```python
import os
import pandas as pd
import numpy as np
import scipy
import math
import matplotlib as mpl
import matplotlib
matplotlib.use('TkAgg')
# matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import datetime
import tables
import matplotlib.cm as cm
import matplotlib.colors as clr
from matplotlib.ticker import FormatStrFormatter
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
from matplotlib import cm
from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
from matplotlib.mlab import griddata
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk
from matplotlib.figure import Figure
import tkinter as tk
from tkinter import ttk
from tkinter.scrolledtext import ScrolledText
from tkinter import *
from tkinter.filedialog import askopenfilename
import time
import sys
from inspect import currentframe, getframeinfo
from tkinter import filedialog
import builtins

pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 2000)
import datetime
import re
import tables
import os
import shutil
import pyarrow as pa
import pyarrow.parquet as pq
from apscheduler.schedulers.background import
```
BackgroundScheduler
# from simCase83 import Simulation
import threading
from multiprocessing import dummy as multiprocessing
import queue
import threading
import Thread
import concurrent.futures
import Future
import concurrent.futures
import ThreadPoolExecutor
import concurrent.futures
import ProcessPoolExecutor
import gc
# from first_class import StdoutToWidget
import mpl_toolkits.axes_grid1
import make_axes_locatable
import tkinter
import messagebox
import random
import decimal
import Decimal
import copy
import pandas.plotting
import register_matplotlib_converters
import register_matplotlib_converters()
import matplotlib.figure
import Figure
import random

LARGE_FONT = ('Verdana', 12)
global_x = 0
global_x_label = []
global_sim_data = []
global_sim_data_listbox = []
dict_param = {}
dict_paramv2 = {}
prep_pageone = {}
current_selection = 'None'
current_tab = 'None'
chosen_rows_alt, chosen_cols_alt = (None, None)

# ----------------
# START OF IMPORT / CONVERT FUNCTION

class Simulation(threading.Thread):
    def __init__(self, save_loc, file_name):
        threading.Thread.__init__(self)
        self.home = save_loc
        self/filepath = file_name

    def convert(self):
        filepath = self/filepath
        start_time = datetime.datetime.now()
x_current_addition = ""

dx, dy, dz, lx, ly, lz, nx, ny, nz, n_name,
n_title, n_temp, n_sim, n_ver, n_file, n_file_type,
pre_line, n_steps, keys, \
check_grid01, pre_pre_line, df_grid_data,
blacklist, x_check, y_check, z_check, k_check, ln,
cells_grid, \
cells_col, pre_pre_pre_line = ([] for ti in range(31))

add_folder, name_of_file, core_path, new_path,
select_folder, print_now, keys2, comp_temp, reg_temp,
well_temp, \
n_pressure, ndaysnow, ndates, complist,
well_list_w, path_comp, path_reg, path_well, comp_list,
temp_list, templistr,
\ntemp_list_w, print_now_r, reg_list, n_reg,
current, wellreads, read_now, current_f, nwells,
n_well_name, ntypes, n_pv_well, \
n_well_temp, well_temp_itemsize, final_line,
x_current, sum_qst, path_data, cell_index = ([] for ti in range(40))

normal_length = 0

n_summary, getcomps, store_once_one,
store_data_once, firstvalues, start_main, x, check_title,
check_grid02, check_grid03, \
key_search, indexing_go, = [0] * 12

start = "01-Jan-2010 00:00:00"
i_day = datetime.datetime.strptime(start, "%d-%b
-%Y %H:%M:%S")

temp_dict, cells_dict, grid_dim_dict,
temp_storage, dict_comp, keys2_dict, summary_info,
store_once, col_width, \
current_dict, store_col, col_width_data,
cells_data = ([] for tj in range(13))

well_param = 0

for temp_items in ['DX', 'DY', 'DZ', 'XKEYS']:
temp_storage[temp_items] = []

def get_cells(*args):
    length = len(args)
    if length != 5:
        return None
    celli = args[0]
    cellj = args[1]
    cellk = args[2]
    grid_dict_data = args[3]
    option = args[4]

    imax = int(grid_dict_data['NX'])
    jmax = int(grid_dict_data['NY'])
    kmax = int(grid_dict_data['NZ'])
    cells_col_names = ['Cell', 'i', 'j', 'k', 'DX', 'DY', 'DZ', 'X', 'Y', 'Z']

    cell_list_dict = {}
    for nok in cells_col_names:
        cell_list_dict[nok] = []

    if option == "single":
        ncell = (cellj - 1) * imax + (cellk - 1)
        * jmax * kmax + celli
        return ncell
    elif option == "full":
        for nk in range(1, kmax + 1):
            for nj in range(1, jmax + 1):
                for ni in range(1, imax + 1):
                    ncell = (nj - 1) * imax + (nk - 1) * jmax * imax + ni
                    cell_list_dict['Cell'].append(int(ncell))
                    cell_list_dict['i'].append(int(ni))
                    cell_list_dict['j'].append(int(nj))
                    cell_list_dict['k'].append(int(nk))

        cell_list_dict['NX'] = imax
        cell_list_dict['NY'] = jmax
        cell_list_dict['NZ'] = kmax
        return cell_list_dict
```python
def get_cell_dim(*args):
    length = len(args)
    if length != 2:
        return None
    first_value = 0
    all_cells = args[0]
    grid_info = args[1]
    dim_dx = grid_info['DX']
    dim_dy = grid_info['DY']
    dim_dz = grid_info['DZ']
    length_x, length_y, length_z = [0] * 3
    dxi_prev, dyj_prev, dzk_prev = [0] * 3
    ivalue_prev, jvalue_prev, kvalue_prev = [0] * 3
    for kl in kvalue:
        dzk = float(dim_dz[kl-1])
        length_z = length_z + float(0.5 * (dzk_prev + dzk))
    for jl in jvalue:
        dyj = float(dim_dy[jl-1])
        length_y = length_y + float(0.5 * (dyj_prev + dyj))
    for il in ivalue:
        dxi = float(dim_dx[il-1])
        length_x = length_x + float(0.5 * (dxi_prev + dxi))
        all_cells['DX'].append(dxi)
        all_cells['DY'].append(dyj)
        all_cells['DZ'].append(dzk)
        all_cells['X'].append(length_x)
        all_cells['Y'].append(length_y)
        all_cells['Z'].append(length_z)
    dxi_prev = dxi
    dyj_prev = dyj
```
dk_prev = dk

return all_cells

def get_keys(*args):
    
    """You must specify: | Full XKEYS list | List of XKEYS you want removed |"""

    length = len(args)
    if length != 2:
        return None

    keys_original = args[0]
    keys_remove = args[1]
    keys_remaining = []

    for m in keys_original:
        if any(m in s for s in keys_remove):
            pass
        else:
            keys_remaining.append(m)

    return keys_remaining

def create_directory(*args):
    
    """You must specify: | Core Path | New Path (+1 increment) |"""

    length = len(args)
    if length != 2:
        return None

    core = args[0]
    new = args[1]

    if os.path.exists(core):
        if os.path.exists(new):
            pass
        else:
            os.mkdir(new)

    else:
        os.mkdir(core)
        os.mkdir(new)

    def write_to_file(*args):
column_width = {}
no_col_restraint = 0
length = len(args)

if length == 6:
    no_col_restraint = 1
elif length == 7:
    no_col_restraint = 0
    column_width = args[6]
else:
    return None

store_method = args[0]
dataframe = args[1]
path_original = args[2]
holder = args[3]
name = args[4]
id_unique = args[5]

datatype = ''
if store_method == 'parquet':
    path = os.path.join(path_original, folder,
    name + '.parquet')
    table = pa.Table.from_pandas(dataframe)
    write_id = id_unique
    if write_id is None:
        write_id = pq.ParquetWriter(path,
        table.schema)
    write_id.write_table(table=table)
    id_unique = write_id
    return id_unique

elif store_method == 'hdf5':
    path = os.path.join(path_original, folder,
    name + '.h5')
    if no_col_restraint == 1:
        dataframe.to_hdf(path, key=name,
        format='table', append=True)
    elif no_col_restraint == 0:
        dataframe.to_hdf(path, key=name,
        format='table', append=True, data_columns=True, complevel=
        9, complib='blosc',
        column_width)
    return None
store_method = 'parquet'

with open(filepath) as fw:
    for final_line, line in enumerate(fw, 1):
        pass

data_input = []
data_main = []
close01 = []
close02 = []
close03 = []
close04 = []
close05 = []
close06 = []
close07 = []
close08 = []
close09 = []
close10 = []
n_timestep = []
write_id_data = None
write_id_time = None
write_id_input = None
write_id_comp = None
write_id_param = None
write_id_reg = None
write_id_wells = None
write_id_wellparam = None

store_parameters = 0
n_count = 0
skip_line = 10

with open(filepath) as fp0:
    for lineNumber, line in enumerate(fp0, 1):
        x = round((lineNumber / final_line) * 100 , 2)
        # print('line: ' + str(lineNumber
        ) + ' ' + str(line))
        if skip_line == 0:
            global global_x_label
            global_x_label.config(text=str(x) + '
            n_count2 = 5
            else:
                skip_line -= 1
                if not data_input:
                    if not close01 and 'Input file' in
pre_line:
current = pre_pre_pre_line.split(
)  
n_sim, n_ver = (current[0], float
(current[2])))  
current = pre_line.split()[-1].split('.'
, '. ' + current[1])
n_file, n_file_type = (current[0]
, add_folder)
add_folder = n_file
core_path = self.home
new_path = os.path.join(core_path
, add_folder)
create_directory(core_path,
new_path)
select_folder, close01 = (add_folder, 1)

elif close01 and 'Run description' in
pre_line and not close02:
    n_title, close02 = (line[:-1], 1)
elif close02 and 'Grid dimensions' in
pre_pre_line and not close03:
current = pre_line.split() + line
.split()
    for i in current:
        if i in ['NX', 'NY', 'NZ', 'LX', 'LY', 'LZ']:
            grid_dim_dict[i] = float(
current[current.index(i) + 2])
            n_count += 1
            if n_count == len(
grid_dim_dict):
                close03 = 1
elif close03 and ('DX' in pre_line or
'DX' not in line) and not close04:
    if 'DY' in line:
        close04 = 1
else:
    current = line.split()
    temp_storage['DX'] =
temp_storage['DX'] + current
elif close04 and ('DY' in pre_line or
'DY' not in line) and not close05:
    if 'DZ' in line:
        close05 = 1
else:
    current = line.split()
    temp_storage['DY'] = temp_storage['DY'] + current
    if close05 and ('DZ' in pre_line or 'DZ' not in line) and not close06:
        if 'XKEYS' in line:
            close06 = 1
        else:
            current = line.split()
            temp_storage['DZ'] = temp_storage['DZ'] + current
    elif close06 and ('XKEYS' in pre_line or 'XKEYS' not in line) and not close07:
        current = line.split()
        for i in current:
            if i == 'ZZZZZE':
                close07 = 1
            else:
                i = i[0:len(i) - 1]
                temp_storage['XKEYS'].append(i)
    elif close07 and 'TAXIS' in pre_line and not close08:
        cells_dict = get_cells(1, 1, 1,
                                grid_dim_dict, 'full')
        cells_pos, close08 = (cells_dict['Cell'], 1)
    elif close08 and not close09:
        for x in temp_storage['XKEYS']:
            if x in line:
                x_current_addition = str(line[:-1])
                blacklist.append(x)
                cells_dict[x_current_addition], close10 = ([], 1)
            break

Timestep' in line:
    close09, data_input = (1, 1)
    keys = get_keys(temp_storage['XKEYS'], blacklist)
    cells_data = get_cell_dim(cells_dict, temp_storage)
    df_grid_data = pd.DataFrame(
cells_data)
    name_of_file, select_folder =
    ('INPUT', add_folder)
    grid_col = df_grid_data.columns
    grid_col_new = []
    for p in range(len(grid_col)):
        y = grid_col[p]
        y = y.translate({ord(i):
            None for i in ['-', '_', '[', ']', '#', '.']})
        grid_col_new.append(y)
    grid_col_new
    df_grid_data.columns =
    write_id_parquet =
    write_to_file(store_method, df_grid_data, core_path,
    select_folder, name_of_file, write_id_input, None)
    if write_id_parquet is not None:
        write_id_input =
        write_id_parquet
    elif (x_current_addition in
    pre_pre_line or x_current_addition not in line) and
    x_current_addition not in pre_line:
        current = line.split()
        cells_dict[x_current_addition] = cells_dict[x_current_addition]
        current = line.split()
        ndaysnow = float(current[current.
        index('Time') + 1])
        ndates = i_day + datetime.
        timedelta(days=float(ndaysnow))
        n_timestep = int(current[current.
        index('Timestep') + 1])
        n_pv = float(current[current.
        index('PV') + 2])
        n_pv_tot = float(pre_line.split())
n_cpu = float(line.split()[2])

add_this = {'nStep': [n_summary],
            'nDays': [ndaysnow], 'nDate': [pd.to_datetime(ndates)],
            'n_timestep': [n_timestep], 'nPV': [n_pv], 'nPVtot': [n_pv_tot], 'nCPU': [n_cpu]}

df_time = pd.DataFrame(add_this)
df_time = df_time.set_index(df_time.columns[2])

write_id_parquet = write_to_file(store_method, df_time, core_path, select_folder,
                                  name_of_file, write_id_time, None)

if write_id_parquet is not None:
    write_id_time = write_id_parquet

elif "-" in pre_line and "Component volume balance" in pre_pre_pre_line:
    current = pre_pre_pre_line.split()
    n_pressure = float(current[-2])
    current = pre_pre_line.split()
    pv_loc = current.index('PV=') + 3
    name_of_file, print_now = ('COMP',
    )
    complist = ['Time'] + ['Pressure'] + ['Component'] + current[pv_loc:-2] + [current[-2] + '
    _' + current[-1]]

    while print_now == 0:
        temp_list = []
        current = line.split()
        if "=" in line:
            print_now = 1
            getcomps = 1
            comp_temp = pd.DataFrame(
                comp_temp, columns=complist)
            comp_temp = comp_temp.
            set_index(comp_temp.columns[0])

write_id_parquet = write_to_file(store_method, comp_temp, core_path,
                                  select_folder, name_of_file, write_id_comp, None)
if write_id_parquet is not None:
    write_id_comp = write_id_parquet

else:
    try:
        string01 = current[2]
        float(string01)
        string01 = current[1]
        for r in current[2:]:
            r = float(r)
            temp_list.append(r)
        current = [pd.to_datetime(ndates)] + [n_pressure] + [string01] + temp_list
    except ValueError:
        string01 = current[1] + current[2]
        for r in current[3:]:
            r = float(r)
            temp_list.append(r)
        current = [pd.to_datetime(ndates)] + [n_pressure] + [string01] + temp_list
    comp_temp.append(current)
    if getcomps == 0:
        comp_list.append(string01)
    elif getcomps == 1:
        pass
        break
    if "-" in pre_line and "Region" in pre_pre_line:
        current = pre_pre_line.split()
        n_reg = float(current[1])
        pv_loc = current.index('PV') + 3
        name_of_file, print_now_r = ('REGION', 0)
        reg_list = ['Time'] + ['Region']
        + ['Component'] + current[pv_loc:-2] + [current[-2]] + '_'
        + current[-1])
    while print_now_r == 0:
        templistr = []
        if "Region" in line or "=" in line or line == '\n':
            print_now_r = 1
reg_temp = pd.DataFrame(
    reg_temp, columns=reg_list)
reg_temp = reg_temp.set_index
    (reg_temp.columns[0])
write_id_parquet =
write_to_file(store_method, reg_temp, core_path, 
select_folder, name_of_file, write_id_reg, None)
    if write_id_parquet is not None:
        write_id_reg =
    write_id_parquet
    reg_temp = []
else:
    current = line.split()
    try:
        string01 = current[2]
        float(string01)
        string01 = current[1]
        for z in current[2:]:
            z = float(z)
            templistr.append(z)
        current = [pd.to_datetime
            (ndates)) + [n_reg] + [string01] + templistr
    except ValueError:
        string01 = current[1] +
    current[2]
        for z in current[3:]:
            z = float(z)
            templistr.append(z)
        current = [pd.to_datetime
            (ndates)) + [n_reg] + [string01] + templistr
    reg_temp.append(current)
    break
    if "Well" in pre_line and "report" in pre_line:
        current = pre_line.split()
nwells = int(current[1])
        pv_loc = current.index('PV') + 2
        n_pv_well = float(current[pv_loc])
        current = line.split()
        ntypes = current[0].replace(',',''
        n_well_name = current[1].replace("imFinal119.py
"
try:
    T_loc = current.index('temperature')

try:
    n_well_temp = float(current[T_loc + 2])
except ValueError:
    n_well_temp = float(current[T_loc + 1].replace('=', ''))
except ValueError:
    n_well_temp = 999  # Indicates well has been closed

wellreads = 0
elif "Connection" in pre_pre_line and wellreads == 0:
    current = pre_pre_line.replace('block', '"
    current = current.split()
    current_n = pre_pre_line.split()[1:]
    current_f = []
    well_param_names = []
    for u in range(len(current)):
        if well_param == 0:
            storethis = str(current[u]) + '' + str(current_n[u])
    well_param_names.append(
        storethis)
    current_f0 = current[u].replace('/', 'per') + current_n[u].replace('/', 'per')
    current_f0 = current_f0.translate((ord(i): None for i in ['\', ',', '3']))
    current_f.append(current_f0)
if well_param == 0:
    well_param_names = ["nPVwell", 'Temperature', 'Connection'] + well_param_names
    well_col = pd.DataFrame(pd.Series(well_param_names)).columns[0])
    name_of_file = 'WELLPARAM'
write_id_parquet = write_to_file(store_method, well_col, core_path,
525 select_folder, name_of_file, write_id_wellparam)
526     if write_id_parquet is not None:
527         write_id_wellparam = write_id_parquet
528         well_param = 1
529         read_now = 0
530         while wellreads == 0 and read_now == 0:  # and line != '\n':
531             current = []
532             temp_list_w = []
533             if "---" in line:
534                 current = pre_line.split()
535                 for z in current[1:]:
536                     # print('current: ' + str(z))
537             if normal_length == 0:
538                 normal_length = len(well_param_names) - 2
539                 current_length = len(current)
540             if normal_length !=
541                 e- ' in z:
542                 pass  # print('yup, there is extra -, and no e-')  # could be just negative number
543                 elif '-' in z and 'e
544                     split('e-')
545                     split_it = z.
546                     base_number =
547                     split_it[0]
548                     if '-' in
549                     base_number.split('-')
550                     new_split =
551                     new_split > 1:
552                     first_number = float(new_split[0])
553                     temp_list_w.append(first_number)
second_number = float('-' + new_split[1] + 'e' + split_it[1])

    temp_list_w.append(second_number)

    else:
        z = float(z)
        temp_list_w.append(z)
        string01 = current[0]
        current = pd.to_datetime(ndates) + [n Wells] + [n well_name] + [n types] + [
        n pv well] + [n well temp] + [string01] + temp_list_w

elif 'Total' in pre_pre_line:
    name_of_file = 'WELLS'
    if store_once_one == 0:
        well_list_w = ['Time'] +
        ['nWell'] + ['nWellName'] + ['nType'] + ['nPVwell'] + ['
        nWellTemp'] + ['Connection'] + current_f

for e in well_list_w[1::]:
        col_width[str(e)] =

store_once_one = 1

    well_temp = pd.DataFrame(
    well_temp, columns=well_list_w)

    well_temp = well_temp.

    set_index(well_temp.columns[0])

    write_id_parquet =

    write_to_file(store_method, well_temp, core_path,
    select_folder, name_of_file, write_id_wells, col_width)

if write_id_parquet is not

None:

    write_id_wells =

write_id_parquet

    well_temp = []

    wellreads, read_now, sum_qst

    = (1, 1, 1)

    break

    if current:

        well_temp.append(current)

    break

if wellreads == 1 and read_now == 1:

    if ("Summary" in line and "

