Simulating Battery Packs

2.1: Modeling approach #1: Equivalent-circuit models

- One function of a battery management system is to compute estimates of a number of fundamental quantities, including:
  - State of charge, state of health, state of life,
  - Available power, available energy / range.
- The best methods require high-fidelity but computationally simple math models of cell *input/output* (current/ voltage) dynamics.
- We believe that future applications will also require insight into cell *internal* dynamics (*e.g.*, to predict and minimize degradation).

Equivalent-circuit models (ECMs):

- Amounts to an empirical curve fit that interpolates between data seen when fitting model (extrapolation not reliable).
- Can predict input/output (current/voltage) behavior only, not internal electro-chemical states.
- *But, yields fast, robust simulations.*

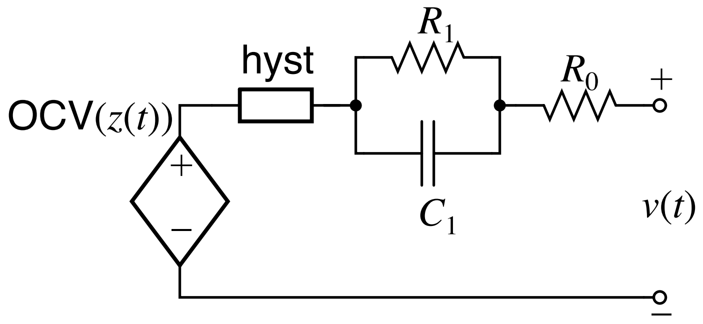
Physics-based models (PBMs):

- *Can predict over wide range of operating conditions.*
- *Can predict internal state of cell (useful for aging prediction).*
- But, slow simulations having robustness/convergence issues.

- We studied both types of models in ECE5710, and also saw how to compute a reduced-order PBM of similar complexity to an ECM.
- We first review these two types of models in this chapter of notes.
- We then apply them to simulating battery packs.

Modeling approach #1: Empirical

- Current/voltage behavior of lithium-ion cell can appear very simple, and is often well approximated by an equivalent circuit:
  - Used extensively in real-time control algorithms.
- The (continuous-time) "enhanced self-correcting" (ESC) cell model equivalent circuit is drawn to the right, modeling a single diffusion voltage via a single R–C pair.



- **Enhanced**: Model includes a (poor) description of hysteresis (which is still better than not including it at all).
- **Self correcting**: Model voltage converges to OCV + hysteresis on rest, and to OCV + hysteresis $-\, i \sum R$ on constant current event.
- R_0, C_1, and R_1 are analogs of physical properties of diffusion processes, but do not directly describe something physical.
- Values of R_0, C_1, and R_1 are chosen to make model fit cell-test data:
  - Values do not come from direct physical measurements.
  - Parameters are typically a function of SOC and temperature.
- The ESC model comprises the following terms:

- State-of-charge: $z_{k+1} = z_k - \eta_k i_k \Delta t / Q$.
- Diffusion resistor current:

$$i_{R_1,k+1} = \exp\left(\frac{-\Delta t}{R_1C_1}\right) i_{R_1,k} + \left(1 - \exp\left(\frac{-\Delta t}{R_1C_1}\right)\right) i_k.$$

- Hysteresis voltage:

$$h_{k+1} = \exp\left(-\left|\frac{\eta_k i_k \gamma \Delta t}{Q}\right|\right) h_k + \left(1 - \exp\left(-\left|\frac{\eta_k i_k \gamma \Delta t}{Q}\right|\right)\right) \operatorname{sgn}(i_k).$$

- Note that the circuit can contain more than a single parallel resistor-capacitor pair. We can define vector valued

$$i_{R,k+1} = \underbrace{\begin{bmatrix} \exp\left(\frac{-\Delta t}{R_1C_1}\right) & 0 & \cdots \\ 0 & \exp\left(\frac{-\Delta t}{R_2C_2}\right) & \\ \vdots & & \ddots \end{bmatrix}}_{A_{RC}} i_{R,k} + \underbrace{\begin{bmatrix} \left(1-\exp\left(\frac{-\Delta t}{R_1C_1}\right)\right) \\ \left(1-\exp\left(\frac{-\Delta t}{R_2C_2}\right)\right) \\ \vdots \end{bmatrix}}_{B_{RC}} i_k.$$

- Then, if we define $A_{H_k} = \exp\left(-\left|\frac{\eta_k i_k \gamma \Delta t}{Q}\right|\right)$,

$$\begin{bmatrix} z_{k+1} \\ i_{R,k+1} \\ h_{k+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & A_{RC} & 0 \\ 0 & 0 & A_{H_k} \end{bmatrix}}_{A(i_k)} \begin{bmatrix} z_k \\ i_{R,k} \\ h_k \end{bmatrix} + \underbrace{\begin{bmatrix} -\frac{\eta_k \Delta t}{Q} & 0 \\ B_{RC} & 0 \\ 0 & (1-A_{H_k}) \end{bmatrix}}_{B(i_k)} \begin{bmatrix} i_k \\ \operatorname{sgn}(i_k) \end{bmatrix}.$$

- If we define $x_k = \begin{bmatrix} z_k & i_{R_k}^T & h_k \end{bmatrix}^T$, then we have

$$x_{k+1} = A(i_k)x_k + B(i_k)i_k.$$

- This is the ESC "state equation", and describes all dynamic effects.
- The "output equation" is

$$v_k = \text{OCV}(z_k) + M h_k - \sum_i R_i i_{R_i,k} - R_0 i_k.$$

- We can define $C = \begin{bmatrix} 0, & -R_1, & -R_2, & \ldots & M \end{bmatrix}$ and $D = -R_0$ to arrive at

$$v_k = \text{OCV}(z_k) + Cx_k + Di_k.$$

- So, we conclude that the ESC model looks similar to, but not identical to, a linear state-space system of the form

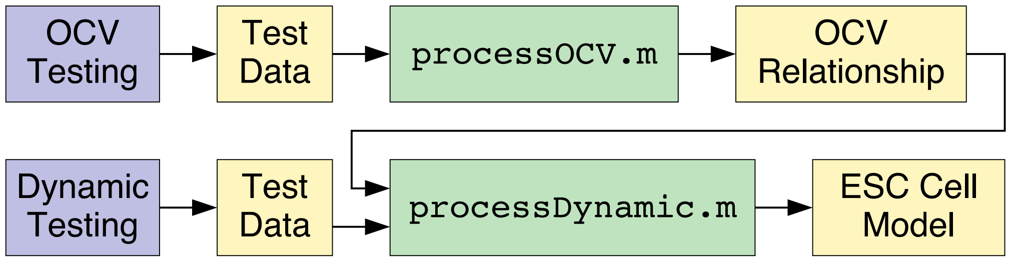
$$x_{k+1} = A_k x_k + B_k i_k$$

$$y_k = C_k x_k + D_k i_k.$$

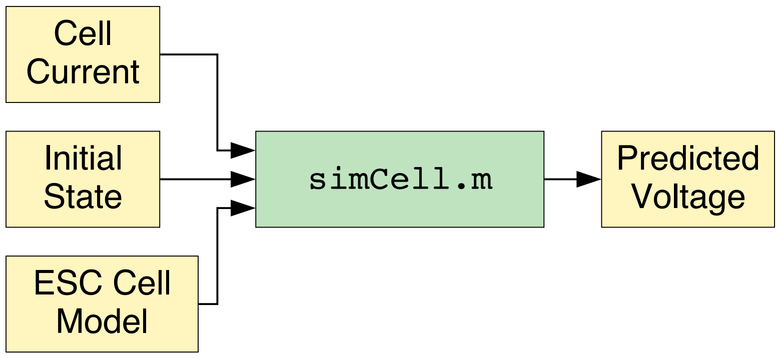
  - The dependencies of A_k and B_k on i_k, and the OCV term in the output make the equations nonlinear.
  - Still, the nonlinear state-space form is conducive to applying control-systems concepts, which we do in the following chapters.

- ECE5710 talked about how to find the unknown model parameter values, so we omit that discussion here.

  - Finding the model:

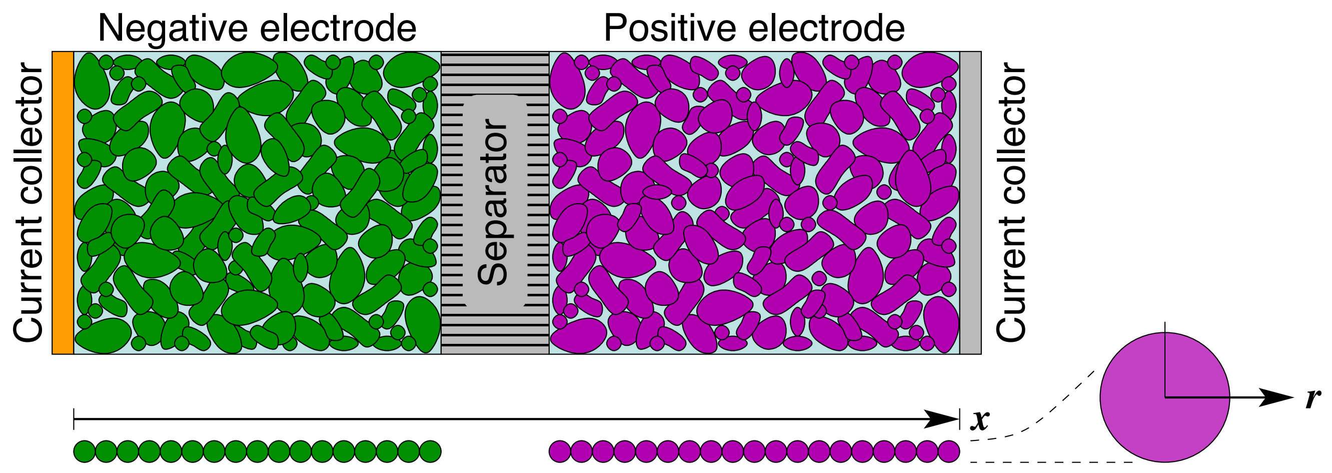


  - Simulating the model:



2.2: Modeling approach #2: Physics-based

- Continuum porous-electrode models use physics to derive equations for all internal cell processes using coupled PDEs.



