 def insert_nrn_recorders(segment, labels, rec=None):
126     '''
127     Inserts recorders for NEURON state variables.
128     Use one per segment.
129     "labels" is a dictionary.
130     Example {'v': '_ref_v'}.
131     Specify 'rec' to append to previous recorders.
132     Records also time if 'rec' is 'None' (default).
133     (Acknowledgements: Daniele Linaro)
134     '''
135     if rec is None:
136         rec = {'t': h.Vector()}
137         rec['t'].record(h._ref_t)
138     for k, v in labels.items():
139         rec[k] = h.Vector()
140         rec[k].record(getattr(segment, v))

```

```

140     return rec
141
142
143 #####
144 # Functions run a simulation of the axon
145 #####
146 def runMRGaxon(rec, ts, xs):
147     h.resetModel()
148     tvec = h.Vector(len(ts)) #time vector
149     for i, t in enumerate(ts):
150         tvec.x[i] = t
151     xvec = h.Vector(len(xs)) #amplitude vector
152     for i, x in enumerate(xs):
153         xvec.x[i] = x
154     xvec.play(h.exIClmp._ref_i, tvec, 1)
155     h.run()
156     rec['i_block'] = {'i': [], 't': rec['voltage']['t']}
157     rec['i_block']['i'] = h.rec_electrode_block
158     rec['i_stim'] = {'i': [], 't': rec['voltage']['t']}
159     rec['i_stim']['i'] = h.rec_electrode_stim
160     return rec
161
162
163 #####
164 # Functions that reset values between simulations
165 #####
166 def updateMRGaxon(par):#, verbose):
167     '''
168     Updates the parameters of the model.
169     '''
170     pass_parameters_to_nrn(par)#, verb=verbose)
171     h.resetModel()
172
173 def quitNeuron():
174     h.quit()

```

MRG_MODEL_edit.hoc

```

1  /*-----
2  Koen Emmer
3  SIMULATION OF PNS MYELINATED AXON WITH EXTRACELLULAR AND INTRACELLULAR ELECTRODE.
4
5  This model is an adapted version of MRGaxon.hoc:
6  Cameron C. McIntyre
7  SIMULATION OF PNS MYELINATED AXON
8  This model is described in detail in:
9  McIntyre CC, Richardson AG, and Grill WM. Modeling the excitability of
10 mammalian nerve fibers: influence of afterpotentials on the recovery
11 cycle. Journal of Neurophysiology 87:995-1006, 2002.
12
13 The basis of this adapted version was a version made by Joao Couto, that can be found here:
14 https://github.com/jcouto/neuron/blob/master/MRGaxon/MRGnodeHFS.hoc
15 -----
16
17 stdgui (or nrnmainmenu) needs to be loaded first from Python to correctly work. Next parameters can be
18 ↪ passed to Neuron, and then this .hoc file can be opened.
19 The following variables can be set://
20 axonnodes          - number of nodes that are going to be used in the simulation
21 fiberD             - diameter of the fiber, can be set to: 5.7, 7.3, 8.7, 10.0, 11.5,
22 ↪ 12.8, 14.0, 15.0 or 16.0
23
24 ipulse1.mod and axnode.mod are necessary for the model to work:
25 ipulse1.mod:
26 ↪ https://senselab.med.yale.edu/ModelDB/ShowModel.cshmtl?model=225086&file=%2Fgfc-gna\_distribution%2Fipulse1.mod#tabs-2
27 axnode.mod can be found with the MRG model on ModelDB
28 -----*/
29
30 proc model_globals() {
31     //celsius = 37
32     //v_init = -80
33     //dt
34     //tstop
35
36     //-----TOPOLOGICAL PARAMETERS-----
37     // FULL SEGMENT = -NODE-MYSA-FLUT-STIN-STIN-STIN-STIN-STIN-STIN-FLUT-MYSA-NODE-
38     paranodes1 = 2*(axonnodes-1) // Number of MYSA paranodes

```

```

36 paranodes2 = 2*(axonnodes-1) // Number of FLUT paranodes
37 axoninter = 6*(axonnodes-1) // Number of STIN internodes
38 axontotal = axonnodes+paranodes1+paranodes2+axoninter // Total amount of segments
39
40 //-----MORPHOLOGICAL PARAMETERS-----
41 // All morphological parameters are in um (micrometers)
42 //fiberD=5.7 // Fiber Diameter, choose from 5.7, 7.3, 8.7, 10.0, 11.5, 12.8, 14.0, 15.0,
43 // 16.0
44 paralength1 = 3 // MYSA Length
45 nodelength = 1.0 // Node length
46 space_p1 = 0.002 // MYSA periaxonal space width
47 space_p2 = 0.004 // FLUT periaxonal space width
48 space_i = 0.004 // STIN periaxonal space width
49
50 //-----ELECTRICAL PARAMETERS-----
51 rhoa = 0.7e6 // Axoplasmic resistivity // Ohm-um //
52 mycm = 0.1 // Myelin capacitance //
53 // uF/cm2/lamella membrane // 2 membranes per lamella
54 mygm = 0.001 // Myelin conductance // S/cm2/lamella
55 // membrane // 2 membranes per lamella
56 rhoe = 5e6 // Resistivity of extracellular medium // Ohm-um//
57
58 // DELETE EVERYTHING BENEATH?
59 //-----RECORDING-----
60 //recordAll = 0 /*Uncomment to record voltage from all nodes.*/
61
62 }
63 model_globals ()
64
65 //-----INITIALIZE THE DEPENDENT VARIABLES-----
66 proc dependent_var() {
67 // The fiber diameter can not be changed in the middle of the simulation.
68 /* fiberD = Fiber diameter g = deltax = Node-node
69 // separation
70 axonD = Axon diameter paraD1 = MYSA diameter paralength2 = FLUT length
71 nodeD = Node diameter paraD2 = FLUT diameter (same as STIN) nl = number of myelin
72 // lemella */
73 if (fiberD==5.7) {g=0.605 axonD=3.4 nodeD=1.9 paraD1=1.9 paraD2=3.4 deltax=500 paralength2=35
74 // nl=80 }
75 if (fiberD==7.3) {g=0.630 axonD=4.6 nodeD=2.4 paraD1=2.4 paraD2=4.6 deltax=750 paralength2=38
76 // nl=100}
77 if (fiberD==8.7) {g=0.661 axonD=5.8 nodeD=2.8 paraD1=2.8 paraD2=5.8 deltax=1000 paralength2=40
78 // nl=110}
79 if (fiberD==10.0) {g=0.690 axonD=6.9 nodeD=3.3 paraD1=3.3 paraD2=6.9 deltax=1150 paralength2=46
80 // nl=120}
81 if (fiberD==11.5) {g=0.700 axonD=8.1 nodeD=3.7 paraD1=3.7 paraD2=8.1 deltax=1250 paralength2=50
82 // nl=130}
83 if (fiberD==12.8) {g=0.719 axonD=9.2 nodeD=4.2 paraD1=4.2 paraD2=9.2 deltax=1350 paralength2=54
84 // nl=135}
85 if (fiberD==14.0) {g=0.739 axonD=10.4 nodeD=4.7 paraD1=4.7 paraD2=10.4 deltax=1400 paralength2=56
86 // nl=140}
87 if (fiberD==15.0) {g=0.767 axonD=11.5 nodeD=5.0 paraD1=5.0 paraD2=11.5 deltax=1450 paralength2=58
88 // nl=145}
89 if (fiberD==16.0) {g=0.791 axonD=12.7 nodeD=5.5 paraD1=5.5 paraD2=12.7 deltax=1500 paralength2=60
90 // nl=150}
91
92 // Inverses of Gp:
93 Rpn0 = (rhoa*.01)/(PI*(((nodeD/2)+space_p1)^2)-((nodeD/2)^2)) // Periaxonal resistance
94 // NODE
95 Rpn1 = (rhoa*.01)/(PI*(((paraD1/2)+space_p1)^2)-((paraD1/2)^2)) // Periaxonal resistance
96 // MYSA
97 Rpn2 = (rhoa*.01)/(PI*(((paraD2/2)+space_p2)^2)-((paraD2/2)^2)) // Periaxonal resistance
98 // FLUT
99 Rpx = (rhoa*.01)/(PI*(((axonD/2)+space_i)^2)-((axonD/2)^2)) // Periaxonal resistance
100 // STIN
101
102 interlength = (deltax-nodelength-(2*paralength1)-(2*paralength2))/6
103 }
104 dependent_var()
105
106 //-----CREATE SECTIONS AND APPEND CHANNELS-----
107 create node[axonnodes], MYSA[paranodes1], FLUT[paranodes2], STIN[axoninter]
108 access node[0]
109 objref s[axontotal]
110 create electrode
111 not_built = 1
112
113
114
115

```

```

96 proc initMRG() {
97     //print "Initializing MRGaxon."
98     for i=0, axonnodes-1 {
99         node[i]{
100             s[i] = new SectionRef()
101             nseg = 1
102             diam = nodeD
103             L = nodelength
104             Ra = rhoa/10000
105             cm = 2
106             insert axnode
107             insert extracellular xrxial=Rpn0 xg=1e10 xc=0
108         }
109     }
110     for i=0, paranodes1-1 {
111         MYSA[i]{
112             s[axonnodes+i] = new SectionRef()
113             nseg = 1
114             diam = fiberD
115             L = paralength1
116             Ra = rhoa*(1/(paraD1/fiberD)^2)/10000
117             cm = 2*paraD1/fiberD
118             insert pas
119             g_pas = 0.001*paraD1/fiberD
120             e_pas = v_init
121             insert extracellular xrxial=Rpn1 xg=mygm/(nl*2) xc=mycm/(nl*2)
122         }
123     }
124     for i=0, paranodes2-1 {
125         FLUT[i]{
126             s[axonnodes+paranodes1+i] = new SectionRef()
127             nseg = 1
128             diam = fiberD
129             L = paralength2
130             Ra = rhoa*(1/(paraD2/fiberD)^2)/10000
131             cm = 2*paraD2/fiberD
132             insert pas
133             g_pas = 0.0001*paraD2/fiberD
134             e_pas = v_init
135             insert extracellular xrxial=Rpn2 xg=mygm/(nl*2) xc=mycm/(nl*2)
136         }
137     }
138     for i=0, axoninter-1 {
139         STIN[i]{
140             s[axonnodes+paranodes1+paranodes2+i]=new SectionRef()
141             nseg = 1
142             diam = fiberD
143             L = interlength
144             Ra = rhoa*(1/(axonD/fiberD)^2)/10000
145             cm = 2*axonD/fiberD
146             insert pas
147             g_pas = 0.0001*axonD/fiberD
148             e_pas = v_init
149             insert extracellular xrxial=Rpx xg=mygm/(nl*2) xc=mycm/(nl*2)
150         }
151     }
152     for i=0, axonnodes-2 {
153         connect MYSA[2*i](0), node[i](1)
154         connect FLUT[2*i](0), MYSA[2*i](1)
155         connect STIN[6*i](0), FLUT[2*i](1)
156         connect STIN[6*i+1](0), STIN[6*i](1)
157         connect STIN[6*i+2](0), STIN[6*i+1](1)
158         connect STIN[6*i+3](0), STIN[6*i+2](1)
159         connect STIN[6*i+4](0), STIN[6*i+3](1)
160         connect STIN[6*i+5](0), STIN[6*i+4](1)
161         connect FLUT[2*i+1](0), STIN[6*i+5](1)
162         connect MYSA[2*i+1](0), FLUT[2*i+1](1)
163         connect node[i+1](0), MYSA[2*i+1](1)
164     }
165 }
166
167 proc initialize() {
168     finitialize(v_init)
169     fcurrent()
170 }
171
172 //-----VECTORS AND OBJECT VARIABLES INITIALIZATION-----
173 objref Ve, X, Y, Z, exIClmp, inIClmp, apc, rec[axonnodes], rec_spk, v_first, v_ref, v_last, rec_Ve

```

