Assignment #1: Quasi-static HEV model and Rule Based Control

Table of Contents

Project introduction

The aim of this project is to design the torque-split control strategy of a p2 parallel HEV (represented in Fig.1).

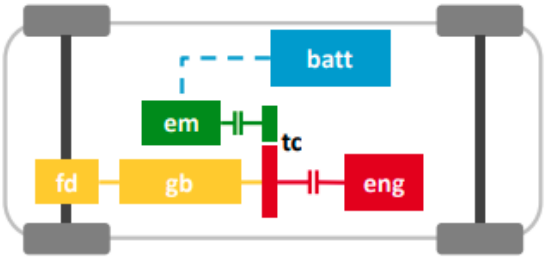


Fig.1: Scheme of a P2 parallel HEV

Moreover, the simulation model evaluates the fuel consumption while tracking the battery state of charge (SOC) for a given driving mission. The inputs of the model are the vehicle and powertrain parameters, as well as the data of the driving cycle. Based on these informations, this script provides at any instant the value of the control parameter α_{eng}, called engine torque-split factor, defined as follows:

$$\alpha_{eng} = \frac{T_{eng}}{T_{dem}}$$

Where T_{eng} is the torque provided by the internal combustion engine (ICE) and T_{dem} is the total torque requested to move the vehicle.

To define the control strategy, a backward (quasi-static) approach is adopted: it is based on the assumption that the driving cycle represents the exact speed profile maintained by the vehicle, calculating in this way the torque required at the wheels using a longitudinal vehicle model. With the knowledge of the engaged gear, which is also calculated through a simple logic inside the model, the torque request at the shaft is obtained. This value is used in the rule based control to determine which operating mode is actuated, among 4 different possibilities: pure electric, pure thermal, power split and battery charging. Combining the calculated values of the torque demand and α_{eng} in each instant, the torque that has to be provided by both electric motor (EM) and

ICE is obtained. Consequently, through motor and engine maps, both energy usage and fuel consumption are evaluated.

Fig.2: Flowchart of a backward simulation approach

In the project, two different approaches in the torque-split factor control logic are considered: the basic one uses a constant target SOC, while the second develops an extra feature (#2) which shows a speed dependent target SOC. The results provided by the two designed models are presented and compared through the analysis of the most significant performance indicators.

Group information

Group number: 49

Students:

- Riccardo Bressani, s323665
- Carlo Vittorio Colucci, s329703
- Luca Marchetto, s323437

Load the cycle and vehicle data

The reference cycle which is used in this project is loaded in the following section: it consists in a vector representing a speed profile with respect to time. The data concerning the cycle are contained in the file 'WLTP3.mat'.

Starting from the speed profile, the acceleration is computed performing a discrete derivative operation: each component of the vector is obtained dividing the difference between two consecutive speed values by the time step. Doing so, the obtained vector of acceleration has a length (N-1), where N is the number of time instants; a first component equal to 0 is added to the acceleration vector in order to work with vectors of the same length. This assumption is coherent with the cycle since the first terms of the speed vector are constant and equal to 0.

```
clear all
clc
% Addition of folders
addpath('data')
addpath('models')
addpath('utilities')
% Extraction of cycle velocities and accelerations
mission = load('data\WLTP3.mat');
vehSpd = mission.speed_kmh./3.6; % [m/s]
time = mission.time_s; % [s]
dt = time(2) - time(1); % [s]
vehAcc = (vehSpd(2:end)-vehSpd(1:end-1))./dt; % [m/s^2]
% First component assumed 0 added to the acceleration vector
vehAcc = [0;vehAcc]; % [m/s^2]
```

```
% The speed and acceleration profiles are plotted against time
t = tiledlayout(2,1);
nexttile(1)
plot(time,vehSpd,'LineWidth',1)
grid minor
title('Vehicle speed')
xlabel('Time [s]')
ylabel('Vehicle speed [m/s]')
nexttile(2)
plot(time,vehAcc,'LineWidth',1)
grid minor
title('Vehicle acceleration')
xlabel('Time [s]')
ylabel('Vehicle acceleration [m/s^2]')
```

With the following commands, the data related to the vehicle are loaded from the file 'vehData.mat'. These data are subsequently rescaled using the function 'scaleVehData.m' considering as additional inputs the values of ICE power [W], EM power [W], battery capacity [Ah] associated to the group number:

- **Group number:** 49
- **ICE power:** 60000 W
- **EM power:** 18000 W
- **Battery capacity:** 6.4 Ah

```
% Data loading and scaling
veh = load('data\vehData.mat');
```

```
engPwr = 60000; % [W]
emPwr = 18000; % [W]
battCap = 6.4; % [Ah]
veh = scaleVehData(veh,engPwr,emPwr,battCap); % scaleVehData(veh, engPwr[W],
emPwr[W], battCap[Ah])
```

Simulation loop

The first step consists in computing the vector of the gear number in all the cycle instants; this is performed through a simple algoritm implemented in the function 'gearControl.m' which compares at every iteration the actual speed with two reference values for the engaged gear, an upshift and a downshift one, keeping the same gear if the speed is inside the interval or performing a shift if it exceeds one of the limits. The reference values for downshift and upshift speed for each gear are contained in two separate vectors inside the file 'trasmControlData.mat'. The product between actual engaged gear and final drive transmission ratios gives the total transmission ratio.

Given the values of the vehicle speed and acceleration as well, at first the resistant force is computed: it takes into account the contributions of road grade (not present in our case), rolling resistance and aerodynamic drag. This force is represented by a second order polinomial function with the vehicle speed as independent variable, whose coefficients f_0, f_1, f_2 are contained in file 'vehData.mat'. Summing the resistance force with the one to accelerate the vehicle, the tractive force is obtained.

Multiplying the tractive force by the wheel radius, the wheel torque is found: the ratio between this and the total transmission ratio gives the shaft torque. The shaft speed is simply given by the product between the wheel speed and the total transmission ratio. All these quantities are calculated in function 'hev_drivetrain.m'. For what concerns the EM, the transmission ratio between this component and the shaft is fixed and the value is stored into the variable taoEM.

```
% Extraction of transmission data
load('transmControlData.mat')
% Initial conditions: SOC = 60% and 1st gear engaged
SOC(1) = 0.6; % [p.u.]
GN0 = 1; % [-]
% EM coupling transmission ratio
taoEM = veh.em.tcSpdRatio;  % [-]
```

As anticipated, two different strategies are developed for the choice of the torque-split factor. In the following lines it is possible to choose between the basic one and the other one, containing extra feature #2. The different choice performed runs the code using different functions and logics.

Basic control strategy:

The choice of the torque-split factor is performed according to the following flow chart:

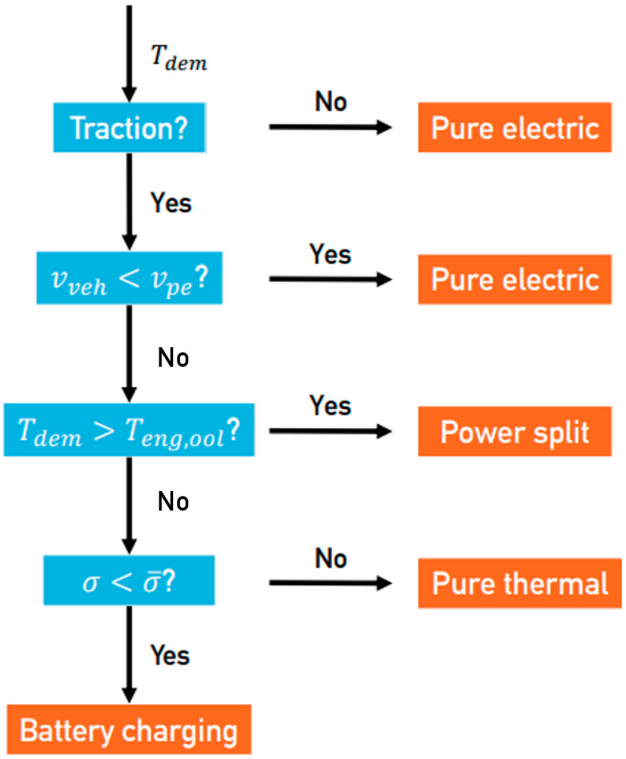


Fig. 3: flowchart of basic control strategy

In this first logic, the value $\bar{\sigma}$ is a constant parameter. It is chosen with a double objective: maintaining the SOC of the battery always close to the initial value (assumed 60%) throughout the entire cycle (charge sustaining operation), trying at the same time to reduce the fuel consumption. By a trial and error procedure, the value chosen for this purpose is 0.57 (stored in variable 'SOCsustain'), corresponding to 57% of total capacity: the final SOC obtained with this value is 0.5865, equivalent to 58.65%; it is observed that a further reduction of 'SOCsustain' value leads to a reduction in fuel consumption, yet causes an eccesive discharge of the battery.

The other parameter to be tuned in the algoritm is the maximum speed at which pure electric drive is allowed: also in this first logic, a simple trial and error procedure is adopted. Starting from a very low value and progressively increasing it by a step of 1 m/s, it is observed that at 10 m/s the first errors arise, due to the fact that the torque request exceeds the maximum torque providable by the EM. For this reason the variable 'pureEVspd' is set equal to 9 m/s.

However, It has to be highlighted that this is not a safe criterion for the choice of this value, since designed on the particular cycle acceleration profile. A different mission could require unfeasible torque levels even at speeds below the selected 'pureEVspd' threshold. A different strategy to overcome this limit is proposed in the 'extra feature' section.

The function 'powerFlowControl.m' takes as input, at any instant, the shaft speed and torque, the vehicle speed, the SOC, as well as the previously defined parameters 'pureEVspd', 'taoEM', 'SOCsustain' and vehicle parameters from the data structure 'vehData.mat'. As a result it computes the instantaneous value of the torque-split factor, following the logic that is explained in detail in the 'Functions implementation' section.

This obtained factor, combined with the instantaneous SOC, engaged gear, driving conditions and vehicle parameters, is provided to function 'hev_model.m'. At first, the shaft speed and torque request are computed

again through 'hev_drivetrain.m'. Then, through the knowledge of the torque-split factor, the ICE instantaneous torque is retrieved and, by means of the fuel consumption map, the fuel flow rate as well.

In the same way, the EM instantaneous torque is calculated and consequently the current requested to the battery can be computed. Then the battery charge variation is evaluated and a new SOC level is retrieved.

In addition, the function is able to detect critical situations, arising unfeasibility flags corresponding to different problems:

- ICE / EM torque or speed exceeds map boundaries;
- battery current exceeds the imposed limits;
- uncoherent EM request: positive EM torque while braking.

As a result, all these data are returned by the function and stored within vectors and structures.

Extra feature (#2) control strategy:

The control strategy developing extra feature #2 implements an adaptive SOC target in which the value $\bar{\sigma}$ is not anymore a constant parameter, but a function of speed, with the aim of favouring battery usage in urban segments and battery charging in highway ones:

$$\bar{\sigma} = \sigma_0 + \frac{1}{2} \cdot m_{veh} \cdot v_{veh}^2 \cdot \frac{f_{cs}}{E_b}$$

where $\frac{1}{2} \cdot m_{veh} \cdot v_{veh}^2$ is the kinetic energy of the vehicle (neglecting the rotating elements inertia), E_b is the battery energy in J, σ_0 and f_{cs} are constant parameters tuned in the 'Functions implementation' section.

Furthermore, some additional checks, to improve the algoritm robustness, are introduced, shown in **Fig.4**. In particular:

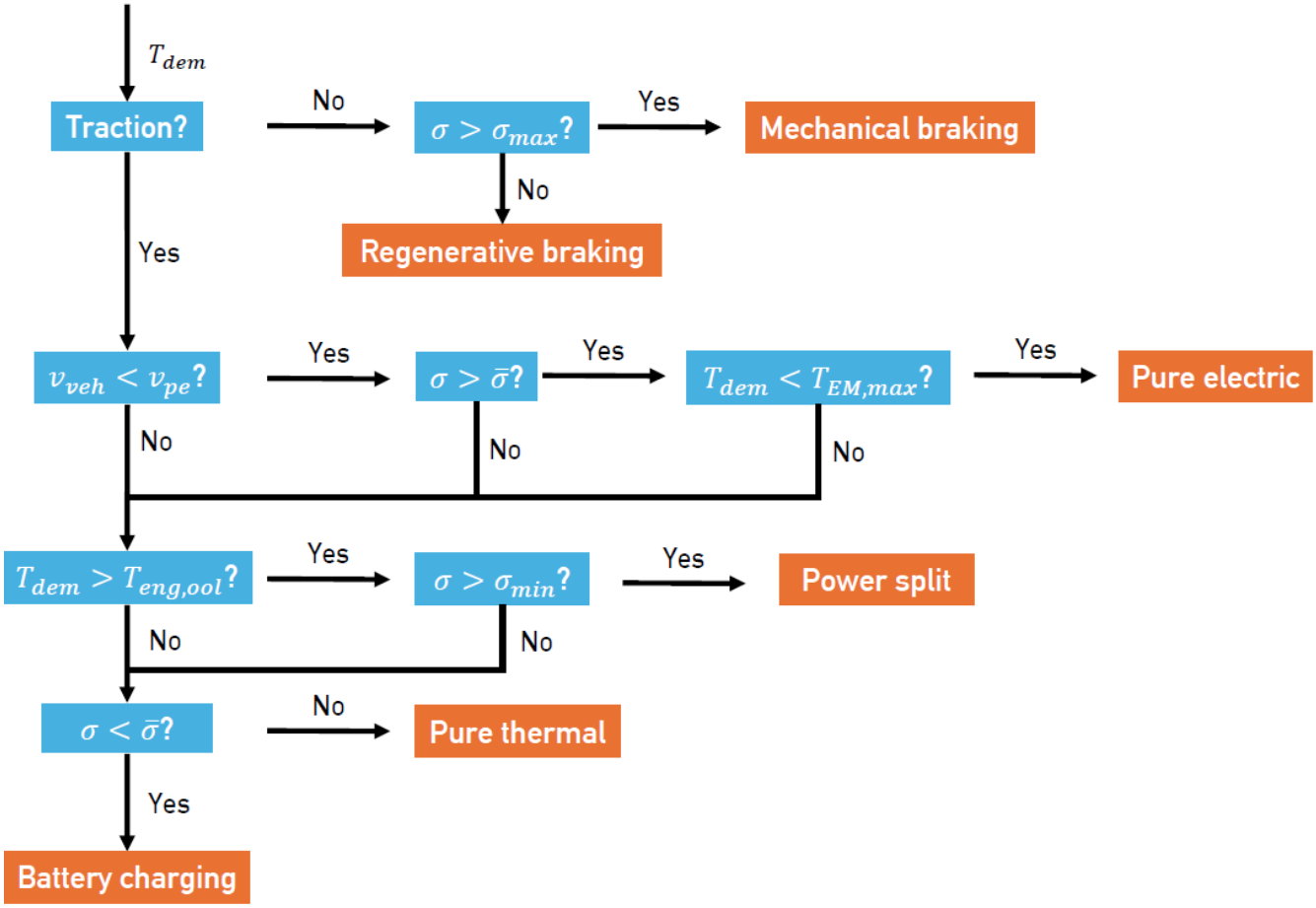


Fig. 4: flowchart of extra feature #2 control strategy

- Two additional conditions to allow pure electric mode: the first verifies that the torque request is less than the maximum torque that the EM can provide, while the second imposes that the SOC must be higher than the target level $\bar{\sigma}$; these verifications have some interesting implications. First of all they allow to make the system work in a wider range of operating conditions, independently on the driving mission. Secondly, they offer the opportunity to set a higher value of 'pureEVspd', preventing the ICE from working in pure thermal mode in a low torque (and low efficiency) region in all situations in which, despite the high speed, the load request is very low. The potential benefit is quite evident in the comparison between the operating points in the ICE map between the two strategies, as well as in the cycle fuel cosumption. In the code 'pureEVspd' is set to an empiric value of 30 m/s, to show the effect of the variation, even if in principle it would be also possible to remove the pure electric speed limit at all. This would radically change the logic of the control strategy, leaving the choice only to a torque and SOC verification;
- A minimum SOC value to allow power split: with the aim of preserving battery health, a minimum SOC condition is introduced also to use power split operation, to avoid excessive discharge due to continuous utilization of this mode; the chosen limit is 0.4 (40% of the nominal battery capacity);
- A maximum SOC value to allow regenerative braking: for safety reasons, its goal is to stop regeneration and switch to full mechanical braking in case of prolonged downhill driving and battery SOC already high. Since no significant value of α_{eng} would correctly represent this situation for the consequent calculations, this check is not implemented in 'powerFlowControlExtraFeature.m' function, but a new

version of 'hev_model.m' is proposed, called 'hev_model_updated.m'. In this function a new check is inserted from line 137 on, switching EM torque to 0 and keeping SOC constant if this situation occurs. In addition, a new unfeasibility flag is arised, called 'overChargeUnfeas', which should be sent to the vehicle control unit to switch to full mechanical braking.

For the rest of the code, the logic is the same of the 'basic' control startegy.

```
% The calculation of the quantities previously introduced is performed iteratively
in
% a cycle for each time instant
powerSplitStrategy = "basic"; % Choice of the control strategy
for n = 1:length(time)
    GN(n) = gearControl(vehSpd(n),GN0,upSpd,downSpd); % [-]
    GN0 = GN(n);
    [shaftSpd(n), shaftTrq(n), vehDrvt(n)] = hev_drivetrain(vehSpd(n), vehAcc(n),
GN0, veh);    % [rad/s, N*m, -]
    switch powerSplitStrategy
        case "basic"
            % Parameters to be tuned
            pureEVspd = 9; % [m/s]
            SOCsustain = 0.57; % [p.u.]
            PowerSplit(n) = powerFlowControl(shaftSpd(n), shaftTrq(n), vehSpd(n),
pureEVspd, taoEM, SOC(n), SOCsustain, veh); % basic function
            [SOC(n+1), stageCost(n), unfeas(n), engPrf(n), emPrf(n), battPrf(n),
vehPrf(n)] = hev_model(SOC(n), [GN(n), PowerSplit(n)], [vehSpd(n), vehAcc(n)],
veh);
        case "extra feature"
            pureEVspd = 30;
            PowerSplit(n) = powerFlowControlExtraFeature(shaftSpd(n), shaftTrq(n),
vehSpd(n), pureEVspd, taoEM, SOC(n), veh); % function with extra feature
            [SOC(n+1), stageCost(n), unfeas(n), engPrf(n), emPrf(n), battPrf(n),
vehPrf(n)] = hev_model_updated(SOC(n), [GN(n), PowerSplit(n)], [vehSpd(n),
vehAcc(n)], veh); % function with additional check
    end
end
```

Being the previous calculations performed in a cycle, multidimensional structures are created. In order to improve the readability and ease of access to data, these are transformed in 1x1 structs containing the data in vectors, with the use of function 'structArray2struct.m'; then, all these are grouped into a single structure called 'prof':

```
% Transform the non-scalar struct containing time profiles into scalar
% structs; this makes their manipulation easier.
engPrf = structArray2struct(engPrf);
emPrf = structArray2struct(emPrf);
battPrf = structArray2struct(battPrf);
vehPrf = structArray2struct(vehPrf);
% Pack profiles into a single structure
```

```
prof.engPrf = engPrf;
prof.emPrf = emPrf;
prof.battPrf = battPrf;
prof.vehPrf = vehPrf;
```

From the results of the cycle it's noteworthy to compute some fundamental outcomes for the evaluation of the controller performance, such as the fuel consumption in [L/100km] and the final SOC value.

```
fuelConsumption = trapz(time, engPrf.fuelFlwRate)/1000;  % Cumulative fuel
consumption [kg]
vehDist = trapz(time, vehSpd);  % Total cycle distance [m]
fuelEconomy=(fuelConsumption*10^5)/(veh.eng.fuelDensity*vehDist)    % Fuel
consumption [L/100km]
```

```
fuelEconomy = 4.7137
```

```
finalSOC = SOC(end) % [p.u.]
```

```
finalSOC = 0.5865
```

Inside the previous cycle, as previously explained, the presence of unfeasibility situations is detected. Instead, in the following cycle, the specific type of error is displayed combined with the correspondent cycle instant.

```
for n = 1:length(time)
    if unfeas(n)==1
        if engPrf.engSpdUnfeas(n)==1
            disp(['At iteration ' num2str(n) ' the engine speed exceeds the
limits.'])
        end
        if engPrf.engTrqUnfeas(n)==1
            disp(['At iteration ' num2str(n) ' the engine torque exceeds the
limits.'])
        end
        if  emPrf.emSpdUnfeas(n)==1
            disp(['At iteration ' num2str(n) ' the electric motor speed exceeds the
limits.'])
        end
        if  emPrf.emTrqUnfeas(n)==1
            disp(['At iteration ' num2str(n) ' the electric motor torque exceeds
the limits.'])
        end
        % In the two strategies battery unfeasibilities are differenciated
        if powerSplitStrategy == 'basic'
            if vehPrf.battUnfeas(n) == 1
                disp(['At iteration ' num2str(n) ' the battery current exceeds the
limits.'])
            end
        elseif powerSplitStrategy == 'extra feature'
            if  vehPrf.overChargeUnfeas(n)==1
                disp(['At iteration ' num2str(n) ' the battery state of charge
doesn''t allow regenerative braking.'])
```

```
            end
            if  vehPrf.overCurrentUnfeas(n)==1
                disp(['At iteration ' num2str(n) ' the battery current exceeds the
limits.'])
            end
        end
        if  vehPrf.pwtUnfeas(n)==1
            disp(['At iteration ' num2str(n) ' uncoherent EM torque request.'])
        end
    end
end
```

Save results

The following code saves the cycle time profiles, the fuel consumption (in kg), the fuel economy (in L/100km), and the final SOC in a mat-file, according to the selected control strategy.

```
% Store results as a function of the control strategy considered
switch powerSplitStrategy
    case "basic"
        save("resultsBasic.mat", "prof", "fuelConsumption", "fuelEconomy",
"finalSOC")
    case "extra feature"
        save("resultsExtraFeature.mat", "prof", "fuelConsumption", "fuelEconomy",
"finalSOC")
end
```

- `prof` is the structure of profiles created in the previous section.
- `fuelConsumption` is the total fuel consumption for the whole mission (in kg).
- `fuelEconomy` is the distance-specific fuel consumption for the whole mission (in L/100km).
- `finalSOC` is the final battery SOC (per unit value).