- Within the cell, the following state variables are of interest:
  - The concentration of lithium in the solid, $c_s(x,r,t)$, and particularly at the surface of the solid, $c_{s,e}(x,t)$,
  - The concentration of lithium in the electrolyte, $c_e(x,t)$,
  - The potential in the solid, $\phi_s(x,t)$,
  - The potential in the electrolyte, $\phi_e(x,t)$, and
  - The rate of lithium movement between phases, $j(x,t)$.
- These five electrochemical variables can be found by solving the five coupled continuum-scale partial-differential equations (along with their associated boundary conditions),
  - Diffusion of lithium in the solid electrode particles

$$\frac{\partial}{\partial t}c_s = \frac{D_s}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial c_s}{\partial r}\right).$$

  - Charge balance in particles; electron current:

$$\nabla\cdot(\sigma_{\text{eff}}\nabla\phi_s) = a_s F\, j\,.$$

- Diffusion of lithium in electrolyte:

$$\frac{\partial(\varepsilon_e c_e)}{\partial t} = \nabla \cdot (D_{e,\text{eff}} \nabla c_e) + a_s(1 - t_+^0)\, j\,.$$

- Charge balance in electrolyte; ion current:

$$\nabla \cdot (\kappa_{\text{eff}} \nabla \phi_e + \kappa_{D,\text{eff}} \nabla \ln c_e) + a_s F\, j = 0.$$

- Reaction rate (where $\eta = (\phi_s - \phi_e) - U_{\text{ocp}} - j\, F R_{\text{film}}$):

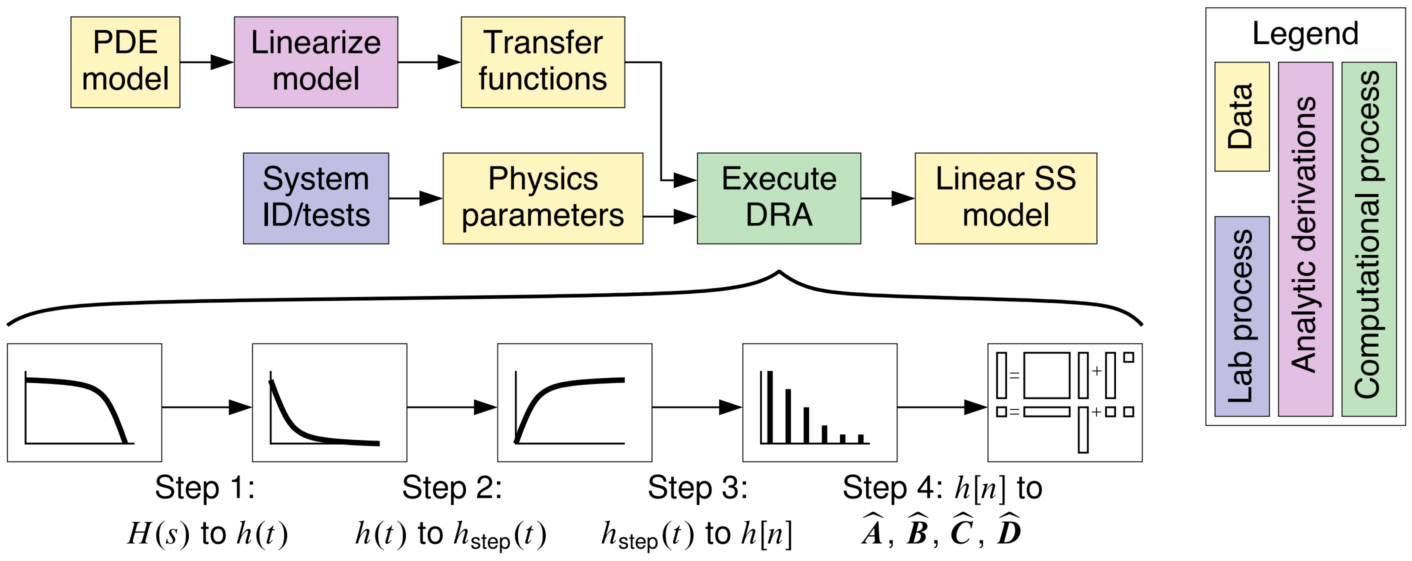
$$j = k_0 {c_e}^{1-\alpha} (c_{s,\max} - c_{s,e})^{1-\alpha} {c_{s,e}}^{\alpha} \left\{ \exp\left(\frac{(1-\alpha)F}{RT} \eta \right) - \exp\left(-\frac{\alpha F}{RT} \eta \right) \right\}.$$

- We found that these equations could be solved by simulation, but that simulation required considerable processor resources.
  - Also, experience shows that PDE simulators can be very fragile—have a hard time converging to robust solutions.
- So, in ECE5710 we proposed a process to create a discrete-time reduced-order model from the PDEs.
- We made two basic assumptions:[1]
  1. We assumed linear behavior: We linearized nonlinear equations using Taylor series;
  2. We assumed that the reaction current $j(x,t)$ was decoupled from (not a function of) the electrolyte concentration $c_e(x,t)$,

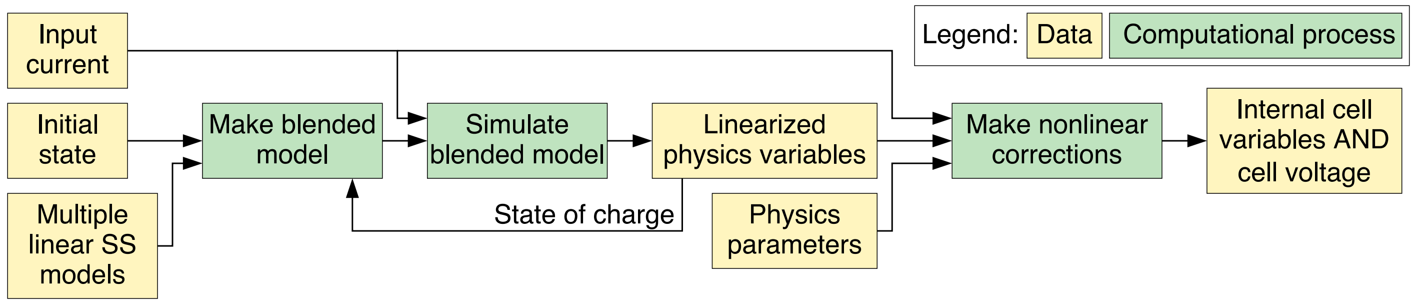
  allowing us to create transfer functions from the linearized equations.
- We then used a method called the "discrete-time realization algorithm" (DRA) to create a discrete-time state-space model.

[1] In simulations, we have found that the second assumption is quite good. The first assumption is less good, but nonlinear corrections help improve linear predictions.

- A pictorial overview of the two process steps is:
  - Finding the model:


PDE model
Linearize model
Transfer functions
System ID/tests
Physics parameters
Execute DRA
Linear SS model
Legend
Data
Lab process
Analytic derivations
Computational process
Step 1: $H(s)$ to $h(t)$
Step 2: $h(t)$ to $h_{\text{step}}(t)$
Step 3: $h_{\text{step}}(t)$ to $h[n]$
Step 4: $h[n]$ to $\widehat{\boldsymbol{A}}, \widehat{\boldsymbol{B}}, \widehat{\boldsymbol{C}}, \widehat{\boldsymbol{D}}$

  - Simulating the model:

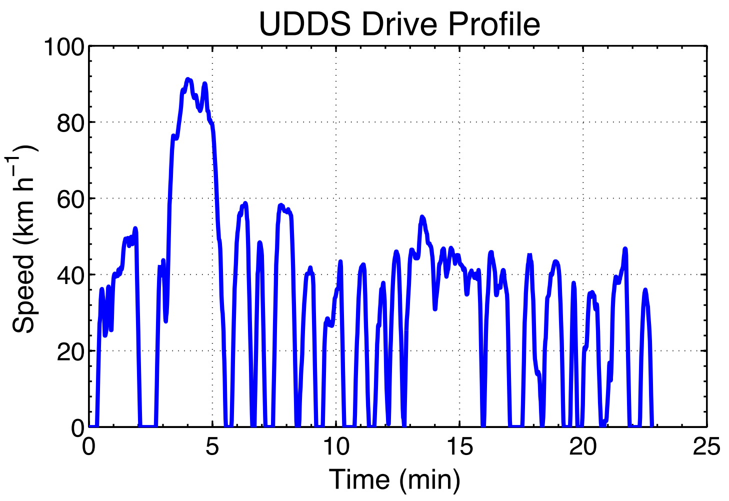

Input current
Initial state
Multiple linear SS models
Make blended model
Simulate blended model
Linearized physics variables
State of charge
Physics parameters
Make nonlinear corrections
Internal cell variables AND cell voltage
Legend: Data Computational process

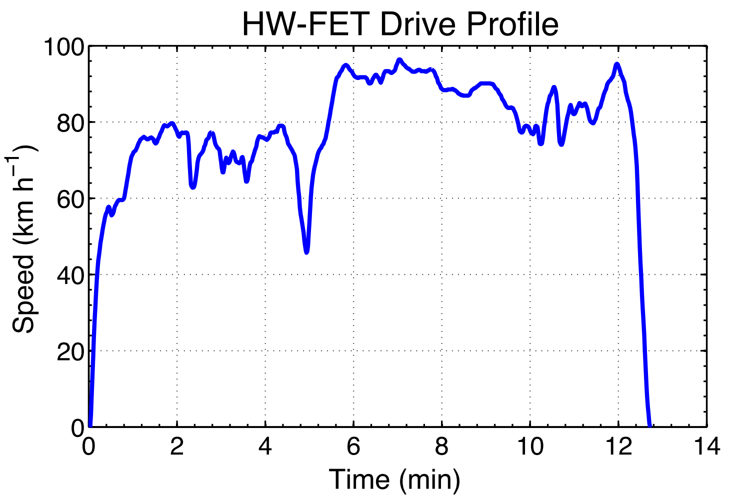
- A key feature of the DRA-produced ROM is that nonlinear corrections can be applied to the linearized model variables, greatly enhancing prediction accuracy.
- We will not discuss physics-based models further in this course—our focus is on battery management using empirical models.
- Battery management and controls using physics-based models is a major focus of our present research efforts and (I hope) of a future course.

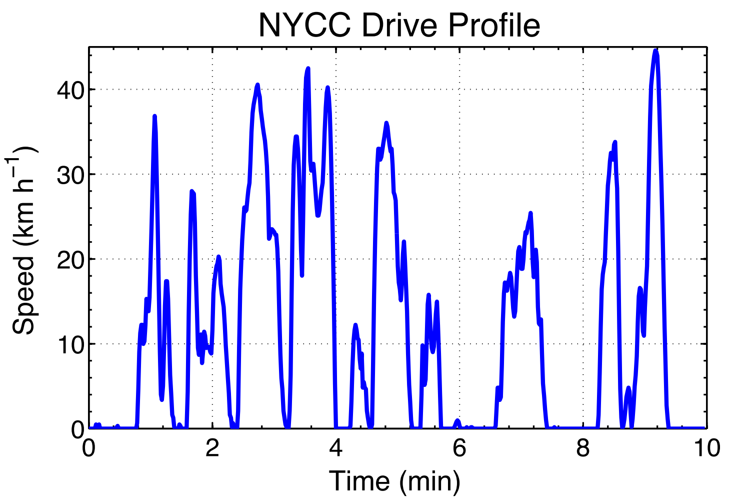
2.3: Simulating an electric vehicle

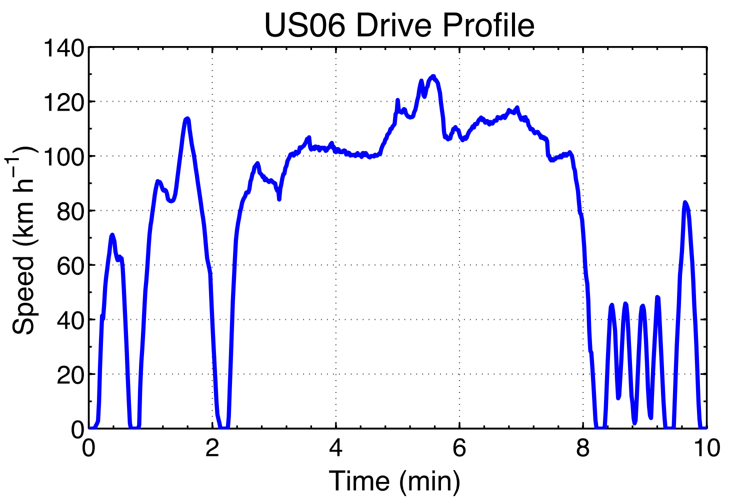
- When designing battery packs, it is important to be able to simulate the pack before proceeding too far down the design path.
  - Helps to ensure the pack will be able to meet all performance requirements before a large investment has been made.
- Battery packs may be used in all kinds of applications, but one area of interest for large battery packs is for energy storage in xEVs.
- To predict battery demand, we must simulate the vehicle over a number of real-world operating scenarios, and see the profile of power or current versus time demanded from the pack.
- HEV simulations are extremely complex, because the internal-combustion engine, multi-speed transmission, and hybrid blending control algorithms need to be simulated accurately.
- However, pure EV (and PHEV in electric-only mode) simulations are fairly straightforward since there is no IC engine, a single-speed fixed transmission is used, and there is no blending of power sources.
- To simulate an electric vehicle, two things are needed.
  1. An accurate description of the vehicle itself;
  2. The task the vehicle is required to accomplish.
- The vehicle description includes all factors needed for a dynamic simulation, including battery cell characteristics, battery module (groups of cells) characteristics, battery pack (groups of modules) characteristics, motor and inverter (motor driving power electronics) characteristics, drivetrain characteristics, and so forth.

