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

```
############################################################
# This main file contains an overview of functions that were
# run to do simulations in the thesis of Koen Emmer
# Uncomment functions to run them
# For questions, contact me via LinkedIn
# https://www.linkedin.com/in/koenemmer/
############################################################

############################################################
# Import Modules
############################################################
# Import Python libraries
# Import for BrainFrame
# import matplotlib

from configoptions import *
from reportplots import *
from datasetproposedmethod import *
from datasetsformodeldesign import *
from monopolar_simulations import *
from bipolar_simulations import *

############################################################
# Build datasets for
############################################################
# runwave("ClassicModel", "square", "complete", "saveall")
# runwave("ClassicModel", "sine", "complete", "saveall")
# runsquarewaveCB("ClassicModel", "complete", "saveall")
# runwave("ProposedModel", "square", "complete", "saveall")
# runwave("ProposedModel", "sine", "complete", "saveall")
# runsquarewaveCB("ProposedModel", "complete", "saveall")

############################################################
# Find blocking amplitudes via proposed method
############################################################
# squarewaveresults()
# sineresults()
# assymmetricalresults()

############################################################
# Plot single KHFAC simulation
############################################################
# plot_square_classic(tstop=20)
# plot_sine_classic()
# plot_assymmetric_classic(f=10, a=0.2, tstop = 50.0, chargebalance=0.1, intrinsic=1)
# plot_square_proposed(f=10, a=0.47)
# plot_sine_proposed(f=10, a=0.52)
# plot_assymmetric_classic(f=10, a=0.55, tstop = 51.0, chargebalance=0.8, intrinsic=1)
# plot_assymmetric_classic(f=10, a=0.6, tstop = 51.0, chargebalance=0.9, intrinsic=1)
# plot_assymmetric_proposed(f=10, a=0.19, chargebalance=0.1)
# plot_assymmetric_proposed(f=10, a=0.2, chargebalance=0.1)
# plot_assymmetric_proposed(f=10, a=0.6, chargebalance=0.9)
# plot_sine_proposed(10,0.52)
# plot_sine_proposed(10,0.45)
# plot_sine_proposed(40,0.66)
# plot_sine_proposed(40,0.60)
# plot_square_proposed_bipolar(10,1.0)


############################################################
# Plot and test waveforms (uncomment one of the lines together with the last two lines to produce a
↪  picture
############################################################
```

```
# t_signal, i_signal = sinewave(0, 0.1, 10, 1, 1000)
# t_signal, i_signal = chargebalanced_asymmetrical(0, 2.0, 1, 0.06, 0.8, 0.1, 0.1)
# t_signal, i_signal = chargebalanced_asymmetrical(0, 0.1, 10, 0.05, 0.9)
# t_signal, i_signal = trianglewave(0, 0.1, 10, 1)
# t_signal, i_signal = squarewave_ip(0, 0.2, 10, 1, 0.1, 0.1)
# t_signal, i_signal = squarewave(0, 0.2, 10, 1)
# t_signal, i_signal = stepwave(0, 0.1, 10, 2, 3)
# t_signal, i_signal = stepwave_sine(0, 0.1, 10, 1, 20)
# plot_signal(plt, t_signal, i_signal)
# plt.show()

################################################################
# Monopolar simulations -> create result files
################################################################
# monopolar_triangularwave_results()
# monopolar_sinewave_results()
# monopolar_squarewave_results()
# monopolar_step("triangular")
# monopolar_step("sine")
# monopolar_assymmetricalwave_results()
# squarewave_ipd_results()
# monopolar_squarewave_realdistance_results()
# squarewave_ipd_results("validation")

################################################################
# Bipolar simulations -> create result files
################################################################
# bipolar_squarewave_ETAdistance_IECdistance_parallel_results()
# bipolar_squarewave_ETAdistance_IECdistance_parallel_results(zoom=True)
# bipolar_squarewave_IECdistance_results('parallel')
# bipolar_squarewave_IECdistance_results('perpendicular')
# bipolar_squarewave_orientation_results('x')
# bipolar_squarewave_orientation_results('z')

################################################################
# Build report plots
################################################################
## Background chapter
# plotAP()
# plotKHFACdemogates()
# plothminftau()

## Method chapter
# plot_2D_blockdescription()
# plot_3D_blockdescription()
# plot_gates_proposedmethodsetup()
# plot_asymmetrical_waveform()
# justificationplots("sine")
# justificationplots("square")
# justificationplots("squareCB")

## Results Chapter
# Monopolar plots
# plot_monopolar("monopolar_basic_waveforms", "amplitude")
# plot_monopolar("monopolar_basic_waveforms", "cpp")
# plot_monopolar("stepfunctions", "amplitude")
# plot_monopolar("stepfunctions", "cpp")
# plot_monopolar("asymmetrical", "amplitude", "colour")
# plot_monopolar("asymmetrical", "cpp", "colour")
# plot_monopolar("asymmetrical", "amplitude", "3D")
# plot_monopolar("asymmetrical", "cpp", "3D")
# plot_monopolar("asymmetrical", "amplitude", "singlefrequency", 10)
# plot_monopolar("squarewave_ipd", "amplitude", "lines")
# plot_monopolar("squarewave_ipd", "cpp", "lines")
# plot_monopolar("squarewave_ipd", "amplitude", "colour")
# plot_monopolar("squarewave_ipd", "cpp", "colour")
# plot_monopolar("squarewave_ipd", "amplitude", "3D")
# plot_monopolar("squarewave_ipd", "cpp", "3D")
# plot_monopolar("squarewave_ipd_validation", "amplitude", "lines")
# plot_monopolar("squarewave_ipd_validation", "cpp", "lines")
# plot_monopolar("squarewave_ipd_validation", "amplitude", "colour")
# plot_monopolar("squarewave_ipd_validation", "cpp", "colour")
# plot_monopolar("realdistance", "amplitude")
# plot_monopolar("realdistance", "cpp")
# Bipolar plots
# plot_bipolar("ETA_IEC_parallel", "amplitude", "lines")
# plot_bipolar("ETA_IEC_parallel", "cpp", "lines")
# plot_bipolar("ETA_IEC_parallel", "amplitude", "colour")
```

```
# plot_bipolar("ETA_IEC_parallel", "cpp", "colour")
# plot_bipolar("ETA_IEC_parallel", "amplitude", "3D")
# plot_bipolar("ETA_IEC_parallel", "cpp", "3D")
# plot_bipolar("ETA_IEC_parallel", "amplitude", "optimal")
# plot_bipolar("ETA_IEC_parallel", "amplitude", "lines", zoom=True)
# plot_bipolar("ETA_IEC_parallel", "cpp", "lines", zoom=True)
# plot_bipolar("ETA_IEC_parallel", "amplitude", "colour", zoom=True)
# plot_bipolar("ETA_IEC_parallel", "cpp", "colour", zoom=True)
# plot_bipolar("ETA_IEC_parallel", "amplitude", "3D", zoom=True)
# plot_bipolar("ETA_IEC_parallel", "cpp", "3D", zoom=True)
# plot_bipolar("ETA_IEC_parallel", "amplitude", "optimal", zoom=True)
# plot_bipolar("IEC_parallelperpendicular", "amplitude")
# plot_bipolar("IEC_parallelperpendicular", "cpp")
# plot_bipolar("orientation", "amplitude")

## Discussion plots
# plot_asymmetrical_bipolar(0.001)
# plot_ipdbasicimplementation(0.25,0.25)
# plot_ipdbasicimplementation(0,0.5)
# plot_ipdcomplexeimplementation()
# plot_ipd_validation(0.4,0,0)

################################################################
## Build presentation plots
################################################################
# buildKHFACgif()


quitNeuron()
```

blockdetection.py

```
################################################################
# This file contains the functions that detect a successful block
################################################################
import numpy as np


################################################################
# This functions returns values that can be used to determine
# the thresholds that categorize a block as successful or
# non-successful
################################################################
def conductionBlock (rec, par, node):
    dt = par['dt']
    tstart = int((par['HFSdelay'] + par['tonset']) / dt) # Start measuring 1 ms after initiating the
    ↪  KHFAC signal, divide by dt to get listindex (2)

    gatevalues = rec['gates']
    # Neuron hocObjects need to be converted to arrays via np.array()!
    blockingnode_h = np.array(gatevalues['h_node' + str(node)])  # The node where the block occurs (a
    ↪  virtual anode of the HFS node)


    h_max = np.max(blockingnode_h[tstart:])
    h_min = np.min(blockingnode_h[tstart:])


    voltages = rec['voltage']
    blockingnode_voltage = np.array(voltages['v_node' + str(node)])
    V_max = np.max(blockingnode_voltage[tstart:])
    V_min = np.min(blockingnode_voltage[tstart:])

    blockingnode_m = np.array(gatevalues['m_node' + str(node)])
    m_max = np.max(blockingnode_m[tstart:])
    m_min = np.min(blockingnode_m[tstart:])

    currents = rec['current']
    blockingnode_h_inf = np.array(currents['h_inf_node' + str(node)])

    blockingnode_h_dist = [a - b for a, b in zip(blockingnode_h_inf, blockingnode_h)]
    h_dist_max = np.max(blockingnode_h_dist)
    h_dist_min = np.min(blockingnode_h_dist)


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

```
    for i in range(20):
        ipd_anodal_list.append(round(0.0 + 0.05 * i, 2))
    if type == "validation":
        ipd_anodal_list = []
        for i in range(5):
            ipd_anodal_list.append(round(0.0 + 0.20 * i, 2))
    ipd_cathodal_list = ipd_anodal_list
    starttime = time.time()
    amplitudelist = []
    for i in range(100):
        amplitudelist.append(round(0.1 + 0.01 * i, 2))
    if type == "validation":
        amplitudelist = []
        for i in range(80):
            amplitudelist.append(round(0.35 + 0.01 * i, 2))
    createMRGaxon(par)
    result = {}
    for ipd1 in ipd_anodal_list:
        result[ipd1] = {}
        for ipd2 in ipd_cathodal_list:
            result[ipd1][ipd2] = {}
            for a in amplitudelist:
                rec = recordMRGaxon(recpar)
                par['HFSamp'] = a
                if ipd1+ipd2 >= 1.0:
                    break
                t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                ↪  par['HFSamp'], ipd1, ipd2)
                updateMRGaxon(par)
                print("--- RUN KHFAC-SIGNAL (%s kHz, %s mA, %s ipd1, %s ipd2, type = %s) ---" %
                ↪  (frequency, a, ipd1, ipd2, type))
                runMRGaxon(rec, t_signal, i_signal)
                block = False
                if type != "validation":
                    block = is_blocked(rec,par)
                else:
                    if blockeffectiveness(rec,par) >= 0.9:
                        block = True
                if block:
                    cpp = (1.0-ipd1-ipd2)/(2*(1000*frequency))*a
                    result[ipd1][ipd2]["frequency"] = frequency
                    result[ipd1][ipd2]["ipd1"] = ipd1
                    result[ipd1][ipd2]["ipd2"] = ipd2
                    result[ipd1][ipd2]["amplitude"] = a
                    result[ipd1][ipd2]["chargeperphase"] = cpp
                    break
    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
    filename = 'results_monopolar/results_squarewave_ipd_proposedmethod'
    if type == "validation":
        filename = 'results_monopolar/results_squarewave_ipd_validation'
    np.save(filename, result)


##############################################################
# Build dataset for step wave triangular shape
##############################################################
def monopolar_step(shape):
    print("--- RUN MONOPOLAR SIMULATION STEPWAVE (" + shape + ") WITH INCREASING STEPSIZES---")
    config = 'ProposedModel'
    par, recpar = selectConfig(config)
    frequency = 10
    par['HFSfrequency'] = frequency
    step_list = []
    for i in range(1,51):
        step_list.append(i)
    starttime = time.time()
    amplitudelist = []
    for i in range(200):
        amplitudelist.append(round(0.1 + 0.01 * i, 2))
    createMRGaxon(par)
    result = {}
    for steps in step_list:
        result[steps] = {}
        for a in amplitudelist:
            rec = recordMRGaxon(recpar)
            par['HFSamp'] = a
            if shape == "triangular":
```

```
                t_signal, i_signal = stepwave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                ↪  par['HFSamp'], steps)
            elif shape == "sine":
                t_signal, i_signal = stepwave_sine(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                ↪  par['HFSamp'], steps)
            else:
                print "Wrong shape selected!"
                break
            updateMRGaxon(par)
            print("--- RUN KHFAC-SIGNAL (%s kHz, %s mA, %s steps (triangular)) ---" % (frequency, a,
            ↪  steps))
            runMRGaxon(rec, t_signal, i_signal)
            if is_blocked(rec, par):
                cpp = 0.0
                if shape == "triangular":
                    stepsize = a / steps
                    for i in range(steps+1):
                        cpp += (i * stepsize / (4 * frequency * steps))
                    for i in range(steps):
                        cpp += (i * stepsize / (4 * frequency * steps))
                elif shape == "sine":
                    for i in range(steps*2):
                        cpp += (a * np.sin(2 * np.pi * i / (steps * 4)) / (4 * frequency * steps))
                result[steps]["frequency"] = frequency
                result[steps]["numberofsteps"] = steps
                result[steps]["amplitude"] = a
                result[steps]["chargeperphase"] = cpp
                break
    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
    filename = 'results_monopolar/results_monopolar_stepwave_' + shape
    np.save(filename, result)


############################################################
# Build dataset for triangular wave tested with Proposed Model
############################################################
def monopolar_triangularwave_results():
    frequencylist = [3]
    for i in range(19):
        frequencylist.append((i + 2) * 2)
    result = find_blockingamplitudes_Freq(frequencylist, "triangular")
    filename = 'results_monopolar/results_monopolar_triangularwave'
    np.save(filename, result)


############################################################
# Build dataset for square wave tested with Proposed Model
############################################################
def monopolar_squarewave_results():
    frequencylist = [3]
    for i in range(19):
        frequencylist.append((i + 2) * 2)
    result = find_blockingamplitudes_Freq(frequencylist, "square")
    filename = 'results_monopolar/results_monopolar_squarewave'
    np.save(filename, result)


############################################################
# Build dataset for sine wave tested with Proposed Model
############################################################
def monopolar_sinewave_results():
    frequencylist = [3]
    for i in range(19):
        frequencylist.append((i + 2) * 2)
    result = find_blockingamplitudes_Freq(frequencylist, "sine")
    filename = 'results_monopolar/results_monopolar_sinewave'
    np.save(filename, result)


############################################################
# Build dataset for assymmetricalwave tested with Proposed Model
############################################################
def monopolar_assymmetricalwave_results():
    frequencylist = [3]
    for i in range(19):
        frequencylist.append((i + 2) * 2)
    chargebalancelist = []
```

```
    for i in range(18):
        chargebalancelist.append(round(0.1 + 0.05*i, 2))
    result = find_blockingamplitudes_CB(frequencylist, chargebalancelist)
    filename = 'results_monopolar/results_monopolar_assymmetricalwave_allfrequencies'
    np.save(filename, result)


