Imperial College London

REDUCED ORDER MODELLING FOR Li-S BATTERIES VIA NUMERICAL SCHEMES

NAME: DHARSHANNAN SUGUNAN

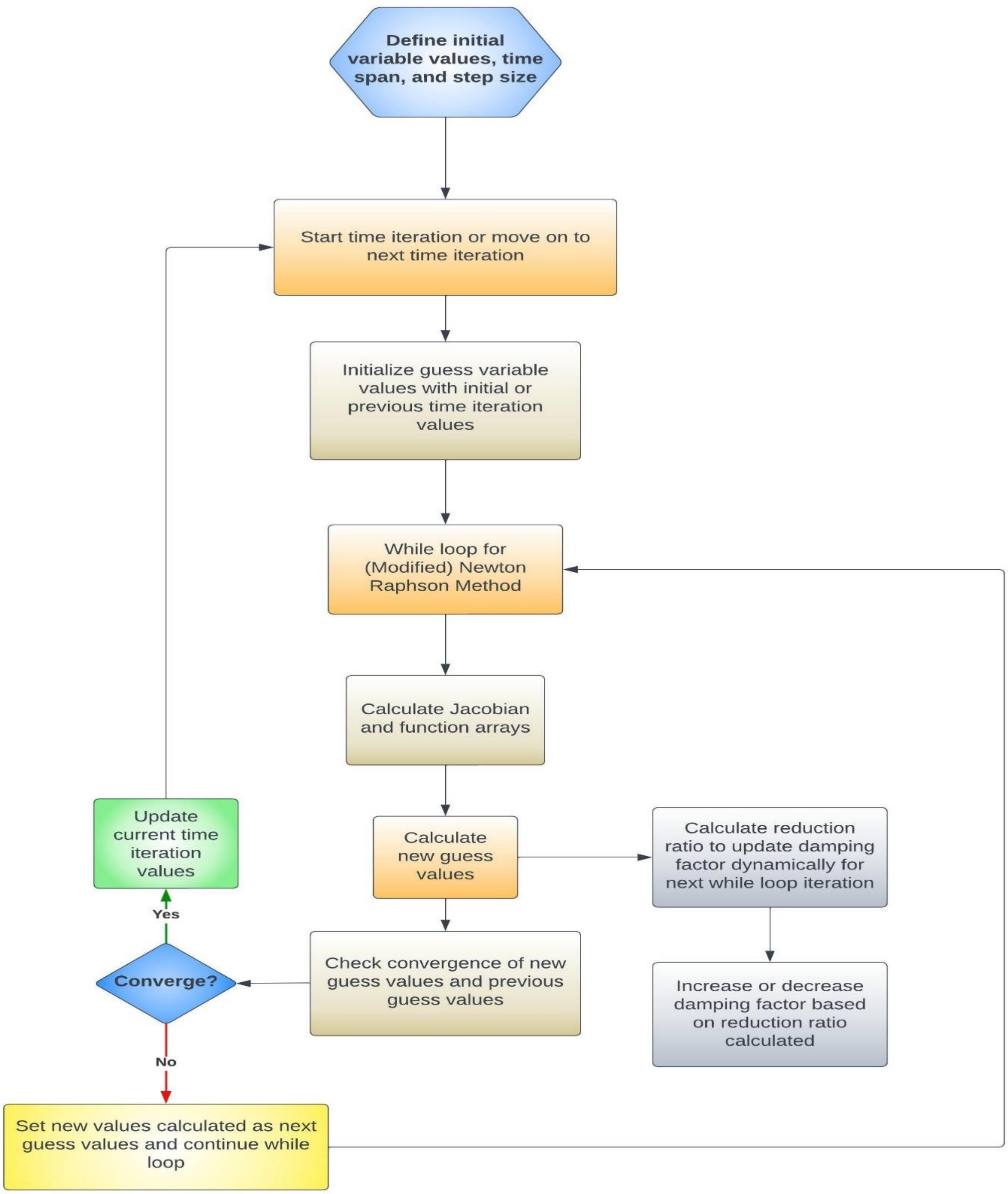
CID: 01842746

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

THE FARADAY INSTITUTION

1) BASE MODEL

The base model is coded using a Backwards-Euler discretization and a (modified) Newton Raphson method to solve the system of non-linear equations describing the time evolution of sulfur species defined using the Nernst and Butler-Volmer equations. The base model uses a pre-defined step size which is simple, however less efficient and unstable (takes a time of 10-45 mins to run). The flowchart below shows the outline of how the solver algorithm functions:



The table below highlights the different hyper-parameters used in the solver, their respective definitions and common values used:

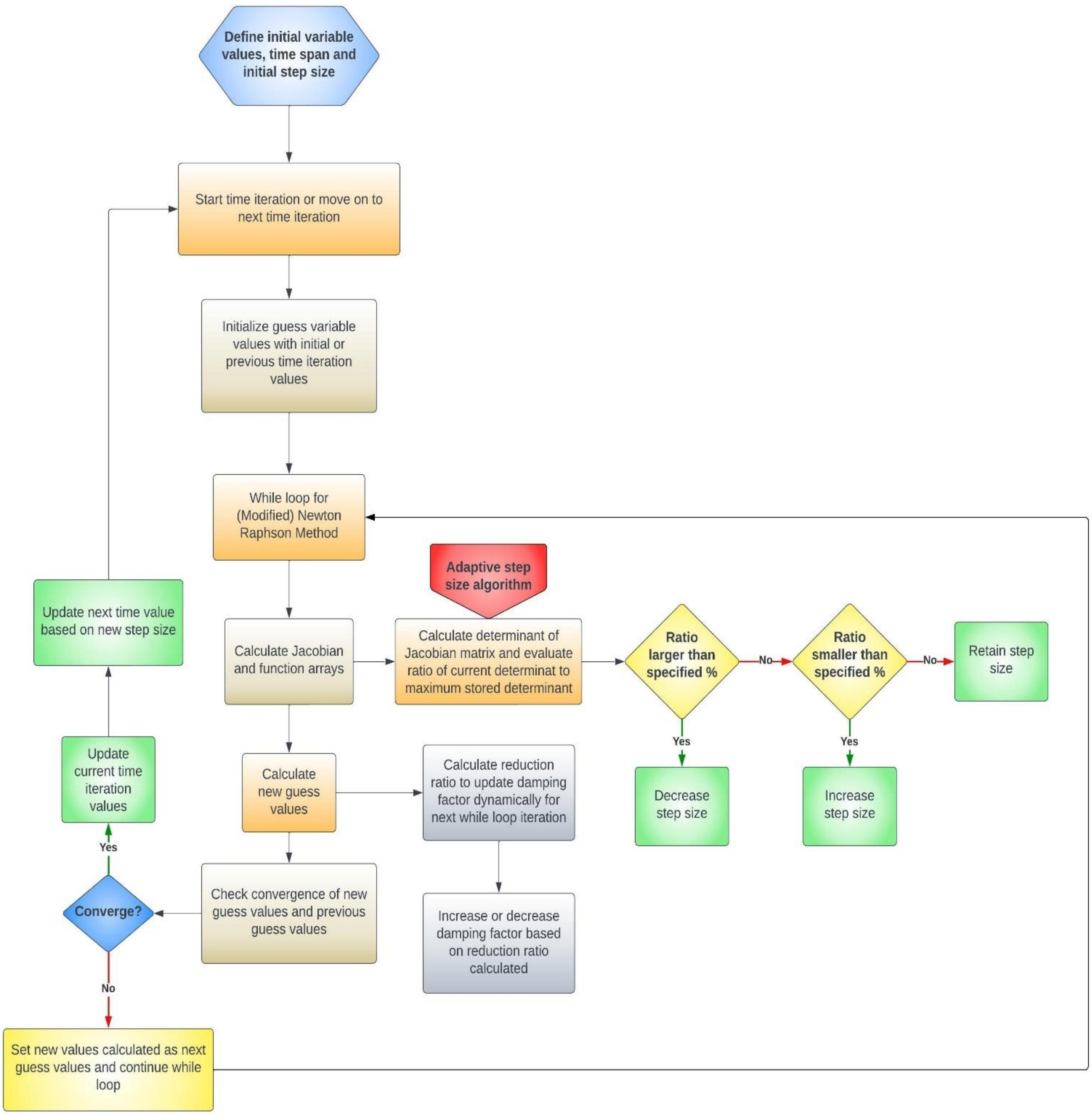
Hyper-parameter	Definition	Common values
lamda	Damping factor used to damp (reduce) Jacobian matrix in case of det(Jacobian) approaching large values.	1.0, 0.8 (Starting value, dynamically updated)
damping_update_factor	Used to dynamically update the damping factor (lamda) for different while loop iterations based on ratio of actual reduction to predicted reduction.	0.5, 0.25
damping_min	Minimum value of damping factor	1e-8, 1e-16
regularization_factor	Used as a sort of penalty method, for which the factor is added to the leading diagonal of the Jacobian matrix to ease the instability of the non-linear system of coupled equations.	5e-4, 2.5e-4
n_damp	Used to raise the power of the damping_update_factor for faster or slower damping.	1, 2