```

174 //----INITIALIZE THE LOCATION STIM WITH REGARDS TO EACH COMPARTMENT----
175 //      The reference node is at position zero.
176 proc XYZ_loc() {
177 //-----VECTORS AND OBJECT VARIABLES INITIALIZATION-----
178   Ve = new Vector(axontotal,0)
179   X = new Vector(axontotal,0)
180   Y = new Vector(axontotal,0)
181   Z = new Vector(axontotal,0)
182   // UNCOMMENT TO RECORD ELECTRODE
183   rec_Ve = new Vector()
184   rec_Ve.record(&s[HFSreferenceNode].sec.e_extracellular) //mV
185   rec_Ve.clear()
186
187   //Set location of center of first node of Ranvier
188   reference_value = (HFSreferenceNode)*(2*paralength1+2*paralength2+6*interlength+nodelength)
189   // This was HFSreferenceNode-1
190   Y.x[0]=-reference_value
191
192   //Set location centers of nodes of Ranvier
193   for i=1, axonnodes-1 {
194     Y.x[i]=Y.x[i-1]+(2*paralength1+2*paralength2+6*interlength+nodelength)
195   }
196   // Set location centers of paranode and internode segments
197   for i=0, axonnodes-2 { // -2, as i starts at 0, and the amount of internode segments is one less
198     // than amount of nodes
199     Y.x[axonnodes+2*i] = Y.x[i] + nodelength/2+paralength1/2
200     // MYSA right to first node of Ranvier
201     Y.x[axonnodes+paranodes1+2*i] = Y.x[axonnodes+2*i] + paralength1/2 + paralength2/2
202     // FLUT right to first MYSA
203     Y.x[axonnodes+paranodes1+paranodes2+6*i] = Y.x[axonnodes+paranodes1+2*i] + paralength2/2 +
204     // interlength/2 // 1st STIN right to first FLUT
205     Y.x[axonnodes+paranodes1+paranodes2+6*i+1] = Y.x[axonnodes+paranodes1+paranodes2+6*i] +
206     // interlength // 2nd STIN right to first FLUT
207     Y.x[axonnodes+paranodes1+paranodes2+6*i+2] = Y.x[axonnodes+paranodes1+paranodes2+6*i+1] +
208     // interlength // 3rd STIN right to first FLUT
209     Y.x[axonnodes+paranodes1+paranodes2+6*i+3] = Y.x[axonnodes+paranodes1+paranodes2+6*i+2] +
210     // interlength // 4th STIN right to first FLUT
211     Y.x[axonnodes+paranodes1+paranodes2+6*i+4] = Y.x[axonnodes+paranodes1+paranodes2+6*i+3] +
212     // interlength // 5th STIN right to first FLUT
213     Y.x[axonnodes+paranodes1+paranodes2+6*i+5] = Y.x[axonnodes+paranodes1+paranodes2+6*i+4] +
214     // interlength // 6th STIN right to first FLUT
215     Y.x[axonnodes+paranodes1+2*i+1] = Y.x[axonnodes+2*i+1] - paralength1/2 -paralength2/2
216     // FLUT left to second MYSA
217     Y.x[axonnodes+2*i+1] = Y.x[i+1] - nodelength/2 - paralength1/2
218     // MYSA left to next node of Ranvier
219   }
220   for i=0, axontotal-1 {
221     X.x[i]=0 //Model build over Y-axis,
222     Z.x[i]=0 //centered over x- and z-axis
223   }
224 }
225
226 //-----CALCULATE THE ELECTRIC POTENTIAL SCALAR-----
227 //Uses the approximation of a current point source and assumes a uniform field. Return electrode placed
228 // at infinity
229 proc calc_voltage() { local i
230   if(bipolar == 0){
231     for i=0, axontotal-1 {
232       Ve.x[i]=(rhoehoe)/(4*PI*sqrt((HFSz-Z.x[i])^2+(HFSy-Y.x[i])^2+(HFSx-X.x[i])^2))
233     }
234   }else if(bipolar == 1){
235     for i=0, axontotal-1 {
236       // Ve.x[i]=(rhoehoe)*((1/(4*PI*sqrt((HFSz-Z.x[i])^2+(HFSy-Y.x[i])^2+(HFSx-X.x[i])^2)))-(1/(4*PI*sqrt((HFSz2-Z.x[i])^2
237     }
238   }
239 }
240
241 objref rec_electrode_block
242 objref rec_electrode_stim
243 proc stimulus() {
244   calc_voltage()
245   //-----INITIALIZE THE EXTRACELLULAR STIM-----
246   electrode{
247     exIClmp = new IClamp(0.5)
248   }
249 }

```

```

238 // UNCOMMENT TO RECORD ELECTRODE
239 rec_electrode_block = new Vector()
240 rec_electrode_block.record(&exIClmp.i)
241 rec_electrode_block.clear()
242 //
243
244 //-----INITIALIZE THE INTRACELLULAR STIM-----
245 if(intrinsicStim == 1){
246     node[intrinsicNode+1] {
247         // inject pattern in one of the ends of the axon.
248         if(intrinsicType == 0){
249             inIClmp = new IClamp(0.5)
250             inIClmp.del = intrinsicDel // The delay until the onset of the stimulus (in ms)
251             inIClmp.dur = intrinsicDur // The duration of the stimulus (in ms)
252             inIClmp.amp = intrinsicAmp // The amplitude of the stimulus (in nA)
253         }else if(intrinsicType == 1){
254             inIClmp = new Ipulse1(0.5)
255             inIClmp.del = intrinsicDel // The delay until the onset of the stimulus (in ms)
256             inIClmp.num = intrinsicNum // The duration of the stimulus (in ms)
257             inIClmp.amp = intrinsicAmp // The amplitude of the stimulus (in nA)
258             inIClmp.ton = intrinsicTON // The duration of ON phase (in ms)
259             inIClmp.toff = intrinsicTOFF // The duration of OFF phase (in ms)
260         }
261     }
262     rec_electrode_stim = new Vector()
263     rec_electrode_stim.record(&inIClmp.i) // inIClmp.i = current ic inIClmp
264     rec_electrode_stim.clear()
265 }
266 }
267
268 //-----DEFINES WHAT TO DO AT EACH TIMESTEP-----
269 proc advance() { local i
270     // Set Vextracellular using neurons mechanisms.
271     for ii=0,axontotal-1 {
272         s[ii].sec.e_extracellular(0.5)=(exIClmp.i)*Ve.x[ii] // Set the extracellular voltages at
273         // each segment
274     }
275     fadvance()
276 }
277
278 proc makeRecorders() {
279     v_first = new Vector()
280     v_ref = new Vector()
281     v_last = new Vector()
282     v_first.record(&node[0].v(0.5))
283     v_ref.record(&node[HFSreferenceNode].v(0.5))
284     v_last.record(&node[axonnodes-1].v(0.5))
285     //-----RECORD ACTION POTENTIAL TIMESTAMPS-----
286     node[axonnodes-1] {
287         apc = new APCount(0.5)
288         apc.thresh = -15
289         rec_spk = new Vector()
290         apc.record(rec_spk)
291     } //RECORDS INTO REC_SPK
292
293     //-----RECORD ALL VOLTAGE TRACES (if recordAll)-----
294     if(recordAll) {
295         for ii=0,axonnodes-1 {
296             rec[ii] = new Vector()
297             rec[ii].record(&node[ii].v(0.5))
298         }
299     }
300 }
301
302 // Build the model
303 proc buildModel() {
304     if (not_built) {
305         initMRG()
306         XYZ_loc() // Storing locations of the segments
307         stimulus()
308         initialize()
309         not_built=0
310     }
311 }
312
313 // Reset the model
314 proc resetModel() {

```

```

315     XYZ_loc()
316     stimulus()
317     initialize()
318     init()
319     //print "Reset MRGawon."
320 }

```

datasetproposedmethod.py

```

1  from simulationfunctions_proposedmodel import *
2  import os
3  if not os.path.exists('justification_files'):
4      os.makedirs('justification_files')          # Ensure that brainframe makes justification folder
5
6
7  #####
8  # Build dataset for square wave tested with Proposed Model
9  #####
10 def squarewaveresults():
11     frequencylist = [3]
12     for i in range(19):
13         frequencylist.append((i + 2) * 2)
14
15     result = find_blockingamplitudes_Freq(frequencylist, "square")
16
17     filename = 'justification_files/results_squarewave_proposedmethod'
18
19     np.save(filename, result)
20     CSVsave_blockingamplitudes_2D(result, "Frequency", "Frequency (kHz)", filename)
21
22
23 #####
24 # Build dataset for sine wave tested with Proposed Model
25 #####
26 def sineresults():
27     frequencylist = [3]
28     for i in range(19):
29         frequencylist.append((i + 2) * 2)
30
31     result = find_blockingamplitudes_Freq(frequencylist, "sine")
32
33     now = datetime.datetime.now()
34     # filename = 'results/results_' + now.strftime("%Y%m%d_%H%M%S")
35     filename = 'justification_files/results_sinewave_proposedmethod' + now.strftime("%Y%m%d_%H%M%S")
36
37     np.save(filename, result)
38     CSVsave_blockingamplitudes_2D(result, "Frequency", "Frequency (kHz)", filename)
39
40
41 #####
42 # Build dataset for asymmetricalwave tested with Proposed Model
43 #####
44 def asymmetricalresults():
45     frequencylist = [3]
46     for i in range(19):
47         frequencylist.append((i + 2) * 2)
48     chargebalancelist = [0.1]
49     for i in range(9):
50         chargebalancelist.append(round(0.1 + 0.1*i, 1))
51     result = find_blockingamplitudes_CB(frequencylist, chargebalancelist)
52
53     filename = 'justification_files/results_squarewave_CB_proposedmethod'
54
55     np.save(filename, result)
56     CSVsave_blockingamplitudes_3D(result, "Frequency", "Frequency (kHz)", "Chargeperphase", "Charge per
    ↳ Phase (uC)", filename)

```

simulationfunctions.py

```

1  #####
2  # This file contains simulation functions.
3  #####
4  # Import Python libraries
5  import datetime

```