############################################################
# Build dataset for assymmetricalwave tested with Proposed Model
############################################################
def monopolar_squarewave_realdistance_results():
    print("--- RUN MONOPOLAR SIMULATION SQUAREWAVE WITH CHANGE OVER REAL PERPENDICULAR DISTANCE ---")
    config = 'ProposedModel'
    par, recpar = selectConfig(config)
    frequency = 10
    par['HFSfrequency'] = frequency
    perpdistance_list = []
    # Max nervediameter is 4.67+1.17 = 5,84. Axondiameter is 10micrometre, range is 10,585. Steps of 10 micrometre
    for i in range(100, 5850, 10):
        perpdistance_list.append(i)
    starttime = time.time()
    amplitudelist = []
    for i in range(200):
        amplitudelist.append(round(0.01 * i, 2))
    for i in range(280):
        amplitudelist.append(round(2.0 + 0.1 * i, 2))
    createMRGaxon(par)
    result = {}
    count = 0
    for dist in perpdistance_list:
        result[dist] = {}
        par['HFSz'] = dist
        for a in amplitudelist[count:]:
            rec = recordMRGaxon(recpar)
            par['HFSamp'] = a
            t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
            updateMRGaxon(par)
            print("--- RUN KHFAC-SIGNAL (%s kHz, %s mA, %s perpdistance) ---" % (frequency, a, dist))
            runMRGaxon(rec, t_signal, i_signal)
            if is_blocked(rec, par):
                cpp = a / (2*frequency)
                result[dist]["frequency"] = frequency
                result[dist]["perpendiculardistance"] = dist
                result[dist]["amplitude"] = a
                result[dist]["chargeperphase"] = cpp
                break
            count += 1
    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
    filename = 'results_monopolar/results_monopolar_squarewave_realdistance'
    np.save(filename, result)


############################################################
# Plot results of  square wave with interphase delay
############################################################
def plot_monopolar(type, restype="amplitude", plottype="lines", plotfreq = 10):
    if type == "squarewave_ipd":
        result = np.load("results_monopolar/results_squarewave_ipd_proposedmethod.npy", encoding="ASCII").item()
        plot_results_monopolar_ipd(result, restype, plottype)
        filename = "results_squarewave_ipd_"+plottype+"_"+restype
    if type == "squarewave_ipd_validation":
        result = np.load("results_monopolar/results_squarewave_ipd_validation.npy", encoding="ASCII").item()
        plot_results_monopolar_ipd(result, restype, plottype, validation=True)
        filename = "results_squarewave_ipd_validation_" + plottype + "_" + restype
    if type == "monopolar_basic_waveforms":
        result_square = np.load("results_monopolar/results_monopolar_squarewave.npy").item()
        result_sine = np.load("results_monopolar/results_monopolar_sinewave.npy").item()
        result_triangular = np.load("results_monopolar/results_monopolar_triangularwave.npy").item()
        plot_results_monopolar_basicwaveforms(result_square, result_sine, result_triangular, type=restype)
        filename = "results_basicwaveforms_"+restype
    if type == "stepfunctions":
        result_stepsine = np.load("results_monopolar/results_monopolar_stepwave_sine.npy").item()
```

```
        result_steptriangular = np.load("results_monopolar/results_monopolar_stepwave_triangular.npy").item()
        plt_results_monopolar_stepwaveforms(result_stepsine, result_steptriangular, type=restype)
        filename = "results_stepfunctions_"+restype
    if type == "realdistance":
        result = np.load("results_monopolar/results_monopolar_squarewave_realdistance.npy").item()
        plt_results_monopolar_realdistance(result, type=restype)
        filename = "results_monopolar_squarewave_realdistance_"+restype
    if type == "asymmetrical":
        result = np.load("results_monopolar/results_monopolar_assymmetricalwave_allfrequencies.npy").item()
        plot_results_asymmetrical(result, restype, plottype, plotfreq)
        filename_plottype = plottype
        if plottype == "singlefrequency":
            filename_plottype = plottype + "_" + str(plotfreq) + "kHz"
        filename = "results_monopolar_asymmetricalwaves_"+filename_plottype+"_"+restype
    plt.savefig("reportplots/monopolar_results/" + filename + ".pdf", format='pdf', dpi=1200)
    plt.show()
```

bipolar_simulations.py

```
##############################################################
# Bipolar simulation functions
##############################################################
import csv
from configoptions import *
import time
import datetime
from MRG_Model import *             # Interface with Neuron MRG Model
from blockdetection import *
from waveforms import *             # Waveform functions for Neuron
from plottingfunctions_results import *
# from simulationfunctions_proposedmodel import *


import os
if not os.path.exists('results_bipolar'):
    os.makedirs('results_bipolar')          # Ensure that brainframe makes justification folder
if not os.path.exists('reportplots'):
    os.makedirs('reportplots')
if not os.path.exists('reportplots/bipolar_results'):
    os.makedirs('reportplots/bipolar_results')

##############################################################
# Build dataset bipolar square wave simulation, changing electrode to axon distance and inter electrode contact distance
##############################################################
def bipolar_squarewave_ETAdistance_IECdistance_parallel_results(zoom=False): #ETA = Electrode To Axon, IEC = Inter Electrode Contact
    print("--- RUN BIPOLAR SIMULATION SQUAREWAVE WITH CHANGING ELECTRODE TO AXON DISTANCE AND INTER ELECTRODE CONTACT DISTANCE, ELECTRODE PARALLEL TO AXON ---")
    config = 'ProposedModel_Bipolar'
    par, recpar = selectConfig(config)
    frequency = 10
    par['HFSfrequency'] = frequency
    perpdistance_list = []
    for i in range(1000, 6001, 500):
        perpdistance_list.append(i)

    poledistance_list = []
    count = {}
    if zoom:
        for i in range (1000, 10000, 100):
            poledistance_list.append(i)
    else:
        for i in range(1000, 60001, 1000):
            poledistance_list.append(i)
    for i in poledistance_list:
        count[i] = 0
    amplitudelist = []
    for i in range(200):
        amplitudelist.append(round(0.01 * i, 2))
    for i in range(280):
        amplitudelist.append(round(2.0 + 0.1 * i, 2))
    starttime = time.time()
    createMRGaxon(par)
```

```
    result = {}
    for ETAdist in perpdistance_list:
        result[ETAdist] = {}
        par['HFSz'] = ETAdist
        par['HFSz2'] = ETAdist
        for IECdist in poledistance_list:
            result[ETAdist][IECdist] = {}
            par['HFSy2'] = -IECdist
            for a in amplitudelist[count[IECdist]:]:
                rec = recordMRGaxon(recpar)
                par['HFSamp'] = a
                t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                ↪  par['HFSamp'])
                updateMRGaxon(par)
                print("--- RUN KHFAC-SIGNAL (%s kHz, %s mA, %s um el-to-ax distance, %s um
                ↪  interelectrodecontact distance) ---" % (frequency, a, ETAdist, IECdist))
                runMRGaxon(rec, t_signal, i_signal)
                if is_blocked(rec, par):
                    cpp = (1.0)/(2*(1000*frequency))*a
                    result[ETAdist][IECdist]["frequency"] = frequency
                    result[ETAdist][IECdist]["electrodetoaxondistance"] = ETAdist
                    result[ETAdist][IECdist]["interelectrodecontactdistance"] = IECdist
                    result[ETAdist][IECdist]["amplitude"] = a
                    result[ETAdist][IECdist]["chargeperphase"] = cpp
                    count[IECdist] = max(0, count[IECdist] - 10)
                    break
                count[IECdist] += 1
    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
    filename_end = ""
    if zoom:
        filename_end = "_zoom"
    filename = 'results_bipolar/results_squarewave_ETAdist_IECdist_parallel'+filename_end
    np.save(filename, result)


############################################################
# Build dataset for square wave with interphase delay tested with Proposed Model
############################################################
def bipolar_squarewave_IECdistance_results(mode): #ETA = Electrode To Axon, IEC = Inter Electrode Contact
    print("--- RUN BIPOLAR SIMULATION SQUAREWAVE WITH CHANGING ELECTRODE TO AXON DISTANCE AND INTER
    ↪  ELECTRODE CONTACT DISTANCE, ELECTRODE %s TO AXON ---" % mode)
    config = 'ProposedModel_Bipolar'
    par, recpar = selectConfig(config)
    frequency = 10
    par['HFSfrequency'] = frequency
    poledistance_list = []
    count = {}
    for i in range(1000, 60001, 500):
        poledistance_list.append(i)
    amplitudelist = []
    for i in range(200):
        amplitudelist.append(round(0.01 * i, 2))
    starttime = time.time()
    createMRGaxon(par)
    result = {}
    for dist in poledistance_list:
        result[dist] = {}
        if mode == 'parallel':
            par['HFSy2'] = -dist
        elif mode == 'perpendicular':
            par['HFSx2'] = -dist
        for a in amplitudelist:
            rec = recordMRGaxon(recpar)
            par['HFSamp'] = a
            t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
            ↪  par['HFSamp'])
            updateMRGaxon(par)
            print("--- RUN KHFAC-SIGNAL (%s kHz, %s mA, %s um interelectrodecontact distance) ---" %
            ↪  (frequency, a, dist))
            runMRGaxon(rec, t_signal, i_signal)
            if is_blocked(rec, par):
                cpp = (1.0)/(2*(1000*frequency))*a
                result[dist]["frequency"] = frequency
                result[dist]["interelectrodecontactdistance"] = dist
                result[dist]["amplitude"] = a
                result[dist]["chargeperphase"] = cpp
                break
```

```
    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
    filename = 'results_bipolar/results_squarewave_IECdist_' + mode
    np.save(filename, result)


############################################################
# Build dataset for square wave with bipolar electrode of changing orientation
############################################################
def bipolar_squarewave_orientation_results(mode = 'x'): #ETA = Electrode To Axon, IEC = Inter Electrode Contact
    print("--- RUN BIPOLAR SIMULATION SQUAREWAVE WITH CHANGING ELECTRODE ORIENTATION OVER %s-AXIS---" % mode)
    config = 'ProposedModel_Bipolar'
    par, recpar = selectConfig(config)
    frequency = 10
    par['HFSfrequency'] = frequency
    orientation_list = []
    count = {}
    poledistance_list = []
    for i in range(45, 91, 5):
        orientation_list.append(i)
    for i in range(1000, 5001, 200):
        poledistance_list.append(i)
    amplitudelist = []
    for i in range(20,200):
        amplitudelist.append(round(0.01 * i, 2))
    starttime = time.time()
    createMRGaxon(par)
    result = {}
    for angle in orientation_list:
        result[angle] = {}
        for dist in poledistance_list:
            result[angle][dist]= {}
            par['HFSy2'] = -np.cos(np.deg2rad(angle))*dist
            if mode == 'x':
                par['HFSx2'] = -np.sin(np.deg2rad(angle))*dist
            if mode == 'z':
                par['HFSz2'] = par['HFSz'] + np.sin(np.deg2rad(angle))*dist
                print par['HFSz2']
                print par['HFSy2']
                print par['HFSx2']
            for a in amplitudelist:
                rec = recordMRGaxon(recpar)
                par['HFSamp'] = a
                t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
                updateMRGaxon(par)
                print("--- RUN KHFAC-SIGNAL (%s kHz, %s mA, %s degree angle, %s um interpolar distance) ---" % (frequency, a, angle, dist))
                runMRGaxon(rec, t_signal, i_signal)
                if is_blocked(rec, par):
                    cpp = (1.0)/(2*(1000*frequency))*a
                    result[angle][dist]["frequency"] = frequency
                    result[angle][dist]["electrodeangle"] = angle
                    result[angle][dist]["interelectrodecontactdistance"] = dist
                    result[angle][dist]["amplitude"] = a
                    result[angle][dist]["chargeperphase"] = cpp
                    break
    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
    filename = 'results_bipolar/results_squarewave_orientation_' + mode
    np.save(filename, result)


############################################################
# Plot results of  square wave with interphase delay
############################################################
def plot_bipolar(type, restype="amplitude", plottype="lines", zoom=False):
    if type == "ETA_IEC_parallel":
        zoomtext = ""
        if zoom:
            zoomtext = "_zoom"
        result = np.load("results_bipolar/results_squarewave_ETAdist_IECdist_parallel" + zoomtext + ".npy", encoding="ASCII").item()
        plot_results_bipolar_ETA_IEC_parallel(result, restype, plottype)
        filename = "results_bipolar_ETA_IEC_parallel" + zoomtext + "_"+plottype+"_"+restype
    elif type == "IEC_parallelperpendicular":
```

```
        result_parallel = np.load("results_bipolar/results_squarewave_IECdist_parallel.npy",
        ↪  encoding="ASCII").item()
        result_perpendicular = np.load("results_bipolar/results_squarewave_IECdist_perpendicular.npy",
        ↪  encoding="ASCII").item()
        plot_results_bipolar_IEC_parallelperp(result_parallel, result_perpendicular, restype)
        filename = "results_bipolar_IEC_parvsperp_"+restype
    elif type == "orientation":
        result_x =  np.load("results_bipolar/results_squarewave_orientation_x.npy",
        ↪  encoding="ASCII").item()
        result_z = np.load("results_bipolar/results_squarewave_orientation_z.npy",
        ↪  encoding="ASCII").item()
        filename = "results_bipolar_orientation_"+restype
        plot_results_bipolar_orientation(restype, result_x, result_z)
    plt.savefig("reportplots/bipolar_results/" + filename + ".pdf", format='pdf', dpi=1200)
    plt.show()
```

waveforms.py

```
############################################################
############################################################
# Pythoncode containing Neuron-vector descriptions of waveforms
############################################################
############################################################
import numpy as np
import matplotlib.pyplot as plt

############################################################
# Transforms a standard waveform into a signal that is ready to be used by Neuron
# t_start is when the signal should start
# t_stop is when the signal should end
# t_period is the time vector of the signal
# i_period is the current amplitude vector of the signal
# f is the frequency of the signal
# amplitude is the amplitude of the signal
# The outputs are:
# t_signal: the time vector of the signal
# i_signal: the amplitude vector of the signal
# Neuron combines the two to construct a signal,
# build up by having amplitude i_signal[x] at time t_signal[x]
############################################################
def KHFACblocksignal(t_start, t_stop, t_period, i_period, f, amplitude):
    f = 1000*f # Translate from KHz to Hz
    period = 1000.0/f #in milliseconds
    t_period = [x * period/max(t_period) for x in t_period] #Neuron t is in ms
    i_period = [x * amplitude for x in i_period]
    timestep = max(t_period)
    t_signal = []
    i_signal = []
    for i in range(0, int(t_stop/period)):
        t_signal = np.concatenate((t_signal, t_period))
        t_period = [x + timestep for x in t_period]
        if i < int(t_start/period):
            i_signal = np.concatenate((i_signal, [x * 0 for x in i_period]))
        else:
            i_signal = np.concatenate((i_signal, i_period))
    return t_signal, i_signal


############################################################
# Creates a sampled sinewave-period for Neuron, consisting of:
# t_period: the time vector of one period
# i_period: the amplitude vector of one period
############################################################
def sinewaveperiod(samplesperperiod):
    t_period = []
    i_period = []
    for i in range(0, samplesperperiod+1):
        t_period = np.concatenate((t_period, i), axis=None)
        i_period = np.concatenate((i_period, np.sin(2*np.pi*i/samplesperperiod)), axis=None)
    return t_period, i_period


############################################################
# Creates a sampled sinewave signal for Neuron
############################################################
def sinewave(t_start, t_stop, f, amplitude, samplesperperiod):
```

```
    t_period, i_period = sinewaveperiod(samplesperperiod)
    return KHFACblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)


############################################################
# Creates a sampled triangle wave signal for Neuron
############################################################
def trianglewave(t_start, t_stop, f, amplitude):
    t_period = [0, 1, 2, 3, 4]
    i_period = [0, 1, 0, -1, 0]
    return KHFACblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)