Timestep" in line) or 'CPU summary report' in line:

        name_of_file, cell_number = (
'DATA', len(cells_dict['Cell']))

current_dict['Time'] = [pd.to_datetime(ndates)] * cell_number

current_dict['Days'] = [float(ndaysnow)] * cell_number

current_dict['Timestep'] = [int(n_timestep)] * cell_number

'Cell' =
cells_data['Cell']

testing = pd.DataFrame(
current_dict)

cols = testing.columns.tolist()

cols = ['Time', 'Days', 'Timestep', 'Cell'] + cols[: -4]

testing = testing[cols]
testing = testing.set_index(testing.columns.tolist(), index='Time'))

if store_data_once == 0:
df_col = testing.columns.tolist()

for f in df_col:
    col_width_data[str(f)] = 50

store_data_once = 1

write_id_parquet =

write_to_file(store_method, testing, core_path, select_folder, name_of_file, write_id_data, col_width_data)

if write_id_parquet is not None:

write_id_parquet

write_id_data =

wellreads, start_main,
current_dict = (0, 0, {})

else:

for xi in keys:
    if xi in pre_pre_line and

start_main == 0:

start_main = 1

remove = pre_pre_line

.rstrip()
if len(store_col) !=
len(keys):
x_current =
remove.translate({ord(i): None for i in ['_', '[', ','], ' ', '(', ')', '-', '/', '%', '°']})
store_col[remove]
= x_current
else:
    if
store_parameters == 0:
test_col = pd
  .DataFrame(pd.Series(store_col))
  test_col.
  set_index(test_col.columns[0])
  name_of_file
= 'PARAMETERS'

write_id_parquet = write_to_file(store_method, test_col, core_path, select_folder, name_of_file, write_id_param)
    if
write_id_parquet is not None:
write_id_param = write_id_parquet
store_parameters = 1
x_current =
store_col[remove]
    elif xi in line and
start_main == 1:
    start_main = 0
    break

while start_main == 1:
current = line.split()
    if x_current not in
current_dict:
current_dict[
  x_current] = list(map(float, current))
    elif current:
current_dict[
  x_current] = current_dict[x_current] + list(map(float, current))
break
pre_pre_pre_line = pre_pre_line
pre_pre_line = pre_line
pre_line = line

if write_id_data:
    write_id_data.close()
if write_id_time:
    write_id_time.close()
if write_id_input:
    write_id_input.close()
if write_id_comp:
    write_id_comp.close()
if write_id_param:
    write_id_param.close()
if write_id_reg:
    write_id_reg.close()
if write_id_wells:
    write_id_wells.close()
if write_id_wellparam:
    write_id_wellparam.close()

time_elapsed = datetime.datetime.now() -
    start_time

# print('Time elapsed (hh:mm:ss.ms) {}'.format(time_elapsed))

# END OF IMPORT / CONVERT FUNCTION
#

#


tp = ThreadPoolExecutor(1)

def threaded(fn):
    def wrapper(*args, **kwargs):
        return tp.submit(fn, *args, **kwargs)
    return wrapper

class PopupWindow(object):
    def __init__(self, master):
        top = self.top = Toplevel(master)
        self.l = Label(top, text='Rows:', relief=SUNKEN)
        self.l.pack(side=LEFT, padx=1, pady=3, ipadx=1, ipady=1)
        self.e = Entry(top, width=4, relief=SUNKEN)
self.e.pack(side=LEFT, padx=3, pady=3, ipady=1)
self.l2 = Label(top, text='Columns: ', relief=SUNKEN)
self.l2.pack(side=LEFT, padx=1, pady=3, ipadx=1, ipady=1)
self.e2 = Entry(top, width=4, relief=SUNKEN)
self.e2.pack(side=LEFT, padx=3, pady=3, ipady=1)
self.b = ttk.Button(top, text='ok', command=self.cleanup)
sel.b.pack(side=LEFT, padx=3, pady=3)
self.value = None
self.value2 = None

def cleanup(self):
    self.value = self.e.get()
    self.value2 = self.e2.get()
    global chosen_rows_alt, chosen_cols_alt
    chosen_rows_alt = self.value
    chosen_cols_alt = self.value2
    self.top.destroy()

root = tk.Tk

class SimPlotJIN(root):
    def __init__(self, *args, **kwargs):
        # When you call
        # the class, this will always run. Restart pc -> want
        # something ie. explorer.exe, keyboard to load, etc..
        tk.Tk.__init__(self, *args, **kwargs)  # tkinter
        tk.Tk.iconbitmap(self, default='gui_icon.ico')
        tk.Tk.wm_title(self, 'SimPlotJIN')
        tk.Tk.geometry(self, '1300x1000')
        status = Label(self, text='..RAM usage', anchor='w', relief=SUNKEN)
        status.pack(side=BOTTOM, fill='both')

        self.nb = ttk.Notebook(self)
        self.nb.pack(expand=1, fill='both')
        self.frames = {}
        labels = ['Start', 'Page One', 'Page Two', 'Page Three']
        classes = [StartPage, PageOne, PageTwo, PageThree}
for i in range(len(classes)):
    page = classes[i]
    frame = page(parent=self.nb, controller=self)

    # Calls the class
    self.frames[page] = frame
    self.nb.add(frame, text=labels[i])

def prep_local_param(event):
    selection = event.widget.select()
    tab = event.widget.tab(selection, 'text')

global current_tab

current_tab = tab

current = global_sim_data

if tab == 'Page One' and current:
    alls = list(global_sim_data_listbox.get(0
    , END))

    for num in reversed(range(len(alls))):
        keys = alls[num]

        if 'DATA' not in keys or 'PARAMETERS' not in keys:
            global_sim_data_listbox.delete(
        num)

else:
    store_path = current[keys][0]
    path_param = os.path.join(  
    store_path, 'PARAMETERS' + '.parquet')
    avail_param = pd.read_parquet(path_param)

    if 'WELLPARAM' in current[keys]:
        path_wellparam = os.path.join(  
        store_path, 'WELLPARAM' + '.parquet')
        avail_wellparam = pd.
        read_parquet(path_wellparam)

    store_wellparam = 
    avail_wellparam.iloc[:, 0].tolist()  
    path_wells = os.path.join(  
    store_path, 'WELLS' + '.parquet')
    wells_col = pd.read_parquet(path_wells).columns.tolist()

    temp_dict = {}
    shown = store_wellparam
    hidden = wells_col[3:]

    for ik in list(range(len(  
shown))):
temp_dict[shown[ik]] = hidden[ik]

global dict_paramv2
dict_paramv2[keys] =
tolist()

store_param = avail_param.index.

full_list = [store_path] +

store_param

global dict_param
dict_param[keys] = full_list

local_dict = {}

for param_user in store_param:
    param_backend = avail_param.

loc[param_user, 0]

local_dict[param_user] =

param_backend

global prep_pageone

prep_pageone[keys] = local_dict

self.nb.bind('<<NotebookTabChanged>>',
p prep_local_param)

def on_closing(self):
    if messagebox.askokcancel('Quit', 'Do you want to quit?'):
        SimPlotJIN().quit()

class StartPage(tk.Frame):
    # Creates a frame that we call the start page. then we can make more pages, and show them with show_frame method

def __init__(self, parent, controller):
    self.controller = controller

self.parent = parent

self.Frame._init_(self, parent)

self.filename = '...'

self.list1 = []

self.collect_thread = []

self.text_here = ''

self.count = 0

self.home_location = os.path.join(os.path.
expanduser('~'), 'Documents', 'ProjIORCoreSim')

self.read_location = self.home_location
self.simulation_data_found = {}
self.simulation_data_to_plot = {}
self.simulation_data_sorted = {}
data_full = {}
bigframe = Frame(self, bg='#CD3333')
bigframe.pack(expand=True, fill='both', padx=1, pady=1)
f1 = Frame(bigframe, bg='orange')
f2 = Frame(bigframe, bg='yellow', bd=3)
f2a = Frame(f2, bg='grey', bd=3)
f2b = Frame(f2, bg='black', bd=3)
f2c = Frame(f2, bg='blue', bd=3)
f3 = Frame(bigframe, bg='green')
f1.pack(side=TOP, expand=0, fill='both', padx=3, pady=3)
f2.pack(side=TOP, expand=0, fill='both', padx=3, pady=3)
f3.pack(side=TOP, expand=1, fill='both', padx=3, pady=3)
f2a.grid(column=0, row=0)
f2b.grid(column=1, row=0)
f2c.grid(column=2, row=0)
button_import = ttk.Button(f1, text='Import..', command=lambda: self.load_file(f1))
button_import.grid(column=0, row=0, sticky='nw', padx=3, pady=3)
button_save = ttk.Button(f1, text='Save to..', command=lambda: self.save_file(f1))
button_save.grid(column=0, row=1, sticky='nw', padx=3, pady=3)
self.button_convert = ttk.Button(f1, text='Convert', command=lambda: self.convert_file(f1))
self.button_convert.grid(column=2, row=0)
button_read = ttk.Button(f1, text='Read from..', command=lambda: self.read_folder(f1, f2a.list_parent))
button_read.grid(column=0, row=2, sticky='nw', padx=3, pady=3)
button_add = ttk.Button(f2b, text='Add', command=lambda: self.add_name(parent=f2a.list_parent, child=f2c)
list_child))
    button_add.pack()
    button_del = ttk.Button(f2b, text='Remove',
                            command=lambda: self.remove_name(child=f2c.list_child))
    button_del.pack()

    f1.label = Label(f1, text=self.filename, width=1,
                     relief=SUNKEN)
    f1.label.grid(column=1, row=0, padx=3, pady=3,
                   ipadx=250, ipady=2)
    f1.label4 = Label(f1, text=self.home_location,
                      width=1, relief=SUNKEN)
    f1.label4.grid(column=1, row=1, padx=3, pady=3,
                   ipadx=250, ipady=2)
    f1.label5 = Label(f1, text=self.read_location,
                      width=1, relief=SUNKEN)
    f1.label5.grid(column=1, row=2, padx=3, pady=3,
                   ipadx=250, ipady=2)
    global global_x_label
    global_x_label = Label(f1, text='0.00 %', width=5)
    global_x_label.grid(column=3, row=0, padx=3, pady=3,
                         ipadx=15, ipady=2)

    f2a.label6 = Label(f2a, text='Available
Simulation Cases ')
    f2a.label6.grid(column=0, row=0, padx=3, pady=3)
    f2a.list_parent = Listbox(f2a, height=10,
                              selectmode=EXTENDED, relief=SUNKEN)
    f2a.list_parent.grid(column=0, row=1, padx=3,
                          pady=3)
    f2c.label7 = Label(f2c, text='Cases available
for plotting ')
    f2c.label7.grid(column=0, row=0, padx=3, pady=3)
    f2c.list_child = Listbox(f2c, height=10,
                             selectmode=EXTENDED, relief=SUNKEN)
    f2c.list_child.grid(column=0, row=1, padx=3, pady=3)

    self.local_simulations(path_to_check=self.
                            read_location, f2a_listbox=f2a.list_parent)

    def local_simulations(self, path_to_check,
                          f2a_listbox):
        store_list = []
for path, dirs, files in os.walk(path_to_check):
    store_list = []
    for i in files:
        current = i.split('.')[
        if current[0] not in ['COMP', 'DATA', 'INPUT', 'PARAMETERS', 'REGION', 'TIME', 'WELLS', 'WELLPARAM']:
            break
        elif current[1] == 'parquet':
            store_list.append(current[0])
        if store_list:
            sim_folder = os.path.basename(path)
            combined = [path] + store_list
            f2a_listbox.insert(END, sim_folder)
            self.simulation_data_found[sim_folder] = combined

def add_name(self, parent, child):
    cursors = parent.curselion()
    alls = list(child.get(0, END))
    global global_sim_data_listbox
    for item in list(cursors):
        x_add = parent.get(item)
        self.simulation_data_to_plot[x_add] = self.
        simulation_data_found[x_add]
        if x_add not in alls:
            child.insert(END, x_add)
            global_sim_data_listbox.insert(END, x_add)

    global global_sim_data
    global_sim_data = self.simulation_data_to_plot

def remove_name(self, child):
    cursors = child.curselion()
    global global_sim_data_listbox
    for item in reversed(cursors):
        x_del = child.get(item)
        self.simulation_data_to_plot.pop(x_del, None)
        child.delete(item)
        global_sim_data_listbox.delete(item)

def load_file(self, cont):
    self.filename = askopenfilename(title='Select .
        out file', filetypes=((OUT File', '*.out'),))
    if self.filename:
def save_file(self, cont):
    self.home_location = filedialog.askdirectory(
        title='Select save folder')
    if self.home_location:
        cont.label4['text'] = self.home_location

def read_folder(self, frame, f2a_listbox):
    path_read = filedialog.askdirectory(title='Select read folder')
    if path_read:
        f2a_listbox.delete(0, END)
        frame.label5['text'] = path_read
        self.local_simulations(path_to_check=path_read,
                               f2a_listbox=f2a_listbox)
    self.read_location = path_read

@threaded
def convert_file(self, cont):
    self.button_convert['state'] = 'disabled'
    Simulation(save_loc=self.home_location, file_name
               =self.filename).convert()
    self.button_convert['state'] = 'normal'

class PageOne(tk.Frame):
    def __init__(self, parent, controller):
        self.controller = controller
        self.parent = parent
        tk.Frame.__init__(self, parent)
        self.f1_input = Frame(self, bg='grey')
        self.f1_input.pack(side=TOP, padx=3, pady=3,
                           expand=0, fill='both')
        self.f2_toolkit = Frame(self)
        self.f2_toolkit.pack(side=TOP, fill='both',
                             expand=False)
        self.f2_plot = Frame(self)
        self.f2_plot.pack(side=TOP, padx=10, pady=10,
                           expand=1, fill='both')
    global global_sim_data_listbox
    global_sim_data_listbox = Listbox(self.f1_input,
                                       height=5, selectmode=SINGLE, relief=SUNKEN,
                                       exportselection=False)
    global_sim_data_listbox.pack(side=LEFT, padx=2,
912    pady=2, fill='y')
913    global_sim_data_listbox.bind('<<ListboxSelect>>',
914        self.get_selected_item_prep)
915    self.prep_sim_parameters = Listbox(self.f1_input,
916        height=5, selectmode=SINGLE, relief=SUNKEN, width=24,
917        exportselection=False)
918    self.prep_sim_parameters.pack(side=LEFT, padx=2,
919        pady=2, fill='y')
920    self.prep_sim_parameters.bind('<<ListboxSelect>>',
921        self.get_folded_properties)
922    self.local_sim_parameters = Listbox(self.f1_input,
923        height=5, selectmode=EXTENDED, relief=SUNKEN)
924    self.local_sim_parameters.pack(side=LEFT, padx=2,
925        pady=2, fill='y')
926    self.local_sim_parameters.bind('<<ListboxSelect>>',
927        self.get_multiple_items)
928    self.f3 = Frame(self.f1_input)
929    self.f3.pack(side=LEFT, expand=0, fill='both')
930    self.f3a = Frame(self.f3)
931    self.f3b = Frame(self.f3)
932    self.f3c = Frame(self.f3)
933    self.f3a.pack(side=TOP, expand=1)
934    self.f3b.pack(side=TOP, expand=1)
935    self.f3c.pack(side=TOP, expand=1)
936    self.fontsize = 12
937    self.hold3 = IntVar()
938    self.hold_choice3 = Checkbutton(self.f3b, text='Hold3',
939        variable=self.hold3, onvalue=0, offvalue=1)
940    self.hold_choice3.pack()
941    self.grid_dropdown_font = ttk.Combobox(self.f3b,
942        height=1, width=7, state='readonly')
943    self.grid_dropdown_font.pack()
944    self.label_font = Label(self.f3b, text='Font size :
945        ' + str(self.fontsize), height=1, width=10)
946    self.label_font.pack()
947    self.grid_dropdown_font['values'] = list(range(1,
948        100+1,1))
949    self.grid_dropdown_font.current(self.fontsize-1)
950    self.grid_dropdown_font.bind('<<ComboboxSelected
951        >>', self.get_fontsize)
952    self.button_add_plots = ttk.Button(self.f3b, text
953        = 'Add', width=10, command=self.add_to_plot_list)
954    self.button_add_plots.pack()
955    self.button_remove_plots = ttk.Button(self.f3b,
956        text='Remove', width=10, command=self.

Page 28 of 111
remove_from_plot_list()
        self.button_remove_plots.pack()
        self.button_remove_x = ttk.Button(self.f3b, text=
            'Clear X', width=10, command=lambda: self.clear_xy(
                typedata='X'))
        self.button_remove_x.pack()
        self.button_remove_y = ttk.Button(self.f3b, text=
            'Clear Y', width=10, command=lambda: self.clear_xy(
                typedata='Y'))
        self.button_remove_y.pack()

        self.listboxes_frame = Frame(self.f1_input, bg='red')
        self.listboxes_frame.pack(side=LEFT, padx=2, pady =2, fill='both', expand=0)

        self.xy_listbox_frame = Frame(self.
                                   listboxes_frame, bg='white')
        self.xy_listbox_frame.pack(side=TOP, padx=2, pady =2, fill='both', expand=0)
        self.x_listbox = Listbox(self.xy_listbox_frame, height=1, selectmode=None, relief=SUNKEN)
        self.x_listbox.grid(column=0, row=0, sticky='nw', padx=2, pady=2, ipady=2)
        self.x_button_frame = Frame(self.xy_listbox_frame , width=50, height=25)
        self.x_button_frame.grid(column=1, row=0, sticky='nw', padx=2, pady=2)
        self.x_button_frame.pack_propagate(0)
        self.x_button = ttk.Button(self.x_button_frame, text='Add X', command=lambda: self.add_to_xy(typedata='X'))
        self.x_button.pack(expand=1, fill='both')

        self.y_listbox = Listbox(self.xy_listbox_frame, height=1, selectmode=None, relief=SUNKEN)
        self.y_listbox.grid(column=0, row=1, sticky='nw', padx=2, pady=2, ipady=2)
        self.y_button_frame = Frame(self.xy_listbox_frame , width=50, height=25)
        self.y_button_frame.grid(column=1, row=1, sticky='nw', padx=2, pady=2)
        self.y_button_frame.pack_propagate(0)
        self.y_button = ttk.Button(self.y_button_frame, text='Add Y', command=lambda: self.add_to_xy(typedata='Y'))
self.y_button.pack(expand=1, fill='both')

self.z_listbox_frame = Frame(self.
    listboxes_frame, bg='blue')
self.z_listbox_frame.pack(side=TOP, padx=2, pady =2, fill='both', expand=1)
self.pageone_listbox_plot = Listbox(self.
    z_listbox_frame, height=5, selectmode=EXTENDED, relief=
    SUNKEN)
self.pageone_listbox_plot.pack(side=LEFT, padx=2
    , pady=2, fill='both', expand=1)
self.pageone_listbox_plot.bind('<<ListboxSelect
    >>', self.get_plot_titles)

self.checkmarks = Frame(self.f1_input, bg='black
')
self.checkmarks.pack(side=LEFT, expand=0, fill='both')
self.checkmarks_a = Frame(self.checkmarks, bg='green')
self.checkmarks_a.pack(side=TOP, expand=0, fill='both')
self.checkmarks_b = Frame(self.checkmarks, bg='white')
self.checkmarks_b.pack(side=TOP, expand=0, fill='both')
self.checkmarks_c = Frame(self.checkmarks, bg='orange')
self.checkmarks_c.pack(side=TOP, expand=1, fill='both')
self.checkmarks_d = Frame(self.checkmarks, bg='orange')
self.checkmarks_d.pack(side=TOP, expand=1, fill='both')

