Equivalent circuit models (ECM): A dynamic programming approach to battery charging

The purpose of this document is to elucidate the governing electrical equations used in the equivalent circuit model (ECM) for both continuous and dynamic programming (DP) frameworks. Namely, we investigate the OCV-R and OCVR-RC ECM models. In both cases, the current I is the controllable input. OCV-R has a single state: state-of-charge (SOC), and the OCVR-RC adds the capacitor voltage as a second state. Parameter values are taken from a A123 2300 mAh Li-ion battery using an equilibrium SOC of 0.5, and temperature of 25 °C. Our optimization function minimizes the least squares error of the SOC to the target SOC.

Dynamic programming approach

The dynamic programming (DP) framework converts the optimization problem into a discrete form. We evoke Bellman's Principal of Optimality, i.e.,:

Let V_k represent the value function for the minimum least squares error (LSQ) problem at each time step $k \in \{0, 1, .., N-1\}$:

$$\boxed{V_k = \min_{I_k} \left\{ \left(z_k - z_{target}\right)^2 + V_{k+1} \right\} + V_N} \tag{1}$$

$$V_N = \left(z_N - z_{target}\right)^2$$

We solve this objective by identifying optimal control I_k^* for all k. We utilize a forward finite difference method for discretization of the continuous state dynamics, with $\Delta t = 1$. We now break down the dynamic programming formulation of each model.

Model 1: OCV-R

Model description

This model is the simplest circuit model using a voltage source V_{oc} and a resistor R_0. We have a single state: the state-of-charge of the battery, hereafter denoted as z. V_{oc} depends on z, and for the purpose of this report we use the data from an A123 2300 mAh battery.

Dynamics

First we understand the state equation for SOC evolution as a function of the current:

$$\frac{dz}{dt} = \frac{I}{C_{\text{batt}}}$$

We now convert the single continuous state equation to the discrete form.

$$\frac{z_{k+1} - z_k}{\Delta t} = \frac{I_k}{dt}$$

$$\boxed{z_{k+1} = \frac{I_k}{C_{\text{batt}}} \cdot \Delta t + z_k} \tag{2}$$

Constraints

We present the constraints for the state and control separately:

$$z_{min} \leq z_k \leq z_{max}$$

$$V_{t,min} \leq V_{t,k} \leq V_{t,max}$$

$$I_{min} \leq I_k \leq I_{max}$$

We rewrite the state constraint given the time-stepping dynamics:

$$z_{min} \leq z_{k+1} - \frac{I_k}{C_{\text{batt}}} \cdot \Delta t \leq z_{max}$$

We convert this form into a bound on the control I_k:

$$C_{\text{batt}} \cdot \frac{z_{k+1} - z_{\text{max}}}{\Delta t} \leq I_k \leq C_{\text{batt}} \cdot \frac{z_{k+1} - z_{\text{min}}}{\Delta t}$$

We now rewrite the terminal voltage constraint also as a bound on I_k:

$$V_{t,min} \leq V_{oc} + I_k R_0 \leq V_{t,max}$$

$$\frac{V_{t,min} - V_{oc}}{R_0} \leq I_k \leq \frac{V_{t,max} - V_{oc}}{R_0}$$

Combining the constraints on I_k, we can tighten the bounds with the following expression:

$$\max\left\{C_{\text{batt}} \cdot \tfrac{z_{k+1} - z_{\text{max}}}{\Delta t}, \tfrac{V_{t,\text{min}} - V_{oc}}{R_0}, I_{\text{min}}\right\} \leq I_k \leq \min\left\{C_{\text{batt}} \cdot \tfrac{z_{k+1} - z_{\text{min}}}{\Delta t}, \tfrac{V_{t,\text{max}} - V_{oc}}{R_0}, I_{\text{max}}\right\} \tag{3}$$