############################################################
# Creates a sampled square wave signal for Neuron
# ipd1 and ipd2 represent interphase delays, as a fraction of the waveform period (e.g. ipd1=0.1 will create an interphase delay of one tenth of the signal period after the anodal pulse
############################################################
def squarewave(t_start, t_stop, f, amplitude, ipd1 = 0, ipd2 = 0):
    period_length = 1.0
    no_ipd_length = period_length - ipd1 - ipd2
    t_period = [0, no_ipd_length / 2, no_ipd_length / 2, no_ipd_length / 2 + ipd1, no_ipd_length / 2 + ipd1, no_ipd_length + ipd1, no_ipd_length + ipd1, period_length]
    i_period = [1, 1, 0, 0, -1, -1, 0, 0]
    return KHFACblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)


############################################################
# Creates a sampled charge balanced asymmetrical square wave for Neuron
# chargeperphase in uC
# f in kHz
# t_start & t_stop in ms
############################################################
def chargebalanced_asymmetrical(t_start, t_stop, f, chargeperphase, ratioHi, ipd1 = 0, ipd2 = 0):
    ratioLo = 1.0-ratioHi
    period = 1.0
    no_ipd_length = period - ipd1 - ipd2
    T_anodal = no_ipd_length * ratioHi                  # period of anodal (positive) stimulation phase
    T_cathodal = no_ipd_length * ratioLo                # period of cathodal (negative) stimulation phase
    A_anodal = 1.0                                      # Anodal pulse is reference amplitude
    A_cathodal = -1* A_anodal*T_anodal/T_cathodal       # Cathodal pulse scaled to match reference amplitude
    amplitude = f * chargeperphase / ratioHi    # Real amplitude of anodal pulse is calculated as multiplication
    t_period = [0, T_anodal, T_anodal, T_anodal + ipd1, T_anodal + ipd1, T_anodal + ipd1 + T_cathodal, T_anodal + ipd1 + T_cathodal,  T_anodal + ipd1 + T_cathodal + ipd2]
    i_period = [A_anodal, A_anodal, 0, 0, A_cathodal, A_cathodal, 0, 0]
    return KHFACblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)


############################################################
# Creates a sampled square wave signal with interphase delays for Neuron
# ############################################################
# def squarewave_ip(t_start, t_stop, f, amplitude, ipd1, ipd2):
#     period_length = 1.0
#     no_ipd_length = period_length-ipd1-ipd2
#     t_period = [0, no_ipd_length/2, no_ipd_length/2, no_ipd_length/2+ipd1, no_ipd_length/2+ipd1, no_ipd_length+ipd1, no_ipd_length+ipd1, period_length]
#     i_period = [1, 1, 0, 0, -1, -1, 0, 0]
#     return KHFACblocksignal(t_start, t_stop, t_period, i_period, f, amplitude)


############################################################
# Creates a stepwave for Neuron, consisting of:
# t_period: the time vector of one period
# i_period: the amplitude vector of one period
############################################################
def stepwave(t_start, t_stop, f, amplitude, steps_per_quartercycle): #Trianglewave
    t_period = [0]
    i_period = [0]
    step_increment = 1.0/steps_per_quartercycle
    for i in range(0, steps_per_quartercycle*4):
        t_period = np.concatenate((t_period, [i,i+1]), axis=None)
        i_period = np.concatenate((i_period, [i_period[-1] + step_increment, i_period[-1] + step_increment]), axis = None)
        if abs(abs(i_period[-1]) - 1.0) <= 1.0/100000000:
            step_increment *= -1
    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)),
            ↪  np.sin(2*np.pi*i/(steps_per_quartercycle*4))]), axis = None)
137     return KHFACblocksignal(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,
           ↪  8.7, 10.0, 11.5, 12.8, 14.0, 15.0, 16.0
15         # SIMULATION PARAMETERS
16         'dt'                         : 0.005,    # Simulation timestep (in ms)
17         'tstop'                            : 10,       # Total simulation time (in ms)
18         # HFS PARAMETERS
19         'HFSreferenceNode'  : 2,  # Node of Ranvier where KHFAC stimulation is applied
20         'HFSamp'                   : 0.0,      # Amplitude of KHFAC waveform (mA)
21         'HFSfrequency'               : 10.0,     # Frequency of KHFAC waveform (kHz)
22         'HFSdelay'                      : 5.0,      # Time when KHFAC waveform starts (ms)
23         'HFSx'                             : 0.0,      # Location of the KHFAC electrode on the x-axis
24         'HFSy'                             : 0.0,      # Location of the KHFAC electrode on the y-axis
25         'HFSz'                             : 1000.0,   # Location of the KHFAC electrode on the z-axis
26         'tonset'                     : 10,               # Duration of onset response
27         'bipolar'           :  0,       # Set to 1 for bipolar electrode
28         # PARAMETERS FOR INTRACELLULAR STIMULUS
29         'intrinsicStim'     : 0,  # Change to 1 to add intrinsic pulse(train)
30     }
31
32     if config=='singleAPdemo':
33         # Create single AP
34         par.update({
35             # ENVIRONMENTAL PARAMETERS
36             'celsius'                    :  37.0,    # Temperature
37             'v_init'                     : -80.0,    # Initial membrane voltage
38             'axonnodes'                     : 21,       # Amount of Nodes of Ranvier on the axon
39             # SIMULATION PARAMETERS
40             'tstop'                                : 10,       # Total simulation time (in ms)
41             # PARAMETERS FOR INTRACELLULAR STIMULUS
42             'intrinsicStim'     : 1,        # Change to 1 to add intrinsic pulse(train)
43             'intrinsicType'               : 0,        # 0 is single pulse, 1 is pulsetrain
44             'intrinsicNode'              : 10.0,      # Node where intracellular stimulus starts
45             'intrinsicAmp'               : 1.0,      # Amplitude of intracellular stimulus (nA)
46             'intrinsicDel'               : 3.0,      # Time when intracellular stimulus starts (ms)
47             'intrinsicDur'               : 1.0,      # Duration of intracellular stimulus (ms)
               ↪  # ONLY SINGLE PULSE
48             'intrinsicTON'               : 1.0,      # Duration of ON phase intracellular stimulus (ms)
               ↪  # ONLY PULSE TRAIN
49             'intrinsicTOFF'               : 1.0,      # Duration of OFF phase intracellular stimulus
               ↪  (ms) # ONLY PULSE TRAIN
50             'intrinsicNum'               : None      # Number of intrinsic pulses (when doing a
               ↪  pulsetrain)
51         })
52
53     elif config=='HFSdemo':
54         par.update({
55             # [MRGnode]
56             # ENVIRONMENTAL PARAMETERS
57             'axonnodes'                : 51.0,     # Amount of Nodes of Ranvier on the axon
58             # SIMULATION PARAMETERS
59             'tstop'                                : 20,       # Total simulation time (in ms)
```

```
            # HFS PARAMETERS
            'HFSreferenceNode'        : 25,       # Node of Ranvier where KHFAC stimulation is applied (First node is 0)
            'HFSamp'                        : 1.0,      # Amplitude of KHFAC waveform (mA)
            'HFSfrequency'                : 10.0,     # Frequency of KHFAC waveform (kHz)
            'HFSdelay'                      : 2.0,      # Time when KHFAC waveform starts (ms)
            'HFSx'                            : 0.0,      # Location of the KHFAC electrode on the x-axis ()
            'HFSy'                            : 0.0,      # Location of the KHFAC electrode on the y-axis ()
            'HFSz'                            : 1000.0,   # Location of the KHFAC electrode on the z-axis ()
        })

    elif config=="ClassicModel":
        par.update({
            # [MRGnode]
            # ENVIRONMENTAL PARAMETERS
            'axonnodes'                : 51.0,     # Amount of Nodes of Ranvier on the axon
            # SIMULATION PARAMETERS
            'tstop'                           : 51,       # Total simulation time (in ms)
            'dt'                            : 0.001,    # Simulation timestep (in ms)
            # HFS PARAMETERS
            'HFSreferenceNode'        : 25,       # Node of Ranvier where KHFAC stimulation is applied (First node is 0)
            # 'HFSamp'                        : 1.5,      # Amplitude of KHFAC waveform (mA)
            # 'HFSfrequency'                : 10.0,     # Frequency of KHFAC waveform (kHz)
            'HFSdelay'                      : 0.0,      # Time when KHFAC waveform starts (ms)
            'HFSx'                            : 0.0,      # Location of the KHFAC electrode on the x-axis (um)
            'HFSy'                            : 0.0,      # Location of the KHFAC electrode on the y-axis (um)
            'HFSz'                            : 1000.0,   # Location of the KHFAC electrode on the z-axis (um)
            # PARAMETERS FOR INTRACELLULAR STIMULUS
            'intrinsicStim'                 : 1,        # Change to 1 to add intrinsic pulse(train)
            'intrinsicType'                 : 1,        # 0 is single pulse, 1 is pulsetrain
            'intrinsicNode'                 : 0,        # Node where intracellular stimulus starts
            'intrinsicAmp'                : 2.0,      # Amplitude of intracellular stimulus (nA)
            'intrinsicDel'                : 10.0,      # Time when intracellular stimulus starts (ms)
            'intrinsicDur'                : 0.1,      # Duration of intracellular stimulus (ms) # ONLY SINGLE PULSE
            'intrinsicTON'                : 0.5,      # Duration of ON phase intracellular stimulus (ms) # ONLY PULSE TRAIN
            'intrinsicTOFF'                 : 1.5,      # Duration of OFF phase intracellular stimulus (ms) # ONLY PULSE TRAIN
            'intrinsicNum'                : None      # Number of intrinsic pulses (when doing a pulsetrain)
        })
    elif config=="ClassicModel_Bipolar":
        par.update({
            # [MRGnode]
            # ENVIRONMENTAL PARAMETERS
            'axonnodes'                : 51.0,     # Amount of Nodes of Ranvier on the axon
            # SIMULATION PARAMETERS
            'tstop'                           : 51,       # Total simulation time (in ms)
            'dt'                            : 0.001,    # Simulation timestep (in ms)
            # HFS PARAMETERS
            'HFSreferenceNode'        : 25,       # Node of Ranvier where KHFAC stimulation is applied (First node is 0)
            # 'HFSamp'                        : 1.5,      # Amplitude of KHFAC waveform (mA)
            # 'HFSfrequency'                : 10.0,     # Frequency of KHFAC waveform (kHz)
            'HFSdelay'                      : 0.0,      # Time when KHFAC waveform starts (ms)
            'HFSx'                            : 0.0,      # Location of the KHFAC electrode on the x-axis (um)
            'HFSy'                            : 0.0,      # Location of the KHFAC electrode on the y-axis (um)
            'HFSz'                            : 1000.0,   # Location of the KHFAC electrode on the z-axis (um)
            'bipolar'                       : 1,  # Set to 1 for bipolar stimulation
            'HFSx2'                           : 0.0,      # Location of the KHFAC electrode on the x-axis (um)
            'HFSy2'                           : 0.0,      # Location of the KHFAC electrode on the y-axis (um)
            'HFSz2'                         : 1000.0,   # Location of the KHFAC electrode on the z-axis (um)
            # PARAMETERS FOR INTRACELLULAR STIMULUS
            'intrinsicStim'                 : 1,        # Change to 1 to add intrinsic pulse(train)
```

```
            'intrinsicType'                 : 1,        # 0 is single pulse, 1 is pulsetrain
            'intrinsicNode'                 : 0,        # Node where intracellular stimulus starts
            'intrinsicAmp'                  : 2.0,      # Amplitude of intracellular stimulus (nA)
            'intrinsicDel'                  : 10.0,      # Time when intracellular stimulus starts (ms)
            'intrinsicDur'                  : 0.1,      # Duration of intracellular stimulus (ms)
            # ONLY SINGLE PULSE
            'intrinsicTON'                  : 0.5,      # Duration of ON phase intracellular stimulus (ms)
            # ONLY PULSE TRAIN
            'intrinsicTOFF'                 : 1.5,      # Duration of OFF phase intracellular stimulus
            (ms) # ONLY PULSE TRAIN
            'intrinsicNum'                  : None      # Number of intrinsic pulses (when doing a
            pulsetrain)
        })
    elif config=="ProposedModel":
        # Create simplified Model
        par.update({
            # [MRGnode]
            # ENVIRONMENTAL PARAMETERS
            'axonnodes'                         : 5,       # Amount of Nodes of Ranvier on the axon
            (standard 5)
            # SIMULATION PARAMETERS
            'dt'                                : 0.001,    # Simulation timestep (in ms)
            'tstop'                             : 20.0,     # Total simulation time (in ms) (standard
            2)
            # HFS PARAMETERS
            # 'HFSamp'                        : 0.5,      # Amplitude of KHFAC waveform (mA)
            # 'HFSfrequency'                : 10.0,     # Frequency of KHFAC waveform (kHz)
            'HFSdelay'                      : 0.0,      # Time when KHFAC waveform starts (ms)
            'HFSx'                              : 0.0,      # Location of the KHFAC electrode on the
            x-axis ()
            'HFSy'                              : 0.0,      # Location of the KHFAC electrode on the
            y-axis ()
            'HFSz'                              : 1000.0,   # Location of the KHFAC electrode on the
            z-axis ()
            'tonset'                        : 18,               # Duration of onset response (standard
            2)
        })
    elif config == "ProposedModel_Bipolar":
        # Create simplified Model
        par.update({
            # [MRGnode]
            # ENVIRONMENTAL PARAMETERS
            'axonnodes'                 : 5,       # Amount of Nodes of Ranvier on the axon (standard 5)
            # SIMULATION PARAMETERS
            'dt'                                : 0.001,    # Simulation timestep (in ms)
            'tstop'                             : 20.0,     # Total simulation time (in ms) (standard
            2)
            # HFS PARAMETERS
            # 'HFSamp'                        : 0.5,      # Amplitude of KHFAC waveform (mA)
            # 'HFSfrequency'                : 10.0,     # Frequency of KHFAC waveform (kHz)
            'HFSdelay'                      : 0.0,      # Time when KHFAC waveform starts (ms)
            'HFSx'                              : 0.0,      # Location of the KHFAC electrode on the
            x-axis ()
            'HFSy'                              : 0.0,      # Location of the KHFAC electrode on the
            y-axis ()
            'HFSz'                              : 1000.0,   # Location of the KHFAC electrode on the
            z-axis ()
            'tonset'                        : 18,               # Duration of onset response (standard
            2)
            'bipolar'           : 1,        # Set to 1 for bipolar stimulation
            'HFSx2'                             : 0.0,      # Location of the KHFAC electrode on the
            x-axis
            'HFSy2'                             : 0.0,      # Location of the KHFAC electrode on the
            y-axis
            'HFSz2'                         : 1000.0,   # Location of the KHFAC electrode on the
            z-axis ()
        })
        par['HFSreferenceNode'] = int(math.floor(par['axonnodes']/2))  # Node of Ranvier where KHFAC
        stimulation is applied (first node = 0)


    if par['intrinsicStim'] == 1:
        par['intrinsicNum'] = int(par['tstop'] / (par['intrinsicTON'] + par['intrinsicTOFF']))

    recpar = {  # Parameters used for recording
        # 'record'                        : True,
        # 'plot'                              : False,
        'nodes'                         : np.array(range(0, int(par['axonnodes']))  ), # Nodes that
        are to be recorded
```