The code snippet below shows how these hyper-parameters are used in conjunction of calculating the reductions. NOTE: it is also worth experimenting on trying different values for the highlighted (in red) values which corresponds to the upper and lower bound of the reduction ratio used to update the damping factor:

```
## Now we solve as usual the new step size will be implemented in the next iteration
jacob = jacob + regularization_factor * np.eye(len(x))
jacobinv = np.linalg.inv(jacob)
# Calculate new values via Newton-Raphson
delta = - np.matmul(jacobinv,u_array)
new_val = x + lamda*delta
new_val = abs(new_val)

model2 = LiSModel(new_val, I) ## Initialize the model
## Change any values if a new value wants to be used apart from the default ones in the model
model2.update_parameters(**upd_params)
unew_array = model2.f(h, s8[-1], s4[-1], s2[-1], s[-1], V[-1], sp[-1])

# Compute the ratio of actual reduction to predicted reduction
actual_reduction = np.linalg.norm(u_array) - np.linalg.norm(unew_array)
predicted_reduction = np.linalg.norm(u_array) - np.linalg.norm(u_array + jacob @ delta)
ratio = abs(actual_reduction / predicted_reduction)

# Update the damping factor based on the ratio
n_damp = 1
if ratio > 1e-3:
    lamda *= (damping_update_factor**n_damp)
elif ratio < 1e-4:
    lamda /= (damping_update_factor**n_damp)

# Ensure the damping factor does not go below the minimum value
lamda = max(lamda, damping_min)
```

2) ADAPTIVE STEP SIZE MODEL

The next model attempted was the adaptive step size model. This model is an extended version of the base model for which instead of using a fixed step size, the step size is dynamically updated via calculating the ratio of the determinant of the Jacobian for the current iteration to the maximum stored determinant. This model shows a significant increase in both stability and computation time and efficiency (takes only 2-3s to run). The flowchart below shows the workflow of the model solver:



As mentioned, this model is a huge improvement to the base model which takes a lengthy time of between 10-45 mins to run a single discharge simulation depending on the step size chosen, however this modified model adaptively updates the step which allows the model to account for unstable solution regions more precisely and increases computational speed for stable solution regions. The table below shows the new added hyper-parameters for the adaptive step size algorithm on top of the previous hyper-parameters:

Hyper-parameters	Definition	Common values
min_h	Minimum allowed step size	1e-4, 1e-6
max_h	Maximum allowed step size	0.5, 1
`h_new = max(h*(0.2), min_h)`	The red highlighted value is the factor used to reduce the step size by every iteration.	0.05, 0.2, 0.25, 0.5
`h_new = min(h/(0.75), max_h)`	The red highlighted value is the factor used to increase the step size by every iteration.	0.25, 0.5, 0.75