self.figs = 0
self.label_figs = Label(self.checkmarks_a, text=
    'Figures: ' + str(self.figs), height=1, relief=SUNKEN,
    width=9)
self.label_figs.grid(column=0, row=0, sticky='nw
    ', padx=3, pady=3, ipady=2)
self.grid_dropdown = ttk.Combobox(self.
    checkmarks_a, height=1, width=4)
self.grid_dropdown.grid(column=1, row=0, sticky=
    'nw', padx=3, pady=3, ipady=2)
self.grid_button = ttk.Button(self.checkmarks_a,
text='Row: Col:', command=self.popup)
    self.grid_button.grid(column=2, row=0, sticky='nw', padx=3, pady=3)
    self.sharex = IntVar()
    self.sharey = IntVar()
    self.showtime = IntVar()
    self.showsimcase = IntVar()
    self.plottype = IntVar()
    self.hold = IntVar()
    self.hold2 = IntVar()
    self.sharex_choice = Checkbutton(self.
        checkmarks_b, text='Share X', variable=self.sharex,
        onvalue=1, offvalue=0, bg='grey')
    self.sharex_choice.grid(column=0, row=0, sticky='nw', padx=3, pady=3)
    self.sharey_choice = Checkbutton(self.
        checkmarks_b, text='Share Y', variable=self.sharey,
        onvalue=1, offvalue=0, bg='grey')
    self.sharey_choice.grid(column=1, row=0, sticky='nw', padx=3, pady=3)
    self.showtime_choice = Checkbutton(self.
        checkmarks_b, text='Time', variable=self.showtime,
        onvalue=0, offvalue=1, bg='grey')
    self.showtime_choice.grid(column=0, row=1, sticky='nw', padx=3, pady=3)
    self.showsimcase_choice = Checkbutton(self.
        checkmarks_b, text='Simcase', variable=self.showsimcase,
        onvalue=0, offvalue=1, bg='grey')
    self.showsimcase_choice.grid(column=1, row=1, sticky='nw', padx=3, pady=3)
    self.hold_choice = Checkbutton(self.checkmarks_b,
        text='Hold', variable=self.hold, onvalue=0, offvalue=1,
        bg='grey')
    self.hold_choice.grid(column=2, row=0, sticky='nw', padx=3, pady=3)
    self.hold_choice2 = Checkbutton(self.
        checkmarks_b, text='Hold2', variable=self.hold2, onvalue=1,
        offvalue=0, bg='grey')
    self.hold_choice2.grid(column=2, row=1, sticky='nw', padx=3, pady=3)
    self.xyz = IntVar()
    self.xy = Radiobutton(self.checkmarks_c, text='xy', variable=self.xyz, value=1, bg='grey', command= lambda: self.set_xyz())
    self.xy = Radiobutton(self.checkmarks_c, text='
1012 yx', variable=self.xyz, value=2, bg='grey', command=
    lambda: self.set_xyz()
1013     self.xz = Radiobutton(self.checkmarks_c, text='xz', variable=self.xyz, value=3, bg='grey', command=
    lambda: self.set_xyz())
1014     self.zx = Radiobutton(self.checkmarks_c, text='zx', variable=self.xyz, value=4, bg='grey', command=
    lambda: self.set_xyz())
1015     self.yz = Radiobutton(self.checkmarks_c, text='zy', variable=self.xyz, value=5, bg='grey', command=
    lambda: self.set_xyz())
1016     self.zy = Radiobutton(self.checkmarks_c, text='yz', variable=self.xyz, value=6, bg='grey', command=
    lambda: self.set_xyz())
1017     self.plottype_choice = Checkbutton(self.
        checkmarks_c, text='Plot type', variable=self.plottype, onvalue=0, offvalue=1, bg='grey')
1018     self.xyz_reset = Radiobutton(self.checkmarks_d, text='reset', variable=self.xyz, value=9, bg='grey', command=lambda: self.set_xyz())
1019     self.xdays = Radiobutton(self.checkmarks_d, text='x-days', variable=self.xyz, value=7, bg='grey', command=lambda: self.set_xyz())
1020     self.ydays = Radiobutton(self.checkmarks_d, text='y-days', variable=self.xyz, value=8, bg='grey', command=lambda: self.set_xyz())
1022     self.xy.grid(column=0, row=1, sticky='nw', padx=3, pady=3)
1023     self.yx.grid(column=1, row=1, sticky='nw', padx=3, pady=3)
1024     self.xz.grid(column=2, row=1, sticky='nw', padx=3, pady=3)
1025     self.zx.grid(column=0, row=2, sticky='nw', padx=3, pady=3)
1026     self.yz.grid(column=1, row=2, sticky='nw', padx=3, pady=3)
1027     self.zy.grid(column=2, row=2, sticky='nw', padx=3, pady=3)
1028     self.plottype_choice.grid(column=3, row=1, sticky='nw', padx=3, pady=3)
1029     self.xyz_reset.grid(column=0, row=0, sticky='nw', padx=3, pady=3)
1030     self.xdays.grid(column=1, row=0, sticky='nw', padx=3, pady=3)
padx=3, pady=3)
        self.ydays.grid(column=2, row=0, sticky='nw',
                        padx=3, pady=3)

        self.slide_and_gelmod = Frame(self.f1_input, bg='black')
        self.slide_and_gelmod.pack(side=LEFT, expand=0, fill='both')
        self.sliders = Frame(self.slide_and_gelmod, bg='red')
        self.sliders.pack(side=TOP, expand=0, fill='both')
        self.prep_gelmods = Frame(self.slide_and_gelmod, bg='grey')
        self.prep_gelmods.pack(side=TOP, expand=1, fill='both')
        self.gelmods = Frame(self.prep_gelmods, bg='grey')
        self.gelmods.pack(side=LEFT, expand=1, fill='both')
        self.buttons_gelmods = Frame(self.prep_gelmods, bg='grey')
        self.buttons_gelmods.pack(side=LEFT, expand=1, fill='both')

        # Component (1) - Na (ppm)
        self.compl1 = Text(self.gelmods, height=1, width=6)
        self.compl1mid = Text(self.gelmods, height=1, width=5)
        self.compl1end = Text(self.gelmods, height=1, width=6)
        self.compl1_label = Label(self.gelmods, text='Na (ppm):', height=1, relief=SUNKEN, width=9)
        self.compl1_label2 = Label(self.gelmods, text=':', height=1)
        self.compl1_label3 = Label(self.gelmods, text=':', height=1)
        self.compl1_label.grid(column=0, row=0, sticky='nw', padx=3, pady=3)
        self.compl1.grid(column=1, row=0, sticky='nw', padx=3, pady=3)
        self.compl1_label2.grid(column=2, row=0, sticky='nw', padx=3, pady=3)
self.complmid.grid(column=3, row=0, sticky='nw', padx=3, pady=3)

self.compl_label3.grid(column=4, row=0, sticky='nw', padx=3, pady=3)

self.complend.grid(column=5, row=0, sticky='nw', padx=3, pady=3)

# Component (2) - Ca (ppm)
self.comp2 = Text(self.gelmods, height=1, width=6)

self.comp2mid = Text(self.gelmods, height=1, width=5)

self.comp2end = Text(self.gelmods, height=1, width=6)

self.comp2_label = Label(self.gelmods, text='Ca (ppm):', height=1, relief=SUNKEN, width=9)

self.comp2_label2 = Label(self.gelmods, text=':', height=1)

self.comp2_label3 = Label(self.gelmods, text=':', height=1)

self.comp2_label.grid(column=0, row=1, sticky='nw', padx=3, pady=3)

self.comp2.grid(column=1, row=1, sticky='nw', padx=3, pady=3)

self.comp2_label2.grid(column=2, row=1, sticky='nw', padx=3, pady=3)

self.comp2mid.grid(column=3, row=1, sticky='nw', padx=3, pady=3)

self.comp2_label3.grid(column=4, row=1, sticky='nw', padx=3, pady=3)

self.comp2end.grid(column=5, row=1, sticky='nw', padx=3, pady=3)

# Component (3) - T (°C)
self.comp3 = Text(self.gelmods, height=1, width=6)

self.comp3mid = Text(self.gelmods, height=1, width=5)

self.comp3end = Text(self.gelmods, height=1, width=6)

self.comp3_label = Label(self.gelmods, text='Temp (°C):', height=1, relief=SUNKEN, width=9)

self.comp3_label2 = Label(self.gelmods, text=':', height=1)
self.comp3_label3 = Label(self.gelmods, text=':', height=1)

self.comp3_label3.grid(column=0, row=2, sticky='nw', padx=3, pady=3)
self.comp3.grid(column=1, row=2, sticky='nw', padx=3, pady=3)
self.comp3_label2.grid(column=2, row=2, sticky='nw', padx=3, pady=3)
self.comp3mid.grid(column=3, row=2, sticky='nw', padx=3, pady=3)
self.comp3_label3.grid(column=4, row=2, sticky='nw', padx=3, pady=3)
self.comp3end.grid(column=5, row=2, sticky='nw', padx=3, pady=3)

# self.complbutton.grid(column=0, row=0, sticky='nw', padx=3, pady=3)
self.complbutton = ttk.Button(self.
buttons_gelmods, text='Set', command=lambda: self.
set_gelmmod())
self.complbutton.pack(side=TOP, expand=1, fill='both', padx=3, pady=1)
self.complbutton2 = ttk.Button(self.
buttons_gelmmods, text='Reset', command=lambda: self.
reset_gelmmod())
self.complbutton2.pack(side=TOP, expand=1, fill='both', padx=3, pady=1)
self.complbutton3 = ttk.Button(self.
builtins_gelmods, text='Plot gelmod', command=lambda: self.
plot_gelmmod())
self.complbutton3.pack(side=TOP, expand=1, fill='both', padx=3, pady=1)

# SLIDERS
self.slidex_label1, self.slidex_left, self.
slidex_right, self.slidex_label2, self.valuex1, self.
freezex1 = [None] * 6
self.slidey_label1, self.slidey_left, self.
slidey_right, self.slidey_label2, self.valuey1, self.
freezey1 = [None] * 6
self.slidz_label1, self.slidz_left, self.
slidz_right, self.slidz_label2, self.valuez1, self.
freeez1 = [None] * 6
self.slidetime_label1, self.slidetime_left, self.
slidetime_right, self.slidetime_label2, self.
freeztime1 = [None] * 6
.slidetime_right, self.slidetime_label2, self.valuetime1, self.freezetimel = [None] * 6
self.last_settings = {}
self.last_settings_old = {}
for dim in ['imin', 'imax', 'jmin', 'jmax', 'kmin', 'kmax', 'tmin', 'tmax']:
    self.last_settings[dim] = 1
self.x_input, self.x_time = (None, None)
# SLIDERS
self.f6 = Frame(self.f1_input, bg='white')
self.f6.pack(side=LEFT, expand=1, fill='both')
self.plotlabels = Frame(self.f6)
self.plotlabels.pack(side=TOP, expand=1, fill='both')
self.plotlabels_toppart = Frame(self.plotlabels, bg='black')
self.plotlabels_toppart.pack(side=TOP, expand=0, fill='x')
self.plotlabels_labels = Frame(self.plotlabels_toppart, bg='red')
self.plotlabels_labels.pack(side=LEFT, expand=0, fill='both')
self.plotlabels_buttons = Frame(self.plotlabels_toppart)
self.plotlabels_buttons.pack(side=LEFT, expand=1, fill='both')
self.ptitle_label = Label(self.plotlabels_labels, text='Plot title: ', height=1, relief=SUNKEN, width=9)
self.ptitle_label = Label(self.plotlabels_labels, text='X label: ', height=1, relief=SUNKEN, width=9)
self.ptitle_label = Label(self.plotlabels_labels, text='Y label: ', height=1, relief=SUNKEN, width=9)
self.ptitle = Entry(self.plotlabels_labels, width=40)
self.ptitle = Entry(self.plotlabels_labels, width=40)
self.ptitle = Entry(self.plotlabels_labels, width=40)
self.ptitle_label.grid(column=0, row=0, sticky='nw', padx=3, pady=3)
self.ptitle.grid(column=1, row=0, sticky='nw', padx=3, pady=3)
    plotlabels_buttons, text='Set', command=lambda: self.
    set_plot_labels())
self.title_set.pack(anchor='nw', expand=1, fill='y', padx=3, pady=3)

self.title_reset = ttk.Button(self.
    plotlabels_buttons, text='Reset', command=lambda: self.
    reset_plot_labels())
self.title_reset.pack(anchor='nw', expand=1, fill='y', padx=3, pady=3)

self.pxtitle_label.grid(column=0, row=1, sticky='nw', padx=3, pady=3)
self.pxtitle.grid(column=1, row=1, sticky='nw', padx=3, pady=3)

self.pytitle_label.grid(column=0, row=2, sticky='nw', padx=3, pady=3)
self.pytitle.grid(column=1, row=2, sticky='nw', padx=3, pady=3)

button1 = ttk.Button(self.fl_input, text='Plot
it!', command=self.plot_graphv2)
button1.pack(padx=3, pady=3)

button2 = ttk.Button(self.fl_input, text='Delete
all', command=lambda: self.delete_figures(2))
button2.pack(padx=3, pady=3)

button3 = ttk.Button(self.fl_input, text='Save
setup', command=lambda: self.store_settings())
button3.pack(padx=3, pady=3)

button4 = ttk.Button(self.fl_input, text='Restore', command=lambda: self.restore_settings())
button4.pack(padx=3, pady=3)
self.doitonce = 0
self.properties_available = {}
self.last_select = []
self.properties_conversion = {}
self.properties_plotthese = {}
self.reference = {}
self.final_plot_data = {}
self.canvas = None
self.toolbar = None
self.fig_grid_size = {}
self.chosen_rows_alt = None
self.chosen_cols_alt = None
self.w = None
self.browse_days = {}
self.tmin_stored, self.tmax_stored = (None, None)

self.current_selection_v2 = None
self.merged_listbox_items = {}
sel.simcase_child = None
self.simcase = None
self.simcase_path = None
self.data_conversion = {}
self.plot_id = None
self.plot_rdy = {}
self.plot_x = {}
self.plot_y = {}
self.simcase_ijk_count = {}
self.plot_id_old = {}
self.plot_id_hist = []
self.settings_stored = 0
self.tlimits = []

self.comp_na_start = 0
self.comp_na_mid = 2000
self.comp_na_end = 16000
self.comp1.insert(END,self.comp_na_start)
self.comp1mid.insert(END,self.comp_na_mid)
self.comp1end.insert(END,self.comp_na_end)

self.comp_ca_start = 0
self.comp_ca_mid = 50
self.comp_ca_end = 500
self.comp2.insert(END,self.comp_ca_start)
self.comp2mid.insert(END,self.comp_ca_mid)
self.comp2end.insert(END,self.comp_ca_end)

self.comp_temp_start = 10
self.comp_temp_mid = 10
self.comp_temp_end = 140
self.comp3.insert(END,self.comp_temp_start)
self.comp3mid.insert(END,self.comp_temp_mid)
self.comp3end.insert(END,self.comp_temp_end)

self.comp_na = list(range(self.comp_na_start,
                         self.comp_na_end+1, self.comp_na_mid))
self.comp_ca = list(range(self.comp_ca_start,
                         self.comp_ca_end+1, self.comp_ca_mid))
self.comp_temp = list(range(self.comp_temp_start,
                           self.comp_temp_end+1, self.comp_temp_mid))
self.change_plot_label_current = None
self.shown_title_old = None
self.plot_title = None
self.plot_xlabel = None
self.plot_ylabel = None

def set_plot_labels(self):
    if self.change_plot_label_current,
        value = self.change_plot_label_current
        title_input = self.pxtitle.get()
        user_title = str(title_input)
        split_title = user_title.split(' ')
        new_title = ' '.join(split_title)
        xlabel, ylabel = (None, None)
        user_xlabel, user_ylabel = (self.pxtitle.get(), self.pytitle.get())
        if user_xlabel:
            xlabel = str(user_xlabel)
            self.plot_xlabel = xlabel
            if user_ylabel:
                ylabel = str(user_ylabel)
                self.plot_ylabel = ylabel
                newvalues = {'shown_title': new_title,
                              'xlabel': xlabel, 'ylabel': ylabel}
                self.plot_rdy[value][1]['title'].update(newvalues)
            else:
                title_input = self.pxtitle.get()
                if title_input:
                    user_title = str(title_input)
                    split_title = user_title.split(' ')
                    new_title = ' '.join(split_title)
                    self.plot_title = new_title
                    xlabel, ylabel = (None, None)
                    user_xlabel, user_ylabel = (self.pxtitle.get(), self.pytitle.get())
                    if user_xlabel:
                        xlabel = str(user_xlabel)
                        self.plot_xlabel = xlabel
                    if user_ylabel:
                        ylabel = str(user_ylabel)
                        self.plot_ylabel = ylabel
def reset_plot_labels(self):
    value = self.change_plot_label_current
    newvalues = {'shown_title': self.shown_title_old,
                 'xlabel': None, 'ylabel': None}
    self.plot_rdy[value][1]['title'].update(
        newvalues)
    self.ptitle.delete(0, END)
    self.pxtitle.insert(END, newvalues['shown_title'])
}

self.pxtitle.delete(0, END)
self.pytitle.delete(0, END)
self.plot_title = None
self.plot_xlabel = None
self.plot_ylabel = None

def get_plot_titles(self, event):
    w = event.widget
    index = w.curselection()
    parent = self.pageone_listbox_plot
    if len(index) == 1:
        value = parent.get(index)
        element = self.plot_rdy[value]
        title_elements = element[1]['title']
        shown_title = title_elements['shown_title']
        self.shown_title_old = shown_title

        self.ptitle.delete(0, END)
        self.pxtitle.delete(0, END)
        self.pytitle.delete(0, END)
        self.ptitle.insert(END, shown_title)

    def get_fontsize(self, event):
        w = event.widget
        self.fontsize = int(w.get())
        self.label_font['text'] = 'Font size: ' + str(self.fontsize)

    def set_gelmod(self):
        na_start = int(self.comp1.get('1.0',END))
        na_mid = int(self.comp1mid.get('1.0',END))
        na_end = int(self.comp1end.get('1.0',END))
        ca_start = int(self.comp2.get('1.0',END))
        ca_mid = int(self.comp2mid.get('1.0', END))
        ca_end = int(self.comp2end.get('1.0', END))
temp_start = int(self.comp3.get('1.0', END))
temp_mid = int(self.comp3mid.get('1.0', END))
temp_end = int(self.comp3end.get('1.0', END))

self.comp_na_start = na_start
self.comp_na_mid = na_mid
self.comp_na_end = na_end
self.comp_ca_start = ca_start
self.comp_ca_mid = ca_mid
self.comp_ca_end = ca_end
self.comp_temp_start = temp_start
self.comp_temp_mid = temp_mid
self.comp_temp_end = temp_end

self.comp_na = list(range(na_start, na_end+1, na_mid))
self.comp_ca = list(range(ca_start, ca_end+1, ca_mid))
self.comp_temp = list(range(temp_start, temp_end+1, temp_mid))

def reset_gelmod(self):
    self.comp1.delete('1.0', END)
    self.comp1mid.delete('1.0', END)
    self.comp1end.delete('1.0', END)
    self.comp2.delete('1.0', END)
    self.comp2mid.delete('1.0', END)
    self.comp2end.delete('1.0', END)
    self.comp3.delete('1.0', END)
    self.comp3mid.delete('1.0', END)
    self.comp3end.delete('1.0', END)
    self.comp1.insert(END, self.comp_na_start)
    self.comp1mid.insert(END, self.comp_na_mid)
    self.comp1end.insert(END, self.comp_na_end)
    self.comp2.insert(END, self.comp_ca_start)
    self.comp2mid.insert(END, self.comp_ca_mid)
    self.comp2end.insert(END, self.comp_ca_end)
    self.comp3.insert(END, self.comp_temp_start)
    self.comp3mid.insert(END, self.comp_temp_mid)
    self.comp3end.insert(END, self.comp_temp_end)

def plot_gelmod(self):
    fontsize = self.fontsize
    matplotlib.rcParams.update({'font.size':
self.delete_figures(2)

share_axis, filename, aspect_wanted, aspect_auto = (False, '', 1, True)

chosen_rows, chosen_cols = (1, 1)

sharex_local, sharey_local = (False, False)

fig, axes = (None, None)

change_plot = self.plottype.get()  # Allow user
to change this

plot_version = None

if change_plot == 1:
    plot_version = 1

    fig, axes = plt.subplots(
        nrows=chosen_rows, ncols=chosen_cols,
        sharex=sharex_local, sharey=sharey_local, figsize=(10, 10))

elif change_plot == 0:

    plot_version = 0

    fig = Figure(figsize=(10, 10))


alpha_values = [2.000, 0.001, 0.017]

beta_values = [1.0, 0.9]

yield_values = [1.0, 0.0, 0.0]

rg, eaq, tref = [math.pow(10, -4), 77, 20]

crit, surface_area = [0.20, 200]

rvalue = 0.008314  # kj / K mol

tref_kelvin = float(tref+273.15)

conc_si = 10

inner_factor_tref = float(eaq/(rvalue* 
    tref_kelvin))

effect_of_si = math.pow(conc_si, alpha_values[0])

    effect_of_tref = math.exp(inner_factor_tref)

xvalues_gelmod = []
yvalues_gelmod = self.comp_temp
zvalues_gelmod = []

lvalue, jvalue, kvalue = [1]*3

kvalues_check = []

combined_check = []

combined_check_values = []

for i_na in self.comp_na:
    for j_ca in self.comp_ca:
        inner_factor_na = math.pow(i_na,
beta_values[0])

na_exponent = alpha_values[1] *
inner_factor_na

effect_of_na = math.exp(na_exponent)
inner_factor_ca = math.pow(j_ca, beta_values[1])

c_exponent = alpha_values[2] *
inner_factor_ca

effect_of_ca = math.exp(c_exponent)