```

6 import csv
7 import pylab as plt
8 import os
9 if not os.path.exists('results'):
10     os.makedirs('results') # Ensure that brainframe makes result folder
11 # Import custom function libraries
12 from MRG_Model import * # Interface with Neuron MRG Model
13 from plottingfunctions import * # Plotting functions
14 from blockdetection import *
15 from waveforms import * # Waveform functions for Neuron
16 import time
17
18
19 #####
20 # Functions to save data to CSV
21 #####
22 def savetocsv(result, now = datetime.datetime.now()):
23     with open('results/results_' + now.strftime("%Y%m%d_%H%M%S") + '.csv', mode='w') as csv_file:
24         fieldnames = ["Node", "Frequency (kHz)", "Amplitude (mA)", "Block %",
25                       "h_max", "h_min", "V_max", "V_min", "m_max", "m_min", "h_dist_max", "h_dist_min"]
26         writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
27         writer.writeheader()
28
29         for i in result:
30             for j in result[i]:
31                 for k in result[i][j]:
32                     if result[i][j][k] != {}: # Empty # Empty
33                         ↪ dicts should be ignored
34                         result[i][j][k]["Frequency (kHz)"] = result[i][j][k].pop("Frequency") # Needed
35                         ↪ to make compliant to fieldnames of CSV
36                         result[i][j][k]["Amplitude (mA)"] = result[i][j][k].pop("Amplitude") # Needed
37                         ↪ to make compliant to fieldnames of CSV
38                         writer.writerow(result[i][j][k])
39
40 # Asymmetrical waves csv-save function
41 def savetocsv2(result, now = datetime.datetime.now()):
42
43     with open('results/results_' + now.strftime("%Y%m%d_%H%M%S") + '.csv', mode='w') as csv_file:
44         fieldnames = ["Node", "Frequency (kHz)", "Chargebalance", "Charge per Phase (uC)", "Amplitude
45                       ↪ (mA)", "Block %",
46                       "h_max", "h_min", "V_max", "V_min", "m_max", "m_min", "h_dist_max", "h_dist_min"]
47         writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
48         writer.writeheader()
49
50         for i in result:
51             for j in result[i]:
52                 for k in result[i][j]:
53                     for l in result[i][j][k]:
54                         if result[i][j][k][l] != {}:
55                             result[i][j][k][l]["Frequency (kHz)"] =
56                                 ↪ result[i][j][k][l].pop("Frequency") # Needed to make
57                                 ↪ compliant to fieldnames of CSV
58                             result[i][j][k][l]["Charge per Phase (uC)"] =
59                                 ↪ result[i][j][k][l].pop("Chargeperphase") # Needed to make compliant
60                                 ↪ to fieldnames of CSV
61                             result[i][j][k][l]["Amplitude (mA)"] =
62                                 ↪ result[i][j][k][l].pop("Amplitude") # Needed to make
63                                 ↪ compliant to fieldnames of CSV
64                             writer.writerow(result[i][j][k][l])
65
66 #####
67 # This functions runs simulations and returns the results as a dataset
68 # Find blockingamplitudes, sweep over frequency
69 #####
70 def run_and_save_FreqAmp(par, recpar, nodes, frequencylist, amplitudelist, waveform="square",
71 ↪ config="ClassicModel", runtime="fast", savetype = "saveall"):
72     # Run simulations for all frequencies and amplitudes and save data of nodes specified
73     # Time Process
74     starttime = time.time()
75
76     # Create the awon
77     createMRGaxon(par) # , verbose)
78     intervaltime = time.time() - starttime
79     print("--- CREATEAXON: %s seconds ---" % (intervaltime))
80     intervaltime = time.time()
81
82

```



```

73 amplitude_counter = 0          # Needed for fast simulations
74 # INITIALIZE RESULT ARRAY
75 result = {}
76 for f in frequencylist:
77     result[f] = {}
78     par['HFSfrequency'] = f
79     for a in amplitudelist[amplitude_counter:]:
80         rec = recordMRGaxon(recpar)
81         # RUN SIMULATION
82         par['HFSamp'] = a
83
84         if waveform == "sine":
85             t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
86             ↪ par['HFSamp'], 1000)
87         elif waveform == "square":
88             t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
89             ↪ par['HFSamp'])
90         else:
91             print "Incompatible waveform selected!"
92             return None
93
94     intervaltime = time.time() - intervaltime
95     print("--- BUILD AND RUN HFS-SIGNAL (%s kHz, %s mA) : %s seconds ---" % (f, a, intervaltime))
96     intervaltime = time.time()
97
98     updateMRGaxon(par) # , verbose)
99     runMRGaxon(rec, t_signal, i_signal)
100
101     if config == "ClassicModel":
102         effectiveness = blockeffectiveness(rec, par)
103     elif is_blocked(rec, par):
104         effectiveness = 1.0
105     else:
106         effectiveness = 0.0
107
108     # STORE RESULTS
109     if (effectiveness >= 0.90 and savetype == "saveblock") or savetype == "saveall":
110         result[f][a] = {}
111         for n in nodes:
112             result[f][a][n] = {}
113             h_max, h_min, V_max, V_min, m_max, m_min, h_dist_max, h_dist_min =
114             ↪ conductionBlock(rec, par, n)
115
116             result[n][f][a]["Node"] = n
117             result[n][f][a]["Frequency"] = f
118             result[n][f][a]["Amplitude"] = a
119             result[n][f][a]["Block %"] = effectiveness
120
121             result[n][f][a]["h_max"] = h_max
122             result[n][f][a]["h_min"] = h_min
123             result[n][f][a]["V_max"] = V_max
124             result[n][f][a]["V_min"] = V_min
125             result[n][f][a]["m_max"] = m_max
126             result[n][f][a]["m_min"] = m_min
127             result[n][f][a]["h_dist_max"] = h_dist_max
128             result[n][f][a]["h_dist_min"] = h_dist_min
129             # result[n][f][a]["rec"] = rec
130
131     # Code for fast runtime (skips amplitudes of which is known that they are not effective)
132     if runtime == "fast":
133         if effectiveness >= 0.90:
134             amplitude_counter = max(0, amplitude_counter - 5) # Optional
135             break
136
137         elif amplitude_counter >= len(amplitudelist) - 1:
138             amplitude_counter = 0
139             break
140
141         amplitude_counter += 1
142
143     totaltime = time.time() - starttime
144     print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
145
146     return result
147
148 #####

```

```

148 # This functions runs asymmetrical wave simulations and returns the results as a dataset
149 # Find blockingamplitudes, sweep over frequency and anode fraction
150 #####
151 def run_and_save_asymmetrical(par, recpar, nodes, frequencylist, amplitudelist, chargebalancelist,
↳ config="ClassicModel", runtime="fast", savetype="saveall"):
152     # Run simulations for all frequencies and amplitudes and save data of nodes specified
153
154     # Time Process
155     starttime = time.time()
156
157     # Create the axon
158     createMRGaxon(par) # , verbose)
159     intervaltime = time.time() - starttime
160     print("--- CREATEAXON: %s seconds ---" % (intervaltime))
161     intervaltime = time.time()
162
163     amplitude_counter = {}
164     for i in chargebalancelist:
165         amplitude_counter[i] = 0
166
167     result = {}
168     for b in chargebalancelist:
169         result[b] = {}
170         for f in frequencylist:
171             result[b][f] = {}
172             par['HFSfrequency'] = f
173             for a in amplitudelist[amplitude_counter[b]:]:
174                 rec = recordMRGaxon(recpar) # verbose)
175                 par['HFSamp'] = a
176                 # Transform amplitude into charge; amplitude is actually only the amplitude when
↳ chargebalance is 0.5
177                 cpp = a/(2*f) # Charge per phase is amplitude times half the period
178
179                 intervaltime = time.time() - intervaltime
180                 print("--- BUILD AND RUN HFS-SIGNAL (%s kHz, %s mA, %s chargebalance): %s seconds ---" %
↳ (f, a, b, intervaltime))
181                 intervaltime = time.time()
182
183                 # RUN SIMULATION
184                 t_signal, i_signal = chargebalanced_asymmetrical(par['HFSdelay'], par['tstop'],
↳ par['HFSfrequency'], cpp, b) # Build signal
185                 updateMRGaxon(par)
186                 runMRGaxon(rec, t_signal, i_signal)
187
188                 if config == "ClassicModel":
189                     effectiveness = blockeffectiveness(rec, par)
190                 elif is_blocked(rec, par):
191                     effectiveness = 1.0
192                 else:
193                     effectiveness = 0.0
194
195                 # STORE RESULTS
196                 if (effectiveness >= 0.90 and savetype == "saveblock") or savetype == "saveall":
197                     result[b][f][a] = {}
198                     for n in nodes:
199                         result[b][f][a][n] = {}
200                         h_max, h_min, V_max, V_min, m_max, m_min, h_dist_max, h_dist_min =
↳ conductionBlock(rec, par, n)
201
202                         result[b][f][a][n]["Node"] = n
203                         result[b][f][a][n]["Frequency"] = f
204                         result[b][f][a][n]["Chargebalance"] = b
205                         result[b][f][a][n]["Chargeperphase"] = cpp
206                         result[b][f][a][n]["Amplitude"] = a
207                         result[b][f][a][n]["Block %"] = effectiveness
208
209                         result[b][f][a][n]["h_max"] = h_max
210                         result[b][f][a][n]["h_min"] = h_min
211                         result[b][f][a][n]["V_max"] = V_max
212                         result[b][f][a][n]["V_min"] = V_min
213                         result[b][f][a][n]["m_max"] = m_max
214                         result[b][f][a][n]["m_min"] = m_min
215                         result[b][f][a][n]["h_dist_max"] = h_dist_max
216                         result[b][f][a][n]["h_dist_min"] = h_dist_min
217
218                 # Code for fast runtime (skips amplitudes of which is known that they are not effective)
219                 if runtime == "fast":
220                     if effectiveness >= 0.90:

```

```

221         amplitude_counter[b] = max(0, amplitude_counter - 5) # Optional
222         break
223
224     elif amplitude_counter[b] >= len(amplitudelist) - 1:
225         if f == min(frequencylist):
226             amplitude_counter[b] = 0
227         else:
228             amplitude_counter[b] = max(0, amplitude_counter[b] - 5)
229             break
230
231     amplitude_counter[b] += 1
232
233
234     totaltime = time.time() - starttime
235     print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
236
237     return result

```

simulationfunctions_proposedmodel.py

```

1  import csv
2  from configurations import *
3  import time
4  import datetime
5  from MRG_Model import *           # Interface with Neuron MRG Model
6  from blockdetection import *
7  from waveforms import *         # Waveform functions for Neuron
8
9  #####
10 # Functions to save data to CSV
11 #####
12 def CSVsave_blockingamplitudes_2D(result, dimname, dim1newname, filename, now =
↳ datetime.datetime.now()):
13     amplitudefield = "Blocking Threshold Amplitude (mA)"
14     with open(filename + '.csv', mode='w') as csv_file:
15         fieldnames = [dim1newname, amplitudefield]#, 'h_max']
16         writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
17         writer.writeheader()
18
19     for i in sorted(result):
20         if result[i] != {}: # Empty dicts should be ignored
21             result[i][dim1newname] = result[i].pop(dimname) # Needed to make compliant to
↳ fieldnames of CSV
22             result[i][amplitudefield] = result[i].pop("Amplitude") # Needed to make compliant to
↳ fieldnames of CSV
23             writer.writerow(result[i])
24
25
26 def CSVsave_blockingamplitudes_3D(result, dimname, dim1newname, dim2name, dim2newname, filename, now =
↳ datetime.datetime.now()):
27     amplitudefield = "Blocking Threshold Amplitude (mA)"
28     with open(filename + '.csv', mode='w') as csv_file:
29         fieldnames = ["Chargebalance", dim1newname, amplitudefield, dim2newname]#, 'h_max']
30         writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
31         writer.writeheader()
32
33     for i in sorted(result):
34         for j in sorted(result[i]):
35             if result[i][j] != {}: # Empty dicts should be ignored
36                 result[i][j][dim1newname] = result[i][j].pop(dimname) # Needed to make compliant to
↳ fieldnames of CSV
37                 result[i][j][dim2newname] = result[i][j].pop(dim2name) # Needed to make compliant to
↳ fieldnames of CSV
38                 result[i][j][amplitudefield] = result[i][j].pop("Amplitude") # Needed to make
↳ compliant to fieldnames of CSV
39                 writer.writerow(result[i][j])
40
41
42 #####
43 # Find blockingamplitudes, sweep over frequency
44 #####
45 def find_blockingamplitudes_Freq(frequencylist, waveform):
46     config = 'ProposedModel'
47     par, recpar = selectConfig(config)
48
49     starttime = time.time()

```