The code snippet below shows how the adaptive step size algorithm is adapted into the solver. NOTE: the red highlighted values are also worth experimenting with as they represent the values for the upper bound of Jacobian determinant ratio to decrease the step size (common values: 1.2, 1.5) and lower bound to increase the step size (common values, 0.2, 0.5, 0.75, 1.0):

```
        # ================================================================================
        #     ### This will dynamically update the step size every iteration ###
        # ================================================================================
        if i > 1: ## Only check after 1st iteration ##
            ## Continuously update the maximum value of determinant of Jacobian encountered
            max_jacobian = max(max_jacobian, abs(jacob_array[i-3]))
            ## Calculate the ratio of current determinant to maximum for Jacobian
            max_ratio = abs(jacob_array[i-2])/max_jacobian
            ## These values need configuration
            if max_ratio >= 1.5: ## This indicates step needs to be reduced
                h_new = max(h*(0.25), min_h) ## Saturate at minimum step size
            elif max_ratio <= 0.2: ## This indicates step size can be increased
                h_new = min(h/(0.75), max_h) ## Saturate at maximum step size
            else:
                h_new = h
        else:
            h_new = h
```

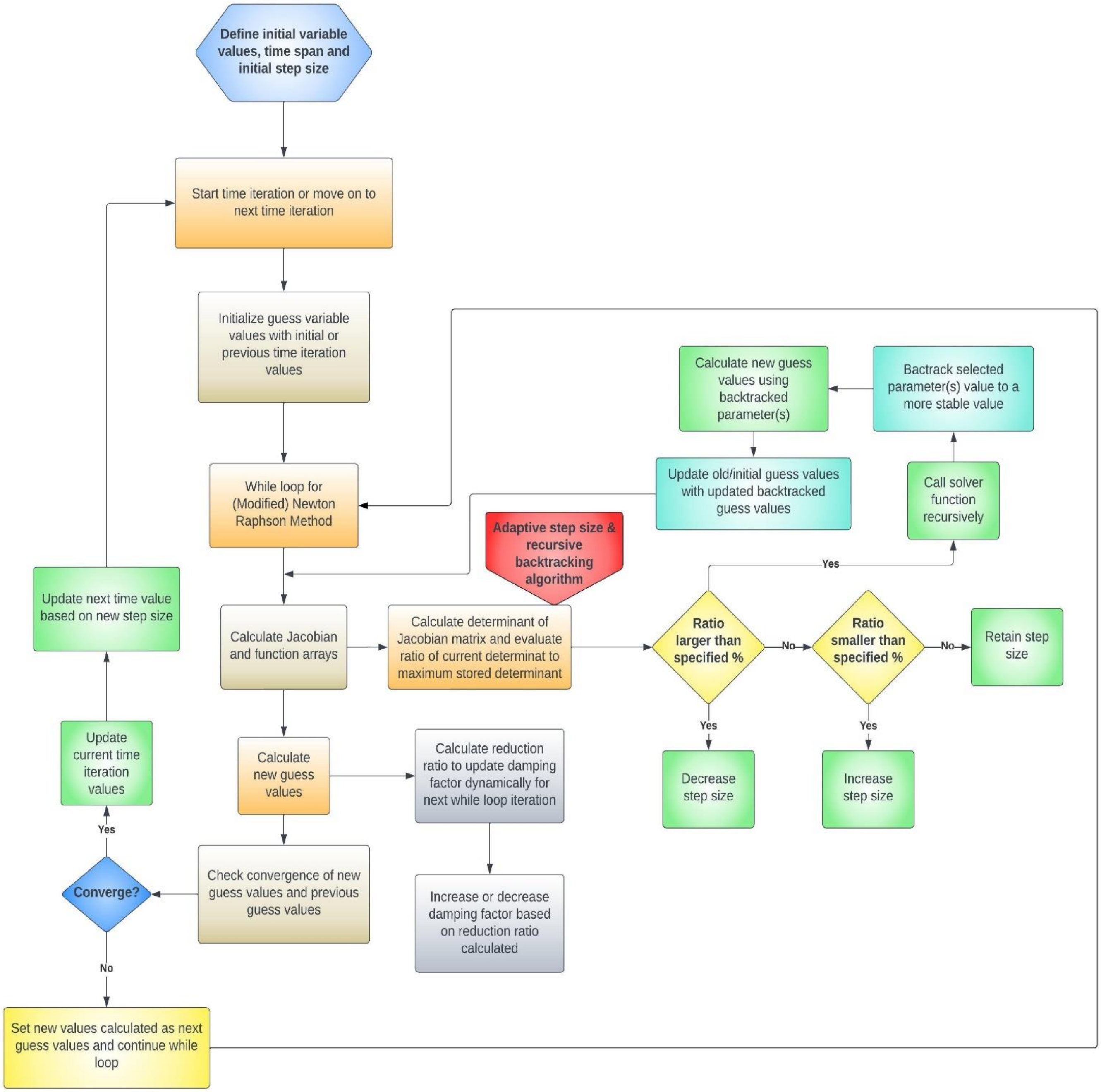
3) PARAMETER BACKTRACKING MODEL

The next model that was devised was the parameter backtracking model, for which the model from the previous adaptive step size is extended to include a backtracking recursive algorithm. This recursive solver call backtracks the solution (at unstable solution regions) to update the guess values with new guesses calculated using stable parameter values, hence labelled backtracking. The code snippet below shows how this new algorithm is used in conjunction with the adaptive step size model:

```python