# xvalue = float(effect_of_na/(effect_of_na+effect_of_ca))
xvalue = float(effect_of_na/effect_of_ca)

# inner_xvalue = (1/(i_na+1)) + (1/(j_ca +1)) + (1/(1+(i_na*j_ca)))
inner_xvalue = (1 / (i_na + 1)) + (1 / (j_ca + 1))

# xvalue = math.pow(inner_xvalue, -1)
xvalues_gelmod.append(xvalue)

for k_temp in self.comp_temp:
  temp_kelvin = float(k_temp+273.15)
  inner_factor_temp = -(eag/(rvalue*
    temp_kelvin))
  effect_of_temp = math.exp(
    inner_factor_temp)

  zvalue = float(rg*effect_of_si*
    effect_of_na*effect_of_ca*effect_of_tref*effect_of_temp)
  zvalues_gelmod.append(zvalue)
  kvalues_check.append(kvalue)
  combined_check.append([ivalue, jvalue
    ,kvalue])
  combined_check_values.append([int(
    i_na),int(j_ca),xvalue])

  kvalue += 1
  jvalue += 1
  print('ivalue: ' + str(ivalue) + ' jvalue: ' + str(jvalue))

  ivalue += 1
  print(np.array(combined_check))
  print(np.array(combined_check_values))

xi, yj = (xvalues_gelmod, yvalues_gelmod)
dx, dy = ([1.0]*len(xi), [self.comp_temp_mid]*
  len(yj))
xbound = self.get_block_boundaries(cellvalues=xi, cellwidths=dx)
ybound = self.get_block_boundaries(cellvalues=yj, cellwidths=dy)

x_id_v2, y_id_v2, z_id_v2 = (xi, yj, zvalues_gelm)

# x_id_v2, y_id_v2, z_id_v2 = (xi, yj, zvalues_gelm)
xlength, ylength = (len(x_id_v2), len(y_id_v2))

transpose_choice, rowshape, colshape = (None, None, None)
rowshape, colshape = (xlength, ylength)
transpose_choice = 0

x_grid, y_grid = np.meshgrid(x_id_v2, y_id_v2)
z_grid = np.reshape(np.array(z_id_v2), (rowshape, colshape))

ax, prev_ax, im = (None, None, None)

if plot_version == 0:
    ax = fig.add_subplot(chosen_rows, chosen_cols, 1)
elif plot_version == 1:
    ax = axes

xlabel, ylabel, title = ('x', 'Temperature (°C)', 'Gelation rate')

if self.plot_title:
    title = self.plot_title

if self.plot_xlabel:
    xlabel = self.plot_xlabel

if self.plot_ylabel:
    ylabel = self.plot_ylabel

ax.set_title(title)
ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel)

ax.xaxis.set_tick_params(which='both', labelbottom=True)

ax.yaxis.set_tick_params(which='both', labelbottom=True)

ax.set_xscale('log')
# ax.set_yscale('log')
# norm = clr.Normalize()
# cmap = cm.get_cmap('gist_rainbow')
# ax.pcolormesh(x_grid, y_grid, z_grid, norm=matplotlib.colors.LogNorm(vmin=z_grid.min(), vmax=z_grid.max()), cmap='PuBu_r')
im = ax.pcolormesh(x_grid, y_grid, z_grid.T, norm=matplotlib.colors.LogNorm(), cmap='PuBu_r')
# im = ax.pcolor(x_grid, y_grid, z_grid, norm=matplotlib.colors.LogNorm(vmin=z_grid.min(), vmax=z_grid.max()), cmap='gist_rainbow')

# if transpose_choice == 0:
#     im = ax.pcolormesh(x_grid, y_grid, z_grid, cmap=cmap, norm=norm)
# elif transpose_choice == 1:
#     im = ax.pcolormesh(x_grid, y_grid, z_grid.T, cmap=cmap, norm=norm)

print('...')
print(': transposed?? ' + str(transpose_choice))
print('xlength: ' + str(len(x_id_v2)) + ' ylength: ' + str(len(y_id_v2)) + ' zlength: ' + str(len(z_id_v2)))

if aspect_auto is False:
    aspect_ratio_wanted = aspect_wanted
    aspect_ratio_correct = abs((x_max - x_min) / (y_max - y_min)) / aspect_ratio_wanted
    ax.set_aspect(aspect_ratio_correct)

fig.colorbar(im, ax=ax)
plt.tight_layout()
self.canvas = FigureCanvasTkAgg(fig, self.f2_plot)
sself.canvas.draw()
sself.canvas.pack(side=tk.TOP,
def set_xyz(self):
    child2 = self.prep_sim_parameters
    child2.selection_clear(0, END)
    if self.freezexl and self.freezeyl and self.freeezl and self.freezetimel:
        data = self.simcase_ijkl_count
        imin, imax, jmin, jmax, kmin, kmax, tmin, tmax = (data['imin'], data['imax'], data['jmin'], data['jmax'], data['kmin'], data['kmax'], data['tmin'], data['tmax'])
        current_xyz = self.xyz.get()
        int(current_xyz[0], int(current_xyz[1]), int(current_xyz[2]))
        self.slidex_left.set(int(current_xyz[0]))
        self.slidex_right.set(int(current_xyz[0]))
        self.slidey_left.set(int(current_xyz[1]))
        self.slidey_right.set(int(current_xyz[1]))
        self.slidez_left.set(int(current_xyz[2]))
        self.slidez_right.set(int(current_xyz[2]))
        self.slidetime_left.set(int(current_xyz[3]))
        self.slidetime_right.set(int(current_xyz[3]))
    if current_xyz == 9:
        self.xyz.set(0)
        self.slidex_left.set(int(current_xyz[0]))
        self.slidex_right.set(int(current_xyz[0]))
        self.slidey_left.set(int(current_xyz[1]))
        self.slidey_right.set(int(current_xyz[1]))
        self.slidez_left.set(int(current_xyz[2]))
        self.slidez_right.set(int(current_xyz[2]))
        self.slidetime_left.set(int(current_xyz[3]))
        self.slidetime_right.set(int(current_xyz[3]))
    elif current_xyz in [7, 8]:
        simcase = current_selection
        core_path = global_sim_data[simcase][0]
        childx = self.x_listbox
        childy = self.y_listbox
allsx = list(childx.get(0, END))
allsy = list(childy.get(0, END))

sliders = {'X': [self.slidex_left, self.slidex_right], 'Y': [self.slidey_left, self.slidey_right], 'Z': [self.slidex_left, self.slidex_right], 'time': [self.slidetime_left, self.slidetime_right]}
freezes = {'X': [self.freezex1, self.freezex2], 'Y': [self.freezey1, self.freezey2], 'Z': [self.freezex1, self.freezex2], 'time': [self.freezetime1, self.freezetime2], 'value_x1': self.value_x1, 'value_x2': self.value_x2, 'X': [self.value_x1, self.value_x2], 'Y': [self.value_y1, self.value_y2], 'Z': [self.value_z1, self.value_z2], 'time': [self.value_time1, self.value_time2], 'ijk_values': [iimin, iimax, imin, imax], 'Y': [jjmin, jjmax, jmin, jmax], 'Z': [kkmin, kkmax, kmin, kmax], 'time': [ttmin, ttmax, tmin, tmax]}

if ttmin == ttmax:
    self.slidetime_left.set(tmin)
    self.slidetime_right.set(tmax)
    self.valuetime.set(0)

openlist, currentlist = (None, None)
self.plot_id_old = self.plot_id
self.plot_id = {'simcase': simcase, 'simcase_path': core_path, 'simcase_child': ['DATA'], ['DATA'], 'entries': ['Days'], 'cells': None, 'time': None, 'X': None, 'Y': None}

choices = ['X', 'Y']
choice = None

if current_xyz == 7 and allsx: # Replace X with Days | Open fully Y
    openlist = self.plot_y[allsy[0]]['entries']
    currentlist = self.plot_x[allsx[0]]['entries']
    choice = choices[0]

elif current_xyz == 8 and allsy: # Replace Y with Days | Open fully X
    openlist = self.plot_x[allsx[0]]['entries']
    currentlist = self.plot_y[allsy[0]]['entries']
'entries'
      choice = choices[1]
      if openlist:
        if openlist[0] in ['X', 'Y', 'Z']
          and currentlist[0] in ['X', 'Y', 'Z']:
            self.add_to_xy(typedata=choice)
            closelist = ['X', 'Y', 'Z']
            closelist.pop(closelist.index(openlist[0]))
      for item in openlist:
        min00, max00 = (ijkvalues[item][2], ijkvalues[item][3])
        min01, max01 = (ijkvalues[item][0], ijkvalues[item][1])
        slide_l, slide_r,
        checkbutton = (sliders[item][0], sliders[item][1],
                       freezes[item][1])
        if min01 == max01:
          slide_l.set(min00)
          slide_r.set(max00)
          checkbutton.set(0)
      for item in closelist:
        min00, max00 = (ijkvalues[item][2], ijkvalues[item][3])
        min01, max01 = (ijkvalues[item][0], ijkvalues[item][1])
        if min01 != max01:
          freezetime, checkbutton,
          slide_l_label, slide_r_label, slide_l, slide_r = (
            freezes[item][0], freezes[item][1], freezes[item][2],
            freezes[item][3], sliders[item][0], freezes[item][4])
          freezetime.select()
          self.freeze_val(
            checkbutton, slide_l_label, slide_r_label, slide_r)
          (min00, max00 + 1))
          slide_l.set(rnd)
      elif currentlist[0] == 'Days':
        self.add_to_xy(typedata=choice)
        # New settings should be added automatically
        self.plot_id = self.plot_id_old
      else:
child = self.prep_sim_parameters
alls = self.prep_sim_parameters.get(0, END)
indexdata = alls.index('DATA')
access = (indexdata,)
sel = self.prep_sim_parameters.selection_set(access)
self.prep_sim_parameters.event_generate('<<ListboxSelect>>')

simcase = current_selection
core_path = global_sim_data[simcase][0]
self.valueX1.set(0)
self.valueY1.set(0)
self.valueZ1.set(0)
self.valueTime1.set(0)
if current_xyz == 1 or current_xyz == 2:
    # XY and YX
    self.freezeTime1.select()
    self.freezeVal_time(self.valueTime1, self.slidetime_label1, self.slidetime_label2, self.slidetime_right)
    self.freezeZ1.select()
    self.freezeVal(self.valueZ1, self.slidez_label1, self.slidez_label2, self.slidez_right)
    self.slideX_left.set(imin)
    self.slideX_right.set(imax)
    self.slideY_left.set(jmin)
    self.slideY_right.set(jmax)
    if kmin != kmax:
rnd_k = int(random.uniform(kmin, kmax+1))
    self.slideZ_left.set(rnd_k)
    if ttmin != ttmax:
rnd_t = int(random.uniform(ttmin, ttmax+1))
    self.slidetime_left.set(rnd_t)
if current_xyz == 1:
    self.plot_id_old = self.plot_id
    self.plot_id = {'simcase': simcase, 'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']], 'entries': ['X'], 'cells': None, '}
time': None, 'X': None, 'Y': None)
    self.add_to_xy(typedata='X')
    self.plot_id = {'simcase':
        simcase, 'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']}, 'entries': ['Y'], 'cells': None, '
    time': None, 'X': None, 'Y': None)
    self.add_to_xy(typedata='Y')
    self.plot_id = self.plot_id_old
    elif current_xyz == 2:
        self.plot_id = self.plot_id_id
        self.plot_id = {'simcase':
            simcase, 'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']}, 'entries': ['Y'], 'cells': None, '
            time': None, 'X': None, 'Y': None)
        self.add_to_xy(typedata='X')
        self.plot_id = {'simcase':
            simcase, 'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']}, 'entries': ['Y'], 'cells': None, '
            time': None, 'X': None, 'Y': None)
        self.add_to_xy(typedata='Y')
        self.plot_id = self.plot_id_old
        elif current_xyz == 3 or current_xyz ==
        4: # XZ and ZX
            self.frozetime1.select()
            self.freeze_val_time(self.valutime1
                , self.slidetime_label1, self.slidetime_label2, self.
                slidetime_right)
            self.freeze_y1.select()
            self.freeze_val(self.valuey1, self.
                slidey_label1, self.slidey_label2, self.slidey_right)
        self.slidex_left.set(imin)
        self.slidex_right.set(imax)
        self.slidex_left.set(kmin)
        self.slidex_right.set(kmax)
        if jjmin != jjmax:
            rnd_j = int(random.uniform(jmin,
            jmax + 1))
            self.slidey_left.set(rnd_j)
        if ttmin != ttmax:
            rnd_t = int(random.uniform(tmin,
            tmax + 1))
            self.slidetime_left.set(rnd_t)
        if current_xyz == 3:
self.plot_id_old = self.plot_id

self.plot_id = {'simcase':
    'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']], 'entries': [{'X': None, 'Y': None}]}

self.add_to_xy(typedata='X')
self.plot_id = {'simcase':
    'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']], 'entries': [{'Z': None, 'Y': None}]}

self.add_to_xy(typedata='Y')
self.plot_id = self.plot_id_old

elif current_xyz == 4:
    self.plot_id_old = self.plot_id
    self.plot_id = {'simcase':
        'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']], 'entries': [{'Z': None, 'Y': None}]}

self.add_to_xy(typedata='X')
self.plot_id = {'simcase':
    'simcase_path': core_path, 'simcase_child': ['INPUT', ['INPUT']], 'entries': [{'X': None, 'Y': None}]}

self.add_to_xy(typedata='Y')
self.plot_id = self.plot_id_old

elif current_xyz == 6:  # YZ and YZ
    self.freeze_time1.select()
    self.freeze_val_time(self.valuetime1, self.slide_time_label1, self.slide_time_label2, self.slide_time_right)
    self.freeze_x1.select()
    self.freeze_val(self.valuex1, self.slide_label1, self.slide_label2, self.slide_right)

self.slidez_left.set(emin)
self.slidez_right.set(imax)
self.slidey_left.set(jmin)
self.slidey_right.set(jmax)

if imin != imax:
    rnd_x = int(random.uniform(emin, imax + 1))

self.slidez_left.set(rnd_x)

if ttmin != ttmax:
    self.slide_x_left.set(rnd_t)

if current_xyz != 2:
    self.slidez_left.set(emin)
    self.slidez_right.set(imax)
    self.slidey_left.set(jmin)
    self.slidey_right.set(jmax)
tmax + 1))
self.slidetime_left.set(rnd_t)
if current_xyz == 5:
selplot_id = self.plot_id
self.plot_id = {'simcase':
    'simcase_path': core_path, 'simcase_child': [{'
        'INPUT', ['INPUT']}, 'entries': ['Z'], 'cells': None,
        'time': None, 'X': None, 'Y': None}
self.add_to_xy(typedata='X')
self.plot_id = {'simcase':
    'simcase_path': core_path, 'simcase_child': [{'
        'INPUT', ['INPUT']}, 'entries': ['Y'], 'cells': None,
        'time': None, 'X': None, 'Y': None}
self.add_to_xy(typedata='Y')
selplot_id = self.plot_id
elif current_xyz == 6:
selplot_id = self.plot_id
self.plot_id = {'simcase':
    'simcase_path': core_path, 'simcase_child': [{'
        'INPUT', ['INPUT']}, 'entries': ['Y'], 'cells': None,
        'time': None, 'X': None, 'Y': None}
self.add_to_xy(typedata='X')
self.plot_id = {'simcase':
    'simcase_path': core_path, 'simcase_child': [{'
        'INPUT', ['INPUT']}, 'entries': ['Z'], 'cells': None,
        'time': None, 'X': None, 'Y': None}
self.add_to_xy(typedata='Y')
selplot_id = self.plot_id
def fetch_data(self, path, value_returned,
    property_chosen, cells, time_element):
sorted_dataframe, sorted_dataframe_small,
sorted_dataframe_selection = (None, None, None)
    path_ar_input = os.path.join(path, 'INPUT' + '.
    parquet')
x_input = pd.read_parquet(path_ar_input)
ilist, jlist, klist, tlist = (time_element[2],
    time_element[3], time_element[4], time_element[5])
    newcells = x_input.loc[(x_input['i'].isin(jlist)
    ) & (x_input['j'].isin(jlist)) & (x_input['k'].isin(klist)), 'Cell'].tolist()
    allcells = x_input['Cell'].tolist()
filename = value_returned[0]

if filename == 'COMP':
    path_ar_comp = os.path.join(path, 'COMP' + '.parquet')
    x_comp = pd.read_parquet(path_ar_comp)
    pressure_chosen = value_returned[1]  # 240.0
    component_chosen = value_returned[2]  # Ca(\n
    sorted_dataframe = x_comp.loc[(x_comp['Component'] == component_chosen) & (x_comp['Pressure'] == pressure_chosen), :].loc[times]
    sorted_dataframe_small = sorted_dataframe[
        property_chosen]
    sorted_dataframe_selection = sorted_dataframe_small.tolist()

elif filename == 'REGION':
    path_ar_region = os.path.join(path, 'REGION' + '.parquet')
    x_region = pd.read_parquet(path_ar_region)
    region_chosen = value_returned[1]
    component_chosen = value_returned[2]
    sorted_dataframe = x_region.loc[(x_region['Component'] == component_chosen) & (x_region['Region'] == region_chosen), :].loc[times]
    sorted_dataframe_small = sorted_dataframe[
        property_chosen]
    sorted_dataframe_selection = sorted_dataframe_small.tolist()

elif filename == 'WELLS':
    path_ar_wells = os.path.join(path, 'WELLS' + '.parquet')
    x_wells = pd.read_parquet(path_ar_wells)
    well_chosen = value_returned[1]
    sorted_dataframe = x_wells.loc[x_wells['nWell'] == well_chosen, :].loc[times]
    sorted_dataframe_small = sorted_dataframe[
        property_chosen]
    sorted_dataframe_selection = sorted_dataframe_small.tolist()

elif filename == 'TIME':
    path_ar_time = os.path.join(path, 'TIME' + '.parquet')
x_time = pd.read_parquet(path_ar_time)
df_prop = x_time[property_chosen]
fetch_something = np.array(tlist) - 1
t_to_plot = df_prop.iloc(axis=0)[
    fetch_something].tolist()

return t_to_plot

elif filename == 'DATA':
    path_ar_data = os.path.join(path, 'DATA' + '.parquet')
    x_data = pd.read_parquet(path_ar_data)
    prop_final = []
df_prop = x_data[property_chosen]
maxcells = max(allcells)
cells_unique = np.array(newcells) - 1
timesteps = int(len(df_prop)/maxcells)
for timestep in list(range(1,timesteps+1)):
    top_pos = (timestep - 1)*maxcells
    bot_pos = timestep*maxcells
    current = df_prop[top_pos:bot_pos]
    if timestep in tlist:
        fetch_cells = cells_unique + (timestep-1)*maxcells
        prop_to_plot = df_prop.iloc(axis=0)[
            fetch_cells].tolist()
        prop_final = prop_final +
        prop_to_plot
    return prop_final

elif filename == 'INPUT':
    path_ar_input = os.path.join(path, 'INPUT' + '.parquet')
x_input = pd.read_parquet(path_ar_input)
ilist, jlist, klist, tlist = (time_element[2], time_element[3], time_element[4], time_element[5])
newcells = x_input.loc[(x_input['i'].isin(ilist)) & (x_input['j'].isin(jlist)) & (x_input['k'].isin(klist)), 'Cell'].tolist()
if property_chosen == 'Cell':
    return newcells
else:
    df_prop = x_input[property_chosen]
    fetch_cells = np.array(newcells) - 1
    prop_to_plot = df_prop.iloc(axis=0)[
        fetch_cells].tolist()
unique_values = np.unique(prop_to_plot)
return unique_values
return sorted_dataframe, sorted_dataframe_small, sorted_dataframe_selection

def get_block_boundaries(self, cellvalues, cellwidths):
    cellvalues, cellwidths = (np.array(cellvalues), np.array(cellwidths))
    left_boundaries = (cellvalues - 0.5 * cellwidths).tolist()
    right_boundaries = [cellvalues[-1] + 0.5 * cellwidths[-1]]
    boundaries = left_boundaries + right_boundaries
    return boundaries

def get_varying_block_boundaries(self, cellvalues):
    boundaries = []
    for item in cellvalues:
        left_boundary = item - item*0.5
        boundaries.append(left_boundary)
        if item == cellvalues[-1]:
            right_boundary = item + item*0.5
            boundaries.append(right_boundary)
    return boundaries

def get_block_centers(self, cellvalues, cellwidths):
    cellvalues, cellwidths = (np.array(cellvalues), np.array(cellwidths)).tolist()
    left_boundaries = (cellvalues - 0.5 * cellwidths).tolist()
    right_boundaries = [cellvalues[-1] + 0.5 * cellwidths[-1]]
    boundaries = left_boundaries + right_boundaries
    return boundaries

def get_time(self, core_path, property_chosen, tlist):
    path_ar_time = os.path.join(core_path, 'TIME' + '.parquet')
    x_time = pd.read_parquet(path_ar_time)
    if property_chosen == 'Days':
        property_chosen = 'nDays'
    df = x_time[property_chosen]
    fetch_days = np.array(tlist) - 1
    values = df.iloc(axis=0)[fetch_days]
    return values
def get_input(self, core_path, property_chosen, ilist, jlist, klist):
    path_ar_input = os.path.join(core_path, 'INPUT' + '.parquet')
    x_input = pd.read_parquet(path_ar_input)
    allcells = x_input['Cell'].tolist()
    ijkcells = x_input.loc[(x_input['i'].isin(ilist) & (x_input['j'].isin(jlist) & (x_input['k'].isin(klist))), 'Cell'].tolist()
    fetch_cells = np.array(ijkcells) - 1
    df = x_input[property_chosen]
    values = df.iloc(axis=0)[fetch_cells]
    return values