```

50
51 amplitudelist = []
52 for i in range(100):
53     amplitudelist.append(round(0.1 + 0.01 * i, 2))
54
55 createMRGaxon(par)
56
57 result = {}
58 for f in frequencylist:
59     par['HFSfrequency'] = f
60     result[f] = {}
61     for a in amplitudelist:
62         rec = recordMRGaxon(recpar)
63         par['HFSamp'] = a
64
65         if waveform == "sine":
66             t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
67             ↪ par['HFSamp'], 1000)
68             cpp = a / (np.pi * f)
69         elif waveform == "square":
70             t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
71             ↪ par['HFSamp'])
72             cpp = a / (2*f)
73         elif waveform == "triangular":
74             t_signal, i_signal = trianglewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
75             ↪ par['HFSamp'])
76             cpp = a / (4 * f)
77         else:
78             print "Incompatible waveform selected!"
79             return None
80
81     updateMRGaxon(par)
82
83     print("--- RUN HFS-SIGNAL (%s kHz, %s mA) ---" % (f, a))
84     runMRGaxon(rec, t_signal, i_signal)
85
86     if is_blocked(rec, par):
87         result[f]["frequency"] = f
88         result[f]["amplitude"] = a
89         result[f]["chargeperphase"] = cpp
90         break
91
92 totaltime = time.time() - starttime
93 print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
94
95 return result
96
97 #####
98 # Find blockingamplitudes, sweep over frequency and chargebalance
99 #####
100 def find_blockingamplitudes_CB(frequencylist, chargebalancelist):
101     config = 'ProposedModel'
102     par, recpar = selectConfig(config)
103
104     starttime = time.time()
105
106     amplitudelist = []
107     for i in range(200):
108         amplitudelist.append(round(0.1 + 0.01 * i, 2))
109
110     createMRGaxon(par)
111
112     result = {}
113     for b in chargebalancelist:
114         result[b] = {}
115         for f in frequencylist:
116             par['HFSfrequency'] = f
117             result[b][f] = {}
118             for a in amplitudelist:
119                 rec = recordMRGaxon(recpar)
120                 par['HFSamp'] = a
121                 cpp = a/(2*f) # Charge per phase is amplitude times half the period
122                 t_signal, i_signal = chargebalanced_asymmetrical(par['HFSdelay'], par['tstop'],
123                 ↪ par['HFSfrequency'], cpp, b)
124                 updateMRGaxon(par)
125                 print("--- RUN HFS-SIGNAL (%s chargebalance, %s kHz, %s mA) ---" % (b, f, a))
126                 runMRGaxon(rec, t_signal, i_signal)

```

```

124
125         if is_blocked(rec, par):
126             result[b][f]["chargebalance"] = b
127             result[b][f]["frequency"] = f
128             result[b][f]["amplitude"] = a
129             result[b][f]["chargeperphase"] = cpp
130             break
131
132     totaltime = time.time() - starttime
133     print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
134
135     return result

```

reportplots.py

```

1  # Import Python libraries
2  import pylab as plt
3  import numpy as np
4  # Import for BrainFrame
5  import matplotlib
6  # Import custom function libraries
7  from MRG_Model import *           # Interface with Neuron MRG Model
8  from configoptions import *
9  from waveforms import *         # Waveform functions for Neuron
10 from plottingfunctions import *  # Plotting functions
11 from plottingfunctions_justification import *
12 from blockdetection import *
13 if not os.path.exists('reportplots'):
14     os.makedirs('reportplots')
15 if not os.path.exists('reportplots/background'):
16     os.makedirs('reportplots/background')
17 if not os.path.exists('reportplots/method'):
18     os.makedirs('reportplots/method')
19 if not os.path.exists('reportplots/implementation'):
20     os.makedirs('reportplots/implementation')
21 import time
22 from matplotlib.pyplot import figure
23
24 plt.rcParams.update({'font.size': 12})
25
26 #####
27 # PLOT 'JUSTIFICATION OF MODEL' PLOTS
28 #####
29 def justificationplots(select):
30     if(select == "square"):
31         result_classic = np.load("justification_files/results_squarewave_classicmethod.npy",
32             ↪ encoding="ASCII").item()
33         result_proposed = np.load("justification_files/results_squarewave_proposedmethod.npy",
34             ↪ encoding="ASCII").item()
35         plotjustification2D(result_classic, result_proposed)
36         plt.show()
37     elif(select == "sine"):
38         result_classic = np.load("justification_files/results_sinewave_classicmethod.npy",
39             ↪ encoding="ASCII").item()
40         result_proposed =
41             ↪ np.load("justification_files/results_sinewave_proposedmethod20190226_171930.npy",
42             ↪ encoding="ASCII").item()
43         plotjustification2Dcpp(result_classic, result_proposed)
44         plt.savefig("reportplots/method/performance_sine_cpp.eps", format='eps', dpi=300)
45         plt.show()
46         plotjustification2D(result_classic, result_proposed)
47         plt.savefig("reportplots/method/performance_sine.eps", format='eps', dpi=300)
48         plt.show()
49     elif(select == "squareCB"):
50         result_classic = np.load("justification_files/results_squarewave_CB_classicmethod.npy",
51             ↪ encoding="ASCII").item()
52         result_proposed = np.load("justification_files/results_squarewave_CB_proposedmethod.npy",
53             ↪ encoding="ASCII").item()
54         # result_proposed =
55             ↪ np.load("results_monopolar/results_monopolar_assymmetricalwave_allfrequencies.npy").item()
56         plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.1)#, freqname =
57             ↪ "frequency", chargename = "chargeperphase")#subplots")
58         plt.show()
59         plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.5)#, freqname =
60             ↪ "frequency", chargename = "chargeperphase")
61         plt.show()

```

```

52     plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.8)#, freqname =
53     ↪ "frequency", chargename = "chargeperphase")
54     plt.show()
55     plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.9)#, freqname =
56     ↪ "frequency", chargename = "chargeperphase")
57     plt.show()
58     else:
59         print "Type non-existent!"
60
61     #####
62     # VISUALIZATION OF SINGLE SIMULATION 2D
63     #####
64     def plotsquareWave():
65         return None
66     #     t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], 4, 0.5)
67     #     runandplotsignal(par, recpar, t_signal, i_signal, config)
68
69     #####
70     # DEMO SINGLE SIMULATION TIME-SPATIAL
71     #####
72     def plot_square_classic(f = 10, a = 0.47, tstop = 40.0, intrinsic = 0):
73         config = "ClassicModel"
74         par, recpar = selectConfig(config)
75         par['HFSfrequency'] = f # kHz
76         par['HFSamp'] = a # mA
77         par['tstop'] = tstop
78         if intrinsic == 0:
79             par['intrinsicStim'] = 0
80         else:
81             par['intrinsicStim'] = 1
82         t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
83         runandplotsignal(par, recpar, t_signal, i_signal, config)
84
85
86     def plot_sine_classic(f = 10, a = 0.47, tstop = 40.0, intrinsic = 0):
87         config = "ClassicModel"
88         par, recpar = selectConfig(config)
89         par['HFSfrequency'] = f # kHz
90         par['HFSamp'] = a # mA
91         par['tstop'] = tstop
92         if intrinsic == 0:
93             par['intrinsicStim'] = 0
94         else:
95             par['intrinsicStim'] = 1
96         HFSsamplesize = 1000 # Number of samples per period
97         t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'],
98         ↪ HFSsamplesize)
99         runandplotsignal(par, recpar, t_signal, i_signal, config)
100
101     def plot_asymmetric_classic(f = 10, a = 0.47, tstop = 40.0, chargebalance=0.5, intrinsic = 0):
102         config = "ClassicModel"
103         par, recpar = selectConfig(config)
104         par['HFSfrequency'] = f # kHz
105         par['HFSamp'] = a # mA
106         par['tstop'] = tstop
107         if intrinsic == 0:
108             par['intrinsicStim'] = 0
109         else:
110             par['intrinsicStim'] = 1
111         chargeperphase = par['HFSamp'] / (2 * par['HFSfrequency'])
112         t_signal, i_signal = chargebalanced_asymmetrical(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
113         ↪ chargeperphase, chargebalance)
114         runandplotsignal(par, recpar, t_signal, i_signal, config)
115
116     def plot_square_proposed(f, a):
117         config = "ProposedModel"
118         par, recpar = selectConfig(config)
119         par['HFSfrequency'] = f # kHz
120         par['HFSamp'] = a # mA
121         par['tstop'] = 20.0
122         t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
123         runandplotsignal(par, recpar, t_signal, i_signal, config)
124
125

```

```

126 def plot_sine_proposed(f,a):
127     config = "ProposedModel"
128     par, reconf = selectConfig(config)
129     par['HFSfrequency'] = f # kHz
130     par['HFSamp'] = a # mA
131     par['tstop'] = 20.0
132     HFSsamplesize = 1000 # Number of samples per period
133     t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'],
134     ~ HFSsamplesize)
135     runandplotsignal(par, reconf, t_signal, i_signal, config)
136
137 def plot_asyymmetric_proposed(f, a, chargebalance):
138     config = "ProposedModel"
139     par, reconf = selectConfig(config)
140     par['HFSfrequency'] = f # kHz
141     par['HFSamp'] = a # mA
142     par['tstop'] = 20.0
143     chargeperphase = par['HFSamp'] / (2 * par['HFSfrequency'])
144     t_signal, i_signal = chargebalanced_asyymetrical(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
145     ~ chargeperphase, chargebalance)
146     runandplotsignal(par, reconf, t_signal, i_signal, config)
147
148 def plot_square_proposed_bipolar(f, a):
149     config = "ProposedModel_Bipolar"
150     par, reconf = selectConfig(config)
151     par['HFSfrequency'] = f # kHz
152     par['HFSamp'] = a # mA
153     par['tstop'] = 20.0
154     t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
155     runandplotsignal(par, reconf, t_signal, i_signal, config)
156
157
158 #####
159 # BACKGROUND CHAPTER
160 #####
161 def plotAP():
162     config = "singleAPdemo"
163     par, reconf = selectConfig(config)
164     a = 0
165     par['HFSreferenceNode'] = 2.0
166     par['HFSamp'] = a
167     par['tstop'] = 150.0
168     par['dt'] = 0.005
169     t_signal, i_signal = [0],[0]
170     createMRGaxon(par)
171     rec = recordMRGaxon(reconf)
172     updateMRGaxon(par)
173     runMRGaxon(rec, t_signal, i_signal)
174     plotMRGgatessinglenodeAP(plt, rec, 10)
175     plt.savefig("reportplots/background/normalAP.eps", format='eps', dpi=1200)
176     plt.show()
177
178     par['tstop'] = 10.0
179     createMRGaxon(par)
180     rec = recordMRGaxon(reconf)
181     updateMRGaxon(par)
182     runMRGaxon(rec, t_signal, i_signal)
183     plotMRGgatessinglenodeAP(plt, rec, 10)
184     plt.savefig("reportplots/background/normalAP_zoom.eps", format='eps', dpi=1200)
185     plt.show()
186
187
188 def plotKHFAcdemogates():
189     config = "HFSdemo"
190     par, reconf = selectConfig(config)
191     t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
192     createMRGaxon(par)
193     rec = recordMRGaxon(reconf)
194     updateMRGaxon(par)
195     runMRGaxon(rec, t_signal, i_signal)
196     plotMRGgatessingletime(plt,rec,20)
197     plt.savefig("reportplots/background/HFSexample_gatesallnodes_T20ms.eps", format='eps', dpi=1200)
198     plt.show()
199     plotMRGgatessinglenode(plt,rec,26)
200     plt.savefig("reportplots/background/HFSexample_virtualanodes.eps", format='eps', dpi=1200)
201     plt.show()

```