# ==============================================================================
#     ### This will dynamically update the step size every iteration ###
# ==============================================================================
if i > 1: ## Only check after 1st iteration ##
    ## Continuously update the maximum value of determinant of Jacobian encountered
    max_jacobian = max(max_jacobian, abs(jacob_array[i-3]))
    ## Calculate the ratio of current determinant to maximum for Jacobian
    max_ratio = abs(jacob_array[i-2])/max_jacobian
    ## These values need configuration
    if max_ratio >= 1.2: ## This indicates step needs to be reduced and parameter backtracking
        ## Define recursive function call for parameter backtracking
        ## Recursive call to only solve for 1 iteration:
        t02 = 0
        t_end2 = t02 + h
        new_guess = LiS_Solver(s8guess, s4guess, s2guess, sguess, Vguess, spguess,
                               t_end2, h, I, break_voltage, state=state, t0=t02, backtracked=True,
                               params_backtrack=params_backtrack, upd_params=upd_params)

        ## Now update the guess values and run solver by updating u_array and jacobian
        x_upd = np.array([new_guess[0][-1], new_guess[1][-1], new_guess[2][-1],
                          new_guess[3][-1], new_guess[4][-1], new_guess[5][-1]])
        # Update the model
        model = LiSModel(x_upd, I)
        ## Change any values if a new value wants to be used apart from the default ones in the model class
        model.update_parameters(**upd_params)
        u_array = model.f(h, s8[-1], s4[-1], s2[-1], s[-1], V[-1], sp[-1])
        jacob = model.jacobian(h)
        x = x_upd ## Use updated guess values from backtracking

        # Update h (step-size)
        h_new = max(h*(0.2), min_h) ## Saturate at minimum step size

    elif max_ratio <= 1.0: ## This indicates step size can be increased
        h_new = min(h/(0.75), max_h) ## Saturate at maximum step size

    else:
        h_new = h

else:
    h_new = h
```