- The vehicle task is characterized by a "drive cycle": a profile of desired vehicle speed versus time. Some examples:
  - The Urban Dynamometer Driving Schedule (UDDS)
  - The Highway Fuel Efficiency Test (HW-FET)
  - The New York City Cycle (NYCC), and
  - The US06 drive cycle (recorded by NREL, near Golden, CO).
- These four drive cycles are provided to you on the course web site. They have a 1 Hz profile rate, which will be assumed below.



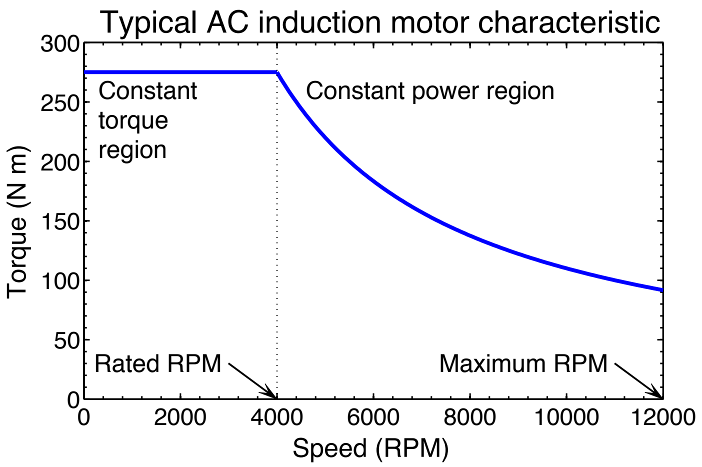




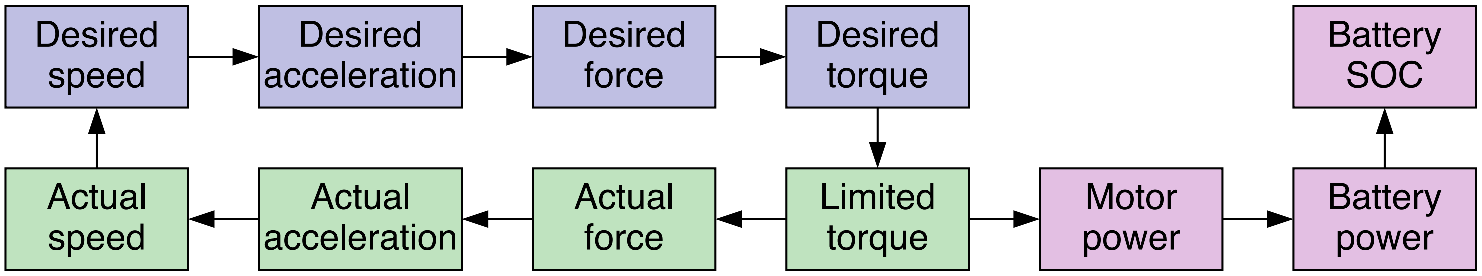


- All drive cycles are simulated using the same basic equations.

- The approach is to compute, on a second-by-second basis, desired accelerations to match the desired speed profile, therefore desired road forces, therefore desired motor torques and power values.
- These desired torques and power values are restricted by the specifications of the motor chosen for the vehicle, and thus achievable torques and power values may be of lesser magnitude.
- Achievable torques are computed, and therefore achieved road force, and achieved velocity (this will not always match the desired velocity).
- Battery power is computed based on motor power, and battery SOC is updated.
- Over the course of the driving profile, the battery SOC is depleted a certain amount, and the projected range of the vehicle is extrapolated from this data.
- We use a three-phase AC induction motor model, which limits torque and power based on three design constants: maximum torque, rated RPM, and maximum RPM.



- Overall simulation flowchart calculations:



2.4: Equations for vehicle dynamics

- In more detail: We calculate the desired vehicle acceleration as:[2]

$$\text{desired acceleration}\,[\text{m s}^{-2}] = (\text{desired speed}\,[\text{m s}^{-1}] - \text{actual speed}\,[\text{m s}^{-1}])/(1\ [\text{s}]).$$

- We compute desired acceleration force that the motor must produce at the road surface:

$$\text{desired acceleration force}\,[\text{N}] = \text{equivalent mass}\,[\text{kg}] \times \text{desired acceleration}\,[\text{m s}^{-2}].$$

- The equivalent mass combines the maximum vehicle mass and the equivalent mass of the rotating inertias

$$\text{equivalent mass}\,[\text{kg}] = \text{maximum vehicle mass}\,[\text{kg}] + \text{rotating equivalent mass}\,[\text{kg}]$$

$$\begin{aligned}\text{rotating equivalent mass}\,[\text{kg}] = \;&((\text{motor inertia}\,[\text{kg m}^2] + \text{gearbox inertia}\,[\text{kg m}^2]) \times N^2 \\ &+ \text{number of wheels} \times \text{wheel inertia}\,[\text{kg m}^2])/(\text{wheel radius}\,[\text{m}])^2,\end{aligned}$$

where N is gearbox ratio [unitless] = (motor RPM)/(wheel RPM), and the gearbox inertia is measured at motor side, not output side.

- Wheel radius is assumed to be that of the rolling wheel; *i.e.*, taking into account flattening due to load.

[2] Vehicle modeling equations are from T. Gillespie, "Fundamentals of Vehicle Dynamics," Society of Automotive Engineers Inc, 1992.

- Other forces acting on the vehicle are assumed to comprise:

$$\text{aerodynamic force [N]} = \frac{1}{2}(\text{air density } \rho \text{ [kg m}^{-3}])\times(\text{frontal area [m}^2])\times$$
$$(\text{drag coefficient } C_d \text{ [unitless]}) \times (\text{prior actual speed [m s}^{-1}])^2$$
$$\text{rolling force [N]} = (\text{rolling friction coefficient } C_r \text{ [unitless]}) \times$$
$$(\text{max. vehicle mass [kg]}) \times (\text{acceleration of gravity [9.81 m s}^{-2}])$$
$$\text{grade force [N]} = (\text{maximum vehicle mass [kg]}) \times$$
$$(\text{acceleration of gravity [9.81 m s}^{-2}]) \times \sin(\text{grade angle [rad]})$$
$$\text{brake drag [N]} = \text{constant road force input by user,}$$

where grade angle is the present (or average) slope of the road (positive is an incline, negative is a decline).

- Note that the rolling force is computed to be zero if the prior actual speed is zero.
- We can now compute the demand torque at the motor

$$\text{demanded motor torque [N m]} = (\text{desired acceleration force [N]} +$$
$$\text{aerodynamic force [N]} + \text{rolling force [N]} + \text{grade force [N]} +$$
$$\text{brake drag [N]}) \times \text{wheel radius [m]}/N \text{ [unitless].}$$

- Demanded torque must be compared to the maximum available motor torque.
- Available torque is based on a model of an ideal three-phase AC induction motor.
- When positive torque (acceleration) is demanded,

  - If prior actual motor speed is less than rated motor speed, then the maximum available torque is the rated maximum available torque;
  - Otherwise, the maximum available torque is computed as (rated maximum available torque [N m]) $\times$ (rated motor speed [RPM]) / (prior actual motor speed [RPM]).

- When negative torque (deceleration) is demanded, torque demand is split between friction brakes (assumed infinitely powerful) and motor.
- Energy recovered from the motor replaces energy depleted from the battery (less inefficiency losses) in a "regeneration" event.
- The maximum motor torque available for regeneration (in an unsigned sense) is calculated as the minimum of the maximum available torque for acceleration and a "regen fraction" times the rated maximum available torque.
- The limited torque at the motor is the lesser of the demanded motor torque and the maximum available torque (in an unsigned sense).
- Now that the motor torque limits have been established, we can compute the actual acceleration force that is available, the actual acceleration, and the actual velocity.
- We can now compute the actual acceleration:

$$\begin{aligned}&\text{actual acceleration force}\,[\text{N}] = \text{limited torque at motor}\,[\text{N m}] \times \\ &\qquad \text{N}\,[\text{unitless}]/\text{wheel radius}\,[\text{m}] - \text{aerodynamic force}\,[\text{N}] - \\ &\qquad \text{rolling force}\,[\text{N}] - \text{grade force}\,[\text{N}] - \text{brake drag}\,[\text{N}] \\ &\text{actual acceleration}\,[\text{m s}^{-2}] = \\ &\qquad \text{actual acceleration force}\,[\text{N}]/\text{equivalent mass}\,[\text{kg}].\end{aligned}$$

- The actual acceleration as just calculated may cause the motor to spin at a higher angular velocity than it is rated for.
- Therefore, we cannot compute actual speed as simply as follows:

$$\text{actual speed}\,[\text{m s}^{-1}] = \text{prior actual speed}\,[\text{m s}^{-1}] + \text{actual acceleration}\,[\text{m s}^{-2}] \times 1\,\text{s}.$$

- Instead, we must compute a motor RPM first, then limit that RPM, and then compute the actual vehicle speed.

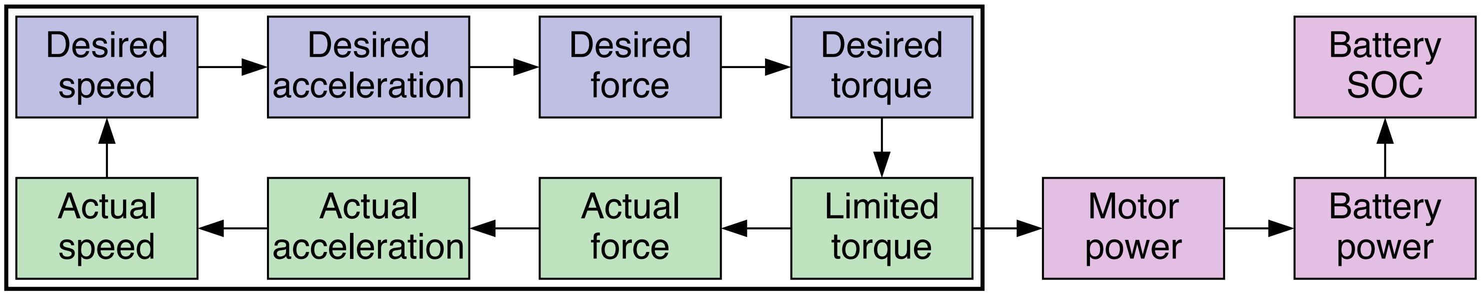
$$\text{test speed}\,[\text{m s}^{-1}] = \text{prior actual speed}\,[\text{m s}^{-1}] + \text{actual acceleration}\,[\text{m s}^{-2}] \times 1\,\text{s}$$

$$\text{motor speed}\,[\text{RPM}] = \text{test speed}\,[\text{m s}^{-1}] \times N\,[\text{unitless}] \times 60\,[\text{s min}^{-1}]/(2\pi \times \text{wheel radius}\,[\text{m}]).$$

- Motor speed is limited by the maximum rated motor speed to make a limited motor speed [RPM].
- Then, actual vehicle speed is computed as:

$$\text{actual speed}\,[\text{m s}^{-1}] = \text{limited motor speed}\,[\text{RPM}] \times 2\pi \times \text{wheel radius}\,[\text{m}]/(60\,[\text{s min}^{-1}] \times N\,[\text{unitless}]).$$

- The full circuit from desired to actual speed has now been described.