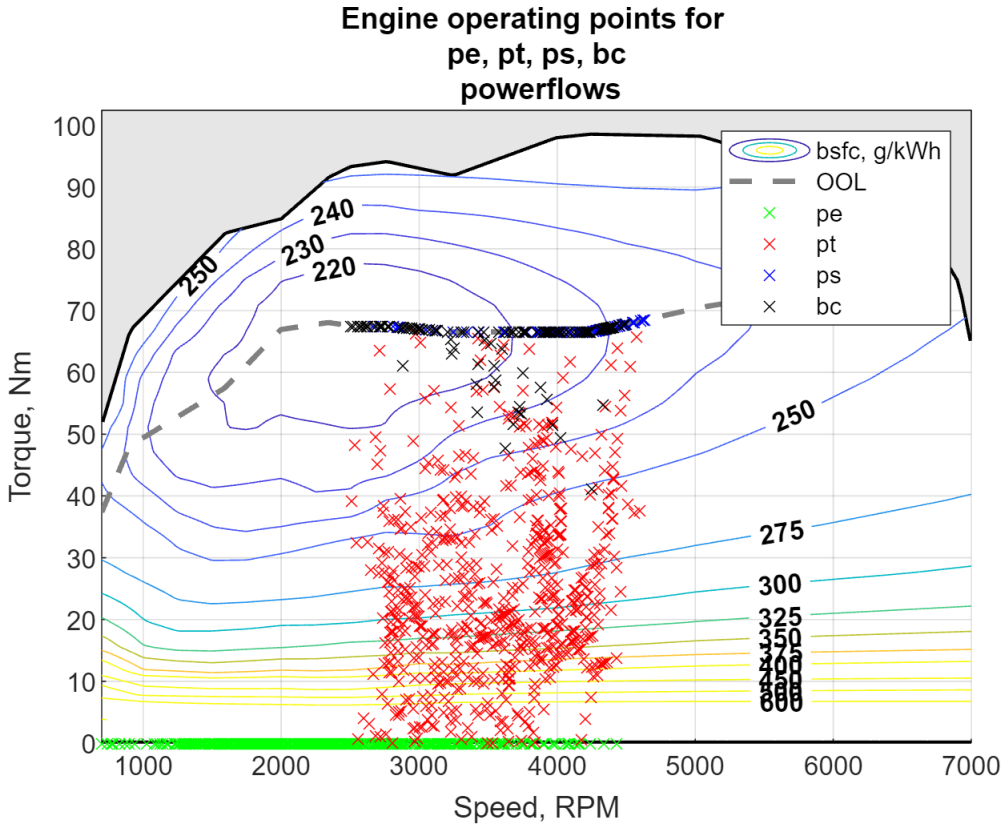
Post-processing tools

The following section proposes some useful tools for the analysis of the controller behaviour; according to the code structure, the displayed quantities are related to the specific selected controller. For this reason the comment on the controllers performance is carried out in the 'Results analysis' section, to compare them side by side.

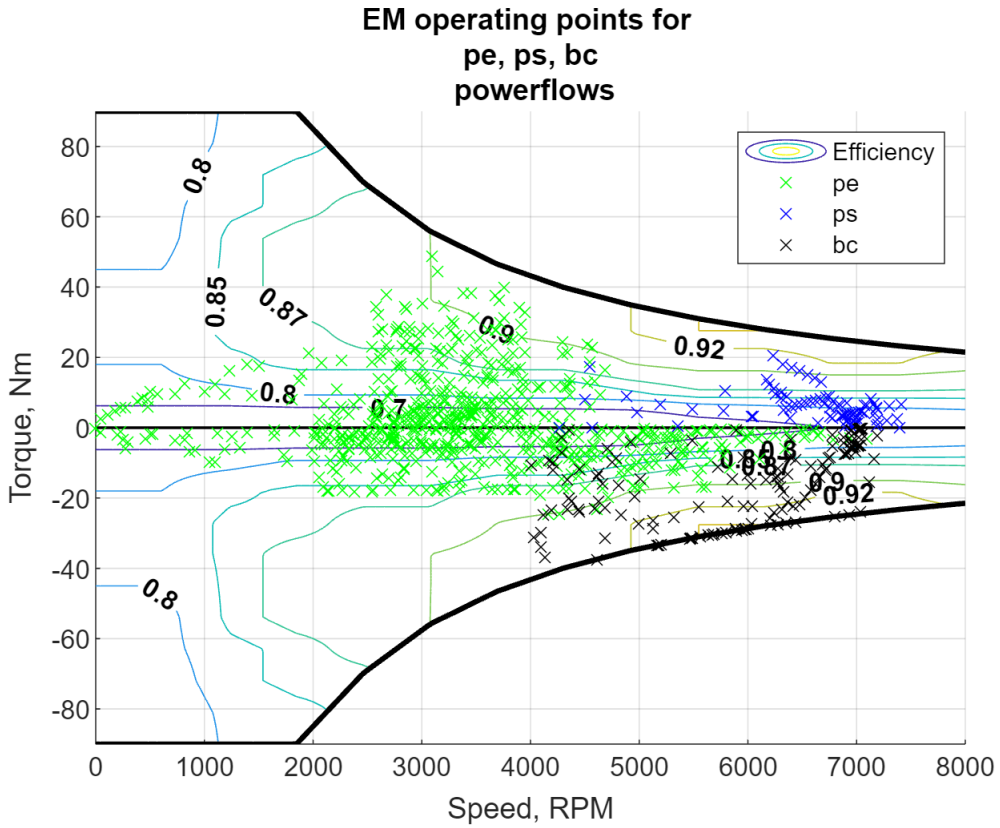
The first two graphs are the ICE and EM maps, where the cycle working points are shown, highlighting the different operating modes. In particular the ICE plot offers the possiblity to display different performance indicators: fuel consumption, brake specific fuel consumption and efficiency. The optimal operating line (ool) is plotted as well, representing the locus of points of maximum efficiency for each engine speed. This is useful to evaluate the controller performance, focusing on the distance between working points and 'ool': the smallest it is, the better the fuel energy is exploited. For what concerns the EM graph, only efficiency isolines are present.

```
% Engine operating points
contour_type="bsfc";
switch contour_type
```

```
    case 'fuel consumption'
        engMapWithPF(veh.eng, prof, "fc", 'all')
    case 'bsfc'
        engMapWithPF(veh.eng, prof, "bsfc", 'all')
    case 'efficiency'
        engMapWithPF(veh.eng, prof, "eff", 'all')
end
```



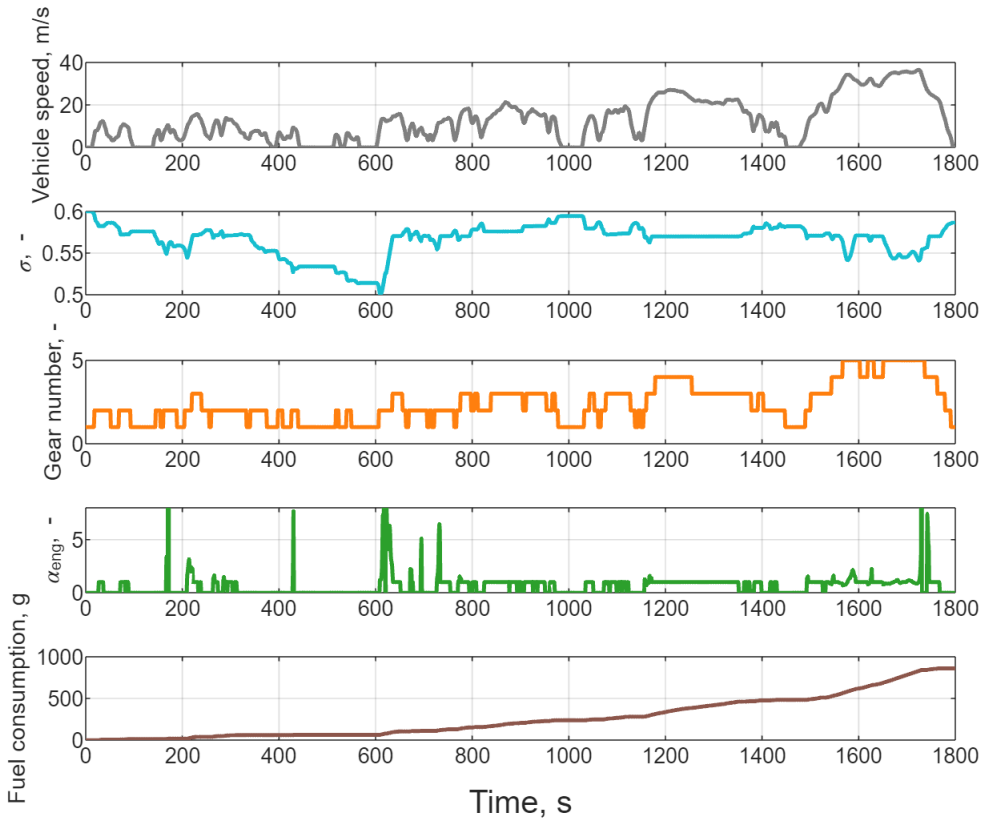

```
% EM operating points
emMapWithPF(veh.em, prof, ["pe", "ps", "bc"]);
```



To complete the analysis, five time profiles of different quantities are reported:

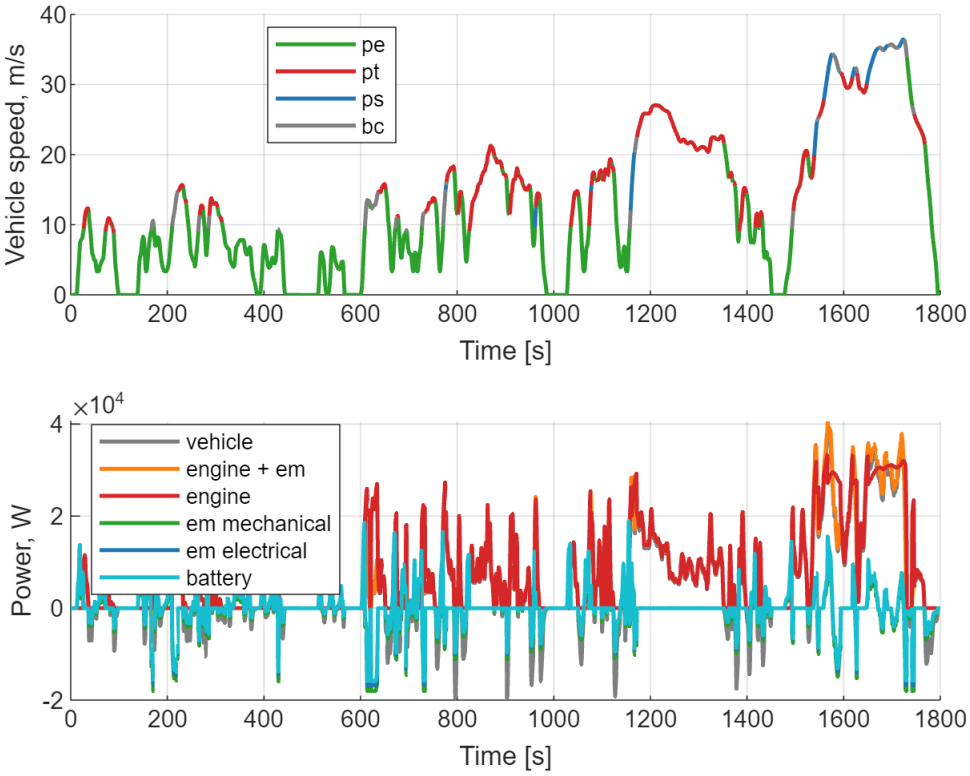
- Vehicle speed [m/s];
- battery SOC [p.u.];
- engaged gear;
- torque-split factor;
- cumulative fuel consumption [g].

```
% Main profiles
mainProfiles(prof);
```



Combining previously obtained information about working points, the speed profile is plotted as a function of time, pointing out with different colours the instantaneous operating mode. Finally, the power flows of all the components of the system are plotted against time.

```
% Power profiles
powerProfiles(prof, 'all');
```



Looking at the second graph it's visible, especially in the negative power region, how the efficiency affects the energy conversion.

Results analysis

To fully understand the obtained results, their analysis is carried out putting in comparison the two controllers:

- ICE working points on bsfc map: this chosen quantity allows to evaluate the quality of the fuel energy exploitation

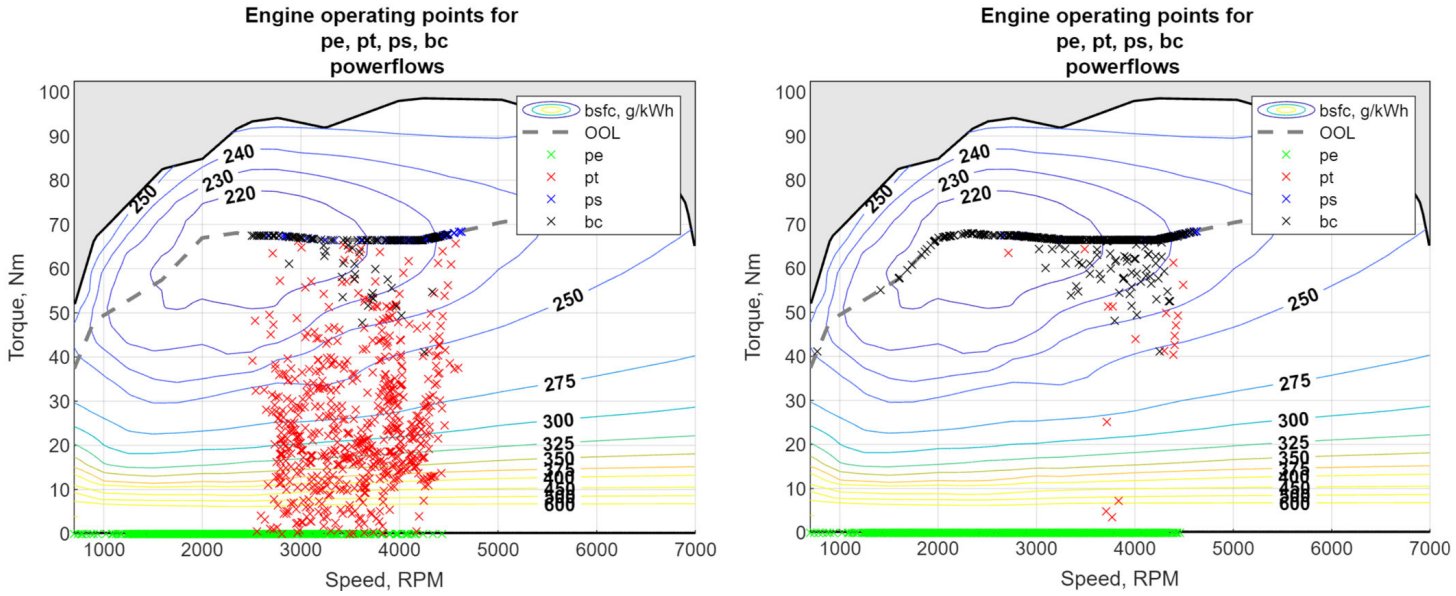


Fig.5: ICE maps with oprating points of basic (left) and extra feature (right) controller

It is evident how in the basic control strategy many operating points are in pure thermal mode and far from ool, figuring bad energy exploitation. This because even if torque demand is low, having a limited 'pureEVspd' threshold doesn't allow pure electric operation. It is possible to observe that this weak point is practically solved by the extra feature strategy with the updated logic, which reduces consistently the number of pure thermal operating points, leaving an envelope of points generally closer to ool.

- EM working points on efficiency map:

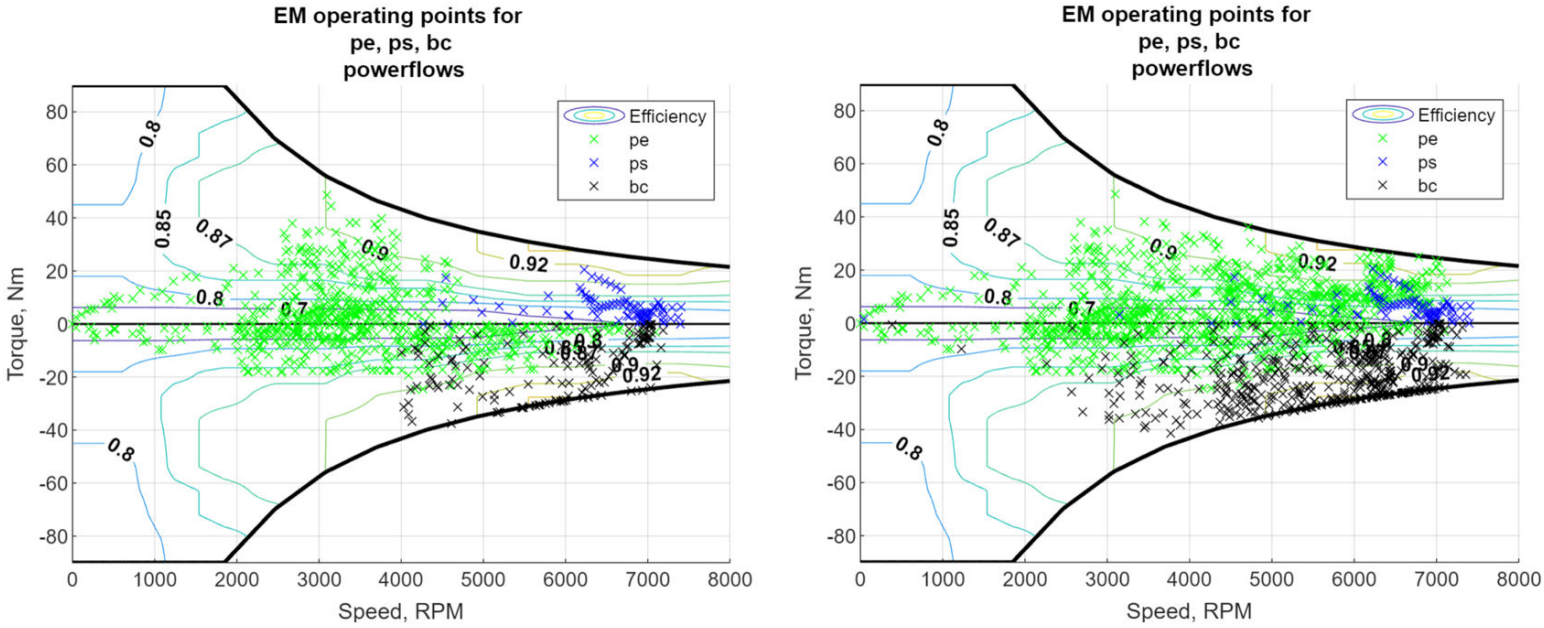


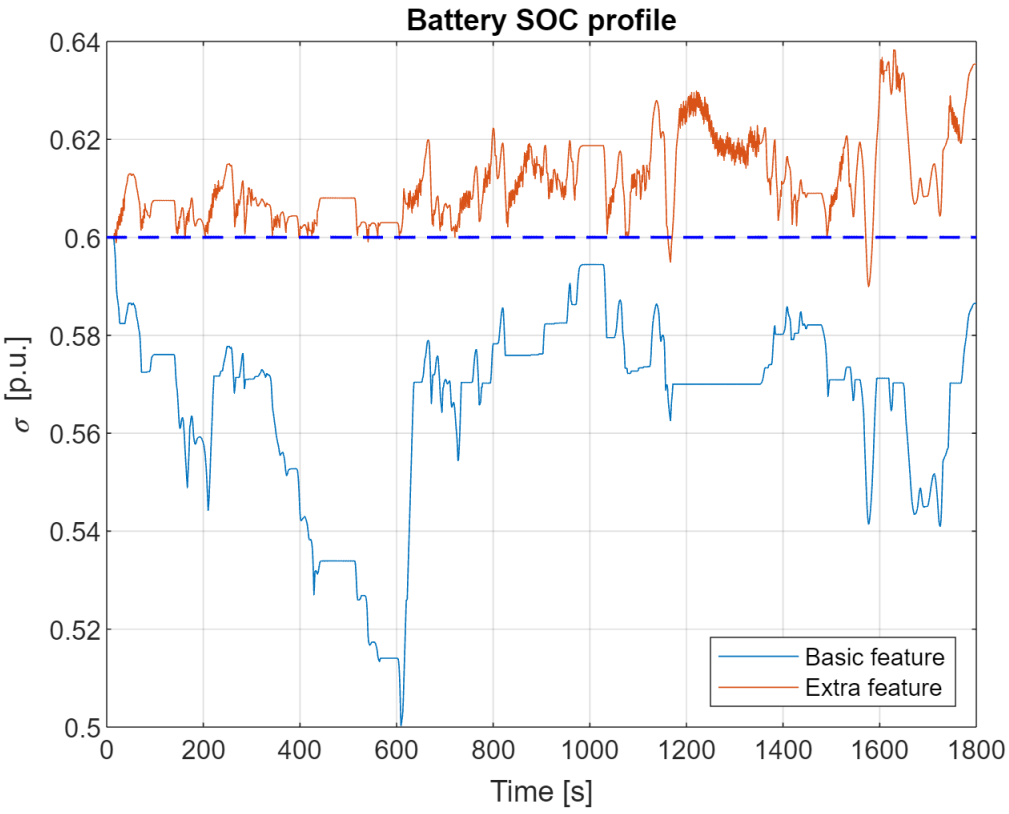
Fig.6: EM maps with oprating points of basic (left) and extra feature (right) controller

The e-machine can be exploited as a motor or as a generator: the blue points are always within positive torque region, because the EM is giving a positive contribution in power split mode. On the contrary the battery charging points (in black) correspond to negative EM torque, meaning that the power excess from ICE is used to recharge the battery. Negative pure electric operation refers, instead, to regenerative braking completely demanded to EM. In the comparison it is possible to observe how, in the right figure, the pure electric operation is present also at higher EM speed, due to the previously explained different controller logic. Moreover, the number of battery charging operation points is visually higher, trend also confirmed in the torque-split profiles.