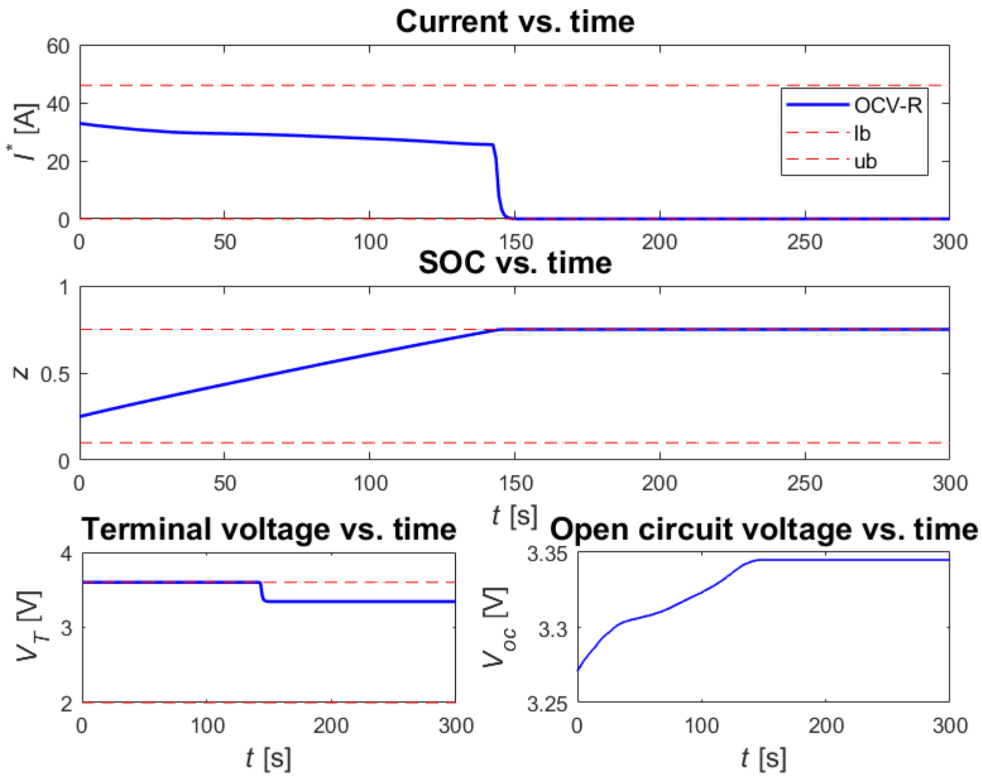
For notation purposes, hereafter we will note the state of charge lower/upper bounds as z_{lb}, z_{ub} and the terminal voltage lower/upper bounds as $V_{t,lb}, V_{t,ub}$ respectively. We can rewrite Equation 3 with an easier framework:

$$\boxed{\max\left\{z_{lb}, V_{t,lb}, I_{\text{min}}\right\} \leq I_k \leq \min\left\{z_{ub}, V_{t,ub}, I_{\text{max}}\right\}} \tag{4}$$

Results

We present the OCV-R solution with a target SOC of 0.75 and initial SOC of 0.25[1]. We note that the optimal policy is constant voltage (CV). The maximum permissible current is drawn so as to charge the battery as quickly as possible while operating at the upper terminal voltage limit. Once the target SOC constraint is reached, the circuit no longer draws current and the terminal voltage decreases slightly. The target SOC is reached around 148s with a 300s charging window. This result is corroborate for the Python solution.

[1] For a full list of the other parameters, please refer to the Appendix


Current vs. time
I^* [A]
OCV-R
lb
ub
SOC vs. time
z
t [s]
Terminal voltage vs. time
V_T [V]
t [s]
Open circuit voltage vs. time
V_{oc} [V]
t [s]

Figures/OCVR_py.png

Model 2: OCV-R-RC

Model description

This model adds an RC pair to the OCV-R model. The capacitor contributes a time-varying voltage V_1. We now have two states and one control.

