USER MANUAL

NAME: DHARSHANNAN SUGUNAN

SUPERVISOR(S): DR. MICHAEL CORNISH, DR. MONICA MARINESCU

THIS MANUAL DETAILS HOW TO IMPLEMENT THE NOVEL SOLVER PYTHON CODE FOR ANY BATTERY MODEL

TABLE OF CONTENTS

1) IMPLEMENTING A MODEL

To implement a model using the Python scripts, the scripts below need to be changed (https://github.com/Dharshannan/FUSE_Li-S_Battery_Modelling/tree/main/Finalized_Scripts):

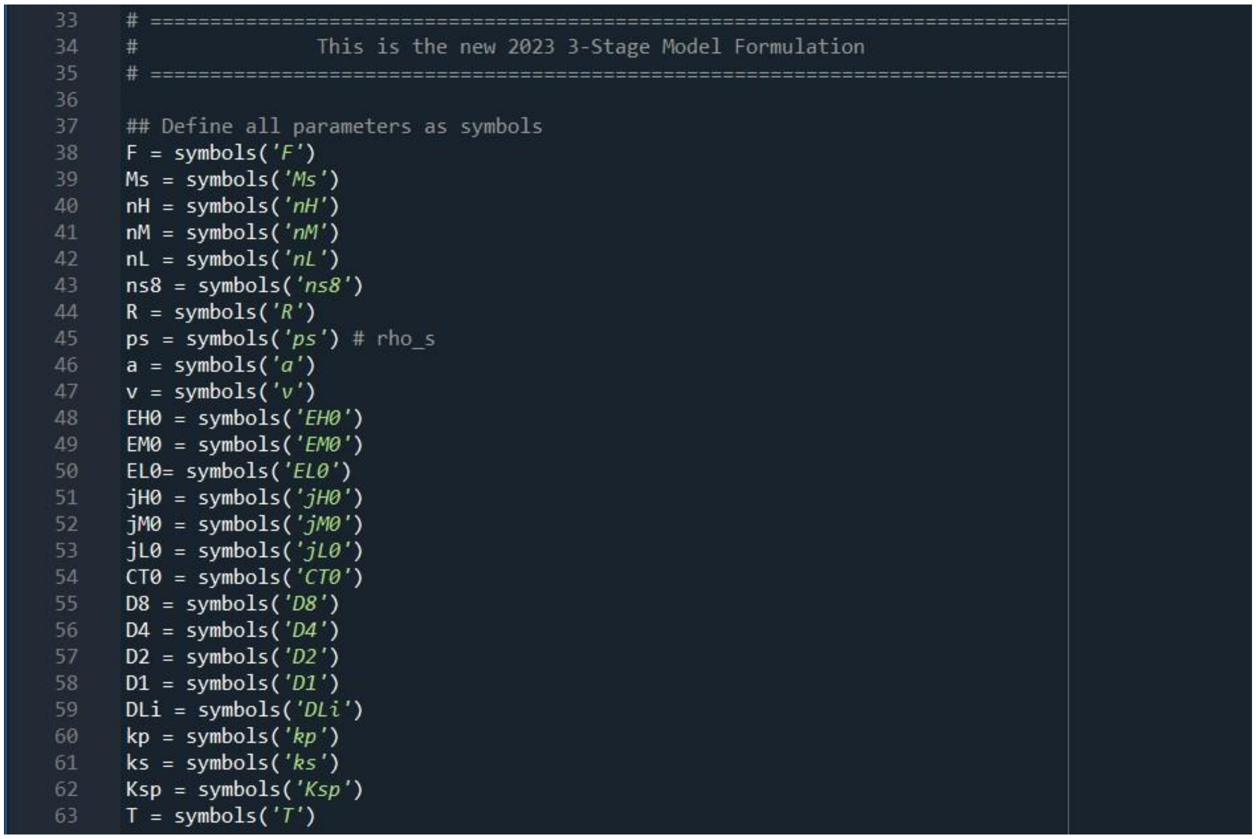
a) func.py

This script contains the defined symbols and discretized equations, including some helper functions to define the function arrays and jacobian matrix.

b) LiS_Backtrack_Solver.py

This script contains the LiS_Model calss variable and the solver function, the only change that needs to be made here is to include all the defined/changed parameters into the __init function of the LiS_Model class.

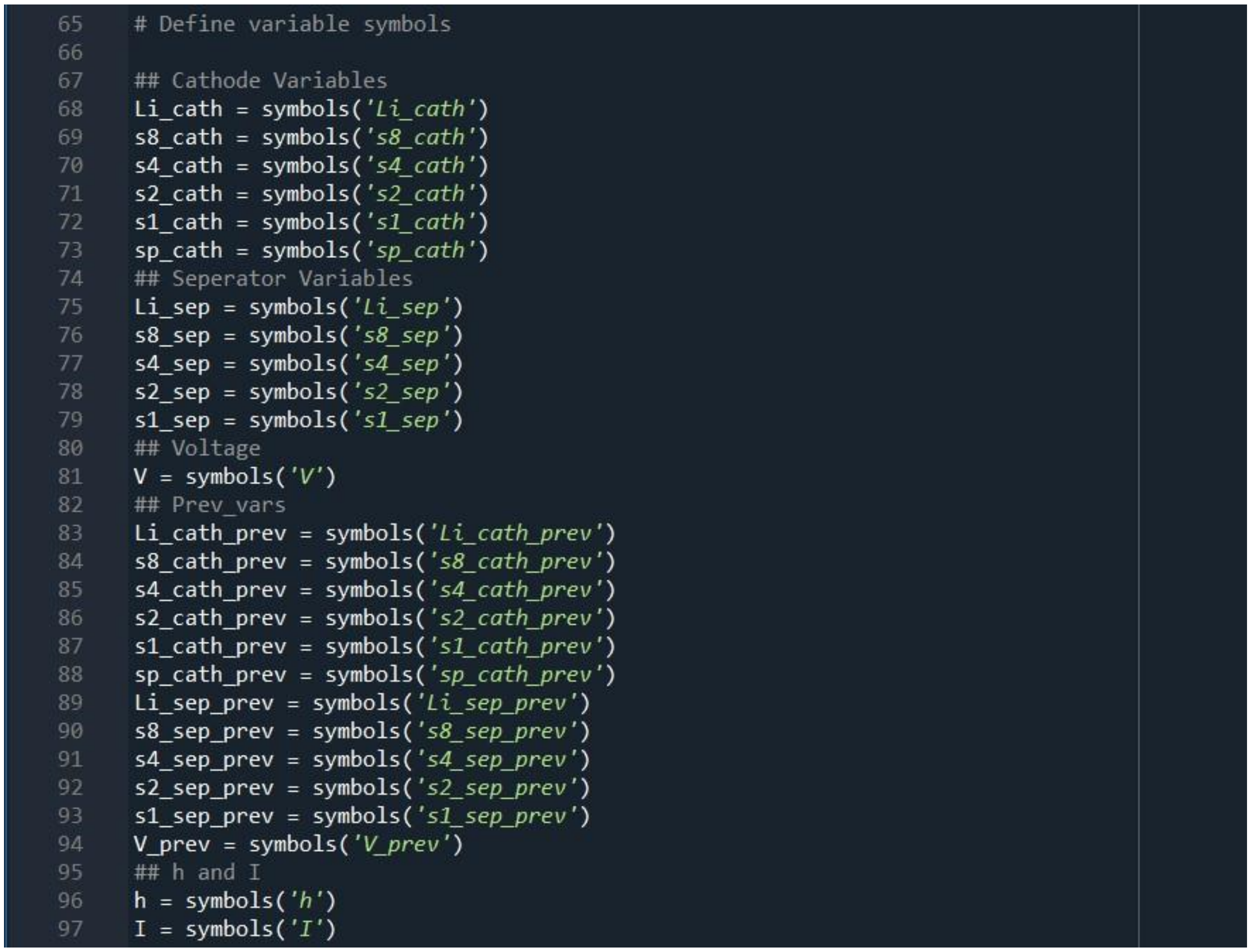
Let's take a look at how to implement a model starting with the *func.py* script:



```
# ===========================================================================
#               This is the new 2023 3-Stage Model Formulation
# ===========================================================================

## Define all parameters as symbols
F = symbols('F')
Ms = symbols('Ms')
nH = symbols('nH')
nM = symbols('nM')
nL = symbols('nL')
ns8 = symbols('ns8')
R = symbols('R')
ps = symbols('ps') # rho_s
a = symbols('a')
v = symbols('v')
EH0 = symbols('EH0')
EM0 = symbols('EM0')
EL0= symbols('EL0')
jH0 = symbols('jH0')
jM0 = symbols('jM0')
jL0 = symbols('jL0')
CT0 = symbols('CT0')
D8 = symbols('D8')
D4 = symbols('D4')
D2 = symbols('D2')
D1 = symbols('D1')
DLi = symbols('DLi')
kp = symbols('kp')
ks = symbols('ks')
Ksp = symbols('Ksp')
T = symbols('T')
```

The first step is to define the parameter as symbols as shown above in the script. The symbols function here is an imported function from the sympy library.