2.5: Vehicle range calculations, example

- The equations developed so far show whether the vehicle is able to develop the accelerations required to follow a specific drive profile.
- They assume that sufficient battery power is available to supply the motor demand at every time step.
- To determine vehicle range based on battery capacity, the battery must also be simulated—when a minimum battery SOC or voltage is reached, the distance driven to that point is the vehicle range.
- First, instantaneous power required by the motor is computed.

$$\begin{aligned}\text{motor power [kW]} &= 2\pi\ [\text{rad revolution}^{-1}]\times\\ &\left(\frac{\text{motor speed [RPM]} + \text{previous motor speed [RPM]}}{2}\right)\times\\ &\text{limited torque at motor [N m]}/(60\ [\text{s min}^{-1}]\times 1000\ [\text{W kW}^{-1}]).\end{aligned}$$

- If motor power is positive, then battery power is calculated as:

$$\begin{aligned}\text{battery power [kW]} &= \text{overhead power [kW]} +\\ &\text{motor power [kW]}/\text{drivetrain efficiency [unitless]},\end{aligned}$$

  where overhead power is the constant power drain from other vehicle systems, such as air conditioners, "infotainment" systems etc.

- If motor power is negative (regen), battery power is calculated as:

$$\begin{aligned}\text{battery power [kW]} &= \text{overhead power [kW]} +\\ &\text{motor power [kW]}\times\text{drivetrain efficiency [unitless]}.\end{aligned}$$

- Assuming for now a constant battery voltage (a poor assumption),

$$\begin{aligned}\text{battery current}\,[\text{A}] &= \text{battery power}\,[\text{kW}] \times 1000\,[\text{W}\,\text{kW}^{-1}]/\\ &\quad \text{battery nominal voltage}\,[\text{V}].\end{aligned}$$

- Battery state of charge is updated as:

$$\begin{aligned}\text{battery SOC}\,[\%] &= \text{prior battery SOC}\,[\%] - \text{battery current}\,[\text{A}] \times 1\,[\text{s}]/\\ &\quad (3600\,[\text{s}\,\text{hr}^{-1}] \times \text{battery capacity}\,[\text{A}\,\text{hr}]) \times 100\,[\%].\end{aligned}$$

- Driving range is extrapolated from the drive cycle calculations

$$\begin{aligned}\text{range}\,[\text{miles}] &= \text{total distance of simulated drive cycle}\,[\text{miles}] \times\\ &\quad (\text{max. rated battery SOC}\,[\%] - \text{min. rated battery SOC}\,[\%])/\\ &\quad (\text{SOC at beginning}\,[\%] - \text{SOC at end of drive cycle}\,[\%]).\end{aligned}$$

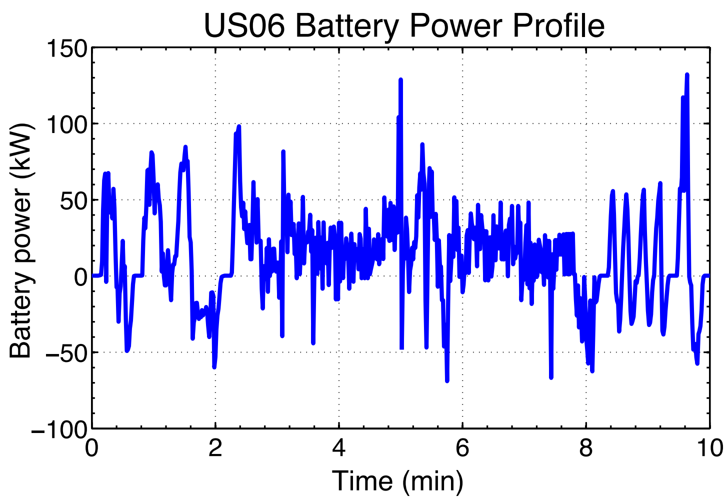
Sample code

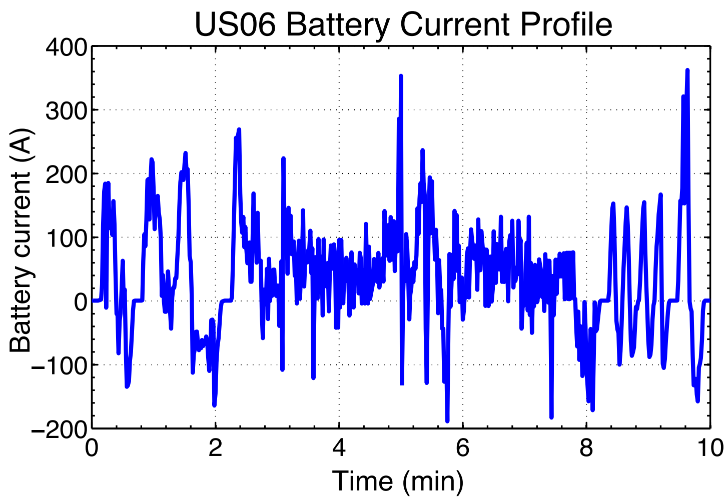
- Some sample EV simulation code is provided in the appendix.
- There are two MATLAB scripts:
  - `setupSimVehicle.m` is an example of how to set up the parameter values that describe the vehicle and the drive cycle;
  - `simVehicle.m` is the code that executes the equations we've just described to accomplish the simulation.
- `setupSimVehicle.m` defines the parameters of the battery cell, module, and pack; the motor, the wheels, and the drivetrain.
- The parameter values are stored in structures, combined to make a vehicle description, and are later used to simulate the vehicle.

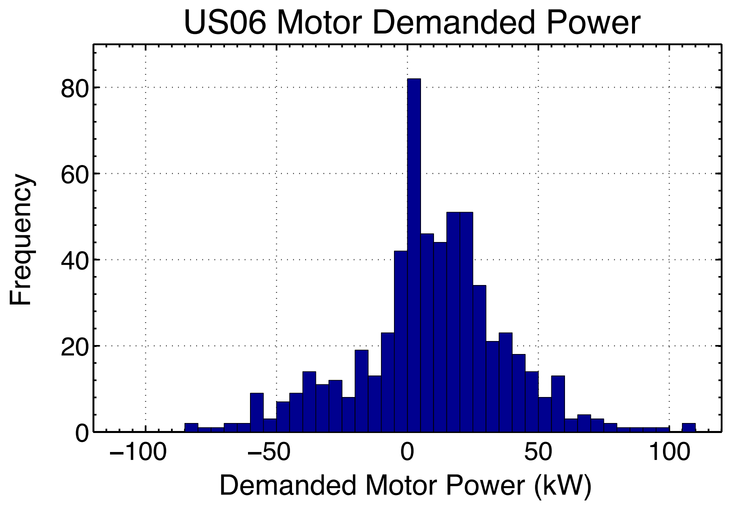
- The values in the example code are a rough description of a Chevy Volt operating in pure-electric mode, based on public information (and speculation) prior to vehicle release.
  - They're probably close, but not exact or verified for this vehicle.
- `simVehicle.m` simulates a drive profile and returns a structure called `results`, which has all kinds of information in it.

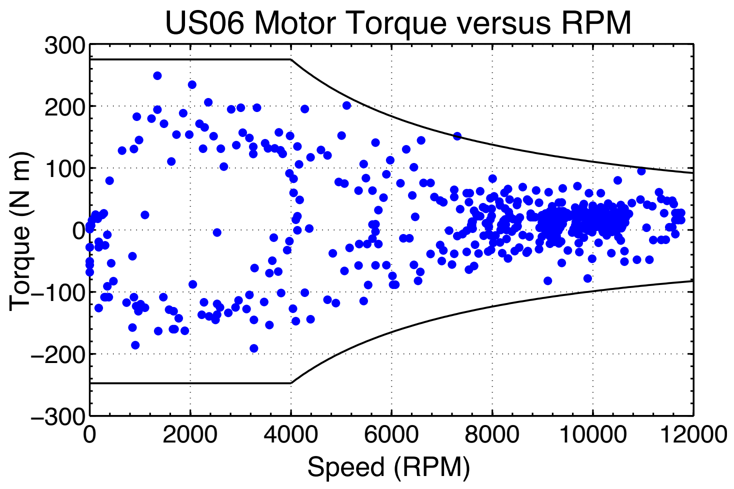
EV simulation results

- Some example results from the EV simulator using "Chevy Volt" parameters and US06 drive profile:









- Results can be used in sizing of motor, battery, electrical systems. . .

2.6: Simulating constant power and voltage

- In the vehicle simulator, we made the simplifying assumption that battery-pack voltage was constant when converting battery power demand into battery current demand.
- We know very well by now that this assumption is crude, at best.
- But, until now, we have considered only battery cell models having input equal to current.
- How do we simulate battery cell models with input equal to desired power or desired terminal voltage?

Constant power simulation

- Consider the ESC cell model,

$$\begin{aligned} x_k &= A_{k-1}x_{k-1} + B_{k-1}i_{k-1} \\ v_k &= \underbrace{\text{OCV}(x_k) + \text{hysteresis}(x_k) - \text{diffusion}(x_k)}_{\text{not a function of instantaneous current}} - R_0 i_k. \end{aligned}$$