Dynamics

The SOC dynamics remain the same. We now describe the state dynamics for the capacitor voltage with the continuous equation:

$$C_1 \frac{dV_1}{dt} = -\frac{V_1}{R_1} + I$$

We use the finite difference method to convert the continuous form to a dynamic one.

$$C_1 \frac{V_{1,k+1} - V_{1,k}}{\Delta t} = -\frac{V_{1,k}}{R_1} + I_k$$

Rewriting,

$$\boxed{V_{1,k+1} = V_{1,k}\left(1 - \frac{\Delta t}{R_1 \cdot C_1}\right) + \frac{I_k \cdot \Delta t}{C_1}} \tag{5}$$

State space

We described the state space for z in the OCV-R model. We consider the minimum voltage for the capacitor. We assume we charge from rest, and as we are only interested in the charging problem, we take $\boxed{V_{1,\min} = 0}$. The maximum capacitor voltage comes from the steady-state solution to: $C_1 \cdot 0 = -\frac{V_1}{R_1} + I$. Solving, we get $\boxed{V_{1,\max} = I_{\max} \cdot R_0}$

Constraints

We retain the SOC constraints z_{lb}, z_{ub} from before, in addition to the current constraints $I_{\min}, I_{\max}$. The terminal voltage constraint only changes slightly:

$$V_{t,min} \leq V_{oc} + V_{1,k} + I_k R_0 \leq V_{t,max}$$

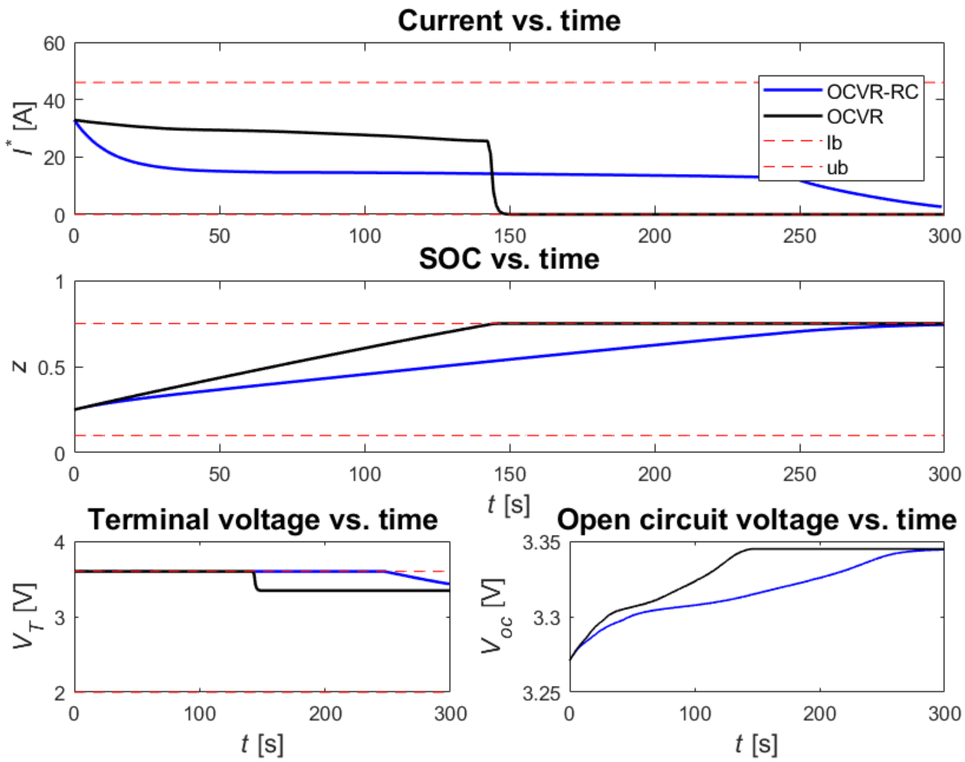
$$\frac{V_{t,min} - V_{oc} - V_{1,k}}{R_0} \leq I_k \leq \frac{V_{t,max} - V_{oc} - V_{1,k}}{R_0}$$

