Assignment #4: dynamic programming

Table of Contents

Group information

Group number: 4

Students:

- Giacomo Chiavetta, 348027
- Massimo Stella, s348062
- Tommaso Massone, s349235

Introduction

In this fourth and last project it is requested to develop a new and improved optimal control strategy that is supposed to fully adress the optimization problem and provide a better managment of the resources within the

vehicle system compared to what have been already done previously. Unlike the preceding internal combustion engine controllers in which the optimization was local, in fact, the dyanamic programming based controller is now able to fully exploit the vehicle speed cycle providing a global optimization outcome resulting from the minimization of a certain mathematical expression of cost function.

Dynamic Programming, in fact, is able to achieve the optimal policy involving the resolution of a single stage sub-problem iterevely extended to the subsequent stages until all the optimal sequences have been found. In order to define which is the stage subproblem it is needed to recall that the dynamic system under investigation is a series hybrid electric vehicle described by the following variables:

Exogenous Inputs:

Vehicle Speed - *VehSpd*

Vehicle Acceleration - *VehAcc*

State Variables:

State of Charge - *SOC*

Control Variables:

Engine Speed - *EngSpd*

Engine Torque - *EngTrq*

Stage Cost:

Fuel Flow Rate - *fuelFlwRate*

Battery Current - *battCurr*

In can be noticed that our Stage Cost variables are now extended to the battery current (*battCurr)* and not only the fuel flow rate (*fuelFlwRate),* meaning that the simulations carried out during the project have a dual target critaria which will increase the complexity of our control strategy.

DYNAMIC PROGRAMMING (DP)

The simulations will be based on implementation of the DP function, to better understand the working principle an insight of the theorotical framework is provided:

Our deterministic discrete-state and finite-state dynamic system can be expressed as:

$$x_{k+1} = f\left(x_k, u_k\right)$$

The cost fuction, i.e. the total cost incurred over the whole multistage problem, is the sum of all the stage costs:

$$J(x_0, u_0, \ldots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g_{k\,(x_k, u_k)}$$

where $g_N(x_N)$ is the terminal cost used in case of a non unique final state variable solution exists.

As it was introduced before, the optimal policy can be constructed by solving the "tail subproblem", that is, the minimiazion problem that include primarly the final stage and then, iteratevely, extend the optimal policy to the previous stages continuing until an optimal policy for the entire problem is found.

This is done during the backward phase by first computing the optimal stage cost for all final stages X_N:

$$J_N^{\text{opt}}(x_N) = g_N(x_N)$$

and then compute for each possible state x_K at each stage $K = N-1, N-2, \ldots 0$:

$$J_K^{\text{opt}}(x_K) = \min\left(g\left(x_k, u_k\right) + J_{K+1}^{\text{opt}}\left(f\left(x_k, u_k\right)\right)\right)$$

The stage cost $J_K^{\text{opt}}(x_K)$ includes all stage costs considering that from that state onward the control sequence is optimal.

Once the backward phase is concluded, depending on the inizial state, the optimal control sequence can be found through the forward phase:

At each stage $K = 0, 1, \ldots . N-1$, in fact, it it possible to trace back the control sequence that minimize the cost function J. To retrive which is the optimal control sequence u^* starting from the initial state x_0 at $K = 0$, it is enough to choose the minimum cost stage $g\left(x_k, u_k\right)$ considering that from that stage onward the control sequence is optimal. This phase can be conceptually expressed as:

$$\text{For} \quad K = 0, 1, \ldots . N-1 \qquad\qquad u_k^*(x_K) = \text{argmin}\left(g\left(x_k, u_k\right) + J_{K+1}^{\text{opt}}\left(f\left(x_k, u_k\right)\right)\right)$$

Once u_k^* is found the simulation can be advanced:

$$x_{k+1} = f\left(x_k, u_k^*\right)$$

In this project the DP is used several times to carried out different simulations in which both the fuel consuption and battery energy consumption is minimize creating a battery aging-aware Energy Management Strategy.

Load the cycle and vehicle data

Project starts with the provided folders acquisition by means of the following commands:

```
clear;clc;close all

addpath("models");      % Add the "models" directory to the search path
addpath("data");        % Add the "data" directory to the search path
addpath("utilities");   % Add the "utilities" directory to the search path
```

Load the vehicle data

The vehicle parameters have been retrieved and the relevant data pertaining to our workgroup have been substituted through the provided function "*scaleVehData*".

```
veh = load("data\vehData.mat"); % Vehicle data are now contained in the struct veh
variable

% Scale vehicle data according to group parameters

veh = scaleVehData(veh, 66e3, 82e3, 945); % Rated motor power, Rated engine power,
Battery nominal energy, respectively
```

Load cycle data

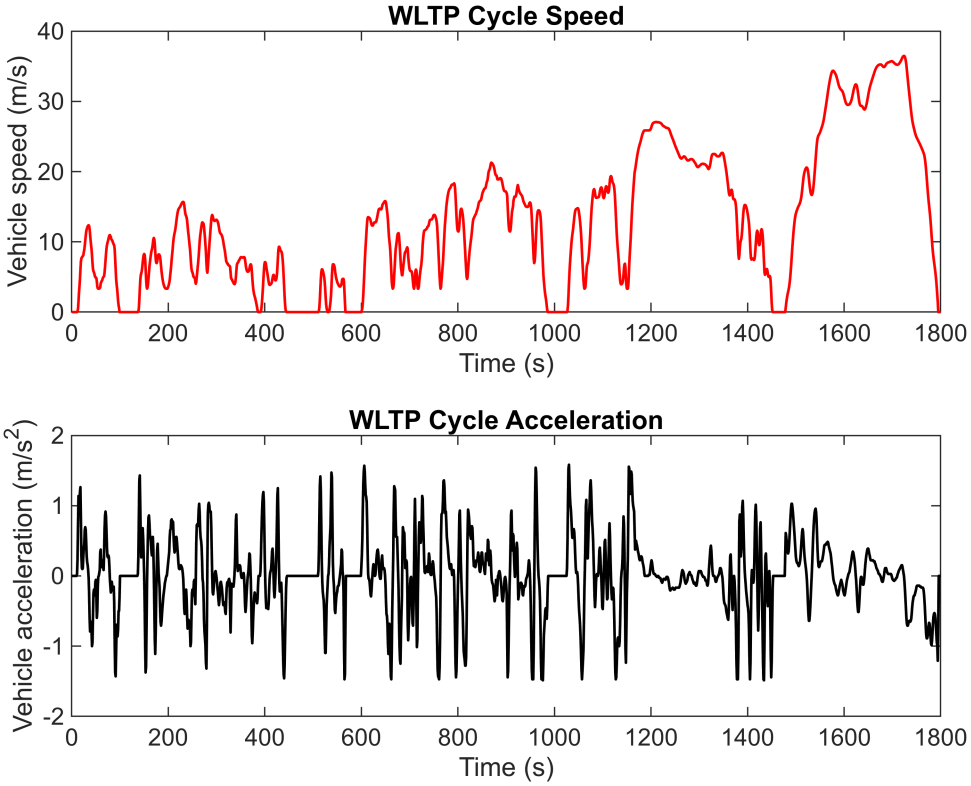
The project considers the WLTP, whose corresponding data are stored within the following directory.

```
mission = load("data\WLTP.mat"); % WLTP data are now contained in the struct
mission variable
time = mission.time_s; % (s)  Vector of the time instants within the considered
cycle
vehSpd = mission.speed_km_h ./ 3.6; % (m/s) Vector of the instant velocities
referred to each time value
vehAcc = mission.acceleration_m_s2; % (m/s^2) Vector of the instant acceleration
referred to each time value
```

Plot the mission cycle data

The velocity and acceleration profiles of the WLTP are now being plotted as a function of time.

```
figure
t = tiledlayout(2,1); % Create a 2-row, 1-column tiled layout for multiple plots

nexttile(1) % Select the first tile (upper plot)
plot(time, vehSpd,"r-",'LineWidth', 1); % Plot vehicle speed over time
title("WLTP Cycle Speed");
xlabel("Time (s)");
ylabel("Vehicle speed (m/s)");

nexttile(2) % Select the second tile (lower plot)
plot(time, vehAcc,"k-", 'LineWidth', 1); % Plot vehicle acceleration over time
title("WLTP Cycle Acceleration");
xlabel("Time (s)");
ylabel("Vehicle acceleration (m/s^2)");
```



Develop a fuel-optimal EMS with dynamic programming

Starting from the problem definition, the DynaProg inputs are computed considering that, as already done in previous projects, both the internal combustion engine (ICE) and the battery are subject to certain operational constraints. The simulation starts with the battery State of Charge (SoC) at 60%, and the control strategy aims to maintain this same value by the end of the simulation, in accordance with the previously mentioned operation known as Charge Sustaining.

DynaProg Problem setup

The DynaProg function must be entered with cell arrays that include:

The nodes of the problem *x_grid* (an array that contains the discretized values of the state variables);

The initial node *x_init (*a single value array that define the initial state of charge);

The final node *x_final* (a double value array that defines a dual final state condition);

The control variables *u_grid* (two arrays containing a control variable vector each);

The exogenous inputs *w* (two arrays containing an exogenous input vector each).

The number of stages, finally, is defined as a vector whose lenght corresponds to the duration of the cycle under analysis in seconds.

```
% Parameters initialization

w_idle = veh.eng.idleSpd; % Engine idle speed (rad/s)
w_max = veh.eng.maxSpd; % Maximum engine speed (rad/s)
w_speed = [0 linspace(w_idle, w_max, 20)]; % (rad/s)

T_max = max(veh.eng.maxTrq(w_speed)); % (Nm)

% State variable grid
x_grid = {0.4:0.01:0.8};
% Initial state
x_init = {0.6};
% Final state constraints
x_final = {[0.59 0.61]};

% Control variable grid
engTrq = linspace(0, T_max, 21);
engSpd = [w_speed];

u_grid = {engSpd, engTrq};

N_stages = length(time); % Length of cycle in seconds

% Definition of the Exogenous Inputs cell arrays

w{1} = vehSpd;

w{2} = vehAcc;

sys = @(x,u,w) hev_cell_model(x,u,w,veh); % Creation of a system function handle
that helps us to converte hev_cell_model from 4 inputs to 3 inputs

prob = DynaProg(x_grid, x_init, x_final, u_grid, N_stages, sys, 'ExogenousInput',
w); % Definition of the problem of optimitazion of Dynamic Programming
```

```
prob = run(prob) % Starts of simulation
```

```
Warning: The state variable grid for SV #1 is very coarse. Consider defining the grid so that at least three
of the grid points lie inside the final state bounds, or loosen the final state bounds.
DP backward progress:100 %
DP forward progress:100 %
prob =
  DynaProg with properties:

                   SysName: @(x,u,w)hev_cell_model(x,u,w,veh)
                SysNameExt: []
                SysNameInt: []
                 StateGrid: {[41×1 double]}
              StateInitial: {[0.6000]}
                StateFinal: {[0.5900 0.6100]}
               ControlGrid: {[0 73.3038 108.0267 142.7496 177.4724 212.1953 246.9182 281.6410 316.3639 351.0868 385
                   Nstages: 1801
            ExogenousInput: [1801×2 double]
               UseLevelSet: 0
        StoreValueFunction: 0
           StoreControlMap: 0
                  SafeMode: 0
                      Time: []
                     myInf: 1000000
          EnforceStateGrid: 1
                 VFPenalty: 'rift'
    LevelSetInitialization: 'linear'
              TerminalCost: @(x)zeros([cellfun(@length,x),1])
                 totalCost: 808.2523
              StateProfile: {[0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0
            ControlProfile: {2×1 cell}
               CostProfile: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.5937 1.2858 0.8026 0 0 0 0 0 0 0 0.8706 1.2858 1.28
         AddOutputsProfile: {[1×1801 struct]}
                ControlMap: []
                      N_SV: 41
                      N_CV: [21 21]
              UseExoInput: 1
             VFPenFactors: 4
                StateName: "x_1"
              ControlName: ["u_1"    "u_2"]
                 CostName: 'Cumulative cost'
                  Display: 'detailed'
                       VF: []
                  Version: "1.6.4"
```

```
profiles = structArray2struct(prob.AddOutputsProfile{1,1}); % Creation of a
structure where it will be store the results of the problem

SOC_1 = profiles.battSOC; % The vecotr of SoC is retrived
```

Estimate the battery's lifetime

In this initial computation stage, the goal is to evaluate the expected battery lifespan using a simplified aging model. This model provides an approximation of battery degradation based on a characteristic degradation factor.