- Note that x_k does not depend on i_k—it depends on i_{k-1}, i_{k-2}, etc.
- So, cell voltage comprises the "fixed" part, which does not depend on the present cell current, and the "variable" part, $-R_0 i_k$, which does depend on present cell current.
- Defining the fixed part to be $v_{f,k}$, we have $v_k = v_{f,k} - R_0 i_k$.
- Power equals voltage times current

$$p_k = v_k i_k = (v_{f,k} - R_0 i_k) i_k$$

$$R_0 i_k^2 - v_{f,k} i_k + p_k = 0.$$

- This quadratic equation can be solved every time sample to determine cell current to meet the power demand

$$i_k = \frac{v_{f,k} \pm \sqrt{v_{f,k}^2 - 4R_0 p_k}}{2R_0}.$$

- Sign of radical must be negative for positive overall cell voltage, so

$$i_k = \frac{v_{f,k} - \sqrt{v_{f,k}^2 - 4R_0 p_k}}{2R_0}.$$

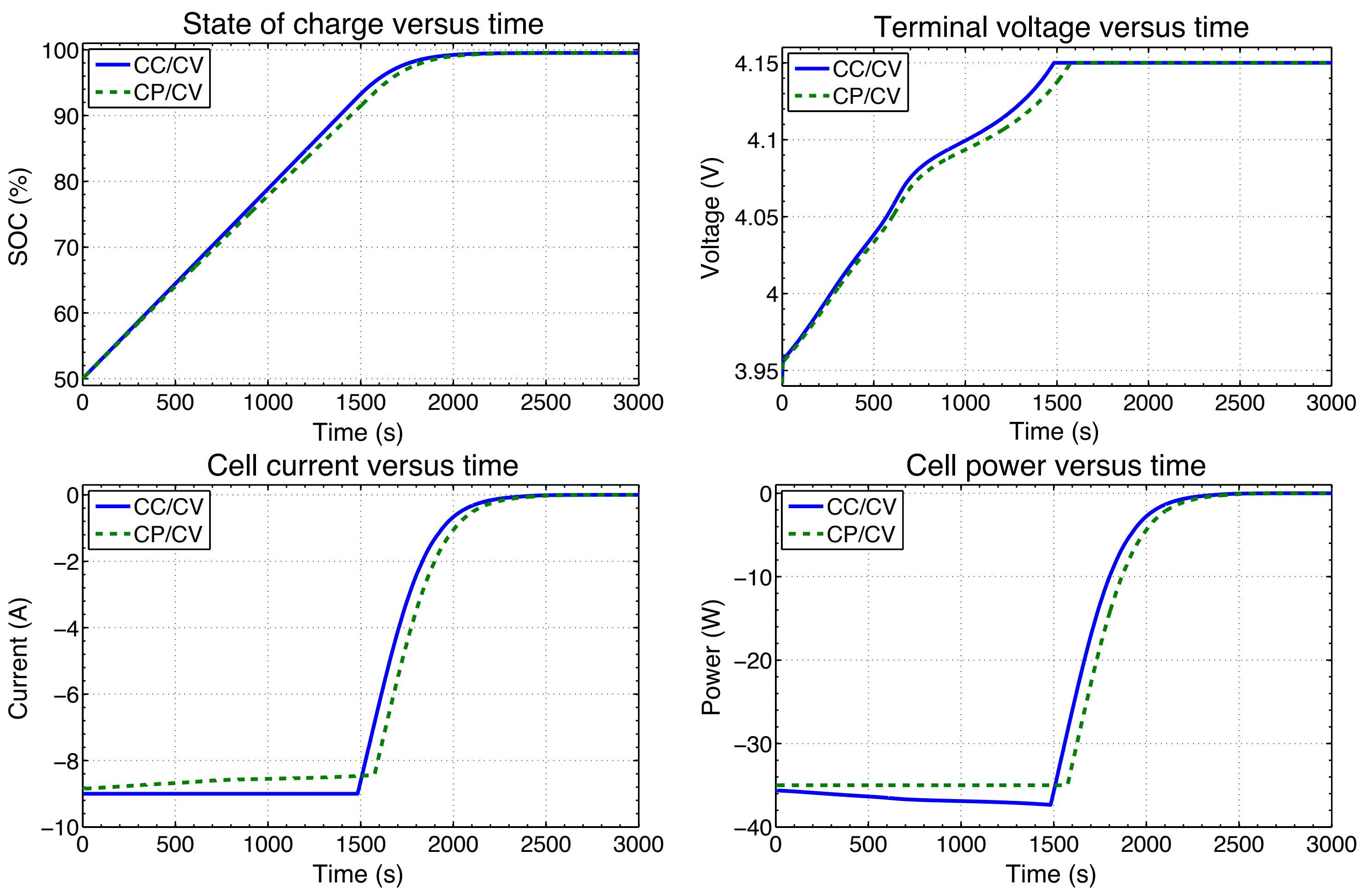
Constant voltage simulation

- Constant-voltage simulation is easier. Determine i_k such that

$$\begin{aligned} v_k &= v_{f,k} - R_0 i_k \\ i_k &= \frac{v_{f,k} - v_k}{R_0}. \end{aligned}$$

Example

- Two common types of battery-cell charging profiles are:
  - CC/CV: Constant current until a voltage limit is reached, then constant voltage at that limit until current is negligible.
  - CP/CV: Constant power until a voltage limit is reached, then constant voltage at that limit until current is negligible.
- CC/CV is often used in laboratory tests of cells, but CP/CV is more commonly used in xEV chargers.
- These are both straightforward to simulate.
- Consider charging a battery cell from 50 % SOC to max voltage using either a CC/CV or CP/CV profile:



- Code to perform this simulation:

```
% ------------------------------------------------------------------
% simCharge: Simulate CC/CV and CP/CV charging of a battery cell
% ------------------------------------------------------------------
clear all; close all; clc;
load E1model; % creates var. "model" with E1 cell parameter values

% ------------------------------------------------------------------
% Get ESC model parameters
% ------------------------------------------------------------------
maxtime = 3001; T = 25; % Simulation run time, temperature
q  = getParamESC('QParam',T,model);
rc = exp(-1./abs(getParamESC('RCParam',T,model)));
r  = (getParamESC('RParam',T,model));
m  = getParamESC('MParam',T,model);
g  = getParamESC('GParam',T,model);
r0 = getParamESC('R0Param',T,model);
maxV = 4.15; % maximum cell voltage of 4.15 V
```

```
% ----------------------------------------------------------------------
% First, simulate CC/CV
% ----------------------------------------------------------------------
storez = zeros([maxtime 1]);  % create storage for SOC
storev = zeros([maxtime 1]);  % create storage for voltage
storei = zeros([maxtime 1]);  % create storage for current
storep = zeros([maxtime 1]);  % create storage for power
z  = 0.5; irc = 0; h  = -1; % initialize to 50% SOC, resting
CC = 9;  % constant current of 9 A in CC/CV charge
for k = 1:maxtime,
  v = OCVfromSOCtemp(z,T,model) + m*h - r*irc; % fixed voltage

  ik = (v - maxV)/r0; % compute test ik to achieve maxV
  ik = max(-CC,ik);   % but limit current to no more than CC in mag.

  z = z - (1/3600)*ik/q;  % Update cell SOC
  irc = rc*irc + (1-rc)*ik; % Update resistor currents
  fac = exp(-abs(g.*ik)./(3600*q));
  h = fac.*h + (1-fac).*sign(ik); % Update hysteresis voltages
  storez(k) = z; % Store SOC for later plotting
  storev(k) = v - ik*r0;
  storei(k) = ik; % store current for later plotting
  storep(k) = ik*storev(k);
end % for k

time = 0:maxtime - 1;
figure(1); clf; plot(time,100*storez); hold on
figure(2); clf; plot(time,storev); hold on
figure(3); clf; plot(time,storei); hold on
figure(4); clf; plot(time,storep); hold on

% ----------------------------------------------------------------------
% Now, simulate CP/CV
% ----------------------------------------------------------------------
z  = 0.5; irc = 0; h  = -1; % initialize to 50% SOC, resting
CP = 35; % constant power limit of 35 W in CP/CV charge
for k = 1:maxtime,
  v = OCVfromSOCtemp(z,T,model) + m*h - r*irc; % fixed voltage

  % try CP first
  ik = (v - sqrt(v^2 - 4*r0*(-CP)))/(2*r0);
  if v - ik*r0 > maxV, % too much!
    ik = (v - maxV)/r0; % do CV instead
  end

  z = z - (1/3600)*ik/q;  % Update cell SOC
  irc = rc*irc + (1-rc)*ik; % Update resistor currents
  fac = exp(-abs(g.*ik)./(3600*q));
  h = fac.*h + (1-fac).*sign(ik); % Update hysteresis voltages
  storez(k) = z; % Store SOC for later plotting
  storev(k) = v - ik*r0;
  storei(k) = ik; % store current for later plotting
  storep(k) = ik*storev(k);
end % for k

green = [0 0.5 0];
figure(1); plot(time,100*storez,'--','color',green); grid on
title('State of charge versus time'); ylim([49 101]);
xlabel('Time (s)'); ylabel('SOC (%)');
legend('CC/CV','CP/CV','location','northwest');

figure(2); plot(time,storev,'--','color',green); grid on
title('Terminal voltage versus time'); ylim([3.94 4.16]);
xlabel('Time (s)'); ylabel('Voltage (V)');
legend('CC/CV','CP/CV','location','northwest');

figure(3); plot(time,storei,'--','color',green); grid on
title('Cell current versus time'); ylim([-10 0.3]);
xlabel('Time (s)'); ylabel('Current (A)');
legend('CC/CV','CP/CV','location','northwest');

figure(4); plot(time,storep,'--','color',green); grid on
title('Cell power versus time'); ylim([-40 1]);
xlabel('Time (s)'); ylabel('Power (W)');
legend('CC/CV','CP/CV','location','northwest');
```

2.7: Simulating battery packs

- We have now reviewed two models of battery cell dynamics.
- To simulate cell behavior, the model equations are evaluated and the model state equations are updated once per sample interval.
- But, how to simulate battery packs?

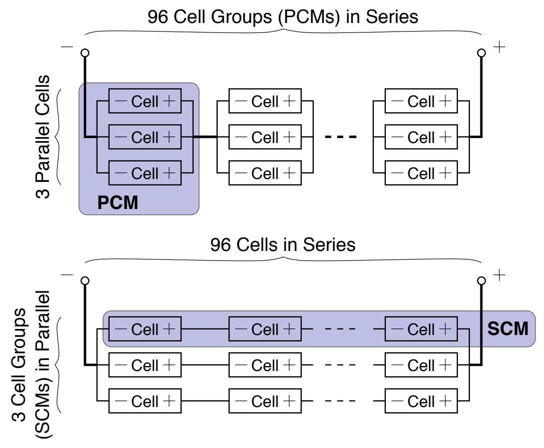
Series-connected cells

- Cells connected in series each experience the same current.
- If all cells have the same initial state and identical parameters, then all cells have exactly the same state and voltage for all time, so we need to simulate only one cell (the others will be identical).
- This is not generally true, however, so we can simulate all cells' dynamics by keeping state and model information for every cell, updating once per sample interval.
- Can also include inter-cell "interconnect" resistance term when computing pack voltage

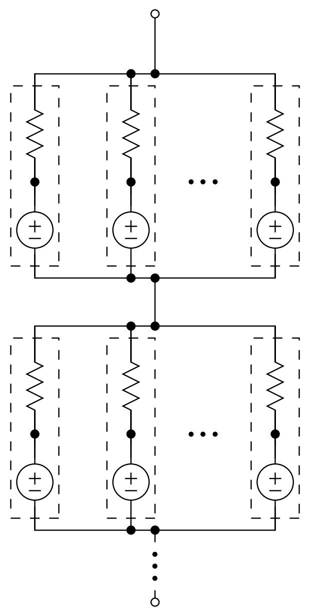
$$v_{\text{pack}}(t) = \left(\sum_{k=1}^{N_s} v_{\text{cell},k}(t)\right) - N_s R_{\text{interconnect}} i(t).$$

Parallel-connected cells and modules

- Series-connected packs are common for low-energy, high-power applications, such as HEV.


96 Cell Groups (PCMs) in Series
3 Parallel Cells
Cell
PCM
96 Cells in Series
3 Cell Groups (SCMs) in Parallel
SCM

- High-energy applications usually have cells and even entire sub-packs that are connected in parallel.
- One extreme is PCM; the other is SCM.
- Again, if cells differ, we will need to simulate all cells individually, but how?