```

202 plotMRGgatessinglenode(plt, rec, 10)
203 plt.savefig("reportplots/background/HFSexample_onsetresponse.eps", format='eps', dpi=1200)
204 plt.show()
205
206
207 def plothinftau():
208     Vm = np.arange(-150,50,1)
209     Vm = np.delete(Vm, np.where(Vm == -114))
210     alpha_h = [(0.34 * (-(v + 114))) / (1.0 - np.exp((v + 114) / 11.0))] for v in Vm]
211     beta_h = [12.6 / (1.0 + np.exp(-(v + 31.8) / 13.4))] for v in Vm]
212     print alpha_h
213     h_inf = [a / (a + b) for a,b in zip(alpha_h, beta_h)]
214     tau_h = [1.0 / (a + b) for a,b in zip(alpha_h, beta_h)]
215
216     alpha_m = [(6.57 * (v + 20.4)) / (1 - np.exp(-(v + 20.4) / 10.3))] for v in Vm]
217     beta_m = [(0.304 * (-(v + 25.7))) / (1 - np.exp((v + 25.7) / 9.16))] for v in Vm]
218     m_inf = [a / (a + b) for a,b in zip(alpha_m, beta_m)]
219     tau_m = [1.0 / (a + b) for a,b in zip(alpha_m, beta_m)]
220
221     plt.rcParams.update({'font.size': 14})
222     plt.figure(figsize=[15,5])
223     plt.plot(Vm, h_inf, label='$h_{\infty}$')
224     plt.plot(Vm, m_inf, label='$m_{\infty}$')
225     plt.ylabel('State value')
226     plt.xlabel('Membrane voltage (mV)')
227     plt.legend()
228     plt.xlim(-150,50)
229     plt.xticks(np.arange(-150,60,20))
230     plt.grid()
231     plt.tight_layout()
232     plt.savefig("reportplots/background/vm_gatevaluedependence_infty.eps", format='eps', dpi=1200)
233     plt.show()
234
235     plt.figure(figsize=[15, 5])
236     plt.plot(Vm, tau_h, label='$\tau_h$')
237     plt.plot(Vm, tau_m, label='$\tau_m$')
238     plt.ylabel('Time constant (ms)')
239     plt.xlabel('Membrane voltage (mV)')
240     plt.legend()
241     plt.xlim(-150, 50)
242     plt.yticks(np.arange(0,1.2,0.2))
243     plt.xticks(np.arange(-150, 60, 20))
244     plt.grid()
245     plt.tight_layout()
246     plt.savefig("reportplots/background/vm_gatevaluedependence_tau.eps", format='eps', dpi=1200)
247     plt.show()
248
249
250 #####
251 # CHAPTER DETERMINING THE EFFECTIVENESS OF A BLOCK
252 #####
253 def plot_2D_blockdescription():
254     config = "ClassicModel"
255     par, recpar = selectConfig(config)
256     f = 10
257     a = 0.6
258     HFSsamplesize = 1000
259     par['HFSfrequency'] = f # kHz
260     par['HFSamp'] = a # mA
261     par['tstop'] = 20.0
262     par['intrinsicDel'] = 15.0
263     par['intrinsicDur'] = 0.1
264     par['intrinsicType'] = 0
265     t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'],
266     ↪ HFSsamplesize)
267     createMRGaxon(par)
268     rec = recordMRGaxon(recpar)
269     updateMRGaxon(par)
270     runMRGaxon(rec, t_signal, i_signal)
271     # plotMRGaxon(rec)
272     dt = par['dt']
273     t = [15.0, 15.2, 15.4, 15.6, 15.8, 16.0]
274     for i in t:
275         figure(num=None, figsize=(15, 3), dpi=1200, facecolor='w', edgecolor='k')
276         plotnodeVoltageessingletime(rec, int(i/dt), plt)
277         plt.tight_layout()
278         plt.savefig("reportplots/method/KHFACdemo_" + str(i) + ".eps", format='eps', dpi=1200)
279     return plt

```



```

279
280
281 def plot_3D_blockdescription():
282     config = "ClassicModel"
283     par, recpar = selectConfig(config)
284     f = 20
285     HFSsamplesize = 1000
286     par['HFSfrequency'] = f # kHz
287     par['tstop'] = 40.0
288     par['intrinsicDel'] = 20.0
289
290     a = [0.53, 0.6]
291     intrinsictype = [0, 1]
292     createMRGaxon(par)
293     for i in intrinsictype:
294         for j in a:
295             par['intrinsicType'] = i
296             par['HFSamp'] = j # mA
297             t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
298             ↪ par['HFSamp'], HFSsamplesize)
299             rec = recordMRGaxon(recpar)
300             updateMRGaxon(par)
301             runMRGaxon(rec, t_signal, i_signal)
302             fig = plotMRGaxon(rec)
303             plt.tight_layout()
304             plt.savefig("reportplots/method/KHFACdemo3D_pulse_" + str(i) + "_amp_" + str(j) + ".eps",
305             ↪ format='eps', dpi=1200)
306
307     return plt
308
309 def plot_gates_proposedmethodsetup():
310     config = "ProposedModel"
311     par, recpar = selectConfig(config)
312     f = 10
313     a = 0.6
314     HFSsamplesize = 1000
315     par['HFSfrequency'] = f # kHz
316     par['HFSamp'] = a # mA
317     t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'],
318     ↪ HFSsamplesize)
319     createMRGaxon(par)
320     rec = recordMRGaxon(recpar)
321     updateMRGaxon(par)
322     runMRGaxon(rec, t_signal, i_signal)
323     # plotMRGaxon(rec)
324     plotMRGgatesinglenode(plt, rec, par['HFSreferenceNode'] + 1, s=False)
325     plt.savefig("reportplots/method/proposedsetupsimulation_virtualanode.eps", format='eps', dpi=1200)
326     plt.show()
327     return plt
328
329 def plot_asymmetrical_waveform():
330     t_start = 0.0
331     f = 10
332     a = 1.0
333     t_stop = 1.0/f
334     chargebalance = [0.2, 0.5, 0.8]
335     cpp = a/(2*f)
336     for i in chargebalance:
337         t_signal = [0]
338         i_signal = [0]
339         t_result, i_result = chargebalanced_asymmetrical(t_start, t_stop, f, cpp, i)
340         t_signal.extend(t_result.tolist())
341         i_signal.append(max(t_result))
342         i_signal.extend(i_result.tolist())
343         i_signal.append(0)
344         plt.plot(t_signal, i_signal, label=i, linewidth=3)
345         # plot_signal(plt, t_signal, i_signal, label = i)
346         # plt.axhline(color='k', lw=1.0)
347         plt.xlabel('Time (ms)')
348         plt.ylabel('Current (mA)')
349         plt.ylim(-3,3)
350         plt.yticks(np.arange(-3.0,3.5,0.5))
351         plt.grid()
352         plt.legend(title="Anode fraction")
353         plt.tight_layout()
354         plt.savefig("reportplots/method/asymmetrical_waveform.eps", format='eps', dpi=300)
355         plt.show()

```

```

354     return plt
355
356
357 #####
358 # RESULTS CHAPTER EXTRA PLOTS
359 #####
360 def plot_ipdbasicimplementation(ipd1,ipd2):
361     t_start = 0.0
362     f = 10
363     a = 1.0
364     t_stop = 1.0/f
365     t_result, i_result = squarewave(t_start, t_stop, f, a, ipd1, ipd2)
366     t_signal = [0]
367     i_signal = [0]
368     t_signal.extend(t_result.tolist())
369     t_signal.append(max(t_result))
370     i_signal.extend(i_result.tolist())
371     i_signal.append(0)
372     plt.xlabel('Time (ms)')
373     plt.ylabel('Current (mA)')
374     plt.xticks(np.arange(0,0.11,0.01))
375     plt.title("Frequency = 10 kHz")
376     plt.grid()
377     plt.xlim(-0.01,0.11)
378     plt.plot(t_signal, i_signal, linewidth=3, color="C0")
379     plt.tight_layout()
380     plt.savefig("reportplots/ipdbasic_"+str(ipd1)+"_"+str(ipd2)+".eps", format='eps', dpi=300)
381     plt.show()
382
383
384 def plot_ipdcompleximplementation():
385     t_start = 0.0
386     f = 10
387     a = 1.0
388     t_stop = 1.0/f
389     ipds = [[0.2,0.3],[0.4,0.1],[0.25,0.25]]
390     plt.figure(figsize=(15,4))
391     for i in range(0,len(ipds)):
392         ipd1, ipd2 = ipds[i]
393         t_signal = [0+i*t_stop]
394         i_signal = [0]
395         t_result, i_result = squarewave(t_start, t_stop, f, a, ipd1, ipd2)
396         t_result = [x + i*t_stop for x in t_result]
397         t_signal.extend(t_result)
398         i_signal.extend(i_result.tolist())
399         i_signal.append(0)
400         t_signal.append(max(t_result))
401     plt.plot(t_signal, i_signal, linewidth=3)
402     plt.xlabel('Time (ms)')
403     plt.ylabel('Current (mA)')
404     plt.xticks(np.arange(0,0.31,0.02))
405     plt.xlim(-0.01,0.31)
406     plt.title("Frequency = 10 kHz")
407     plt.grid()
408     plt.tight_layout()
409     plt.savefig("reportplots/ipdcomplex.eps", format='eps', dpi=300)
410     plt.show()
411
412
413 def plot_asymmetrical_bipolar(cpp):
414     # par, recpar = selectConfig("ClassicModel")
415     par,recpar = selectConfig("ClassicModel_Bipolar")
416     par['tstop'] = 20.0
417     par['intrinsicStim'] = 0
418     par['HFSy2'] = 2200
419     t_signal, i_signal = chargebalanced_asymmetrical(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
420     ↪     cpp, 0.7)
421     # plot_signal(plt, t_signal, i_signal)
422     # plt.show()
423     createMRGaxon(par)
424     rec = recordMRGaxon(recpar)
425     updateMRGaxon(par)
426     runMRGaxon(rec, t_signal, i_signal)
427     fig = plotMRGaxon(rec)
428     plt.show()
429
430 def plot_ipd_validation(amp, ipd1,ipd2):

```

```

431 par, recpar = selectConfig("ClassicModel")
432 par['tstop'] = 20.0
433 t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], amp, ipd1, ipd2)
434 # plot_signal(plt, t_signal, i_signal)
435 # plt.show()
436 createMRGaxon(par)
437 rec = recordMRGaxon(recpar)
438 updateMRGaxon(par)
439 runMRGaxon(rec, t_signal, i_signal)
440 fig = plotMRGaxon(rec)
441 plt.tight_layout()
442 plt.show()
443
444 #####
445 # RUN AND PLOT A SINGLE SIMULATION
446 #####
447 def runandplotsignal(par, recpar, t_signal, i_signal, config):
448     starttime = time.time()
449
450     createMRGaxon(par)
451     intervaltime = time.time() - starttime
452     print("--- CREATEAXON: %s seconds ---" % (intervaltime))
453     intervaltime = time.time()
454
455     rec = recordMRGaxon(recpar)
456     updateMRGaxon(par)
457     intervaltime = time.time() - intervaltime
458     print("--- PREPAREAXON: %s seconds ---" % (intervaltime))
459     intervaltime = time.time()
460
461     runMRGaxon(rec, t_signal, i_signal)
462     intervaltime = time.time() - intervaltime
463     print("--- RUNAXON: %s seconds ---" % (intervaltime))
464     intervaltime = time.time()
465
466     fig = plotMRGaxon(rec)
467
468     # plotMRGgatessingletime(plt, rec, int(par['tstop']/par['dt']))
469
470     # Plot single AP
471     if config=='singleAPdemo':
472         plotMRGgatessinglenode(plt, rec, 4)
473
474
475     # # Plot single KHfAC signal
476     if config=='HFSdemo':
477         plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']))
478         plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']) + 1)
479         # Onset response
480         plotMRGgatessinglenode(plt, rec, int(par['axonnodes']-1))
481
482     if config == 'ClassicModel':
483         plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']))
484         plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']) + 1)
485
486     if config == 'ProposedModel' or config == 'ProposedModel_Bipolar':
487         plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']))
488         plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']) + 1)
489         plotMRGgatessinglenode_timeframe(plt, rec, int(par['HFSreferenceNode']), 19/par['dt'])
490         plotMRGgatessinglenode_timeframe(plt, rec, int(par['HFSreferenceNode']) + 1, 19/par['dt'])
491
492
493     intervaltime = time.time() - intervaltime
494     print("--- PLOT: %s seconds ---" % (intervaltime))
495     plt.show()
496
497
498 #####
499 # Parameters for configuration of the simulation
500 #####
501 # # config = 'singleAPdemo'
502 # # config = 'HFSdemo'
503 # # config = 'HFSModel'
504 # config = 'ClassicModel'
505 # par, recpar = selectConfig(config)
506 # verbose = False
507 #
508 #

```