def get_data(self, core_path, property_chosen, allcells, newcells, ilist, jlist, klist, tlist):
    values = []
    if property_chosen in ['Days', 'Timestep']:
        df = self.get_time(core_path, property_chosen, tlist)
        values = df.values.tolist()
        return values
    elif property_chosen == 'Cell':
        df = self.get_input(core_path=core_path, property_chosen=property_chosen, ilist=ilist, jlist=jlist, klist=klist)
        values = df.values.tolist()
        return values
    elif property_chosen in ['X', 'Y', 'Z']:
        df = self.get_input(core_path=core_path, property_chosen=property_chosen, ilist=ilist, jlist=jlist, klist=klist)
        values = df.values.tolist()
        return values
    else:
        path_ar_data = os.path.join(core_path, 'DATA' + '.parquet')
        x_data = pd.read_parquet(path_ar_data)
        z_final = []
        df_z = x_data[property_chosen]
        z_id = df_z.tolist()
        maxcells = max(allcells)
cells_unique = np.array(newcells) - 1
timesteps = int(len(z_id) / maxcells)

for timestep in list(range(1, timesteps + 1)):
    top_pos = (timestep - 1) * maxcells
    bot_pos = timestep * maxcells
    current = z_id[top_pos:bot_pos]
    if timestep in tlist:
        fetch_cells = cells_unique + (timestep - 1) * maxcells
        z_to_plot = df_z.iloc(axis=0)[fetch_cells].tolist()
        z_final = z_final + z_to_plot
        values = z_final
    return values

def plot_graphv2(self):
    self.delete_figures(2)
    child, prev_simcase, local_dict, store_dict,
    chosen_parameters_rawtest = (self.pageone_listbox_plot,
        {}, {}, {}, [], [])
    alls = list(child.get(0, END))
    if self.hold.get() == 0 and self.xyz.get() != 0:
        simcase = current_selection
        corepath = global_sim_data[simcase][0]
        current_xyz = self.xyz.get()  # 1,2,3,etc
        xnow, ynow, simcase_child = (None, None, None)

        self.plot_id_old = self.plot_id
        xcoord = [[X], [Y], [X], [Z], [Z], [Y]]
        ycoord = [[Y], [X], [Z], [X], [Y], [Z]]
        if current_xyz in [7, 8]:
            childx = self.x_listbox
            childy = self.y_listbox
            allsx = childx.get(0, END)
            allsy = childy.get(0, END)
            xnow = self.plot_x[allsx[0]]['entries']
            ynow = self.plot_y[allsy[0]]['entries']
            simcase_child_01, simcase_child_02 = (None, None)
            if current_xyz == 7:  # Means X is days
                simcase_child_01 = ['DATA', ['DATA']]
        }
```python
simcase_child_02 = ['INPUT', ['INPUT']]

elif current_xyz == 8:  # Means Y is days
days

    simcase_child_01 = ['INPUT', ['INPUT']]
simcase_child_02 = ['DATA', ['DATA']]

    self.plot_id = {'simcase': simcase, 'simcase_path': corepath, 'simcase_child': simcase_child_01, 'entries': xnow, 'cells': None, 'time': None, 'X': None, 'Y': None}
    self.add_to_xy(typedata='X')
    self.plot_id = {'simcase': simcase, 'simcase_path': corepath, 'simcase_child': simcase_child_02, 'entries': ynow, 'cells': None, 'time': None, 'X': None, 'Y': None}
    self.add_to_xy(typedata='Y')

else:
    simcase_child = ['INPUT', ['INPUT']]
xnow, ynow = (xcoord[current_xyz - 1], ycoord[current_xyz - 1])

    self.plot_id = {'simcase': simcase, 'simcase_path': corepath, 'simcase_child': simcase_child, 'entries': xnow, 'cells': None, 'time': None, 'X': None, 'Y': None}
    self.add_to_xy(typedata='X')
    self.plot_id = {'simcase': simcase, 'simcase_path': corepath, 'simcase_child': simcase_child, 'entries': ynow, 'cells': None, 'time': None, 'X': None, 'Y': None}
    self.add_to_xy(typedata='Y')

    self.plot_id = self.plot_id_old

if not alls:
    plottings = list(self.plot_rdy.keys())
    if plottings:
        for prev_plotted_item in plottings:
            oldcontent = self.plot_rdy[prev_plotted_item][1]
            current_path = oldcontent['simcase_path']
cells, times = self.get_cell_time(corepath=current_path)
    newvalues = {'cells': cells, '}
```
time': times, 'X': self.plot_x, 'Y': self.plot_y
    self.plot_rdy[prev_plotted_item][1].update(newvalues)
    child.insert(END, prev_plotted_item)
    alls = list(child.get(0, END))
    self.figs = len(alls)
    self.grid_size_figures()
    self.label_figs['text'] = 'Figures: ' + str(self.figs)
    elif alls:
        for prev_plotted_item in alls:
            # print('prev_plotted_item: ' + str(prev_plotted_item))
            oldcontent = self.plot_rdy[prev_plotted_item][1]
            core_path2 = oldcontent['simcase_path']
            # print('oldcontent: ' + str(oldcontent))
            marker = self.plot_rdy[prev_plotted_item][2]
            # print('marker: ' + str(marker))
            if marker == 1:
                current_path = oldcontent['simcase_path']
                cells2, time_element2 = self.get_cell_time(corepath=core_path2)
                ilist2, jlist2, klist2, tlist2 =
                    (time_element2[2], time_element2[3], time_element2[4],
                    time_element2[5])
                timedays2 = self.get_time(core_path2, 'Days', tlist2).values.tolist()
                data2 = self.plot_rdy[prev_plotted_item][1]['title']
                old_basetitle = data2['shown_title'].split('=')[0]
                new_basetitle = old_basetitle + ' = ' + str(timedays2[0]) + ' days'
                newvalues = {'shown_title': new_basetitle, 'timedays': timedays2}
                self.plot_rdy[prev_plotted_item][1]['title'].update(newvalues)
            newvalues = {'cells': cells2, 'time': time_element2, 'X': self.plot_x, 'Y': self.plot_y}
```python
self.plot_rdy[prev_plotted_item][1].update(newvalues)

if marker == 0:
    current_path = oldcontent['simcase_path']
    cells2, time_element2 = self.get_cell_time(corepath=current_path)
    newvalues = { 'cells': cells2, 'X': self.plot_x, 'Y': self.plot_y }
    self.plot_rdy[prev_plotted_item][1].update(newvalues)

share_axis, filename, aspect_wanted, aspect_auto = (False, '', 1, True)
chosen_rows, chosen_cols = ([], [])
global chosen_rows_alt, chosen_cols_alt
if chosen_rows_alt is not None and
chosen_cols_alt is not None:
    chosen_rows, chosen_cols = (int(chosen_rows_alt), int(chosen_cols_alt))
    self.grid_button.config(text='Rows: ' +
                             chosen_rows_alt + ' Cols: ' + chosen_cols_alt)
    chosen_rows_alt, chosen_cols_alt = (None, None)
else:
    dimensions = self.fig_grid_size[self.grid_dropdown.get()]
    chosen_rows, chosen_cols = dimensions

tight_plot = True
sharex_local, sharey_local = (False, False)
fig, axes = (None, None)
change_plot = self.plottype.get()  # Allow user to change this

plot_version = None
if change_plot == 1:
    plot_version = 1
    if self.sharex.get() == 1:
        sharex_local = True
    if self.sharey.get() == 1:
        sharey_local = True
    fig, axes = plt.subplots(
        nrow=chosen_rows, ncol=chosen_cols,
        sharex=sharex_local, sharey=sharey_local, figsize=(10, 6)
```
elif change_plot == 0:
    plot_version = 0
if self.sharex.get() == 1:
    sharex_local = 'all'
if self.sharey.get() == 1:
    sharey_local = 'all'
fig = Figure(figsize=(10, 10))
fig_ploected = 0
prev_ax = None
for item in alls:
    xvalues, yvalues, zvalues = (None, None, None)
    raw = self.plot_rdy[item]
data, identifier = (raw[1], raw[0])
simcase_path = data['simcase_path']
titledata = data['title']
title = titledata['shown_title']
new_xlabel = titledata['xlabel']
new_ylabel = titledata['ylabel']
fontsize = int(data['fontsize'])
if self.hold3.get() == 0:
    fontsize = self.fontsize
    matplotlbr.rcParams.update({'font.size': fontsize})
cells, time_element = (data['cells'], data['time'])
if self.hold == 0:
cells, time_element = self.get_cell_time
corepath=simcase_path
times, days = (time_element[0].tolist(),
time_element[1].tolist())
    xvalues, yvalues = (None, None)
x_property_chosen, y_property_chosen = (None,
    None)
    x_path, y_path = (None, None)
test = None
xvalues_unique, yvalues_unique,
zvalues_unique = (None, None, None)
if data['X']:
x_key = list(data['X'].keys())[0]
x_key_prop = x_key.split(' ')[2]
x_coord_data = data['X'][x_key]
x_simcase_child = x_coord_data['simcase_child'][1]
x_property_chosen = x_coord_data['entries'][0]

x_path = x_coord_data['simcase_path']
x_simcase = x_coord_data['simcase']
if x_simcase_child == 'DATA' and x_key_prop == 'Cell':
x_simcase_child = ['INPUT']
x_filename = x_simcase_child[0]
xvalues_unique = self.fetch_data(path=x_path, value_returned=x_simcase_child, property_chosen=x_property_chosen, cells=cells, time_element=time_element)

if data['Y']:
y_key = list(data['Y'].keys())[0]
y_key_prop = y_key.split(' '')[2]
y_coord_data = data['Y'][y_key]
y_simcase_child = y_coord_data['simcase_child'][1]
if y_simcase_child == 'DATA' and y_key_prop == 'Cell':
y_simcase_child = ['TIME']
y_property_chosen = y_coord_data['entries'][0]
y_path = y_coord_data['simcase_path']
y_simcase = y_coord_data['simcase']
y_filename = y_simcase_child[0]
yvalues_unique = self.fetch_data(path=y_path, value_returned=y_simcase_child, property_chosen=y_property_chosen, cells=cells, time_element=time_element)

keys = list(data.keys())
simcase_child = data['simcase_child']
filename = simcase_child[1][0]
z_property_chosen = data['entries']
simcase = data['simcase']
xlabel = 'Cell Numbering (unique)'
if new_xlabel:
xlabel = new_xlabel
ylabel = 'Simulation runtime (days)'
if new_ylabel:
ylabel = new_ylabel

if filename == 'DATA':
z_property_chosen = prep_pageone[simcase][z_property_chosen]
zvalues_unique = self.fetch_data(path=simcase_path, value_returned=simcase_child, property_chosen=z_property_chosen,
                cells=cells, time_element=time_element)

if self.hold == 0:
    imin, imax = (int(self.slidex_left.get()), int(self.slidex_right.get()))
    jmin, jmax = (int(self.slidey_left.get()), int(self.slidey_right.get()))
    kmin, kmax = (int(self.slidez_left.get()), int(self.slidez_right.get()))
    tmin, tmax = (int(self.slidetime_left.get()), int(self.slidetime_right.get()))

    ilist = list(range(imin, imax + 1))
    jlist = list(range(jmin, jmax + 1))
    klist = list(range(kmin, kmax + 1))
    tlist = list(range(tmin, tmax + 1))

    path_ar_input = os.path.join(simcase_path, 'INPUT' + '.parquet')
    x_input = pd.read_parquet(path_ar_input)
    newcells = x_input.loc[(x_input['i'].isin(ilist)) & (x_input['j'].isin(jlist)) & (x_input['k'].isin(klist)), 'Cell'].tolist()
    allcells = x_input['Cell']
    dims = x_input.loc[x_input['Cell'].isin(newcells), ['DX', 'DY', 'DZ']]

    x_data = None
    x_id, y_id, z_id = (None, None, None)
    y_id_v2, y_id_v2_dates = ([], [])
    y_id_v3_days, y_id_v3_times = ([], [])
    for i in tlist:
        y_id_v2.append(i - 1)
2012    dimi, dimj, dimk = (None, None, None)
2013    xdim, ydim, zdim = (None, None, None)
2014    dim_dx, dim_dy, dim_dz = ([], [], [])
2015    grid_xticks, grid_yticks, grid_zticks = (None, None, None)
2016    rowshape, colshape = (None, None)
2017    rowtype, coltype = (None, None)
2018    x_grid, y_grid, z_grid = (None, None, None)
2019
2020    plot_type = None
2021    ilen, jlen, klen, tlen = (len(olist), len(jlist), len(klist), len(tlist))
2022    if ilen == 1 and jlen == 1 and klen == 1 and tlen >= 1:
2023        plot_type = 'time'
2024        # elif tlen==1 and ((ilen==1 and jlen==1 and
2025        # klen==1) or (ilen==1 and jlen!=1 and klen==1) or (ilen
2026        # ==1 and jlen==1 and klen!=1)):
2027        #    plot_type = 'position'
2028    else:
2029        plot_type = '2d'
2030    ax = None
2031    if filename == 'DATA' and plot_type == 'time':
2032        core_path = simcase_path
2033        y_id_v2 = self.get_data(core_path=core_path, property_chosen='Days', allcells=allcells, newcells=newcells, ilist=ibilist, jlist=jlist, klist=klist, tlist=tlist)
2034        z_id_v2 = self.get_data(core_path=core_path, property_chosen=z_property_chosen, allcells=allcells, newcells=newcells, ilist=ibilist, jlist=jlist, klist=klist, tlist=tlist)
2035        row_id, col_id = self.find_row_col(identifier, chosen_rows, chosen_cols)
2036        ax = None
2037        if plot_version == 0:
2038            if not prev_ax:
2039                ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)
2040            else:
2041                if sharex_local == 'all' and sharey_local != 'all':
2042                    ax = fig.add_subplot(
chosen_rows, chosen_cols, identifier, sharex=prev_ax)
    elif sharey_local != 'all' and sharey_local == 'all':
        ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharey=prev_ax)
    elif sharey_local == 'all':
        ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharex=prev_ax, sharey=prev_ax)
    else:
        ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)
    elif plot_version == 1:
        if (row_id, col_id) == (None, None):
            break
        if chosen_rows == 1 and chosen_cols == 1:  # Only plot one figure
            ax = axes
    elif chosen_rows == 1 and chosen_cols != 1:  # Only plot against col_id
        ax = axes[col_id]
        tight_plot = True
    elif chosen_rows != 1 and chosen_cols == 1:  # Only plot against row_id
        ax = axes[row_id]
    elif chosen_rows != 1 and chosen_cols != 1:  # Use both row_id and col_id
        ax = axes[row_id, col_id]
        title = z_property_chosen
        ax.set_title(title)
        xlabel, ylabel = (y_property_chosen, z_property_chosen)
    if new_xlabel:
        xlabel = new_xlabel
    if new_ylabel:
        ylabel = new_ylabel
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.plot(y_id_v2, z_id_v2)
    elif filename == 'DATA' and plot_type == 'position':
        core_path = simcase_path
    if ilen > 1:
    x_property_chosen = 'X'
    elif jlen > 1:
        x_property_chosen = 'Y'
    elif klen > 1:
        x_property_chosen = 'Z'
    dmax = self.get_input(core_path=
                            core_path, property_chosen=x_property_chosen,
                            ilist=ilist, jlist=jlist
        z_id_v2 = dmax.tolist()
        z_id_v2 = self.get_data(core_path=
                                core_path, property_chosen=z_property_chosen, allcells=
                                allcells, newcells=newcells, ilist=ilist, jlist=jlist,
                                klist=klist, tlist=tlist)
    row_id, col_id = self.find_row_col(
        identifier, chosen_rows, chosen_cols)
    ax = None
    if plot_version == 0:
        if not prev_ax:
            ax = fig.add_subplot(chosen_rows
                              , chosen_cols, identifier)
        else:
            if sharex_local == 'all' and
               sharey_local != 'all':
                ax = fig.add_subplot(
                               chosen_rows, chosen_cols, identifier, sharex=prev_ax)
            elif sharex_local == 'all' and
               sharey_local == 'all':
                ax = fig.add_subplot(
                               chosen_rows, chosen_cols, identifier, sharex=prev_ax)
            else:
                ax = fig.add_subplot(
                               chosen_rows, chosen_cols, identifier)
    elif plot_version == 1:
        if (row_id, col_id) == (None, None):
            break
        if chosen_rows == 1 and chosen_cols
        == 1:  # Only plot one figure
            ax = axes
elif chosen_rows == 1 and chosen_cols != 1: # Only plot against col_id
    ax = axes[col_id]
tight_plot = True
elif chosen_rows != 1 and chosen_cols == 1: # Only plot against row_id
    ax = axes[row_id]
elif chosen_rows != 1 and chosen_cols != 1: # Use both row_id and col_id
    ax = axes[row_id, col_id]
    ax.set_title(title)
xlabel, ylabel = (x_property_chosen, z_property_chosen)

if new_xlabel:
xlabel = new_xlabel
if new_ylabel:
ylabel = new_ylabel
ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel)
ax.plot(x_id_v2, z_id_v2)
elif filename == 'DATA' and plot_type == '2d' and ((x_property_chosen == 'Cell' and y_property_chosen == 'Days') or (x_property_chosen == 'Days' and y_property_chosen == 'Cell')):
core_path = simcase_path

x_id_v2_alt = self.get_data(core_path=core_path, property_chosen=x_property_chosen, allcells=allcells, newcells=newcells, ilist=ilist, jlist=jlist, klist=klist, tlist=tlist)
y_id_v2_alt = self.get_data(core_path=core_path, property_chosen=y_property_chosen, allcells=allcells, newcells=newcells, ilist=ilist, jlist=jlist, klist=klist, tlist=tlist)
z_id_v2_alt = self.get_data(core_path=core_path, property_chosen=z_property_chosen, allcells=allcells, newcells=newcells, ilist=ilist, jlist=jlist, klist=klist, tlist=tlist)

xi, yj = (x_id_v2_alt, y_id_v2_alt)
dxi, dyj = ([1.0]*len(xi), [1.0]*len(yj))

xnew = self.get_block_boundaries(cellvalues=xi, cellwidths=dxi)
2130         ynew = self.get_block_boundaries(
2131             cellvalues=yj, cellwidths=dyj)
2132         z_id_v2 = self.get_data(core_path=
2133             core_path, property_chosen=z_property_chosen, allcells=
2134             allcells, newcells=newcells,  ilist=ilist, jlist=jlist,
2135             klist=klist, tlist=tlist)
2136         x_centers = xi
2137         x_bound = xnew
2138         y_centers = yj
2139         y_bound = ynew
2140         x_id_v2, y_id_v2 = (x_id_v2_alt,
2141             y_id_v2, y_id_v2_alt)
2142         xlength, ylength = (len(x_id_v2), len(
2143             y_id_v2))
2144         transpose_choice = None
2145         if xlength >= ylength:
2146             rowshape, colshape = (ylength,
2147             xlength)
2148         else:
2149             transpose_choice = 0
2150         elif xlength < ylength:
2151             rowshape, colshape = (xlength,
2152             ylength)
2153         transpose_choice = 1
2154         x_grid, y_grid = np.meshgrid(x_id_v2,
2155             y_id_v2)
2156         z_grid = np.reshape(np.array(z_id_v2), (rowshape, colshape))
2157         fig_plotted, ax = (fig_plotted + 1, None)
2158         row_id, col_id = self.find_row_col(
2159             identifier, chosen_rows, chosen_cols)
2160         if plot_version == 0:
2161             if not prev_ax:
2162                 ax = fig.add_subplot(chosen_rows
2163                     , chosen_cols, identifier)
2164             else:
2165                 if sharex_local == 'all' and
2166                     sharey local != 'all':
2167                     ax = fig.add_subplot(chosen_rows
2168                         , chosen_cols, identifier, sharex=prev_ax)
elif sharey_local != 'all' and sharey_local == 'all':
    ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharey=prev_ax)
elif sharey_local == 'all':
    ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharey=prev_ax, sharey=prev_ax)
else:
    ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)
    if plot_version == 1:
        if (row_id, col_id) == (None, None):
            break
        if chosen_rows == 1 and chosen_cols == 1:  # Only plot one figure
            ax = axes
        elif chosen_rows == 1 and chosen_cols != 1:  # Only plot against col_id
            ax = axes[col_id]
            tight_plot = True
        elif chosen_rows != 1 and chosen_cols == 1:  # Only plot against row_id
            ax = axes[row_id]
        elif chosen_rows != 1 and chosen_cols != 1:
            ax = axes[row_id, col_id]
    ax.set_title(title)
    xlabel, ylabel = (x_property_chosen, y_property_chosen)
    if new_xlabel:
        xlabel = new_xlabel
    if new_ylabel:
        ylabel = new_ylabel
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.xaxis.set_tick_params(which='both', labelbottom=True)
    ax.yaxis.set_tick_params(which='both', labelbottom=True)
    norm = clr.Normalize()
    cmap = cm.get_cmap('gist_rainbow')
if transpose_choice == 0:
    im = ax.pcolormesh(x_grid, y_grid, 
    z_grid, cmap=cmap, norm=norm)
elif transpose_choice == 1:
    im = ax.pcolormesh(x_grid, y_grid, 
    z_grid.T, cmap=cmap, norm=norm)

print('...')
print(str(x_property_chosen) + str( 
    y_property_chosen) + ': transposed?? ' + str( 
    transpose_choice))
print('xlength: ' + str(len(x_id_v2)) + ' ylength: ' + str(len(y_id_v2)) + ' zlength: ' + str( 
    len(z_id_v2)))
print('(', rowshape, colshape, '): (' + str( 
    rowshape) + '[' + str(rowtype) + '], ' + str(colshape) + '][' + str(coltype) + '])'
print('xshape: ' + str(x_grid.shape) + ' yshape: ' + str(y_grid.shape) + ' zshape: ' + str( 
    z_grid.shape))

print('...')