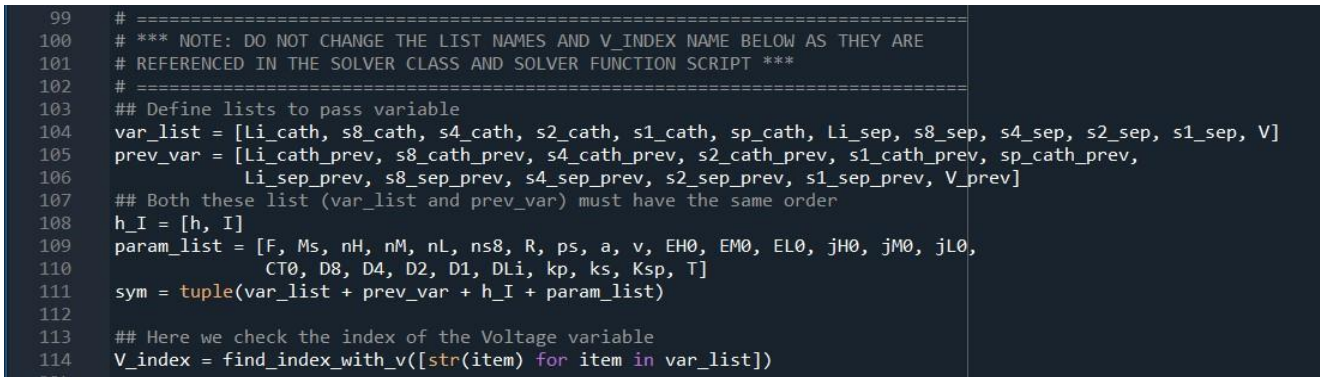
The next step is to define the variables as symbols as well, followed by previous variable symbols (* this is used for discretization of the model) as shown below:



```python
# Define variable symbols

## Cathode Variables
Li_cath = symbols('Li_cath')
s8_cath = symbols('s8_cath')
s4_cath = symbols('s4_cath')
s2_cath = symbols('s2_cath')
s1_cath = symbols('s1_cath')
sp_cath = symbols('sp_cath')
## Seperator Variables
Li_sep = symbols('Li_sep')
s8_sep = symbols('s8_sep')
s4_sep = symbols('s4_sep')
s2_sep = symbols('s2_sep')
s1_sep = symbols('s1_sep')
## Voltage
V = symbols('V')
## Prev_vars
Li_cath_prev = symbols('Li_cath_prev')
s8_cath_prev = symbols('s8_cath_prev')
s4_cath_prev = symbols('s4_cath_prev')
s2_cath_prev = symbols('s2_cath_prev')
s1_cath_prev = symbols('s1_cath_prev')
sp_cath_prev = symbols('sp_cath_prev')
Li_sep_prev = symbols('Li_sep_prev')
s8_sep_prev = symbols('s8_sep_prev')
s4_sep_prev = symbols('s4_sep_prev')
s2_sep_prev = symbols('s2_sep_prev')
s1_sep_prev = symbols('s1_sep_prev')
V_prev = symbols('V_prev')
## h and I
h = symbols('h')
I = symbols('I')
```

Leave the h, and I symbols as they are, which represents the step size and current respectively. As it can be seen all the variables defined should have respective previous variable definitions as well.

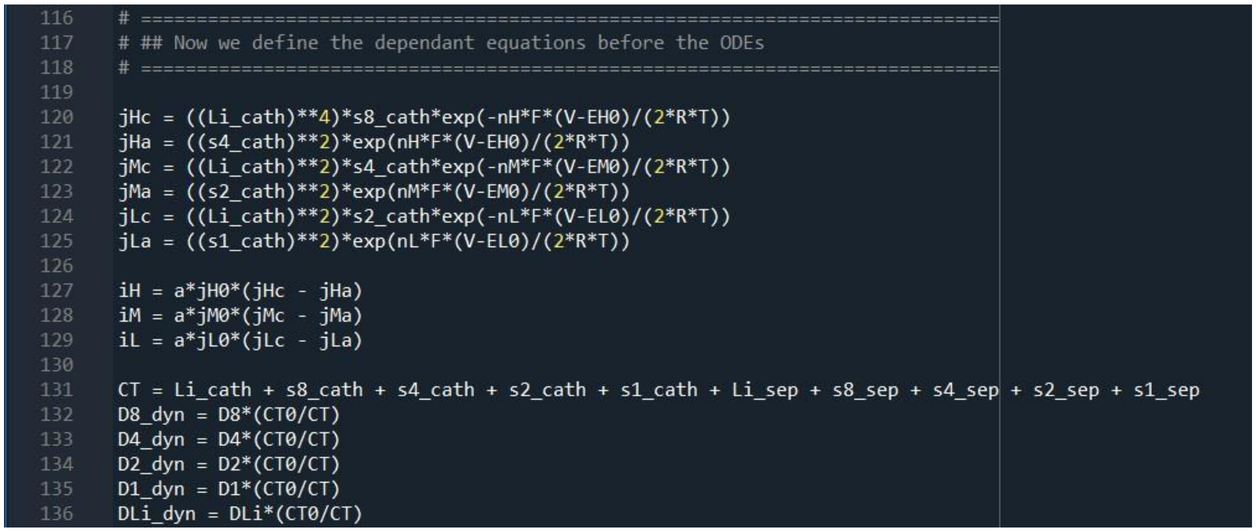
The next step is to include the variables, previous variables and parameters into the respective lists as shown below, CAUTION TO NOT CHANGE ANY OF THE NAMES OF THESE LISTS AND V_INDEX (*as these are referenced in the solver function in the other script).



```python
# ============================================================================
# *** NOTE: DO NOT CHANGE THE LIST NAMES AND V_INDEX NAME BELOW AS THEY ARE
# REFERENCED IN THE SOLVER CLASS AND SOLVER FUNCTION SCRIPT ***
# ============================================================================
## Define lists to pass variable
var_list = [Li_cath, s8_cath, s4_cath, s2_cath, s1_cath, sp_cath, Li_sep, s8_sep, s4_sep, s2_sep, s1_sep, V]
prev_var = [Li_cath_prev, s8_cath_prev, s4_cath_prev, s2_cath_prev, s1_cath_prev, sp_cath_prev,
            Li_sep_prev, s8_sep_prev, s4_sep_prev, s2_sep_prev, s1_sep_prev, V_prev]
## Both these list (var_list and prev_var) must have the same order
h_I = [h, I]
param_list = [F, Ms, nH, nM, nL, ns8, R, ps, a, v, EH0, EM0, EL0, jH0, jM0, jL0,
              CT0, D8, D4, D2, D1, DLi, kp, ks, Ksp, T]
sym = tuple(var_list + prev_var + h_I + param_list)

## Here we check the index of the Voltage variable
V_index = find_index_with_v([str(item) for item in var_list])
```

The order in which the variables are defined within the var_list is important, and the prev_var list needs to follow this order as well. The significance of the order will be further elucidated in the coming explanations. The order of the param_list is not important and can be ordered in any way. The V_index here is the index of the V (voltage) variable which is found within the list using a helper function called find_index_with_v().