```
        'recordVoltage'                 : True,     # Switch to allow recording of voltages
    }

    return par, recpar
```

MRG_Model.py

```
##########################################################################################################
# INTERFACE FUNCTIONS FOR PYTHON<->NEURON
# This is an adopted version from the work of Baquer Gomez (not published in thesis):
# https://repository.tudelft.nl/islandora/object/uuid%3A5c875b14-08a3-4ef0-8e90-4fc997dc19d3
# Extended by Koen Emmer to record gate variables, all other credits go to Baquer Gomez and the
# names mentioned in the code
##########################################################################################################
import sys
sys.path.append("<path>")
from neuron import h
import numpy as np


##########################################################################################################
# Functions that load variables into Neuron
##########################################################################################################
def pass_parameters_to_nrn(parameters):#, verb=False): # USED
    '''
    Passes parameters from a dictionary to NEURON.
    If the the element is a vector it assumes that the a vector
    has been created as objref and new Vector() in the hoc code.
    Set 'verb' to True for verbose.
    '''
    for k, v in parameters.iteritems():
        if type(v) is not type(np.array([])):
            h("{" + k + " = " + str(v) + "}")
            # if verb:
            #     print(k + " = " + str(v))


##########################################################################################################
# Functions that initialize the axon in Neuron
##########################################################################################################
def createMRGaxon(par):#, verbose): # USED
    '''
    Initializes the model.
    Creates the axon and stimulation according to the parameters.
    '''
    h('{load_file("stdgui.hoc")}')                  # Starts the Neuron session
    pass_parameters_to_nrn(par)#, verb=verbose)       # Passes the configuration file
    h('{load_file("MRG_MODEL_edit.hoc")}')          # Loads the hoc file of the model
    h('{buildModel()}')                             # Runs the buildmodel function of the hoc file,
    ↪  creating the axon


##########################################################################################################
# Functions that set up recording process in Neuron
##########################################################################################################
def recordMRGaxon(recpar):#, verbose):
    '''
    Inserts the recorders as specified in recpar.
    '''
    k = recpar['nodes']                                         # List with al the nodes [0, 1, 2, ...,
    ↪  N]
    rec = {}

    if recpar['recordVoltage']:
        rec['voltage'] = record_node_voltage(k)                 # Maybe not used when looking for gating
        ↪  variables
    rec['gates'] = record_node_gates(k)
    rec['current'] = record_node_current(k)
    rec['spiketimes'], rec['apcount'] = record_node_spikes(k)
    # if verbose:
    #     print('Now recording from ' + str(k))
    return rec


def record_node_voltage(nodenumber, rec=None):
    '''
```

```
    Records the membrane potential of a particular set of nodes.
    '''
    segments = []
    for n in nodenumber:
        segments.append(h.node[n](0.5))
    for seg, n in zip(segments, nodenumber):
        rec = insert_nrn_recorders(seg, {'v_node' + str(n): '_ref_v'}, rec) # Another function layer; necessary?
    return rec


def record_node_gates(nodenumber, rec=None):
    segments = []
    for n in nodenumber:
        segments.append(h.node[n](0.5))
    for seg, n in zip(segments, nodenumber):
        rec = insert_nrn_recorders(seg, {'h_node' + str(n): '_ref_h_axnode'}, rec)      # Nonlinear fast sodium inactivation gate
        rec = insert_nrn_recorders(seg, {'m_node' + str(n): '_ref_m_axnode'}, rec)      # Nonlinear fast sodium activation gate
        rec = insert_nrn_recorders(seg, {'mp_node' + str(n): '_ref_mp_axnode'}, rec)    # Persistent sodium activation gate
        rec = insert_nrn_recorders(seg, {'s_node' + str(n): '_ref_s_axnode'}, rec)      # Slow potassium activation gate
    return rec


def record_node_current(nodenumber, rec=None):
    segments = []
    for n in nodenumber:
        segments.append(h.node[n](0.5))
    for seg, n in zip(segments, nodenumber):
        rec = insert_nrn_recorders(seg, {'i_Na_node' + str(n): '_ref_ina_axnode'}, rec)
        rec = insert_nrn_recorders(seg, {'i_Nap_node' + str(n): '_ref_inap_axnode'}, rec)
        rec = insert_nrn_recorders(seg, {'i_K_node' + str(n): '_ref_ik_axnode'}, rec)
        rec = insert_nrn_recorders(seg, {'i_l_node' + str(n): '_ref_il_axnode'}, rec)

        rec = insert_nrn_recorders(seg, {'m_inf_node' + str(n): '_ref_m_inf_axnode'}, rec)
        rec = insert_nrn_recorders(seg, {'tau_m_node' + str(n): '_ref_tau_m_axnode'}, rec)
        rec = insert_nrn_recorders(seg, {'h_inf_node' + str(n): '_ref_h_inf_axnode'}, rec)
        rec = insert_nrn_recorders(seg, {'tau_h_node' + str(n): '_ref_tau_h_axnode'}, rec)
    return rec


def record_node_spikes(nodenumber, rec=None,
                       apc=None, threshold = -15):
    '''
    Records the action potentials of a particular set of nodes.
    Returns a "rec" dictionary.
    '''
    if rec is None:
        rec = {}
    if apc is None:
        apc = {}
    for n in nodenumber:
        apc['apc'+str(n)] = h.APCount(h.node[int(n)](0.5))
        apc['apc'+str(n)].thresh = threshold
        rec['spk'+str(n)] = h.Vector()
        apc['apc'+str(n)].record(rec['spk'+str(n)])
    return rec,apc


def insert_nrn_recorders(segment, labels, rec=None):
    '''
    Inserts recorders for NEURON state variables.
    Use one per segment.
    "labels" is a dictionary.
    Example {'v': '_ref_v'}.
    Specify 'rec' to append to previous recorders.
    Records also time if 'rec' is 'None' (default).
    (Acknowledgements: Daniele Linaro)
    '''
    if rec is None:
        rec = {'t': h.Vector()}
        rec['t'].record(h._ref_t)
    for k, v in labels.items():
        rec[k] = h.Vector()
        rec[k].record(getattr(segment, v))
```

```
    return rec


##########################################################################################################
# Functions run a simulation of the axon
##########################################################################################################
def runMRGaxon(rec, ts, xs):
    h.resetModel()
    tvec = h.Vector(len(ts)) #time vector
    for i, t in enumerate(ts):
        tvec.x[i] = t
    xvec = h.Vector(len(ts)) #amplitude vector
    for i, x in enumerate(xs):
        xvec.x[i] = x
    xvec.play(h.exIClmp._ref_i, tvec, 1)
    h.run()
    rec['i_block'] = {'i': [], 't': rec['voltage']['t']}
    rec['i_block']['i'] = h.rec_electrode_block
    rec['i_stim'] = {'i': [], 't': rec['voltage']['t']}
    rec['i_stim']['i'] = h.rec_electrode_stim
    return rec


##########################################################################################################
# Functions that reset values between simulations
##########################################################################################################
def updateMRGaxon(par):#, verbose):
    '''
    Updates the parameters of the model.
    '''
    pass_parameters_to_nrn(par)#, verb=verbose)
    h.resetModel()

def quitNeuron():
    h.quit()
```

MRG_MODEL_edit.hoc

```
/*----------------------------------------------------------------------
Koen Emmer
SIMULATION OF PNS MYELINATED AXON WITH EXTRACELLULAR AND INTRACELLULAR ELECTRODE.

This model is an adapted version of MRGaxon.hoc:
Cameron C. McIntyre
SIMULATION OF PNS MYELINATED AXON
This model is described in detail in:
McIntyre CC, Richardson AG, and Grill WM. Modeling the excitability of
mammalian nerve fibers: influence of afterpotentials on the recovery
cycle. Journal of Neurophysiology 87:995-1006, 2002.

The basis of this adapted version was a version made by Joao Couto, that can be found here:
https://github.com/jcouto/neuron/blob/master/MRGaxon/MRGnodeHFS.hoc
----------------------------------------------------------------------
----------------------------------------------------------------------
stdgui (or nrnmainmenu) needs to be loaded first from Python to correctly work. Next parameters can be passed to Neuron, and then this .hoc file can be opened.
The following variables can be set://
    axonnodes                - number of nodes that are going to be used in the simulation
    fiberD                       - diameter of the fiber, can be set to: 5.7, 7.3, 8.7, 10.0, 11.5, 12.8, 14.0, 15.0 or 16.0

ipulse1.mod and axnode.mod are necessary for the model to work:
    ipulse1.mod:
    https://senselab.med.yale.edu/ModelDB/ShowModel.cshtml?model=225086&file=%2FGC-gna_distribution%2Fipulse1.mod#tabs-2
    axnode.mod can be found with the MRG model on ModelDB
----------------------------------------------------------------------*/

proc model_globals() {
        //celsius = 37
    //v_init = -80
    //dt
    //tstop

    //---------------TOPOLOGICAL PARAMETERS------------------------------
    // FULL SEGMENT = -NODE-MYSA-FLUT-STIN-STIN-STIN-STIN-STIN-STIN-FLUT-MYSA-NODE-
    paranodes1 = 2*(axonnodes-1)                                // Number of MYSA paranodes
```

```
    paranodes2 = 2*(axonnodes-1)                            // Number of FLUT paranodes
    axoninter = 6*(axonnodes-1)                             // Number of STIN internodes
    axontotal = axonnodes+paranodes1+paranodes2+axoninter   // Total amount of segments

    //---------------MORPHOLOGICAL PARAMETERS----------------------------
    // All morphological parameters are in um (micrometers)
    //fiberD=5.7            // Fiber Diameter, choose from 5.7, 7.3, 8.7, 10.0, 11.5, 12.8, 14.0, 15.0, 16.0
    paralength1 = 3     // MYSA Length
    nodelength = 1.0    // Node length
    space_p1 = 0.002    // MYSA periaxonal space width
    space_p2 = 0.004    // FLUT periaxonal space width
    space_i = 0.004     // STIN periaxonal space width

    //---------------ELECTRICAL PARAMETERS--------------------------------
    rhoa = 0.7e6                            // Axoplasmic resistivity               // Ohm-um //
    mycm = 0.1                                  // Myelin capacitance                   //
    uF/cm2/lamella membrane // 2 membranes per lamella
    mygm = 0.001                            // Myelin conductance                   //  S/cm2/lamella membrane // 2 membranes per lamella
    rhoe = 5e6                                  // Resistivity of extracellular medium  // Ohm-um//

    // DELETE EVERYTHING BENEATH?
    //-----------------------RECORDING-----------------------------------

    //recordAll = 0   /*Uncomment to record voltage from all nodes.*/

}
model_globals ()

//---------------INITIALIZE THE DEPENDENT VARIABLES---------------------
proc dependent_var() {
    // The fiber diameter can not be changed in the middle of the simulation.
    /*  fiberD = Fiber diameter          g =                                     deltax = Node-node separation
         axonD = Axon diameter      paraD1 = MYSA diameter                paralength2 = FLUT length
         nodeD = Node diameter      paraD2 = FLUT diameter (same as STIN)          nl = number of myelin lemella */
    if (fiberD==5.7)  {g=0.605 axonD=3.4  nodeD=1.9 paraD1=1.9 paraD2=3.4  deltax=500  paralength2=35 nl=80 }
    if (fiberD==7.3)  {g=0.630 axonD=4.6  nodeD=2.4 paraD1=2.4 paraD2=4.6  deltax=750  paralength2=38 nl=100}
    if (fiberD==8.7)  {g=0.661 axonD=5.8  nodeD=2.8 paraD1=2.8 paraD2=5.8  deltax=1000 paralength2=40 nl=110}
    if (fiberD==10.0) {g=0.690 axonD=6.9  nodeD=3.3 paraD1=3.3 paraD2=6.9  deltax=1150 paralength2=46 nl=120}
    if (fiberD==11.5) {g=0.700 axonD=8.1  nodeD=3.7 paraD1=3.7 paraD2=8.1  deltax=1250 paralength2=50 nl=130}
    if (fiberD==12.8) {g=0.719 axonD=9.2  nodeD=4.2 paraD1=4.2 paraD2=9.2  deltax=1350 paralength2=54 nl=135}
    if (fiberD==14.0) {g=0.739 axonD=10.4 nodeD=4.7 paraD1=4.7 paraD2=10.4 deltax=1400 paralength2=56 nl=140}
    if (fiberD==15.0) {g=0.767 axonD=11.5 nodeD=5.0 paraD1=5.0 paraD2=11.5 deltax=1450 paralength2=58 nl=145}
    if (fiberD==16.0) {g=0.791 axonD=12.7 nodeD=5.5 paraD1=5.5 paraD2=12.7 deltax=1500 paralength2=60 nl=150}

    // Inverses of Gp:
    Rpn0 = (rhoa*.01)/(PI*((((nodeD/2)+space_p1)^2)-((nodeD/2)^2)))             // Periaxonal resistance NODE
    Rpn1 = (rhoa*.01)/(PI*((((paraD1/2)+space_p1)^2)-((paraD1/2)^2)))           // Periaxonal resistance MYSA
    Rpn2 = (rhoa*.01)/(PI*((((paraD2/2)+space_p2)^2)-((paraD2/2)^2)))           // Periaxonal resistance FLUT
    Rpx  = (rhoa*.01)/(PI*((((axonD/2)+space_i)^2)-((axonD/2)^2)))              // Periaxonal resistance STIN

    interlength = (deltax-nodelength-(2*paralength1)-(2*paralength2))/6
}
dependent_var()

//---------------CREATE SECTIONS AND APPEND CHANNELS-----------------
create node[axonnodes], MYSA[paranodes1], FLUT[paranodes2], STIN[axoninter]
access node[0]
objref s[axontotal]
create electrode
not_built = 1

```

```
proc initMRG() {
    //print "Initializing MRGaxon."
    for i=0,axonnodes-1 {
                node[i]{
                    s[i] = new SectionRef()
                    nseg = 1
                    diam = nodeD
                    L = nodelength
                    Ra = rhoa/10000
                    cm = 2
                    insert axnode
                    insert extracellular xraxial=Rpn0 xg=1e10 xc=0
                }
    }
    for i=0, paranodes1-1 {
        MYSA[i]{
            s[axonnodes+i] = new SectionRef()
            nseg = 1
            diam = fiberD
            L = paralength1
            Ra = rhoa*(1/(paraD1/fiberD)^2)/10000
            cm = 2*paraD1/fiberD
            insert pas
            g_pas = 0.001*paraD1/fiberD
            e_pas = v_init
            insert extracellular xraxial=Rpn1 xg=mygm/(nl*2) xc=mycm/(nl*2)
        }
    }
    for i=0, paranodes2-1 {
        FLUT[i]{
            s[axonnodes+paranodes1+i] = new SectionRef()
            nseg = 1
            diam = fiberD
            L = paralength2
            Ra = rhoa*(1/(paraD2/fiberD)^2)/10000
            cm = 2*paraD2/fiberD
            insert pas
            g_pas = 0.0001*paraD2/fiberD
            e_pas = v_init
            insert extracellular xraxial=Rpn2 xg=mygm/(nl*2) xc=mycm/(nl*2)
        }
    }
    for i=0, axoninter-1 {
        STIN[i]{
            s[axonnodes+paranodes1+paranodes2+i]=new SectionRef()
            nseg = 1
            diam = fiberD
            L = interlength
            Ra = rhoa*(1/(axonD/fiberD)^2)/10000
            cm = 2*axonD/fiberD
            insert pas
            g_pas = 0.0001*axonD/fiberD
            e_pas = v_init
            insert extracellular xraxial=Rpx xg=mygm/(nl*2) xc=mycm/(nl*2)
        }
    }
    for i=0, axonnodes-2 {
        connect MYSA[2*i](0), node[i](1)
        connect FLUT[2*i](0), MYSA[2*i](1)
        connect STIN[6*i](0), FLUT[2*i](1)
        connect STIN[6*i+1](0), STIN[6*i](1)
        connect STIN[6*i+2](0), STIN[6*i+1](1)
        connect STIN[6*i+3](0), STIN[6*i+2](1)
        connect STIN[6*i+4](0), STIN[6*i+3](1)
        connect STIN[6*i+5](0), STIN[6*i+4](1)
        connect FLUT[2*i+1](0), STIN[6*i+5](1)
        connect MYSA[2*i+1](0), FLUT[2*i+1](1)
        connect node[i+1](0), MYSA[2*i+1](1)
    }
}

proc initialize() {
    finitialize(v_init)
    fcurrent()
}

//---------------VECTORS AND OBJECT VARIABLES INITIALIZATION----------
objref Ve, X, Y, Z, exIClmp, inIClmp, apc, rec[axonnodes], rec_spk, v_first, v_ref, v_last, rec_Ve
```

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