We use this updated framework for $V_{1,lb}, V_{1,ub}$. Our control constraints remain the same as described in Equation 4. For completeness, the full version is included below:

$$\max\left\{C_{\text{batt}} \cdot \frac{z_{k+1} - z_{\max}}{\Delta t}, \frac{V_{t,\min} - V_{1,k} - V_{oc}}{R_0}, I_{\min}\right\} \leq I_k \leq \min\left\{C_{\text{batt}} \cdot \frac{z_{k+1} - z_{\min}}{\Delta t}, \frac{V_{t,\max} - V_{1,k} - V_{oc}}{R_0}, I_{\max}\right\} \tag{6}$$

Results

We present the OCVR-RC solution with a target SOC of 0.75 and initial SOC of 0.25. We note that the optimal policy is constant voltage (CV). The maximum permissible current is drawn so as to charge the battery as quickly as possible while operating at the upper terminal voltage limit. Once the target SOC constraint is reached, the circuit no longer draws current and the terminal voltage decreases slightly. The target SOC is reached around 148s with a 300s charging window.



Appendix

This appendix includes a list of the parameters used in the DP formulation and the original code from MATLAB and Python used.

Figure A1. MATLAB Code for OCV-R dynamic programming formulation

```
%% OCV-R
%   DP formulation for the min OCV-R SOC error problem
%   Raja Selvakumar
%   07/10/2018
%   energy, Controls, and Application Lab (eCAL)

clc; clear;
%% OCV save
Rc = 1.94;
Ru = 3.08;
Cc = 62.7;
Cs = 4.5;
z_0 = 0.25;
T_inf = 25;
t_0 = 0;
C_1 = 2500;
C_2 = 5.5;
R_0 = 0.01;
R_1 = 0.01;
R_2 = 0.02;
I_min = 0;
```

Table 1: List of parameters used in DP formulation

Parameter	Value	Units
$z_{\min}$	0.1	[-]
$z_{\max}$	0.95	[-]
z_{target}	0.75	[-]
$V_{t,\min}$	2	[V]
$V_{t,\max}$	3.6	[V]
$I_{\min}$	0	[A]
$I_{\max}$	46	[A]
z_{eq}	0.5	[-]
T_{∞}	25	[°C]
Δt	1	[s]
R_0	0.01	[Ω]
C_{batt}	8280	[coulombs]
C_1	2500	[F]
R_1	0.01	[Ω]
$V_{1,\min}$	0	[V]
$V_{1,\max}$	0.46	[V]

```
I_max = 46;
V_min = 2;
V_max = 3.6;
z_min = 0.1;
z_max = 0.75;
C_batt = 2.3*3600;
t_max = 5*60;
dt = 1;
save ECM_params.mat;
%% Voc save
VOC_data = csvread('Voc.dat',1,0);
soc = VOC_data(:,1);
voc = VOC_data(:,2);
save OCV_params.mat;
%% Load data
clc; clear;
load ECM_params.mat;
load OCV_params.mat;
fs = 15;
clear VOC VOC_data;
%% Playground
%% Grid State and Preallocate
SOC_grid = (z_min:0.005:z_max)';
ns = length(SOC_grid);  % #states
N = (t_max-t_0)/dt; % #iterations
V = inf*ones(ns,N+1); % #value function
u_star = zeros(ns,N);% #control
%% Solve DP
tic;
```

```
for i=1:ns
    V(i,N+1) = (SOC_grid(i)-z_max)^2; %Bellman terminal boundary condition
end

for k = N:-1:1 %time
    for idx = 1:ns %state (SOC)
        c_soc = SOC_grid(idx);
        c_voc = voc(soc==round(c_soc,3)); %return the voc value when soc = c_soc

        % Bounds
        lb = max([I_min, C_batt/dt*(c_soc-z_max),(V_min-c_voc)/R_0]);
        ub = min([I_max, C_batt/dt*(c_soc-z_min),(V_max-c_voc)/R_0]);

        % Control grid
        I_grid = linspace(lb,ub,200)';
        % Cost-per-time-step
        cv = ones(length(I_grid),1).*abs(c_voc-V_max) + I_grid.*R_0;
        %g_k = dt.*I_grid-cv;
        g_k = -(c_soc-z_max)^2;

        % State dynamics
        SOC_nxt = c_soc+ dt/C_batt.*I_grid;

        % Linear interpolation for value function
        V_nxt = interp1(SOC_grid,V(:,k+1),SOC_nxt,'linear');
        % Bellman
        [V(idx, k), ind] = min(-g_k + V_nxt);

        % Save Optimal Control
        u_star(idx,k) = I_grid(ind);
    end
end

solveTime = toc;
clc;
fprintf(1,'DP Solver Time %2.2f sec \n',solveTime);

%% Simulate Results

% Preallocate
SOC_sim = zeros(N,1);
I_sim = zeros(N,1);
V_sim = zeros(N,1);
Voc_sim = zeros(N,1);

% Initialize
SOC_sim(1) = z_0;

% Simulate Battery Dynamics
for k = 1:N
    % Calculate optimal control for given state
    I_sim(k) = interp1(SOC_grid,u_star(:,k),SOC_sim(k),'linear');
```

```
    % Voc, terminal voltage
    Voc_sim(k) = voc(soc==round(SOC_sim(k),3));
    V_sim(k) = Voc_sim(k) + I_sim(k).*R_0;

    % SOC dynamics
    SOC_sim(k+1) = SOC_sim(k) + dt/C_batt.*I_sim(k);
end
```