- SOC profile comparison:

New graphs are reported, showing in the same figure the quantities of the two different strategies:

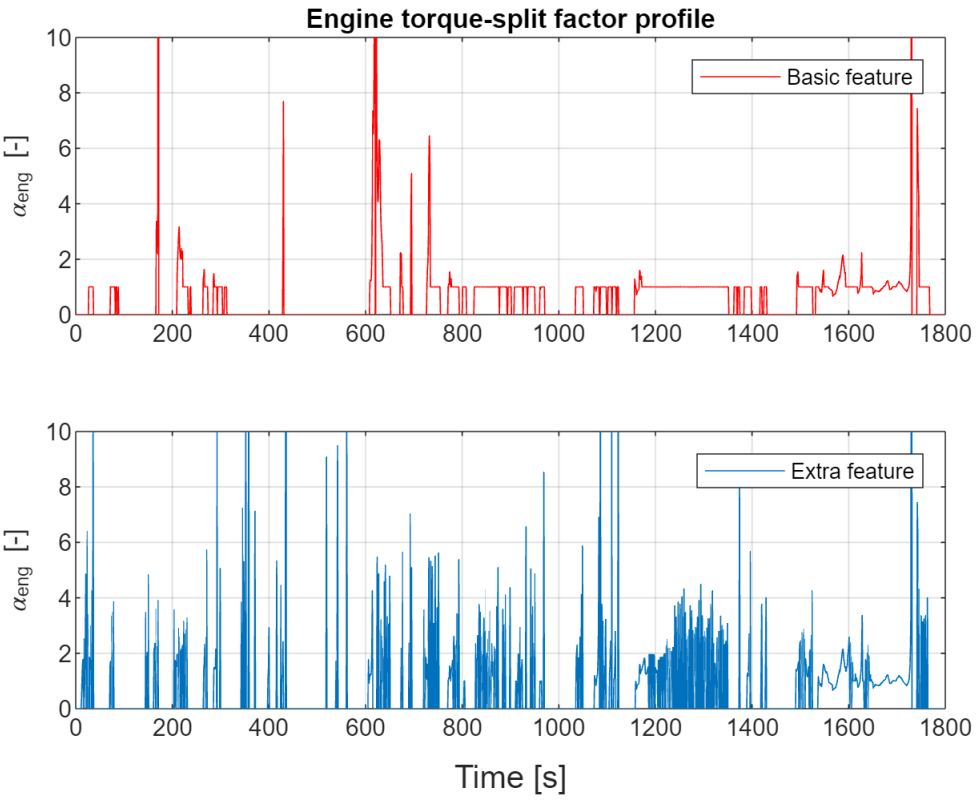
```
% results previously collected are loaded here
load("resultsBasic.mat")
% basic control strategy profiles
battSOCBasic = prof.battPrf.battSOC; % battery SOC
alphabasic = prof.vehPrf.engAlpha; % torque-split factor
fcbasic = cumtrapz(time, prof.engPrf.fuelFlwRate)/1000; % cumulative fuel
consumption
load("resultsExtraFeature.mat")
% extra feature control strategy
battSOCextra = prof.battPrf.battSOC; % battery SOC
alphaextra = prof.vehPrf.engAlpha; % torque-split factor
fcextra = cumtrapz(time, prof.engPrf.fuelFlwRate)/1000; % cumulative fuel
consumption
% SOC profiles
figure
plot(time, battSOCBasic), hold on
plot(time, battSOCextra), grid on
plot(time,0.6*ones(1,1801),LineStyle="--",Color="b",LineWidth=1.2)
xlabel("Time [s]"); ylabel("\sigma [p.u.]")
title('Battery SOC profile')
legend('Basic feature', 'Extra feature',location ='southeast')
xlim([0, 1800]);
```



It is observed how, not having any SOC constraint to allow pure electric operation, the basic control strategy shows a deeper battery discharge at the beginning of the cycle. This trend could in principle lead to a complete discharge of the battery in a low speed driving mission. Putting a check in the logic provides a more constant SOC profile, almost always above 0.6, as well as a safer system behaviour for battery health. In addition, the final SOC is higher with the updated controller version.

- Engine torque-spilt factor profile comparison:

```
% Engine torque-spilt factor profile
figure
t = tiledlayout(2,1);

ax1 = nexttile;
plot(time, alphabasic,Color="r"), grid on
ylabel("\alpha_{eng} [-]")
title('Engine torque-split factor profile')
legend('Basic feature')
xlim([0, 1800]), ylim([0,10])
ax2 = nexttile;
plot(time, alphaextra), grid on
ylabel("\alpha_{eng} [-]")
legend('Extra feature')
xlabel(t, "Time [s]", 'FontSize', 12)
linkaxes([ax1 ax2], 'x')
xlim([0, 1800]), ylim([0,10])
```

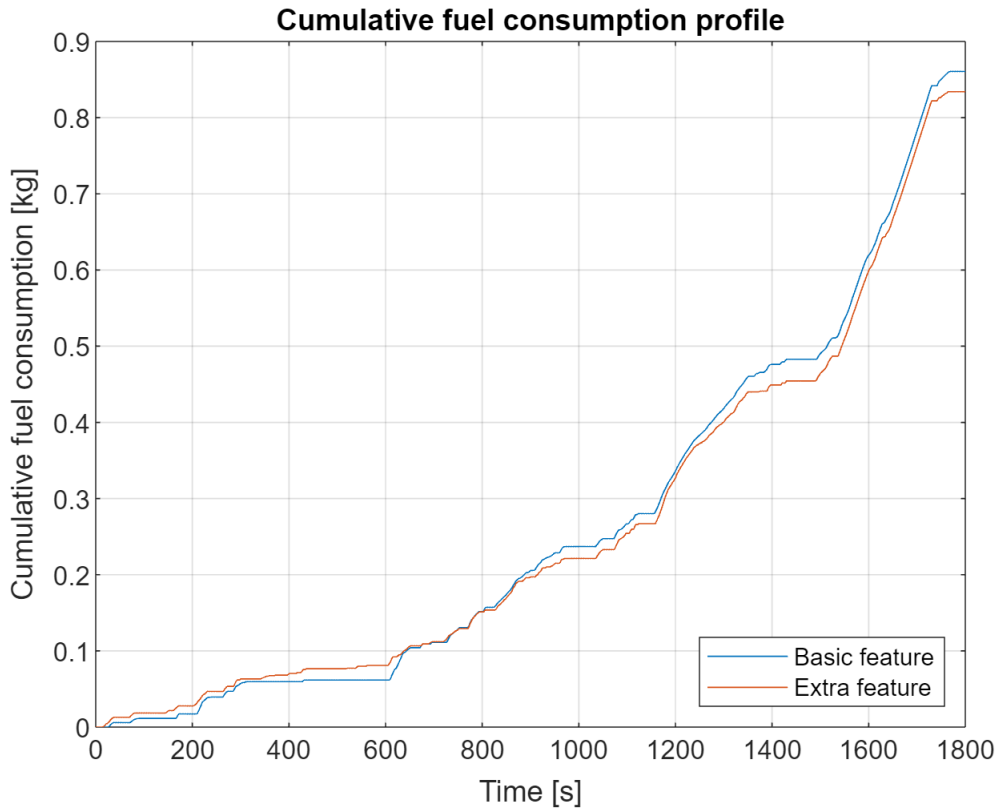


The torque-split factor of the basic control strategy, as highlighted in ICE maps, commands pure thermal operation in many parts of the cycle ($\alpha_{eng} = 1$). With the modified logic, many of these points are substituted by pure electric and battery charging. This trend underlines a criticity of the system: having set the same threshold level for these two operations ($\bar{\sigma}$), a continuously swinging behaviour is observed between them, negatively affecting driveability. A possible proposed solution is to introduce an hysteresys in the SOC behaviour, so that, after a change from pure electric to battery charging, a flag is rised setting the new pure electric SOC threshold to ($\bar{\sigma} + \Delta\sigma$), where $\Delta\sigma$ is the hysteresys amplitude. Doing so, the continuous switch between modes is limited.

- Fuel consumption along the cycle:

As done until now, a comparison between the two controllers is undertaken.

```
figure
plot(time, fcbasic), hold on
plot(time, fcextra), grid on
xlabel("Time [s]"); ylabel("Cumulative fuel consumption [kg]")
title('Cumulative fuel consumption profile')
legend('Basic feature', 'Extra feature',location ='southeast')
xlim([0, 1800])
```



Starting from the observations carried out until now, the fuel consumption trend clarifies the effectiveness of the controller. In the first part, the fuel consumption for the extra feature is higher, coherently with the fact that the battery discharges much less than in the cycle with the basic controller. While, interestingly, the trend switches for the other part of the cycle in favour of the modified controller, resulting in a lower final fuel consumption value.

- Speed profile showing the different operating modes:

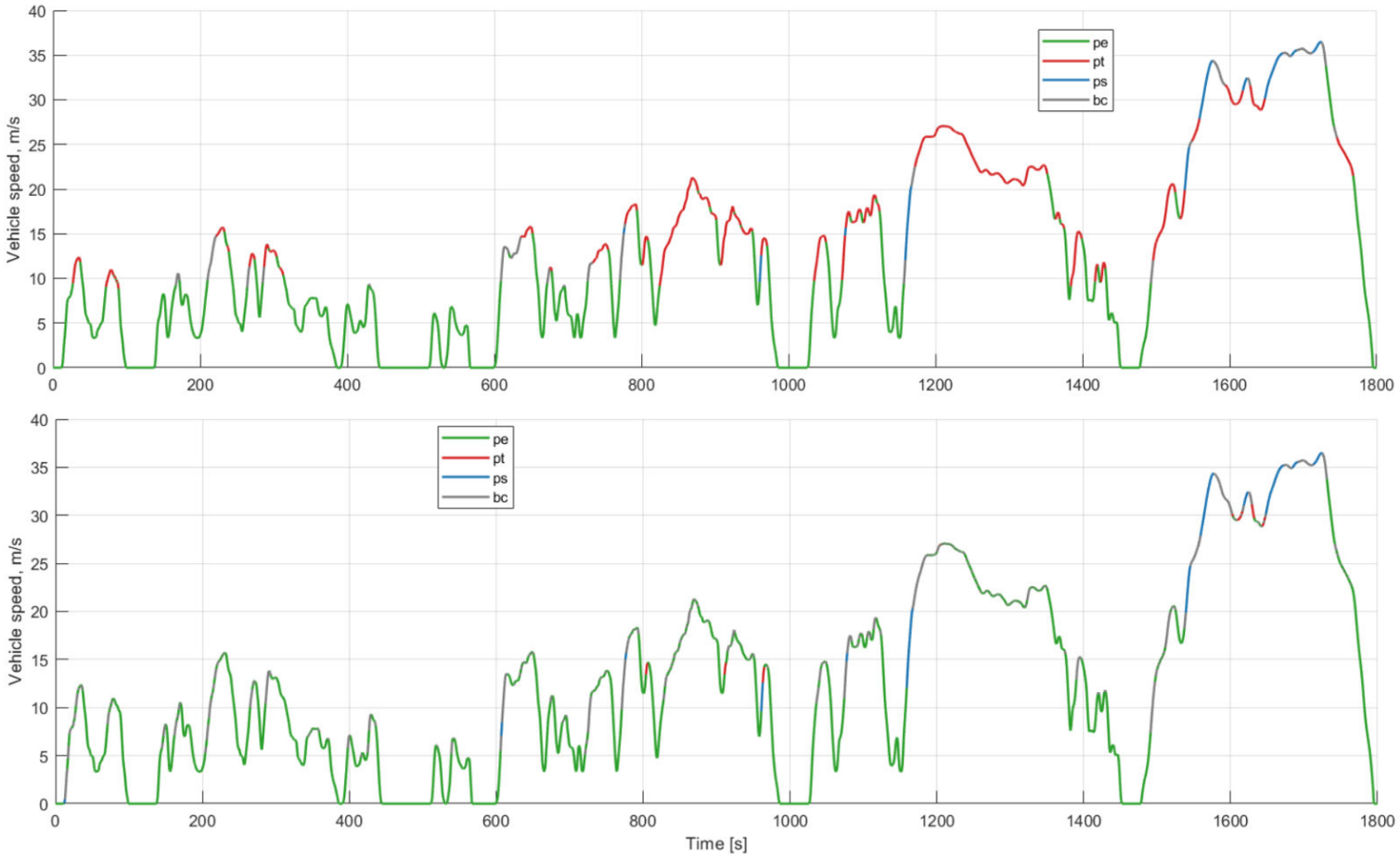


Fig.6: speed profile showing instantaneous operating mode for basic (top) and extra feature (bottom) controller

To conclude, also this last graph shows the trends previously highlighted: pure thermal mode experiences a significant reduction when switching to the updated logic; many points are substituted by pure electric operation also in high speed and low load situations, as well as battery charging. The general behaviour of the second strategy can be in this way summed up:

PROs:

- more regular SOC management and sustain;
- wider pure electric exploitation;
- lower cycle fuel consumption;

CONs:

- Bad driveability due to continuous switching behaviour;

Possible improvements:

- Implement an hysteresys logic for target SOC to avoid continuous switching.

	Basic	Extra feature
Fuel consumption [L/100 km]	4.714	4.567
Final battery SOC [p.u.]	0.586	0.635

Tab.1: Fuel consumption and final SOC comparison

Functions implementation

In this section the functions used in the code to calculate the torque-split factor are implemented.

Basic controller

For the basic control strategy, the function 'powerFlowControl.m' is designed, with the following sintax:

[PowerSplit] = powerFlowControl(shaftSpd, shaftTrq, vehSpd, pureEVspd, taoEM, SOC, SOCsustain, veh)

Function inputs:

- **shaftSpd:** speed of engine shaft [rad/s];
- **shaftTrq:** torque of engine shaft [Nm];
- **vehSpd:** vehicle speed [m/s];
- **pureEVspd:** maximum vehicle speed at which pure electric mode is allowed [m/s];
- **taoEM:** transmission ratio between EM and ICE shaft [-];
- **SOC:** battery state of charge [p.u.];
- **SOCsustain:** minimum SOC at which battery charging is allowed [p.u.];
- **veh:** data structure containing vehicle parameters.

Function output:

- **PowerSplit:** engine power-split factor.

Function logics:

Depending on the torque request, vehicle speed and actual SOC the controller falls in one of the following cases:

- **Pure electric mode:** ICE is switched off and all the torque is provided by EM; this condition is allowed during braking to recover energy and in traction only when the vehicle speed is lower than 'pureEVspd' to prevent ICE from working in its low efficiency region. The corresponding value is $\alpha_{eng} = 0$ because $T_{eng} = 0$;
- **Power split mode:** the torque request is provided partially by ICE and partially by EM; this condition is actuated when the torque demand T_{dem} is higher than ICE optimal operating line. The best solution would be setting ICE operating point on ool, with EM providing the torque difference. This is only possible when

the EM request is below its maximum torque. Differently, EM torque is set equal to its maximum value and ICE provides the difference. The corresponding value of α_{eng} is calculated according to this logic;

- **Pure thermal mode:** all the torque is provided by the ICE; this condition is chosen when the torque demand T_{dem} is lower than ICE optimal operating line and battery SOC is higher than 'SOCsustain' value; in this case the use of EM would further move ICE away from ool, deteriorating efficiency. The corresponding value is $\alpha_{eng} = 1$ because $T_{eng} = T_{dem}$;
- **Battery charging mode:** the torque sent by ICE is higher than the request, using the excess to motor the EM charging the battery; this condition is selcted when the torque demand T_{dem} is lower than ICE optimal operating line and battery SOC is lower than 'SOCsustain'; in analogy to power split mode, the optimal solution would be setting ICE operation on ool, with EM absorbing the negative torque difference. This is only possible when the EM request is above its minimum torque. If not, EM torque is set equal to its minimum value and ICE consequently provides the torque difference to satisfy the total request. The corresponding value of α_{eng} is calculated according to this logic.

```
function [PowerSplit] = powerFlowControl(shaftSpd, shaftTrq, vehSpd, pureEVspd,
taoEM, SOC, SOCsustain, veh)
% Extraction from structure veh of:
    oolTrq = veh.eng.oolTrq(shaftSpd); % ICE torque optimal operating line
    minTrqEM = veh.em.minTrq(taoEM*shaftSpd); % EM minimum torque given the EM
speed
    maxTrqEM = veh.em.maxTrq(taoEM*shaftSpd); % EM maximum torque given the EM speed
% Definition of engine torque-split factor in different cases
    if shaftTrq <= 0 || vehSpd < pureEVspd
        PowerSplit = 0; % pure electric mode
    elseif shaftTrq > oolTrq
        PowerSplit = max(oolTrq, shaftTrq-taoEM*maxTrqEM)/shaftTrq; % power split
mode
    elseif SOC > SOCsustain
        PowerSplit = 1; % pure thermal
    elseif SOC < SOCsustain
         PowerSplit = min(oolTrq, shaftTrq-taoEM*minTrqEM)/shaftTrq; % battery
charging
    end
end
```

Extra feature #2

For the updated control strategy, developing extra feature #2 and improved control logics, the function 'powerFlowControlExtraFeature.m' is designed, with the following sintax:

[PowerSplit] = powerFlowControlExtraFeature(shaftSpd, shaftTrq, vehSpd, pureEVspd, taoEM, SOC, veh)

Function inputs:

- **shaftSpd:** speed of engine shaft [rad/s];
- **shaftTrq:** torque of engine shaft [Nm];

- **vehSpd:** vehicle speed [m/s];
- **pureEVspd:** maximum vehicle speed at which pure electric mode is allowed [m/s];
- **taoEM:** transmission ratio between EM and ICE shaft [-];
- **SOC:** battery state of charge [p.u.];
- **veh:** data structure containing vehicle parameters.

Function output:

- **PowerSplit:** engine power-split factor.

Function logics:

The function is based on the same logical structure of 'powerFlowControl.m' with the following variations:

- Extra feature #2 is developed: the threshold 'SOCsustain' is not a constant but a function of time, given by:

$$\bar{\sigma} = \sigma_0 + \frac{1}{2} \cdot m_{veh} \cdot v_{veh}^2 \cdot \frac{f_{cs}}{E_b}$$

Parameters σ_0 and f_{cs} need to be tuned in this function. σ_0 must be set to 0.6 because even at very low speed the SOC must be mantained close to this value, being the system working in charge sustaining. For what concerns f_{cs}, a simple reasoning is performed: in order to push battery charging as speed increases, a reference value for σ_0 is imposed at highway speed $\left(v_{veh} = 130 \ \ km/h\right)$. After some trials, the value of 0.65 is chosen, since it provides a satisfactory trade-off between fuel consumption and final SOC. Once this value is set, f_{cs} is calculated via inverse formula.

- Additional conditions are added to allow pure electric: a check is performed on torque, verifying if the the demand is lower than the EM maximum one available, and a constraint is imposed to SOC, to keep it higher than $\bar{\sigma}$ (at least in pure electric);
- An additional check is added on SOC to engage power split mode: to avoid eccessive discharge in case of prolonged operation in this mode, a minimum value of 0.4 is imposed for battery health preservation;

```
function [PowerSplit] = powerFlowControlExtraFeature(shaftSpd, shaftTrq, vehSpd,
pureEVspd, taoEM, SOC, veh)
% Extraction from structure veh of:
    oolTrq = veh.eng.oolTrq(shaftSpd); % ICE torque optimal operating line
    minTrqEM = veh.em.minTrq(taoEM*shaftSpd);  % EM minimum torque given the EM
speed
    maxTrqEM = veh.em.maxTrq(taoEM*shaftSpd);  % EM maximum torque given the EM
speed
    SOC0 = 0.6; % [p.u.]
    SOC130 = 0.65; % SOC to be maintained at 130 km/h, used to compute f_cs
automatically [p.u.]
    f_cs = 2*(SOC130-SOC0)*veh.batt.nomEnergy*3600/(veh.body.mass*(130/3.6)^2); %
[-]
```

```
    SOCsustain = SOC0 + 0.5*(veh.body.mass*(vehSpd)^2)*f_cs/
(veh.batt.nomEnergy*3600); % extra feature formula [p.u.]
    if shaftTrq <= 0 || (vehSpd < pureEVspd && shaftTrq<maxTrqEM*taoEM &&
SOC>SOCsustain)
        PowerSplit = 0; % pure electric
    elseif shaftTrq > oolTrq && SOC > 0.4
        PowerSplit = max(oolTrq, shaftTrq-taoEM*maxTrqEM)/shaftTrq; % power split
    elseif SOC > SOCsustain
        PowerSplit = 1; % pure thermal
    elseif SOC < SOCsustain
         PowerSplit = min(oolTrq, shaftTrq-taoEM*minTrqEM)/shaftTrq; % battery
charge
    end
end
```

Assignment #2: ECMS

Table of Contents

Project introduction

The aim of the project is to design a controller for the evaluation of the torque-split factor and engaged gear of a p2 parallel HEV (represented in Fig.1) at each time instant of a given driving profile.