The next step involves defining preliminary functions before defining the governing ODEs, as shown below, NOTE: This model shown as the example a newer implementation which has not been published at the time this guide/user manual is written.



```
# ============================================================================
# ## Now we define the dependant equations before the ODEs
# ============================================================================

jHc = ((Li_cath)**4)*s8_cath*exp(-nH*F*(V-EH0)/(2*R*T))
jHa = ((s4_cath)**2)*exp(nH*F*(V-EH0)/(2*R*T))
jMc = ((Li_cath)**2)*s4_cath*exp(-nM*F*(V-EM0)/(2*R*T))
jMa = ((s2_cath)**2)*exp(nM*F*(V-EM0)/(2*R*T))
jLc = ((Li_cath)**2)*s2_cath*exp(-nL*F*(V-EL0)/(2*R*T))
jLa = ((s1_cath)**2)*exp(nL*F*(V-EL0)/(2*R*T))

iH = a*jH0*(jHc - jHa)
iM = a*jM0*(jMc - jMa)
iL = a*jL0*(jLc - jLa)

CT = Li_cath + s8_cath + s4_cath + s2_cath + s1_cath + Li_sep + s8_sep + s4_sep + s2_sep + s1_sep
D8_dyn = D8*(CT0/CT)
D4_dyn = D4*(CT0/CT)
D2_dyn = D2*(CT0/CT)
D1_dyn = D1*(CT0/CT)
DLi_dyn = DLi*(CT0/CT)
```

The mathematical identities used such as exp(), sqrt() are also imported from the sympy library, (look at the 1st line of the *func.py* script to see the imported dependencies).

Next, we will define the ODEs which follows a very specific format. Now remember previously mentioned that the order of the var_list (list of variables) is important, this is where the significance of the order matters. The ODEs defined will need to follow the order of this list. So for this example, since the 1st element in the list is the Li_cath (Lithium cathode) concentration, the 1st ODE defined will be the ODE describing the time evolution of the Lithium cathode species, followed by s8_cath, s4_cath and so on until the last variable V (voltage). Since the voltage does not have a time dependent ODE, the equation used to describe the voltage will be the algebraic constraint, i.e:

$$I = iH + iM + iL$$

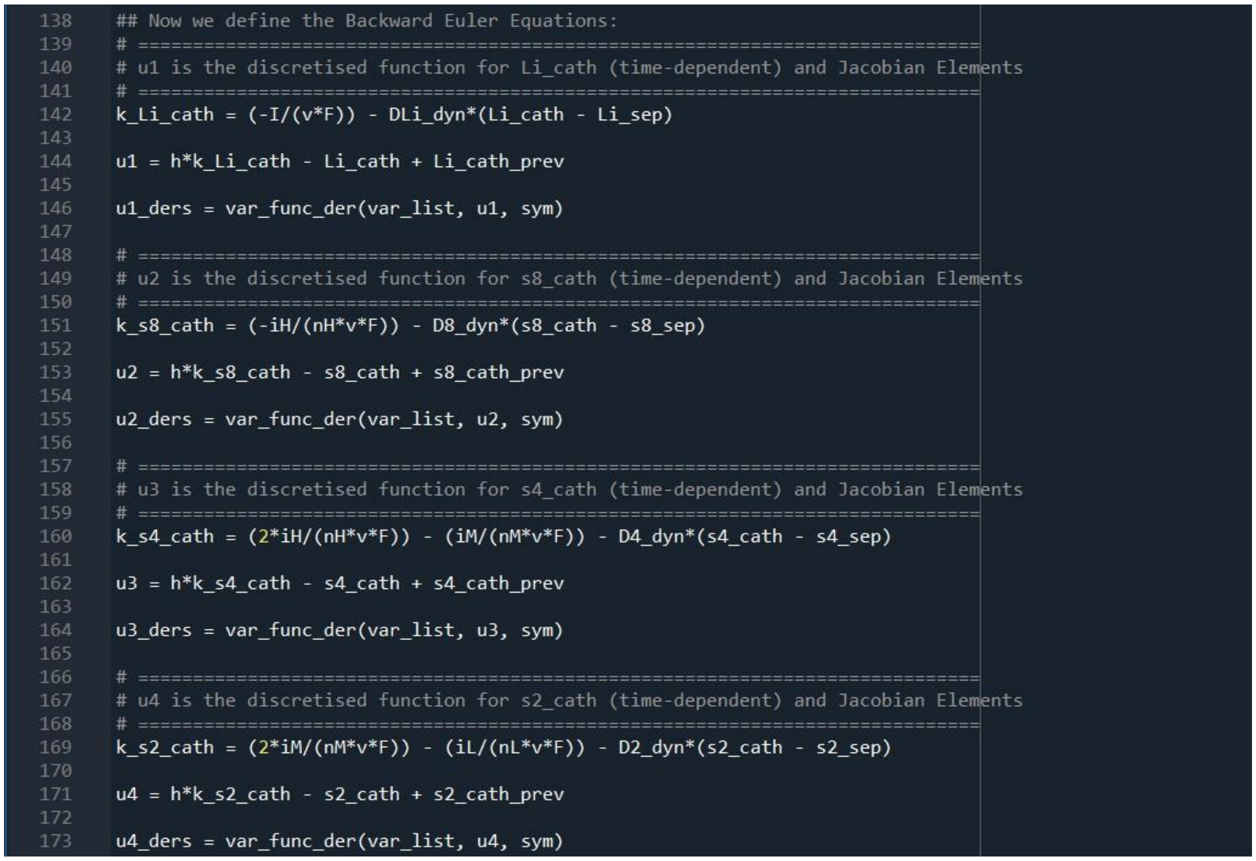
The ODEs defined will be discretized using the Backwards Euler implicit method (*most stable method although other methods can be used). The ODEs are first defined, followed by the discretization, for example for the Li_cath species its is given in equation form as below.

$$\frac{\partial C_{Li_{cath}}}{\partial t} = -\frac{I}{vF} - D_{Li_{dyn}}(C_{Li_{cath}} - C_{Li_{sep}})$$

Discretizing:

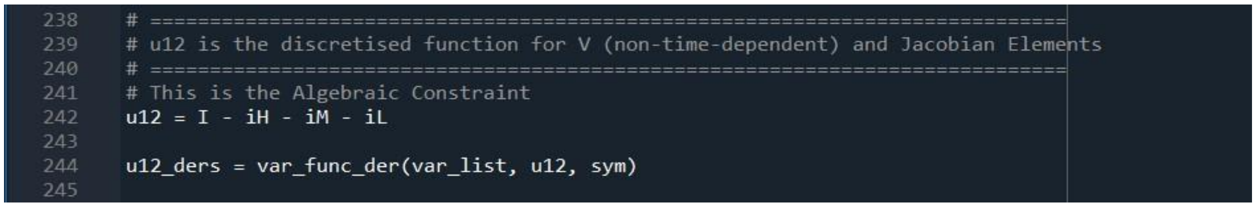
$$u1 = h\left(\frac{\partial C_{Li_{cath}}}{\partial t}\right) - C_{Li_{cath}} + C_{Li_{cath,prev}}$$

The discretized equation u1, is the obtained by re-arranging the terms to the RHS. The ODEs for all the other species are discretized in a similar manner. After discretizing, each equation is passed into a helper function called var_func_der(), which is used to return an array containing the partial derivative of the equation with respect to all the variables following the same order as the variable list as mentioned above. This is shown in the code snippet below:



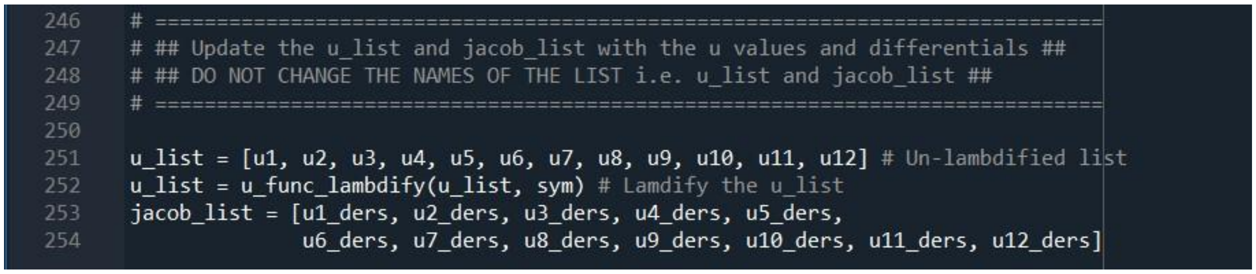
```
## Now we define the Backward Euler Equations:
# ============================================================================
# u1 is the discretised function for Li_cath (time-dependent) and Jacobian Elements
# ============================================================================
k_Li_cath = (-I/(v*F)) - DLi_dyn*(Li_cath - Li_sep)

u1 = h*k_Li_cath - Li_cath + Li_cath_prev

u1_ders = var_func_der(var_list, u1, sym)

# ============================================================================
# u2 is the discretised function for s8_cath (time-dependent) and Jacobian Elements
# ============================================================================
k_s8_cath = (-iH/(nH*v*F)) - D8_dyn*(s8_cath - s8_sep)

u2 = h*k_s8_cath - s8_cath + s8_cath_prev

u2_ders = var_func_der(var_list, u2, sym)

# ============================================================================
# u3 is the discretised function for s4_cath (time-dependent) and Jacobian Elements
# ============================================================================
k_s4_cath = (2*iH/(nH*v*F)) - (iM/(nM*v*F)) - D4_dyn*(s4_cath - s4_sep)

u3 = h*k_s4_cath - s4_cath + s4_cath_prev

u3_ders = var_func_der(var_list, u3, sym)

# ============================================================================
# u4 is the discretised function for s2_cath (time-dependent) and Jacobian Elements
# ============================================================================
k_s2_cath = (2*iM/(nM*v*F)) - (iL/(nL*v*F)) - D2_dyn*(s2_cath - s2_sep)

u4 = h*k_s2_cath - s2_cath + s2_cath_prev

u4_ders = var_func_der(var_list, u4, sym)
```

The snippet above shows the definition for the 1st four ODEs, while the rest follows the same format except for the Voltage (V) which is the final variable in the list, for which it is defined as below:



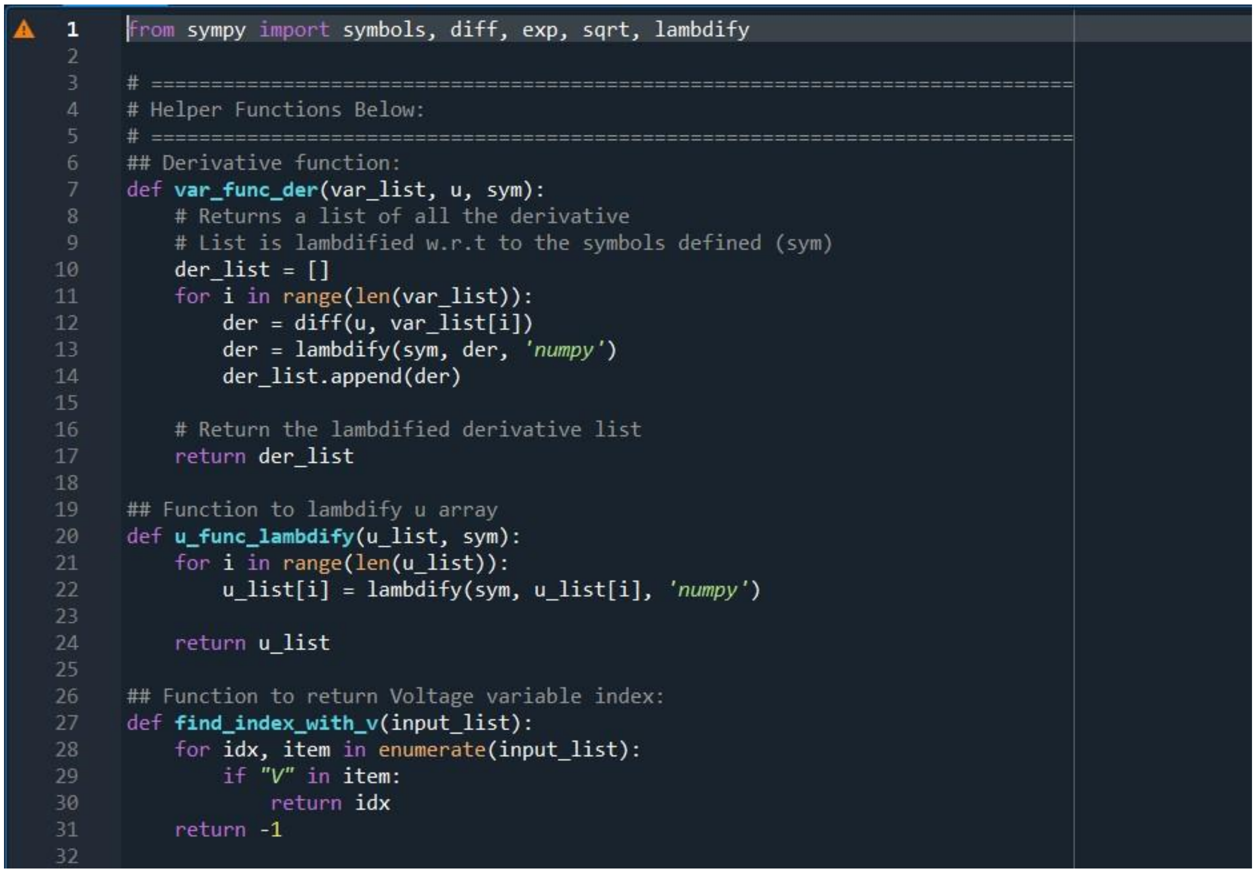
```
# ============================================================================
# u12 is the discretised function for V (non-time-dependent) and Jacobian Elements
# ============================================================================
# This is the Algebraic Constraint
u12 = I - iH - iM - iL

u12_ders = var_func_der(var_list, u12, sym)
```

The next step is to change the final 2 lists in the *func.py* script, which are the u_list and jacob_list which are the function list and the jacobian respectively, as shown below:



```python
# ============================================================================
# ## Update the u_list and jacob_list with the u values and differentials ##
# ## DO NOT CHANGE THE NAMES OF THE LIST i.e. u_list and jacob_list ##
# ============================================================================

u_list = [u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12] # Un-lambdified list
u_list = u_func_lambdify(u_list, sym) # Lamdify the u_list
jacob_list = [u1_ders, u2_ders, u3_ders, u4_ders, u5_ders,
              u6_ders, u7_ders, u8_ders, u9_ders, u10_ders, u11_ders, u12_ders]
```