if grid_xticks:
    ax.set_xticks(grid_xticks)
if grid_yticks:
    ax.set_yticks(grid_yticks)

if aspect_auto is False:
    aspect_ratio_wanted = aspect_wanted
    aspect_ratio_correct = abs((x_max - x_min) / (y_max - y_min)) / aspect_ratio_wanted
    ax.set_aspect(aspect_ratio_correct)

fig.colorbar(im, ax=ax)

if tight_plot is True and aspect_auto is 
    True and plot_version == 1:
    plt.tight_layout()
elif filename == 'DATA' and plot_type == '2d' 
    and ((x_property_chosen in ['X', 'Y', 'Z']) and 
    y_property_chosen in ['Days']) or (x_property_chosen in 
    ['Days'] and y_property_chosen in ['X', 'Y', 'Z']):
    core_path = simcase_path 
    datatypes = {'X': 'DX', 'Y': 'DY', 'Z': 'DZ'}
dimi, dimj, dims, xi, yj, dxi, dyj = (None, None, None, None, None, None, None)

x_id_v2_alt = self.get_data(core_path=core_path, property_chosen=x_property_chosen, allcells=allcells, newcells=newcells, ilist=ilist, jlist=jlist, klist=klist, tlist=tlist)

y_id_v2_alt = self.get_data(core_path=core_path, property_chosen=y_property_chosen, allcells=allcells, newcells=newcells, ilist=ilist, jlist=jlist, klist=klist, tlist=tlist)

z_id_v2_alt = self.get_data(core_path=core_path, property_chosen=z_property_chosen, allcells=allcells, newcells=newcells, ilist=ilist, jlist=jlist, klist=klist, tlist=tlist)

xi, yj = (x_id_v2_alt, y_id_v2_alt)

if x_property_chosen in ['Days']:
    dimj = datatypes[y_property_chosen]
dims = self.get_input(core_path=core_path, property_chosen=[dimj], ilist=ilist, jlist=jlist, klist=klist)
dxi, dyj = ([1.0] * len(xi), list(dims[dimj]))

elif y_property_chosen in ['Days']:
    dimi = datatypes[x_property_chosen]
dims = self.get_input(core_path=core_path, property_chosen=[dimi], ilist=ilist, jlist=jlist, klist=klist)
dxi, dyj = (list(dims[dimi]), [1.0] * len(yj))

xnew = self.get_block_boundaries(cellvalues=xi, cellwidths=dxi)
ynew = self.get_block_boundaries(cellvalues=yj, cellwidths=dyj)

x_centers, x_bound, y_centers, y_bound = (None, None, None, None)

if x_property_chosen in ['Days']:
    x_centers = xi
    x_bound = xnew
    y_centers = (np.unique(yj)).tolist()
    y_bound = (np.unique(ynew)).tolist()
elif y_property_chosen in ['Days']:
    x_centers = (np.unique(xi)).tolist()
    x_bound = (np.unique(xnew)).tolist()
    y_centers = yj
    y_bound = ynew

    print('x_centers len: ' + str(len(x_centers)) + ' x_centers: ' + str(x_centers))
    print('y_centers len: ' + str(len(y_centers)) + ' y_centers: ' + str(y_centers))
    print('x_bound len: ' + str(len(x_bound)) + ' x_bound: ' + str(x_bound))
    print('y_bound len: ' + str(len(y_bound)) + ' y_bound: ' + str(y_bound))

    x_id_v2, y_id_v2, z_id_v2 = (x_bound, y_bound, z_id_v2_alt)
    xlength, ylength = (len(x_id_v2), len(y_id_v2))

    transpose_choice = None
    if xlength > ylength:
        rowshape, colshape = (ylength - 1, xlength - 1)

    transpose_choice = 0
    elif xlength < ylength:
        rowshape, colshape = (xlength - 1, ylength - 1)

    transpose_choice = 1
    elif xlength == ylength:
        if (x_property_chosen == 'X' and y_property_chosen == 'Days') or (x_property_chosen == 'Y' and y_property_chosen == 'Days') or (x_property_chosen == 'Z' and y_property_chosen == 'Days'):
            rowshape, colshape = (ylength - 1, xlength - 1)

        transpose_choice = 0
    elif (x_property_chosen == 'Days' and y_property_chosen == 'X') or (x_property_chosen == 'Days' and y_property_chosen == 'Y') or (x_property_chosen == 'Days' and y_property_chosen == 'Z'):
        rowshape, colshape = (xlength - 1, ylength - 1)

    transpose_choice = 1
x_grid, y_grid = np.meshgrid(x_id_v2, y_id_v2)
z_grid = np.reshape(np.array(z_id_v2), (rowshape, colshape))

fig_plotted, ax = (fig_plotted + 1, None)
row_id, col_id = self.find_row_col(identifier, chosen_rows, chosen_cols)
title = z_property_chosen
if plot_version == 0:
    subplot_id = str(chosen_rows) + str(chosen_cols) + str(identifier)
    if not prev_ax:
        ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)
    else:
        if sharex_local == 'all' and sharey_local != 'all':
            ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharex=prev_ax)
        elif sharex_local != 'all' and sharey_local == 'all':
            ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharey=prev_ax)
        elif sharex_local == 'all' and sharey_local != 'all':
            ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharex=prev_ax, sharey=prev_ax)
        else:
            ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)
    elif plot_version == 1:
        if (row_id, col_id) == (None, None):
            break
        if chosen_rows == 1 and chosen_cols == 1:  # Only plot one figure
            ax = axes
        elif chosen_rows == 1 and chosen_cols != 1:  # Only plot against col_id
            ax = axes[col_id]
        tight_plot = True
    elif chosen_rows != 1 and chosen_cols == 1:  # Only plot against row_id
ax = axes[row_id]

elif chosen_rows != 1 and chosen_cols != 1:
    # Use both row_id and col_id
    ax = axes[row_id, col_id]

ax.set_title(title)
xlabel, ylabel = (x_property_chosen, y_property_chosen)

if new_xlabel:
xlabel = new_xlabel

if new_ylabel:
ylabel = new_ylabel

ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel)
ax.xaxis.set_tick_params(which='both', labelbottom=True)
ax.yaxis.set_tick_params(which='both', labelbottom=True)

norm = clr.Normalize()
cmap = cm.get_cmap('gist_rainbow')

if transpose_choice == 0:
im = ax.pcolormesh(x_grid, y_grid, z_grid, cmap=cmap, norm=norm)

elif transpose_choice == 1:
im = ax.pcolormesh(x_grid, y_grid, z_grid.T, cmap=cmap, norm=norm)

print('...')
print(str(x_property_chosen) + str(y_property_chosen) + ': ' + str(transpose_choice))
print('x_length: ' + str(len(x_id_v2)) + ' y_length: ' + str(len(y_id_v2)) + ' z_length: ' + str(len(z_id_v2)))

print('rowshape, colshape): (' + str(rowshape) + '[' + str(rowtype) + '], ' + str(colshape) + '[' + str(coltype) + ']')

print('xshape: ' + str(x_grid.shape) + ' yshape: ' + str(y_grid.shape) + ' zshape: ' + str(z_grid.shape))

print('...')

if grid_xticks:
    ax.set_xticks(grid_xticks)
if grid_yticks:
    ax.set_yticks(grid_yticks)

if aspect_auto is False:
    aspect_ratio_wanted = aspect_wanted
    aspect_ratio_correct = abs((x_max - x_min) / (y_max - y_min)) / aspect_ratio_wanted
    ax.set_aspect(aspect_ratio_correct)

fig.colorbar(im, ax=ax)
plt.tight_layout()

# if tight_plot is True and aspect_auto is True and plot_version == 1:
    # plt.tight_layout()

elif filename == 'DATA' and plot_type == '2d' and x_property_chosen in ['X', 'Y', 'Z'] and y_property_chosen in ['X', 'Y', 'Z']:
    core_path = simcase_path
    datatypes = {'X': 'DX', 'Y': 'DY', 'Z': 'DZ'}
    dimi, dimj = (datatypes[x_property_chosen], datatypes[y_property_chosen])
    dims = self.get_input(core_path=core_path, property_chosen=[x_property_chosen, y_property_chosen], dimi, dimj),
    ilist=ilist, jlist=jlist, klist=klist)
    xi, yj, dx, dyj = (list(dims[x_property_chosen]), list(dims[y_property_chosen]), list(dims[dimi]), list(dims[dimj]))

xnew = self.get_block_boundaries( cellvalues=xi, cellwidths=dx)
ynew = self.get_block_boundaries( cellvalues=yj, cellwidths=dy)

z_id_v2 = self.get_data(core_path=core_path, property_chosen=z_property_chosen, allcells=allcells, newcells=newcells, ilist=ilist, jlist=jlist, klist=klist, tlist=tlist)

x_centers = (np.unique(xi)).tolist()
x_bound = (np.unique(xnew)).tolist()
y_centers = (np.unique(yj)).tolist()
y_bound = (np.unique(ynew)).tolist()

x_id_v2, y_id_v2 = (x_bound, y_bound)
xlength, ylength = (len(x_id_v2), len(y_id_v2))

transpose_choice = None

if xlength > ylength:
    rowshape, colshape = (ylength - 1, xlength - 1)
    transpose_choice = 0
elif xlength < ylength:
    rowshape, colshape = (xlength - 1, ylength - 1)
    transpose_choice = 1
elif xlength == ylength:
    rowshape, colshape = (ylength - 1, xlength - 1)
    transpose_choice = 0
elif (x_property_chosen == 'X' and y_property_chosen == 'Y') or (x_property_chosen == 'X' and y_property_chosen == 'Z') or (x_property_chosen == 'Y' and y_property_chosen == 'Z'):
    rowshape, colshape = (xlength - 1, ylength - 1)
    transpose_choice = 1
elif (x_property_chosen == 'Y' and y_property_chosen == 'X') or (x_property_chosen == 'Z' and y_property_chosen == 'X') or (x_property_chosen == 'Z' and y_property_chosen == 'Y'):
    rowshape, colshape = (xlength - 1, ylength - 1)
    transpose_choice = 1

x_grid, y_grid = np.meshgrid(x_id_v2, y_id_v2)
z_grid = np.reshape(np.array(z_id_v2), (rowshape, colshape))

fig_plotted, ax = (fig_plotted + 1, None)

row_id, col_id = self.find_row_col(identifier, chosen_rows, chosen_cols)

if plot_version == 0:
    subplot_id = str(chosen_rows) + str(chosen_cols) + str(identifier)
    if not prev_ax:
        ax = fig.add_subplot(chosen_rows
else:
    if sharex_local == 'all' and sharey_local != 'all':
        ax = fig.add_subplot(
            chosen_rows, chosen_cols, identifier, sharex=prev_ax)
    elif sharex_local == 'all' and sharey_local == 'all':
        ax = fig.add_subplot(
            chosen_rows, chosen_cols, identifier, sharex=prev_ax, sharey=prev_ax)
    else:
        ax = fig.add_subplot(
            chosen_rows, chosen_cols, identifier)
    elif plot_version == 1:
        if (row_id, col_id) == (None, None):
            break
        if chosen_rows == 1 and chosen_cols
            == 1: # Only plot one figure
            ax = axes
        elif chosen_rows == 1 and chosen_cols != 1: # Only plot against col_id
            ax = axes[col_id]
        elif chosen_rows != 1 and chosen_cols == 1: # Only plot against row_id
            ax = axes[row_id]
        elif chosen_rows != 1 and chosen_cols != 1: # Use both row_id and col_id
            ax = axes[row_id, col_id]
        ax.set_title(title)
        xlabel, ylabel = (x_property_chosen,
        y_property_chosen)
    if new_xlabel:
        xlabel = new_xlabel
    if new_ylabel:
        ylabel = new_ylabel
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.xaxis.set_tick_params(which='both',
labelbottom=True)
    ax.yaxis.set_tick_params(which='both', labelbottom=True)
    norm = clr.Normalize()
    cmap = cm.get_cmap('gist_rainbow')
    if transpose_choice == 0:
        im = ax.pcolormesh(x_grid, y_grid, z_grid, cmap=cmap, norm=norm)
    elif transpose_choice == 1:
        im = ax.pcolormesh(x_grid, y_grid, z_grid.T, cmap=cmap, norm=norm)
    print('...
    print(str(x_property_chosen) + str(y_property_chosen) + ': transposed?? ' + str(transpose_choice))
    print('xlength: ' + str(len(x_id_v2)) + ' ylength: ' + str(len(y_id_v2)) + ' zlength: ' + str(len(z_id_v2)))
    print('(rowshape, colshape): (' + str(rowshape) + ' [' + str(rowtype) + '], ' + str(colshape) + ' [' + str(coltype) + ']])
    print('xshape: ' + str(x_grid.shape) + ' yshape: ' + str(y_grid.shape) + ' zshape: ' + str(z_grid.shape))
    print('...
    if grid_xticks:
        ax.set_xticks(grid_xticks)
    if grid_yticks:
        ax.set_yticks(grid_yticks)
    if aspect_auto is False:
        aspect_ratio_wanted = aspect_wanted
        aspect_ratio_correct = abs((x_max - x_min) / (y_max - y_min)) / aspect_ratio_wanted
        ax.set_aspect(aspect_ratio_correct)
    fig.colorbar(im, ax=ax)
    plt.tight_layout()
    #if tight_plot is True and aspect_auto is True and plot_version == 1:
elif filename == 'INPUT' and plot_type == '2d' and x_property_chosen in ['X', 'Y', 'Z'] and y_property_chosen in ['X', 'Y', 'Z']:
    core_path = simcase_path
    datatypes = {'X': 'DX', 'Y': 'DY', 'Z': 'DZ'}
    dimi, dimj = (datatypes[x_property_chosen], datatypes[y_property_chosen])
    dms = self.get_input(core_path=core_path, property_chosen=[x_property_chosen, y_property_chosen, z_property_chosen, dimi, dimj],
                          ilist=ilist, jlist=jlist, klist=klist)
    xi, yj, dxj, dyj = (list(dms[x_property_chosen]), list(dms[y_property_chosen]), list(dms[1]), list(dms[2]))
    listitems = [xi, yj]
    for litem in listitems:
        litem = [float(i) for i in litem]
        xnew = self.get_block_boundaries(cellvalues=xi, cellwidths=dxj)
        ynew = self.get_block_boundaries(cellvalues=yj, cellwidths=dyj)
        z_id_v2 = [float(i) for i in litem(dims[z_property_chosen])]
        x_centers = list(np.unique(xi))
        x_bound = list(np.unique(xnew))
        y_centers = list(np.unique(yj))
        y_bound = list(np.unique(ynew))
        x_id_v2, y_id_v2 = (x_bound, y_bound)
xlength, ylength = (len(x_id_v2), len(y_id_v2))
        transpose_choice = None
        if xlength >= ylength:
            rowshape, colshape = (ylength - 1, xlength - 1)
            transpose_choice = 0
        elif xlength < ylength:
            rowshape, colshape = (xlength - 1, ylength - 1)
            transpose_choice = 1
x_grid, y_grid = np.meshgrid(x_id_v2, y_id_v2)
z_grid = np.reshape(np.array(z_id_v2), (rowshape, colshape))

fig_plotted, ax = (fig_plotted + 1, None)
row_id, col_id = self.find_row_col(identifier, chosen_rows, chosen_cols)
ax = None
if plot_version == 0:
    if not prev_ax:
        ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)
    else:
        if sharex_local == 'all' and sharey_local != 'all':
            ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharex=prev_ax)
        elif sharex_local != 'all' and sharey_local == 'all':
            ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharey=prev_ax)
        elif sharex_local == 'all' and sharey_local == 'all':
            ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharex=prev_ax, sharey=prev_ax)
    else:
        ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)

elif plot_version == 1:
    if (row_id, col_id) == (None, None):
        break
    if chosen_rows == 1 and chosen_cols == 1:  # Only plot one figure
        ax = axes
    elif chosen_rows == 1 and chosen_cols != 1:  # Only plot against col_id
        ax = axes[col_id]
    elif chosen_rows != 1 and chosen_cols == 1:  # Only plot against row_id
        ax = axes[row_id]


```python
elif chosen_rows != 1 and
    chosen_cols != 1:  # Use both row_id and col_id
        ax = axes[row_id, col_id]

    ax.set_title(title)
    xlabel, ylabel = (x_property_chosen, y_property_chosen)
    if new_xlabel:
        xlabel = new_xlabel
    if new_ylabel:
        ylabel = new_ylabel
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.xaxis.set_tick_params(which='both', labelbottom=True)
    ax.yaxis.set_tick_params(which='both', labelbottom=True)

    norm = clr.Normalize()
    cmap = cm.get_cmap('gist_rainbow')

    print('

    print('...')
    print(str(x_property_chosen) + str(
        y_property_chosen) + ': transposed?? ' + str(
        transpose_choice))
    print('xlength: ' + str(len(x_id_v2)) + ' ylength: ' + str(len(y_id_v2)) + ' zlength: ' + str(
        len(z_id_v2)))
    print('rowshape, colshape: (' + str(
        rowshape) + '[' + str(rowtype) + '], ' + str(colshape) + '[' + str(coltype) + '])
    print('xshape: ' + str(x_grid.shape) + ' yshape: ' + str(y_grid.shape) + ' zshape: ' + str(
        z_grid.shape))

    print('...')

    if transpose_choice == 0:
        im = ax.pcolormesh(x_grid, y_grid, z_grid, cmap=cmap, norm=norm)
    elif transpose_choice == 1:
        im = ax.pcolormesh(x_grid, y_grid, z_grid.T, cmap=cmap, norm=norm)

    print('...')
    print(str(x_property_chosen) + str(
        y_property_chosen) + ': transposed?? ' + str(
```
transpose_choice))
    print('xlength: ' + str(len(x_id_v2)) + ' ylength: ' + str(len(y_id_v2)) + ' zlength: ' + str(len(z_id_v2)))
    print('(' + str(rowshape) + '[' + str(rowtype) + ']' + str(colshape) + '[' + str(coltype) + '])')
    print('xshape: ' + str(x_grid.shape) + ' yshape: ' + str(y_grid.shape) + ' zshape: ' + str(z_grid.shape))
    print('...')

    if grid_xticks:
        ax.set_xticks(grid_xticks)
    if grid_yticks:
        ax.set_yticks(grid_yticks)

    if aspect_auto is False:
        aspect_ratio_wanted = aspect_wanted
        aspect_ratio_correct = abs((x_max - x_min) / (y_max - y_min)) / aspect_ratio_wanted
        ax.set_aspect(aspect_ratio_correct)

    fig.colorbar(im, ax=ax)

    plt.tight_layout()