The degradation factor is determined according to the battery capacity using the table below and is used to estimate the total amount of absolute charge the battery can exchange over its lifetime before reaching its end-of-life condition (defined as a State of Health, SOH, of 80%).

This key result is obtained through the linear relationship shown below. By comparing the battery performance data from the selected driving cycle (WLTP), it is then possible to estimate the total distance, in kilometers, that the vehicle can drive before the battery reaches its end-of-life.

$$C_{\text{loss},\%,\text{cycle}} = \beta\, Q_{\text{cycle}}$$

Battery energy (Wh)	β **(1/Ah)**
1260	5.00E-6
1155	5.45E-6
1050	6.00E-6
945	6.67E-6

Results Extrapolation (alpha = 1)

The following section presents and analyzes the simulation results obtained for the case with α = 1. The main outputs include:

- The vector of battery currents, both incoming and outgoing, considered in absolute value and expressed in Amperes (A);
- The total charge exchanged by the battery, calculated as the time integral of the absolute value of the current, expressed in Ampers-hours (Ah);
- The number of WLTP cycles that can be performed before reaching the 80% State of Health (SoH) threshold;
- The total distance traveled, expressed in kilometers (km), corresponding to the battery degradation up to 80% SoH;
- The time profile of the fuel flow rate, expressed in Grams per Second (g/s);
- The total fuel consumption, obtained by integrating the fuel flow rate over time, expressed in both Kilograms (kg) and Liters (l);
- The fuel economy indicator, expressed in Liters per 100 kilometers (l/100 km).

```
Batt_energy = veh.batt.nomEnergy %(Wh) - The battery capacity is shown
```

```
Batt_energy =
945
```

```
beta = 6.67E-6; % (1/Ah) - By enter the table with the above value the degradation
factor beta is retrieved

C_loss = 0.2; % Lost capacity of the battery (1-SOH)
```

```
Q_cycles = C_loss/beta; % Total charge within the battery life-span

curr_WLTP_1 = abs(profiles.battCurr); % The vector in absolute value of the WLTP
current is retrieved

Q_WLTP_1 = trapz(curr_WLTP_1)/3600; % The total absolute charge is computed for the
considered cycle

cycles1 = Q_cycles/Q_WLTP_1; % The number of WLTP cycles is computed as multiple of
the total charge

distance = trapz(vehSpd)/1000; % The WLTP distance in Km is calculated

distance_eol1 = distance * cycles1 % The distance driven to the battery's end-of-
life is estimated
```

```
distance_eol1 =
7.3810e+04
```

```
% Calculate total fuel consumption using numerical integration (trapz)
fuelFlwRate1 = profiles.fuelFlwRate;

fuelConsumption1 = trapz(fuelFlwRate1)/1e3 % (Kg)
```

```
fuelConsumption1 =
0.8083
```

```
% Convert the fuel consumed from mass to volume quantity
Fuel_l_1 = fuelConsumption1/(veh.eng.fuelDensity); % (l)

% Calculate the cycle total dinstance using numerical integration (trapz)
distance = trapz(vehSpd)/1e3; % (Km)

fuelEconomy1 = Fuel_l_1/distance * 100; % (l/100km)

finalSOC1 = SOC_1(end);

% Calculate cumulative fuel consumption using numerical integration (trapz)
% FlwCnsp_dyn = 0; % Initialize the "Dynamic Fuel consumption" variable
FlwCnsp_dyn1 = zeros(length(time), 1);
for n = 1: length(time)-1
    % Perfom iterative summation of each instant fuel consumption contribution
    FlwCnsp_dyn1(n+1) = trapz(fuelFlwRate1(n:n+1)) + FlwCnsp_dyn1(n);
end

% Compute the average dynamic fuel consumption rate (g/s) over the WLTP cycle
meanFlwRate1 = mean(fuelFlwRate1);

% Calculate cumulative charge (q) using numerical integration (trapz)
```

```
% Charge_dyn = 0; % Initialize the "Dynamic Charge" variable
Charge_dyn1 = zeros(length(time), 1);
for n = 1: length(time)-1
    % Perfom iterative summation of each instant charge contribution
    Charge_dyn1(n+1) = trapz(curr_WLTP_1(n:n+1))/3600 + Charge_dyn1(n);
end

% Compute the average absolute battery current (A) over the WLTP cycle
meanCurrent1 = mean(curr_WLTP_1);
```

Save results

```
% Store results
save("results_base.mat", "fuelConsumption1", "fuelEconomy1", "finalSOC1",
"distance_eol1")
```

Developing of an aging-aware EMS with dynamic programming

In this second part of the progect it is asked to refine the control strategy considering that, along with the concerning of the fuel economy, in case of a HEV, some other important resources should be optimezed.

The formula shown is used to calculate a cost function

$$L_k = \alpha \frac{\dot{m}_{\mathrm{f},k}}{\dot{m}_{\mathrm{f,max}}} + (1-\alpha)\frac{|i_{\mathrm{b},k}|}{i_{\mathrm{b,max}}};$$

whre $\dot{m}_{\mathrm{f,max}}$ is the maximum fuel flow rate of the ICE and $i_{\mathrm{b,max}}$ is the limit current of the battery, and they both act as normalization constants.

Alpha is a weighting parameter that controls the balance between fuel consumption and battery usage, if alpha = 1, the cost is based only on fuel use, if alpha = 0, the cost is based only on the battery current

Simulation (Alpha = 0)

In this section, the simulation was performed for the case α = 0. To do so, the stage cost within the `hev_cell_model` function was modified by introducing a variable weighting factor α, aimed at tuning the relative contribution of the terms in the cost function.

To adapt the `hev_cell_model_alpha` function to the format required by the `DynaProg` algorithm (which expects a function with three inputs), a function handle named `sys_alpha` was created. This handle reduces the number of arguments from five to three by fixing the previously defined value of α.

Subsequently, the optimization problem was defined using the `prof_alpha` function, and a data structure was created to store the results obtained from the execution of the `prob_alpha` problem.

```
% Initialization of the alpha parameter, used as a weighting factor in the cost
function
```

```
alpha = 0;

% Definition of a function handle that embeds the fixed alpha value into the
hev_cell_model_alpha function
% This reduces the number of inputs from 5 to 3, as required by the DynaProg
optimizer
sys_alpha = @(x,u,w) hev_cell_model_alpha(x,u,w,veh,alpha);

% Definition of the dynamic optimization problem using DynaProg
% Inputs: state grid, initial and final states, control grid, number of stages,
system dynamics function (sys_alpha), and exogenous input vector w
prob_alpha = DynaProg(x_grid, x_init, x_final, u_grid, N_stages, sys_alpha,
'ExogenousInput', w);

% Execution of the dynamic optimization
prob_alpha = run(prob_alpha)
```

```
Warning: The state variable grid for SV #1 is very coarse. Consider defining the grid so that at least three
of the grid points lie inside the final state bounds, or loosen the final state bounds.
DP backward progress:100 %
DP forward progress:100 %
prob_alpha =
  DynaProg with properties:

                  SysName: @(x,u,w)hev_cell_model_alpha(x,u,w,veh,alpha)
               SysNameExt: []
               SysNameInt: []
                StateGrid: {[41×1 double]}
             StateInitial: {[0.6000]}
               StateFinal: {[0.5900 0.6100]}
              ControlGrid: {[0 73.3038 108.0267 142.7496 177.4724 212.1953 246.9182 281.6410 316.3639 351.0868 385
                  Nstages: 1801
            ExogenousInput: [1801×2 double]
               UseLevelSet: 0
        StoreValueFunction: 0
           StoreControlMap: 0
                  SafeMode: 0
                      Time: []
                     myInf: 1000000
          EnforceStateGrid: 1
                VFPenalty: 'rift'
    LevelSetInitialization: 'linear'
              TerminalCost: @(x)zeros([cellfun(@length,x),1])
                 totalCost: 213.3137
              StateProfile: {[0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0
            ControlProfile: {2×1 cell}
               CostProfile: [0 0 0 0 0 0 0 0 0 0 0 0 0.0017 8.3614e-04 9.0535e-04 0.0022 0.0094 0.0025 0.0134 0.0060
         AddOutputsProfile: {[1×1801 struct]}
                ControlMap: []
                      N_SV: 41
                      N_CV: [21 21]
              UseExoInput: 1
             VFPenFactors: 4
                StateName: "x_1"
              ControlName: ["u_1"    "u_2"]
                 CostName: 'Cumulative cost'
                  Display: 'detailed'
                       VF: []
                  Version: "1.6.4"
```

```
% Conversion of the output profile from struct array to standard struct format
profiles_alpha_0 = structArray2struct(prob_alpha.AddOutputsProfile{1,1});

% Extraction of the battery State of Charge (SOC) profile from the simulation
results
SOC_2 = profiles_alpha_0.battSOC;
```

Results Extrapolation (Alpha = 0)

The following section presents and analyzes the simulation results obtained for the case with α = 0. The main outputs include:

- The vector of battery currents, both incoming and outgoing, considered in absolute value and expressed in amperes (A);
- The total charge exchanged by the battery, calculated as the time integral of the absolute value of the current, expressed in coulombs (Ah);
- The number of WLTP cycles that can be performed before reaching the 80% State of Health (SoH) threshold;
- The total distance traveled, expressed in kilometers (km), corresponding to the battery degradation up to 80% SoH;
- The time profile of the fuel flow rate, expressed in grams per second (g/s);
- The total fuel consumption, obtained by integrating the fuel flow rate over time, expressed in both kilograms (kg) and liters (l);
- The fuel economy indicator, expressed in liters per 100 kilometers (l/100 km).

```
% Extract the battery current profile and take the absolute value to account for
both charge and discharge phases
curr_WLTP_2 = abs(profiles_alpha_0.battCurr);

% Compute the total charge exchanged during the WLTP cycle (in Ah) using numerical
integration (trapz)
Q_WLTP_2 = trapz(curr_WLTP_2)/3600;

% Estimate the number of WLTP cycles the battery can perform before reaching the
predefined SoH degradation
cycles2 = Q_cycles/Q_WLTP_2;

% Calculate the total distance driven until the end-of-life (EoL) of the battery
distance_eol2 = distance * cycles2
```

```
distance_eol2 =
1.2918e+05
```

```
% Calculate total fuel consumption using numerical integration (trapz)
fuelFlwRate2 = profiles_alpha_0.fuelFlwRate;
```

```
fuelConsumption2 = trapz(fuelFlwRate2)/1e3 % (Kg)
```

```
fuelConsumption2 =
1.0688
```

```
% Convert the fuel consumed from mass to volume quantity
Fuel_l_2 = fuelConsumption2/(veh.eng.fuelDensity); % (l)

fuelEconomy2 = Fuel_l_2/distance * 100; % (l/100km)

% Store the final State of Charge (SOC) value at the end of the cycle
finalSOC2 = SOC_2(end);
```

Simulation (Alpha = 0.4)

In this section, the simulation was performed for the case α = 0.4. To do so, the stage cost within the `hev_cell_model_alpha` function was modified by introducing a variable weighting factor α, aimed at tuning the relative contribution of the terms in the cost function.

To adapt the `hev_cell_model_alpha` function to the format required by the `DynaProg` algorithm (which expects a function with three inputs), a function handle named `sys_alpha` was created. This handle reduces the number of arguments from five to three by fixing the previously defined value of α.