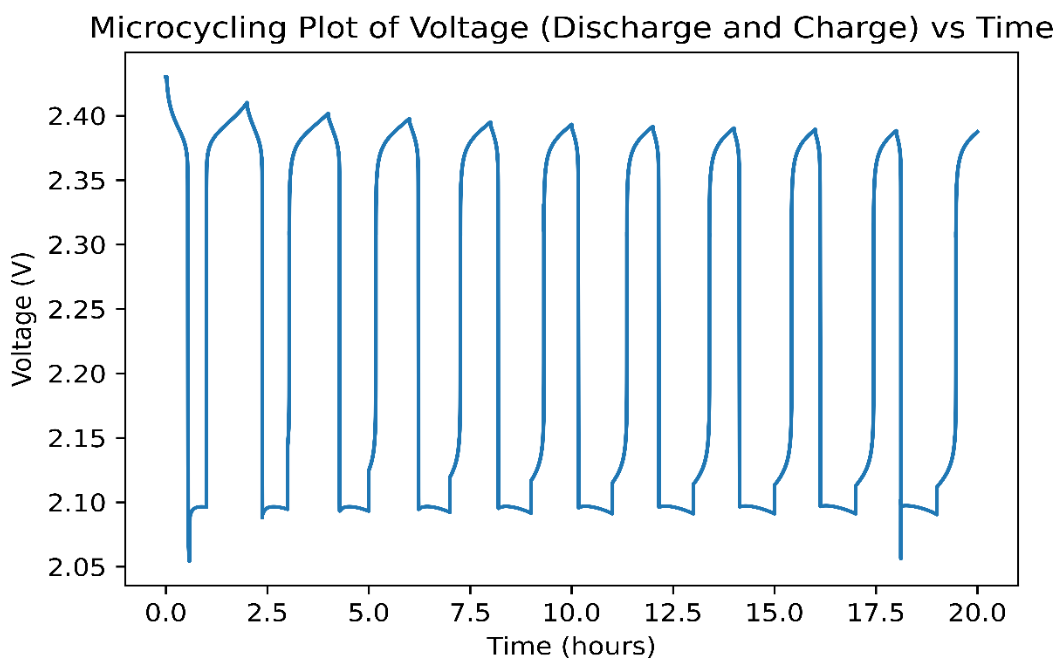
NOTE: This snippet is within the LiS_Solver function, which as can be seen is called recursively. This recursive call is ensured to only occur for a single time step. The backtracked parameter is used to recalculate the guess values and update the current guess values. Through this the model solver seems to achieve greater stability compared to the previous solver iterations.

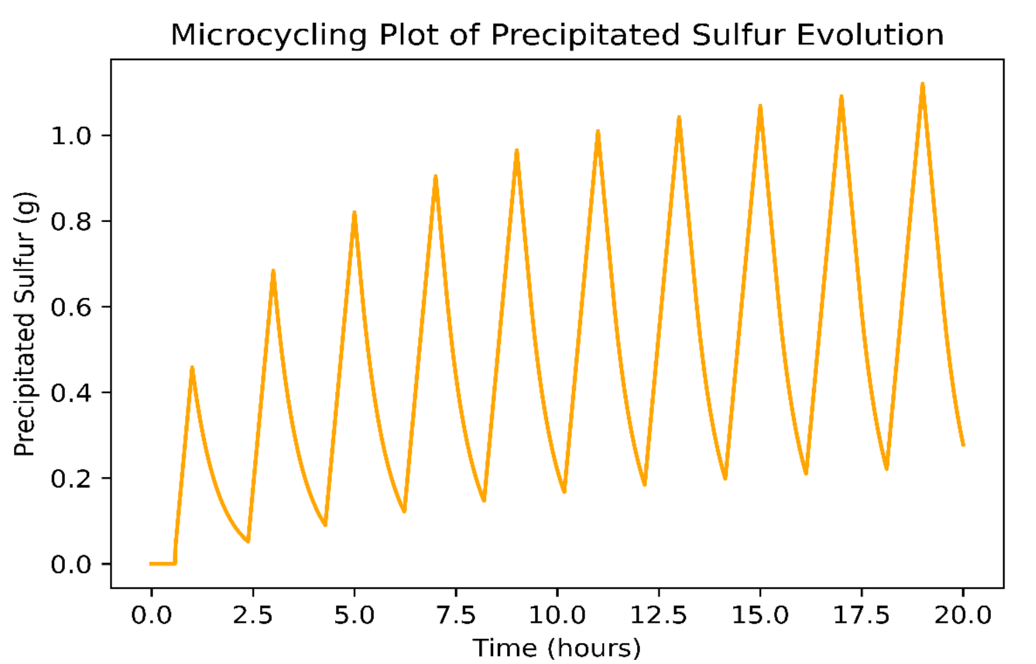
The flowchart in the next page shows the workflow of the improved solver via backtracking. As it can be seen that all the solver iterations are based on the base model with slight added complexity:



The new algorithm has no newly added hyper-parameters and only uses the previously defined hyper-parameters in the previous sections.