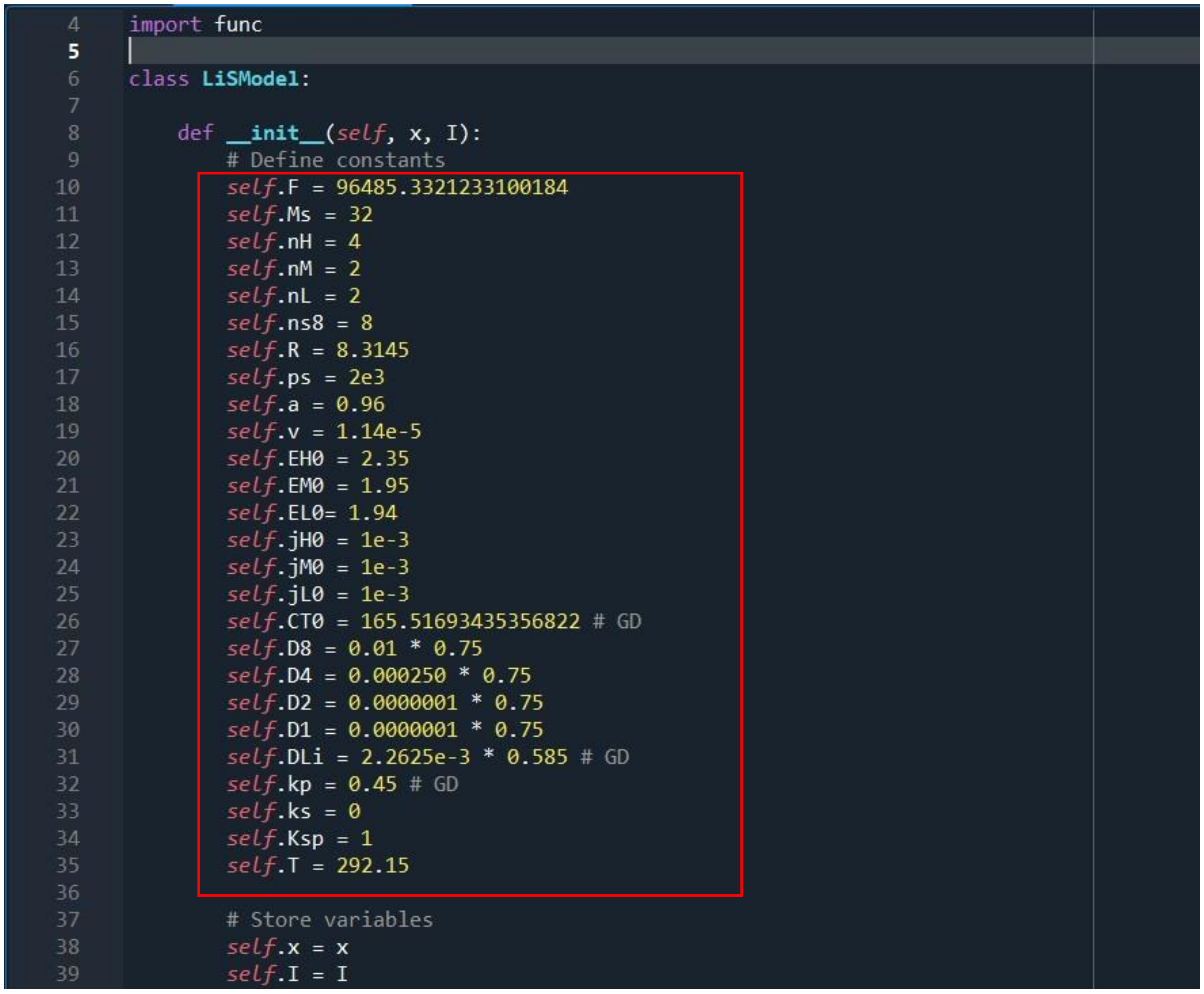
Update the u_list to contain all the u functions, and the jacob_list to conatin all the u derivative functions. Just a reminder that each u_ders variables are an array containing the partial differentials of each u function with respect to the variables defined. NOTE: AGAIN DO NOT CHANGE THE NAMES OF THESE LISTS AS THEY ARE REFERENCED IN THE OTHER SCRIPTS. The u_func_lambify() function is another helper function used to lambdify the functions from symbolic functions into numpy functions to speed us processing speed. The helper functions are as below:



```python
from sympy import symbols, diff, exp, sqrt, lambdify

# ============================================================================
# Helper Functions Below:
# ============================================================================
## Derivative function:
def var_func_der(var_list, u, sym):
    # Returns a list of all the derivative
    # List is lambdified w.r.t to the symbols defined (sym)
    der_list = []
    for i in range(len(var_list)):
        der = diff(u, var_list[i])
        der = lambdify(sym, der, 'numpy')
        der_list.append(der)

    # Return the lambdified derivative list
    return der_list

## Function to lambdify u array
def u_func_lambdify(u_list, sym):
    for i in range(len(u_list)):
        u_list[i] = lambdify(sym, u_list[i], 'numpy')

    return u_list

## Function to return Voltage variable index:
def find_index_with_v(input_list):
    for idx, item in enumerate(input_list):
        if "V" in item:
            return idx
    return -1

```

Feel free to play around with the *func.py* script to get a better feel of how everything works.

The last step is to include all the defined parameters into the __init function of the LiS_Model class in the *LiS_Backtrack_Solver.py* script, as below:



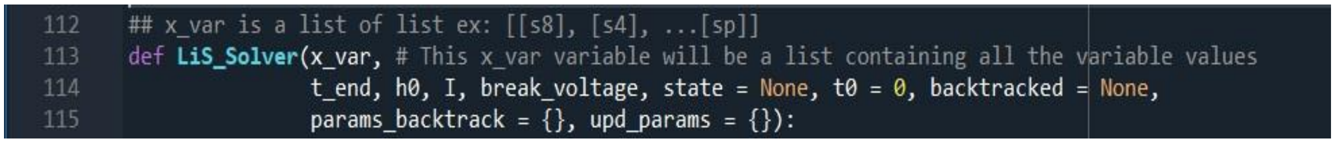
```python
import func

class LiSModel:

    def __init__(self, x, I):
        # Define constants
        self.F = 96485.3321233100184
        self.Ms = 32
        self.nH = 4
        self.nM = 2
        self.nL = 2
        self.ns8 = 8
        self.R = 8.3145
        self.ps = 2e3
        self.a = 0.96
        self.v = 1.14e-5
        self.EH0 = 2.35
        self.EM0 = 1.95
        self.EL0= 1.94
        self.jH0 = 1e-3
        self.jM0 = 1e-3
        self.jL0 = 1e-3
        self.CT0 = 165.51693435356822 # GD
        self.D8 = 0.01 * 0.75
        self.D4 = 0.000250 * 0.75
        self.D2 = 0.0000001 * 0.75
        self.D1 = 0.0000001 * 0.75
        self.DLi = 2.2625e-3 * 0.585 # GD
        self.kp = 0.45 # GD
        self.ks = 0
        self.Ksp = 1
        self.T = 292.15

        # Store variables
        self.x = x
        self.I = I
```

Ensure that the parameters here are defined with the same exact spelling (*Case Sensitive) as they are in the *func.py* script, ignore the self.x and self.I variables.

NOW WE ARE DONE DEFINING THE NEEDED PARAMETERS, VARIABLES AND EQUATIONS AND CAN PROCEED TO CALLING THE SOLVER FUNCTION TO START SOLVING BASED ON INITIAL VALUES OF THE VARIABLES. REFER TO THE NEXT SECTION TO LEARN HOW TO CALL THE SOLVER FUNCTION, ITS ARGUMENTS AND HOW TO INTREPRET THE SOLVED ARRAY RETURNED BY THE FUNCTION.

2) USING THE SOLVER FUNCTION

The solver function within the *LiS_Backtrack_Solver.py* script is defined with a few arguments and keyword arguments (args and kwargs) that are explained further below:



```
## x_var is a list of list ex: [[s8], [s4], ...[sp]]
def LiS_Solver(x_var, # This x_var variable will be a list containing all the variable values
               t_end, h0, I, break_voltage, state = None, t0 = 0, backtracked = None,
               params_backtrack = {}, upd_params = {}):
```

a) x_var

- This argument is the variable list, for which a list of the initial values of the variables are passed. NOTE: THE ORDER IN WHICH THE INITIAL CONDITIONS ARE PASSED NEEDS TO FOLLOW THE PREVIOUS var_list IN THE *func.py* SCRIPT.

b) t_end

- This is the end time for the simulation.

c) h0

- This is the initial step size.

d) I

- Current for which simulation is carried out.

e) break_voltage

- The voltage at which the simulation is cut-off (*normally done to avoid any singular matrix errors).

f) state

- This is a kwarg used to differentiate between charge and discharge, if discharge is to be simulated set state = “discharge”, if charge is to be simulated set state = “charge”.

g) t0

- Start time, defaults to 0, if not defined otherwise.