Subsequently, the optimization problem was defined using the `prof_alpha` function, and a data structure was created to store the results obtained from the execution of the `prob_alpha` problem.

```
% Initialization of the alpha parameter, used as a weighting factor in the cost
function
alpha_trade_off = 0.4;

% Definition of a function handle that embeds the fixed alpha value into the
hev_cell_model_alpha function
% This reduces the number of inputs from 5 to 3, as required by the DynaProg
optimizer
sys_alpha = @(x,u,w) hev_cell_model_alpha(x,u,w,veh,alpha_trade_off);

% Definition of the dynamic optimization problem using DynaProg
% Inputs: state grid, initial and final states, control grid, number of stages,
system dynamics function (sys_alpha), and exogenous input vector w
prob_alpha = DynaProg(x_grid, x_init, x_final, u_grid, N_stages, sys_alpha,
'ExogenousInput', w);

% Execution of the dynamic optimization
prob_alpha = run(prob_alpha)
```

```
Warning: The state variable grid for SV #1 is very coarse. Consider defining the grid so that at least three
of the grid points lie inside the final state bounds, or loosen the final state bounds.
DP backward progress:100 %
DP forward progress:100 %
prob_alpha =
  DynaProg with properties:
```

```
                SysName: @(x,u,w)hev_cell_model_alpha(x,u,w,veh,alpha_trade_off)
             SysNameExt: []
             SysNameInt: []
              StateGrid: {[41×1 double]}
           StateInitial: {[0.6000]}
             StateFinal: {[0.5900 0.6100]}
            ControlGrid: {[0 73.3038 108.0267 142.7496 177.4724 212.1953 246.9182 281.6410 316.3639 351.0868 385.
                Nstages: 1801
          ExogenousInput: [1801×2 double]
             UseLevelSet: 0
      StoreValueFunction: 0
         StoreControlMap: 0
                SafeMode: 0
                    Time: []
                   myInf: 1000000
        EnforceStateGrid: 1
               VFPenalty: 'rift'
  LevelSetInitialization: 'linear'
            TerminalCost: @(x)zeros([cellfun(@length,x),1])
               totalCost: 188.7575
            StateProfile: {[0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.
          ControlProfile: {2×1 cell}
             CostProfile: [0 0 0 0 0 0 0 0 0 0 0 0 9.9468e-04 0.0226 0.1119 0.0294 0.0447 0.0559 0.0710 0.0554 0.0
       AddOutputsProfile: {[1×1801 struct]}
              ControlMap: []
                    N_SV: 41
                    N_CV: [21 21]
             UseExoInput: 1
            VFPenFactors: 4
               StateName: "x_1"
             ControlName: ["u_1"    "u_2"]
                CostName: 'Cumulative cost'
                 Display: 'detailed'
                      VF: []
                 Version: "1.6.4"
```

```
% Conversion of the output profile from struct array to standard struct format
profiles_alpha_0_4 = structArray2struct(prob_alpha.AddOutputsProfile{1,1});

% Extraction of the battery State of Charge (SOC) profile from the simulation
results
SOC_3 = profiles_alpha_0_4.battSOC;
```

Results Extrapolation (alpha = 0.4)

The following section presents and analyzes the simulation results obtained for the case with α = 0.4. The main outputs include:

- The vector of battery currents, both incoming and outgoing, considered in absolute value and expressed in amperes (A);
- The total charge exchanged by the battery, calculated as the time integral of the absolute value of the current, expressed in coulombs (Ah);
- The number of WLTP cycles that can be performed before reaching the 80% State of Health (SoH) threshold;

- The total distance traveled, expressed in kilometers (km), corresponding to the battery degradation up to 80% SoH;
- The time profile of the fuel flow rate, expressed in grams per second (g/s);
- The total fuel consumption, obtained by integrating the fuel flow rate over time, expressed in both kilograms (kg) and liters (l);
- The fuel economy indicator, expressed in liters per 100 kilometers (l/100 km).

```
% Extract the battery current profile and take the absolute value to account for
both charge and discharge phases
curr_WLTP_3 = abs(profiles_alpha_0_4.battCurr);

% Compute the total charge exchanged during the WLTP cycle (in Ah) using numerical
integration (trapz)
Q_WLTP_3 = trapz(curr_WLTP_3)/3600;

% Estimate the number of WLTP cycles the battery can perform before reaching the
predefined SoH degradation
cycles3 = Q_cycles/Q_WLTP_3;

% Calculate the total distance driven until the end-of-life (EoL) of the battery
distance_eol3 = distance * cycles3
```

```
distance_eol3 =
1.2696e+05
```

```
% Calculate total fuel consumption using numerical integration (trapz)
fuelFlwRate3 = profiles_alpha_0_4.fuelFlwRate;

fuelConsumption3 = trapz(fuelFlwRate3)/1e3 % (Kg)
```

```
fuelConsumption3 =
0.8193
```

```
% Convert the fuel consumed from mass to volume quantity
Fuel_l_3 = fuelConsumption3/(veh.eng.fuelDensity); % (l)

% Conversion of the output profile from struct array to standard struct format
fuelEconomy3 = Fuel_l_3/distance * 100; % (l/100km)

% Extraction of the battery State of Charge (SOC) profile from the simulation
results
finalSOC3 = SOC_3(end);

% Calculate cumulative fuel consumption using numerical integration (trapz)
% FlwCnsp_dyn = 0; % Initialize the "Dynamic Fuel consumption" variable
FlwCnsp_dyn3 = zeros(length(time), 1);
for n = 1: length(time)-1
    % Perfom iterative summation of each instant fuel consumption contribution
```

```
    FlwCnsp_dyn3(n+1) = trapz(fuelFlwRate3(n:n+1)) + FlwCnsp_dyn3(n);
end

% Compute the average dynamic fuel consumption rate (g/s) over the WLTP cycle
meanFlwRate3 = mean(fuelFlwRate3);

% Calculate cumulative charge (q) using numerical integration (trapz)
% Charge_dyn = 0; % Initialize the "Dynamic Charge" variable
Charge_dyn3 = zeros(length(time), 1);
for n = 1: length(time)-1
    % Perfom iterative summation of each instant charge contribution
    Charge_dyn3(n+1) = trapz(curr_WLTP_3(n:n+1))/3600 + Charge_dyn3(n);
end

% Compute the average absolute battery current (A) over the WLTP cycle
meanCurrent3 = mean(curr_WLTP_3);
```

Simulation (Alpha = 0.7)

In this section, the simulation was performed for the case α = 0.7. To do so, the stage cost within the `hev_cell_model_alpha` function was modified by introducing a variable weighting factor α, aimed at tuning the relative contribution of the terms in the cost function.

To adapt the `hev_cell_model_alpha` function to the format required by the `DynaProg` algorithm (which expects a function with three inputs), a function handle named `sys_alpha` was created. This handle reduces the number of arguments from five to three by fixing the previously defined value of α.

Subsequently, the optimization problem was defined using the `prof_alpha` function, and a data structure was created to store the results obtained from the execution of the `prob_alpha` problem.

```
% Initialization of the alpha parameter, used as a weighting factor in the cost
function
alpha = 0.7;

% Definition of a function handle that embeds the fixed alpha value into the
hev_cell_model_alpha function
% This reduces the number of inputs from 5 to 3, as required by the DynaProg
optimizer
sys_alpha = @(x,u,w) hev_cell_model_alpha(x,u,w,veh,alpha);

% Definition of the dynamic optimization problem using DynaProg
% Inputs: state grid, initial and final states, control grid, number of stages,
system dynamics function (sys_alpha), and exogenous input vector w
prob_alpha = DynaProg(x_grid, x_init, x_final, u_grid, N_stages, sys_alpha,
'ExogenousInput', w);

% Execution of the dynamic optimization
prob_alpha = run(prob_alpha)
```

```
Warning: The state variable grid for SV #1 is very coarse. Consider defining the grid so that at least three
of the grid points lie inside the final state bounds, or loosen the final state bounds.
DP backward progress:100 %
DP forward progress:100 %
prob_alpha =
  DynaProg with properties:

                  SysName: @(x,u,w)hev_cell_model_alpha(x,u,w,veh,alpha)
               SysNameExt: []
               SysNameInt: []
                StateGrid: {[41×1 double]}
             StateInitial: {[0.6000]}
               StateFinal: {[0.5900 0.6100]}
              ControlGrid: {[0 73.3038 108.0267 142.7496 177.4724 212.1953 246.9182 281.6410 316.3639 351.0868 385.
                  Nstages: 1801
           ExogenousInput: [1801×2 double]
              UseLevelSet: 0
       StoreValueFunction: 0
          StoreControlMap: 0
                 SafeMode: 0
                     Time: []
                    myInf: 1000000
        EnforceStateGrid: 1
                VFPenalty: 'rift'
   LevelSetInitialization: 'linear'
             TerminalCost: @(x)zeros([cellfun(@length,x),1])
                totalCost: 167.4189
             StateProfile: {[0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.6000 0.
          ControlProfile: {2×1 cell}
              CostProfile: [0 0 0 0 0 0 0 0 0 0 0 0 4.9734e-04 0.0113 0.0560 0.0922 0.0605 0.0861 0.1133 0.0913 0.0
       AddOutputsProfile: {[1×1801 struct]}
              ControlMap: []
                    N_SV: 41
                    N_CV: [21 21]
             UseExoInput: 1
            VFPenFactors: 4
               StateName: "x_1"
             ControlName: ["u_1"    "u_2"]
                CostName: 'Cumulative cost'
                 Display: 'detailed'
                      VF: []
                 Version: "1.6.4"
```

```
% Conversion of the output profile from struct array to standard struct format
profiles_alpha_0_7 = structArray2struct(prob_alpha.AddOutputsProfile{1,1});

% Store the final State of Charge (SOC) value at the end of the cycle
SOC_4 = profiles_alpha_0_7.battSOC;
```

Results Extrapolation (alpha = 0.7)

The following section presents and analyzes the simulation results obtained for the case with α = 0.7. The main outputs include:

- The vector of battery currents, both incoming and outgoing, considered in absolute value and expressed in amperes (A);

- The total charge exchanged by the battery, calculated as the time integral of the absolute value of the current, expressed in coulombs (Ah);
- The number of WLTP cycles that can be performed before reaching the 80% State of Health (SoH) threshold;
- The total distance traveled, expressed in kilometers (km), corresponding to the battery degradation up to 80% SoH;
- The time profile of the fuel flow rate, expressed in grams per second (g/s);
- The total fuel consumption, obtained by integrating the fuel flow rate over time, expressed in both kilograms (kg) and liters (l);
- The fuel economy indicator, expressed in liters per 100 kilometers (l/100 km).

```
% Extract the battery current profile and take the absolute value to account for
both charge and discharge phases
curr_WLTP_4 = abs(profiles_alpha_0_7.battCurr);

% Compute the total charge exchanged during the WLTP cycle (in Ah) using numerical
integration (trapz)
Q_WLTP_4 = trapz(curr_WLTP_4)/3600;

% Estimate the number of WLTP cycles the battery can perform before reaching the
predefined SoH degradation
cycles4 = Q_cycles/Q_WLTP_4;

% Calculate the total distance driven until the end-of-life (EoL) of the battery
distance_eol4 = distance * cycles4
```

```
distance_eol4 =
1.2683e+05
```

```
% Calculate total fuel consumption using numerical integration (trapz)
fuelFlwRate4 = profiles_alpha_0_4.fuelFlwRate;

fuelConsumption4 = trapz(fuelFlwRate4)/1e3 % (Kg)
```

```
fuelConsumption4 =
0.8193
```

```
% Convert the fuel consumed from mass to volume quantity
Fuel_l_4 = fuelConsumption4/(veh.eng.fuelDensity); % (l)

% Conversion of the output profile from struct array to standard struct format
fuelEconomy4 = Fuel_l_4/distance * 100; % (l/100km)

% Extraction of the battery State of Charge (SOC) profile from the simulation
results
finalSOC4 = SOC_4(end);
```

Post - Processing of Results (Analysis of the fuel-optimal EMS and aging-aware EMS)

SOC - State of Charge

In this first graph it is possible to assess differents behaviour of the SoC with 4 alpha's values; 1, 0, 0.4 and 0.7 respectively.