The plots below show the voltage evolution and precipitated sulfur evolution for micro-cycling over 10 cycles (20 hours) using the new parameter backtracking model (EH0=2.4, EL0=2.0):





APPENDIX

All the models discussed are formulated using the Backward-Euler method, the equations formed are expressed as functions, and these functions are differentiated partially with respect to its variables, the partial differentials are collected to form the Jacobian matrix and the model is then solved using the Newton-Raphson approach. Below are all the functions:

$$u1 = h\left(\left(\frac{(n_{s8} \cdot M_{s8} \cdot iH0 \cdot ar)}{ne \cdot F}\right)\left(\frac{e^{2 \cdot F \cdot (V_{j+1} - EH0)/(R \cdot T)}}{\sqrt{fh \cdot s_{8,j+1}}} \cdot s4_{j+1} - \frac{e^{2 \cdot F \cdot (EH0 - V_{j+1})/(R \cdot T)}}{s4_{j+1}} \cdot \sqrt{fh \cdot s_{8,j+1}}\right) - k_s \cdot s_{8,j+1}\right) - s_{8,j+1} + s_{8,j}$$

$$u2 = h\left(\left(-\frac{(n_{s8} \cdot M_{s8} \cdot iH0 \cdot ar)}{n_e \cdot F}\right)\left(\frac{e^{2 \cdot F \cdot (V_{j+1} - EH0)/(R \cdot T)}}{\sqrt{f_h \cdot s_{8,j+1}}} \cdot s_{4,j+1} - \frac{e^{2 \cdot F \cdot (EH0 - V_{j+1})/(R \cdot T)}}{s_{4,j+1}} \cdot \sqrt{f_h \cdot s_{8,j+1}}\right) + k_s \cdot s_{8,j+1} + \left(\frac{n_{s4} \cdot M_{s8} \cdot iL0 \cdot ar}{n_e \cdot F}\right)\left(\frac{e^{2 \cdot F \cdot (V_{j+1} - EL0)/(R \cdot T)}}{\sqrt{f_l \cdot s_{4,j+1}}} \cdot \sqrt{s_{2,j+1} \cdot (s_{g,j+1})^2} - \frac{e^{2 \cdot F \cdot (EL0 - V_{j+1})/(R \cdot T)}}{\sqrt{s_{2,j+1} \cdot (s_{g,j+1})^2}} \cdot \sqrt{f_l \cdot s_{4,j+1}}\right)\right) - s_{4,j+1} + s_{4,j}$$

$$u3 = h\left(\left(-\frac{(n_{s2} \cdot M_{s8} \cdot iL0 \cdot ar)}{n_e \cdot F}\right)\left(\frac{e^{2 \cdot F \cdot (V_{j+1} - EL0)/(R \cdot T)}}{\sqrt{f_l \cdot s_{4,j+1}}} \cdot \sqrt{s_{2,j+1} \cdot (s_{g,j+1})^2} - \frac{e^{2 \cdot F \cdot (EL0 - V_{j+1})/(R \cdot T)}}{\sqrt{s_{2,j+1} \cdot (s_{g,j+1})^2}} \cdot \sqrt{f_l \cdot s_{4,j+1}}\right)\right) - s_{2,j+1} + s_{2,j}$$

$$u4 = h\left(\left(-2 \cdot \frac{(n_s \cdot M_{s8} \cdot iL0 \cdot ar)}{n_e \cdot F}\right)\left(\frac{e^{2 \cdot F \cdot (V_{j+1} - EL0)/(R \cdot T)}}{\sqrt{f_l \cdot s_{4,j+1}}} \cdot \sqrt{s_{2,j+1} \cdot (s_{j+1})^2} - \frac{e^{2 \cdot F \cdot (EL0 - V_{j+1})/(R \cdot T)}}{\sqrt{s_{2,j+1} \cdot (s_{j+1})^2}} \cdot \sqrt{f_l \cdot s_{4,j+1}}\right) - \left(\frac{k_p \cdot s_{p,j+1}}{v \cdot \rho_s} \cdot (s_{j+1} - s_{sat})\right)\right) - s_{j+1} + s_j$$