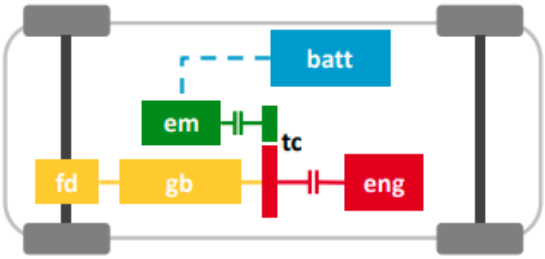


Fig.1: Scheme of a P2 parallel HEV

The control variable engine torque-split factor is defined accordingly:

$$\alpha_{eng} = \frac{T_{eng}}{T_{dem}}$$

In particular, the controller follows an equivalent consumption minimization strategy (ECMS); this is a local minimization strategy which selects, at any instant, a combination of the torque-split factor (α_{eng}) and gear (γ) that provides the minimum value of a cost function among a significant number of different combinations. This function is called equivalent fuel consumption and considers two contributions: the actual fuel flow rate in the operating point and a virtual one, representing the equivalent fuel that has to flow in a following instant to compensate for the induced variation of the battery state of charge (SOC, σ).

$$\dot{m}_{f,eq} = \dot{m}_f(\gamma, \alpha_{eng}) - s \cdot \frac{E_b}{Q_{LHV}} \cdot \dot{\sigma}(\gamma, \alpha_{eng})$$

Equivalence factor

In particular, the virtual fuel consumption term can have a different weight in the function, depending on the value of the Equivalence factor (s): the higher s, the higher the cost attributed to battery energy depletion. So this parameter strongly affects the SOC evolution throughout the driving cycle and it is strictly dependent on the considered mission. In the project, s is calibrated adopting a bisection algorithm, with the goal to obtain a final SOC value sufficiently close (within given boundaries) to the initial one.

Eventually, the behaviour of the controller with the calibrated equivalence factor is evaluated by the analysis of the most significant performance indicators.

Group information

Group number: 49

Students:

- Carlo Vittorio Colucci, s329703
- Riccardo Bressani, s323665
- Luca Marchetto, s323437

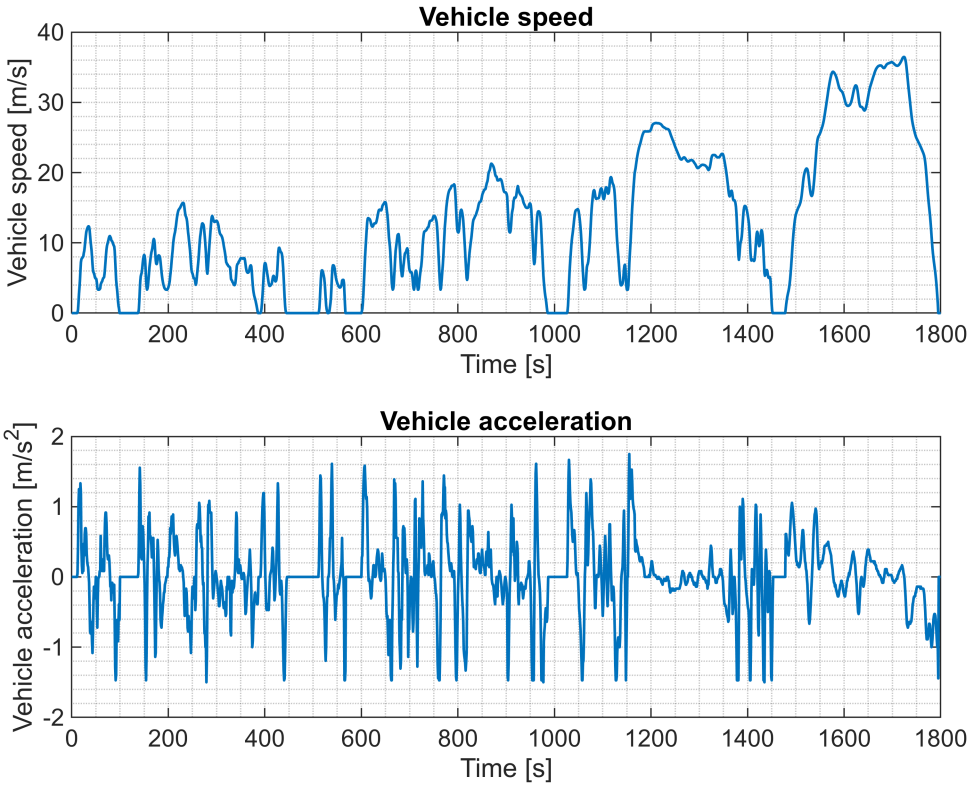
Load the cycle and vehicle data

The reference cycle which is used in this project is loaded in the following section: it consists in a vector representing a speed profile with respect to time. The data concerning the cycle are contained in the file 'WLTP3.mat'.

Starting from the speed profile, the acceleration is computed performing a discrete derivative operation: each component of the vector is obtained dividing the difference between two consecutive speed values by the time step. Doing so, the obtained vector of acceleration has a length (N-1), where N is the number of time instants; a first component equal to 0 is added to the acceleration vector in order to work with vectors of the same length. This assumption is coherent with the cycle since the first terms of the speed vector are constant and equal to 0.

```
clear all
clc
% Addition of folders
addpath('data')
addpath('models')
addpath('utilities')
% Extraction of cycle velocities and accelerations
mission = load('data\WLTP3.mat');
vehSpd = mission.speed_kmh./3.6; % [m/s]
time = mission.time_s; % [s]
dt = time(2) - time(1); % [s]
vehAcc = (vehSpd(2:end)-vehSpd(1:end-1))./dt; % [m/s^2]
% First component assumed 0 added to the acceleration vector
vehAcc = [0;vehAcc]; % [m/s^2]
% The speed and acceleration profiles are plotted against time
t = tiledlayout(2,1);
nexttile(1)
plot(time,vehSpd,'LineWidth',1)
```

```
grid minor
title('Vehicle speed')
xlabel('Time [s]')
ylabel('Vehicle speed [m/s]')
nexttile(2)
plot(time,vehAcc,'LineWidth',1)
grid minor
title('Vehicle acceleration')
xlabel('Time [s]')
ylabel('Vehicle acceleration [m/s^2]')
```



By using the following commands, the data related to the vehicle are loaded from the file 'vehData.mat'. These data are subsequently rescaled using the function 'scaleVehData.m' considering as additional inputs the values of ICE power [W], EM power [W], battery capacity [Ah] associated to the group number:

- **Group number:** 49
- **ICE power:** 60000 W
- **EM power:** 18000 W
- **Battery capacity:** 6.4 Ah

```
% Data loading and scaling
veh = load('data\vehData.mat');
engPwr = 60000; % [W]
emPwr = 18000; % [W]
battCap = 6.4; % [Ah]
```

```
veh = scaleVehData(veh,engPwr,emPwr,battCap); % scaleVehData(veh, engPwr[W],
emPwr[W], battCap[Ah])
```

Equivalence factor calibration

The bisection algorithm is the method implemented for the calibration of the equivalence factor s. Its goal is to find an s value for which the final SOC is equal to the initial value, with a precision of $\pm 1\%$ (i.e. $0.59 < \sigma_f < 0.61$). This is realized by an iterative procedure.

To start the calibration process of the equivalence factor s, it is necessary to identify suitable boundary values: they must provide two values of ψ of opposite sign, where $\psi = \sigma_f - \sigma_0$. It is shown that the initial interval [2,3] satisfies this requirement. The final SOC variation is computed by the function 'SOCvariation' which, starting from the values of α_{eng} and γ elaborated by ECMS control (explained later), provides the SOC evolution throughout the driving cycle.

```
% Initial conditions:
SOC0 = 0.6; % [p.u.]
% Boundary values for bisection algoritm start
a = 2;

[psi, ~] = SOCvariation(a,SOC0,mission,veh); % psi = SOC_final-SOC0 [p.u.]
psi % check that psi(a)<0
```

```
psi = -0.1829
```

```
b = 3;
[psi, ~] = SOCvariation(b,SOC0,mission,veh); % [p.u.]
psi % check that psi(b)>0
```

```
psi = 0.2000
```

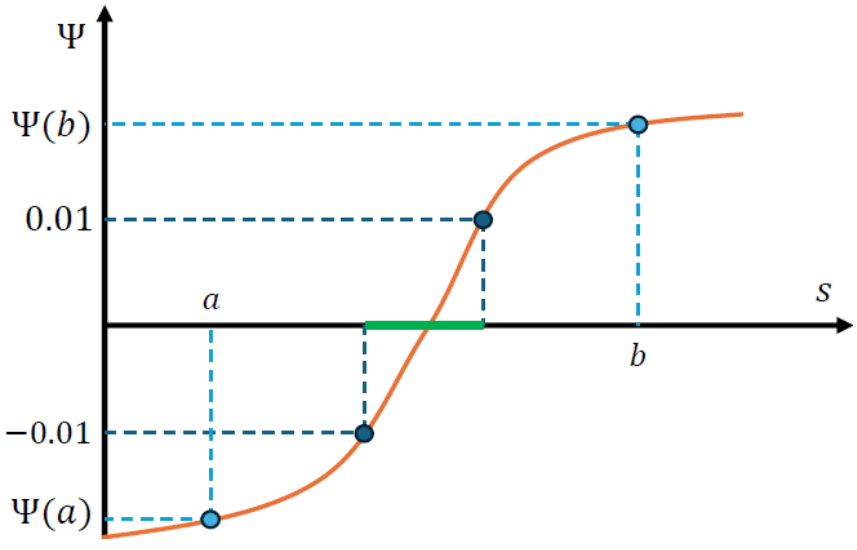


Fig.2: Schematic illustration of bisection logic

Considering the set interval [a,b], the function ψ is calculated in its mid point ***c***. Then, if the value assumed by the function is out of the desired range centered in 0, the obtained value of ***c*** is taken as new interval boundary,

coherently with the sign logic, regarding ψ, previously explained. Thus, the procedure is repeated until a ψ value inside the range ($-0.01 < \psi < 0.01$) is found out, standing for an equivalent factor falling inside the green interval depicted in Fig.2. To check the correctness of the controller operation, the presence of any unfeasibility condition is checked at any time instant, also for non optimal equivalence factor values.

```
% Bisection algorithm loop
while abs(psi) > 0.01 % Final SOC within 0.59 and 0.61
    EqFactor = (a+b)/2; % [-]
    [psi, prof] = SOCvariation(EqFactor,SOC0,mission,veh);
    % Unfeasibility check
    for n = 1:length(time)
        if prof.unfeas(n)==1
            if prof.engPrf.engSpdUnfeas(n)==1
                disp(['At iteration ' num2str(n) ' for equivalence factor s='
num2str(EqFactor) ' the engine speed exceeds the limits.'])
            end
            if prof.engPrf.engTrqUnfeas(n)==1
                disp(['At iteration ' num2str(n) ' for equivalence factor s='
num2str(EqFactor) ' the engine torque exceeds the limits.'])
            end
            if  prof.emPrf.emSpdUnfeas(n)==1
                disp(['At iteration ' num2str(n) ' for equivalence factor s='
num2str(EqFactor) ' the electric motor speed exceeds the limits.'])
            end
            if  prof.emPrf.emTrqUnfeas(n)==1
                disp(['At iteration ' num2str(n) ' for equivalence factor s='
num2str(EqFactor) ' the electric motor torque exceeds the limits.'])
            end
            if prof.vehPrf.battUnfeas(n) == 1
                disp(['At iteration ' num2str(n) ' for equivalence factor s='
num2str(EqFactor) ' the battery current exceeds the limits.'])
            end
            if  prof.vehPrf.pwtUnfeas(n)==1
                disp(['At iteration ' num2str(n) ' for equivalence factor s='
num2str(EqFactor) ' uncoherent EM torque request.'])
            end
        end
    end
    % Set new bisection boundaries
    if psi>0
        b = EqFactor;
    else
        a = EqFactor;
    end
end
EqFactorCalibrated = EqFactor % Final equivalence factor [-]
```

```
EqFactorCalibrated = 2.5781
```

Calibrated equivalence factor simulation

A further simulation is undertaken with the final ***s*** value; moreover the results are used to compute the values of fuel consumption, in terms of absolute mass of fuel, and fuel economy, in terms of volume of fuel by distance.

In addition, the 'prof' structure collects all the vehicle state variables time profiles, that can be monitored to analyse the controller performance later on.

```
[psi, prof] = SOCvariation(EqFactorCalibrated,SOC0,mission,veh);
fuelConsumption = trapz(time, prof.engPrf.fuelFlwRate)/1000;  % Cumulative fuel
consumption [kg]
vehDist = trapz(time, vehSpd);  % Total cycle distance [m]
fuelEconomy=(fuelConsumption*10^5)/(veh.eng.fuelDensity*vehDist)    % Fuel
consumption [L/100km]
```

```
fuelEconomy = 4.1524
```

```
finalSOC = SOC0+psi % [p.u.]
```

```
finalSOC = 0.6045
```

In the 'Results analysis' section a comparison between these results and the ones obtained in Project 1 is shown.

Save results

The following code saves the cycle time profiles, the fuel consumption (in kg), the fuel economy (in L/100km), the final SOC and the calibrated equivalence factor in a mat-file.

```
% Store results
save("results_ecms.mat", "prof", "fuelConsumption", "fuelEconomy", "finalSOC",
"EqFactorCalibrated")
```

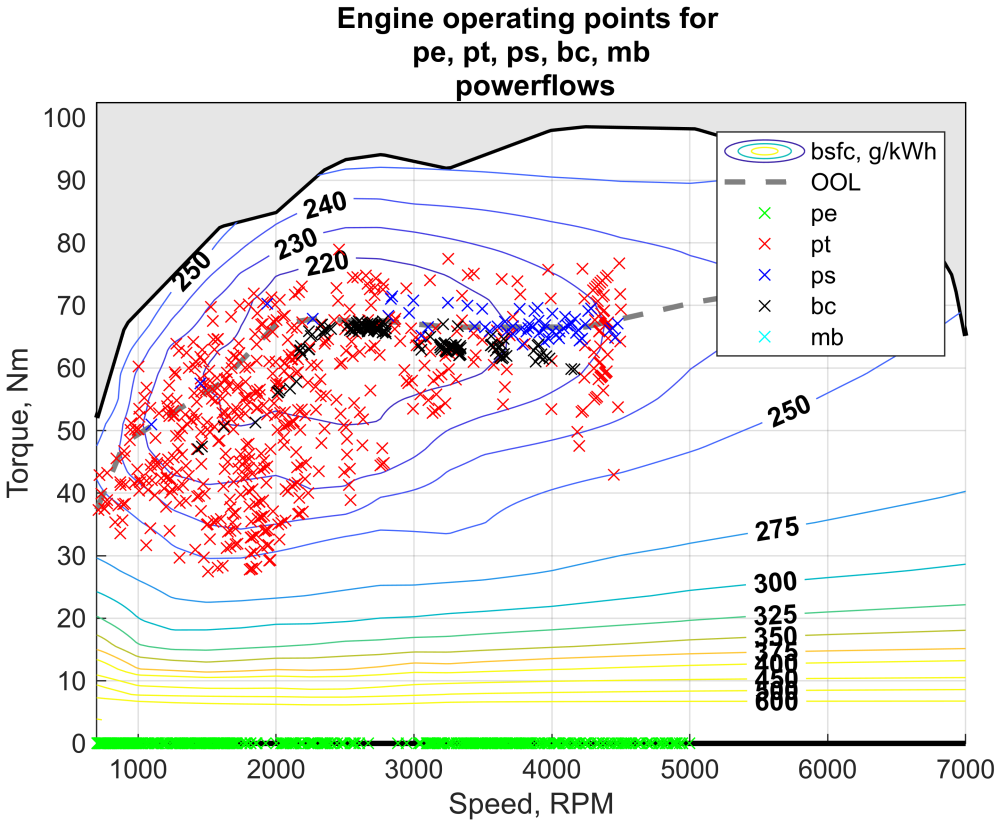
- `prof` is the structure of profiles created in the previous section;
- `fuelConsumption` is the total fuel consumption for the whole mission (in kg);
- `fuelEconomy` is the distance-specific fuel consumption for the whole mission (in L/100km);
- `finalSOC` is the final battery SOC (per unit value);
- `EqFactorCalibrated` is the value of s calibrated by means of the bisection algorithm.

Controller analysis

The following section proposes some useful tools for the analysis of the controller behaviour; the first two graphs are the ICE and EM maps, where the cycle working points are shown, highlighting the different operating modes. In particular the ICE plot offers the possiblity to display different performance indicators: fuel consumption, brake specific fuel consumption and efficiency. The optimal operating line (ool) is plotted as well, representing the locus of points of maximum efficiency for each engine speed. This is useful to evaluate the controller performance, focusing on the distance between working points and 'ool': the smallest it is, the better the fuel energy is exploited. For what concerns the EM graph, only efficiency isolines are present.

- **ICE map:**

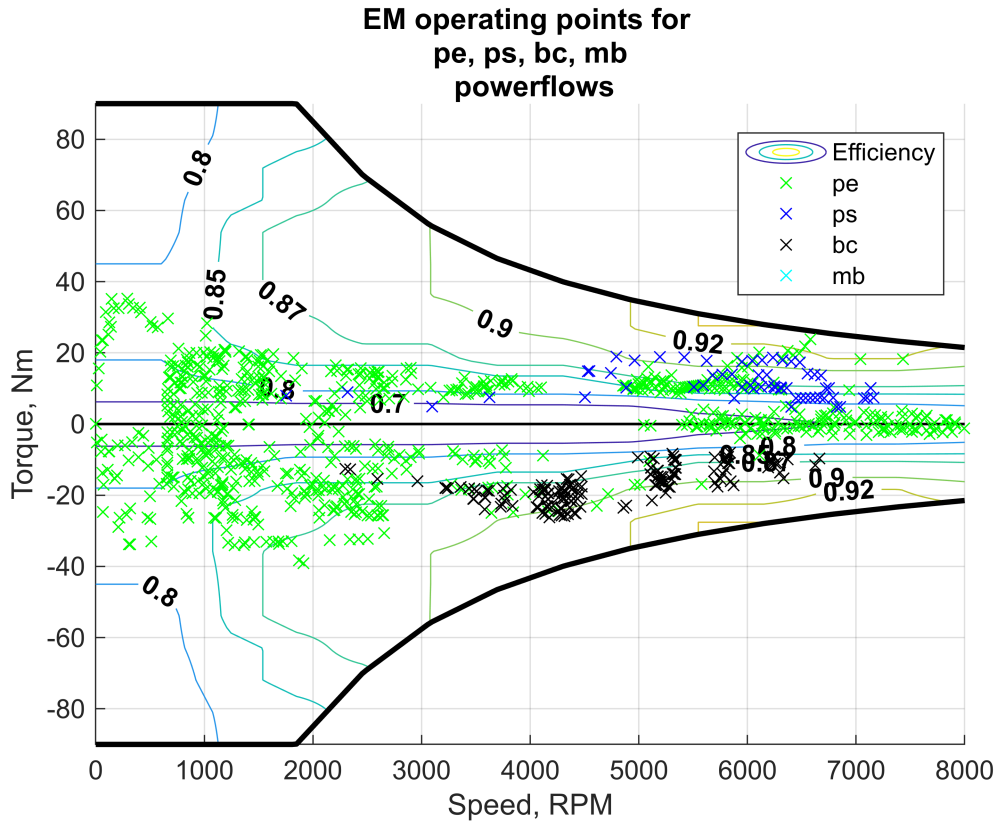
```
% Engine operating points
contour_type="bsfc";
switch contour_type
    case 'fuel consumption'
        engMapWithPF(veh.eng, prof, "fc", 'all')
    case 'bsfc'
        engMapWithPF(veh.eng, prof, "bsfc", 'all')
    case 'efficiency'
        engMapWithPF(veh.eng, prof, "eff", 'all')
end
```



It is interesting to observe that, even if the logic of the controller is not explicitly conceived to force the ICE operation towards the ool, still the envelope of operating points is located in a quite narrow region close to it. By the way, this is a consequence of the minimization strategy adopted, bringing the engine to work in its high efficiency region. Especially in combined operation (ps and bc) the points are very close to ool, since an additional degree of freedom is present with respect to pure thermal mode, in which the exact torque request must be satisfied by the ICE.

- **EM map:**

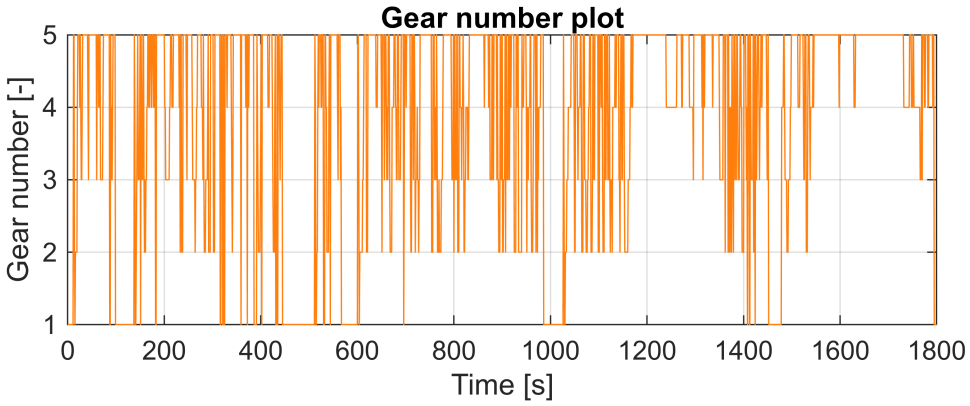
```
% EM operating points
emMapWithPF(veh.em, prof, ["pe", "ps", "bc", "mb"]);
```



The working points of the EM are widely spread around the efficiency map: this can be explained considering that the goal of the controller is not to optimize the efficieny of the two machines individually, but the overall system one.