Calculated using Dynamic Programming. There are four different curves, each representing a different value of alpha:

- Red line: alpha = 1
- Blue line: alpha = 0
- Green line: alpha = 0.4
- Black line: alpha = 0.7

From the graph, we can observe the effect of changing alpha:

- When alpha = 0 (blue line), the system tries to minimize only battery usage resulting in a general SoC increase as consequence of the total lack of cooperation between the two sources since, at lower loads, the battery neither take part of the traction power requested nor accept supplementary energy from some ICE load shifiting criteria. The only power that the battery is forced to accept, and then to give back necessarily, is the one from regenerative braking phases. The traction power, in summary, is almost totally provided by the ICE except for the battery energy derived from the brakings that is opportunally released.
- When alpha = 1 (red line), the system focuses on minimizing fuel consumption, so the battery is used intensively and the SoC decreases more strongly presenting more pure electric modes and charging phases. The battery therefore is exploited at the maximum of its capability to meet the economic policy related to the use of the ICE.
- The green (alpha = 0.4) and black (alpha = 0.7) lines show intermediate behavior between the two extremes demostrating that a small decrease of the weighting factor alpha results in an effective change in the battery managment strategy.

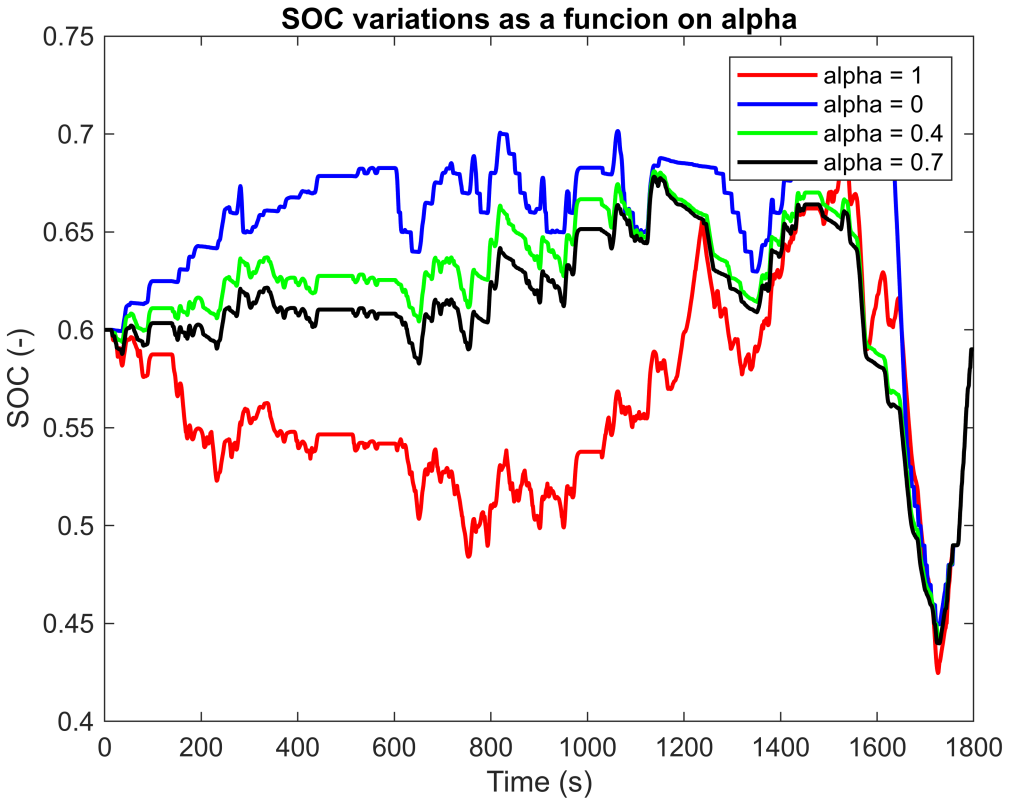
The 0.4 scenario is selected as candidate for the evaluation of the best trade off value. The reasons and consequences of this choice will be explained in detail in the description of the next graphs.

To sum up, the higher the alpha, the more the system exploites the battery to save fuel, and this is visible in the SoC curve dropping lower during the first phase over time (from 0 to 1250 more or less) . On the other hand, lower alpha values result in less battery use, and a higher average SoC over time.

This graph clearly shows how the cooperation between fuel and battery usage changes with different alpha values.

```
figure
plot(time,SOC_1,  'r-', 'LineWidth', 1.5)
hold on
plot(time,SOC_2,  'b-', 'LineWidth', 1.5)
hold on
```

```
plot(time,SOC_3,  'g-', 'LineWidth', 1.5)
hold on
plot(time,SOC_4,  'k-', 'LineWidth', 1.5)
legend("alpha = 1", "alpha = 0", "alpha = 0.4", "alpha = 0.7")
title("SOC variations as a funcion on alpha", 'FontWeight', 'bold')
xlabel("Time (s)");
ylabel("SOC (-)");
```



Resources utilization as function of alpha

In this second graph, the plot of the fuel economy (red curve) and the distance that the battery can cover within its life-span (blue curve) for different values of weighting factor has been carried out.

Thanks to the interpolation made over the results obained from the four simulations performed before, a cubic poliniomal function is exploited to approximate the performance of the control strategy for all values of alpha between 0 and 1.

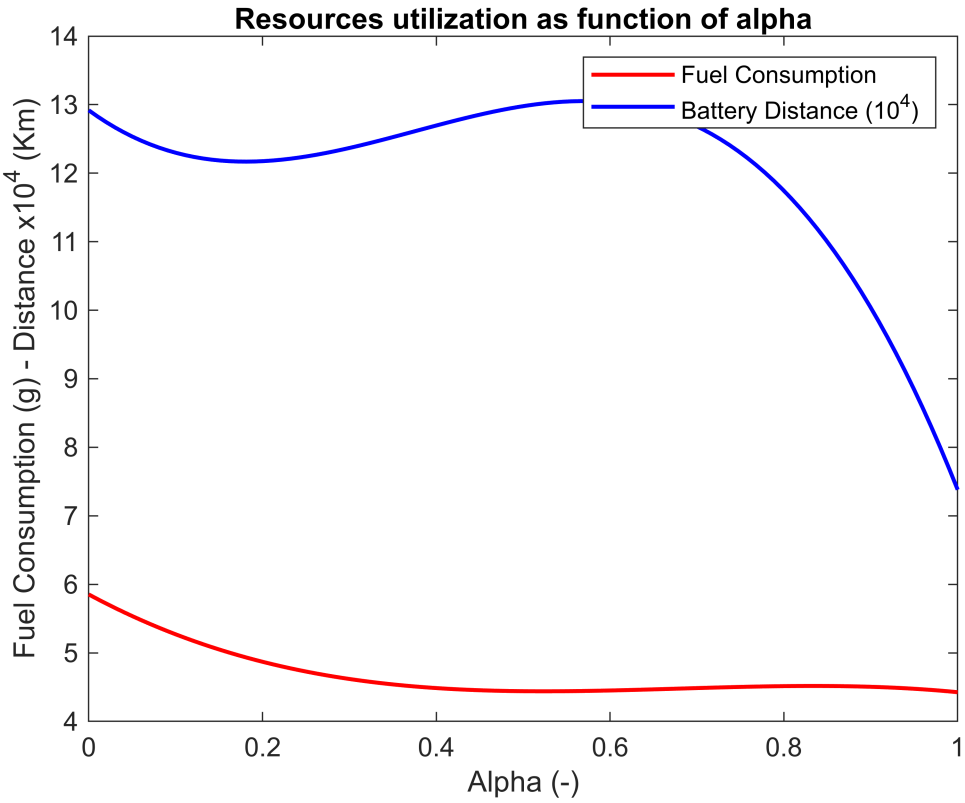
The blue curve shows how many kilometers the vehicle can travel before the battery wears out. When alpha is close to 0 (on the left side of the graph), the battery lasts for more kilometers because the strategy tries to minimize battery usage without caring of how much and when the usage of the internal combustion engine (ICE) is needed.

On the other hand, when alpha is close to 1 (right side of the graph), the strategy tries to minimize the fuel consumption, so, conversely, more energy is required from the battery to enable the ICE to operate close to its optimum operating points. The battery, therefore, is used more intensively and wears out faster, meaning it covers fewer kilometers, while the fuel consumption, shown as L/100 km, is minimized untill its absolute values.

By observing the graph, it can be understood that alpha equal 0.4 can be chosen as candidate for the selection of a trade-off value. In corrispondance of it, in fact, the red curve doesn't show a relevant increase, while the blue curve approaches very quickly its maximum.

The introduction of an ageing cost in the EMS increase battery lifetime significally since its extension can be obtained through a small penality on the fuel consumption that can be considered convinient taking into account the total expenditures within the entire vehicle life.

```
alpha = [0 0.4 0.7 1]; % Vector of alpha values used to weight the cost function in
different simulations
Fuel_weighted = [fuelEconomy2 fuelEconomy3 fuelEconomy4 fuelEconomy1]; %
Corresponding fuel economy values (l/100km) for each alpha
c = polyfit(alpha, Fuel_weighted, 3); % Fit a 3rd-degree polynomial to model the
fuel economy trend as a function of alpha
x_fit = linspace(min(alpha), max(alpha), 100); % Create a dense set of alpha values
for a smooth curve
y_fit_fuel = polyval(c, x_fit); % Evaluate the fitted polynomial for fuel economy
over the dense alpha range

batt_distance = [distance_eol2 distance_eol3 distance_eol4 distance_eol1]/1e4; %
Normalize battery lifespan distance values (in units of 10,000 km) for each alpha
p = polyfit(alpha, batt_distance, 3); % Fit a 3rd-degree polynomial to the battery
distance data
y_fit_batt = polyval(p, x_fit); % Evaluate the fitted polynomial for battery
distance over the dense alpha range

figure
% Plot the fuel economy curve (red)
plot(x_fit,y_fit_fuel,'r-', 'LineWidth', 1.5)
hold on
% Overlay the battery distance curve (blue)
plot(x_fit,y_fit_batt,'b-', 'LineWidth', 1.5)
legend("Fuel Consumption", "Battery Distance (10^4)")
title("Resources utilization as function of alpha", 'FontWeight', 'bold')
xlabel("Alpha (-)");
ylabel("Fuel Consumption (g) - Distance x10^4 (Km)");
```



Engine operating points (α = 1, 0.4, 0)

In this series of three graphs, we decided to show the engine operating points for different values of the usual parameter alpha, considering only three specific values: 1, 0.4 and 0.

We can see that for the first two alpha values, 1 and 0.4, the engine behavior is not very different. However, it is clear that, as already explained, for alpha = 1, the engine works at higher efficiency points compared to alpha = 0.4, while for alpha = 0, the strategy focuses only on reducing battery aging, without considering fuel consumption.

Let's now analyze each case in more detail:

In the first case (alpha = 1), the strategy aims only to minimize fuel consumption, ignoring battery wear. With this value of alpha, the battery always works between Charge Depleting (CD) and Battery Charging (BC) phases to support the ICE (Internal Combustion Engine) as much as possible. However, this intensive usage of this two operating modes (CD and BC) accelerates battery aging and limits its usable lifetime, reducing the total kilometers the battery was designed to cover and increasing its degradation.

In the trade-off case (alpha = 0.4), there is a better handling between fuel usage and battery usage, as we saw in the previous graph. At the same time, analyzing the engine operating points, it can be noticed notice that the ICE still works mostly at high-efficiency points (similar to alpha = 1), but also operating without a power shifting and accepting to work in some less convinient regions taking the battery degradation into account . This is because, as alpha moves away from 1, the strategy starts to limit battery usage in certain phases, balancing performance and durability.

For the third and last case (alpha = 0), it is immediately clear that the engine is working almost in every operating points suggesting an usage more similar to an elettric trasmission. Very often the ICE is forced to work in low-efficiency areas therefore increasing fuel consumption. This happens because, as explained before, with very low alpha values, the strategy tries only to reduce battery aging. However, we can clearly consider this case, like alpha = 1, as an extreme scenario. The imbalance between ICE use and battery use is too large and not realistic.