Figure A2. MATLAB Code for OCV-R-RC dynamic programming formulation

```
%% OCV-R-RC
%   DP formulation for the min OCV-R-RC SOC error problem
%   Raja Selvakumar
%   07/12/2018
%   energy, Controls, and Application Lab (eCAL)

clc; clear;
%% OCV save
Rc = 1.94;
Ru = 3.08;
Cc = 62.7;
Cs = 4.5;
z_0 = 0.25;
T_inf = 25;
t_0 = 0;
C_1 = 2500;
C_2 = 5.5;
R_0 = 0.01;
R_1 = 0.01;
R_2 = 0.02;
I_min = 0;
I_max = 46;
V_min = 2;
V_max = 3.6;
z_min = 0.1;
z_max = 0.95;
z_target = 0.75;
C_batt = 2.3*3600;
t_max = 300;
dt = 1;
beta = 0.5;
save ECM_params.mat;
%% Voc save
clear;
VOC_data = csvread('Voc.dat',1,0);
soc = VOC_data(:,1);
voc = VOC_data(:,2);
save OCV_params.mat;
%% Load data
clc; clear;
load ECM_params.mat;
load OCV_params.mat;
load OCVRRC.mat;
fs = 15;
clear VOC VOC_data;
%% Playground
%% Grid State and Preallocate
SOC_grid = (z_min:0.01:z_max)';
V1_min = 0;
V1_max = I_max*R_0;
V1_grid = (V1_min:0.01:V1_max)';
```

```
ns = [length(SOC_grid) length(V1_grid)];  % #states
N = (t_max-t_0)/dt; % #iterations

V = inf*ones(N,ns(1),ns(2)); % #value function
u_star = nan*ones(N-1,ns(1),ns(2));% #control
%% Solve DP
clc;
tic;
for i=1:ns(1)
    V(end,i,:) = (1-beta)*(SOC_grid(i)-z_target)^2; %terminal boundary condition
end

for k = (N-1):-1:1 %time
    if mod(k,10)==0
        fprintf(1,'Computing Principle of Optimality at %3.0f sec\n',k*dt);
    end
    for ii = 1:ns(1) %SOC
        for jj=1:ns(2) %V_1
            c_soc = SOC_grid(ii);
            c_v1 = V1_grid(jj);
            c_voc = voc(soc==round(c_soc,3)); %return voc when soc = c_soc

            % Bounds
            z_lb = C_batt/dt*(c_soc-z_max);
            z_ub = C_batt/dt*(c_soc-z_min);
            V_lb = (V_min-c_voc-c_v1)/R_0;
            V_ub = (V_max-c_voc-c_v1)/R_0;
            lb = max([I_min,z_lb,V_lb]);
            ub = min([I_max,z_ub,V_ub]);

            % Control grid
            I_grid = linspace(lb,ub,200)';

            % Cost-per-time-step
            %cv = ones(length(I_grid),1).*abs(c_voc-V_max) + I_grid.*R_0;
            %g_k = dt.*I_grid-cv;
            g_k = -beta*(c_soc-z_target)^2;

            % State dynamics
            SOC_nxt = c_soc+ dt/C_batt.*I_grid;
            V1_nxt = c_v1*(1-dt/(R_1*C_1))+dt/C_1.*I_grid;

            % Linear interpolation for value function
            V_nxt = interp2(SOC_grid,V1_grid,squeeze(V(k+1,:,:))',SOC_nxt,...
                V1_nxt,'linear');
            [V(k,ii,jj), ind] = min(-g_k + V_nxt); %Bellman

            % Save Optimal Control
            u_star(k,ii,jj) = I_grid(ind);
        end
    end
end