    #if tight_plot is True and aspect_auto is True and plot_version == 1:
    #    plt.tight_layout()
    elif filename == 'COMP':
        core_path = simcase_path
        path_ar_comp = os.path.join(core_path, 'COMP' + '.parquet')
        path_ar_time = os.path.join(core_path, 'TIME' + '.parquet')
        x_comp = pd.read_parquet(path_ar_comp)
        x_time = pd.read_parquet(path_ar_time)
        pressure = [simcase_child[1][1]]
        component = [simcase_child[1][2]]
        compbase = x_comp.loc[(x_comp.index.isin(time_element[0])) & (x_comp['Component'].isin(component)) & (x_comp['Pressure'].isin(pressure))]
        z_property_chosen = timebase = compbase.index.tolist()
timesnew = []
for timeitem in timebase:
currenttime = datetime.datetime.
strptime(str(timeitem), "%Y-%m-%d %H:%M:%S")
daysnow = currenttime.timetuple().
tm_yday

timesnew.append(daysnow)

row_id, col_id = self.find_row_col(
identifier, chosen_rows, chosen_cols)

ax = None

if plot_version == 0:
    if not prev_ax:
        ax = fig.add_subplot(chosen_rows
', chosen_cols, identifier)
else:
    if sharex_local == 'all' and
sharey_local != 'all':
        ax = fig.add_subplot(
chosen_rows, chosen_cols, identifier, sharex=prev_ax)

elif sharex_local == 'all' and
sharey_local == 'all':
    ax = fig.add_subplot(
chosen_rows, chosen_cols, identifier, sharey=prev_ax)

elif sharex_local == 'all' and
sharey_local == 'all':
    ax = fig.add_subplot(
chosen_rows, chosen_cols, identifier, sharex=prev_ax,
sharey=prev_ax)

else:
    ax = fig.add_subplot(
chosen_rows, chosen_cols, identifier)

elif plot_version == 1:
    if (row_id, col_id) == (None, None):
        break

if chosen_rows == 1 and chosen_cols
== 1:  # Only plot one figure
    ax = axes

elif chosen_rows == 1 and
chosen_cols != 1:  # Only plot against col_id
    ax = axes[col_id]

    tight_plot = True

elif chosen_rows != 1 and
chosen_cols == 1:  # Only plot against row_id
    ax = axes[row_id]
elif chosen_rows != 1 and chosen_cols != 1:  # Use both row_id and col_id
    ax = axes[row_id, col_id]
    ax.set_title(title)
    xlabel, ylabel = ('Days',
                     z_property_chosen)
    if new_xlabel:
        xlabel = new_xlabel
    if new_ylabel:
        ylabel = new_ylabel
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.plot(timesnew, compbase)

    elif filename == 'REGION':
        core_path = simcase_path
        path_ar_reg = os.path.join(core_path, 'REGION' + '.parquet')
        path_ar_time = os.path.join(core_path, 'TIME' + '.parquet')
        x_reg = pd.read_parquet(path_ar_reg)
        x_time = pd.read_parquet(path_ar_time)

        region = [simcase_child[1][1]]
        component = [simcase_child[1][2]]
        regbase = x_reg.loc[(x_reg.index.isin(time_element[0])) & (x_reg['Component'].isin(component))
                             & (x_reg['Region'].isin(region)), z_property_chosen]
        timebase = regbase.index.tolist()
        timesnew = []
        for timeitem in timebase:
            currenttime = datetime.datetime.strptime(str(timeitem), "%Y-%m-%d %H:%M:%S")
            daysnow = currenttime.timetuple().tm_yday
            timesnew.append(daysnow)

        row_id, col_id = self.find_row_col(
            identifier, chosen_rows, chosen_cols)
        ax = None
        if plot_version == 0:
            if not prev_ax:
                ax = fig.add_subplot(chosen_rows
                                      , chosen_cols, identifier)
        else:
            if sharex_local == 'all' and
sharey_local != 'all':
    ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharex=prev_ax)
elif sharey_local != 'all' and sharey_local == 'all':
    ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharey=prev_ax)
elif sharey_local == 'all':
    ax = fig.add_subplot(chosen_rows, chosen_cols, identifier, sharex=prev_ax, sharey=prev_ax)
else:
    ax = fig.add_subplot(chosen_rows, chosen_cols, identifier)
elif plot_version == 1:
    if (row_id, col_id) == (None, None):
        break
elif chosen_rows == 1 and chosen_cols
    == 1:  # Only plot one figure
    ax = axes
elif chosen_rows == 1 and chosen_cols != 1:  # Only plot against col_id
    ax = axes[col_id]
tight_plot = True
elif chosen_rows != 1 and chosen_cols == 1:  # Only plot against row_id
    ax = axes[row_id]
elif chosen_rows != 1 and chosen_cols != 1:  # Use both row_id and col_id
    ax = axes[row_id, col_id]
ax.set_title(title)
xlabel, ylable = ('Days',
z_property_chosen)
if new_xlabel:
    xlabel = new_xlabel
if new_ylabel:
    ylabel = new_ylabel
ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel)
ax.plot(timesnew, regbase)
elif filename == 'WELLS':
core_path = simcase_path
path_ar_wells = os.path.join(core_path, 'WELLS' + '.parquet')
path_ar_time = os.path.join(core_path, 'TIME' + '.parquet')
x_wells = pd.read_parquet(path_ar_wells)
x_time = pd.read_parquet(path_ar_time)
dates = time_element[0]
if len(tlist) == 1:
dates = x_time.index.tolist()
wellnumber = [simcase_child[1][1]]
wellname = [simcase_child[1][2]]
welltype = [simcase_child[1][3]]
connections = x_wells['Connection'].tolist()
newconnect = []
for connection in connections:
    if 'Total' in connection:
        newconnect.append(connection)
try:
    new_z = dict_paramv2[simcase][z_property_chosen]
except KeyError:
    new_z = z_property_chosen
wellbase = x_wells.loc[(x_wells.index.isin(dates)) & (x_wells['nWell'].isin(wellnumber)) & (x_wells['nWellName'].isin(wellname)) & (x_wells['nType'].isin(welltype)) & (x_wells['Connection'].isin(newconnect))], new_z
timebase = wellbase.index.tolist()
timesnew = []
for timeitem in timebase:
currenttime = datetime.datetime.strptime(str(timeitem), '%Y-%m-%d %H:%M:%S')
daysnow = currenttime.timetuple().tm_yday
timesnew.append(daysnow)

row_id, col_id = self.find_row_col(
    identifier, chosen_rows, chosen_cols)

ax = None
if plot_version == 0:
    if not prev_ax:
        ax = fig.add_subplot(chosen_rows
, chosen_cols, identifier)
else:
    if sharex_local == 'all' and
sharey_local != 'all':
    ax = fig.add_subplot(
        chosen_rows, chosen_cols, identifier, sharex=prev_ax)

elif sharey_local != 'all' and sharey_local == 'all':
    ax = fig.add_subplot(
        chosen_rows, chosen_cols, identifier, sharex=prev_ax)

elif sharey_local == 'all':
    ax = fig.add_subplot(
        chosen_rows, chosen_cols, identifier, sharex=prev_ax, sharey=prev_ax)

else:
    ax = fig.add_subplot(
        chosen_rows, chosen_cols, identifier)

elif plot_version == 1:
    if (row_id, col_id) == (None, None):
        break

    if chosen_rows == 1 and chosen_cols == 1:  # Only plot one figure
        ax = axes
    elif chosen_rows == 1 and chosen_cols != 1:  # Only plot against col_id
        ax = axes[col_id]
        tight_plot = True
    elif chosen_rows != 1 and chosen_cols == 1:  # Only plot against row_id
        ax = axes[row_id]
    elif chosen_rows != 1 and chosen_cols != 1:  # Use both row_id and col_id
        ax = axes[row_id, col_id]

    xlabel, ylabel = ('Days',
        z_property_chosen)

    if new_xlabel:
        xlabel = new_xlabel

    if new_ylabel:
        ylabel = new_ylabel

    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)

    ax.plot(timesnew, wellbase)

    prev_ax = ax

if plot_version == 1:
    for delete_empty_fig in list(range(}
fig_plotted + 1, chosen_cols * chosen_rows + 1)):
    row_id_del, col_id_del = self.
    find_row_col(delete_empty_fig, chosen_rows, chosen_cols)
    if chosen_rows == 1 and chosen_cols != 1:
        axes[col_id_del].remove()
        axes[col_id_del] = None
    elif chosen_rows != 1 and chosen_cols == 1:
        axes[row_id_del].remove()
        axes[row_id_del] = None
    elif chosen_rows != 1 and chosen_cols != 1:
        axes[row_id_del, col_id_del].remove()
    axes[row_id_del, col_id_del] = None
    elif chosen_rows == 1 and chosen_cols == 1:
        axes.remove()
        axes = None

    if alls:
        self.canvas = FigureCanvasTkAgg(fig, self.
            f2_plot)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(side=tk.TOP,
            fill=tk.BOTH, expand=True)
        self.canvas._tkcanvas.pack(side=tk.BOTTOM,
            fill=tk.BOTH, expand=True)
        self.toolbar = NavigationToolbar2Tk(self.
            canvas, self.f2_toolkit)  # Toolbar is added to canvas
        self.toolbar.update()
        if self.hold.get() == 1:
            self.figs = 0
            self.label_figs['text'] = 'Figures: ' +
            str(self.figs)
            self.grid_size_figures()
        for i_del in reversed(alls):
            x_del = i_del
            child.delete(alls.index(x_del))

def clear_xy(self, typedata):
    child = None
    if typedata == 'X':
        child = self.x_listbox
    self.plot_x = None
    elif typedata == 'Y':
        child = self.y_listbox
        self.plot_y = None
        alls = list(child.get(0, END))
        if alls:
            child.delete(0, END)

    def add_to_xy(self, typedata):
        if self.plot_id and len(self.plot_id['entries']) == 1:
            core_path = self.plot_id['simcase_path']
            cells, times = self.get_cell_time(corepath=core_path)
            newvalues = {'cells': cells, 'time': times}
            self.plot_id.update(newvalues)
            simcase = self.plot_id['simcase']
            simcase_child = self.plot_id['simcase_child'][0]
            entry = self.plot_id['entries']
            child = None
            if typedata == 'X':
                self.plot_x = {}
                child = self.x_listbox
                alls = list(child.get(0, END))
                if alls:
                    child.delete(0, END)
                    element_shown = str(simcase_child) + ' ' + str(entry[0])
                    element = self.plot_id
                    self.plot_x[element_shown] = element
                    child.insert(END, element_shown)
            elif typedata == 'Y':
                self.plot_y = {}
                child = self.y_listbox
                alls = list(child.get(0, END))
                if alls:
                    child.delete(0, END)
                    element_shown = str(simcase_child) + ' ' + str(entry[0])
                    element = self.plot_id
                    self.plot_y[element_shown] = element
                    child.insert(END, element_shown)
            def add_to_plot_list(self):
child_clear = self.local_sim_parameters
child_clear.selection_clear(0, END)
childx = self.x_listbox
childy = self.y_listbox

choicedays = [7, 8]
currentdays = self.xyz.get()

if currentdays in choicedays:
    self.xyz.set(currentdays)  # Updates chosen cells and times manually

if self.plot_id:
    child = self.pageone_listbox_plot  # Listbox where to insert
    data = self.plot_id  # New data potentially incoming
    core_path = self.plot_id['simcase_path']
    corepath=core_path
    cells, time_element = self.get_cell_time(core_path)
    ilist, jlist, klist, tlist = (time_element[2], time_element[3], time_element[4], time_element[5])
    timedays = self.get_time(core_path, 'Days', tlist).values.tolist()
    simcase = self.plot_id['simcase']
    simcase_child = self.plot_id['simcase_child'][0]
    newvalues = {'cells': cells, 'time': time_element, 'X': self.plot_x, 'Y': self.plot_y}
    self.plot_id.update(newvalues)
    entries = list(self.plot_id['entries'])

for item in entries:
    newdict = {}
    for i in list(self.plot_id.keys()):
        if i == 'entries':
            newdict[i] = item
        else:
            newdict[i] = self.plot_id[i]
    alls = list(child.get(0, END))
    count = len(alls) + 1
    marker = self.hold2.get()
    element_shown = str(count) + ':' + str(simcase) + ' ' + str(simcase_child) + ' ' + str(item)
    shown_title = None
    if self.showsimcase.get() == 0:
shown_title = str(simcase) + ' ' + str(item)

elif self.showsimcase.get() == 1:
    shown_title = str(item)

if self.showtime.get() == 0:
    shown_title = shown_title + ' t=' + str(timedays[0]) + ' days'

dict_title = {'shown_title': shown_title, 'simcase': str(simcase) + ':', 'parameter': str(item), 'timedays': timedays, 'xlabel': None, 'ylabel': None}

newdict['title'] = dict_title
newdict['fontsize'] = selffontsize

element = [count, newdict, marker]

s, self.plot_rdy[elementShown] = element
child.insert(END, elementShown)

alls = list(child.get(0, END))
self.figs = len(alls)

self.label_figs['text'] = 'Figures: ' + str(self.figs)

self.grid_size_figures()


def get_cell_time(self, corepath):
    core_path = corepath
    path_ar_input = os.path.join(core_path, 'INPUT' + '.parquet')
    path_ar_time = os.path.join(core_path, 'TIME' + '.parquet')

    x_input = pd.read_parquet(path_ar_input)
    x_time = pd.read_parquet(path_ar_time)
    imin, imax = (int(self.slidex_left.get()), int(self.slidex_right.get()))
    jmin, jmax = (int(self.slidey_left.get()), int(self.slidey_right.get()))
    kmin, kmax = (int(self.slidez_left.get()), int(self.slidez_right.get()))
    tmin, tmax = (int(self.slidetime_left.get()), int(self.slidetime_right.get()))

    ilist = list(range(imin, imax + 1))
    jlist = list(range(jmin, jmax + 1))
    klist = list(range(kmin, kmax + 1))
    tlist = list(range(tmin, tmax + 1))

    cells = x_input.loc[(x_input['i'].isin(ilist)) & (x_input['j'].isin(jlist)) & (x_input['k'].isin(klist))]
    'Cell').tolist()

    times = x_time.loc[x_time['nStep'].isin(tlist),
def remove_from_plot_list(self):
    child = self.pageone_listbox_plot
    cursors = child.currenselect()
    alls = None
    for item in reversed(cursors):
        x_del = child.get(item)
        child.delete(item)
        numbering = self.plot_rdy[x_del][0]  # Position before deletion
        alls = list(child.get(0, END))
        self.plot_rdy.pop(x_del, None)
        for i in list(range(numbering, len(alls)+1)):
            access = (i-1,)
            old_key = child.get(access)
            old_key_data = self.plot_rdy[old_key]
            self.plot_rdy.pop(old_key, None)
            current_numbering = old_key_data[0]
            current_marker = old_key_data[2]
            new_numbering = current_numbering - 1
            new_key_data = [new_numbering,
                            old_key_data[1],
                            current_marker]
            new_key = str(new_numbering) + ':' +
            old_key.split(':')[1]
            self.plot_rdy[new_key] = new_key_data
            child.delete(access)
            child.insert(access, new_key)
            self.figs = len(alls)
            self.label_figs['text'] = 'Figures: ' + str(self.figs)
            self.grid_size_figures()
simcase_child = self.simcase_child

element_list = []
w = event.widget
index = 0
child = self.pageone_listbox_plot
alls = list(child.get(0, END))
self.plot_id = []

for i in ['simcase', 'simcase_path', 'simcase_child', 'entries', 'cells', 'time', 'X', 'Y']:
    self.plot_id[i] = None

try:
    index = w.curdeselection()[0]
    properties = [w.get(int(i)) for i in w.curdeselection()]
    simcase_child_element = [simcase_child] + [
        self.merged_listbox_items[simcase_child]
    ]
    newvalues = {'simcase': simcase, 'simcase_path': simcase_path, 'simcase_child': simcase_child_element, 'entries': properties}
    self.plot_id.update(newvalues)

except IndexError:
    pass

def get_folded_properties(self, event):
    if current_tab == 'Page One':
        simcase = self.simcase
        w = event.widget
        index = 0
        try:
            index = w.curdeselection()[0]
            print('value: ' + str(value))
            if value != self.simcase_child:
                self.simcase_child = value
                filename = self.merged_listbox_items[value][0]
                core_path = self.simcase_path
                columns = None
                if filename == 'INPUT':
                    path_ar_input = os.path.join(
                        core_path, 'INPUT' + '.parquet')
                    columns = pd.read_parquet(path_ar_input).columns.tolist()
                elif filename == 'DATA':
                    path_ar_data = os.path.join(}
core_path, 'DATA' + '.parquet')
    path_ar_param = os.path.join(
core_path, 'PARAMETERS' + '.parquet')
    x_param = pd.read_parquet(
    path_ar_param)
unconverted = list(x_param.index
}
converted = list(x_param.iloc[:,
0])
extra_columns = pd.read_parquet(
    path_ar_data).columns.tolist()[:len(converted)]
unconverted = extra_columns +
converted
for pos in list(range(len(
converted))):
    self.data_conversion[
unconverted[pos]] = converted[pos]
columns = unconverted
elif filename == 'TIME':
    path_ar_time = os.path.join(
core_path, 'TIME' + '.parquet')
columns = pd.read_parquet(
    path_ar_time).columns.tolist()
elif filename == 'COMP':
    path_ar_comp = os.path.join(
core_path, 'COMP' + '.parquet')
columns = pd.read_parquet(
    path_ar_comp).columns.tolist()[2:]
elif filename == 'REGION':
    path_ar_region = os.path.join(
core_path, 'REGION' + '.parquet')
columns = pd.read_parquet(
    path_ar_region).columns.tolist()[2:]
elif filename == 'WELLS':
    path_ar_wells = os.path.join(
core_path, 'WELLS' + '.parquet')
try:
columns = list(dict_paramv2[
    simcase].keys())
except KeyError:
columns = pd.read_parquet(
    path_ar_wells).columns.tolist()[3:]}
values_to_be_inserted = columns
alls = list(self.
        local_sim_parameters.get(0, END))
        if all:
            self.local_sim_parameters.delete(0, END)
        for item in values_to_be_inserted:
            self.local_sim_parameters.insert(END, item)
    except IndexError:
        pass

def get_selected_item_prep(self, event):
    if current_tab == 'Page One':
        global current_selection
        w = event.widget
        index = 0
        try:
            index = w.curselection()[0]
            value = w.get(index)
        if value != current_selection:
            self.merged_listbox_items = {}
            self.simcase_child = None
            self.clear_xy(typedata='X')
            self.clear_xy(typedata='Y')
            child = self.local_sim_parameters
            child.selection_clear(0, END)
            child.delete(0,END)
            child2 = self.prep_sim_parameters
            child2.selection_clear(0, END)
            child2.delete(0, END)