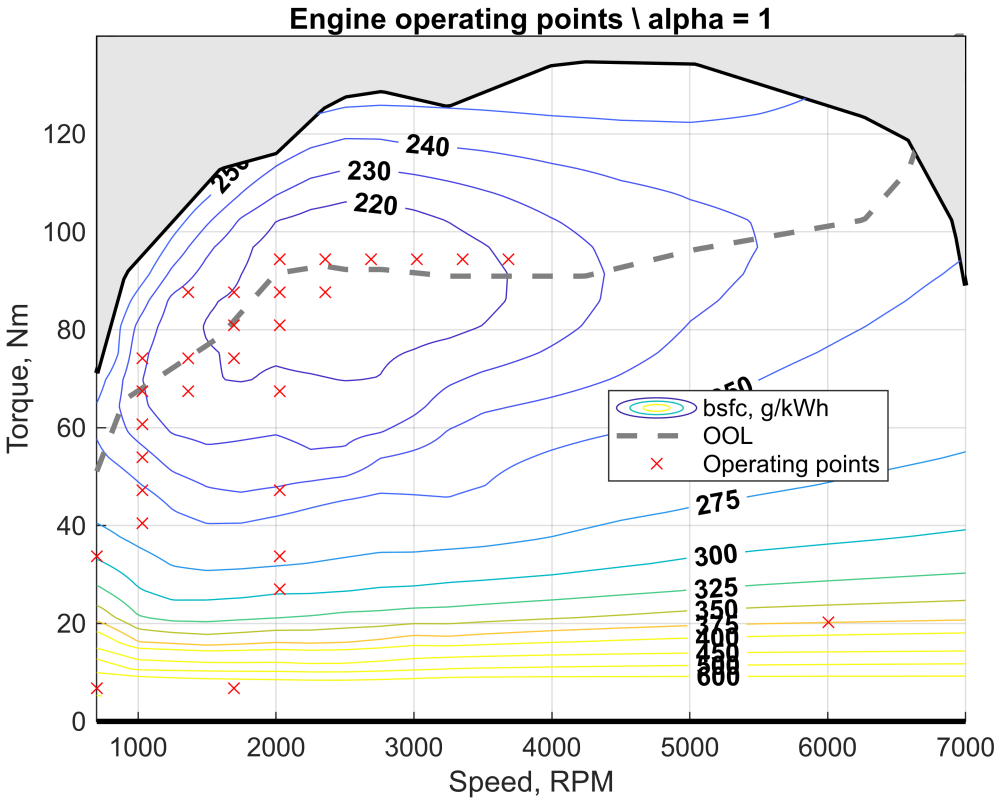
Still, this comparison helps us understand how our control strategy works:

With alpha values close to 1, it mainly focuses on reducing fuel consumption.

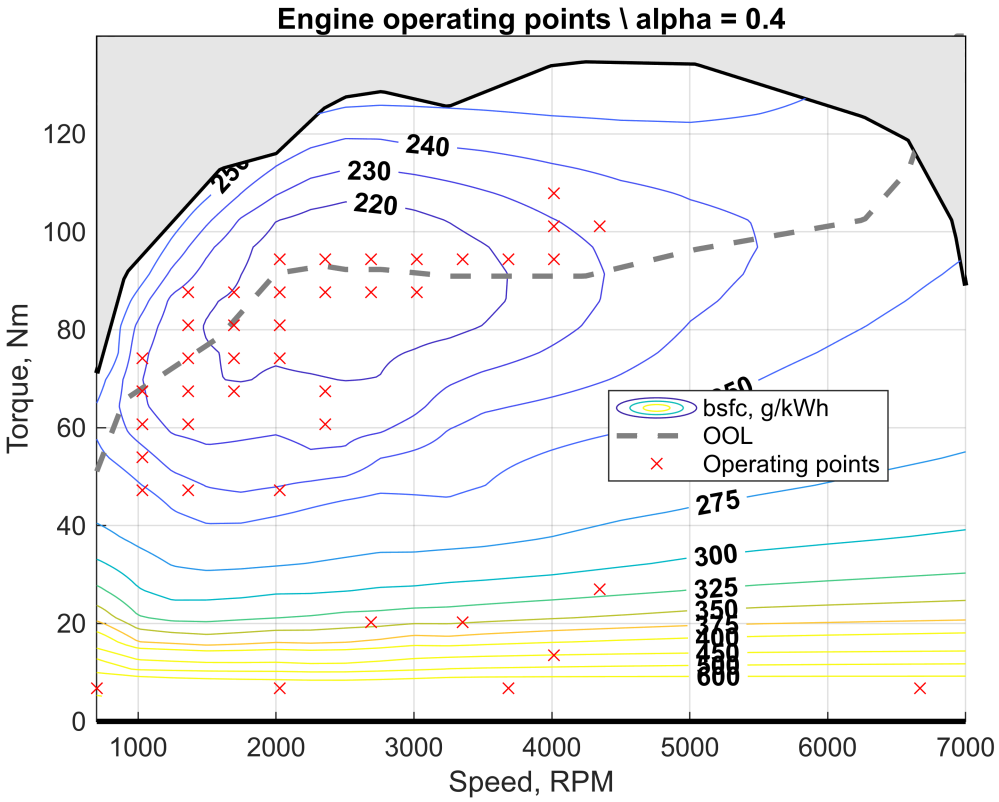
With alpha values close to 0, it aims to maximize battery lifespan, allowing the vehicle to cover as many kilometers as possible using the battery.

The goal is to find the right compromise that allows the powertrain to operate in the most efficient and balanced way.

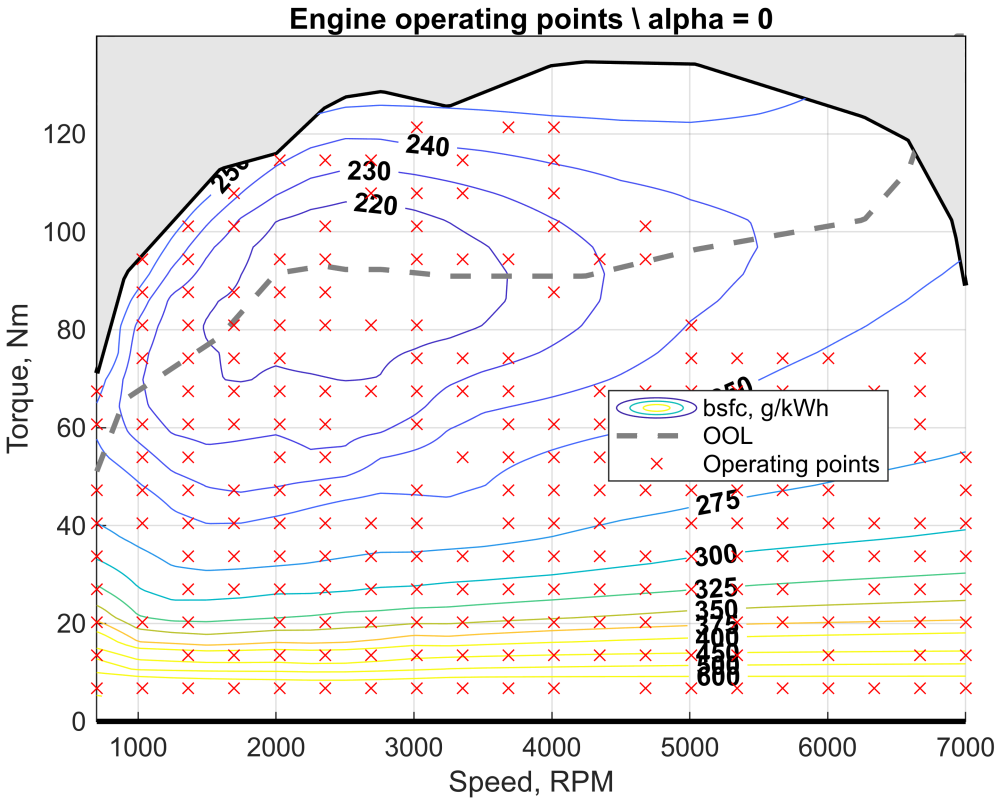
```
engMapWithPF(veh.eng, profiles, 'bsfc');
title("Engine operating points \ alpha = 1", 'FontWeight', 'bold')
```




```
engMapWithPF(veh.eng, profiles_alpha_0_4, 'bsfc');
title("Engine operating points \ alpha = 0.4", 'FontWeight', 'bold')
```




```
engMapWithPF(veh.eng, profiles_alpha_0, 'bsfc');
title("Engine operating points \ alpha = 0", 'FontWeight', 'bold')
```



Instantaneous and cumulative fuel consumption (α = 1, 0.4)

This section presents the plots of both instantaneous and cumulative fuel consumption for the hybrid vehicle, considering the case with α = 1. The final fuel consumption over the entire cycle is approximately 808 g, with a trend that approximate the power demand profile but consider alternative operations to optimize its working functioning. During phases of rapid acceleration, however, the specific fuel consumption increases instantaneously, leading to corresponding updates in the total fuel consumed.

The fuel flow rate reaches a maximum value of around 2 g/s, without any significant peaks.

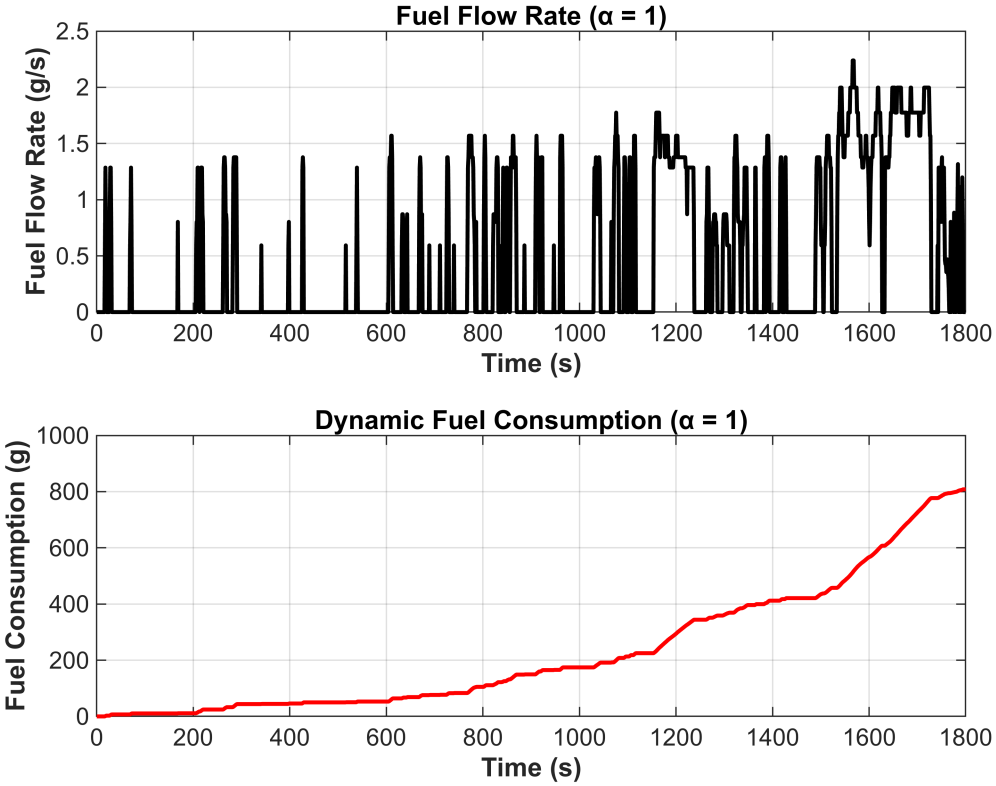
The analysis was then repeated for the trade-off case with α = 0.4, again plotting both the instantaneous and cumulative fuel consumption. In this case, the total fuel consumption is slightly higher, reaching approximately 819 g. This increase is due to the fact that, with α = 0.4 , greater weight is given to the current flowing in and out of the battery, resulting in reduced battery usage but a slightly higher fuel demand.

It is also observed that, in this second scenario, the internal combustion engine operates with less uniform power delivery over time, showing instantaneous fuel consumption peaks up to 3 g/s.

Ultimately, the optimization of the α parameter allows for a reduction in battery degradation at the expense of a marginal increase in fuel consumption, highlighting an effective trade-off between system durability and energy performance.