```

```
solveTime = toc;
fprintf(1,'DP Solver Time %2.0f min %2.0f sec \n',floor(solveTime/60),mod(solveTime,60));

%% Simulate Results

% Preallocate
SOC_sim = nan*ones(N,1);
I_sim = nan*ones(N-1,1);
V_sim = nan*ones(N-1,1);
V1_sim = nan*ones(N,1);
Voc_sim = nan*ones(N-1,1);

% Initialize
SOC_sim(1) = z_0;
V1_sim(1) = 0;

% Simulate Battery Dynamics
tic;
for k = 1:N-1
    if mod(k,10)==0
        fprintf(1,'Simulating at %3.0f sec\n',k*dt);
    end
    % Calculate optimal control for given state
    I_sim(k) = interp2(SOC_grid,V1_grid,squeeze(u_star(k,:,:))',SOC_sim(k),...
        V1_sim(k),'linear');

    % Voc, terminal voltage
    Voc_sim(k) = voc(soc==round(SOC_sim(k),3));
    V_sim(k) = Voc_sim(k) + V1_sim(k) + I_sim(k).*R_0;

    % Dynamics
    SOC_sim(k+1) = SOC_sim(k) + dt/C_batt.*I_sim(k);
    V1_sim(k+1) = V1_sim(k)*(1-dt/(R_1*C_1)) + dt/C_1.*I_sim(k);
end
solveTime = toc;
```

Figure A3. Python Code for OCV-R dynamic programming formulation

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jun 25 11:57:24 2018

@author: rselvak6
"""
import numpy as np
import time
import math
import matplotlib.pyplot as plt

#hyperparameters
Rc = 1.94 #K/W
Ru = 3.08
Cc = 62.7 #J/K
Cs = 4.5

#initial conditions
SOC_o = 0.5
T_inf = 25 #C
t_0 = 0 #sec

#Values taken from Perez et. al 2015 based on initial conditions
C_1 = 2500 #F
C_2 = 5.5
R_0 = 0.01 #Ohms
R_1 = 0.01
R_2 = 0.02
I_min = 0 #A
I_max = 46
V_min = 2 #V
V_max = 3.6
SOC_min = 0.1
SOC_max = 0.75
C_batt = 2.3*3600;

#Voc from file
V_oc = np.loadtxt('Voc.dat',delimiter=',',dtype=float)