```
    // UNCOMMENT TO RECORD ELECTRODE
    rec_electrode_block = new Vector()
    rec_electrode_block.record(&exIClmp.i)
    rec_electrode_block.clear()
    //

    //---------------INITIALIZE THE INTRACELLULAR STIM----------
    if(intrinsicStim == 1){
        node[intrinsicNode+1] {
        // inject pattern in one of the ends of the axon.
            if(intrinsicType == 0){
                inIClmp = new IClamp(0.5)
                inIClmp.del = intrinsicDel      // The delay until the onset of the stimulus (in ms)
                inIClmp.dur = intrinsicDur      // The duration of the stimulus (in ms)
                inIClmp.amp = intrinsicAmp      // The amplitude of the stimulus (in nA)
                }else if(intrinsicType == 1){
                inIClmp = new Ipulse1(0.5)
                inIClmp.del = intrinsicDel      // The delay until the onset of the stimulus (in ms)
                inIClmp.num = intrinsicNum      // The duration of the stimulus (in ms)
                inIClmp.amp = intrinsicAmp      // The amplitude of the stimulus (in nA)
                inIClmp.ton = intrinsicTON      // The duration of ON phase (in ms)
                inIClmp.toff = intrinsicTOFF    // The duration of OFF phase (in ms)
                }
            }
            rec_electrode_stim = new Vector()
        rec_electrode_stim.record(&inIClmp.i)       // inIClmp.i = current ic inIClmp
            rec_electrode_stim.clear()
            }
}

//---------------DEFINES WHAT TO DO AT EACH TIMESTEP--------------
proc advance() { local i
    // Set Vextracellular using neurons mechanisms.
    for ii=0,axontotal-1 {
            s[ii].sec.e_extracellular(0.5)=(exIClmp.i)*Ve.x[ii]     // Set the extracellular voltages at
             ↪  each segment
    }
    fadvance()
}


proc makeRecorders() {
    v_first = new Vector()
    v_ref = new Vector()
    v_last = new Vector()
    v_first.record(&node[0].v(0.5))
    v_ref.record(&node[HFSreferenceNode].v(0.5))
    v_last.record(&node[axonnodes-1].v(0.5))
    //---------------RECORD ACTION POTENTIAL TIMESTAMPS---------
    node[axonnodes-1] {
        apc = new APCount(0.5)
        apc.thresh = -15
        rec_spk = new Vector()
        apc.record(rec_spk)
    }//RECORDS INTO REC_SPK

    //---------------RECORD ALL VOLTAGE TRACES (if recordAll)---------
    if(recordAll) {
        for ii=0,axonnodes-1 {
            rec[ii] = new Vector()
            rec[ii].record(&node[ii].v(0.5))
        }
    }
}

// Build the model
proc buildModel() {
    if (not_built) {
                initMRG()
                XYZ_loc()                   // Storing locations of the segments
                stimulus()
                initialize()
                not_built=0
        }
}

// Reset the model
proc resetModel() {
```

```
    XYZ_loc()
    stimulus()
    initialize()
    init()
    //print "Reset MRGaxon."
}
```

datasetproposedmethod.py

```
from simulationfunctions_proposedmodel import *
import os
if not os.path.exists('justification_files'):
    os.makedirs('justification_files')          # Ensure that brainframe makes justification folder


############################################################
# Build dataset for square wave tested with Proposed Model
############################################################
def squarewaveresults():
    frequencylist = [3]
    for i in range(19):
        frequencylist.append((i + 2) * 2)

    result = find_blockingamplitudes_Freq(frequencylist, "square")

    filename = 'justification_files/results_squarewave_proposedmethod'

    np.save(filename, result)
    CSVsave_blockingamplitudes_2D(result, "Frequency", "Frequency (kHz)", filename)


############################################################
# Build dataset for sine wave tested with Proposed Model
############################################################
def sineresults():
    frequencylist = [3]
    for i in range(19):
        frequencylist.append((i + 2) * 2)

    result = find_blockingamplitudes_Freq(frequencylist, "sine")

    now = datetime.datetime.now()
    # filename = 'results/results_' + now.strftime("%Y%m%d_%H%M%S")
    filename = 'justification_files/results_sinewave_proposedmethod' + now.strftime("%Y%m%d_%H%M%S")

    np.save(filename, result)
    CSVsave_blockingamplitudes_2D(result, "Frequency", "Frequency (kHz)", filename)


############################################################
# Build dataset for assymmetricalwave tested with Proposed Model
############################################################
def assymmetricalresults():
    frequencylist = [3]
    for i in range(19):
        frequencylist.append((i + 2) * 2)
    chargebalancelist = [0.1]
    for i in range(9):
        chargebalancelist.append(round(0.1 + 0.1*i, 1))
    result = find_blockingamplitudes_CB(frequencylist, chargebalancelist)

    filename = 'justification_files/results_squarewave_CB_proposedmethod'

    np.save(filename, result)
    CSVsave_blockingamplitudes_3D(result, "Frequency", "Frequency (kHz)", "Chargeperphase", "Charge per
    ↪  Phase (uC)", filename)
```

simulationfunctions.py

```
############################################################
# This file contains simulation functions.
############################################################
# Import Python libraries
import datetime
```

```
import csv
import pylab as plt
import os
if not os.path.exists('results'):
    os.makedirs('results')          # Ensure that brainframe makes result folder
# Import custom function libraries
from MRG_Model import *             # Interface with Neuron MRG Model
from plottingfunctions import *     # Plotting functions
from blockdetection import *
from waveforms import *             # Waveform functions for Neuron
import time


############################################################
# Functions to save data to CSV
############################################################
def savetocsv(result, now = datetime.datetime.now()):
    with open('results/results_' + now.strftime("%Y%m%d_%H%M%S") + '.csv', mode='w') as csv_file:
        fieldnames = ["Node", "Frequency (kHz)", "Amplitude (mA)", "Block %",
                      "h_max", "h_min", "V_max", "V_min", "m_max", "m_min", "h_dist_max", "h_dist_min"]
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
        writer.writeheader()

        for i in result:
            for j in result[i]:
                for k in result[i][j]:
                    if result[i][j][k] != {}:                                                   # Empty dicts should be ignored
                        result[i][j][k]["Frequency (kHz)"] = result[i][j][k].pop("Frequency")   # Needed to make compliant to fieldnames of CSV
                        result[i][j][k]["Amplitude (mA)"] = result[i][j][k].pop("Amplitude")    # Needed to make compliant to fieldnames of CSV
                        writer.writerow(result[i][j][k])


# Asymmetrical waves csv-save function
def savetocsv2(result, now = datetime.datetime.now()):

    with open('results/results_' + now.strftime("%Y%m%d_%H%M%S") + '.csv', mode='w') as csv_file:
        fieldnames = ["Node", "Frequency (kHz)", "Chargebalance", "Charge per Phase (uC)", "Amplitude (mA)", "Block %",
                      "h_max", "h_min", "V_max", "V_min", "m_max", "m_min", "h_dist_max", "h_dist_min"]
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
        writer.writeheader()

        for i in result:
            for j in result[i]:
                for k in result[i][j]:
                    for l in result[i][j][k]:
                            if result[i][j][k][l] != {}:
                                result[i][j][k][l]["Frequency (kHz)"] = result[i][j][k][l].pop("Frequency")             # Needed to make compliant to fieldnames of CSV
                                result[i][j][k][l]["Charge per Phase (uC)"] = result[i][j][k][l].pop("Chargeperphase")  # Needed to make compliant to fieldnames of CSV
                                result[i][j][k][l]["Amplitude (mA)"] = result[i][j][k][l].pop("Amplitude")              # Needed to make compliant to fieldnames of CSV
                                writer.writerow(result[i][j][k][l])


############################################################
# This functions runs simulations and returns the results as a dataset
# Find blockingamplitudes, sweep over frequency
############################################################
def run_and_save_FreqAmp(par, recpar, nodes, frequencylist, amplitudelist, waveform="square", config="ClassicModel", runtype="fast", savetype = "saveall"):
    # Run simulations for all frequencies and amplitudes and save data of nodes specified
    # Time Process
    starttime = time.time()

    # Create the axon
    createMRGaxon(par)  # , verbose)
    intervaltime = time.time() - starttime
    print("--- CREATEAXON: %s seconds ---" % (intervaltime))
    intervaltime = time.time()

```

```
    amplitude_counter = 0        # Needed for fast simulations
    # INITIALIZE RESULT ARRAY
    result = {}
    for f in frequencylist:
        result[f] = {}
        par['HFSfrequency'] = f
        for a in amplitudelist[amplitude_counter:]:
            rec = recordMRGaxon(recpar)
            # RUN SIMULATION
            par['HFSamp'] = a

            if waveform == "sine":
                t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                ↪  par['HFSamp'], 1000)
            elif waveform == "square":
                t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                ↪  par['HFSamp'])
            else:
                print "Incompatible waveform selected!"
                return None

            intervaltime = time.time() - intervaltime
            print("--- BUILD AND RUN HFS-SIGNAL (%s kHz, %s mA) : %s seconds ---" % (f, a, intervaltime))
            intervaltime = time.time()

            updateMRGaxon(par)  # , verbose)
            runMRGaxon(rec, t_signal, i_signal)

            if config == "ClassicModel":
                effectiveness = blockeffectiveness(rec, par)
            elif is_blocked(rec, par):
                effectiveness = 1.0
            else:
                effectiveness = 0.0


            # STORE RESULTS
            if (effectiveness >= 0.90 and savetype == "saveblock") or savetype == "saveall":
                result[f][a] = {}
                for n in nodes:
                    result[f][a][n] = {}
                    h_max, h_min, V_max, V_min, m_max, m_min, h_dist_max, h_dist_min =
                    ↪  conductionBlock(rec, par, n)

                    result[n][f][a]["Node"] = n
                    result[n][f][a]["Frequency"] = f
                    result[n][f][a]["Amplitude"] = a
                    result[n][f][a]["Block %"] = effectiveness

                    result[n][f][a]["h_max"] = h_max
                    result[n][f][a]["h_min"] = h_min
                    result[n][f][a]["V_max"] = V_max
                    result[n][f][a]["V_min"] = V_min
                    result[n][f][a]["m_max"] = m_max
                    result[n][f][a]["m_min"] = m_min
                    result[n][f][a]["h_dist_max"] = h_dist_max
                    result[n][f][a]["h_dist_min"] = h_dist_min
                    # result[n][f][a]["rec"] = rec

            # Code for fast runtype (skips amplitudes of which is known that they are not effective)
            if runtype == "fast":
                if effectiveness >= 0.90:
                    amplitude_counter = max(0, amplitude_counter - 5)  # Optional
                    break

                elif amplitude_counter >= len(amplitudelist) - 1:
                    amplitude_counter = 0
                    break

                amplitude_counter += 1

    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))

    return result


############################################################
```

```python
# This functions runs asymmetrical wave simulations and returns the results as a dataset
# Find blockingamplitudes, sweep over frequency and anode fraction
#############################################################
def run_and_save_assymmetrical(par, recpar, nodes, frequencylist, amplitudelist, chargebalancelist, config="ClassicModel", runtype="fast", savetype="saveall"):
    # Run simulations for all frequencies and amplitudes and save data of nodes specified

    # Time Process
    starttime = time.time()

    # Create the axon
    createMRGaxon(par)  # , verbose)
    intervaltime = time.time() - starttime
    print("--- CREATEAXON: %s seconds ---" % (intervaltime))
    intervaltime = time.time()

    amplitude_counter = {}
    for i in chargebalancelist:
        amplitude_counter[i] = 0

    result = {}
    for b in chargebalancelist:
        result[b] = {}
        for f in frequencylist:
            result[b][f] = {}
            par['HFSfrequency'] = f
            for a in amplitudelist[amplitude_counter[b]:]:
                rec = recordMRGaxon(recpar)  # verbose)
                par['HFSamp'] = a
                # Transform amplitude into charge; amplitude is actually only the amplitude when chargebalance is 0.5
                cpp = a/(2*f)           # Charge per phase is amplitude times half the period

                intervaltime = time.time() - intervaltime
                print("--- BUILD AND RUN HFS-SIGNAL (%s kHz, %s mA, %s chargebalance): %s seconds ---" % (f, a, b, intervaltime))
                intervaltime = time.time()

                # RUN SIMULATION
                t_signal, i_signal = chargebalanced_asymmetrical(par['HFSdelay'], par['tstop'], par['HFSfrequency'], cpp, b) # Build signal
                updateMRGaxon(par)
                runMRGaxon(rec, t_signal, i_signal)

                if config == "ClassicModel":
                    effectiveness = blockeffectiveness(rec, par)
                elif is_blocked(rec, par):
                    effectiveness = 1.0
                else:
                    effectiveness = 0.0

                # STORE RESULTS
                if (effectiveness >= 0.90 and savetype == "saveblock") or savetype == "saveall":
                    result[b][f][a] = {}
                    for n in nodes:
                        result[b][f][a][n] = {}
                        h_max, h_min, V_max, V_min, m_max, m_min, h_dist_max, h_dist_min = conductionBlock(rec, par, n)

                        result[b][f][a][n]["Node"] = n
                        result[b][f][a][n]["Frequency"] = f
                        result[b][f][a][n]["Chargebalance"] = b
                        result[b][f][a][n]["Chargeperphase"] = cpp
                        result[b][f][a][n]["Amplitude"] = a
                        result[b][f][a][n]["Block %"] = effectiveness

                        result[b][f][a][n]["h_max"] = h_max
                        result[b][f][a][n]["h_min"] = h_min
                        result[b][f][a][n]["V_max"] = V_max
                        result[b][f][a][n]["V_min"] = V_min
                        result[b][f][a][n]["m_max"] = m_max
                        result[b][f][a][n]["m_min"] = m_min
                        result[b][f][a][n]["h_dist_max"] = h_dist_max
                        result[b][f][a][n]["h_dist_min"] = h_dist_min

                # Code for fast runtype (skips amplitudes of which is known that they are not effective)
                if runtype == "fast":
                    if effectiveness >= 0.90:
```