- Consider first PCM, recalling that cell voltage comprises a "fixed" part that does not depend on the present cell current, and a "variable" part, $-R_0 i_k$ that does depend on present cell current.
- We can model cells in parallel as drawn, where the voltage source in each branch is the fixed part, and the resistor current is the variable part.
- By KVL, all terminal voltages must be equal; by KCL, the sum of branch currents must equal the total battery pack current.
- Define current through branch j of the PCM at time k as $i_{j,k}$; the "fixed" voltage as $v_{j,k}$, the PCM overall voltage as v_k, and the resistance of the jth branch as $R_{0,j}$. Then,

$$i_{j,k} = \frac{v_{j,k} - v_k}{R_{0,j}}.$$

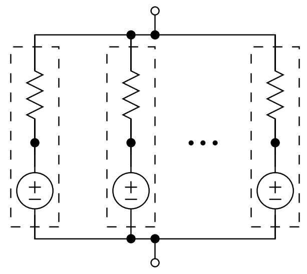
- We arrive at the total battery-pack current by summing:

$$i_k = \frac{v_{1,k} - v_k}{R_{0,1}} + \frac{v_{2,k} - v_k}{R_{0,2}} + \cdots + \frac{v_{N_p,k} - v_k}{R_{0,N_p}}.$$

- By re-arranging, we find the PCM voltage

$$v_k = \frac{\sum_{j=1}^{N_p} \frac{v_{j,k}}{R_{0,j}} - i_k}{\sum_{j=1}^{N_p} \frac{1}{R_{0,j}}}.$$

- All terms in this equation are known, so we can find v_k and then from it we can find all the $i_{j,k}$.
- Once we have the independent branch currents $i_{j,k}$, we can update the cell models associated with each cell.
- When simulating SCM, the approach is very similar to simulating PCM.
- Each cell in a SCM has its own "fixed" and "variable" parts.
- All "fixed" parts sum together to get an equivalent voltage source; all "variable" parts sum together to get an equivalent resistance.



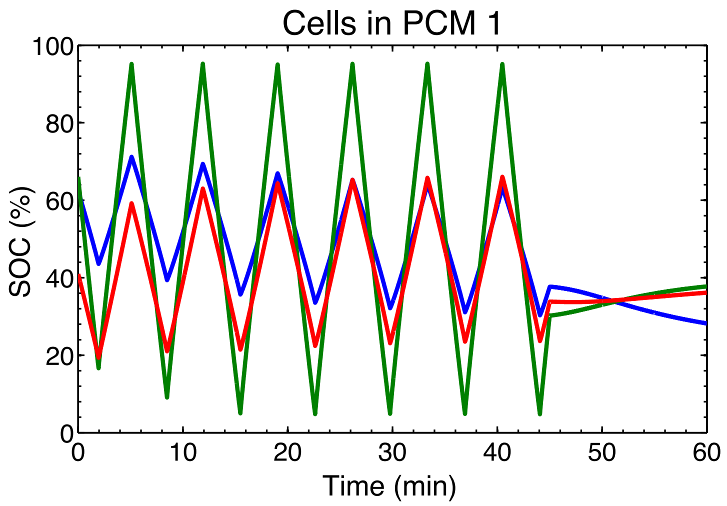
- Each SCM collapses to something that looks like a single high-voltage cell.
- Then, the total lumped voltage of an SCM is $v_{j,k}$ and the total lumped resistance is $R_{0,j}$.
- The bus voltage is

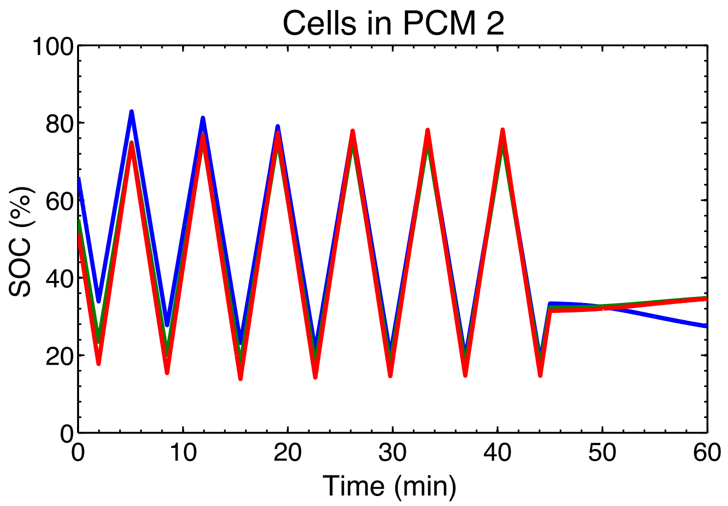
$$v_k = \frac{\sum_{j=1}^{N_p} \frac{v_{j,k}}{R_{0,j}} - i_k}{\sum_{j=1}^{N_p} \frac{1}{R_{0,j}}}.$$

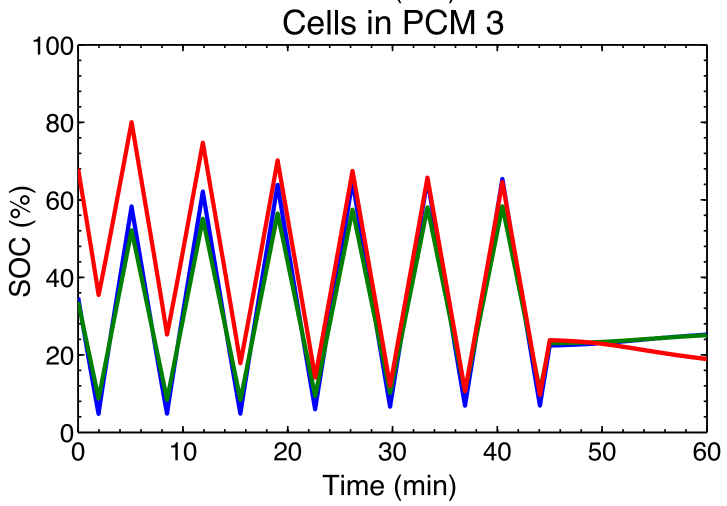
- The SCM currents are then found as $i_{j,k} = (v_{j,k} - v_k)/R_{0,j}$.
- With the currents through all cells now known, we can update all cell models.

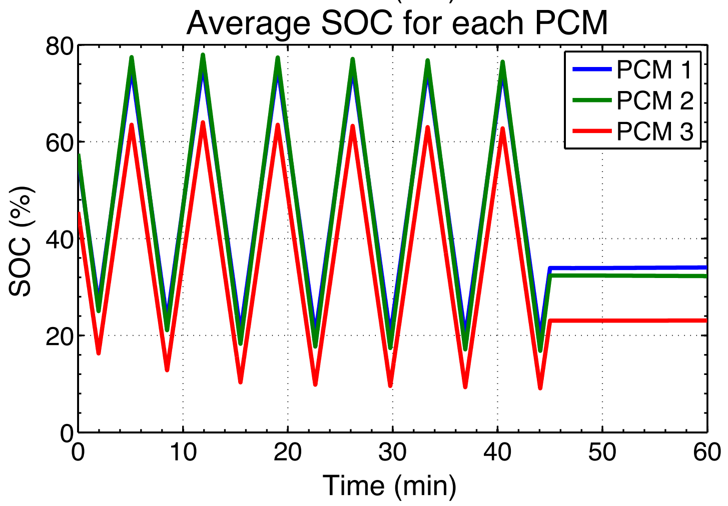
Example: Simulation of PCM

- Here we look at an example simulation with $N_s = 2$ and $N_p = 3$.
- Cell SOCs are randomly initialized between 30 % and 70 %.
- Cell R_0 values are randomly initialized between 5 mΩ and 25 mΩ.
- Cell capacities are randomly initialized between 4.5 Ah and 5.5 Ah.
- The pack is repeatedly discharged until lowest cell hits 5 % and then charged until highest cell hits 95 %. After 45 min, the pack rests.
- Individual cell SOCs and PCM-average SOCs are shown below:

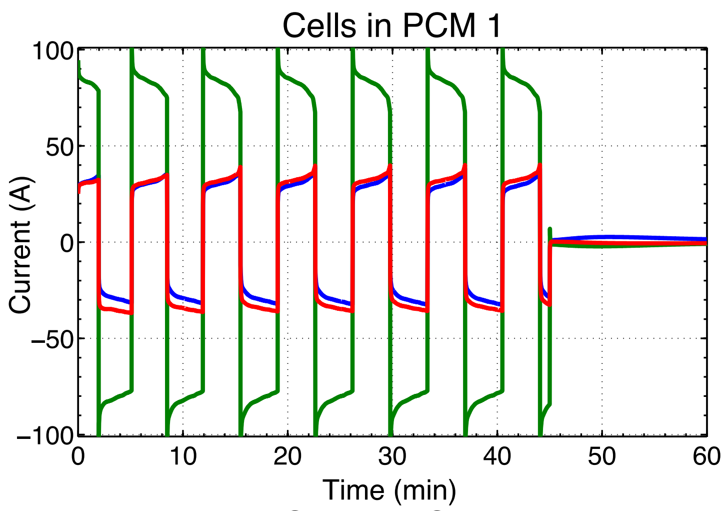


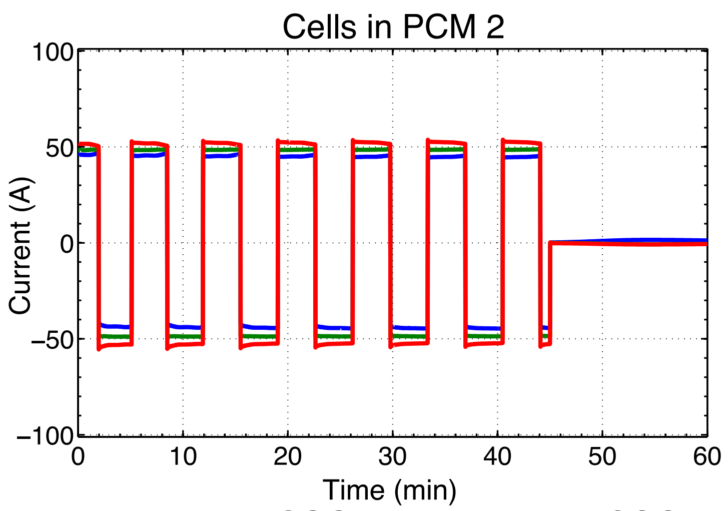


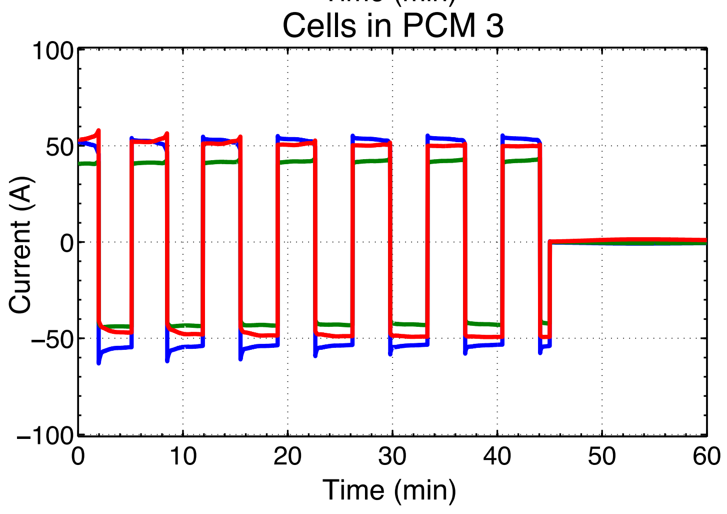


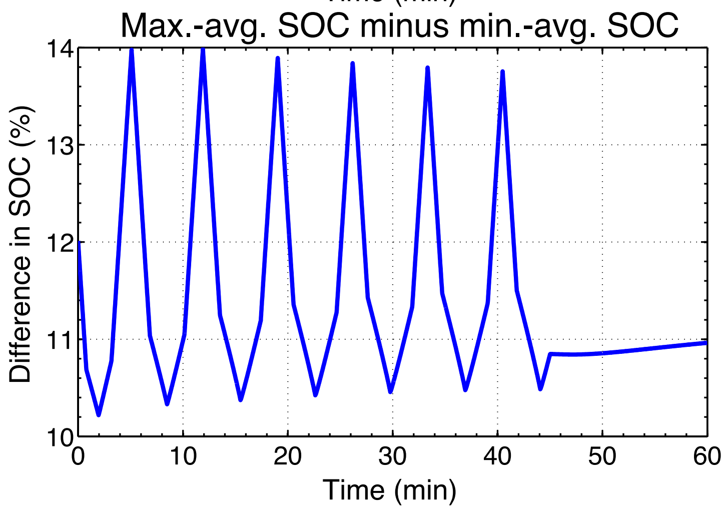


- The individual cell currents and the dispersion between PCM-average SOCs are plotted below:









- The code to reproduce this example is in an appendix to this chapter. It's pretty easy to modify and use for other configurations as well.