#free parameters
t_max = 5*60
dt = 1
N = t_max-t_0
#%%Playground
#%% Preallocate grid space
#state grids
SOC_grid = np.arange(SOC_min,SOC_max+0.005,0.005)
num_states = len(SOC_grid)

#control and utility
V = np.ones((num_states,N+1))*math.inf
I_opt = np.zeros((num_states,N))
```

```
#terminal Bellman
V[:,N] = [(SOC_grid[k]-SOC_max)**2 for k in range(0,num_states)]

#%% DP
def DP():
    start = time.time()
    for k in range(t_max-1,t_0-1,-dt):
        for idx in range(0,num_states):

            #lower/upper control bounds
            v_oc = [V_oc[x,1] for x in range(0,len(V_oc[:,0])-1)
                   if V_oc[x,0]==round(SOC_grid[idx],3)][0]
            lb = max(I_min, C_batt/dt*(SOC_grid[idx]-SOC_max), (V_min-v_oc)/R_0)
            ub = min(I_max, C_batt/dt*(SOC_grid[idx]-SOC_min), (V_max-v_oc)/R_0)

            #control initialization
            I_grid = np.linspace(lb,ub,200)

            #value function
            c_k = (SOC_grid[idx]-SOC_max)**2

            #iterate next SOC
            SOC_nxt = SOC_grid[idx] + I_grid/C_batt*dt

            #value function interpolation
            V_nxt = np.interp(SOC_nxt,SOC_grid,V[:,k+1])

            #Bellman
            V[idx,k] = min(c_k+V_nxt)
            ind = np.argmin(c_k+V_nxt)

            #save optimal control
            I_opt[idx,k] = I_grid[ind]
    end = time.time()
    print('DP solver time: %2.2f[s]\n',end-start)
    return I_opt
ret = DP()
#%% Simulation: Initialize
t_sim = range(0,N)
SOC_sim = np.zeros(N)
Vt_sim = np.zeros(N)
Voc_sim = np.zeros(N)
I_sim = np.zeros(N)

I_ub = [I_max]*N
I_lb = [I_min]*N
z_ub = [SOC_max]*N
z_lb = [SOC_min]*N
v_lb = [V_min]*N
v_ub = [V_max]*N