```
                        amplitude_counter[b] = max(0, amplitude_counter - 5)  # Optional
                        break

                    elif amplitude_counter[b] >= len(amplitudelist) - 1:
                        if f == min(frequencylist):
                            amplitude_counter[b] = 0
                        else:
                            amplitude_counter[b] = max(0, amplitude_counter[b] - 5)
                        break

                    amplitude_counter[b] += 1


    totaltime = time.time() - starttime
    print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))

    return result
```

simulationfunctions_proposedmodel.py

```
import csv
from configoptions import *
import time
import datetime
from MRG_Model import *              # Interface with Neuron MRG Model
from blockdetection import *
from waveforms import *              # Waveform functions for Neuron

############################################################
# Functions to save data to CSV
############################################################
def CSVsave_blockingamplitudes_2D(result, dim1name, dim1newname, filename, now =
↪  datetime.datetime.now()):
    amplitudefield = "Blocking Threshold Amplitude (mA)"
    with open(filename + '.csv', mode='w') as csv_file:
        fieldnames = [dim1newname, amplitudefield]#, 'h_max']
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
        writer.writeheader()

        for i in sorted(result):
            if result[i] != {}:  # Empty dicts should be ignored
                result[i][dim1newname] = result[i].pop(dim1name)  # Needed to make compliant to
                ↪  fieldnames of CSV
                result[i][amplitudefield] = result[i].pop("Amplitude")  # Needed to make compliant to
                ↪  fieldnames of CSV
                writer.writerow(result[i])


def CSVsave_blockingamplitudes_3D(result, dim1name, dim1newname, dim2name, dim2newname, filename, now =
↪  datetime.datetime.now()):
    amplitudefield = "Blocking Threshold Amplitude (mA)"
    with open(filename + '.csv', mode='w') as csv_file:
        fieldnames = ["Chargebalance", dim1newname, amplitudefield, dim2newname]#, 'h_max']
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames, extrasaction='ignore', delimiter='|')
        writer.writeheader()

        for i in sorted(result):
            for j in sorted(result[i]):
                if result[i][j] != {}:  # Empty dicts should be ignored
                    result[i][j][dim1newname] = result[i][j].pop(dim1name)  # Needed to make compliant to
                    ↪  fieldnames of CSV
                    result[i][j][dim2newname] = result[i][j].pop(dim2name)  # Needed to make compliant to
                    ↪  fieldnames of CSV
                    result[i][j][amplitudefield] = result[i][j].pop("Amplitude")  # Needed to make
                    ↪  compliant to fieldnames of CSV
                    writer.writerow(result[i][j])


############################################################
# Find blockingamplitudes, sweep over frequency
############################################################
def find_blockingamplitudes_Freq(frequencylist, waveform):
    config = 'ProposedModel'
    par, recpar = selectConfig(config)

    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'],
                        ↪  par['HFSamp'], 1000)
67                  cpp = a / (np.pi * f)
68              elif waveform == "square":
69                  t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                        ↪  par['HFSamp'])
70                  cpp = a / (2*f)
71              elif waveform == "triangular":
72                  t_signal, i_signal = trianglewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
                        ↪  par['HFSamp'])
73                  cpp = a / (4 * f)
74              else:
75                  print "Incompatible waveform selected!"
76                  return None
77
78              updateMRGaxon(par)
79
80              print("--- RUN HFS-SIGNAL (%s kHz, %s mA) ---" % (f, a))
81              runMRGaxon(rec, t_signal, i_signal)
82
83              if is_blocked(rec, par):
84                  result[f]["frequency"] = f
85                  result[f]["amplitude"] = a
86                  result[f]["chargeperphase"] = cpp
87                  break
88
89      totaltime = time.time() - starttime
90      print("--- TOTAL RUNTIME: %s seconds ---" % (totaltime))
91
92      return result
93
94
95  ################################################################
96  # Find blockingamplitudes, sweep over frequency and chargebalance
97  ################################################################
98  def find_blockingamplitudes_CB(frequencylist, chargebalancelist):
99      config = 'ProposedModel'
100     par, recpar = selectConfig(config)
101
102     starttime = time.time()
103
104     amplitudelist = []
105     for i in range(200):
106         amplitudelist.append(round(0.1 + 0.01 * i, 2))
107
108     createMRGaxon(par)
109
110     result = {}
111     for b in chargebalancelist:
112         result[b] = {}
113         for f in frequencylist:
114             par['HFSfrequency'] = f
115             result[b][f] = {}
116             for a in amplitudelist:
117                 rec = recordMRGaxon(recpar)
118                 par['HFSamp'] = a
119                 cpp = a/(2*f)           # Charge per phase is amplitude times half the period
120                 t_signal, i_signal = chargebalanced_asymmetrical(par['HFSdelay'], par['tstop'],
                        ↪  par['HFSfrequency'], cpp, b)
121                 updateMRGaxon(par)
122                 print("--- RUN HFS-SIGNAL (%s chargebalance, %s kHz, %s mA) ---" % (b, f, a))
123                 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",
            ↪  encoding="ASCII").item()
32          result_proposed = np.load("justification_files/results_squarewave_proposedmethod.npy",
            ↪  encoding="ASCII").item()
33          plotjustification2D(result_classic, result_proposed)
34          plt.show()
35      elif(select == "sine"):
36          result_classic = np.load("justification_files/results_sinewave_classicmethod.npy",
            ↪  encoding="ASCII").item()
37          result_proposed =
            ↪  np.load("justification_files/results_sinewave_proposedmethod20190226_171930.npy",
            ↪  encoding="ASCII").item()
38          plotjustification2Dcpp(result_classic, result_proposed)
39          plt.savefig("reportplots/method/performance_sine_cpp.eps", format='eps', dpi=300)
40          plt.show()
41          plotjustification2D(result_classic, result_proposed)
42          plt.savefig("reportplots/method/performance_sine.eps", format='eps', dpi=300)
43          plt.show()
44      elif(select == "squareCB"):
45          result_classic = np.load("justification_files/results_squarewave_CB_classicmethod.npy",
            ↪  encoding="ASCII").item()
46          result_proposed = np.load("justification_files/results_squarewave_CB_proposedmethod.npy",
            ↪  encoding="ASCII").item()
47          # result_proposed =
            ↪  np.load("results_monopolar/results_monopolar_assymmetricalwave_allfrequencies.npy").item()
48          plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.1)#, freqname =
            ↪  "frequency", chargename = "chargeperphase")#"subplots")
49          plt.show()
50          plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.5)#, freqname =
            ↪  "frequency", chargename = "chargeperphase")
51          plt.show()
```

```
        plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.8)#, freqname =
        ↪  "frequency", chargename = "chargeperphase")
        plt.show()
        plotjustification3Dcpp(result_classic, result_proposed, "singleCB", 0.9)#, freqname =
        ↪  "frequency", chargename = "chargeperphase")
        plt.show()
    else:
        print "Type non-existent!"


############################################################
# VISUALIZATION OF SINGLE SIMULATION 2D
############################################################
def plotsquareWave():
    return None
#     t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], 4, 0.5)
#     runandplotsignal(par, recpar, t_signal, i_signal, config)


############################################################
# DEMO SINGLE SIMULATION TIME-SPATIAL
############################################################
def plot_square_classic(f = 10, a = 0.47, tstop = 40.0, intrinsic = 0):
    config = "ClassicModel"
    par, recpar = selectConfig(config)
    par['HFSfrequency']    = f    # kHz
    par['HFSamp']          = a  # mA
    par['tstop'] = tstop
    if intrinsic == 0:
        par['intrinsicStim'] = 0
    else:
        par['intrinsicStim'] = 1
    t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
    runandplotsignal(par, recpar, t_signal, i_signal, config)


def plot_sine_classic(f = 10, a = 0.47, tstop = 40.0, intrinsic = 0):
    config = "ClassicModel"
    par, recpar = selectConfig(config)
    par['HFSfrequency']    = f    # kHz
    par['HFSamp']          = a  # mA
    par['tstop'] = tstop
    if intrinsic == 0:
        par['intrinsicStim'] = 0
    else:
        par['intrinsicStim'] = 1
    HFSsamplesize   = 1000  # Number of samples per period
    t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'],
    ↪  HFSsamplesize)
    runandplotsignal(par, recpar, t_signal, i_signal, config)


def plot_assymmetric_classic(f = 10, a = 0.47, tstop = 40.0, chargebalance=0.5, intrinsic = 0):
    config = "ClassicModel"
    par, recpar = selectConfig(config)
    par['HFSfrequency'] = f  # kHz
    par['HFSamp'] = a  # mA
    par['tstop'] = tstop
    if intrinsic == 0:
        par['intrinsicStim'] = 0
    else:
        par['intrinsicStim'] = 1
    chargeperphase = par['HFSamp'] / (2 * par['HFSfrequency'])
    t_signal, i_signal = chargebalanced_asymmetrical(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
    ↪  chargeperphase, chargebalance)
    runandplotsignal(par, recpar, t_signal, i_signal, config)


def plot_square_proposed(f, a):
    config = "ProposedModel"
    par, recpar = selectConfig(config)
    par['HFSfrequency']    = f    # kHz
    par['HFSamp']          = a  # mA
    par['tstop'] = 20.0
    t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
    runandplotsignal(par, recpar, t_signal, i_signal, config)


```

```
126 def plot_sine_proposed(f,a):
127     config = "ProposedModel"
128     par, recpar = 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'],
        ↪  HFSsamplesize)
134     runandplotsignal(par, recpar, t_signal, i_signal, config)
135
136
137 def plot_assymmetric_proposed(f, a, chargebalance):
138     config = "ProposedModel"
139     par, recpar = 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_asymmetrical(par['HFSdelay'], par['tstop'], par['HFSfrequency'],
        ↪  chargeperphase, chargebalance)
145     runandplotsignal(par, recpar, t_signal, i_signal, config)
146
147
148 def plot_square_proposed_bipolar(f, a):
149     config = "ProposedModel_Bipolar"
150     par, recpar = 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, recpar, t_signal, i_signal, config)
156
157
158 ############################################################
159 # BACKGROUND CHAPTER
160 ############################################################
161 def plotAP():
162     config = "singleAPdemo"
163     par, recpar = 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(recpar)
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(recpar)
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, recpar = selectConfig(config)
191     t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
192     createMRGaxon(par)
193     rec = recordMRGaxon(recpar)
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()
```

```
    plotMRGgatessinglenode(plt, rec, 10)
    plt.savefig("reportplots/background/HFSexample_onsetresponse.eps", format='eps', dpi=1200)
    plt.show()


def plothminftau():
    Vm = np.arange(-150,50,1)
    Vm = np.delete(Vm, np.where(Vm == -114))
    alpha_h = [(0.34 * (-(v + 114))) / (1.0 - np.exp((v + 114) / 11.0)) for v in Vm]
    beta_h = [12.6 / (1.0 + np.exp(-(v + 31.8) / 13.4)) for v in Vm]
    print alpha_h
    h_inf = [a / (a + b) for a,b in zip(alpha_h, beta_h)]
    tau_h = [1.0 / (a + b) for a,b in zip(alpha_h, beta_h)]

    alpha_m = [(6.57 * (v + 20.4)) / (1 - np.exp(-(v + 20.4) / 10.3)) for v in Vm]
    beta_m = [(0.304 * (-(v + 25.7))) / (1 - np.exp((v + 25.7) / 9.16)) for v in Vm]
    m_inf = [a / (a + b) for a,b in zip(alpha_m, beta_m)]
    tau_m = [1.0 / (a + b) for a,b in zip(alpha_m, beta_m)]

    plt.rcParams.update({'font.size': 14})
    plt.figure(figsize=[15,5])
    plt.plot(Vm, h_inf, label='$h_\infty$')
    plt.plot(Vm, m_inf, label='$m_\infty$')
    plt.ylabel('State value')
    plt.xlabel('Membrane voltage (mV)')
    plt.legend()
    plt.xlim(-150,50)
    plt.xticks(np.arange(-150,60,20))
    plt.grid()
    plt.tight_layout()
    plt.savefig("reportplots/background/vm_gatevaluedependence_infty.eps", format='eps', dpi=1200)
    plt.show()

    plt.figure(figsize=[15, 5])
    plt.plot(Vm, tau_h, label='$\\tau_h$')
    plt.plot(Vm, tau_m, label='$\\tau_m$')
    plt.ylabel('Time constant (ms)')
    plt.xlabel('Membrane voltage (mV)')
    plt.legend()
    plt.xlim(-150, 50)
    plt.yticks(np.arange(0,1.2,0.2))
    plt.xticks(np.arange(-150, 60, 20))
    plt.grid()
    plt.tight_layout()
    plt.savefig("reportplots/background/vm_gatevaluedependence_tau.eps", format='eps', dpi=1200)
    plt.show()


############################################################
# CHAPTER DETERMINING THE EFFECTIVENESS OF A BLOCK
############################################################
def plot_2D_blockdescription():
    config = "ClassicModel"
    par, recpar = selectConfig(config)
    f = 10
    a = 0.6
    HFSsamplesize = 1000
    par['HFSfrequency'] = f  # kHz
    par['HFSamp'] = a  # mA
    par['tstop'] = 20.0
    par['intrinsicDel'] = 15.0
    par['intrinsicDur'] = 0.1
    par['intrinsicType'] = 0
    t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'],
    ↪  HFSsamplesize)
    createMRGaxon(par)
    rec = recordMRGaxon(recpar)
    updateMRGaxon(par)
    runMRGaxon(rec, t_signal, i_signal)
    # plotMRGaxon(rec)
    dt = par['dt']
    t = [15.0, 15.2, 15.4, 15.6, 15.8, 16.0]
    for i in t:
        figure(num=None, figsize=(15, 3), dpi=1200, facecolor='w', edgecolor='k')
        plotnodeVoltagessingletime(rec, int(i/dt), plt)
        plt.tight_layout()
        plt.savefig("reportplots/method/KHFACdemo_" + str(i) + ".eps", format='eps', dpi=1200)
    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'],
                ↪  par['HFSamp'], HFSsamplesize)
298             rec = recordMRGaxon(recpar)
299             updateMRGaxon(par)
300             runMRGaxon(rec, t_signal, i_signal)
301             fig = plotMRGaxon(rec)
302             plt.tight_layout()
303             plt.savefig("reportplots/method/KHFACdemo3D_pulse_" + str(i) + "_amp_" + str(j) + ".eps",
                ↪  format='eps', dpi=1200)
304     return plt
305
306
307 def plot_gates_proposedmethodsetup():
308     config = "ProposedModel"
309     par, recpar = selectConfig(config)
310     f = 10
311     a = 0.6
312     HFSsamplesize = 1000
313     par['HFSfrequency'] = f  # kHz
314     par['HFSamp'] = a  # mA
315     t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'],
        ↪  HFSsamplesize)
316     createMRGaxon(par)
317     rec = recordMRGaxon(recpar)
318     updateMRGaxon(par)
319     runMRGaxon(rec, t_signal, i_signal)
320     # plotMRGaxon(rec)
321     plotMRGgatessinglenode(plt, rec, par['HFSreferenceNode'] + 1, s=False)
322     plt.savefig("reportplots/method/proposedsetupsimulation_virtualanode.eps", format='eps', dpi=1200)
323     plt.show()
324     return plt
325
326
327 def plot_asymmetrical_waveform():
328     t_start = 0.0
329     f = 10
330     a = 1.0
331     t_stop = 1.0/f
332     chargebalance = [0.2, 0.5, 0.8]
333     cpp = a/(2*f)
334     for i in chargebalance:
335         t_signal = [0]
336         i_signal = [0]
337         t_result, i_result = chargebalanced_asymmetrical(t_start, t_stop, f, cpp, i)
338         t_signal.extend(t_result.tolist())
339         t_signal.append(max(t_result))
340         i_signal.extend(i_result.tolist())
341         i_signal.append(0)
342         plt.plot(t_signal, i_signal, label=i, linewidth=3)
343         # plot_signal(plt, t_signal, i_signal, label = i)
344     # plt.axhline(color='k', lw=1.0)
345     plt.xlabel('Time (ms)')
346     plt.ylabel('Current (mA)')
347     plt.ylim(-3,3)
348     plt.yticks(np.arange(-3.0,3.5,0.5))
349     plt.grid()
350     plt.legend(title="Anode fraction")
351     plt.tight_layout()
352     plt.savefig("reportplots/method/asymmetrical_waveform.eps", format='eps', dpi=300)
353     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'],
        ↪  cpp, 0.7)
420     # plot_signal(plt,t_signal, i_signal)
421     # plt.show()
422     createMRGaxon(par)
423     rec = recordMRGaxon(recpar)
424     updateMRGaxon(par)
425     runMRGaxon(rec, t_signal, i_signal)
426     fig = plotMRGaxon(rec)
427     plt.show()
428
429
430 def plot_ipd_validation(amp, ipd1,ipd2):
```