            # child = self.prep_sim_parameters

            current_selection = value
            self.simcase = value
            values_to_be_inserted = []
            core_path = global_sim_data[value][0]
        print('core_path: ' + str(core_path))
    self.simcase_path = core_path
    ref_values = {'INPUT': 1, 'DATA': 2,
for filename in global_sim_data[value][1]:
    if filename not in ['PARAMETERS', 'WELLPARAM']:
        element = [filename, ref_values[filename]]
        values_to_be_inserted.append(element)

    sorted_version = sorted(values_to_be_inserted, key=lambda x: x[1])
    values_to_be_inserted = []
    for i in sorted_version:
        values_to_be_inserted.append(i[0])

for filename in values_to_be_inserted:
    if filename == 'COMP':
        path_ar_comp = os.path.join(core_path, 'COMP' + '.parquet')
        x_comp = pd.read_parquet(path_ar_comp)
        components = np.unique(x_comp['Component'].tolist())
        pressures = np.unique(x_comp[['Pressure']])
        for pressure in pressures:
            for component in components:
                element = ['COMP', 'COMP' + 'P=' + str(pressure) + ' ' + str(component)]
                element_shown = '
                merged_listbox_items[element_shown] = element

    elif filename == 'DATA':
        self.merged_listbox_items['DATA'] = ['DATA']

    elif filename == 'INPUT':
        self.merged_listbox_items['INPUT'] = ['INPUT']

    elif filename == 'REGION':
path_ar_region = os.path.join(core_path, 'REGION' + '.parquet')
x_region = pd.read_parquet(path_ar_region)

components = np.unique(x_region['Component'].tolist())
regions = np.unique(x_region['Region'])

for region in regions:
    for component in components:
        element = ['REGION',
                   REGION ' + str(region) + ' ' + str(component)]
        element_shown = '
REGION ' + str(region) + ' ' + str(component)
        self.merged_listbox_items[element_shown] = element

if filename == 'TIME':
    self.merged_listbox_items['TIME'] = ['TIME']

if filename == 'WELLS':
    path_ar_wells = os.path.join(core_path, 'WELLS' + '.parquet')
x_wells = pd.read_parquet(path_ar_wells)
columns.tolist()
wells_columns = x_wells.columns.tolist()

properties_wells = []
adapt_wellname = None

try:
    well_names = np.unique(x_wells['nWellName'].tolist())
adapt_wellname = '

nWellName'

except KeyError:
    well_names = np.unique(x_wells['n_well_name'].tolist())
adapt_wellname = '

n_well_name'

for column_name in wells_columns:
    if column_name not in ['
nWell', adapt_wellname, 'nType', 'Connection']:
    properties_wells.
    append(column_name)

    for well in well_names:
        well_attributes =
            x_wells.loc[x_wells[adapt_wellname] == well].iloc[0, :]
        well_number =
            well_attributes['nWell']
        well_type =
            well_attributes['nType']
        element = ['WELLS',
            well_number, well, well_type]
        element_shown = 'WELLS ' + str(well_number) + ' ' + str(well) + ' ' + str(well_type)
        self.
        merged_listbox_items[element_shown] = element
        values_to_be_inserted = list(self.
            merged_listbox_items.keys())
        alls = list(self.prep_sim_parameters
            .get(0, END))
        if alls:
            self.prep_sim_parameters.delete(0, END)
        for item in values_to_be_inserted:
            self.prep_sim_parameters.insert(END, item)

        if self.xyz.get() != 0:
            alls = self.prep_sim_parameters.
            get(0, END)
            indexdata = alls.index('DATA')
            access = (indexdata,)
            self.prep_sim_parameters.
            selection_set(access)
            self.prep_sim_parameters.
        event_generate('<<ListboxSelect>>')

    core_path = self.simcase_path
    path_ar_input = os.path.join(
        core_path, 'INPUT' + '.parquet')
    path_ar_time = os.path.join(
        core_path, 'TIME' + '.parquet')
    self.x_input = pd.read_parquet(
3096 path_ar_input)
3097     self.x_time = pd.read_parquet(
3098         path_ar_time)
3099     ival, jval, kval = (self.x_input['i'], self.x_input['j'], self.x_input['k'])
3100     imin, imax, jmin, jmax, kmin, kmax = (ival.min(), ival.max(), jval.min(), jval.max(), kval.min(), kval.max())
3101     tmin, tmax = (self.x_time['nStep'].min(), self.x_time['nStep'].max())
3102     tmin, tmax = (int(tmin), int(tmax))
3103     self.tlimits = [tmin, tmax]
3104     nsteps, ndays = (self.x_time['nStep'].tolist(), self.x_time['nDays'].tolist())
3105     for i in range(len(nsteps)):
3106         item = str(nsteps[i])
3107         days = ndays[i]
3108         self.browse_days[item] = days
3109     self.create_slider_widgets(imin, imax, jmin, jmax, kmin, kmax, tmin, tmax)
3110
3111     if self.xyz.get() != 0:
3112         self.set_xyz()
3113         if self.settings_stored:
3114             self.restore_settings()
3115     except IndexError:
3116         pass
3117
3118 def get_selected_item(self, event):
3119     '''Get current selected item in listbox. Prevents data from registering when the same selection is clicked (ie. same item still in focus)'''
3120     if current_tab == 'Page One':
3121         global current_selection
3122         w = event.widget
3123         index = 0
3124         try:
3125             index = w.cursellection()[0]
3126         value = w.get(index)
if value != current_selection:
    current_selection = value
    values_to_be_inserted = dict_param[current_selection][1:]

    if self.doitonce == 0:
        for item in values_to_be_inserted:
            self.local_sim_parameters.insert(END, item)
    self.doitonce = 1

else:
    self.properties_available = {}
    # Deleted to refill with new properties
    self.local_sim_parameters.delete(0, END)  # Can use a separate parameter for the ones they still want plotted

    for item in values_to_be_inserted:
        self.local_sim_parameters.insert(END, item)  # These are just 'potential candidates' for plotting (to be ready for the user when he/she needs to plot them fast.

    self.doitonce = 0

    core_path = dict_param[value][0]
    path_ar_input = os.path.join(core_path, 'INPUT' + '.parquet')
    path_ar_time = os.path.join(core_path, 'TIME' + '.parquet')

    self.x_input = pd.read_parquet(path_ar_input, columns=['Cell', 'i', 'j', 'k', 'X', 'Y', 'Z'])
    self.x_time = pd.read_parquet(path_ar_time)

    ival, jval, kval = (self.x_input['i'], self.x_input['j'], self.x_input['k'])

    imin, imax, jmin, jmax, kmin, kmax = (ival.min(), ival.max(), jval.min(), jval.max(), kval.min(), kval.max())

    tmin, tmax = (self.x_time['nStep'].min(), self.x_time['nStep'].max())

    nsteps, ndays = (int(tmin), int(tmax))
3153 ).tolist(), self.x_time['nDays'].tolist()
3154     for i in range(len(nsteps)):
3155         item = str(nsteps[i])
3156         days = ndays[i]
3157         self.browse_days[item] = days
3158         self.create_slider_widgets(imax, jmin, jmax, kmin, kmax, tmin, tmax)
3159     except IndexError:
3160         pass
3161
3162     def left_range_x(self, val):
3163         w1, w2, label1, label2 = (self.slidex_left, self.slidex_right, self.slidex_label1, self.slidex_label2)
3164         range_type, value, lower, upper = (['LEFT', 'X'], int(self.valuelx1.get()), int(val), int(label2['text']))
3165         self.range_calculation(ranger=range_type, w1=w1, w2=w2, label1=label1, label2=label2, single=value, lower=lower, upper=upper)
3166
3167     def right_range_x(self, val):
3168         w1, w2, label1, label2 = (self.slidex_left, self.slidex_right, self.slidex_label1, self.slidex_label2)
3169         range_type, value, lower, upper = (['RIGHT', 'X'], int(self.valuelx1.get()), int(label1['text']), int(val))
3170         self.range_calculation(ranger=range_type, w1=w1, w2=w2, label1=label1, label2=label2, single=value, lower=lower, upper=upper)
3171
3172     def left_range_y(self, val):
3173         w1, w2, label1, label2 = (self.slidey_left, self.slidey_right, self.slidey_label1, self.slidey_label2)
3174         range_type, value, lower, upper = (['LEFT', 'Y'], int(self.valueyl1.get()), int(val), int(label2['text']))
3175         self.range_calculation(ranger=range_type, w1=w1, w2=w2, label1=label1, label2=label2, single=value, lower=lower, upper=upper)
3176
3177     def right_range_y(self, val):
3178         w1, w2, label1, label2 = (self.slidey_left, self.slidey_right, self.slidey_label1, self.slidey_label2)
3179         range_type, value, lower, upper = (['RIGHT', 'Y'], int(self.valueyl1.get()), int(label1['text']), int(val)
def left_range_z(self, val):
    w1, w2, label1, label2 = (self.slidez_left, self.slidez_right, self.slidez_label1, self.slidez_label2)
    value, lower, upper = (["LEFT", "Z"], int(self.valuez1.get()), int(val), int(label2["text"]))
    self.range_calculation(ranger=range_type, w1=w1, w2=w2, label1=label1, label2=label2, single=value, lower=lower, upper=upper)

def right_range_z(self, val):
    w1, w2, label1, label2 = (self.slidez_left, self.slidez_right, self.slidez_label1, self.slidez_label2)
    value, lower, upper = (["RIGHT", "Z"], int(self.valuez1.get()), int(label1["text"]), int(val))
    self.range_calculation(ranger=range_type, w1=w1, w2=w2, label1=label1, label2=label2, single=value, lower=lower, upper=upper)

def left_range_time(self, val):
    w1, w2, label1, label2 = (self.slidetime_left, self.slidetime_right, self.slidetime_label1, self.slidetime_label2)
    value, lower, upper = ("LEFT", int(self.valuetimel.get()), int(val), int(w2.get()))
    self.range_calculation_time(ranger=range_type, w1=w1, w2=w2, label1=label1, label2=label2, single=value, lower=lower, upper=upper)

def right_range_time(self, val):
    w1, w2, label1, label2 = (self.slidetime_left, self.slidetime_right, self.slidetime_label1, self.slidetime_label2)
    value, lower, upper = ("RIGHT", int(self.valuetimel.get()), int(w1.get()), int(val))
    self.range_calculation_time(ranger=range_type, w1=w1, w2=w2, label1=label1, label2=label2, single=value, lower=lower, upper=upper)
```python
def restore_settings(self):
    imin, imax = (int(self.last_settings['imin']), int(self.last_settings['imax']))
    jmin, jmax = (int(self.last_settings['jmin']), int(self.last_settings['jmax']))
    kmin, kmax = (int(self.last_settings['kmin']), int(self.last_settings['kmax']))
    tmin, tmax = (int(self.last_settings['tmin']), int(self.last_settings['tmax']))
    self.slidy_left.set(jmin)
    self.slidy_right.set(jmax)
    self.slidez_left.set(kmin)
    self.slidez_right.set(kmax)
    self.slidetime_left.set(tmin)
    self.slidetime_right.set(tmax)

def store_settings(self):
    imin, imax = (int(self.slidy_left.get()), int(self.slidy_right.get()))
    jmin, jmax = (int(self.slidey_left.get()), int(self.slidey_right.get()))
    kmin, kmax = (int(self.slidez_left.get()), int(self.slidez_right.get()))
    tmin, tmax = (int(self.slidetime_left.get()), int(self.slidetime_right.get()))
    self.last_settings['imin'] = imin
    self.last_settings['imax'] = imax
    self.last_settings['jmin'] = jmin
    self.last_settings['jmax'] = jmax
    self.last_settings['kmin'] = kmin
    self.last_settings['kmax'] = kmax
    self.last_settings['tmin'] = tmin
    self.last_settings['tmax'] = tmax
    self.settings_stored = 1

def freeze_val(self, int_var, label1, label2, w2):
    single_value = int(int_var.get())
    lower_bound = label1['text']
    if single_value == 1:
        w2.set(lower_bound)
        label2['text'] = lower_bound

def freeze_val_time(self, int_var, label1, label2,
```
3238     w2):
3239         single_value, lower_bound, lower = (int(int_var.
3240             get()), label1['text'], None)
3241         for keys in list(self.browse_days.keys()):
3242             if self.browse_days[keys] == lower_bound:
3243                 lower = int(keys)
3244                 break
3245         if single_value == 1:
3246             w2.set(lower)
3247             label2['text'] = lower_bound
3248     def range_calculation(self, ranger, w1, w2, label1,
3249         label2, single, lower, upper):
3250         if ranger[0] == 'LEFT':
3251             if single == 1:
3252                 w2.set(lower)
3253                 label1['text'] = lower
3254                 label2['text'] = lower
3255             elif lower > upper:
3256                 w1.set(upper)
3257             else:
3258                 label1['text'] = lower
3259         elif ranger[0] == 'RIGHT':
3260             if single == 1:
3261                 if label1['text'] == label2['text']:
3262                     w1.set(upper)
3263                     label1['text'] = upper
3264                     label2['text'] = upper
3265                 else:
3266                     w2.set(lower)
3267                     label2['text'] = lower
3268             elif upper < lower:
3269                 w2.set(lower)
3270             else:
3271                 label2['text'] = upper
3272         def range_calculation_time(self, ranger, w1, w2,
3273             label1, label2, single, lower, upper):
3274             if ranger == 'LEFT':
3275                 if single == 1:
3276                     w2.set(lower)
3277                     label1['text'] = self.browse_days[str(lower)]
3278                 label2['text'] = self.browse_days[str(lower)]
elif lower > upper:
    w1.set(upper)
else:
    label1['text'] = self.browse_days[str(lower)]

elif ranger == 'RIGHT':
    if single == 1:
        if label1['text'] == label2['text']:
            w1.set(upper)
            label1['text'] = self.browse_days[str(upper)]
            label2['text'] = self.browse_days[str(upper)]
        else:
            w2.set(lower)
            label2['text'] = self.browse_days[str(lower)]
    elif upper < lower:
        w2.set(lower)
    else:
        label2['text'] = self.browse_days[str(upper)]

def popup(self):
    self.w = PopupWindow(self.master)
    self.grid_button['state'] = 'disabled'
    self.master.wait_window(self.w.top)
    self.grid_button['state'] = 'normal'

def grid_size_figures(self):
    # x = lambda lx: x+i+1
    if x % 2 == 0 else x+i
    figs = self figs
    self.fig_grid_size = {}
    self.grid_dropdown.delete(0, END)
    if figs == 1:
        chosen_cols, chosen_rows = (1, 1)
        self.fig_grid_size['1x1'] = [1, 1]
        self.grid_dropdown['values'] = ['1x1']
        self.grid_dropdown.current(0)

        def figs:
            if figs % 2 == 0:
                even = 1
                factors01, factors02, even = ([], [], 0)
                for i in range(-1, 11, 2):
                    if i == -1:
i = 0
factors01.append([figs, 1])
x = figs + i + even
x_step = x
depth = 1
while x_step % 2 == 0:
    x_step = int(x_step / 2)
if x_step != 1:
    factors01.append([x_step, 2 ** depth])
depth += 1

for item in factors01:
    new_item = [item[1], item[0]]
    if new_item not in factors01:
        factors02.append(new_item)
factors01 = factors01 + factors02
sortlist = []
for item in factors01:
    combobox_item = str(item[0]) + 'x' + str(item[1])
x = item[0] + item[1]
x2 = random.uniform(0.10, 0.20)
xnew = round(x + x2, 2)
sortlist.append([xnew, combobox_item])
self.fig_grid_size[combobox_item] = item
newsortlist = sorted(sortlist, key=lambda x: x[0])
sorted_combobox_list = []
for element in newsortlist:
    sorted_combobox_list.append(element[1])
self.grid_dropdown['values'] = sorted_combobox_list
self.grid_dropdown.current(0)

def delete_figures(self, choice):
    if self.canvas:
        plt.clf()
        self.f2_toolkit.destroy()
        self.f2_toolkit = Frame(self)
        self.f2_toolkit.pack(side=TOP, fill='both',
expand=False)
    if choice == 2:
        self.f2_plot.destroy()
self.f2_plot = Frame(self)
    self.f2_plot.pack(side=TOP, padx=10, pady=10, expand=1, fill='both')
    gc.collect()
    self.figs = 0

def create_slider_widgets(self, imin, imax, jmin, jmax, kmin, kmax, tmin, tmax):
    newvalues = {'imin': imin, 'imax': imax, 'jmin': jmin, 'jmax': jmax, 'kmin': kmin, 'kmax': kmax, 'tmin': tmin, 'tmax': tmax}
    self.simcase_ijk_count.update(newvalues)

        # X-DIRECTION SLIDER
    self.slidex_label1 = Label(self.sliders, width=5, text=str(imin), bg='white', relief=SUNKEN)
    self.slidex_label1.grid(column=0, row=0, sticky='nw', padx=3, pady=3, ipady=2)
    self.slidex_left = Scale(self.sliders, from_=imin, to=imax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.left_range_x)
    self.slidex_left.grid(column=1, row=0, sticky='nw', padx=3, pady=3)
    self.slidex_right = Scale(self.sliders, from_=imin, to=imax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.right_range_x)
    self.slidex_right.grid(column=2, row=0, sticky='nw', padx=3, pady=3)
    self.slidex_label2 = Label(self.sliders, width=5, text=str(imax), bg='white', relief=SUNKEN)
    self.slidex_label2.grid(column=3, row=0, sticky='nw', padx=3, pady=3, ipady=2)
    self.valuexl1 = IntVar()
    self.freezexl1 = Checkbutton(self.sliders, variable=self.valuexl1, command=lambda: self.freeze_val(self.valuexl1, self.slidex_label1, self.slidex_label2, self.slidex_right))
    self.freezexl1.grid(column=4, row=0, sticky='nw', padx=3, pady=3)
    self.slidex_left.set(imin)
    self.slidex_right.set(imax)

        # X-DIRECTION SLIDER
# Y-DIRECTION SLIDER
self.slidey_label1 = Label(self.sliders, width=5, text=str(jmin), bg='white', relief=SUNKEN)
self.slidey_label1.grid(column=0, row=1, sticky='nw', padx=3, pady=3, ipady=2)

self.slidey_left = Scale(self.sliders, from_=jmin, to=jmax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.left_range_y)
self.slidey_left.grid(column=1, row=1, sticky='nw', padx=3, pady=3)

self.slidey_right = Scale(self.sliders, from_=jmin, to=jmax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.right_range_y)
self.slidey_right.grid(column=2, row=1, sticky='nw', padx=3, pady=3)

self.slidey_label2 = Label(self.sliders, width=5, text=str(jmax), bg='white', relief=SUNKEN)
self.slidey_label2.grid(column=3, row=1, sticky='nw', padx=3, pady=3, ipady=2)

self.valuey1 = IntVar()
self.freeze1y1 = Checkbutton(self.sliders, variable=self.valuey1,
                             command=lambda: self.freeze_val(self.valuey1, self.slidey_label1, self.slidey_label2))
self.freeze1y1.grid(column=4, row=1, sticky='nw', padx=3, pady=3)

self.slidey_left.set(jmin)
self.slidey_right.set(jmax)

# Y-DIRECTION SLIDER

# Z-DIRECTION SLIDER
self.slidez_label1 = Label(self.sliders, width=5, text=str(kmin), bg='white', relief=SUNKEN)
self.slidez_label1.grid(column=0, row=2, sticky='nw', padx=3, pady=3, ipady=2)

self.slidez_left = Scale(self.sliders, from_=kmin, to=kmax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.left_range_z)
self.slidez_left.grid(column=1, row=2, sticky='nw', padx=3, pady=3)

self.slidez_right = Scale(self.sliders, from_=kmin, to=kmax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.right_range_z)
self.slidez_right.grid(column=2, row=2, sticky='nw', padx=3, pady=3)
self.slidez_label2 = Label(self.sliders, width=5, text=str(kmax), bg='white', relief=SUNKEN)
self.slidez_label2.grid(column=3, row=2, sticky='nw', padx=3, pady=3, ipady=2)
self.valuezl = IntVar()
self.freezezl = Checkbutton(self.sliders, variable=self.valuezl, command=lambda: self.freeze_val(self.valuezl, self.slidez_label1, self.slidez_label2, self.slidez_right))
self.freezezl.grid(column=4, row=2, sticky='nw', padx=3, pady=3)
self.slidez_left.set(kmin)
self.slidez_right.set(kmax)
# Z-DIRECTION SLIDER

# TIME SLIDER

tmin_time, tmax_time = (self.browse_days[str(tmin)], self.browse_days[str(tmax)])
self.tmin_stored, self.tmax_stored = (tmin, tmax)
self.slidez_time_label1 = Label(self.sliders, width=5, text=tmin_time, bg='white', relief=SUNKEN)
self.slidez_time_label1.grid(column=0, row=3, sticky='nw', padx=3, pady=3, ipady=2)
self.slidez_time_left = Scale(self.sliders, from_=tmin, to=tmax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.left_range_time)
self.slidez_time_left.grid(column=1, row=3, sticky='nw', padx=3, pady=3)
self.slidez_time_right = Scale(self.sliders, from_=tmin, to=tmax, orient=HORIZONTAL, showvalue=0, relief=SUNKEN, width=17, command=self.right_range_time)
self.slidez_time_right.grid(column=2, row=3, sticky='nw', padx=3, pady=3)
self.slidez_time_label2 = Label(self.sliders, width=5, text=tmax_time, bg='white', relief=SUNKEN)
self.slidez_time_label2.grid(column=3, row=3, sticky='nw', padx=3, pady=3, ipady=2)
self.valueztime1 = IntVar()
sself.freezeztime1 = Checkbutton(self.sliders, variable=self.valueztime1, command=lambda: self.freeze_val(self.valueztime1, self.slidez_time_label1, self.slidez_time_label2, self.slidez_time_right))
sself.freezeztime1.grid(column=4, row=3, sticky='nw', padx=3, pady=3, ipady=2)
def find_row_col(self, identifier, user_rows, user_cols):
    ni, chosen_rows, chosen_cols = (identifier, user_cols, user_rows)
    row, col = (0, 0)
    if ni > chosen_cols * chosen_rows:
        return None, None
    else:
        if ni <= chosen_cols:
            row = 0
            col = ni - 1
        elif ni > chosen_cols:
            row = 0
            nb = ni
            while nb not in list(range(1, chosen_cols + 1)):
                nb = nb - chosen_cols
                row += 1
                col = nb - 1
            return row, col

class PageTwo(tk.Frame):
    def __init__(self, parent, controller):
        self.controller = controller
        self.parent = parent
        tk.Frame.__init__(self, parent)
        label = tk.Label(self, text='Page Two', font=LARGE_FONT)
        label.pack(padx=10, pady=10)

class PageThree(tk.Frame):
    def __init__(self, parent, controller):
        self.controller = controller
        self.parent = parent
        tk.Frame.__init__(self, parent)
label = ttk.Label(self, text='Page One..', font=LARGE_FONT)
label.pack(padx=10, pady=10)

if __name__ == '__main__':
    app = SimPlotJIN()
    app.protocol('WM_DELETE_WINDOW', app.on_closing)
    app.mainloop()