```

509 # t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'], 1000)
510 # testandplotsignal(par, recpar, verbose, t_signal, i_signal, config)
511 #
512 # h.quit()
513
514
515 #####
516 # Presentation plots
517 #####
518 def buildKHFAcgif():
519     config = "ClassicModel"
520     par, recpar = selectConfig(config)
521     par['HFSfrequency'] = 10
522     par['HFSamp'] = 0.5
523     par['tstop'] = 40
524     par['intrinsicDel'] = 15.0
525     t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
526     createMRGaxon(par)
527     rec = recordMRGaxon(recpar)
528     updateMRGaxon(par)
529     runMRGaxon(rec, t_signal, i_signal)
530
531     dt = par['dt']
532     tstart = int((par['HFSdelay'] + par['tonset']) / dt)
533
534     for i in range(0, par['tstop'] * 10):
535         t = i/10.0
536         plt.figure()
537         plotnodeVoltageessingletime(rec, int(t/dt), plt)
538         filename = str(int(t*10))
539         plt.savefig('figures/' + filename + '.png', bbox_inches='tight')
540         plt.close()

```

plottingfunctions_results.py

```

1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
3 from matplotlib import cm
4 import numpy as np
5 # plt.style.use('seaborn-dark-palette')
6
7 #####
8 # MONOPOLAR SIMULATIONS
9 #####
10 def plot_results_monopolar_ipd(data, type, plottype, validation=False):
11     datascalar = 1
12     if type == "amplitude":
13         datatype = 'amplitude'
14         datalabel = "Block threshold (mA)"
15         ylim = 1.0
16     elif type == "cpp":
17         datatype = 'chargeperphase'
18         datascalar = 1000
19         datalabel = "Block threshold charge per phase ( $\mu$ C)"
20         ylim = 0.02
21
22     if plottype == "lines":
23         count = 0
24         colmulti = 1
25         if validation:
26             colmulti = 2
27         for i in sorted(data.keys()):
28             if 20*i % 2 == 0 and i != 0.9:
29                 x = sorted(data[i].keys())
30                 x = [n / 10 for n in x]
31                 y = []
32                 for j in sorted(data[i].keys()):
33                     if data[i][j] != {}:
34                         y = np.append(y, data[i][j][datatype]*datascalar)
35                     else:
36                         y = np.append(y, np.nan)
37                 line, = plt.plot(x,y,label=str(i/10) + "ms", color="C"+str(count*colmulti))
38                 count+=1
39     plt.xlabel("Cathodal interphase delay (ms)")
40     plt.ylabel(datalabel)
41     plt.ylim(bottom=0, top=ylim)

```

```

42     plt.xlim(0,0.09)
43     if type == "amplitude":
44         plt.yticks(np.arange(0, 1.1, step=0.1))
45     plt.xticks(np.arange(0, 0.09, step=0.01))
46     plt.grid(True)
47     plt.legend(title = "Anodal interphase delay")
48
49 elif plottype == "colour" or plottype == "3D":
50     ipd1 = []
51     ipd2 = []
52     for i in sorted(data.keys()):
53         ipd1 = np.append(ipd1, i)
54         for j in sorted(data[i].keys()):
55             if j not in ipd2:
56                 ipd2=np.append(ipd2, j)
57     matrix = np.zeros((len(ipd1), len(ipd2)))
58     for i in range(len(ipd1)):
59         for j in range(len(ipd2)):
60             if data[ipd1[i]][ipd2[j]] != {}:
61                 matrix[i][j] = data[ipd1[i]][ipd2[j]][datatype]*datascalar
62             else:
63                 matrix[i, j] = np.nan
64
65     # Set up a regular grid of interpolation points
66     ipd1 = [x / 10 for x in ipd1]
67     ipd2 = [x / 10 for x in ipd2]
68     X, Y = np.meshgrid(ipd1, ipd2)
69
70     if plottype == "colour":
71         plt.pcolormesh(X, Y, matrix.T, cmap=cm.viridis_r)
72         clb = plt.colorbar()
73         clb.set_label(datalabel)
74         # clb.ax.set_title(datalabel)
75         plt.xlabel("Anodal interphase delay (ms)")
76         plt.ylabel("Cathodal interphase delay (ms)")
77     elif plottype == "3D":
78         ax = plt.gca(projection='3d')
79         ax.plot_surface(X, Y, matrix.T, cmap=cm.viridis_r, linewidth=0, antialiased=False,
80             → vmin=np.nanmin(matrix), vmax=np.nanmax(matrix))
81         ax.set_xlabel("Anodal interphase delay (ms)")
82         ax.set_ylabel("Cathodal interphase delay (ms)")
83         ax.set_zlabel(datalabel)
84         ax.view_init(15, -160)
85     title = ""
86     for i in data:
87         for j in data[i]:
88             if data[i][j] != {}:
89                 title = "Frequency = " + str(data[i][j]["frequency"]) + "kHz"
90                 break
91         if title != "":
92             break
93     plt.title(title)
94     plt.tight_layout()
95     return plt
96
97 def plot_results_monopolar_basicwaveforms(squarereres, sineres, triangularres, type):
98     frequencies = []
99     amps_square = []
100    amps_sine = []
101    amps_triangle = []
102    resvar = ""
103    xlabelvar = ""
104    if type == "amplitude":
105        resvar = "Amplitude"
106        ylabelvar = "Block threshold (mA)"
107        ylim = 1.0
108    elif type == "cpp":
109        resvar = "Chargeperphas"
110        ylabelvar = "Block threshold charge per phase ( $\mu\text{C}$ )"
111        ylim = 0.05
112    else:
113        print "Incorrect type"
114        return
115    for i in sorted(squarereres):
116        frequencies.append(i)
117        amps_square.append(squarereres[i][resvar])
118    for i in sorted(sineres):

```

```

119     amps_sine.append(sineres[i][resvar])
120 for i in sorted(triangularres):
121     amps_triangle.append(triangularres[i][resvar])
122 plt.plot(frequencies,amps_square, label="Square")
123 plt.plot(frequencies,amps_sine, label="Sine")
124 plt.plot(frequencies,amps_triangle, label="Triangular")
125 plt.xlabel("KHFAC frequency (kHz)")
126 plt.ylabel(ylabelvar)
127 plt.legend(title="Waveform type")
128 plt.ylim(0, ylim)
129 plt.yticks(np.arange(0,ylim+ylim/10,ylim/10))
130 plt.xlim(left=0, right=max(frequencies)+5)
131 plt.xticks(np.arange(0,max(frequencies)+5,5))
132 plt.grid()
133 plt.tight_layout()
134 return plt
135
136
137 def plt_results_monopolar_stepwaveforms(stepsineres, steptriangleres, type):
138     if type == "amplitude":
139         resvar = "amplitude"
140         ylabelvar = "Block threshold (mA)"
141         ylim = 1.0
142     elif type == "cpp":
143         resvar = "chargeperphase"
144         ylabelvar = "Block threshold charge per phase ( $\mu\text{C}$ )"
145         ylim = 0.05
146     steps = []
147     amps_sine = []
148     amps_triangle = []
149     for i in sorted(stepsineres):
150         steps.append(i)
151         amps_sine.append(stepsineres[i][resvar])
152     for i in sorted(steptriangleres):
153         amps_triangle.append(steptriangleres[i][resvar])
154     steps = [x * 4 for x in steps]
155     plt.plot(steps, amps_sine, label="Stepped sine", color="orange")
156     plt.plot(steps, amps_triangle, label="Stepped triangular", color="green")
157     plt.xlabel("Steps per period")
158     plt.ylabel(ylabelvar)
159     plt.legend(title="Waveform type")
160     plt.ylim(0, ylim)
161     plt.yticks(np.arange(0,ylim+ylim/10,ylim/10))
162     plt.xlim(left=0, right=max(steps)+20)
163     plt.xticks(np.arange(0,max(steps)+5,20))
164     plt.grid()
165     plt.tight_layout()
166     return plt
167
168
169 def plt_results_monopolar_realdistance(data, type):
170     if type == "amplitude":
171         resvar = "amplitude"
172         ylabelvar = "Block threshold (mA)"
173         ylim = 30.0
174     elif type == "cpp":
175         resvar = "chargeperphase"
176         ylabelvar = "Block threshold charge per phase ( $\mu\text{C}$ )"
177         ylim = 1.5
178     distances = []
179     amps = []
180     for i in sorted(data):
181         distances.append(i/1000.0)
182         amps.append(data[i][resvar])
183     plt.plot(distances, amps)#, label="Stepped sine")
184     plt.xlabel("Electrode perpendicular distance (mm)")
185     plt.ylabel(ylabelvar)
186     plt.title("KHFAC frequency = 10kHz")
187     plt.grid(True)
188     # plt.legend()
189     plt.ylim(bottom = 0, top = ylim)
190     plt.xlim(left = 0, right = 6)
191     plt.tight_layout()
192     return plt
193
194
195 def plot_results_asymmetrical(data, type, plotype, plotfreq):
196     datascalar = 1

```