Where from here?

- We have now reviewed two types of models: our focus in this course is on applying the empirical model for battery management (we are actively researching physics-based models for same applications).
- We have seen how to use the model to predict and simulate cell performance.
- The next step is to use these models in a model-based-estimation scheme to estimate the internal state of the cell.

Appendix: setupSimVehicle.m

```
% setup simulation of vehicle - pass on to simVehicle.m
function results = setupSimVehicle
  global cell module pack motor drivetrain vehicle cycle results
  files = {'nycc.txt','udds.txt','us06col.txt','hwycol.txt'};

  % Setup the Chevy Volt
  % set up cell: capacity [Ah], weight [g], (vmax, vnom, vmin) [V]
  cell = setupCell(15,450,4.2,3.8,3.0);

  % set up module: number of cells in parallel, number of cells in
  % series, overhead of module by fraction of total cells' weight
  module = setupModule(3,8,0.08,cell);

  % set up pack: number of modules in series, overhead of pack by
  % fraction of total modules' weight, (full SOC, empty SOC) [%],
  % efficiency for this module
  pack = setupPack(12,0.1,75,25,0.96,module);

  % set up motor: max torque "Lmax" [Nm], (RPMrated, RPMmax) [RPM],
  % efficiency, inertia [kg/m2]
  motor = setupMotor(275,4000,12000,0.95,0.2);

  % set up wheel: radius [m], inertia [kg/m2], rollCoef
  wheel = setupWheel(0.35,8,0.0111);

  % set up drivetrain: inverter efficiency, fractional regen torque
  % limit, gear ratio, gear inertia [kg/m2], gear efficiency for this
  % pack, motor, and wheel
  drivetrain = setupDrivetrain(0.94,0.9,12,0.05,0.97,pack,motor,wheel);

  % set up vehicle: # wheels, roadForce [N], Cd, frontal area [m2],
  % weight [kg], payload [kg], overhead power [W] for this drivetrain
  vehicle = setupVehicle(4,0,0.22,1.84,1425,75,200,drivetrain);

  fprintf('\n\nStarting sims...\n');
  for theCycle = 1:length(files),
    cycle = dlmread(files{theCycle},'\t',2,0);
    results = simVehicle(vehicle,cycle,0.3);
    range = (vehicle.drivetrain.pack.socFull - ...
             vehicle.drivetrain.pack.socEmpty) /...
            (vehicle.drivetrain.pack.socFull - ...
             results.batterySOC(end)) * ...
            results.distance(end);
    fprintf('Cycle = %s, range = %6.1f [km]\n',files{theCycle},range);
  end
end

function cell = setupCell(capacity,weight,vmax,vnom,vmin)
  cell.capacity = capacity; % ampere hours
  cell.weight = weight; % grams
  cell.vmax = vmax; % volts
  cell.vnom = vnom; % volts
  cell.vmin = vmin; % volts
  cell.energy = vnom * capacity; % Watt-hours
  cell.specificEnergy = 1000 * cell.capacity * cell.vnom/ ...
                        cell.weight; % Wh/kg
end

function module = setupModule(numParallel,numSeries,overhead,cell)
  module.numParallel = numParallel;
  module.numSeries = numSeries;
  module.overhead = overhead;
  module.cell = cell;
  module.numCells = numParallel * numSeries;
  module.capacity = numParallel * cell.capacity;
  module.weight = module.numCells * cell.weight * 1/(1 - overhead)/1000; %
    kg
  module.energy = module.numCells * cell.energy/1000; % kWh
  module.specificEnergy = 1000 * module.energy / module.weight; % Wh/kg
end

function pack = setupPack(numSeries,overhead,socFull,socEmpty,...
                          efficiency,module)
  pack.numSeries = numSeries;
  pack.overhead = overhead;
  pack.module = module;
  pack.socFull = socFull;
  pack.socEmpty = socEmpty; % unitless
  pack.efficiency = efficiency; % unitless, captures I*I*R losses
  pack.numCells = module.numCells * numSeries;
  pack.weight = module.weight * numSeries * 1/(1 - overhead); % kg
  pack.energy = module.energy * numSeries; % kWh
  pack.specificEnergy = 1000 * pack.energy / pack.weight; % Wh/kg
  pack.vmax = numSeries*module.numSeries*module.cell.vmax;
  pack.vnom = numSeries*module.numSeries*module.cell.vnom;
  pack.vmin = numSeries*module.numSeries*module.cell.vmin;
end

function motor = setupMotor(Lmax,RPMrated,RPMmax,efficiency,inertia)
  motor.Lmax = Lmax; % N-m
  motor.RPMrated = RPMrated;
  motor.RPMmax = RPMmax;
  motor.efficiency = efficiency;
  motor.inertia = inertia; %kg-m2
  motor.maxPower = 2*pi*Lmax*RPMrated/60000; % kW
end

function wheel = setupWheel(radius,inertia,rollCoef)
  wheel.radius = radius; % m
  wheel.inertia = inertia; % km-m2
  wheel.rollCoef = rollCoef;
end

function drivetrain = setupDrivetrain(inverterEfficiency,regenTorque,...
      gearRatio,gearInertia,gearEfficiency,pack,motor,wheel)
  drivetrain.inverterEfficiency = inverterEfficiency;
  % regen torque is fraction of braking power that is used to charge
  % battery; e.g., value of 0.9 means 90% of braking power contributes
  % to charging battery; 10% lost to heat in friction brakes
  drivetrain.regenTorque = regenTorque;
  drivetrain.pack = pack;
  drivetrain.motor = motor;
  drivetrain.wheel = wheel;
  drivetrain.gearRatio = gearRatio;
  drivetrain.gearInertia = gearInertia; % kg-m2, measured on motor side
  drivetrain.gearEfficiency = gearEfficiency;
  drivetrain.efficiency = pack.efficiency * inverterEfficiency * ...
                          motor.efficiency * gearEfficiency;
end

function vehicle = setupVehicle(wheels,roadForce,Cd,A,weight,payload,
 overheadPwr,drivetrain)
  vehicle.drivetrain = drivetrain;
  vehicle.wheels = wheels; % number of them
  vehicle.roadForce = roadForce; % N
  vehicle.Cd = Cd; % drag coeff
  vehicle.A = A; % frontal area, m2
  vehicle.weight = weight; % kg
  vehicle.payload = payload; % kg
  vehicle.overheadPwr = overheadPwr; % W
  vehicle.curbWeight = weight + drivetrain.pack.weight;
  vehicle.maxWeight = vehicle.curbWeight + payload;
  vehicle.rotWeight = ((drivetrain.motor.inertia + ...
                         drivetrain.gearInertia) * ...
                        drivetrain.gearRatio^2 + ...
                        drivetrain.wheel.inertia*wheels)/...
                       drivetrain.wheel.radius^2;
  vehicle.equivMass = vehicle.maxWeight + vehicle.rotWeight;
  vehicle.maxSpeed = 2 * pi * drivetrain.wheel.radius * ...
                      drivetrain.motor.RPMmax * 60 / ...
                      (1000 * drivetrain.gearRatio); % km/h
end
```