$$u5 = I + iH0 \cdot ar\left(\frac{e^{2\cdot F\cdot(V_{j+1}-EH0)/(R\cdot T)}}{\sqrt{f_h \cdot s_{8,j+1}}} \cdot s_{4,j+1} - \frac{e^{2\cdot F\cdot(EH0-V_{j+1})/(R\cdot T)}}{s_{4,j+1}} \cdot \sqrt{f_h \cdot s_{8,j+1}}\right) + iL0 \cdot ar\left(\frac{e^{2\cdot F\cdot(V_{j+1}-EL0)/(R\cdot T)}}{\sqrt{f_l \cdot s_{4,j+1}}} \cdot \sqrt{s_{2,j+1} \cdot (s_{j+1})^2} - \frac{e^{2\cdot F\cdot(EL0-V_{j+1})/(R\cdot T)}}{\sqrt{s_{2,j+1} \cdot (s_{j+1})^2}} \cdot \sqrt{f_l \cdot s_{4,j+1}}\right)$$

$$u6 = \frac{h \cdot k_p \cdot s_{p,j+1}}{v \cdot \rho_s} \cdot (s_{j+1} - s_{\text{sat}}) - s_{p,j+1} + s_{p,j}$$

The equations are partially differentiated with respect to each of the (j+1) variables, the Jacobian is obtained and model solved using Newton-Raphson, as detailed below:

$$\begin{bmatrix} s_{8,j+1} \\ s_{4,j+1} \\ s_{2,j+1} \\ s_{j+1} \\ V_{j+1} \\ s_{p,j+1} \end{bmatrix}_{(n+1)} = \begin{bmatrix} s_{8,j+1} \\ s_{4,j+1} \\ s_{2,j+1} \\ s_{j+1} \\ V_{j+1} \\ s_{p,j+1} \end{bmatrix}_{(n)} - \begin{bmatrix} \frac{\partial u1}{\partial s_{8,j+1}} & \frac{\partial u1}{\partial s_{4,j+1}} & 0 & 0 & \frac{\partial u1}{\partial V_{j+1}} & 0 \\ \frac{\partial u2}{\partial s_{8,j+1}} & \frac{\partial u2}{\partial s_{4,j+1}} & \frac{\partial u2}{\partial s_{2,j+1}} & \frac{\partial u2}{\partial s_{j+1}} & \frac{\partial u2}{\partial V_{j+1}} & 0 \\ 0 & \frac{\partial u3}{\partial s_{4,j+1}} & \frac{\partial u3}{\partial s_{2,j+1}} & \frac{\partial u3}{\partial s_{j+1}} & \frac{\partial u3}{\partial V_{j+1}} & 0 \\ 0 & \frac{\partial u4}{\partial s_{4,j+1}} & \frac{\partial u4}{\partial s_{2,j+1}} & \frac{\partial u4}{\partial s_{j+1}} & \frac{\partial u4}{\partial V_{j+1}} & \frac{\partial u4}{\partial s_{p,j+1}} \\ \frac{\partial u5}{\partial s_{8,j+1}} & \frac{\partial u5}{\partial s_{4,j+1}} & \frac{\partial u5}{\partial s_{2,j+1}} & \frac{\partial u5}{\partial s_{j+1}} & \frac{\partial u5}{\partial V_{j+1}} & 0 \\ 0 & 0 & 0 & \frac{\partial u6}{\partial s_{j+1}} & 0 & \frac{\partial u6}{\partial s_{p,j+1}} \end{bmatrix}_{(n)}^{-1} \cdot \begin{bmatrix} u1 \\ u2 \\ u3 \\ u4 \\ u5 \\ u6 \end{bmatrix}_{(n)}$$

The (n) terms represent the n^{th} guess values and the (n+1) terms represents the next guess values or new values if converged.