#initial condition
```

```
SOC_sim[0] = 0.25
 #%% Simulation: Simulate

for k in range(0,(N-1)):
    I_sim[k] = np.interp(SOC_sim[k],SOC_grid,ret[:,k])
    SOC_sim[k+1] = SOC_sim[k]+I_sim[k]*dt/C_batt
    Voc_sim[k] = [V_oc[x,1] for x in range(0,len(V_oc[:,0])-1)
                 if V_oc[x,0]==round(SOC_sim[k],3)][0]
    Vt_sim[k] = Voc_sim[k] + I_sim[k]*R_0
```

Figure A4. Python Code for OCV-R-RC dynamic programming formulation

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 31 16:53:18 2018

@author: Raja Selvakumar
"""

import numpy as np
from scipy import interpolate as ip
import time
import math
import matplotlib.pyplot as plt

#hyperparameters
Rc = 1.94 #K/W
Ru = 3.08
Cc = 62.7 #J/K
Cs = 4.5

#initial conditions
SOC_o = 0.5
T_inf = 25 #C
t_0 = 0 #sec

#Values taken from Perez et. al 2015 based on initial conditions
C_1 = 2500 #F
C_2 = 5.5
R_0 = 0.01 #Ohms
R_1 = 0.01
R_2 = 0.02
I_min = 0 #A
I_max = 46
V_min = 2 #V
V_max = 3.6
SOC_min = 0.1
SOC_max = 0.75
C_batt = 2.3*3600;

#Voc from file
V_oc = np.loadtxt('Voc.dat',delimiter=',',dtype=float)

#free parameters
```

```
t_max = 1*20
dt = 1
N = int((t_max-t_0)/dt)
#%%Playground
#%% Preallocate grid space
#state grids
SOC_grid = np.arange(SOC_min,SOC_max+0.05,0.05)
V1_grid = np.arange(0,I_max*R_0+0.1,0.1)
n1 = len(SOC_grid)
n2 = len(V1_grid)

#control and utility
V = np.ones((N+1,n1,n2))*math.inf
I_opt = np.zeros((N,n1,n2))

#terminal Bellman
for i in range(0,n1):
    V[N,i,:] = (SOC_grid[i]-SOC_max)**2
#%% DP
start = time.time()
for k in range(N-1,t_0-1,-dt):
    if(k%10==0):
        print("Computing the Principle of Optimality at %.0f s" % (k*dt))
    for ii in range(0,n1):
        for jj in range(0,n2):

            c_soc = SOC_grid[ii]
            c_v1 = V1_grid[jj]

            #lower/upper control bounds
            v_oc = [V_oc[x,1] for x in range(0,len(V_oc[:,0])-1)
                   if V_oc[x,0]==round(c_soc,3)][0]
            I_vec = np.linspace(I_min,I_max,200)

            z_nxt_test = c_soc + dt/C_batt*I_vec
            V_nxt_test = v_oc + c_v1 + I_vec*R_0
            ind = np.argmin((z_nxt_test-SOC_min >= 0) &
                            (SOC_max-z_nxt_test >= 0) & (V_nxt_test-V_min >= 0)
                            & (V_max-V_nxt_test >= 0))

            #value function
            c_k = (c_soc-SOC_max)**2

            #iterate next SOC
            SOC_nxt = c_soc + I_vec[ind]/C_batt*dt
            V1_nxt = c_v1*(1-dt/(R_1*C_1))+dt/C_1*I_vec[ind]

            S_mesh, V_mesh = np.meshgrid(SOC_grid,V1_grid)
            #value function interpolation
            z = ip.interp2d(S_mesh,V_mesh,np.squeeze(V[k+1,:,:]).T)
            V_nxt = z(SOC_grid[int(n1/2)], V1_grid[int(n2/2)])

            #Bellman
```

```
            V[k,ii,jj] = min(c_k + V_nxt)
            ind2 = np.argmin(c_k+V_nxt)

            #save optimal control
            I_opt[k,ii,jj] = I_vec[ind2]
end = time.time()
print("DP solver time: %.2f s" % (end-start))
#%% Simulation: Initialize
t_sim = range(0,N)
SOC_sim = np.zeros(N)
Vt_sim = np.zeros(N)
Voc_sim = np.zeros(N)
V1_sim = np.zeros(N)
I_sim = np.zeros(N)

I_ub = [I_max]*N
I_lb = [I_min]*N
z_ub = [SOC_max]*N
z_lb = [SOC_min]*N
v_lb = [V_min]*N
v_ub = [V_max]*N

#initial condition
SOC_sim[0] = 0.25
V1_sim[0] = 0
 #%% Simulation: Simulate

for k in range(0,(N-1)):
    #Control
    if(k%10==0):
        print("Simulating results at %.0f s" % (k*dt))
    S_mesh, V_mesh = np.meshgrid(SOC_grid,V1_grid)
    z = ip.interp2d(S_mesh[0,:],V_mesh[:,0],
                    np.squeeze(I_opt[k+1,:,:]).T,kind='cubic')
    I_sim[k] = z(SOC_sim[k],V1_sim[k])

    Voc_sim[k] = [V_oc[x,1] for x in range(0,len(V_oc[:,0])-1)
                 if V_oc[x,0]==round(SOC_sim[k],3)][0]
    Vt_sim[k] = Voc_sim[k] + V1_sim[k] + I_sim[k]*R_0

    #Dynamics
    SOC_sim[k+1] = SOC_sim[k]+I_sim[k]*dt/C_batt
    V1_sim[k+1] = V1_sim[k]*(1-dt/(R_1*C_1)) + dt/C_1*I_sim[k]
```