- **Gear number evolution:**

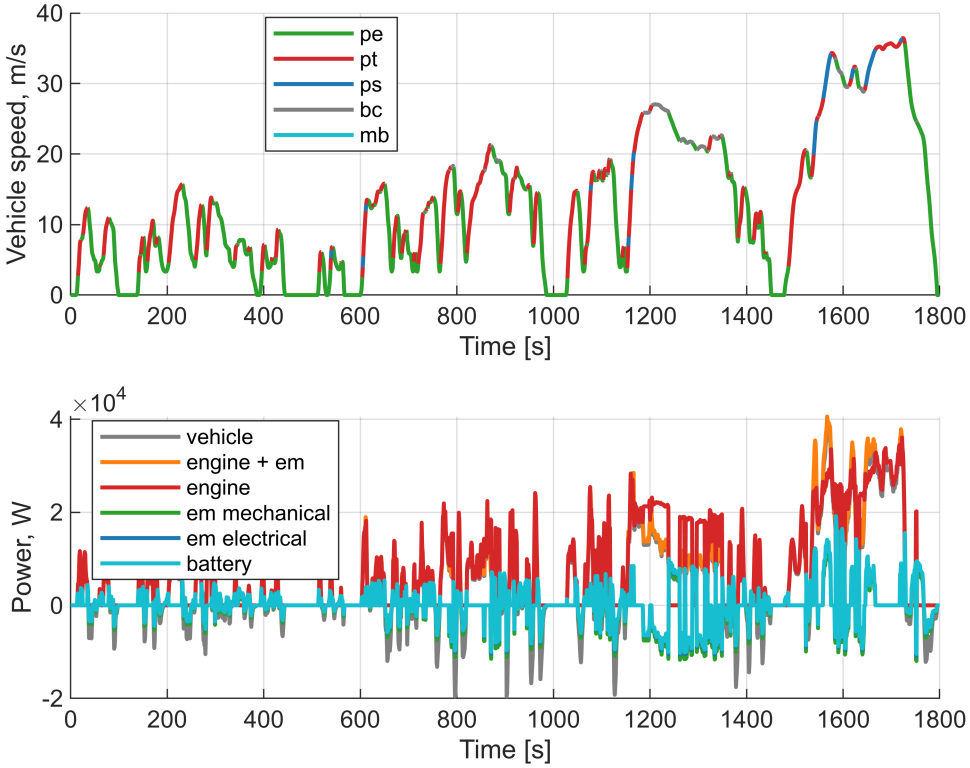
```
% Main profiles
figure
plot(time, prof.vehPrf.gearNumber, 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
ylabel("Gear number [-]")
ylim([1 5])
pbaspect([3 1 1])
title('Gear number plot')
```



An interesting observation on the system behaviour can be provided by the gear number time profile: being the logic of the controller only focused on equivalent consumption minimization, no constraint is applied to the gear shifting logic, leading to a behaviour characterized by very frequent gear changes, often between non consecutive gears. This can be a significant drawback from the driveability point of view, which for sure cannot be neglected in the system final implementation. Thus, an additional control logic should be implemented in order to fix this problem.

- **Power flows:**

```
powerProfiles(prof, 'all');
```



It is observed that, with the aim of equivalent consumption minimization, the traction zones of the cycle are almost completely governed by pure thermal and power split modes. This is justified observing that, in ICE map, many working points of these modes fall in the high efficiency region (close to ool). By converse, battery charging and pure electric traction are only involved in low torque demand conditions. Moreover, decelerating situations are always managed in pure electric, providing regenerative braking. The condition of pure mechanical braking is never verified since, with the calibrated value of equivalence factor, the SOC never exceeds the maximum limit.

Results comparison with rule based controller of Project 1

It is now of interest to compare the results of the ECMS controller with the ones of the rule based control of Project 1, observing how for the same vehicle and powertrain parameters, a different logic can significantly affect the results. The 'extra feature' results are considered as reference.

- **ICE maps:**

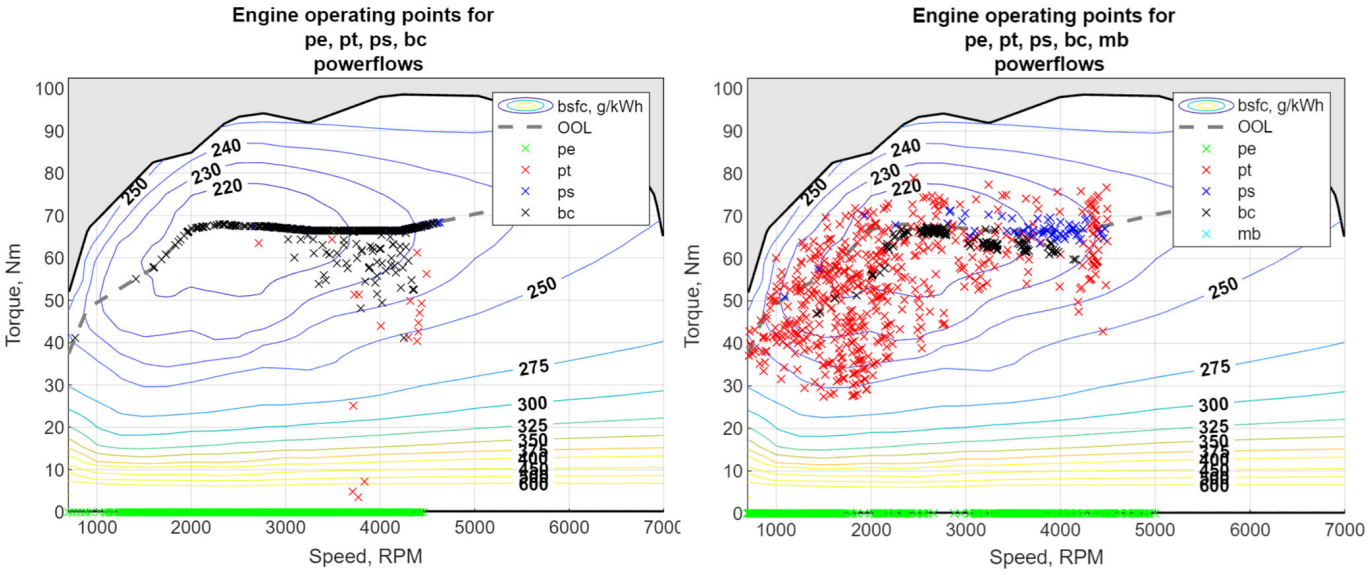


Fig.3: ICE maps with operating points of rule based (left) and ECMS (right) controller

It's possible to notice that visually the working points of the rule based controller are more concentrated around ool, with a more consistent utilization of battery charging mode. By the way, this evidences a limit of this kind of controllers: it is focused on the optimization of the ICE operation, although this does not necessarily coincide with the optimization of the overall system. The ECMS control is able, by converse, to get closer to the global system optimal operation. This can be confirmed observing the total energy spent throughout the cycle, given by the sum of the fuel energy and the electric one, related to battery SOC variation.

```
% Avoid to overwrite the data when the Extra feature ones are loaded
prof_ecms = prof;
fuelConsumption_ecms = fuelConsumption; % [kg]
load('resultsExtraFeature.mat')

% Rule based controller
elEnergyConsumption_rb = trapz(time, prof.battPrf.battVolt.*prof.battPrf.battCurr);
% battery electrical energy [J]
fuelEnergyConsumption_rb = fuelConsumption*veh.eng.fuelLHV; % [J]
totalEnergyConsumption_rb = (elEnergyConsumption_rb + fuelEnergyConsumption_rb)/
1000/3600 % [kWh]
```

```
totalEnergyConsumption_rb = 9.9940
```

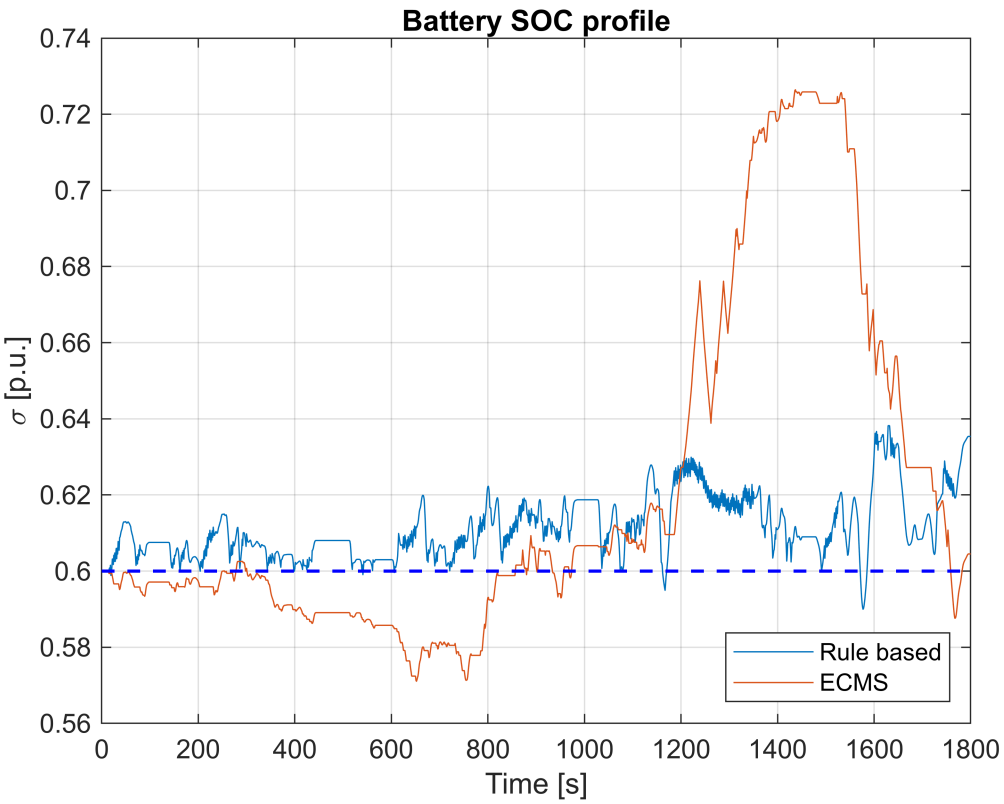
```
% ECMS controller
elEnergyConsumption_ecms = trapz(time,
prof_ecms.battPrf.battVolt.*prof_ecms.battPrf.battCurr); % [J]
fuelEnergyConsumption_ecms = fuelConsumption_ecms*veh.eng.fuelLHV; % [J]
totalEnergyConsumption_ecms = (elEnergyConsumption_ecms +
fuelEnergyConsumption_ecms)/1000/3600 % [kWh]
```

```
totalEnergyConsumption_ecms = 9.1298
```

These numbers take into account not only the fuel consumption contribution, but also the difference of the final SOC between the two controllers, providing a complete information about the energy globally used.

- **SOC profiles:**

```
figure
plot(time, prof.battPrf.battSOC), hold on
plot(time, prof_ecms.battPrf.battSOC), grid on
plot(time,0.6*ones(size(time)),LineStyle="--",Color="b",LineWidth=1.2)
xlabel("Time [s]"); ylabel("\sigma [p.u.]")
title('Battery SOC profile')
legend('Rule based', 'ECMS',location ='southeast')
xlim([0, 1800]);
```

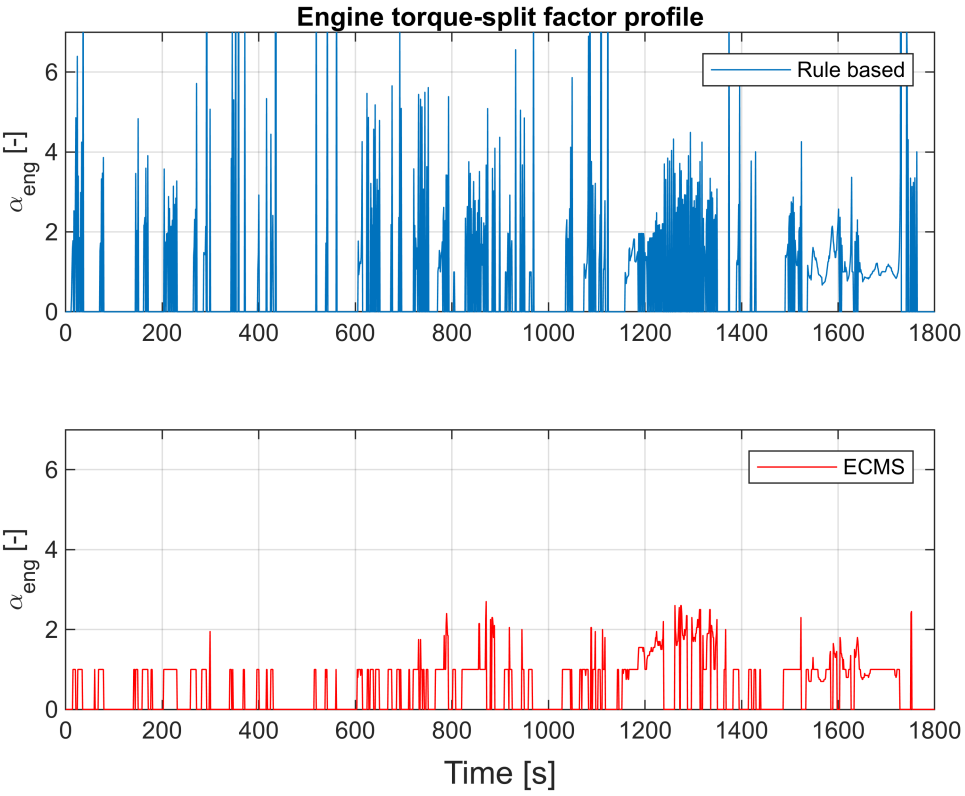


The ECMS controller SOC profile shows a much higher variation throughout the cycle, since it is not directly controlled in real time, but is only influenced by the initial and final desired values. Instead, it is observed that the rule based logic is characterized by a more controlled trend of the SOC, since this quantity is directly governed inside the algorithm. Besides, it must be highlighted that the desired behaviour of the ECMS is obtainable only if the driving mission is known in advance, while a different cycle could lead to unexpected results.

- **Engine torque-split factor profiles:**

```
% Engine torque-split factor profile
figure
```

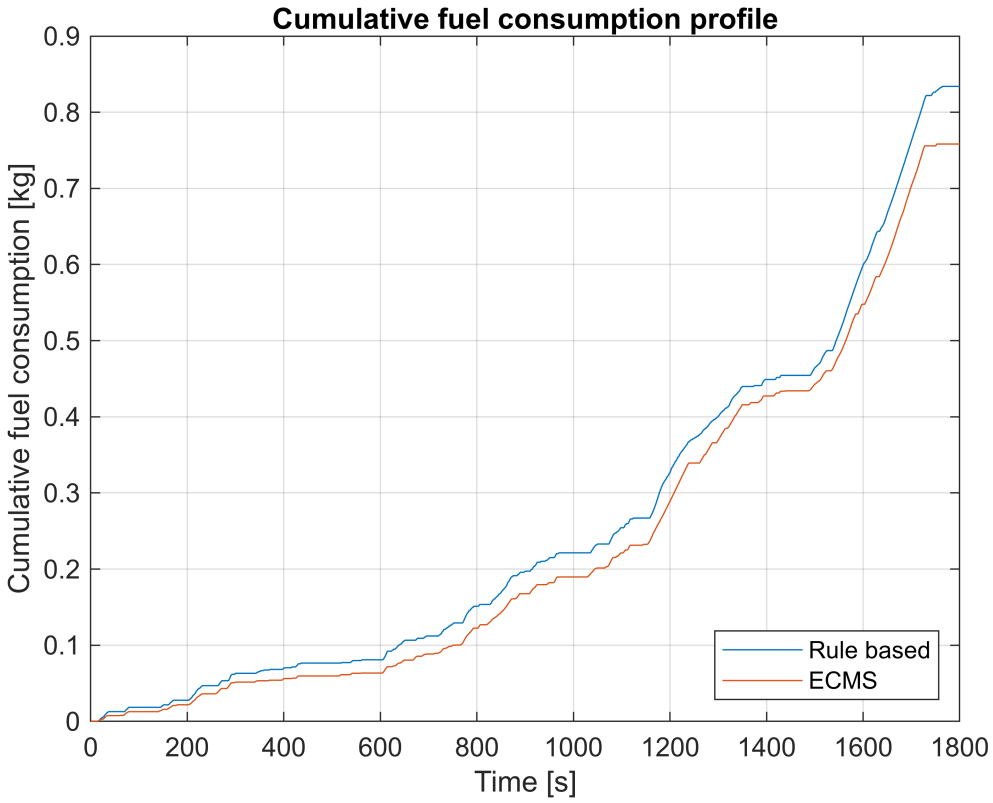
```
t = tiledlayout(2,1);

ax1 = nexttile;
plot(time, prof.vehPrf.engAlpha), grid on
ylabel("\alpha_{eng} [-]")
title('Engine torque-split factor profile')
legend('Rule based')
xlim([0, 1800]), ylim([0,7])

ax2 = nexttile;
plot(time, prof_ecms.vehPrf.engAlpha, Color="r"), grid on
ylabel("\alpha_{eng} [-]")
legend('ECMS')
xlabel(t, "Time [s]", 'FontSize', 12)
linkaxes([ax1 ax2], 'x')
xlim([0, 1800]), ylim([0,7])
```



For what concerns the torque-split factor, from a graphical interpretation it is evident how the ECMS selects values within a more restricted range, providing at the same time a more regular behaviour. This can provide a positive effect on driving experience, avoiding frequent mode switching, which may be acoustically percieved by the occupants.

- **Fuel consumption along the cycle:**

```
figure
fc_rb = cumtrapz(time, prof.engPrf.fuelFlwRate)/1000;   % [kg]
fc_ecms = cumtrapz(time, prof_ecms.engPrf.fuelFlwRate)/1000;    % [kg]
plot(time, fc_rb), hold on
plot(time, fc_ecms), grid on
xlabel("Time [s]"); ylabel("Cumulative fuel consumption [kg]")
title('Cumulative fuel consumption profile')
legend('Rule based', 'ECMS',location ='southeast')
xlim([0, 1800])
```



The result is coherent with the considerations undertaken so far, showing that, at any time instant, the trend of ECMS is lower than the one from the rule based controller.

To conclude, the most significant results are recalled in Tab.1:

	Project 1 – Rule based	Project 2 - ECMS
Fuel consumption [L/100km]	4.5670	4.1524
Final battery SOC [p.u.]	0.6350	0.6045
Total energy consumption [kWh]	9.9940	9.1298

Tab.1: Comparison of fuel consumption, final SOC and total energy consumption between different controllers.

The ECMS behaviour is summarized as follows:

PROs:

- Lower fuel and energy consumption;
- Close-to-optimal hybrid system exploitation;
- Regular torque-split factor behaviour.

CONs:

- Driveability issues related to gear shifting;
- Higher computational cost;
- Knowledge of the driving mission required for the equivalence factor calibration.

Possible improvements:

- Implementation of a specific logic to provide smoother gear changes, limiting the shifts between far gears and avoiding too frequent shifts;
- Development of an algorithm to shift from fixed equivalence factor to an online calibrated one; this could be done in two different ways: first, multiplying the constant factor by a penalty function, attributing an appropriate weight to the battery energy depletion, depending on the actual SOC; this would prevent uncontrolled behaviour of the SOC. Second, analysing the actual driving profile, to develop a predictive model to estimate a possible subsequent driving cycle. The equivalence factor is calibrated as well on this prediction.

Functions implementation

In this section the functions 'ecmsControl' and 'SOCvariation' used in the code are implemented. The first contains the logic for the choice of the control variables, while the second exploits the first to perform the driving cycle simulation, to retrieve the time profiles of the main quantities of interest.

ECMS controller

For the ECMS control strategy, the function 'ecmsControl.m' is designed, with the following sintax:

[GN, PowerSplit] = ecmsControl(SOCactual, EqFactor, vehSpd, vehAcc, veh)

Function inputs:

- **SOCactual**: instantaneous battery state of charge [p.u.];
- **EqFactor:** value of equivalence factor s [-] ;
- **vehSpd:** vehicle speed [m/s];
- **vehAcc:** vehicle acceleration [m/s^2];
- **veh:** data structure containing vehicle parameters.

Function outputs:

- **GN:** engaged gear [-];
- **PowerSplit:** engine torque-split factor [-].

Function logics:

The goal of the function is to create a matrix, in each time instant, of the equivalent fuel consumption values for all the combinations of α_{eng} and γ given the speed and acceleration cycle requests; the minimum value, among the computed ones, is considered as working point, retrieving the corresponding gear number and torque-split factor.

To avoid the usage of computationally demanding concatenate cycles, the 'hev_dp_model' function is used to directly deal with input vectors, computing the fuel consumption for all the combinations of (γ, α_{eng}) within a single function iteration, leading to a significant reduction in the simulation time from several minutes to less than one minute (at least on the computer used). For reference, an equivalent logic, based on concatenate for cycles, iterating the basic version of 'hev_model.m', was adopted as first attempt. This version worked as well, but required much more time in the execution of the code.

The improved function has the following sintax:

[x_next, stageCost, unfeas, engPrf, emPrf, battPrf, vehPrf] = hev_dp_model(x, u, w, veh).