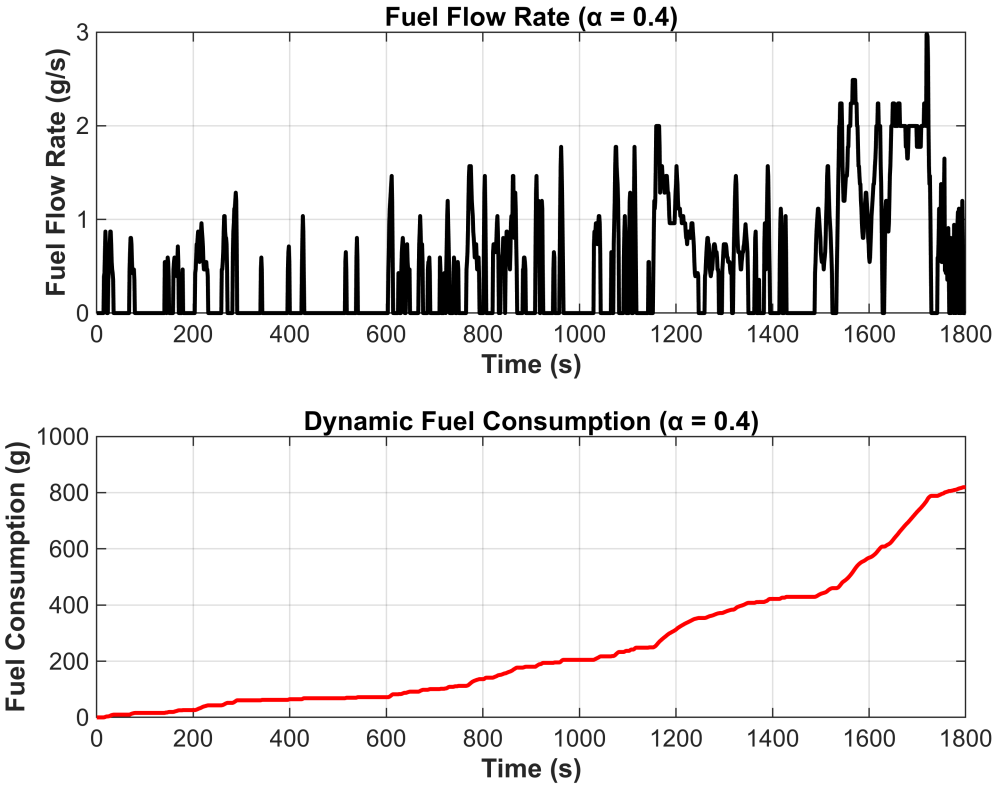
```
% Fuel Flow Rate alpha = 1
figure;
ax1 = nexttile;
plot(time, fuelFlwRate1, 'k-', 'LineWidth', 1.5)
```

```
grid on
title("Fuel Flow Rate (α = 1)", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Fuel Flow Rate (g/s)", 'FontWeight', 'bold')

% Dynamic Fuel Consumption alpha = 1
ax2 = nexttile;
plot(time, FlwCnsp_dyn1, 'r-', 'LineWidth', 1.5)
grid on
title("Dynamic Fuel Consumption (α = 1)", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Fuel Consumption (g)", 'FontWeight', 'bold')

linkaxes([ax1, ax2], 'x')
```




```
% Fuel Flow Rate alpha = 0.4
figure;
ax1 = nexttile;
plot(time, fuelFlwRate3, 'k-', 'LineWidth', 1.5)
grid on
title("Fuel Flow Rate (α = 0.4)", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Fuel Flow Rate (g/s)", 'FontWeight', 'bold')

% Dynamic Fuel Consumption alpha = 0.4
ax2 = nexttile;
```

```
plot(time, FlwCnsp_dyn3, 'r-', 'LineWidth', 1.5)
grid on
title("Dynamic Fuel Consumption (α = 0.4)", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Fuel Consumption (g)", 'FontWeight', 'bold')

linkaxes([ax1, ax2], 'x')
```



Instantaneous Current and Dynamic Charge exchange (α = 1, 0.4)

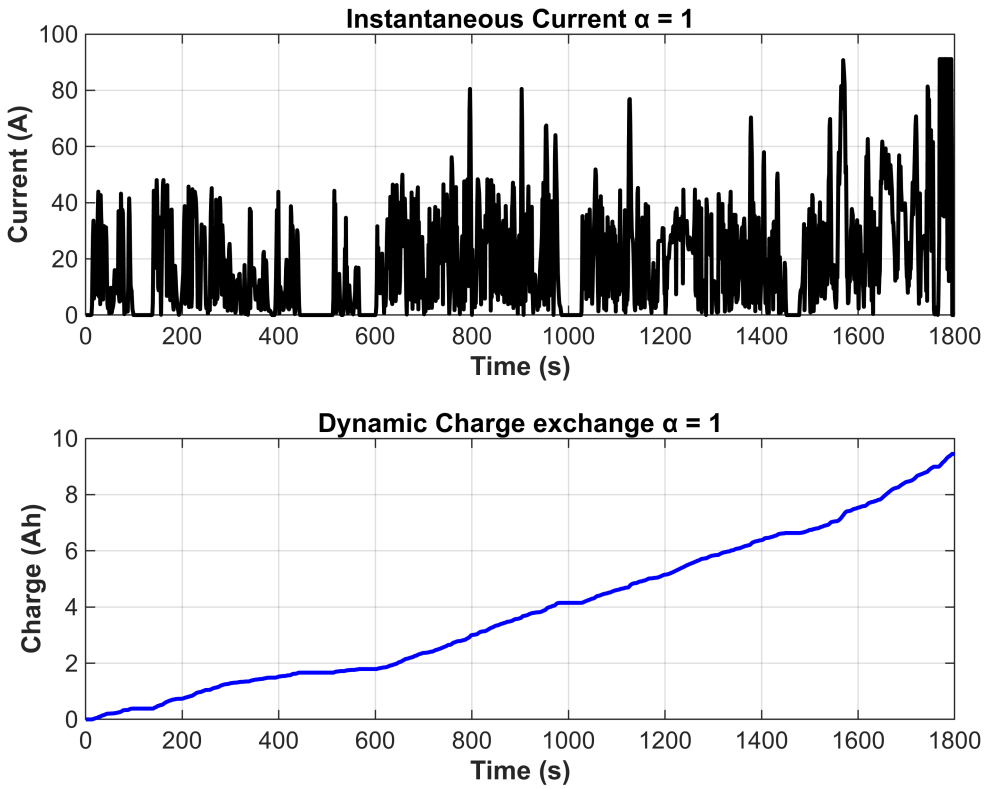
This section presents the plots of the absolute values of the incoming and outgoing battery current at each time step, along with the total charge exchanged (both in and out) for different values of the parameter α. Specifically, the cases α = 1 e α = 0.7 the latter representing the optimized value have been analyzed.

From the first plot, corresponding to α = 1, the total exchanged charge is approximately 9,44 Ah, which is significantly higher than the 5.49 Ah observed in the second case. This behavior is due to the fact that, with α = 1, the battery current is not penalized in the cost function and therefore is not minimized during the optimization process.

Conversely, in the second plot (with α = 0.4), the introduction of a weight on the battery current leads to a significant reduction in the total exchanged charge. This indicates a more efficient use of the battery, helping to reduce its premature aging. It is noteworthy that the peak value of the instantaneous current remains almost unchanged compared to the previous case, while the average current is lower, indicating a less intensive usage of the battery.

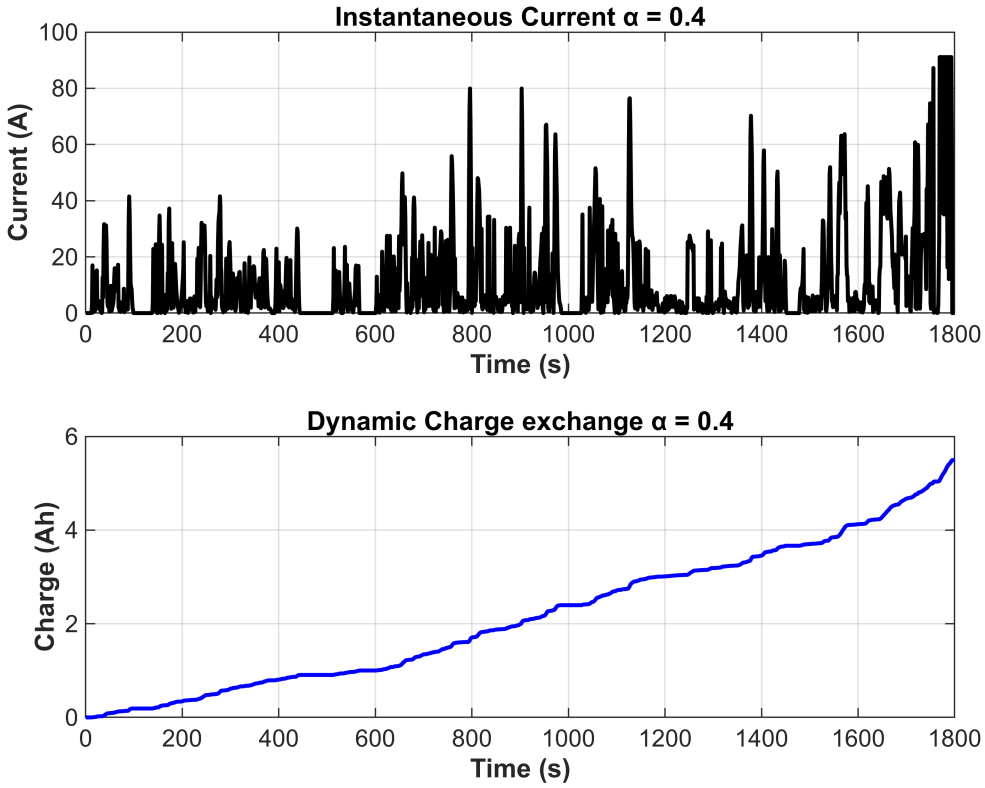
```
% Current alpha = 1
```

```
figure;
ax1 = nexttile;
plot(time, curr_WLTP_1, 'k-', 'LineWidth', 1.5)
grid on
title("Instantaneous Current α = 1", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Current (A)", 'FontWeight', 'bold')

% Dynamic Charge alpha = 1
ax2 = nexttile;
plot(time, Charge_dyn1, 'b-', 'LineWidth', 1.5)
grid on
title("Dynamic Charge exchange α = 1", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Charge (Ah)", 'FontWeight', 'bold')

linkaxes([ax1, ax2], 'x')
```




```
% Current alpha = 0.4
figure;
ax1 = nexttile;
plot(time, curr_WLTP_3, 'k-', 'LineWidth', 1.5)
grid on
title("Instantaneous Current α = 0.4", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Current (A)", 'FontWeight', 'bold')
```

```
% Dynamic Charge alpha = 0.4
ax2 = nexttile;
plot(time, Charge_dyn3, 'b-', 'LineWidth', 1.5)
grid on
title("Dynamic Charge exchange α = 0.4", 'FontWeight', 'bold')
xlabel("Time (s)", 'FontWeight', 'bold')
ylabel("Charge (Ah)", 'FontWeight', 'bold')

linkaxes([ax1, ax2], 'x')
```



Power profiles (α = 1, 0, 0.4, 0.7)

In this section, the different engine modes implemented during the prescribed driving cycle and the corresponding internal combustion engine (ICE) power profiles have been analyzed for all considered values of the α parameter (α = 1, α = 0.4, α = 0.7, α = 0).

Case α = 1

When α = 1, the control strategy is exclusively focused on fuel consumption minimization. This behavior is evidenced by the regular ICE activation pattern, which allows the engine to work less frequently and on higher and efficient loads. The average power output is around 20 kW with peaks reaching 35 kW. The system continuously alternates between battery charging and charge depleting phases, thereby optimizing overall energy efficiency.

Case α = 0

For α = 0, the control strategy prioritizes battery current flow management to minimize aging effects. This results in nearly exclusive operation in pure electric and especially charge depleting modes, with the ICE activated more frequently to meet power demand requirements, never for battery recharge purposes but, as already said, to operate in an elettric transition type . The power profile exhibits greater discontinuity compared to the previous case, with peaks up to 45 kW, as the ICE operates intermittently to accommodate instantaneous power demand variations without contributing to battery charging.