```
    par, recpar = selectConfig("ClassicModel")
    par['tstop'] = 20.0
    t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], amp, ipd1, ipd2)
    # plot_signal(plt,t_signal, i_signal)
    # plt.show()
    createMRGaxon(par)
    rec = recordMRGaxon(recpar)
    updateMRGaxon(par)
    runMRGaxon(rec, t_signal, i_signal)
    fig = plotMRGaxon(rec)
    plt.tight_layout()
    plt.show()

##########################################################
# RUN AND PLOT A SINGLE SIMULATION
##########################################################
def runandplotsignal(par, recpar, t_signal, i_signal, config):
    starttime = time.time()

    createMRGaxon(par)
    intervaltime = time.time() - starttime
    print("--- CREATEAXON: %s seconds ---" % (intervaltime))
    intervaltime = time.time()

    rec = recordMRGaxon(recpar)
    updateMRGaxon(par)
    intervaltime = time.time() - intervaltime
    print("--- PREPAREAXON: %s seconds ---" % (intervaltime))
    intervaltime = time.time()


    runMRGaxon(rec, t_signal, i_signal)
    intervaltime = time.time() - intervaltime
    print("--- RUNAXON: %s seconds ---" % (intervaltime))
    intervaltime = time.time()

    fig = plotMRGaxon(rec)

    # plotMRGgatessingletime(plt, rec, int(par['tstop']/par['dt']))

    # Plot single AP
    if config=='singleAPdemo':
        plotMRGgatessinglenode(plt, rec, 4)


    # # Plot single KHFAC signal
    if config=='HFSdemo':
        plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']))
        plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']) + 1)
        # Onset response
        plotMRGgatessinglenode(plt, rec, int(par['axonnodes']-1))

    if config == 'ClassicModel':
        plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']))
        plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']) + 1)

    if config == 'ProposedModel' or config == 'ProposedModel_Bipolar':
        plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']))
        plotMRGgatessinglenode(plt, rec, int(par['HFSreferenceNode']) + 1)
        plotMRGgatessinglenode_timeframe(plt, rec, int(par['HFSreferenceNode']), 19/par['dt'])
        plotMRGgatessinglenode_timeframe(plt, rec, int(par['HFSreferenceNode']) + 1, 19/par['dt'])


    intervaltime = time.time() - intervaltime
    print("--- PLOT: %s seconds ---" % (intervaltime))
    plt.show()


##########################################################
# Parameters for configuration of the simulation
# ########################################################
# # config = 'singleAPdemo'
# # config = 'HFSdemo'
# # config = 'HFSModel'
# config = 'ClassicModel'
# par, recpar = selectConfig(config)
# verbose = False
#
```

```
# t_signal, i_signal = sinewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'], 1000)
# testandplotsignal(par, recpar, verbose, t_signal, i_signal, config)
#
# h.quit()


##############################################################
# Presentation plots
##############################################################
def buildKHFACgif():
    config = "ClassicModel"
    par, recpar = selectConfig(config)
    par['HFSfrequency'] = 10
    par ['HFSamp'] = 0.5
    par['tstop'] = 40
    par['intrinsicDel'] = 15.0
    t_signal, i_signal = squarewave(par['HFSdelay'], par['tstop'], par['HFSfrequency'], par['HFSamp'])
    createMRGaxon(par)
    rec = recordMRGaxon(recpar)
    updateMRGaxon(par)
    runMRGaxon(rec, t_signal, i_signal)

    dt = par['dt']
    tstart = int((par['HFSdelay'] + par['tonset']) / dt)