```

197     if type == "amplitude":
198         datatype = "amplitude"
199         datalabel = "Peak current amplitude at block threshold (mA)"
200         # ylim = 30.0
201     elif type == "cpp":
202         datatype = "chargeperphase"
203         datascalar = 1000
204         datalabel = "Block threshold charge per phase ( $\mu\text{C}$ )"
205         # ylim = 30.0
206
207     if plottype == "colour" or plottype == "3D":
208         cb = []
209         f = []
210         for i in sorted(data.keys()):
211             cb.append(i)
212             for j in sorted(data[i].keys()):
213                 if j not in f:
214                     f.append(j)
215         matrix = np.zeros((len(cb), len(f)))
216         for i in range(len(cb)):
217             for j in range(len(f)):
218                 if data[cb[i]][f[j]] != {}:
219                     if datatype == "amplitude":
220                         datascalar = 0.5 / np.min([cb[i], 1.0-cb[i]])
221                         matrix[i][j] = data[cb[i]][f[j]][datatype] * datascalar
222                     else:
223                         matrix[i, j] = np.nan
224         X, Y = np.meshgrid(cb, f)
225         if plottype == "colour":
226             plt.pcolormesh(X, Y, matrix.T, cmap=cm.viridis_r)
227             clb = plt.colorbar()
228             clb.set_label(datalabel)
229             plt.xlabel("Anode fraction")
230             plt.ylabel("Frequency (kHz)")
231         elif plottype == "3D":
232             ax = plt.gca(projection='3d')
233             ax.plot_surface(X, Y, matrix.T, cmap=cm.viridis_r, linewidth=0, antialiased=False,
234                             vmin=np.nanmin(matrix), vmax=np.nanmax(matrix))
235             ax.set_xlabel("Anode fraction")
236             ax.set_ylabel("Frequency (kHz)")
237             ax.set_zlabel(datalabel)
238             ax.view_init(15, -160)
239     elif plottype == "singlefrequency":
240         cb = []
241         maxamp = []
242         cpp = []
243         for i in sorted(data.keys()):
244             cb.append(i)
245             if data[i][plotfreq] != {}:
246                 maxamp.append(data[i][plotfreq]['amplitude']*0.5/(np.min([i, 1.0-i])))
247                 cpp.append(data[i][plotfreq]['chargeperphase'])
248         fig, ax1 = plt.subplots(figsize=(8,5))
249         cb = [n/plotfreq for n in cb]
250         color = 'CO'
251         ax1.set_xlabel("$T_{anode}$ (ms)")
252         ax1.set_ylabel("Peak current amplitude at block threshold (mA)", color=color)
253         ax1.tick_params(axis='y', labelcolor=color)
254         ax1.plot(cb, maxamp, color=color)
255         ax1.set_ylim(bottom=0, top=4.0)
256         ax1.set_xlim(left=0, right=0.1)
257         ax1.set_xticks(np.arange(0.01, 0.1, step=0.01))
258
259         ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis
260         color = 'r'
261         ax2.set_ylabel("Block threshold charge per phase ( $\mu\text{C}$ ", color=color) # we already handled
262             the x-label with ax1
263         ax2.plot(cb, cpp, color=color)
264         ax2.tick_params(axis='y', labelcolor=color)
265         ax2.set_ylim(bottom=0, top=0.04)
266         fig.tight_layout()
267         plt.grid(axis="x")
268         plt.title("KHfAC frequency = " + str(plotfreq) + "kHz")
269         ax1.grid()
270     plt.tight_layout()
271     return plt
272
273 #####
274 # BIPOLAR SIMULATIONS

```

```

273 #####
274 def plot_results_bipolar_ETA_IEC_parallel(data, type, plottype):
275     # fig = plt.figure()
276     datascalar = 1
277     if type == "amplitude":
278         datatype = 'amplitude'
279         datalabel = "Block threshold (mA)"
280     elif type == "cpp":
281         datatype = 'chargeperphase'
282         datascalar = 1000
283         datalabel = "Block threshold charge per phase ( $\mu\text{C}$ )"
284
285     if plottype == "lines":
286         for i in sorted(data.keys(), reverse=True):
287             x = sorted(data[i].keys())
288             x = [n / 1000.0 for n in x] # convert to mm
289             y = []
290             for j in sorted(data[i].keys()):
291                 if data[i][j] != {}:
292                     y = np.append(y, data[i][j][datatype]*datascalar)
293                 else:
294                     y = np.append(y, np.nan)
295             plt.plot(x,y,label=str(i) + " $\mu\text{m}$ ")
296         plt.xlabel("Interpolar distance (mm)")
297         plt.ylabel(datalabel)
298         plt.legend(title = "Electrode to axon distance", loc="upper right")
299         plt.grid()
300
301     elif plottype == "colour" or plottype == "3D":
302         ax1 = []
303         ax2 = []
304         for i in sorted(data.keys()):
305             ax1.append(i)
306             for j in sorted(data[i].keys()):
307                 if j not in ax2:
308                     ax2.append(j)
309         matrix = np.zeros((len(ax1), len(ax2)))
310         for i in range(len(ax1)):
311             for j in range(len(ax2)):
312                 if data[ax1[i]][ax2[j]] != {}:
313                     matrix[i,j] = data[ax1[i]][ax2[j]][datatype] * datascalar
314                 else:
315                     matrix[i,j] = np.nan
316         # ax1 = [x / 1000 for x in ax1]
317         ax2 = [x / 1000.0 for x in ax2]
318         # Set up a regular grid of interpolation points
319         X, Y = np.meshgrid(ax1, ax2)
320
321         if plottype == "colour":
322             plt.pcolormesh(Y, X, matrix.T, cmap=cm.viridis_r)
323             clb = plt.colorbar()
324             clb.set_label(datalabel)
325             plt.xlabel("Interpolar distance (mm)")
326             plt.ylabel("Electrode to axon distance ( $\mu\text{m}$ )")
327         elif plottype == "3D":
328             ax = plt.gca(projection='3d')
329             ax.plot_surface(X, Y, matrix.T, cmap=cm.viridis_r, linewidth=0, antialiased=False,
330                 vmin=np.nanmin(matrix),
331                 vmax=np.nanmax(matrix))
332             ax.set_xlabel("Electrode to axon distance ( $\mu\text{m}$ )")
333             ax.set_ylabel("Interpolar distance (mm)")
334             ax.set_zlabel(datalabel)
335             ax.view_init(15, -160)
336
337     elif plottype == "optimal":
338         x = sorted(data.keys()) # x-axis is electrode to axon distance
339         bestamplitudes = []
340         bestcontactdistances = []
341         for i in x:
342             amplitudes = []
343             contactdistances = []
344             for j in data[i]:
345                 if data[i][j] != {}:
346                     amplitudes.append(data[i][j][datatype]*datascalar)
347                     contactdistances.append(j)
348             if amplitudes != []:
349                 bestamplitudes.append(np.min(amplitudes))

```



```

350         bestcontactdistances.append(contactdistances[amplitudes.index(np.min(amplitudes))])
351     else:
352         bestamplitudes.append(np.nan)
353         bestcontactdistances.append(np.nan)
354
355     bestcontactdistances = [n / 1000.0 for n in bestcontactdistances] # convert to mm
356     fig, ax1 = plt.subplots()
357     color = 'CO'
358     ax1.set_xlabel("Electrode to axon distance ( $\mu\text{m}$ )")
359     ax1.set_ylabel("Lowest block threshold (mA)", color=color)
360     ax1.plot(x, bestamplitudes, color=color)
361     ax1.set_ylim(bottom=0, top=12)
362     ax1.tick_params(axis='y', labelcolor=color)
363     ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis
364     color = 'r'
365     ax2.set_ylabel('Optimal interpolator distance (mm)', color=color) # we already handled the x-label
366     # with ax1
367     ax2.plot(x, bestcontactdistances, color=color)
368     ax2.set_ylim(bottom=0, top=6)
369     ax2.tick_params(axis='y', labelcolor=color)
370     ax1.set_yticks(np.arange(0, 12, step = 1))
371     ax1.grid()
372     ax2.set_yticks(np.arange(0, 6, step = 1))
373     ax2.grid()
374
375     fig.tight_layout()
376
377     plt.title("KHFAC frequency = 10kHz")
378     plt.tight_layout()
379     return plt
380
381 def plot_results_bipolar_IEC_parallelperp(datapar, dataperp, type):
382     datascalar = 1
383     if type == "amplitude":
384         datatype = 'amplitude'
385         datalabel = "Block threshold (mA)"
386         ylim = 0.8
387     elif type == "cpp":
388         datatype = 'chargeperphase'
389         datascalar = 1000
390         datalabel = "Block threshold charge per phase ( $\mu\text{C}$ )"
391         ylim = 0.04
392     x = sorted(datapar.keys())
393     x = x[:x.index(25000)]
394
395     ypar = []
396     yperp = []
397     for i in x:
398         if datapar[i] != {}:
399             ypar.append(datapar[i][datatype]*datascalar)
400         else: ypar.append(np.nan)
401         if dataperp[i] != {}:
402             yperp.append(dataperp[i][datatype]*datascalar)
403         else: yperp.append(np.nan)
404     x = [n / 1000.0 for n in x]
405     plt.plot(x, ypar, label = "parallel")
406     plt.plot(x, yperp, label = "perpendicular")
407     plt.xlabel("Interpolator distance (mm)")
408     plt.ylabel(datalabel)
409     plt.ylim(bottom=0, top=ylim)
410     plt.xlim(0,25)
411     plt.legend(title="Electrode orientation")
412     plt.title("Frequency 10kHz, electrode 1000 $\mu\text{m}$  above axon")
413     plt.tight_layout()
414     plt.grid()
415     return plt
416
417 def plot_results_bipolar_orientation(type, result_x, result_z, plotype="lines", IECdist=2200):
418     datascalar = 1
419     if type == "amplitude":
420         datatype = 'amplitude'
421         datalabel = "Block threshold (mA)"
422     elif type == "cpp":
423         datatype = 'chargeperphase'
424         datascalar = 1000
425         datalabel = "Block threshold charge per phase ( $\mu\text{C}$ )"
426     data = result_z

```

```

427
428     if plottype == "lines":
429         angles = []
430         amps_x = []
431         amps_z = []
432         for i in sorted(result_x.keys()):
433             angles.append(i)
434             amps_x.append(result_x[i][IECdist][datatype])
435             amps_z.append(result_z[i][IECdist][datatype])
436         plt.plot(angles, amps_x, label="Electrode in $xy$-plane")
437         plt.plot(angles, amps_z, label="Electrode in $yz$-plane")
438         plt.xlabel("Angle (degrees)")
439         plt.ylabel(datalabel)
440         plt.ylim(bottom=0, top=1.0)
441         plt.yticks(np.arange(0, 1, step=0.1))
442         plt.xticks(np.arange(0, 90, step=10))
443         plt.grid(True)
444         plt.legend(title="Electrode orientation")
445         plt.title("Frequency 10kHz, anode 1000 $\mu$ m above axon")
446
447     elif plottype == "colour":
448         ax1 = []
449         ax2 = []
450         for i in sorted(data.keys()):
451             ax1.append(i)
452             for j in sorted(data[i].keys()):
453                 if j not in ax2:
454                     ax2.append(j)
455         matrix = np.zeros((len(ax1), len(ax2)))
456         for i in range(len(ax1)):
457             for j in range(len(ax2)):
458                 if data[ax1[i]][ax2[j]] != {}:
459                     matrix[i, j] = data[ax1[i]][ax2[j]][datatype] * datascalar
460                 else:
461                     matrix[i, j] = np.nan
462         # ax1 = [x / 1000 for x in ax1]
463         ax2 = [x / 1000.0 for x in ax2]
464         # Set up a regular grid of interpolation points
465         X, Y = np.meshgrid(ax1, ax2)
466
467         plt.pcolormesh(Y, X, matrix.T, cmap=cm.viridis_r)
468         clb = plt.colorbar()
469         clb.set_label(datalabel)
470         plt.xlabel("Interpolar distance (mm)")
471         plt.ylabel("Angle (degrees)")
472     plt.tight_layout()
473     return plt

```

plottingfunctions_justification.py

```

1  import pylab as plt
2  import numpy as np
3
4  #####
5  # VISUALIZATION OF PROPOSED METHOD RESULTS VERSUS CLASSIC METHOD RESULTS
6  #####
7  # Plots '2 dimensional' results
8  def plotjustification2D(classicres, proposedres):
9      amplitudes = {}
10     for i in classicres:
11         for j in classicres[i]:
12             for k in classicres[i][j]:
13                 if classicres[i][j][k] != {}:
14                     if classicres[i][j][k]["Frequency"] in amplitudes:
15                         if classicres[i][j][k]["Amplitude"] >
16                             amplitudes[classicres[i][j][k]["Frequency"]]:
17                             amplitudes[classicres[i][j][k]["Frequency"]] =
18                                 classicres[i][j][k]["Amplitude"]
19                     else:
20                         amplitudes[classicres[i][j][k]["Frequency"]] = classicres[i][j][k]["Amplitude"]
21
22     lists = sorted(amplitudes.items())
23     fc, ac = zip(*lists)
24     plt.plot(fc, ac, color='darkgrey', linestyle='-', label='Classic Method')
25
26     amplitudes_prop = {}

```