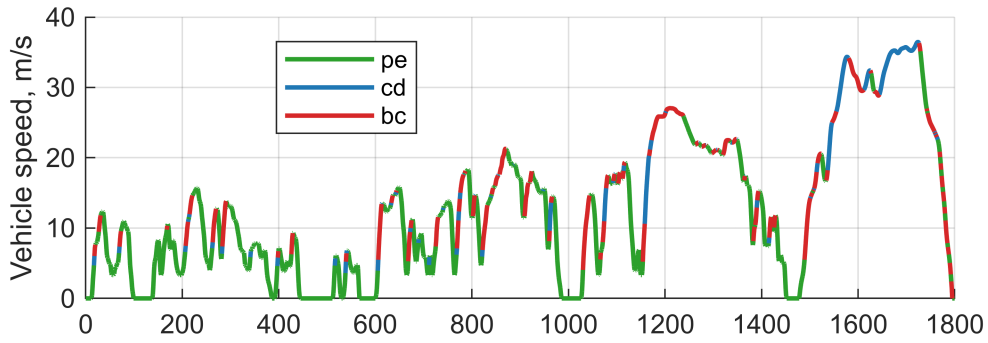
Case α = 0.7

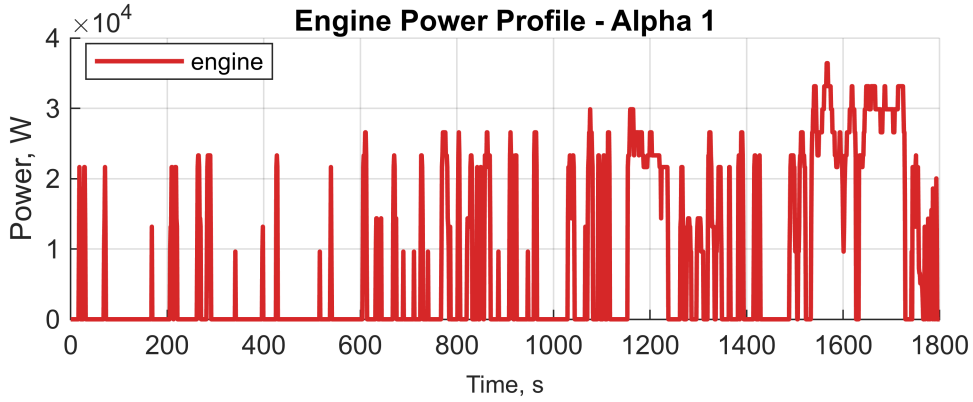
For α = 0.7, the cost function minimization is primarily governed by the battery current flow management component. This approach results in limited utilization of battery charging mode, while favoring charge depleting and pure electric modes instead. The power profile analysis reveals ICE operation strictly correlated with the instantaneous cycle demand, featuring frequent start-stop transitions, significant output power variability, and peaks reaching 45 kW.

Case α = 0.4 (Trade-off)

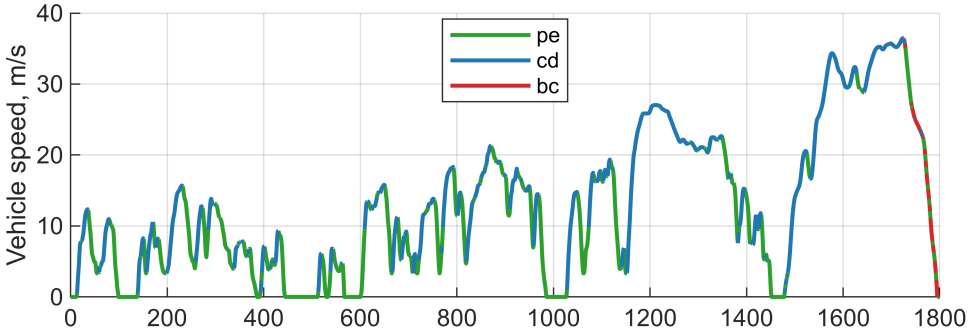
For α = 0.4, the system implements a hybrid strategy that balances all three available operating modes. This configuration represents an optimal trade-off between fuel consumption reduction and battery stress minimization. The temporal power profile nevertheless exhibits dynamic behavior similar to the α = 0.7 case, characterized by frequent operational state transitions, substantial instantaneous power fluctuations, and power peaks but achieving an overall efficient operation very similar to the ones featured by the larger values of α. This choice thus enables the reconciliation of competing requirements between energy efficiency and battery preservation, while maintaining the thermal propulsion system's dynamic response to load variations.

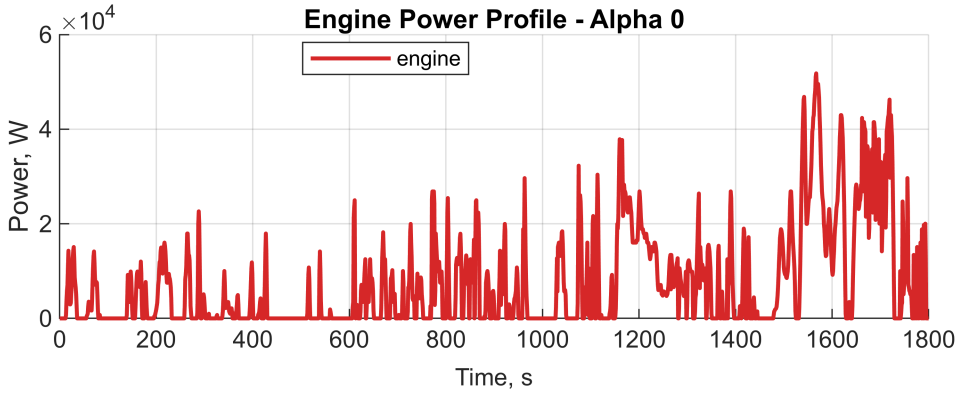
```
% Engine Power Profiles

% alpha = 1
powerProfiles(profiles, 'eng');
title("Engine Power Profile - Alpha 1")
```



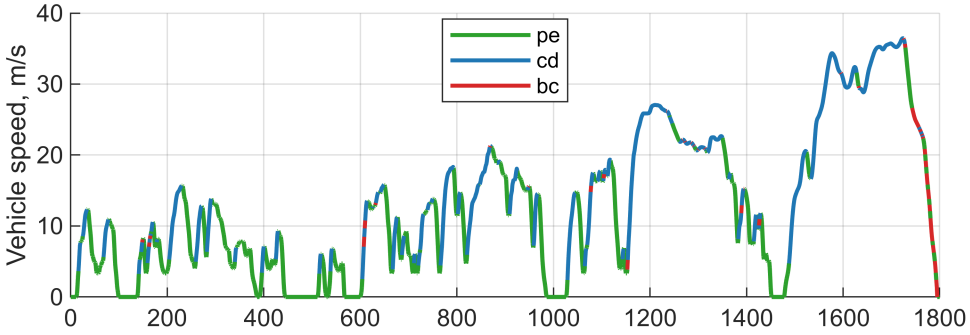


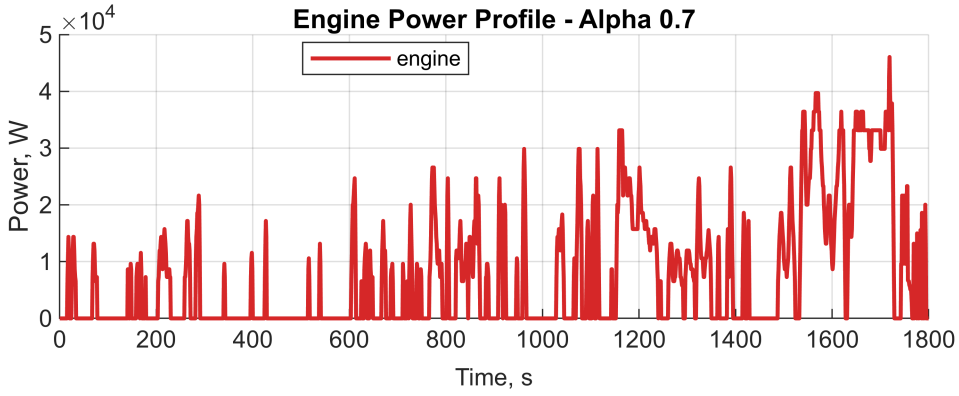

```
% alpha = 0
powerProfiles(profiles_alpha_0, 'eng');
title("Engine Power Profile - Alpha 0")
```



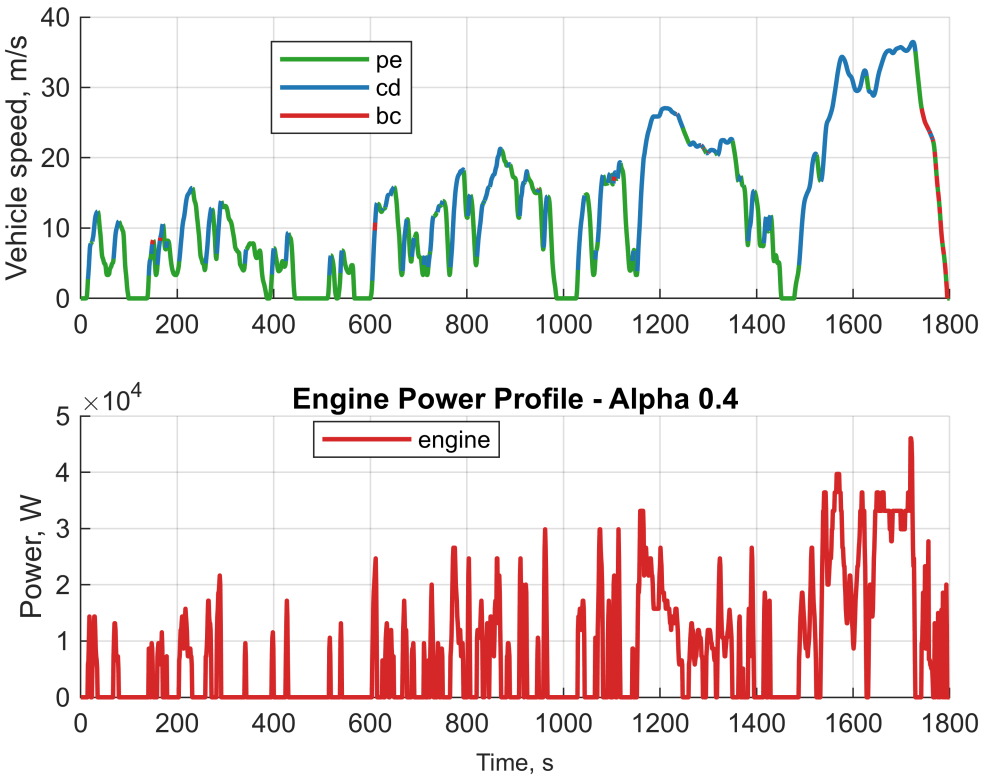



```
% alpha = 0.7
powerProfiles(profiles_alpha_0_7, 'eng');
title("Engine Power Profile - Alpha 0.7")
```






```
% alpha = 0.4
powerProfiles(profiles_alpha_0_4, 'eng');
title("Engine Power Profile - Alpha 0.4")
```



Fuel Economy vs Distance as a function of α

The presented graph illustrates the relationship between the two variables resulting from the minimization of the cost function:

the x-axis represents the fuel economy (expressed in l/100 km), while the y-axis shows the distance that can be traveled before reaching the battery's end of life (EOL), multiplied by a factor of 10^4 to improve graphical readability.

Both quantities are analyzed as functions of the parameter α, introduced in the objective function as a weighting coefficient to balance the trade-off between fuel consumption and long-term battery usage.