With respect to 'hev_model' the u input of the function is a cell containing two vectors: one is a column vector of the gear numbers (from 1 to 5 in this case) and the other one is a row vector containing all the discrete values of the torque-split factor chosen for the simulation: a range between 0 and 3 is adopted with a step of 0.1, since it is observed that, even trying with a wider range, values above 3 are never selected by the algorithm. In addition, increasing the discretization accuracy doesn't provide significant changes in the results.

The function provides the fuel consumption value, as well as the next SOC value, for each combination of control variables, collecting them into matrices.

With the obtained values of possible future SOC, the discrete time derivative is computed as difference with the actual value, divided by the time step. This, in combination with the fuel conumption data, allows to calculate the equivalent fuel consumption matrix.

Among these possible combinations, not all can be selected by the control. In particular, the situations that must be avoided are the following:

- Unfeasible working points of ICE or EM;
- $SOC_{NEXT} < 0.4$;
- $SOC_{NEXT} > 0.8$.

The first two cases can be easily managed by the application of a penalty to the equivalent fuel consumption value so that, once its minimum value is searched, these are automatically ignored.

The last presents a further criticity: in a case in which the actual SOC is already close to 0.8 and the torque demand becomes negative, the only feasible solution for the system is to choose pure electric (regenerative braking); indeed, all the other torque-split values would arise an unfeasible condition, requiring negative torque to ICE. However, this keeps increasing the SOC. Penalising also this solution would result in a penalisation of all the possible combinations, bringing the system to select a meaningless operating condition.

To avoid this, only overcharge conditions occurring during traction (SOCnext>0.8 & demTrq>0) are immediately penalized. For the ones occurring during braking, a specific strategy is developed to interrupt regeneration

enabling only mechanical braking: the ECMS control returns a meaningless value (-1) of α_{eng}, specifically conceived to identify this situation. Then, when this value is subsequently detected by the function 'SOCvariation', a specific logic implemented in 'hev_model' manages this condition.

This countermeasure is conceived with a particular attention for battery safety: it avoids that, during prolongued regenerative braking, the SOC increases uncontrolled going above the upper limit. With the adopted strategy the SOC can never exceed 0.8.

Eventually, the minimum equivalent fuel consumption is identified and, consequently, the corresponding α_{eng} and γ indexes. The values of the control parameters are so chosen and returned by the function.

```
function [GN, PowerSplit] = ecmsControl(SOCactual, EqFactor, vehSpd, vehAcc, veh)
    PowerSplit_vec = linspace(0,3,61); % alpha values [-]
    GN_vec = [1:length(veh.gb.spdRatio.Values)]'; % column vector of gear number [-]
    [SOCnext, stageCost, unfeas, engPrf, emPrf, ~, ~] = hev_dp_model({SOCactual},
{GN_vec,PowerSplit_vec}, {vehSpd, vehAcc}, veh);
    SOCnext = SOCnext{1,1}; % Conversion from cell to matrix

    sigma_dot = (SOCnext-SOCactual)/veh.dt; % discrete SOC derivative [p.u./s]
    fcEquivalent = stageCost - EqFactor*(veh.batt.nomEnergy*3600)*1000/
(veh.eng.fuelLHV)*sigma_dot; % equivalent fuel consumption matrix [g/s]
    demTrq = engPrf.engTrq + emPrf.emTrq; % Needed to determine if traction or
braking [Nm]
    fcEquivalent(unfeas==1 | SOCnext<0.4 | (SOCnext>0.8 & demTrq>0)) =
fcEquivalent(unfeas==1 | SOCnext<0.4 | (SOCnext>0.8 & demTrq>0)) + 100; % exclusion
of unfeasible solutions through penalty

    [~, idx] = min(fcEquivalent(:)); % returns scalar index identifying the minimum
position
    [GN_idx, PowerSplit_idx] = ind2sub(size(fcEquivalent), idx); % transforms the
scalar index 'idx' into row and column numbers
    GN = GN_idx; % gear number coincides with row index [-]
    PowerSplit = PowerSplit_vec(PowerSplit_idx); % [-]

    % Check if regenerative braking is charging over 0.8
    [SOCnext, ~, ~, ~, ~, ~, ~] = hev_model(SOCactual, [GN, PowerSplit], [vehSpd,
vehAcc], veh); % [p.u.]
    if SOCnext > 0.8
        PowerSplit = -1; % Sets a meaningless value to detect overcharge situation
[-]
    end
end
```

SOCvariation

The following function performs the calculations related to the entire driving cycle and returns the SOC variation $\sigma(t_f) - \sigma(t_0)$, useful for the application of the bisection algorithm; moreover, time profiles of the main vehicle quantities are stored in the structure 'prof'. The function sintax is shown:

[psi, prof] = SOCvariation(EqFactor,SOC0,mission,veh)

Function inputs:

- **EqFactor:** value of the equivalence factor s [-] ;
- **SOC0:** initial value of the battery SOC [p.u.];
- **mission:** data structure containing cycle speed profile [km/h] and time vector [s] ;
- **veh:** data structure containing vehicle parameters.

Function outputs:

- **psi:** SOC variation between final and initial values ($\sigma(t_f) - \sigma(t_0)$) [p.u.];
- **prof:** structure containing time profiles of ICE and EM torque and speed, battery SOC, cumulative fuel consumption, α_{eng}, engaged gear number and unfeasibility vectors.

Function logics:

The starting point is the extraction of the speed and acceleration profiles characteristic of the driving cycle.

Then, by means of an iterative procedure, for each cycle time instant the actual gear and torque-split strategy are computed by the ECMS controller and the evolution of the vehicle variables is evaluated by the 'hev_model' function. Its working principle is the same as the previous project, making exception for the implementation of a new control for the mechanical braking condition; this operating mode (identified when $\alpha_{eng} = -1$) is necessary in order to deal with the previously explained situation, in which the torque request is negative with the SOC close to its upper bound. In this case, to stop the power delivery to the battery, the electrical machine torque, as well as the ICE torque, are set equal to 0, assuming that all the braking torque is provided by the mechanical brake system. As a consquence, the vehicle states and unfeasibilities are correctly calculated and the system never exceeds the maximum SOC. Moreover, an additional power flow related to this condition is introduced, which allows to graphically visualize it in the data analysis section when it occurs.

To conclude, the variation between final and initial SOC is calculated and the time profiles of the main quantities are converted into monodimensional structures and consequently packed into a unique structure called 'prof'.

```
function [psi, prof] = SOCvariation(EqFactor,SOC0,mission,veh)
    SOC(1) = SOC0; % [p.u.]
    time = mission.time_s; % [s]
    vehSpd = mission.speed_kmh./3.6; % [m/s]
    vehAcc = (vehSpd(2:end)-vehSpd(1:end-1))./veh.dt; % [m/s^2]
    % First component assumed 0 added to the acceleration vector
    vehAcc = [0;vehAcc]; % [m/s^2]
    for n = 1:length(time)
        [GN(n), PowerSplit(n)] = ecmsControl(SOC(n), EqFactor, vehSpd(n),
vehAcc(n), veh);
        [SOC(n+1), stageCost(n), unfeas(n), engPrf(n), emPrf(n), battPrf(n),
vehPrf(n)] = hev_model(SOC(n), [GN(n), PowerSplit(n)], [vehSpd(n), vehAcc(n)], veh);
    end
    finalSOC = SOC(end); % [p.u.]
    psi = finalSOC-SOC0;
```

```
    % Transform the non-scalar structure, containing time profiles, into scalar
structures; this makes their manipulation easier.
    engPrf = structArray2struct(engPrf);
    emPrf = structArray2struct(emPrf);
    battPrf = structArray2struct(battPrf);
    vehPrf = structArray2struct(vehPrf);
    % Pack profiles into a single structure
    prof.engPrf = engPrf;
    prof.emPrf = emPrf;
    prof.battPrf = battPrf;
    prof.vehPrf = vehPrf;
    prof.unfeas = unfeas;
end
```

Assignment #3: a-ECMS

Table of Contents

Group information

Group number: 49

Students:

- Carlo Vittorio Colucci, s329703
- Riccardo Bressani, s323665
- Luca Marchetto, s323437

Project introduction

The aim of the project is to design a controller for the evaluation of the torque-split factor and engaged gear of a p2 parallel HEV (represented in Fig.1) at each time instant of given driving profiles.

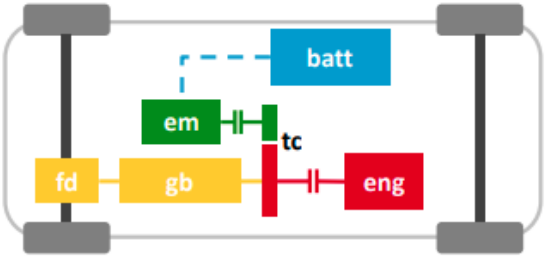


Fig.1: Scheme of a P2 parallel HEV

The control variable engine torque-split factor is defined accordingly:

$$\alpha_{eng} = \frac{T_{eng}}{T_{dem}}$$

As for Project 2, the controller follows an equivalent consumption minimization strategy (ECMS) for the choice of the torque-split factor (α_{eng}) and gear (γ), with the aim to minimize the following cost function:

$$\dot{m}_{f,eq} = \dot{m}_f(\gamma, \alpha_{eng}) - s \cdot \frac{E_b}{Q_{LHV}} \cdot \dot{\sigma}(\gamma, \alpha_{eng})$$

Equivalence factor

The meaning of this quantity is the same as for the previous project, representing the instantaneous equivalent fuel consumption. Although, as highlighted in the results analysis of the ECMS controller already implemented, the use of a constant equivalence factor 's' has some drawbacks:

- The choice of this parameter is dependent on a particular driving mission, which in general isn't known in advance;
- The adoption of a constant factor value on a cycle which is different from the one used in calibration phase can lead to unpredictable system behaviour (e.g. SOC behaviour different from the charge sustaining one, desired for a full HEV);

To counteract these problems, an adaptive ECMS is designed in this project.

This approach consists in using a variable equivalence factor, updated periodically to modify the weight given to the electric energy usage according to the instantaneous battery SOC. The law that regulates 's' is the following:

$$s_{n+1} = \frac{s_n + s_{n-1}}{2} + k_p \cdot (\sigma_0 - \sigma(t))$$

where s_{N+1}, s_N, s_{N-1} are the values of equivalence factor respectively used in the subsequent iteration, the actual one and the one adopted before the last update, k_p is a proportional factor tuned in the simulation and $\sigma_0 - \sigma(t)$ is the difference between the target SOC and the actual one. This calculation is not performed at every time instant, but only every time a fixed period elapses. In the controller, two logics are implemented: in the first one this period is a constant time interval (30 s), while in the second one it is a travelled distance (1 km).

The a-ECMS is tested over 3 different cycles belonging to Artemis standard, representing the typical driving conditions of urban, rural and motorway environments. The goal of the calibration of k_p parameter is to provide a final SOC value within a given interval ($0.5 < \sigma_f < 0.7$) for the three considered missions, assuming the starting SOC equal to 0.6 for each of them.

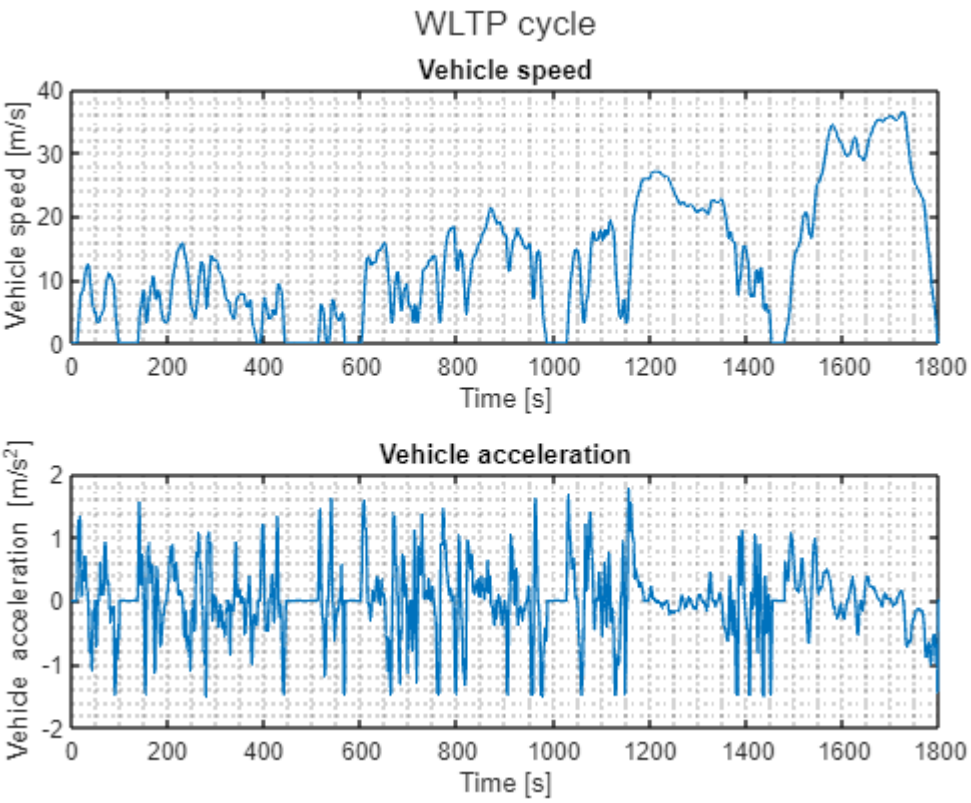
Loading of the cycles and vehicle data

The reference cycles which are used in this project are loaded in the following section: they consist in vectors representing the speed profiles with respect to time. The data concerning the cycles are contained in .mat files stored in 'data' folder. The considered cycles are the WLTP (used for preliminary testing) and the Artemis, which is composed of AUDC, ARDC and AMDC.

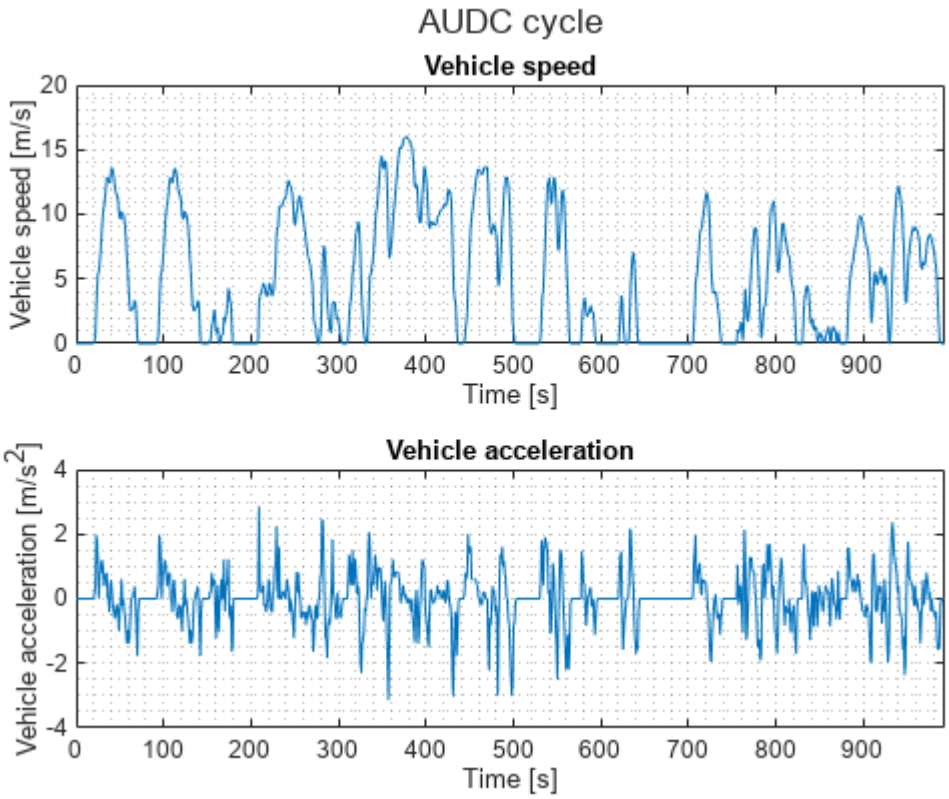
After the loading of the data, the variable 'cycleName' is created, for representation purposes and for the identification of the cycle in case of unfeasibilities detection.

For the sake of code tideness, the calculation of the acceleration vectors, as well as the representation of the profiles, are performed by function 'missionProfiles'; the acceleration vector is included in the structure relative to the cycle returned by the function.

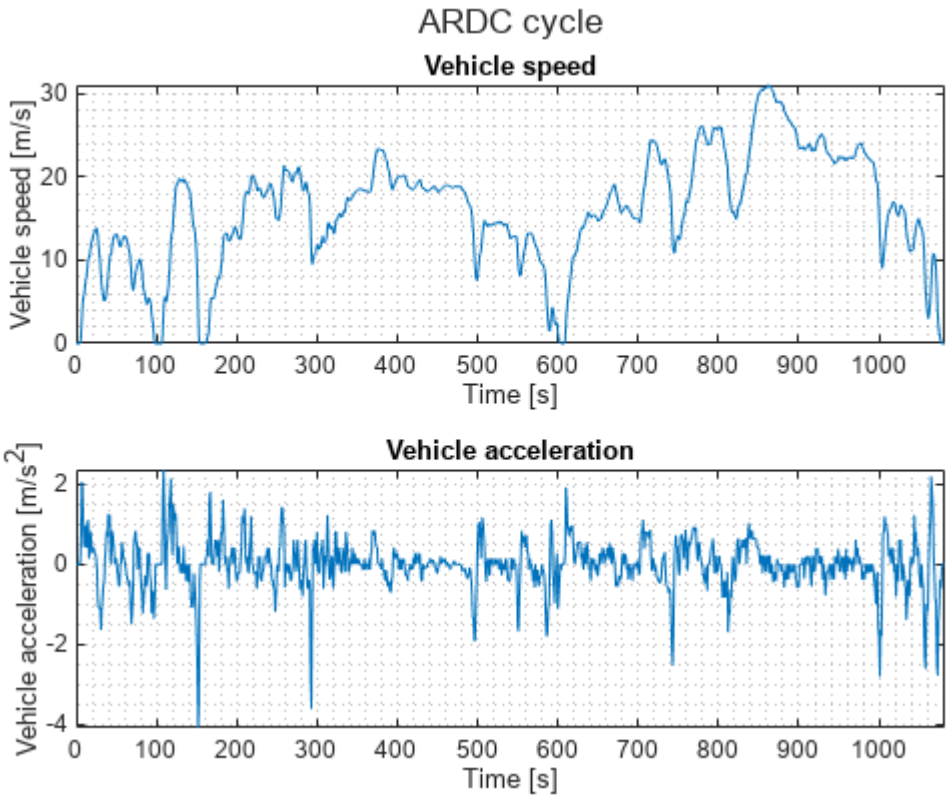
```
clear all
clc

% Addition of folders
addpath('data')
addpath('models')
addpath('utilities')

% Extraction of cycle velocities and accelerations
% WLTP
mission_WLTP = load('data\WLTP3.mat');
mission_WLTP.cycleName = 'WLTP cycle';
mission_WLTP = missionProfiles(mission_WLTP);
```



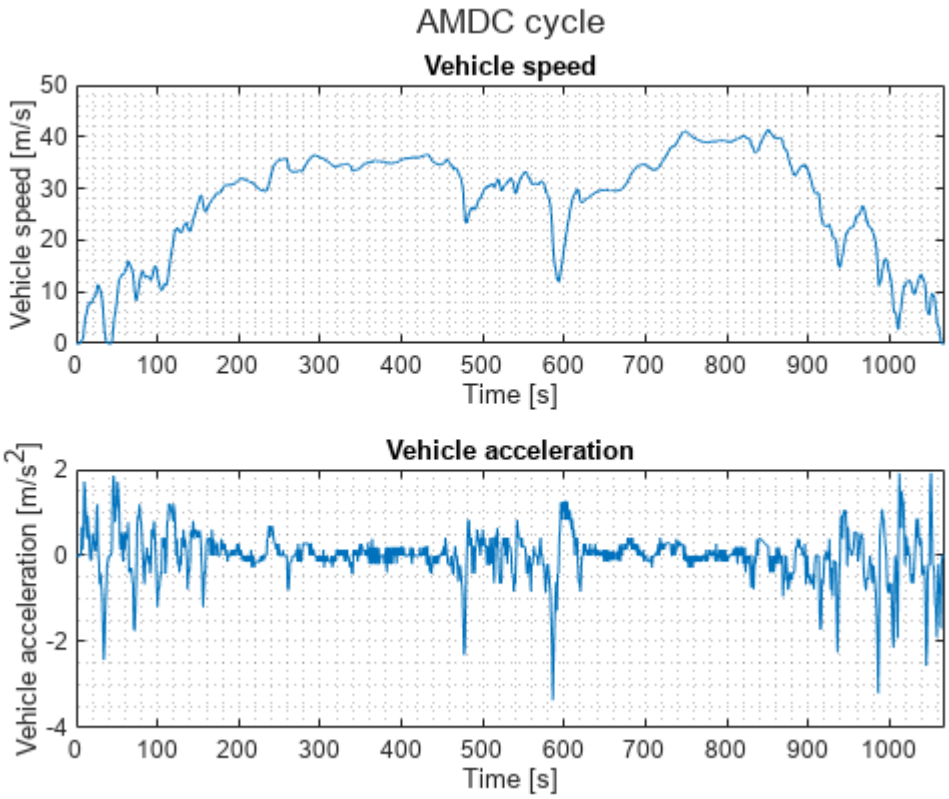

```
% Artemis cycles
% AUDC
mission_AUDC = load('data\AUDC.mat');
mission_AUDC.cycleName = 'AUDC cycle';
mission_AUDC = missionProfiles(mission_AUDC);
```




```
% ARDC
mission_ARDC = load('data\ARDC.mat');
mission_ARDC.cycleName = 'ARDC cycle';
mission_ARDC = missionProfiles(mission_ARDC);
```




```
% AMDC
mission_AMDC = load('data\AMDC.mat');
mission_AMDC.cycleName = 'AMDC cycle';
mission_AMDC = missionProfiles(mission_AMDC);
```



By using the following commands, the data related to the vehicle are loaded from the file 'vehData.mat'. These data are subsequently rescaled using the function 'scaleVehData.m' considering as additional inputs the values of ICE power [W], EM power [W], battery capacity [Ah] associated to the group number:

- **Group number:** 49
- **ICE power:** 60000 W
- **EM power:** 18000 W
- **Battery capacity:** 6.4 Ah

```
% Data loading and scaling
veh = load('data\vehData.mat');
engPwr = 60000; % [W]
emPwr = 18000; % [W]
battCap = 6.4; % [Ah]
veh = scaleVehData(veh,engPwr,emPwr,battCap); % scaleVehData(veh, engPwr[W],
emPwr[W], battCap[Ah])
```

Test of the a-ECMS on the WLTP cycle

After the developement of the adaptive ECMS controller, a simulation is performed on the WLTP mission to check the functionality of the logic; since the parameter k_p hasn't been calibrated yet, an initial value of 1 is assumed to perform the simulation. Moreover, the strategy adopted by the controller updates the equivence factor on a time basis, with a period of 30s.