```

102         if k not in amplitudes:
103             amplitudes[k] = {}
104
105         if classicres[i][j][k][1]["Frequency"] in amplitudes[k]:
106             if classicres[i][j][k][1]["Amplitude"] >
107                 ↪ amplitudes[k][classicres[i][j][k][1]["Frequency"]]:
108                 ↪ classicres[i][j][k][1]["Amplitude"]
109             else:
110                 amplitudes[k][classicres[i][j][k][1]["Frequency"]] =
111                 ↪ classicres[i][j][k][1]["Amplitude"]
112
113     amplitudes_prop = {}
114     for i in proposedres:
115         if i not in amplitudes_prop:
116             amplitudes_prop[i] = {}
117         for j in proposedres[i]:
118             if proposedres[i][j] != {}:
119                 if proposedres[i][j]["Frequency"] in amplitudes_prop[i]:
120                     if proposedres[i][j]["Amplitude"] >
121                         ↪ amplitudes_prop[i][proposedres[i][j]["Frequency"]]:
122                         ↪ amplitudes_prop[i][proposedres[i][j]["Frequency"]] =
123                         ↪ proposedres[i][j]["Amplitude"]
124             else:
125                 # frequencies.append(proposedres[i][j]["Frequency"])
126                 amplitudes_prop[i][proposedres[i][j]["Frequency"]] = proposedres[i][j]["Amplitude"]
127
128     if type == "largeplot":
129         lists = {}
130         for i in sorted(amplitudes):
131             lists[i] = sorted(amplitudes[i].items())
132             fc, ac = zip(*lists[i])
133             plt.plot(fc, ac, 'k-', label='Classic Method CB=' + str(i))
134
135         lists = {}
136         for i in sorted(amplitudes_prop):
137             lists[i] = sorted(amplitudes_prop[i].items())
138             fc, ac = zip(*lists[i])
139             plt.plot(fc, ac, 'b-', label='Proposed Method CB=' + str(i))
140
141     elif type == "subplots":
142         figheight = 3
143         figwidth = 4
144         fig, axarr = plt.subplots(figheight, figwidth)
145
146         lists = {}
147         lists_prop = {}
148         plotnum = 0
149         for i in sorted(amplitudes):
150             lists[i] = sorted(amplitudes[i].items())
151             fc, ac = zip(*lists[i])
152             subplot = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
153             subplot.plot(fc, ac, 'k-', label='Classic Method CB=' + str(i))
154             subplot.set_title('Chargebalance = ' + str(i))
155
156             lists_prop[i] = sorted(amplitudes_prop[i].items())
157             fc_prop, ac_prop = zip(*lists_prop[i])
158             subplot.plot(fc_prop, ac_prop, 'b-', label='Proposed Method CB=' + str(i))
159
160             plotnum += 1
161             subplot.set_ylim([0, 1])
162             subplot.legend()
163
164         for i in range(plotnum, figheight*figwidth):
165             subplot = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
166             fig.delaxes(subplot)
167             plotnum += 1
168
169     else :
170         print "No type selected! Choose 'largeplot' or 'subplots'"
171         return None
172
173     return plt
174
175 # Plots '3 dimensional' results Chargeperphase(e.g. chargebalance)

```

```

175 def plotjustification3Dcpp(classicres, proposedres, type, balance = 0.5, freqname = "Frequency",
176 ↪ chargename = "Chargeperphase"): # If type is subplots, build all results in smaller subplots
177 amplitudes = {}
178
179 for i in classicres:
180     for j in classicres[i]:
181         for k in classicres[i][j]:
182             for l in classicres[i][j][k]:
183                 if classicres[i][j][k][l] != {}:
184                     if k not in amplitudes:
185                         amplitudes[k] = {}
186
187                     if classicres[i][j][k][l]["Frequency"] in amplitudes[k]:
188                         if classicres[i][j][k][l]["Chargeperphase"] >
189                             ↪ amplitudes[k][classicres[i][j][k][l]["Frequency"]]:
190                             amplitudes[k][j][classicres[i][j][k][l]["Frequency"]] =
191                             ↪ classicres[i][j][k][l]["Chargeperphase"]
192
193                     else:
194                         amplitudes[k][classicres[i][j][k][l]["Frequency"]] =
195                         ↪ classicres[i][j][k][l]["Chargeperphase"]
196
197 amplitudes_prop = {}
198
199 for i in proposedres:
200     if i not in amplitudes_prop:
201         amplitudes_prop[i] = {}
202     for j in proposedres[i]:
203         if proposedres[i][j] != {}:
204             if proposedres[i][j][freqname] in amplitudes_prop[i]:
205                 if proposedres[i][j][chargename] > amplitudes_prop[i][proposedres[i][j][freqname]]:
206                     amplitudes_prop[i][proposedres[i][j][freqname]] = proposedres[i][j][chargename]
207             else:
208                 # frequencies.append(proposedres[i][j]["Frequency"])
209                 amplitudes_prop[i][proposedres[i][j][freqname]] = proposedres[i][j][chargename]
210
211 if type == "largeplot" or type == "singleCB":
212     lists = {}
213     for i in sorted(amplitudes):
214         lists[i] = sorted(amplitudes[i].items())
215         fc, ac = zip(*lists[i])
216         if type == "largeplot" or (type == "singleCB" and i == balance):
217             plt.plot(fc, ac, color='darkgrey', linestyle='-', label='Classic Method')
218
219     lists = {}
220     for i in sorted(amplitudes_prop):
221         lists[i] = sorted(amplitudes_prop[i].items())
222         fc, ac = zip(*lists[i])
223         if type == "largeplot" or (type == "singleCB" and i == balance):
224             plt.plot(fc, ac, color='b', linestyle='-', label='Proposed Method')
225         if type == "singleCB":
226             plt.title('$T_{anode}$ = ' + str(i) + '$T_{signal}$', y=0.7)
227     plt.ylim([0, 0.04])
228     plt.xlabel("Frequency (kHz)")
229     plt.ylabel("Threshold charge per phase ($\mu$ C)")
230     plt.legend()
231     plt.grid()
232     plt.xlim(left=0, right=45)
233     plt.tight_layout()
234
235 elif type == "subplots":
236     figheight = 3
237     figwidth = 4
238     fig, axarr = plt.subplots(figheight, figwidth)
239
240     lists = {}
241     lists_prop = {}
242     plotnum = 0
243     for i in sorted(amplitudes):
244         lists[i] = sorted(amplitudes[i].items())
245         fc, ac = zip(*lists[i])
246         subplt = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
247         ac = [1000 * a for a in ac]
248         l1 = subplt.plot(fc, ac, 'k-')
249         subplt.set_title('$T_{anode}$ = ' + str(i) + '$T$', y=0.8)
250

```

```

249     lists_prop[i] = sorted(amplitudes_prop[i].items())
250     fc_prop, ac_prop = zip(*lists_prop[i])
251     ac_prop = [1000 * a for a in ac_prop]
252     l2 = subplt.plot(fc_prop, ac_prop, 'b-')
253
254     plotnum += 1
255     subplt.set_ylim([0, 30])
256     subplt.set_xlabel("Frequency (kHz)")
257     subplt.set_ylabel("Charge per phase ($nC$)")
258     # subplt.legend()
259
260     for i in range(plotnum, figheight*figwidth):
261         subplt = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
262         fig.delaxes(subplt)
263         plotnum += 1
264
265     fig.legend([l1, l2], labels=["Classical Method", "Proposed Method"]#, loc='lower right')#,
266     ↪ bbox_to_anchor=(0.6, 0.3))
267     # plt.tight_layout()
268
269     else :
270         print "No type selected! Choose 'largeplot' or 'subplots'"
271         return None
272
273     return plt
274
275
276     #####
277     # Plots classic versus proposed method per frequency -> REWRITE!
278     #####
279     def plotjustification3Dcppreverse(classicres, proposedres, type): # If type is subplots, build all
280     ↪ results in smaller subplots
281         amplitudes = {}
282
283         for i in classicres:
284             for j in classicres[i]:
285                 for k in classicres[i][j]:
286                     for l in classicres[i][j][k]:
287                         if classicres[i][j][k][l] != {}:
288                             if j not in amplitudes:
289                                 amplitudes[j] = {}
290
291                             if classicres[i][j][k][l]["Chargeperphase"] in amplitudes[j]:
292                                 if classicres[i][j][k][l]["Chargeperphase"] >
293                                     ↪ amplitudes[j][classicres[i][j][k][l]["Chargebalance"]]:
294                                     amplitudes[j][k][classicres[i][j][k][l]["Chargebalance"]] =
295                                     ↪ classicres[i][j][k][l]["Chargeperphase"]
296
297                             else:
298                                 amplitudes[j][classicres[i][j][k][l]["Chargebalance"]] =
299                                 ↪ classicres[i][j][k][l]["Chargeperphase"]
300
301         amplitudes_prop = {}
302         for i in proposedres:
303             for j in proposedres[i]:
304                 if j not in amplitudes_prop:
305                     amplitudes_prop[j] = {}
306                 if proposedres[i][j] != {}:
307                     if proposedres[i][j]["Frequency"] in amplitudes_prop[j]:
308                         if proposedres[i][j]["Chargeperphase"] >
309                             ↪ amplitudes_prop[j][proposedres[i][j]["Chargebalance"]]:
310                             amplitudes_prop[j][proposedres[i][j]["Chargebalance"]] =
311                             ↪ proposedres[i][j]["Chargeperphase"]
312
313                     else:
314                         # frequencies.append(proposedres[i][j]["Frequency"])
315                         amplitudes_prop[j][proposedres[i][j]["Chargebalance"]] =
316                         ↪ proposedres[i][j]["Chargeperphase"]
317
318         if type == "largeplot":
319             lists = {}
320             for i in range(4,6,2):# sorted(amplitudes):
321                 lists[i] = sorted(amplitudes[i].items())
322                 fc, ac = zip(*lists[i])
323                 plt.plot(fc, ac, 'k-', label='Classic Method CB=' + str(i))
324
325             lists = {}
326             for i in range(4,6,2):# sorted(amplitudes_prop):

```

```

319         lists[i] = sorted(amplitudes_prop[i].items())
320         fc, ac = zip(*lists[i])
321         plt.plot(fc, ac, 'b-', label='Proposed Method CB=' + str(i))
322
323     plt.ylim([0, 0.1])
324
325     elif type == "subplots":
326         figheight = 3
327         figwidth = 4
328         fig, axarr = plt.subplots(figheight, figwidth)
329
330         lists = {}
331         lists_prop = {}
332         plotnum = 0
333         for i in sorted(amplitudes):
334             lists[i] = sorted(amplitudes[i].items())
335             fc, ac = zip(*lists[i])
336             subplt = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
337             l1 = subplt.plot(fc, ac, 'k-', label='Classic Method')
338             subplt.set_title('$T_{hi}$ = ' + str(i) + '$T$')
339
340             lists_prop[i] = sorted(amplitudes_prop[i].items())
341             fc_prop, ac_prop = zip(*lists_prop[i])
342             l2 = subplt.plot(fc_prop, ac_prop, 'b-', label='Proposed Method')
343
344             plotnum += 1
345             subplt.set_ylim([0, 0.03])
346             # subplt.legend()
347
348         for i in range(plotnum, figheight*figwidth):
349             subplt = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
350             fig.delaxes(subplt)
351             plotnum += 1
352
353         fig.legend([l1, l2], labels=["Classical Method", "Proposed Method"], loc='lower right')#,
354             ↪ bbox_to_anchor=(0.6, 0.3))
355
356     else :
357         print "No type selected! Choose 'largeplot' or 'subplots'"
358         return None
359
360     return plt

```