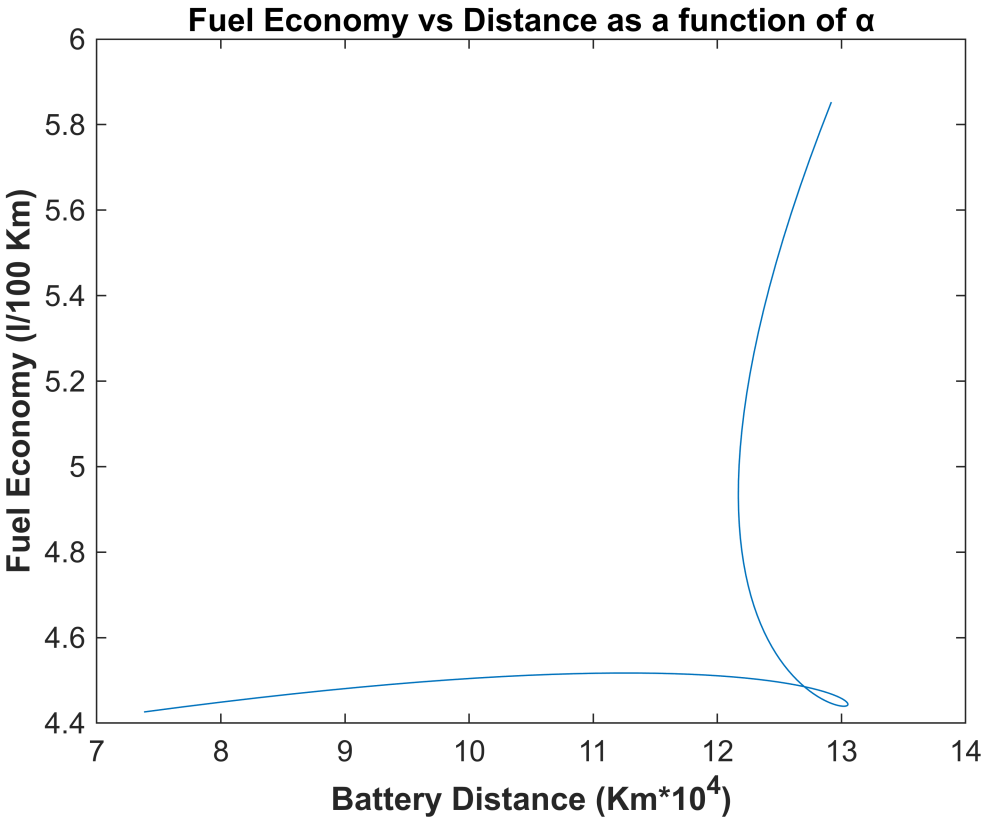
The analysis reveals that starting from α = 1, when the cost function is strongly focused on minimizing fuel consumption, fuel economy remains almost unchanged as α decreases. However, even modest changes in this region lead to a significant increase in the distance achievable before the battery reaches its end of life. This suggests that a small compromise in fuel efficiency can result in a substantial improvement in battery longevity, which is a desirable feature in energy management strategies.

As α continues to decrease, after reaching values smaller than 0.4, a marked deterioration in fuel economy is observed, with a substantial increase in fuel consumption. In this case, however, the battery life does not improve proportionally. This implies that excessively penalizing fuel consumption does not yield a meaningful gain in system sustainability, making such a choice inefficient from an cost standpoint.

In summary, the results indicate that adopting a value of α lower thant 0.5 and less leads to a slight increase in fuel consumption while significantly extending battery life. This trade-off proves beneficial, making α ≈ 0.4 an optimal choice for effectively balancing energy efficiency and battery durability.

Although the data presented are based on polynomial approximations of numerically simulated values, the observed trends provide useful qualitative insights into the interaction between the terms of the cost function and the effect of the weighting parameter α.

```
figure
plot(y_fit_batt, y_fit_fuel)
title("Fuel Economy vs Distance as a function of α", 'FontWeight', 'bold')
xlabel("Battery Distance (Km*10^4)", 'FontWeight','bold')
ylabel("Fuel Economy (l/100 Km)", 'FontWeight', 'bold')
```



Save results

As you did for the base model, save the main results in a mat-file; this time named `results_ageing.mat`. You will save vectors of results for varying values of `alpha`.

```
% Store results
save("results_ageing.mat", "fuelConsumption3", "fuelEconomy3", "finalSOC3",
"distance_eol3", "alpha_trade_off")
```

Aeging aware EMS for `hev_model`

This improved version of the hev_model (called hev_model_alpha) introduces a more advanced modulation within the objective function compared to previous implementations. As previously described, this modulation is achieved through the inclusion of an additional stage cost in the cost function, that is the absolute value of the battery current magnitude, expressed as *abs(battCurr)*. This new term allows the controller to account not only for the instantaneous fuel consumption but also for the impact of battery usage in terms of battery aging.

The balance between these two aspects is governed by the parameter α, which acts as a weighting factor between the two main objectives: minimizing fuel consumption and limiting the intensity of the current exchanged by the battery. In this way, the control strategy becomes more flexible and adaptable to different operational goals, depending on the imposed energy management priorities.

```
function [x_next, stageCost, unfeas, prof] = hev_cell_model_alpha(x, u, w,
veh,alpha)
%hev_cell_model series HEV powertrain model, vectorized inputs
%   Implements a backward quasi-static model for a series HEV,
%   advancing the simulation by one timestep. This alternative
%   implementation allows vectorized function calls by passing non-scalar
%   inputs in cell arrays.
%
% Input arguments
% ---------------
% x : cell
%   current value of the state variable(s)
% u : cell
%   current value of the control variable(s)
% w : cell
%   current value of the exogenous input(s)
% veh : struct
%   structure containing all parameters for the powertrain components
%
% Outputs
% ---------------
% x_next : cell
%   value of the state variable(s) at the end of the current timestep
% stageCost : double
%   stage (running) cost incurred
% unfeas : logical
%   when set to true, at least one of the constraints was violated
% prof : struct
%   data structure to return additional quantities of interest for
%   visualization/postprocessing
%
% Model details
% ---------------
% State variables
%   x{1}    Battery SOC, -
% Control variables
%   u{1}    Engine speed, rad/s
%   u{2}    Engine torque, Nm
```

```
% Exogenous inputs
%   w{1}    Vehicle speed, m/s
%   w{2}    Vehicle acceleration, m/s^2
% Stage cost
%   stageCost   fuel flow rate, g/s
%
% Usage examples
% ---------------
% Basic usage
%   If SOC, engSpd, engTrq, vehSpd, vehAcc are all scalars, you can find the
%   SOC at the next timestep, fuel consumption, unfeasibilty flag as:
%       [SOC_next, fuelFlwRate, unfeas] = hev_cell_model(SOC, {engSpd, engTrq},
{vehSpd, vehAcc}, veh)
%   SOC_next, fuelFlwRate and unfeas will be scalars.
%
% Returning additional time profiles
%   Also return the fourth to last outputs:
%       [SOC_next, fuelFlwRate, unfeas, prof] = hev_cell_model( ... )
%   prof will be structures with additional time profiles, (motor power,
%   battery current, ...).

%% Driveline (Vehicle + Final Drive + Transmission)
[shaftSpd, demTrq, prof] = hev_drivetrain(w{1}, w{2}, veh);

%% Motor
% Torque-coupling device (ideal)
motSpd = shaftSpd;
motTrq = demTrq; % Nm

% Electric motor efficiency
motSpd = motSpd.*ones(size(motSpd));
motEff = (shaftSpd~=0) .* veh.mot.effMap(motSpd, motTrq) + (shaftSpd==0);

% Limit Torque
motMaxTrq = veh.mot.maxTrq(motSpd); % Nm
motMinTrq = veh.mot.minTrq(motSpd); % Nm

% Saturate regen braking torque
motTrq = max(motTrq, motMinTrq); % Nm

% Calculate electric power consumption
motElPwr = (motTrq<0) .* motSpd.*motTrq.*motEff + (motTrq>=0) .* motSpd.*motTrq./
motEff; % W

% Constraints
motSpdUnfeas = motSpd > veh.mot.maxSpd;
motTrqUnfeas = ( motTrq < motMinTrq ) | ( motTrq > motMaxTrq );
motUnfeas = ( motSpdUnfeas | motTrqUnfeas );

%% Engine
```

```
engSpd = u{1}; % rad/s
engTrq = u{2}; % Nm

% Engine state
% Assume the engine is turned off when the engine torque is null
engState = engSpd > veh.eng.idleSpd & engTrq > 0;

% Expand engSpd and engTrq to the same grid
engSpd = engSpd.*ones(size(engTrq));
engTrq = engTrq.*ones(size(engSpd));

% Fuel mass flow rate
fuelFlwRate = veh.eng.fuelMap(engSpd, engTrq); % g/s
fuelFlwRate( engSpd == 0 & engTrq == 0 ) = 0;

% Maximum engine torque
engMaxTrq = veh.eng.maxTrq(engSpd); % Nm

% Constraints
engSpdUnfeas = ( engState == 1 ) & ( ( engSpd < veh.eng.idleSpd ) | ( engSpd >
veh.eng.maxSpd ) );
engTrqUnfeas = ( engTrq > engMaxTrq ) | ( engTrq < 0 );
engUnfeas =  ( engSpdUnfeas | engTrqUnfeas );

%% Gen
% Torque-coupling device (ideal)
genSpd = engSpd .* veh.gen.tcSpdRatio;
genTrq = - engTrq ./ veh.gen.tcSpdRatio; % Nm

% Generator efficiency
genSpd = genSpd.*ones(size(genTrq));
genEff = (genSpd~=0) .* veh.gen.effMap(genSpd, genTrq) + (genSpd==0);

% Calculate electric power consumption
genElPwr = genSpd.*genTrq.*genEff; % W

% Limit Torque
genMaxTrq = veh.gen.maxTrq(genSpd); % Nm
genMinTrq = veh.gen.minTrq(genSpd); % Nm

% Constraints
genSpdUnfeas = genSpd > veh.gen.maxSpd;
genTrqUnfeas = ( genTrq < genMinTrq ) | ( genTrq > genMaxTrq );
genUnfeas = ( genSpdUnfeas | genTrqUnfeas );

%% Power split
auxPwr = 0;
battPwr = motElPwr + genElPwr + auxPwr;

%% Battery
```

```
% Battery internal resistance
battR = veh.batt.eqRes(x{1}); % ohm
% Battery voltage
battOCVolt = veh.batt.ocv(x{1}); % V

% Battery current
battCurr = (battOCVolt-sqrt(battOCVolt.^2 - 4.*battR.*battPwr))./(2.*battR); % A
battCurr = real(battCurr);
% Current limits
maxChrgBattCurr = veh.batt.minCurr; % A
maxDisBattCurr = veh.batt.maxCurr; % A
% Saturate charge current in regen braking
battCurr = max(battCurr, maxChrgBattCurr);

% New battery state of charge
x_next{1}  = - battCurr ./ (veh.batt.nomCap * 3600) .* veh.dt + x{1};
% Constraints
battUnfeas = ( battPwr <= 0 & battCurr < maxChrgBattCurr ) | ( battPwr > 0 &
battCurr > maxDisBattCurr );

%% Stage cost

maxFFR = maxFuelFlwRate(veh.eng);

% HERE THE STAGE COST IS UPDATED -----> ITS VALUE IS WEIGHTED BETWEEEN FUEL
% FLOW RARE AND CURRENT THROUGH ALPHA
stageCost  = alpha*(fuelFlwRate/maxFFR) + (1-alpha)* (abs(battCurr)/
veh.batt.maxCurr);

%% Unfeasibilites
% Combine unfeasibilities
unfeas = ( motUnfeas | engUnfeas | genUnfeas | battUnfeas );

%% Operating mode
opMode = repmat("", size(battCurr));
opMode( engState == 0 ) = 'pe';
opMode( engState == 1 & battCurr >= 0 ) = 'cd';
opMode( engState == 1 & battCurr < 0 ) = 'bc';

%% Pack additional outputs
prof.fuelFlwRate = fuelFlwRate;
prof.engSpd = engSpd;
prof.engTrq = engTrq;
prof.engSpdUnfeas = engSpdUnfeas;
prof.engTrqUnfeas = engTrqUnfeas;
prof.engineState = engState;

prof.genSpd = genSpd;
prof.genTrq = genTrq;
prof.genElPwr = genElPwr;
```

```
prof.genSpdUnfeas = genSpdUnfeas;
prof.genTrqUnfeas = genTrqUnfeas;

prof.motSpd = motSpd;
prof.motTrq = motTrq;
prof.motElPwr = motElPwr;
prof.motSpdUnfeas = motSpdUnfeas;
prof.motTrqUnfeas = motTrqUnfeas;

prof.battSOC = x{1};
prof.battCurr = battCurr;
prof.battVolt = battOCVolt - battR.*battCurr;
prof.battRes = battR;

prof.vehSpd = w{1};
prof.vehAcc = w{2};
prof.opMode = opMode;

prof.motUnfeas = motUnfeas;
prof.engUnfeas = engUnfeas;
prof.genUnfeas = genUnfeas;
prof.battUnfeas = battUnfeas;
prof.unfeas = unfeas;
end
```