The cycle simulation is performed in the function SOCvariation, mostly equal to the one already used in Project 2; the difference consists in the addition of a counter which is incremented at every iteration and provided to the adaptive ECMS controller: if the counter reaches the threshold value, a new equivalence factor is computed by 'adaptiveEcmsControl' and the counter is reset to 0. The update strategy can work both exploiting a time based or distance based threshold; this is selected changing the variable 'updateMode', defining it respectively 'time' or 'distance'.

Exploiting the equivalence factor returned by the previous iteration, 'adaptiveEcmsControl' performs the choice of the torque-split factor (α_{eng}) and gear (γ) with the same local minimization logic of the non adaptive one; in addition, if the counter reaches the target value, calculates the new value of 's' for the subsequent iterations. For such calculations, the initial value assumed for 's' is the one retrieved in Project 2, obtained by the bisection algorithm, calibrated on the WLTP cycle.

Then, by the knowledge of these obtained control variables and the mission, the vehicle longitudinal dynamic is solved using the already explained function 'hev_dp_model', retrieving the instantaneous evolution of battery variables and fuel consumption. As usual, all the main quantities are returned by 'SOCvariation' in a unique structure containing their time profiles, as well as the vectors of unfeasibilities. The scalar value of 'psi_WLTP' allows to indentify the SOC variation of the entire cycle.

The function 'unfeasCheck' is used to check that all the operating points selected by the controller are feasible according to the machines. Otherwise, an error message would appear, specifying the current mission, the time instant and the type of fault that occurred.

To conclude, the function 'energeticCalculations' provides useful parameters, such as cumulative fuel consumption [kg], fuel economy [L/100km] and total energy consumption [kWh], to grade the global behaviour of the controller.

Eventually, the final cycle SOC is calculated.

```
kp = 1; % [-]
SOC0 = 0.6; % [p.u.]
updateMode = 'time';
[psi_WLTP, prof_WLTP] = SOCvariation(SOC0, mission_WLTP, kp, veh, updateMode);
unfeasCheck(prof_WLTP, mission_WLTP)
[fuelConsumption_WLTP, fuelEconomy_WLTP, totalEnergyConsumption_WLTP] =
energeticCalculations(mission_WLTP, prof_WLTP, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_WLTP = 0.7613
fuelEconomy_WLTP = 4.1690
totalEnergyConsumption_WLTP = 9.1615
```

```
finalSOC_WLTP = SOC0+psi_WLTP % [p.u.]
```

```
finalSOC_WLTP = 0.6079
```

Test of the a-ECMS on the Artemis cycles and kp tuning

The main part of the project is focused on the tuning of the k_p value for the Artemis cycles. The requirement imposed for the choice of this parameter is to obtain a final SOC within 0.5 and 0.7, considering an initial SOC equal to 0.6, for each of the three cycles of the Artemis standard.

The simulations are performed on the following three missions:

- five consecutive repetitions of the Artemis Urban cycle;
- two consecutive repetitions of the Artemis Rural cycle;
- one Artemis Motorway cycle.

The cycles repetition is intended to give similar weight to each mission typology (urban, rural and motorway) in terms of travelled distance, according to the different average speeds. The following part of the code is just meant to create the vectors of the repeated cycles, starting from the individual ones:

```
% AUDC repeated 5 times
time_AUDC_5x = mission_AUDC.time_s; % [s]
speed_AUDC_5x = [];
vehAcc_AUDC_5x = [];
for i = 1:5
    if i<5
        time_AUDC_5x = [time_AUDC_5x; mission_AUDC.time_s + time_AUDC_5x(end)]; %
[s]
    end
    speed_AUDC_5x = [speed_AUDC_5x; mission_AUDC.speed_kmh]; % [km/h]
    vehAcc_AUDC_5x = [vehAcc_AUDC_5x; mission_AUDC.vehAcc]; % [m/s^2]
end
mission_AUDC_5x.time_s = time_AUDC_5x; % [s]
mission_AUDC_5x.speed_kmh = speed_AUDC_5x;  % [km/h]
mission_AUDC_5x.vehAcc = vehAcc_AUDC_5x; % [m/s^2]
mission_AUDC_5x.cycleName = 'AUDC cycle repeated 5 times';

% ARDC repeated twice
time_ARDC_2x = [mission_ARDC.time_s; mission_ARDC.time_s +
mission_ARDC.time_s(end)]; % [s]
speed_ARDC_2x = [mission_ARDC.speed_kmh; mission_ARDC.speed_kmh]; % [km/h]
vehAcc_ARDC_2x = [mission_ARDC.vehAcc; mission_ARDC.vehAcc]; % [m/s^2]
mission_ARDC_2x.time_s = time_ARDC_2x; % [s]
mission_ARDC_2x.speed_kmh = speed_ARDC_2x; % [km/h]
mission_ARDC_2x.vehAcc = vehAcc_ARDC_2x; % [m/s^2]
mission_ARDC_2x.cycleName = 'ARDC cycle repeated 2 times';
```

In the following step the proportional gain is tuned using the function 'tuneKp'; this simulates all the cycles used for the calibration adopting different values for this parameter; the values can be provided arbitrarily within the input vector 'k_p_values'; among the values that satisfy the already mentioned requirement ($0.5 < \sigma_f < 0.7$ for all the cycles), the one that provides the minimum sum of SOC variation ($|\sigma_f - \sigma_0|$), obtained in every mission, is selected; it is interesting to observe that this minimization strategy is only one of the possible choices: once the logic is implemented, different objective functions can be developed, such as fuel or energy consumption, as well as a combination of them.

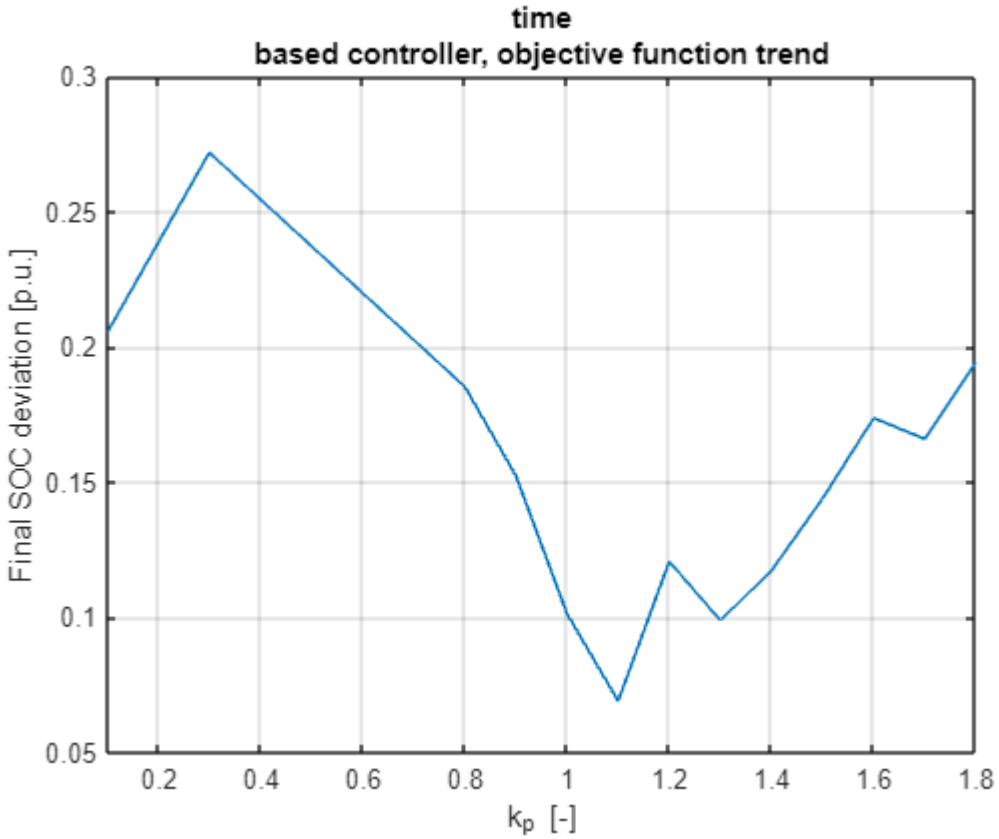
For instance, a function that takes into account both the SOC variation and energy consumption minimization can be of the form:

$$J = \alpha \cdot \sum_{i=1}^{n_{cycles}} |\Psi_i| + (1-\alpha) \cdot \delta \cdot \sum_{i=1}^{n_{cycles}} (E_{ICE} + E_{EM})_i$$

where $\Psi_i = \sigma_f - \sigma_0$, α is used to change the weight given to each cost variable and δ is a proper coefficient to make the two terms magnitude comparable. This strategy is not practically implemented in the function because the tuning of the additional parameters would require a much more complex results analysis, which is out of the interest of this project.

In addition, the function provides a figure representing the trend of the objective function against the different tested values of k_p which satisfy the above mentioned requirement.

```
k_p_values = 0:0.1:2;
tuned_Kp_time = tuneKp(k_p_values, SOC0, mission_WLTP, mission_AUDC_5x,
mission_ARDC_2x, mission_AMDC, veh, updateMode)
```




```
tuned_Kp_time = 1.1000
```

Cycles simulation with tuned kp

In the following section the Artemis cycles (urban, rural and motorway) are simulated with the obtained value of calibrated k_p, using the time based adaptive ECMS logic. As mentioned, the simulation results are provided in structures as output of the function 'SOCvariation'; from the results of 'energeticCalculations' it is possible to observe also the fuel consumption [kg], the fuel economy [L/100km] and the total energy consumption [kWh]. In addition, the calculations are carried out on the WLTP cycle as well, to store the data to allow a comparison with the controllers previously implemented in projects 1 and 2. It is observed that, with the chosen value of k_p, also

the results of the WLTP cycle are compliant with the final SOC requirement: this is a consequence of having considered this mission, as well, in the logic of 'tune_k_p'. This choice comes from the fact that, simulating the WLTP with a value of k_p tuned only on the Artemis cycle, gave a final SOC out of the range [0.5,0.7] for this mission.

```
% WLTP (for controllers comparison)
[psi_WLTP_time, prof_WLTP_time] = SOCvariation(SOC0, mission_WLTP, tuned_Kp_time,
veh, updateMode);
unfeasCheck(prof_WLTP_time, mission_WLTP) % Unfeasibility check
[fuelConsumption_WLTP_time, fuelEconomy_WLTP_time,
totalEnergyConsumption_WLTP_time] = energeticCalculations(mission_WLTP,
prof_WLTP_time, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_WLTP_time = 0.7580
fuelEconomy_WLTP_time = 4.1510
totalEnergyConsumption_WLTP_time = 9.1369
```

```
finalSOC_WLTP_time = SOC0+psi_WLTP_time % [p.u.]
```

```
finalSOC_WLTP_time = 0.5965
```

```
% AUDC 5x
[psi_AUDC_time, prof_AUDC_time] = SOCvariation(SOC0, mission_AUDC_5x,
tuned_Kp_time, veh, updateMode);
unfeasCheck(prof_AUDC_time, mission_AUDC_5x) % Unfeasibility check
[fuelConsumption_AUDC_time, fuelEconomy_AUDC_time,
totalEnergyConsumption_AUDC_time] = energeticCalculations(mission_AUDC_5x,
prof_AUDC_time, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_AUDC_time = 0.6864
fuelEconomy_AUDC_time = 3.5908
totalEnergyConsumption_AUDC_time = 8.2844
```

```
finalSOC_AUDC_time = SOC0+psi_AUDC_time % [p.u.]
```

```
finalSOC_AUDC_time = 0.5865
```

```
% ARDC 2x
[psi_ARDC_time, prof_ARDC_time] = SOCvariation(SOC0,mission_ARDC_2x, tuned_Kp_time,
veh, updateMode);
unfeasCheck(prof_ARDC_time, mission_ARDC_2x) % Unfeasibility check
[fuelConsumption_ARDC_time, fuelEconomy_ARDC_time,
totalEnergyConsumption_ARDC_time] = energeticCalculations(mission_ARDC_2x,
prof_ARDC_time, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_ARDC_time = 0.9992
fuelEconomy_ARDC_time = 3.6846
totalEnergyConsumption_ARDC_time = 12.0192
```

```
finalSOC_ARDC_time = SOC0+psi_ARDC_time % [p.u.]
```

```
finalSOC_ARDC_time = 0.6127
```

```
% AMDC 1x
```

```
[psi_AMDC_time, prof_AMDC_time] = SOCvariation(SOC0,mission_AMDC, tuned_Kp_time,
veh, updateMode);
unfeasCheck(prof_AMDC_time, mission_AMDC) % Unfeasibility check
[fuelConsumption_AMDC_time, fuelEconomy_AMDC_time,
totalEnergyConsumption_AMDC_time] = energeticCalculations(mission_AMDC,
prof_AMDC_time, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_AMDC_time = 1.4690
fuelEconomy_AMDC_time = 6.3351
totalEnergyConsumption_AMDC_time = 17.6522
```

```
finalSOC_AMDC_time = SOC0+psi_AMDC_time % [p.u.]
```

```
finalSOC_AMDC_time = 0.6394
```

To analyse the average fuel consumption for the entire Artemis cycle, made up of urban, rural and motorway, the total distance travelled is computed and the combined fuel and energy consumption are found. These data summarize the behaviour of the system and can be easily used to compare different controllers on the same mission.

```
% Average fuel Economy Artemis
vehDist_Artemis = trapz(mission_AUDC_5x.time_s, mission_AUDC_5x.speed_kmh/
3.6) + trapz(mission_ARDC_2x.time_s, mission_ARDC_2x.speed_kmh/3.6) +
trapz(mission_AMDC.time_s, mission_AMDC.speed_kmh/3.6); % [m]
fuelConsumption_Artemis_time = fuelConsumption_AUDC_time +
fuelConsumption_ARDC_time + fuelConsumption_AMDC_time; % [kg]
fuelEconomy_Artemis_time = (fuelConsumption_Artemis_time*10^5) /
(veh.eng.fuelDensity*vehDist_Artemis) % [L/100km]
```

```
fuelEconomy_Artemis_time = 4.5441
```

```
totalEnergyConsumption_Artemis_time = totalEnergyConsumption_AUDC_time +
totalEnergyConsumption_ARDC_time + totalEnergyConsumption_AMDC_time % [kWh]
```

```
totalEnergyConsumption_Artemis_time = 37.9558
```

Time based controller analysis

To carry out a proper analysis of the results obtained using the time based controller, the thermal and electrical machines maps are reported, as well as the power flow trends and the control variables gear number γ and torque split factor α_{eng}.

ICE operating points

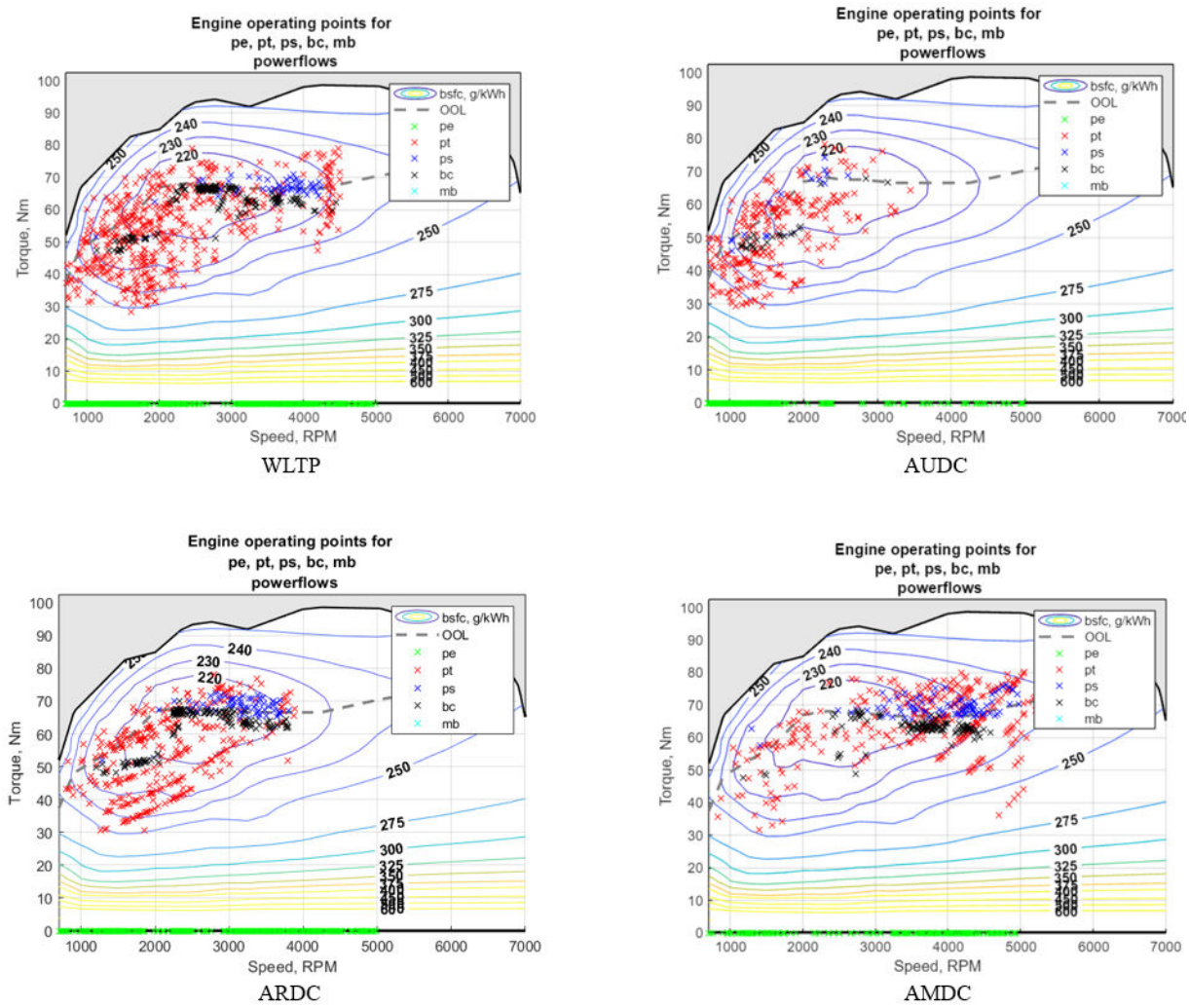


Fig.2: ICE maps with operating points for the 4 different driving missions with time based strategy

The adopted controller logic for the a-ECMS is almost the same of the ECMS, apart from the equivalent factor periodical updating, thus the working points positioning is expected to be similar to the previous project controller for what concerns the WLTP mission. The main difference is related to the particular cycle, driving the working points in different maps regions according to the torque and speed requirements, and influencing the power split choices resulting in different operating mode adoptions. For instance the AMDC and the ARDC attend a much higher content of battery charging mode, with respect to the AUDC, because of the medium/ high torque request leading to battery recharging by means of the ICE. As highlited in the previous project, the working points are located in high efficiency zone of the maps, as expected by the adopted minimization strategy.