    for i in range(0, par['tstop'] * 10):
        t = i/10.0
        plt.figure()
        plotnodeVoltagessingletime(rec, int(t/dt), plt)
        filename = str(int(t*10))
        plt.savefig('figures/' + filename + '.png', bbox_inches='tight')
        plt.close()
```

plottingfunctions_results.py

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import numpy as np
# plt.style.use('seaborn-dark-palette')

##########################################
# MONOPOLAR SIMULATIONS
##########################################
def plot_results_monopolar_ipd(data, type, plottype, validation=False):
    datascalar = 1
    if type == "amplitude":
        datatype = 'amplitude'
        datalabel = "Block threshold (mA)"
        ylim = 1.0
    elif type == "cpp":
        datatype = 'chargeperphase'
        datascalar = 1000
        datalabel = "Block threshold charge per phase ($\mu$C)"
        ylim = 0.02

    if plottype == "lines":
        count = 0
        colmulti = 1
        if validation:
            colmulti = 2
        for i in sorted(data.keys()):
            if 20*i %2 == 0 and i != 0.9:
                x = sorted(data[i].keys())
                x = [n / 10 for n in x]
                y = []
                for j in sorted(data[i].keys()):
                    if data[i][j] != {}:
                        y = np.append(y, data[i][j][datatype]*datascalar)
                    else:
                        y = np.append(y, np.nan)
                line, = plt.plot(x,y,label=str(i/10) + "ms", color="C"+str(count*colmulti))
                count+=1
        plt.xlabel("Cathodal interphase delay (ms)")
        plt.ylabel(datalabel)
        plt.ylim(bottom=0, top=ylim)
```

```
        plt.xlim(0,0.09)
        if type == "amplitude":
            plt.yticks(np.arange(0, 1.1, step=0.1))
        plt.xticks(np.arange(0, 0.09, step=0.01))
        plt.grid(True)
        plt.legend(title = "Anodal interphase delay")

    elif plottype == "colour" or plottype == "3D":
        ipd1 = []
        ipd2 = []
        for i in sorted(data.keys()):
            ipd1 = np.append(ipd1, i)
            for j in sorted(data[i].keys()):
                if j not in ipd2:
                    ipd2=np.append(ipd2, j)
        matrix = np.zeros((len(ipd1), len(ipd2)))
        for i in range(len(ipd1)):
            for j in range(len(ipd2)):
                if data[ipd1[i]][ipd2[j]] != {}:
                    matrix[i][j] = data[ipd1[i]][ipd2[j]][datatype]*datascalar
                else:
                    matrix[i, j] = np.nan

        # Set up a regular grid of interpolation points
        ipd1 = [x / 10 for x in ipd1]
        ipd2 = [x / 10 for x in ipd2]
        X, Y = np.meshgrid(ipd1, ipd2)

        if plottype == "colour":
            plt.pcolormesh(X, Y, matrix.T, cmap=cm.viridis_r)
            clb = plt.colorbar()
            clb.set_label(datalabel)
            # clb.ax.set_title(datalabel)
            plt.xlabel("Anodal interphase delay (ms)")
            plt.ylabel("Cathodal interphase delay (ms)")
        elif plottype == "3D":
            ax = plt.gca(projection='3d')
            ax.plot_surface(X, Y, matrix.T, cmap=cm.viridis_r, linewidth=0, antialiased=False,
            ↪  vmin=np.nanmin(matrix), vmax=np.nanmax(matrix))
            ax.set_xlabel("Anodal interphase delay (ms)")
            ax.set_ylabel("Cathodal interphase delay(ms)")
            ax.set_zlabel(datalabel)
            ax.view_init(15, -160)
    title = ""
    for i in data:
        for j in data[i]:
            if data[i][j] != {}:
                title = "Frequency = " + str(data[i][j]["frequency"]) + "kHz"
                break
        if title != "":
            break
    plt.title(title)
    plt.tight_layout()
    return plt


def plot_results_monopolar_basicwaveforms(squareres, sineres, triangularres, type):
    frequencies = []
    amps_square = []
    amps_sine = []
    amps_triangle = []
    resvar = ""
    xlabelvar = ""
    if type == "amplitude":
        resvar = "Amplitude"
        ylabelvar = "Block threshold (mA)"
        ylim = 1.0
    elif type == "cpp":
        resvar = "Chargeperphas"
        ylabelvar = "Block threshold charge per phase ($\mu$C)"
        ylim = 0.05
    else:
        print "Incorrect type"
        return
    for i in sorted(squareres):
        frequencies.append(i)
        amps_square.append(squareres[i][resvar])
    for i in sorted(sineres):
```

```
        amps_sine.append(sineres[i][resvar])
    for i in sorted(triangularres):
        amps_triangle.append(triangularres[i][resvar])
    plt.plot(frequencies,amps_square, label="Square")
    plt.plot(frequencies,amps_sine, label="Sine")
    plt.plot(frequencies,amps_triangle, label="Triangular")
    plt.xlabel("KHFAC frequency (kHz)")
    plt.ylabel(ylabelvar)
    plt.legend(title="Waveform type")
    plt.ylim(0, ylim)
    plt.yticks(np.arange(0,ylim+ylim/10,ylim/10))
    plt.xlim(left=0, right=max(frequencies)+5)
    plt.xticks(np.arange(0,max(frequencies)+5,5))
    plt.grid()
    plt.tight_layout()
    return plt


def plt_results_monopolar_stepwaveforms(stepsineres, steptriangleres, type):
    if type == "amplitude":
        resvar = "amplitude"
        ylabelvar = "Block threshold (mA)"
        ylim = 1.0
    elif type == "cpp":
        resvar = "chargeperphase"
        ylabelvar = "Block threshold charge per phase ($\mu$C)"
        ylim = 0.05
    steps = []
    amps_sine = []
    amps_triangle = []
    for i in sorted(stepsineres):
        steps.append(i)
        amps_sine.append(stepsineres[i][resvar])
    for i in sorted(steptriangleres):
        amps_triangle.append(steptriangleres[i][resvar])
    steps = [x * 4 for x in steps]
    plt.plot(steps, amps_sine, label="Stepped sine", color="orange")
    plt.plot(steps, amps_triangle, label="Stepped triangular", color="green")
    plt.xlabel("Steps per period")
    plt.ylabel(ylabelvar)
    plt.legend(title="Waveform type")
    plt.ylim(0, ylim)
    plt.yticks(np.arange(0,ylim+ylim/10,ylim/10))
    plt.xlim(left=0, right=max(steps)+20)
    plt.xticks(np.arange(0,max(steps)+5,20))
    plt.grid()
    plt.tight_layout()
    return plt


def plt_results_monopolar_realdistance(data, type):
    if type == "amplitude":
        resvar = "amplitude"
        ylabelvar = "Block threshold (mA)"
        ylim = 30.0
    elif type == "cpp":
        resvar = "chargeperphase"
        ylabelvar = "Block threshold charge per phase ($\mu$C)"
        ylim = 1.5
    distances = []
    amps = []
    for i in sorted(data):
        distances.append(i/1000.0)
        amps.append(data[i][resvar])
    plt.plot(distances, amps)#, label="Stepped sine")
    plt.xlabel("Electrode perpendicular distance (mm)")
    plt.ylabel(ylabelvar)
    plt.title("KHFAC frequency = 10kHz")
    plt.grid(True)
    # plt.legend()
    plt.ylim(bottom = 0, top = ylim)
    plt.xlim(left = 0, right = 6)
    plt.tight_layout()
    return plt


def plot_results_asymmetrical(data, type, plottype, plotfreq):
    datascalar = 1
```

```
    if type == "amplitude":
        datatype = "amplitude"
        datalabel = "Peak current amplitude at block threshold (mA)"
        # ylim = 30.0
    elif type == "cpp":
        datatype = "chargeperphase"
        datascalar = 1000
        datalabel = "Block threshold charge per phase ($\mu$C)"
        # ylim = 30.0

    if plottype == "colour" or plottype == "3D":
        cb = []
        f = []
        for i in sorted(data.keys()):
            cb.append(i)
            for j in sorted(data[i].keys()):
                if j not in f:
                    f.append(j)
        matrix = np.zeros((len(cb), len(f)))
        for i in range(len(cb)):
            for j in range(len(f)):
                if data[cb[i]][f[j]] != {}:
                    if datatype == "amplitude":
                        datascalar = 0.5 / np.min([cb[i],1.0-cb[i]])
                    matrix[i][j] = data[cb[i]][f[j]][datatype] * datascalar
                else:
                    matrix[i, j] = np.nan
        X, Y = np.meshgrid(cb, f)
        if plottype == "colour":
            plt.pcolormesh(X, Y, matrix.T, cmap=cm.viridis_r)
            clb = plt.colorbar()
            clb.set_label(datalabel)
            plt.xlabel("Anode fraction")
            plt.ylabel("Frequency (kHz)")
        elif plottype == "3D":
            ax = plt.gca(projection='3d')
            ax.plot_surface(X, Y, matrix.T, cmap=cm.viridis_r, linewidth=0, antialiased=False,
            ↪  vmin=np.nanmin(matrix), vmax=np.nanmax(matrix))
            ax.set_xlabel("Anode fraction")
            ax.set_ylabel("Frequency (kHz)")
            ax.set_zlabel(datalabel)
            ax.view_init(15, -160)
    elif plottype == "singlefrequency":
        cb = []
        maxamp = []
        cpp = []
        for i in sorted(data.keys()):
            cb.append(i)
            if data[i][plotfreq] != {}:
                maxamp.append(data[i][plotfreq]['amplitude']*0.5/(np.min([i, 1.0-i])))
                cpp.append(data[i][plotfreq]['chargeperphase'])
        fig, ax1 = plt.subplots(figsize=(8,5))
        cb = [n/plotfreq for n in cb]
        color = 'C0'
        ax1.set_xlabel("$T_{anode}$ (ms)")
        ax1.set_ylabel("Peak current amplitude at block threshold (mA)", color=color)
        ax1.tick_params(axis='y', labelcolor=color)
        ax1.plot(cb, maxamp, color=color)
        ax1.set_ylim(bottom=0, top=4.0)
        ax1.set_xlim(left=0, right=0.1)
        ax1.set_xticks(np.arange(0.01, 0.1, step=0.01))

        ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis
        color = 'r'
        ax2.set_ylabel("Block threshold charge per phase ($\mu$C)", color=color)  # we already handled
        ↪  the x-label with ax1
        ax2.plot(cb, cpp, color=color)
        ax2.tick_params(axis='y', labelcolor=color)
        ax2.set_ylim(bottom=0, top=0.04)
        fig.tight_layout()
        plt.grid(axis="x")
        plt.title("KHFAC frequency = " + str(plotfreq) + "kHz")
        ax1.grid()
    plt.tight_layout()
    return plt

##########################################
# BIPOLAR SIMULATIONS
```

```
########################################
def plot_results_bipolar_ETA_IEC_parallel(data, type, plottype):
    # fig = plt.figure()
    datascalar = 1
    if type == "amplitude":
        datatype = 'amplitude'
        datalabel = "Block threshold (mA)"
    elif type == "cpp":
        datatype = 'chargeperphase'
        datascalar = 1000
        datalabel = "Block threshold charge per phase ($\mu$C)"

    if plottype == "lines":
        for i in sorted(data.keys(), reverse=True):
            x = sorted(data[i].keys())
            x = [n / 1000.0 for n in x] # convert to mm
            y = []
            for j in sorted(data[i].keys()):
                if data[i][j] != {}:
                    y = np.append(y, data[i][j][datatype]*datascalar)
                else:
                    y = np.append(y, np.nan)
            plt.plot(x,y,label=str(i) + "$\mu$m")
        plt.xlabel("Interpolar distance (mm)")
        plt.ylabel(datalabel)
        plt.legend(title = "Electrode to axon distance", loc="upper right")
        plt.grid()

    elif plottype == "colour" or plottype == "3D":
        ax1 = []
        ax2 = []
        for i in sorted(data.keys()):
            ax1.append(i)
            for j in sorted(data[i].keys()):
                if j not in ax2:
                    ax2.append(j)
        matrix = np.zeros((len(ax1), len(ax2)))
        for i in range(len(ax1)):
            for j in range(len(ax2)):
                if data[ax1[i]][ax2[j]] != {}:
                    matrix[i,j] = data[ax1[i]][ax2[j]][datatype] * datascalar
                else:
                    matrix[i,j] = np.nan
        # ax1 = [x / 1000 for x in ax1]
        ax2 = [x / 1000.0 for x in ax2]
        # Set up a regular grid of interpolation points
        X, Y = np.meshgrid(ax1, ax2)

        if plottype == "colour":
            plt.pcolormesh(Y, X, matrix.T, cmap=cm.viridis_r)
            clb = plt.colorbar()
            clb.set_label(datalabel)
            plt.xlabel("Interpolar distance (mm)")
            plt.ylabel("Electrode to axon distance ($\mu$m)")
        elif plottype == "3D":
            ax = plt.gca(projection='3d')
            ax.plot_surface(X, Y, matrix.T, cmap=cm.viridis_r, linewidth=0, antialiased=False,
            ↪  vmin=np.nanmin(matrix),
                            vmax=np.nanmax(matrix))
            ax.set_xlabel("Electrode to axon distance ($\mu$m)")
            ax.set_ylabel("Interpolar distance (mm)")
            ax.set_zlabel(datalabel)
            ax.view_init(15, -160)


    elif plottype == "optimal":
        x = sorted(data.keys()) # x-axis is electrode to axon distance
        bestamplitudes = []
        bestcontactdistances = []
        for i in x:
            amplitudes = []
            contactdistances = []
            for j in data[i]:
                if data[i][j] != {}:
                    amplitudes.append(data[i][j][datatype]*datascalar)
                    contactdistances.append(j)
            if amplitudes != []:
                bestamplitudes.append(np.min(amplitudes))
```

```
                bestcontactdistances.append(contactdistances[amplitudes.index(np.min(amplitudes))])
            else:
                bestamplitudes.append(np.nan)
                bestcontactdistances.append(np.nan)

        bestcontactdistances = [n / 1000.0 for n in bestcontactdistances]  # convert to mm
        fig, ax1 = plt.subplots()
        color = 'C0'
        ax1.set_xlabel("Electrode to axon distance ($\mu$m)")
        ax1.set_ylabel("Lowest block threshold (mA)", color=color)
        ax1.plot(x, bestamplitudes, color=color)
        ax1.set_ylim(bottom=0, top=12)
        ax1.tick_params(axis='y', labelcolor=color)
        ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis
        color = 'r'
        ax2.set_ylabel('Optimal interpolar distance (mm)', color=color)  # we already handled the x-label
        ↪  with ax1
        ax2.plot(x, bestcontactdistances, color=color)
        ax2.set_ylim(bottom=0, top=6)
        ax2.tick_params(axis='y', labelcolor=color)
        ax1.set_yticks(np.arange(0, 12, step = 1))
        ax1.grid()
        ax2.set_yticks(np.arange(0, 6, step = 1))
        ax2.grid()

        fig.tight_layout()

    plt.title("KHFAC frequency = 10kHz")
    plt.tight_layout()
    return plt

def plot_results_bipolar_IEC_parallelperp(datapar, dataperp, type):
    datascalar = 1
    if type == "amplitude":
        datatype = 'amplitude'
        datalabel = "Block threshold (mA)"
        ylim = 0.8
    elif type == "cpp":
        datatype = 'chargeperphase'
        datascalar = 1000
        datalabel = "Block threshold charge per phase ($\mu$C)"
        ylim = 0.04
    x = sorted(datapar.keys())
    x = x[:x.index(25000)]

    ypar = []
    yperp = []
    for i in x:
        if datapar[i] != {}:
            ypar.append(datapar[i][datatype]*datascalar)
        else: ypar.append(np.nan)
        if dataperp[i] != {}:
            yperp.append(dataperp[i][datatype]*datascalar)
        else: ypar.append(np.nan)
    x = [n / 1000.0 for n in x]
    plt.plot(x, ypar, label = "parallel")
    plt.plot(x, yperp, label = "perpendicular")
    plt.xlabel("Interpolar distance (mm)")
    plt.ylabel(datalabel)
    plt.ylim(bottom=0, top=ylim)
    plt.xlim(0,25)
    plt.legend(title="Electrode orientation")
    plt.title("Frequency 10kHz, electrode 1000$\mu m$ above axon")
    plt.tight_layout()
    plt.grid()
    return plt


def plot_results_bipolar_orientation(type, result_x, result_z, plottype="lines", IECdist=2200):
    datascalar = 1
    if type == "amplitude":
        datatype = 'amplitude'
        datalabel = "Block threshold (mA)"
    elif type == "cpp":
        datatype = 'chargeperphase'
        datascalar = 1000
        datalabel = "Block threshold charge per phase ($\mu$C)"
    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"] >
                            ↪  amplitudes[classicres[i][j][k]["Frequency"]]:
16                              amplitudes[classicres[i][j][k]["Frequency"]] =
                                ↪  classicres[i][j][k]["Amplitude"]
17                      else:
18                          amplitudes[classicres[i][j][k]["Frequency"]] = classicres[i][j][k]["Amplitude"]
19
20      lists = sorted(amplitudes.items())
21      fc, ac = zip(*lists)
22      plt.plot(fc, ac, color= 'darkgrey', linestyle='-', label='Classic Method')
23
24      amplitudes_prop = {}
```

```
    for i in proposedres:
        if proposedres[i] != {}:
            if proposedres[i]["Frequency"] in amplitudes_prop:
                if proposedres[i]["Amplitude"] > amplitudes_prop[proposedres[i]["Frequency"]]:
                    amplitudes_prop[proposedres[i]["Frequency"]] = proposedres[i]["Amplitude"]
            else:
                amplitudes_prop[proposedres[i]["Frequency"]] = proposedres[i]["Amplitude"]

    lists = sorted(amplitudes_prop.items())
    fp, ap = zip(*lists)
    plt.plot(fp, ap, 'b-', label='Proposed Method')

    plt.xlabel("Frequency (kHz)")
    plt.xticks(np.arange(0,45,5))
    plt.xlim(left=0,right=45)
    plt.ylabel("Minimal blocking amplitude (mA)")
    plt.ylim([0, 1])
    plt.grid()
    plt.legend()
    plt.tight_layout()

    return plt


def plotjustification2Dcpp(classicres, proposedres):
    amplitudes  = {}
    for i in classicres:
        for j in classicres[i]:
            for k in classicres[i][j]:
                if classicres[i][j][k] != {}:
                    cpp = classicres[i][j][k]["Amplitude"] / (np.pi * classicres[i][j][k]["Frequency"])
                    if classicres[i][j][k]["Frequency"] in amplitudes:
                        if cpp > amplitudes[classicres[i][j][k]["Frequency"]]:
                            amplitudes[classicres[i][j][k]["Frequency"]] = cpp
                    else:
                        amplitudes[classicres[i][j][k]["Frequency"]] = cpp

    lists = sorted(amplitudes.items())
    fc, ac = zip(*lists)
    plt.plot(fc, ac, color= 'darkgrey', linestyle='-', label='Classic Method')

    amplitudes_prop = {}
    for i in proposedres:
        if proposedres[i] != {}:
            cpp = proposedres[i]["Amplitude"] / (np.pi * proposedres[i]["Frequency"])
            if proposedres[i]["Frequency"] in amplitudes_prop:
                if cpp > amplitudes_prop[proposedres[i]["Frequency"]]:
                    amplitudes_prop[proposedres[i]["Frequency"]] = cpp
            else:
                amplitudes_prop[proposedres[i]["Frequency"]] = cpp

    lists = sorted(amplitudes_prop.items())
    fp, ap = zip(*lists)
    plt.plot(fp, ap, 'b-', label='Proposed Method')

    plt.xlabel("Frequency (kHz)")
    plt.xticks(np.arange(0,45,5))
    plt.xlim(left=0,right=45)

    plt.ylabel("Threshold charge per phase ($\mu C$)")
    plt.ylim([0, 0.05])

    plt.grid()
    plt.legend()
    plt.tight_layout()
    return plt


# Plots '3 dimensional' results (e.g. chargebalance)
def plotjustification3D(classicres, proposedres, type):     # If type is subplots, build all results in
↪   smaller subplots
    amplitudes  = {}

    for i in classicres:
        for j in classicres[i]:
            for k in classicres[i][j]:
                for l in classicres[i][j][k]:
                    if classicres[i][j][k][l] != {}:
```

```
                        if k not in amplitudes:
                            amplitudes[k] = {}

                        if classicres[i][j][k][l]["Frequency"] in amplitudes[k]:
                            if classicres[i][j][k][l]["Amplitude"] >
                            ↪  amplitudes[k][classicres[i][j][k][l]["Frequency"]]:
                                amplitudes[k][j][classicres[i][j][k][l]["Frequency"]] =
                                ↪  classicres[i][j][k][l]["Amplitude"]
                        else:
                            amplitudes[k][classicres[i][j][k][l]["Frequency"]] =
                            ↪  classicres[i][j][k][l]["Amplitude"]


    amplitudes_prop = {}
    for i in proposedres:
        if i not in amplitudes_prop:
            amplitudes_prop[i] = {}
        for j in proposedres[i]:
            if proposedres[i][j] != {}:
                if proposedres[i][j]["Frequency"] in amplitudes_prop[i]:
                    if proposedres[i][j]["Amplitude"] >
                    ↪  amplitudes_prop[i][proposedres[i][j]["Frequency"]]:
                        amplitudes_prop[i][proposedres[i][j]["Frequency"]] =
                        ↪  proposedres[i][j]["Amplitude"]
                else:
                    # frequencies.append(proposedres[i][j]["Frequency"])
                    amplitudes_prop[i][proposedres[i][j]["Frequency"]] = proposedres[i][j]["Amplitude"]

    if type == "largeplot":
        lists = {}
        for i in sorted(amplitudes):
            lists[i] = sorted(amplitudes[i].items())
            fc, ac = zip(*lists[i])
            plt.plot(fc, ac, 'k-', label='Classic Method CB=' + str(i))

        lists = {}
        for i in sorted(amplitudes_prop):
            lists[i] = sorted(amplitudes_prop[i].items())
            fc, ac = zip(*lists[i])
            plt.plot(fc, ac, 'b-', label='Proposed Method CB=' + str(i))


    elif type == "subplots":
        figheight = 3
        figwidth = 4
        fig, axarr = plt.subplots(figheight, figwidth)

        lists = {}
        lists_prop = {}
        plotnum = 0
        for i in sorted(amplitudes):
            lists[i] = sorted(amplitudes[i].items())
            fc, ac = zip(*lists[i])
            subplt = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
            subplt.plot(fc, ac, 'k-', label='Classic Method CB=' + str(i))
            subplt.set_title('Chargebalance = ' + str(i))

            lists_prop[i] = sorted(amplitudes_prop[i].items())
            fc_prop, ac_prop = zip(*lists_prop[i])
            subplt.plot(fc_prop, ac_prop, 'b-', label='Proposed Method CB=' + str(i))

            plotnum += 1
            subplt.set_ylim([0, 1])
            subplt.legend()

        for i in range(plotnum, figheight*figwidth):
            subplt = axarr[int(plotnum // figwidth), int(plotnum % figwidth)]
            fig.delaxes(subplt)
            plotnum +=1

    else :
        print "No type selected! Choose 'largeplot' or 'subplots'"
        return None

    return plt


# Plots '3 dimensional' results  Chargeperphase(e.g. chargebalance)
```

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

```
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')#,
            ↪ bbox_to_anchor=(0.6, 0.3))
266         # plt.tight_layout()
267 
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
    ↪ results in smaller subplots
280     amplitudes  = {}
281 
282     for i in classicres:
283         for j in classicres[i]:
284             for k in classicres[i][j]:
285                 for l in classicres[i][j][k]:
286                     if classicres[i][j][k][l] != {}:
287                         if j not in amplitudes:
288                             amplitudes[j] = {}
289 
290                         if classicres[i][j][k][l]["Chargeperphase"] in amplitudes[j]:
291                             if classicres[i][j][k][l]["Chargeperphase"] >
                                ↪ amplitudes[j][classicres[i][j][k][l]["Chargebalance"]]:
292                                 amplitudes[j][k][classicres[i][j][k][l]["Chargebalance"]] =
                                    ↪ classicres[i][j][k][l]["Chargeperphase"]
293                         else:
294                             amplitudes[j][classicres[i][j][k][l]["Chargebalance"]] =
                                ↪ classicres[i][j][k][l]["Chargeperphase"]
295 
296 
297     amplitudes_prop = {}
298     for i in proposedres:
299         for j in proposedres[i]:
300             if j not in amplitudes_prop:
301                 amplitudes_prop[j] = {}
302             if proposedres[i][j] != {}:
303                 if proposedres[i][j]["Frequency"] in amplitudes_prop[j]:
304                     if proposedres[i][j]["Chargeperphase"] >
                        ↪ amplitudes_prop[j][proposedres[i][j]["Chargebalance"]]:
305                         amplitudes_prop[j][proposedres[i][j]["Chargebalance"]] =
                            ↪ proposedres[i][j]["Chargeperphase"]
306                 else:
307                     # frequencies.append(proposedres[i][j]["Frequency"])
308                     amplitudes_prop[j][proposedres[i][j]["Chargebalance"]] =
                        ↪ proposedres[i][j]["Chargeperphase"]
309 
310     if type == "largeplot":
311         lists = {}
312         for i in range(4,6,2):# sorted(amplitudes):
313             lists[i] = sorted(amplitudes[i].items())
314             fc, ac = zip(*lists[i])
315             plt.plot(fc, ac, 'k-', label='Classic Method CB=' + str(i))
316 
317         lists = {}
318         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')#,
            ↪ bbox_to_anchor=(0.6, 0.3))
354 
355 
356     else :
357         print "No type selected! Choose 'largeplot' or 'subplots'"
358         return None
359 
360     return plt
```