h) backtracked

- This is a kwarg used for backtracking, **a user can ignore this.**

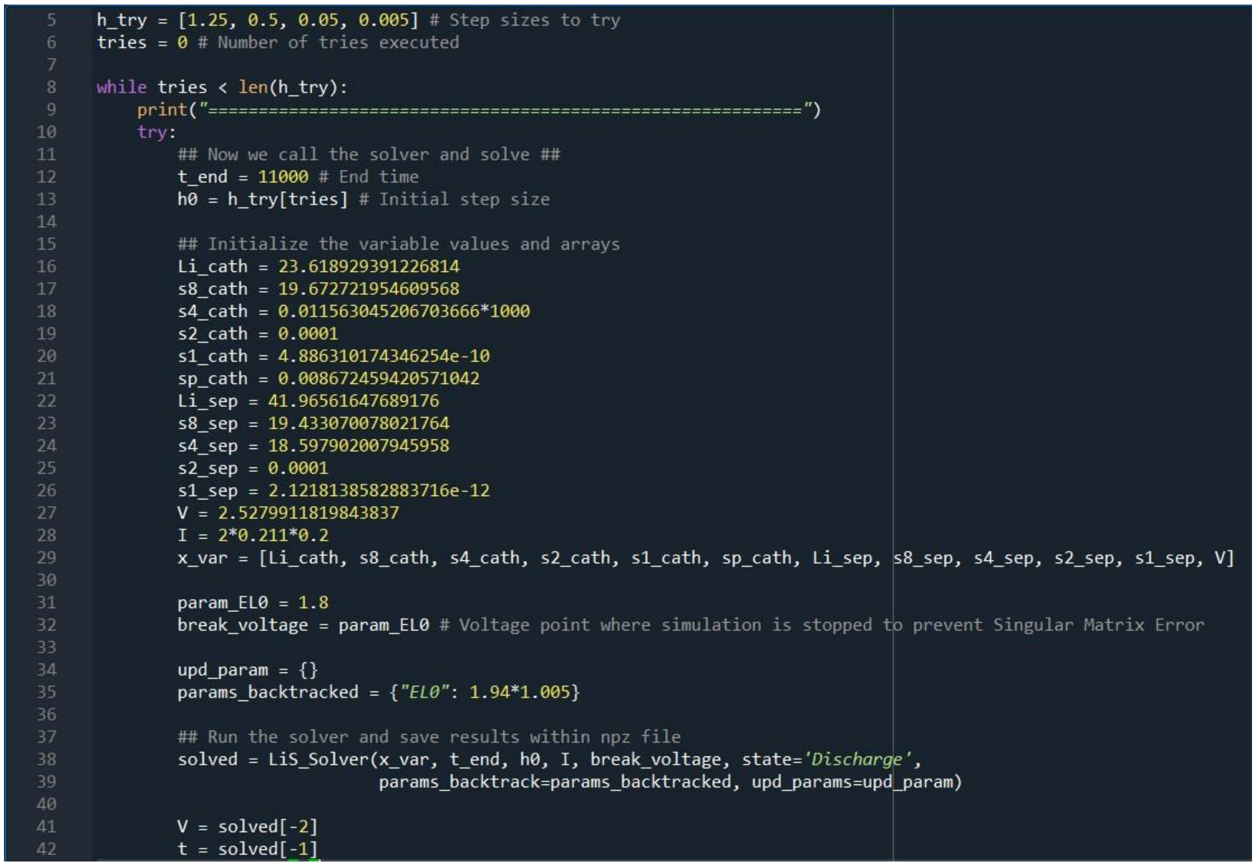
i) params_backtrack

- This is a kwarg that takes a dictionary input which specifies the parameters to backtrack and the value to backtrack to. Defaults to an empty dictionary {}.

j) upd_params

- This is also a kwarg that takes a dictionary input which specifies parameters to update and the value to update to. Defaults to an empty dictionary {}.

The code snippet below shows how to define the initial values and each argument for the solver:



```python
h_try = [1.25, 0.5, 0.05, 0.005] # Step sizes to try
tries = 0 # Number of tries executed

while tries < len(h_try):
    print("==========================================================")
    try:
        ## Now we call the solver and solve ##
        t_end = 11000 # End time
        h0 = h_try[tries] # Initial step size

        ## Initialize the variable values and arrays
        Li_cath = 23.618929391226814
        s8_cath = 19.672721954609568
        s4_cath = 0.011563045206703666*1000
        s2_cath = 0.0001
        s1_cath = 4.886310174346254e-10
        sp_cath = 0.008672459420571042
        Li_sep = 41.96561647689176
        s8_sep = 19.433070078021764
        s4_sep = 18.597902007945958
        s2_sep = 0.0001
        s1_sep = 2.1218138582883716e-12
        V = 2.5279911819843837
        I = 2*0.211*0.2
        x_var = [Li_cath, s8_cath, s4_cath, s2_cath, s1_cath, sp_cath, Li_sep, s8_sep, s4_sep, s2_sep, s1_sep, V]

        param_EL0 = 1.8
        break_voltage = param_EL0 # Voltage point where simulation is stopped to prevent Singular Matrix Error

        upd_param = {}
        params_backtracked = {"EL0": 1.94*1.005}

        ## Run the solver and save results within npz file
        solved = LiS_Solver(x_var, t_end, h0, I, break_voltage, state='Discharge',
                            params_backtrack=params_backtracked, upd_params=upd_param)

        V = solved[-2]
        t = solved[-1]
```

The solved array returned by the solver follows the same order as the defined variable list, for example since the 1st variable defined is the Li_cath concentration, the 1st index position (index 0) for the solved array (solved[0]) will be a list containing all the Li_cath concentration value over the span of the simulation time. The solved array will contain an additional array at the end index (solved[-1]) which is the time array, containing the time values starting from 0 (unless defined otherwise) to the end time.

It would be good idea to mess around with the *Test_Backtrack.py* script to get a better understanding of how the solver function and its arguments work.

3) MICROCYCLING EXAMPLE

The solver function can be used to micro-cycle between discharge and charge states, shown in the snippet below:

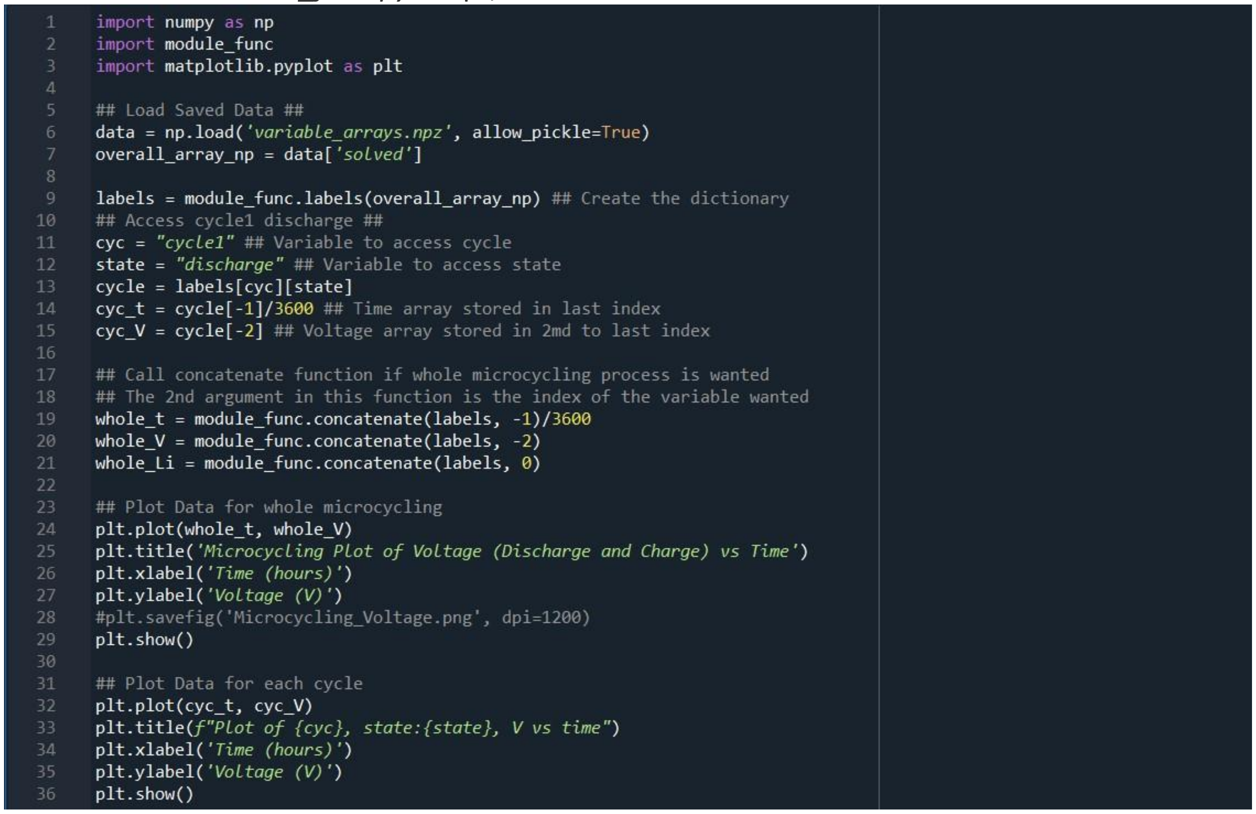
```
import numpy as np
from LiS_Backtrack_Solver import LiS_Solver, LiS_Solver2
import timeit

## Now we call the solver and solve ##
span = 11000
t0 = 0
t_end = span # End time
h0_try = [1, 0.5, 0.05, 0.005] # Initial step size for discharge
h01_try = [1.5, 0.5, 0.05, 0.005] # Initial step size for charge

## Initialize the variable values and arrays for microcycling
## Start with Discharge 1st
Li_cath = 23.618929391226814
s8_cath = 19.672721954609568
s4_cath = 0.011563045206703666*1000
s2_cath = 0.0001
s1_cath = 4.886310174346254e-10
sp_cath = 0.008672459420571042
Li_sep = 41.96561647689176
s8_sep = 19.433070078021764
s4_sep = 18.597902007945958
s2_sep = 0.0001
s1_sep = 2.1218138582883716e-12
V = 2.5279911819843837
I = 1.6*0.211*0.2

x_var = [Li_cath, s8_cath, s4_cath, s2_cath, s1_cath, sp_cath, Li_sep,
s8_sep, s4_sep, s2_sep, s1_sep, V]
# Voltage point where simulation is stopped to prevent Singular Matrix Error
discharge_break = 1.9
charge_break = 2.5

cycles = 1 ## Number of cycles to run (1 cycle is Discharge followed by
Charge)
overall_array = []
for j in range(cycles):
    overall_array.append([])

param_EL0 = 1.94
## Define backtracking parameter values
params_backtracked = {"EL0": param_EL0*1.005}
## Define different parameter values discharging
discharge_update = {"EL0": param_EL0}
## Define different parameter values charging
charge_upd = {}
```

```
## Define  solver for microcycling:
start = timeit.default_timer() ## Start timer
for i in range(cycles):
    tries = 0
    while tries < len(h0_try):
        # Discharge
        try:
            h0 = h0_try[tries]
            solved = LiS_Solver(x_var,
                                    t_end, h0, I, discharge_break,
state='Discharge', t0=t0, params_backtrack = params_backtracked,
upd_params = discharge_update)

            print(f'Cycle {i+1} Discharge Solved')
            break

        except Exception as e:
            print(e)
            if tries >= len(h0_try) - 1:
                raise
                break
            tries += 1

    list1 = solved
    overall_array[i].append(list1)
    Li_cath = solved[0][-1]
    s8_cath = solved[1][-1]
    s4_cath = solved[2][-1]
    s2_cath = solved[3][-1]
    s1_cath = solved[4][-1]
    sp_cath = solved[5][-1]
    Li_sep = solved[6][-1]
    s8_sep = solved[7][-1]
    s4_sep = solved[8][-1]
    s2_sep = solved[9][-1]
    s1_sep = solved[10][-1]
    V = solved[11][-1]
    x_var = [Li_cath, s8_cath, s4_cath, s2_cath, s1_cath, sp_cath, Li_sep,
s8_sep, s4_sep, s2_sep, s1_sep, V]
    t0 = solved[12][-1]
    t_end = t0 + span

    tries = 0
    while tries < len(h01_try):
        # Discharge
        try:
            # Charge
            h01 = h01_try[tries]
            solved2 = LiS_Solver2(x_var,
                                    t_end, h01, -I, charge_break,
state='Charge',t0=t0, upd_params = charge_upd)
            print(f'Cycle {i+1} Charge Solved')
```

```
                break

            except Exception as e:
                print(e)
                if tries >= len(h01_try) - 1:
                    raise
                    break
                tries += 1

        list2 = solved2
        overall_array[i].append(list2)
        Li_cath = solved2[0][-1]
        s8_cath = solved2[1][-1]
        s4_cath = solved2[2][-1]
        s2_cath = solved2[3][-1]
        s1_cath = solved2[4][-1]
        sp_cath = solved2[5][-1]
        Li_sep = solved2[6][-1]
        s8_sep = solved2[7][-1]
        s4_sep = solved2[8][-1]
        s2_sep = solved2[9][-1]
        s1_sep = solved2[10][-1]
        V = solved2[11][-1]
        x_var = [Li_cath, s8_cath, s4_cath, s2_cath, s1_cath, sp_cath, Li_sep,
    s8_sep, s4_sep, s2_sep, s1_sep, V]
        #print(x_var)
        t0 = solved2[12][-1]
        t_end = t0 + span
        # Update charge break
        charge_break = min(charge_break, max(solved2[11]))

        print(f'No. Cycles: {i+1}/{cycles}')

    overall_array_np = np.empty(len(overall_array), dtype=object)
    overall_array_np[:] = overall_array

    print("The time taken for completion :", (timeit.default_timer() - start),
    "s")
    np.savez('variable_arrays.npz', solved=overall_array_np, I=I)
```

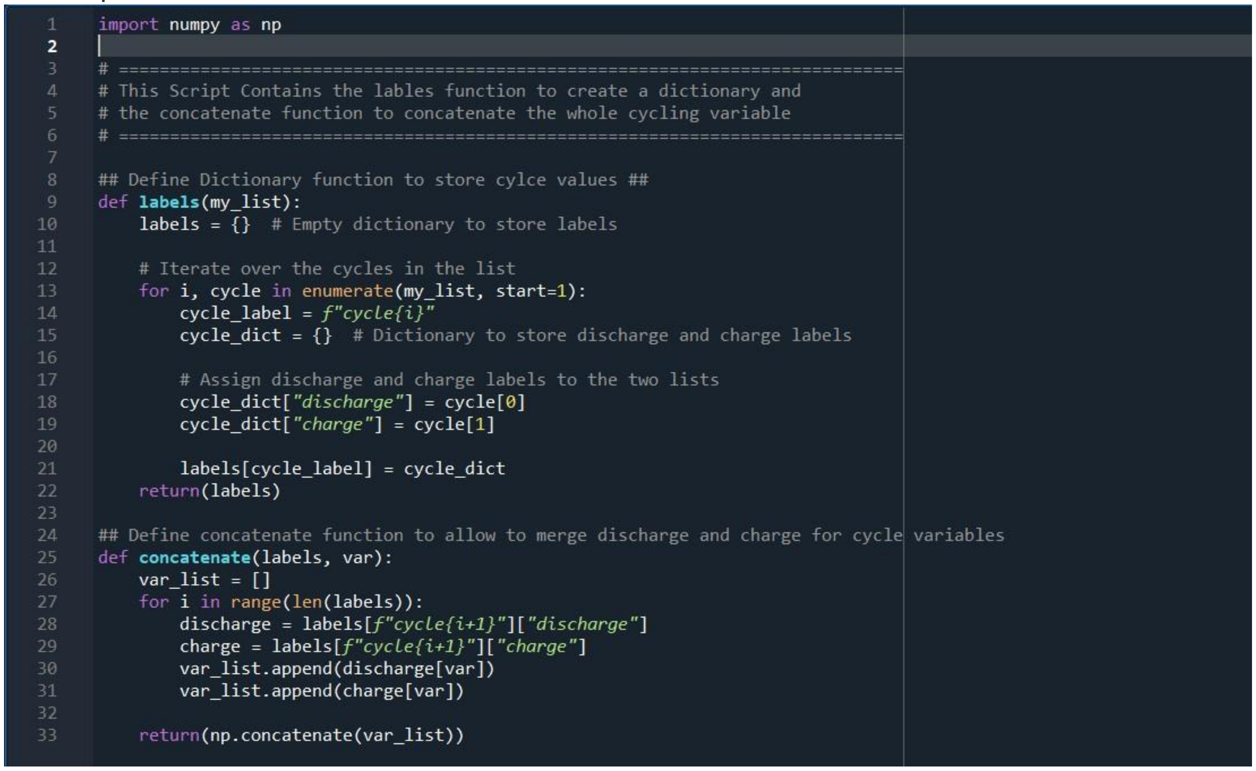
This code can be accessed via the *Test_Microcycling_v3.py* script, the solution array is saved in an npz file and can be accessed in a different script using some helper functions to ease the data processing which are defined within the *module_func.py* script.