Appendix: simVehicle.m

```
% results = simVehicle(vehicle,cycle,grade)
%   - simulate a vehicle defined by "vehicle", perhaps created using
%     setupSimVehicle.m
%   - cycle is Nx2, where first column is time in seconds and second
%     column is desired speed in miles per hour
%   - grade is road grade in percent - either a constant grade for all
%     time, or a different grade value for every point in time
function results = simVehicle(vehicle,cycle,grade)
  rho = 1.225; % air density, kg/m3

  results.vehicle = vehicle;
  results.cycle = cycle; % time in s, desired speed in miles/hour
  results.time = cycle(:,1); % s
  if isscalar(grade),
    results.grade = repmat(atan(grade/100),size(results.time)); % rad
  else
    results.grade = atan(grade/100); % rad
  end
  results.desSpeedKPH = cycle(:,2) * 1.609344; % convert to km/h
  results.desSpeed = min(vehicle.maxSpeed,...
                         results.desSpeedKPH*1000/3600); % m/s

  % pre-allocate storage for results
  results.desAccel = zeros(size(results.desSpeed)); % m/s2
  results.desAccelForce = zeros(size(results.desSpeed)); % N
  results.aeroForce = zeros(size(results.desSpeed)); % N
  results.rollGradeForce = zeros(size(results.desSpeed)); % N
  results.demandTorque = zeros(size(results.desSpeed)); % N-m
  results.maxTorque = zeros(size(results.desSpeed)); % N-m
  results.limitRegen = zeros(size(results.desSpeed)); % N-m
  results.limitTorque = zeros(size(results.desSpeed)); % N-m
  results.motorTorque = zeros(size(results.desSpeed)); % N-m
  results.demandPower = zeros(size(results.desSpeed)); % kW
  results.limitPower = zeros(size(results.desSpeed)); % kW
  results.batteryDemand = zeros(size(results.desSpeed)); % kW
  results.current = zeros(size(results.desSpeed)); % A
  results.batterySOC = zeros(size(results.desSpeed)); % 0..100
  results.actualAccelForce = zeros(size(results.desSpeed)); % N
  results.actualAccel = zeros(size(results.desSpeed)); % m/s2
  results.motorSpeed = zeros(size(results.desSpeed)); % RPM
  results.actualSpeed = zeros(size(results.desSpeed)); % m/s
  results.actualSpeedKPH = zeros(size(results.desSpeed)); % km/h
  results.distance = zeros(size(results.desSpeed)); % km

  prevSpeed = 0; prevTime = results.time(1) - 1; prevMotorSpeed = 0;
  prevSOC = vehicle.drivetrain.pack.socFull; prevDistance = 0;
  for k = 1:length(results.desSpeed),
    results.desAccel(k) = (results.desSpeed(k) - prevSpeed)/ ...
                          (results.time(k) - prevTime);
    results.desAccelForce(k) = vehicle.equivMass * results.desAccel(k);
    results.aeroForce(k) = 0.5 * rho * vehicle.Cd * vehicle.A * ...
                           prevSpeed^2;
    results.rollGradeForce(k) = vehicle.maxWeight * 9.81 * ...
                                sin(results.grade(k));
    if abs(prevSpeed) > 0,
      results.rollGradeForce(k) = results.rollGradeForce(k) + ...
        vehicle.drivetrain.wheel.rollCoef * vehicle.maxWeight * 9.81;
    end
    results.demandTorque(k) = (results.desAccelForce(k) + ...
                               results.aeroForce(k) + ...
                               results.rollGradeForce(k) + ...
                               vehicle.roadForce) * ...
                               vehicle.drivetrain.wheel.radius / ...
                               vehicle.drivetrain.gearRatio;
    if prevMotorSpeed < vehicle.drivetrain.motor.RPMrated,
      results.maxTorque(k) = vehicle.drivetrain.motor.Lmax;
    else
      results.maxTorque(k) = vehicle.drivetrain.motor.Lmax * ...
          vehicle.drivetrain.motor.RPMrated / prevMotorSpeed;
    end
    results.limitRegen(k) = min(results.maxTorque(k),...
                                vehicle.drivetrain.regenTorque * ...
                                vehicle.drivetrain.motor.Lmax);
    results.limitTorque(k) = min(results.demandTorque(k),...
                                 results.maxTorque(k));
    if results.limitTorque(k) > 0,
      results.motorTorque(k) = results.limitTorque(k);
    else
      results.motorTorque(k) = max(-results.limitRegen(k),...
                                   results.limitTorque(k));
    end

    results.actualAccelForce(k) = results.limitTorque(k) * ...
```

```
                             vehicle.drivetrain.gearRatio / ...
                             vehicle.drivetrain.wheel.radius - ...
                             results.aeroForce(k) - ...
                             results.rollGradeForce(k) - ...
                             vehicle.roadForce;
   results.actualAccel(k) = results.actualAccelForce(k) / ...
                         vehicle.equivMass;
   results.motorSpeed(k) = min(vehicle.drivetrain.motor.RPMmax,...
                        vehicle.drivetrain.gearRatio * ...
                        (prevSpeed + results.actualAccel(k) * ...
                        (results.time(k) - prevTime)) * 60 / ...
                        (2*pi*vehicle.drivetrain.wheel.radius));
   results.actualSpeed(k) = results.motorSpeed(k) * ...
                         2*pi*vehicle.drivetrain.wheel.radius / ...
                         (60 * vehicle.drivetrain.gearRatio);
   results.actualSpeedKPH(k) = results.actualSpeed(k) * 3600/1000;
   deltadistance = (results.actualSpeed(k) + prevSpeed)/2 *...
                   (results.time(k) - prevTime)/1000;
   results.distance(k) = prevDistance + deltadistance;

   if results.limitTorque(k) > 0,
     results.demandPower(k) = results.limitTorque(k);
   else
     results.demandPower(k) = max(results.limitTorque(k),...
                               -results.limitRegen(k));
   end
   results.demandPower(k) = results.demandPower(k) * 2*pi * ...
           (prevMotorSpeed + results.motorSpeed(k))/2 / 60000;
   results.limitPower(k) = max(-vehicle.drivetrain.motor.maxPower,...
                        min(vehicle.drivetrain.motor.maxPower,...
                            results.demandPower(k)));
   results.batteryDemand(k) = vehicle.overheadPwr/1000;
   if results.limitPower(k) > 0,
     results.batteryDemand(k) = results.batteryDemand(k) + ...
           results.limitPower(k)/vehicle.drivetrain.efficiency;
   else
     results.batteryDemand(k) = results.batteryDemand(k) + ...
           results.limitPower(k)*vehicle.drivetrain.efficiency;
   end
   results.current(k) = results.batteryDemand(k)*1000/...
                     vehicle.drivetrain.pack.vnom;
   results.batterySOC(k) = prevSOC - results.current(k) * ...
```

```
                            (results.time(k) - prevTime) / ...
                       (36*vehicle.drivetrain.pack.module.capacity);

    prevTime = results.time(k);
    prevSpeed = results.actualSpeed(k);
    prevMotorSpeed = results.motorSpeed(k);
    prevSOC = results.batterySOC(k);
    prevDistance = results.distance(k);
  end
```

Appendix: Code to simulate PCM

```
% ------------------------------------------------------------------------
% simPCM: Simulate parallel-connected-module packs (cells are connected in
% parallel to make modules; these modules are connected in series to make
% packs).
%
% The parameters for each cell may be different (e.g., capacity,
% resistance, etc.)
% ------------------------------------------------------------------------

clear all; close all; clc;
% ------------------------------------------------------------------------
% Initialize some pack configuration parameters...
% ------------------------------------------------------------------------
load E2model; % creates var. "model" with E2 cell parameter values
Ns = 2;       % Number of modules connected in series to make a pack
Np = 3;       % Number of cells connected in parallel in each module

% ------------------------------------------------------------------------
% Initialize some simulation configuration parameters...
% ------------------------------------------------------------------------
maxtime = 3600; % Simulation run time in simulated seconds
t0 = 2700; % Pack rests after time t0
storez = zeros([maxtime Ns Np]);  % create storage for SOC
storei = zeros([maxtime Ns Np]);  % create storage for current

% ------------------------------------------------------------------------
% Initialize states for ESC cell model
% ------------------------------------------------------------------------
z  = 0.25*ones(Ns,Np);
irc = zeros(Ns,Np);
h  = zeros(Ns,Np);


% ------------------------------------------------------------------------
% Default initialization for cells within the pack
% ------------------------------------------------------------------------
kvec = [0; maxtime+1]; % Iteration (time) vector for temp. profile
tvec = [25; 25]; % Default temperature profile
q    = getParamESC('QParam',25,model)*ones(Ns,Np);
rc   = exp(-1./abs(getParamESC('RCParam',25,model)))'*ones(Ns,Np);
r    = (getParamESC('RParam',25,model))';
```

```
m    = getParamESC('MParam',25,model)*ones(Ns,Np);
g    = getParamESC('GParam',25,model)*ones(Ns,Np);
r0   = getParamESC('R0Param',25,model)*ones(Ns,Np);
rt   = 0.000125; % 125 microOhm resistance for each tab


% ------------------------------------------------------------------------
% Modified initialization for cell variability
% ------------------------------------------------------------------------
% Set individual random "initial SOC" values
if 1, % set to "if 1," to execute, or "if 0," to skip this code
  z=0.30+0.40*rand([Ns Np]); % rand. init. SOC for ea. cell
end

% Set individual random cell-capacity values
if 1, % set to "if 1," to execute, or "if 0," to skip this code
  q=4.5+rand([Ns Np]);      % random capacity for ea. cell
end

% Set individual random cell-resistance relationships
if 1, % set to "if 1," to execute, or "if 0," to skip this code
  r0 = 0.005+0.020*rand(Ns,Np);
end
r0 = r0 + 2*rt; % add tab resistance to cell resistance


% ------------------------------------------------------------------------
% Add faults to pack: cells faulted open- and short-circuit
% ------------------------------------------------------------------------
% To delete a PCM (open-circuit fault), set a resistance to Inf
%r0(1,1) = Inf; % for example...

% To delete a cell from a PCM (short-circuit fault), set its SOC to NaN
%z(1,2) = NaN; % for example, delete cell 2 in PCM 1
Rsc = 0.0025; % Resistance value to use for cell whose SOC < 0%

% ------------------------------------------------------------------------
% Get ready to simulate... first compute pack capacity in Ah
% ------------------------------------------------------------------------
totalCap = min(sum(q,2)); % pack capacity = minimum module capacity
I = 10*totalCap; % cycle at 10C... not realistic, faster simulation
```

```
% ------------------------------------------------------------------------
% Okay... now to simulate pack performance using ESC cell model.
% ------------------------------------------------------------------------
for k = 1:maxtime,
  T = interp1(kvec,tvec,k); % cell temperature
  v = OCVfromSOCtemp(z,T,model); % get OCV for each cell: Ns * Np matrix
  v = v + m.*h - r.*irc; % add in capacitor voltages and hysteresis
  r0(isnan(z)) = Rsc; % short-circuit fault has "short-circuit" resistance
  V = (sum(v./r0,2) - I)./sum(1./r0,2);
  ik = (v-repmat(V,1,Np))./r0;
  z = z - (1/3600)*ik./q;  % Update each cell SOC
  z(z<0) = NaN; % set over-discharged cells to short-circuit fault
  irc = rc.*irc + (1-rc).*ik; % Update capacitor voltages
  fac = exp(-abs(g.*ik)./(3600*q));
  h = fac.*h + (1-fac).*sign(ik); % Update hysteresis voltages
  minz = min(z(:)); maxz = max(z(:)); % Check to see if SOC limit hit
  if minz < 0.05, I = -abs(I); end % stop discharging
  if maxz > 0.95, I = abs(I);  end % stop charging
  if k>t0, I = 0; end % rest
  storez(k,:,:) = z; % Store SOC for later plotting
  storei(k,:,:) = ik; % store current for later plotting
end % for k

% ------------------------------------------------------------------------
% In Figure 1, plot the individual SOC vs. time for all cells in all
% series PCMs. There is one subplot for each PCM.
% ------------------------------------------------------------------------
nonocPCMs = ~isinf(sum(r0,2)); % modules that are not open-circuit faulted
    figure(1); clf; t = (0:(length(storez(:,:,1))-1))/60;
xplots = round(1.0*ceil(sqrt(Ns))); yplots = ceil(Ns/xplots); means = [];
for k = 1:Ns,
  zr=squeeze(100*storez(:,k,:));
  subplot(yplots,xplots,k); plot(t,zr); axis([0 ceil(maxtime/60) 0 100]);
  title(sprintf('Cells in PCM %d',k));
  ylabel('SOC (%)'); xlabel('Time (min)');
  zr(isnan(zr))=0; % exclude dead cells (failed short) from mean
  if nonocPCMs(k), % exclude from average if open-circuit!
    means = [means; mean(zr,2)']; %#ok<AGROW>
  end
end
```

```
% ------------------------------------------------------------------------
% In Figure 2, plot the average SOC vs. time for all PCMs
% ------------------------------------------------------------------------
figure(2); clf; plot(t,means'); grid on
xlabel('Time (min)'); ylabel('SOC (%)');
title('Average SOC for each PCM');
legendstrings = [];
for k=1:Ns,
  if nonocPCMs(k),
    legendstrings = [legendstrings; ...
                     { sprintf('String %d',k) }]; %#ok<AGROW>
  end
end
legend(legendstrings);

% ------------------------------------------------------------------------
% In Figure 3, plot the (maximum average SOC) minus (minimum average SOC)
% ------------------------------------------------------------------------
figure(3); plot(t,(max(means)-min(means))); grid on
xlabel('Time (min)'); ylabel('Difference in SOC (%)');
title('Maximum-average SOC minus minimum-average SOC');

% ------------------------------------------------------------------------
% In Figure 4, plot the individual cell current vs. time for all cells in
% all series PCMs. There is one subplot for each PCM.
% ------------------------------------------------------------------------
figure(4); clf; t = (0:(length(storei(:,:,1))-1))/60;
for k = 1:Ns,
  zr=squeeze(storei(:,k,:));
  subplot(yplots,xplots,k); plot(t,zr);
  axis([0 ceil(maxtime/60) -101 101]);
  title(sprintf('Cells in PCM %d',k));
  ylabel('Current (A)'); xlabel('Time (min)');
end
```