The *Test_Saved_Microcycling.py* script shows how to access the npz file and use the helper functions from *module_func.py* script, as below:



```python
import numpy as np
import module_func
import matplotlib.pyplot as plt

## Load Saved Data ##
data = np.load('variable_arrays.npz', allow_pickle=True)
overall_array_np = data['solved']

labels = module_func.labels(overall_array_np) ## Create the dictionary
## Access cycle1 discharge ##
cyc = "cycle1" ## Variable to access cycle
state = "discharge" ## Variable to access state
cycle = labels[cyc][state]
cyc_t = cycle[-1]/3600 ## Time array stored in last index
cyc_V = cycle[-2] ## Voltage array stored in 2md to last index

## Call concatenate function if whole microcycling process is wanted
## The 2nd argument in this function is the index of the variable wanted
whole_t = module_func.concatenate(labels, -1)/3600
whole_V = module_func.concatenate(labels, -2)
whole_Li = module_func.concatenate(labels, 0)

## Plot Data for whole microcycling
plt.plot(whole_t, whole_V)
plt.title('Microcycling Plot of Voltage (Discharge and Charge) vs Time')
plt.xlabel('Time (hours)')
plt.ylabel('Voltage (V)')
#plt.savefig('Microcycling_Voltage.png', dpi=1200)
plt.show()

## Plot Data for each cycle
plt.plot(cyc_t, cyc_V)
plt.title(f"Plot of {cyc}, state:{state}, V vs time")
plt.xlabel('Time (hours)')
plt.ylabel('Voltage (V)')
plt.show()
```

The helper functions are as below:



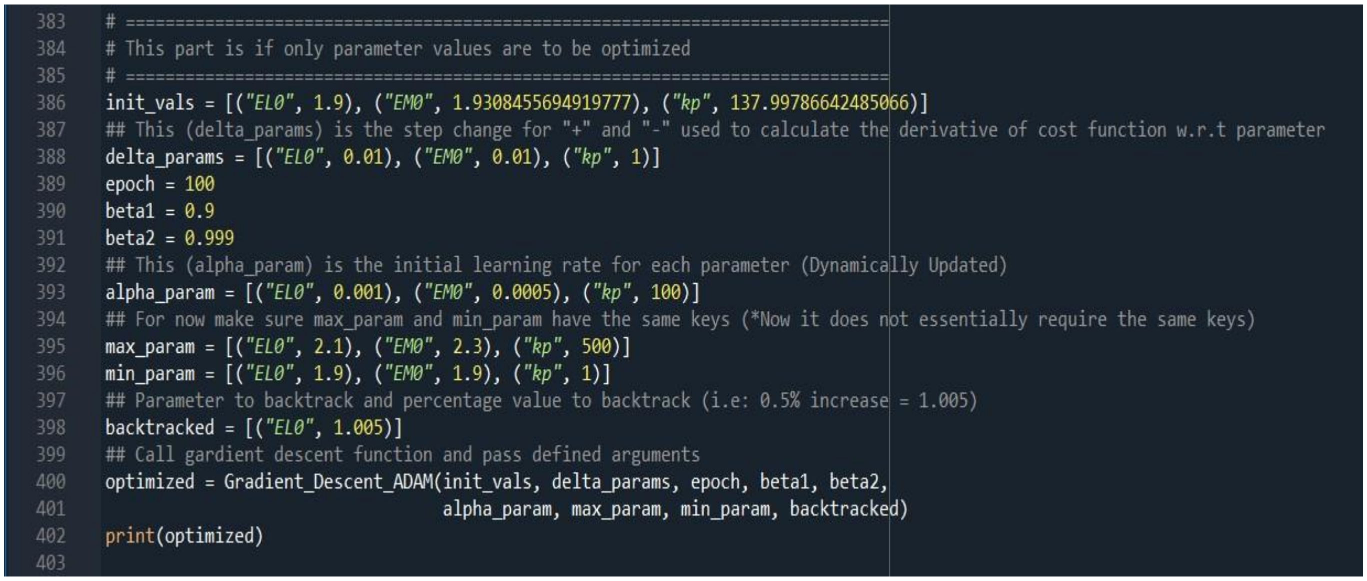
```python
import numpy as np

# ==============================================================================
# This Script Contains the lables function to create a dictionary and
# the concatenate function to concatenate the whole cycling variable
# ==============================================================================

## Define Dictionary function to store cylce values ##
def labels(my_list):
    labels = {}  # Empty dictionary to store labels

    # Iterate over the cycles in the list
    for i, cycle in enumerate(my_list, start=1):
        cycle_label = f"cycle{i}"
        cycle_dict = {}  # Dictionary to store discharge and charge labels

        # Assign discharge and charge labels to the two lists
        cycle_dict["discharge"] = cycle[0]
        cycle_dict["charge"] = cycle[1]

        labels[cycle_label] = cycle_dict
    return(labels)

## Define concatenate function to allow to merge discharge and charge for cycle variables
def concatenate(labels, var):
    var_list = []
    for i in range(len(labels)):
        discharge = labels[f"cycle{i+1}"]["discharge"]
        charge = labels[f"cycle{i+1}"]["charge"]
        var_list.append(discharge[var])
        var_list.append(charge[var])

    return(np.concatenate(var_list))
```

4) GRADIENT DESCENT

The solver can also be used within a gradient descent scheme for parameter optimization based on experimental results.

The scripts containing the gradient descent functions and another test script are the *Test_General_GD_ADAM.py* and *Test_Gradient_Descent.py* scripts respectively.

The gradient descent function takes in a few arguments as described in the code snippet below:



```python
# ============================================================================
# This part is if only parameter values are to be optimized
# ============================================================================
init_vals = [("EL0", 1.9), ("EM0", 1.9308455694919777), ("kp", 137.99786642485066)]
## This (delta_params) is the step change for "+" and "-" used to calculate the derivative of cost function w.r.t parameter
delta_params = [("EL0", 0.01), ("EM0", 0.01), ("kp", 1)]
epoch = 100
beta1 = 0.9
beta2 = 0.999
## This (alpha_param) is the initial learning rate for each parameter (Dynamically Updated)
alpha_param = [("EL0", 0.001), ("EM0", 0.0005), ("kp", 100)]
## For now make sure max_param and min_param have the same keys (*Now it does not essentially require the same keys)
max_param = [("EL0", 2.1), ("EM0", 2.3), ("kp", 500)]
min_param = [("EL0", 1.9), ("EM0", 1.9), ("kp", 1)]
## Parameter to backtrack and percentage value to backtrack (i.e: 0.5% increase = 1.005)
backtracked = [("EL0", 1.005)]
## Call gardient descent function and pass defined arguments
optimized = Gradient_Descent_ADAM(init_vals, delta_params, epoch, beta1, beta2,
                                  alpha_param, max_param, min_param, backtracked)
print(optimized)
```

In this example only 3 parameters i.e. EL0, EM0 and kp are optimized however the gradient descent solver can as many parameters or even initial values for optimization. To see how to implement the solver further including the initial value optimization please refer to the *Test_General_GD_ADAM.py* script, which contains comments on how to implement these. Feel free to play around with the code for the gradient descent.

NOTE: The gradient descent used in this code is an optimized approach using Adaptive Moment (ADAM), and to also ensure that the simulated results and the experimental results have a common time step and end time, the simulated data and experimental data are interpolated, refer to the *ret_data()* function in the script above for more in depth explanation of how this is done. The format for which each argument is passed into the *Gradient_Descent_ADAM()* function follows the same format as shown above which is a list containing a tuple with a string and a float element, ex: the initial guess values are [("EL0", 1.9), ("EM0", 1.9308455694919777), ("kp", 137.99786642485066)]. Only the arguments for epoch (*number of iterations), beta1 and beta2 (*ADAM hyper-parameters) are constant values as shown in the snippet above.

5) CLOSING REMARKS

The code for the novel solver above is developed as an alternative to the existing PyBaMM solver. Feel free to experiment with the code from the GitHub repository linked. The long-term goal of this novel solver is to become an open-source solver that can be further developed for use of not only Lithium Sulfur cell simulations but also as a more general-purpose battery simulation tool. The link to the finalized Python scripts as stated in the beginning of the document is:

https://github.com/Dharshannan/FUSE_Li-S_Battery_Modelling/tree/main/Finalized_Scripts

As a closing statement I would like to thank my supervisor Dr. Michael Cornish for guidance in developing this novel solver during my summer research, special thanks to the project director Dr. Monica Marinescu for organizing the research project and I would also like to express my gratitude to the Faraday Institute to sponsoring this summer research project.

Feel free to contact the author, Dharshannan Sugunan via:

Email: dharshannan1607@gmail.com

LinkedIn: https://www.linkedin.com/in/dharshannan-sugunan-833577283/

GitHub: https://github.com/Dharshannan