EM operating points

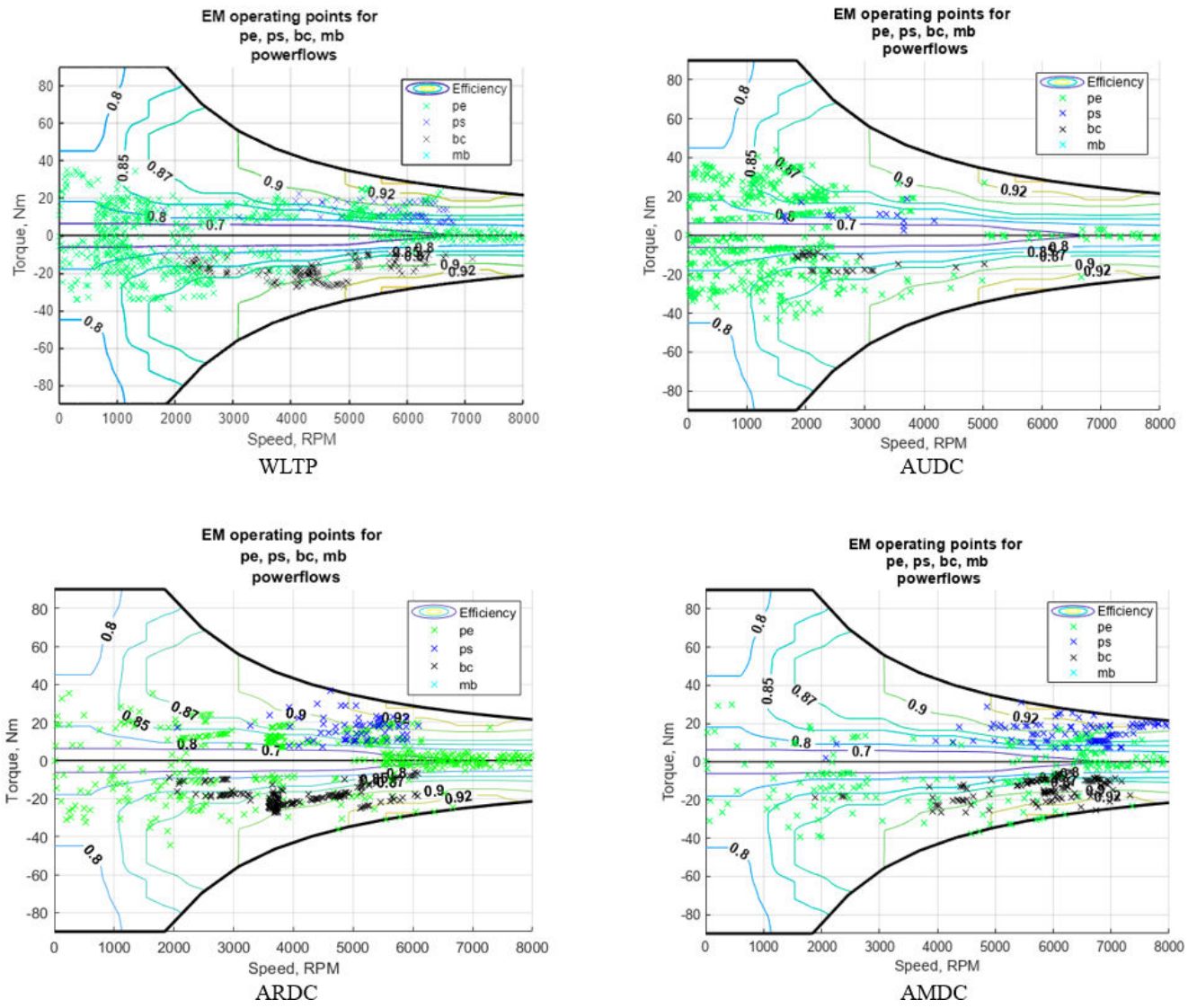


Fig.3: EM maps with operating points for the 4 different driving missions with time based strategy

As previoulsy said for the ICE maps, the working points are differently spread on the maps, depending on the particular driving mission. With respect to ICE maps, the operating points are in this case less concentrated, since the efficiency variation throughout the map of the EM is less significant with respect to the ICE one; thus, the global optimization strategy of the controller leads to widely spread EM operating points.

Power flow trends

The Artemis cycle power flow graphs are reported here: for the AUDC and the ARDC only one repetition of the cycles is plotted for the analysis, to better highlight the working modes zones, while for AMDC the complete mission is reported.

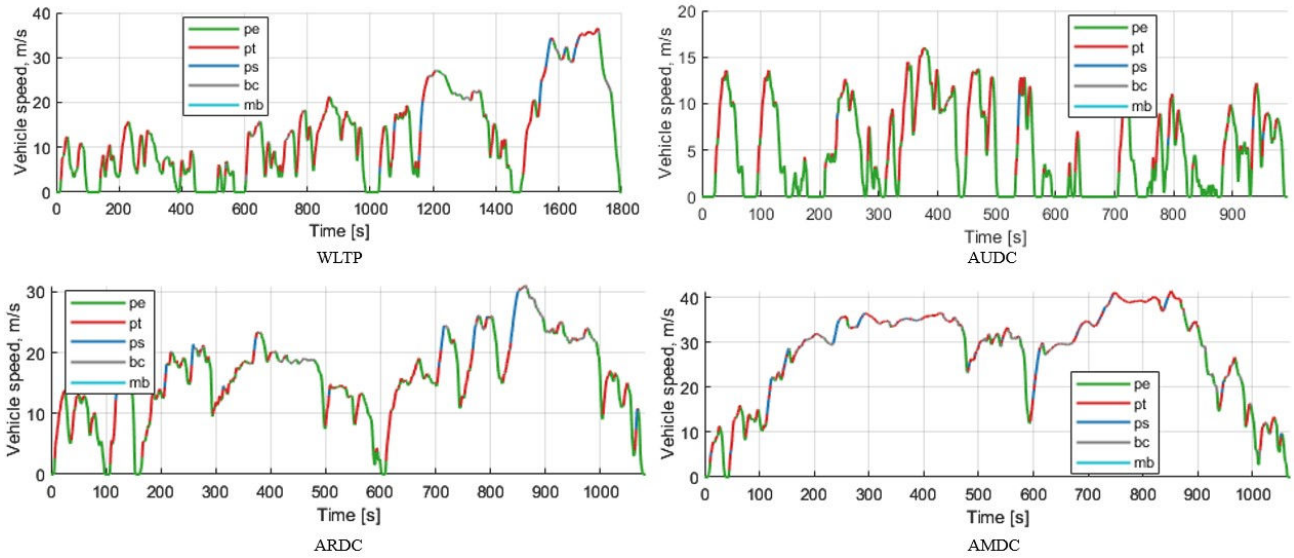


Fig.4: power flow trends for the 4 different driving missions with time based strategy

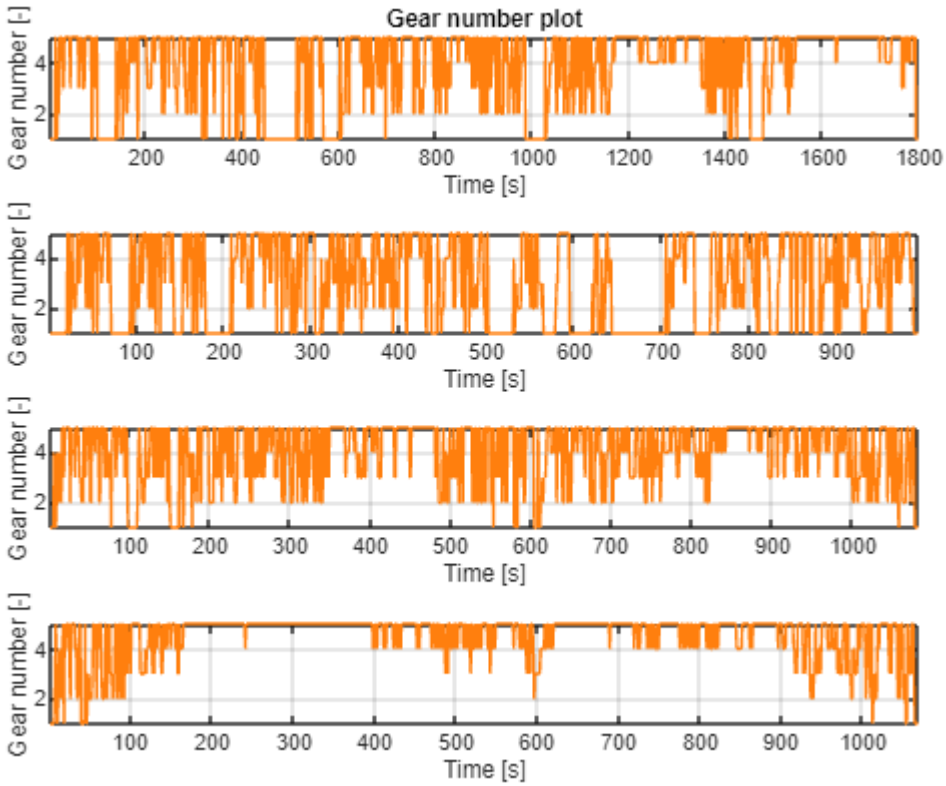
It can be highlighted how the urban and rural missions are performed exploiting more the pure electric mode, with respect to the motorway one, also considering a higher number of braking phases. This results in a more relevant battery charging in cycles with a high average speed, for example the motorway and some portions of the rural one.

Gear number

The time profiles of the gear numbers for the WLTP, AUDC, ARDC and AMDC cycles (from top to bottom) are reported to inspect the behaviour of the controller.

```
% Plots of the gear numbers
figure
tiledlayout(4,1)
% The 4 figures represent WLTP, AUDC, ARDC and AMDC
nexttile(1)
plot(mission_WLTP.time_s, prof_WLTP_time.vehPrf.gearNumber, 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
xlim([1 length(mission_WLTP.time_s)])
ylabel("Gear number [-]")
ylim([1 5])
title('Gear number plot')
nexttile(2)
plot(mission_AUDC.time_s,
prof_AUDC_time.vehPrf.gearNumber(1:length(mission_AUDC.time_s)), 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
xlim([1 length(mission_AUDC.time_s)])
ylabel("Gear number [-]")
ylim([1 5])
nexttile(3)
```

```
plot(mission_ARDC.time_s,
prof_ARDC_time.vehPrf.gearNumber(1:length(mission_ARDC.time_s)), 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
xlim([1 length(mission_ARDC.time_s)])
ylabel("Gear number [-]")
ylim([1 5])
nexttile(4)
plot(mission_AMDC.time_s, prof_AMDC_time.vehPrf.gearNumber, 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
xlim([1 length(mission_AMDC.time_s)])
ylabel("Gear number [-]")
ylim([1 5])
```



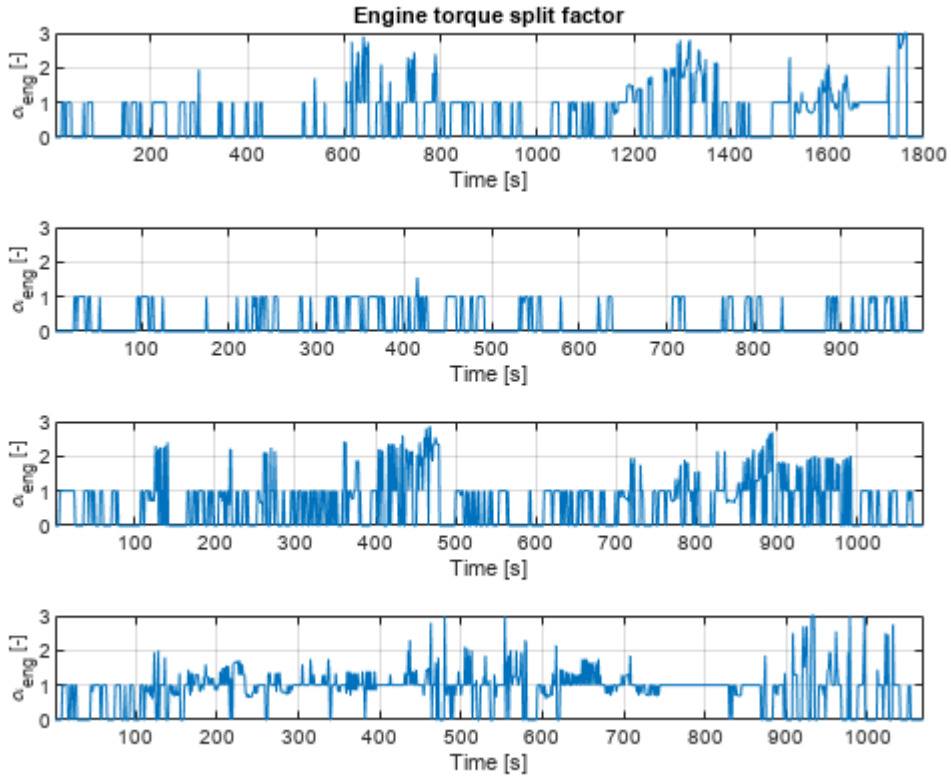
The behaviour of the gear shifting logic is similar to what observed for the non adaptive ECMS controller, showing frequent gear shifts also between non consecutive gears. The behaviour is more regular only in high speed regions (AMDC), where the longest gear is maintained for long periods.

Engine torque split factor

The behaviour of the control parameter α_{eng} for the WLTP, AUDC, ARDC and AMDC cycles (from top to bottom) is inspected below.

```
% Plots of the torque split factor
figure
tiledlayout(4,1)
```

```
% The 4 figures represent WLTP, AUDC, ARDC and AMDC
nexttile(1)
plot(mission_WLTP.time_s, prof_WLTP_time.vehPrf.engAlpha)
grid on
xlabel("Time [s]")
xlim([1 length(mission_WLTP.time_s)])
ylabel("\alpha_{eng} [-]")
ylim([0 3])
title('Engine torque split factor')
nexttile(2)
plot(mission_AUDC.time_s,
prof_AUDC_time.vehPrf.engAlpha(1:length(mission_AUDC.time_s)))
grid on
xlabel("Time [s]")
xlim([1 length(mission_AUDC.time_s)])
ylabel("\alpha_{eng} [-]")
ylim([0 3])
nexttile(3)
plot(mission_ARDC.time_s,
prof_ARDC_time.vehPrf.engAlpha(1:length(mission_ARDC.time_s)))
grid on
xlabel("Time [s]")
xlim([1 length(mission_ARDC.time_s)])
ylabel("\alpha_{eng} [-]")
ylim([0 3])
nexttile(4)
plot(mission_AMDC.time_s, prof_AMDC_time.vehPrf.engAlpha)
grid on
xlabel("Time [s]")
xlim([1 length(mission_AMDC.time_s)])
ylabel("\alpha_{eng} [-]")
ylim([0 3])
```

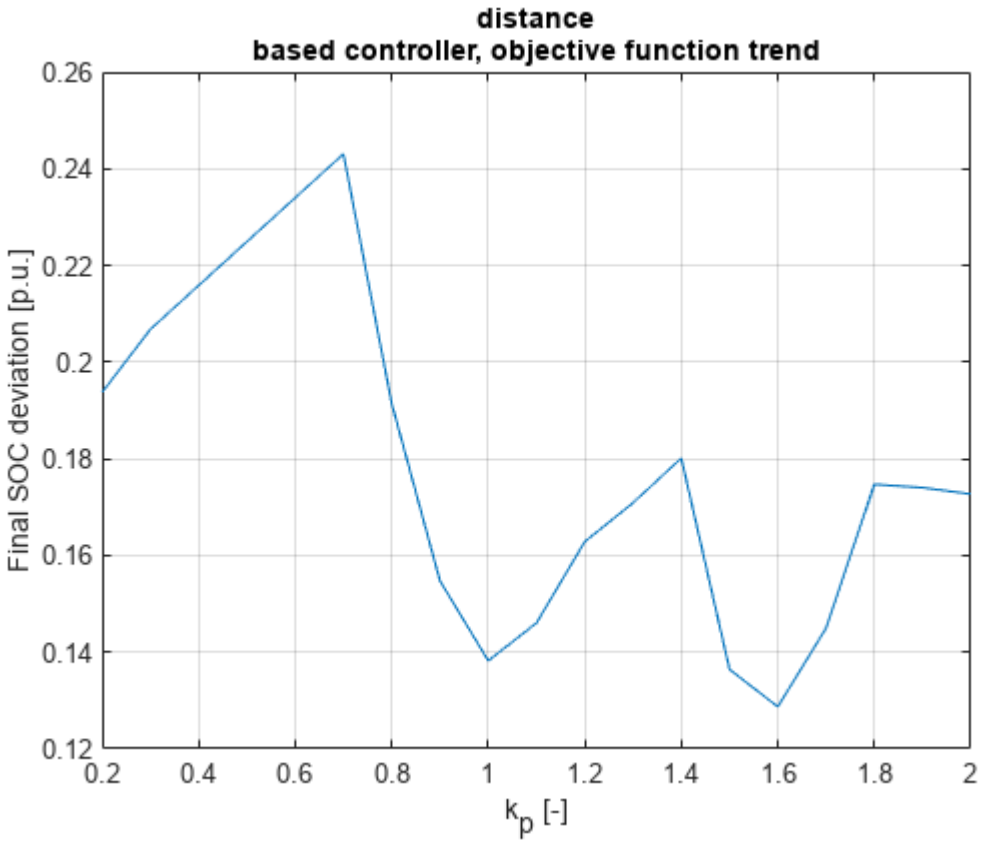


Also for the engine torque split factor, no significant difference of behaviour is observed with respect to the non adaptive ECMS controller. At low and medium speeds the predominant modes are pure electic and power split, while in high speed regions pure thermal and battery charging are mostly adopted.

Test of the distance-based a-ECMS on the Artemis cycles

As for the time based controller, a value for the proportional gain k_p is chosen by the 'tuneKp' function, considering the different update strategy for the equivalence factor, according to the value of updateMode. In particular, for the distance based controller, the threshold is updated every 1 km instead of every 30 seconds. An advantage of tuning k_p by means of a function is that it is possible to retrieve automatically the calibrated value even changing the controller update logic.

```
updateMode = 'distance';
tuned_Kp_dist = tuneKp(k_p_values, SOC0, mission_WLTP, mission_AUDC_5x,
mission_ARDC_2x, mission_AMDC, veh, updateMode)
```




```
tuned_Kp_dist = 1.6000
```

As done in the time based controller section, for each mission the 'SOCvariation' function retrieves the final SOC deviation for the considered cycle and all the profiles relative to the main vehicle quantities. Then, a check on the unfeasible conditions is carried out and the main energetic quantities are computed by the 'energeticCalculations' function; the final SOC is finally calculated. Also for this controller the simulations are carried out on the three Artemis cycles, as well as the WLTP.

```
% WLTP (for controllers comparison)
[psi_WLTP_dist, prof_WLTP_dist] = SOCvariation(SOC0, mission_WLTP, tuned_Kp_dist,
veh, updateMode);
unfeasCheck(prof_WLTP_dist, mission_WLTP) % Unfeasibility check
[fuelConsumption_WLTP_dist, fuelEconomy_WLTP_dist,
totalEnergyConsumption_WLTP_dist] = energeticCalculations(mission_WLTP,
prof_WLTP_dist, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_WLTP_dist = 0.7656
fuelEconomy_WLTP_dist = 4.1925
totalEnergyConsumption_WLTP_dist = 9.1912
```

```
finalSOC_WLTP_dist = SOC0+psi_WLTP_dist % [p.u.]
```

```
finalSOC_WLTP_dist = 0.6248
```

```
% AUDC 5x
[psi_AUDC_dist, prof_AUDC_dist] = SOCvariation(SOC0,mission_AUDC_5x, tuned_Kp_dist,
veh, updateMode);
unfeasCheck(prof_AUDC_dist, mission_AUDC_5x) % Unfeasibility check
```

```
[fuelConsumption_AUDC_dist, fuelEconomy_AUDC_dist,
totalEnergyConsumption_AUDC_dist] = energeticCalculations(mission_AUDC_5x,
prof_AUDC_dist, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_AUDC_dist = 0.6909
fuelEconomy_AUDC_dist = 3.6148
totalEnergyConsumption_AUDC_dist = 8.3060
```

```
finalSOC_AUDC_dist = SOC0+psi_AUDC_dist % [p.u.]
```

```
finalSOC_AUDC_dist = 0.6123
```

```
% ARDC 2x
[psi_ARDC_dist, prof_ARDC_dist] = SOCvariation(SOC0,mission_ARDC_2x, tuned_Kp_dist,
veh, updateMode);
unfeasCheck(prof_ARDC_dist, mission_ARDC_2x) % Unfeasibility check
[fuelConsumption_ARDC_dist, fuelEconomy_ARDC_dist,
totalEnergyConsumption_ARDC_dist] = energeticCalculations(mission_ARDC_2x,
prof_ARDC_dist, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_ARDC_dist = 1.0016
fuelEconomy_ARDC_dist = 3.6934
totalEnergyConsumption_ARDC_dist = 12.0385
```

```
finalSOC_ARDC_dist = SOC0+psi_ARDC_dist % [p.u.]
```

```
finalSOC_ARDC_dist = 0.6202
```

```
% AMDC 1x
[psi_AMDC_dist, prof_AMDC_dist] = SOCvariation(SOC0,mission_AMDC, tuned_Kp_dist,
veh, updateMode);
unfeasCheck(prof_AMDC_dist, mission_AMDC) % Unfeasibility check
[fuelConsumption_AMDC_dist, fuelEconomy_AMDC_dist,
totalEnergyConsumption_AMDC_dist] = energeticCalculations(mission_AMDC,
prof_AMDC_dist, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_AMDC_dist = 1.4791
fuelEconomy_AMDC_dist = 6.3788
totalEnergyConsumption_AMDC_dist = 17.7322
```

```
finalSOC_AMDC_dist = SOC0+psi_AMDC_dist % [p.u.]
```

```
finalSOC_AMDC_dist = 0.6714
```

The average fuel consumption and the total energy used for the three Artemis cycles are again computed for the distance based controller:

```
% Average fuel Economy Artemis
fuelConsumption_Artemis_dist = fuelConsumption_AUDC_dist +
fuelConsumption_ARDC_dist + fuelConsumption_AMDC_dist; % [kg]
fuelEconomy_Artemis_dist = (fuelConsumption_Artemis_dist*10^5) /
(veh.eng.fuelDensity*vehDist_Artemis) % [L/100km]
```

```
fuelEconomy_Artemis_dist = 4.5688
```

```
totalEnergyConsumption_Artemis_dist = totalEnergyConsumption_AUDC_dist +
totalEnergyConsumption_ARDC_dist + totalEnergyConsumption_AMDC_dist % [kWh]
```

```
totalEnergyConsumption_Artemis_dist = 38.0767
```

Distance based controller analysis

The same quantities shown for the time based controller are now proposed for the distance based one. From the maps and the power flows trends it is difficult to appreciate significant differences with respect to the other control update strategy, since these are more related to the control parameters updating rate, rather than a different control logic. For this reason, the following figures are just reported for completeness, while a more detailed results comparison is undertaken in the following section 'Results comparison'.

ICE operating points

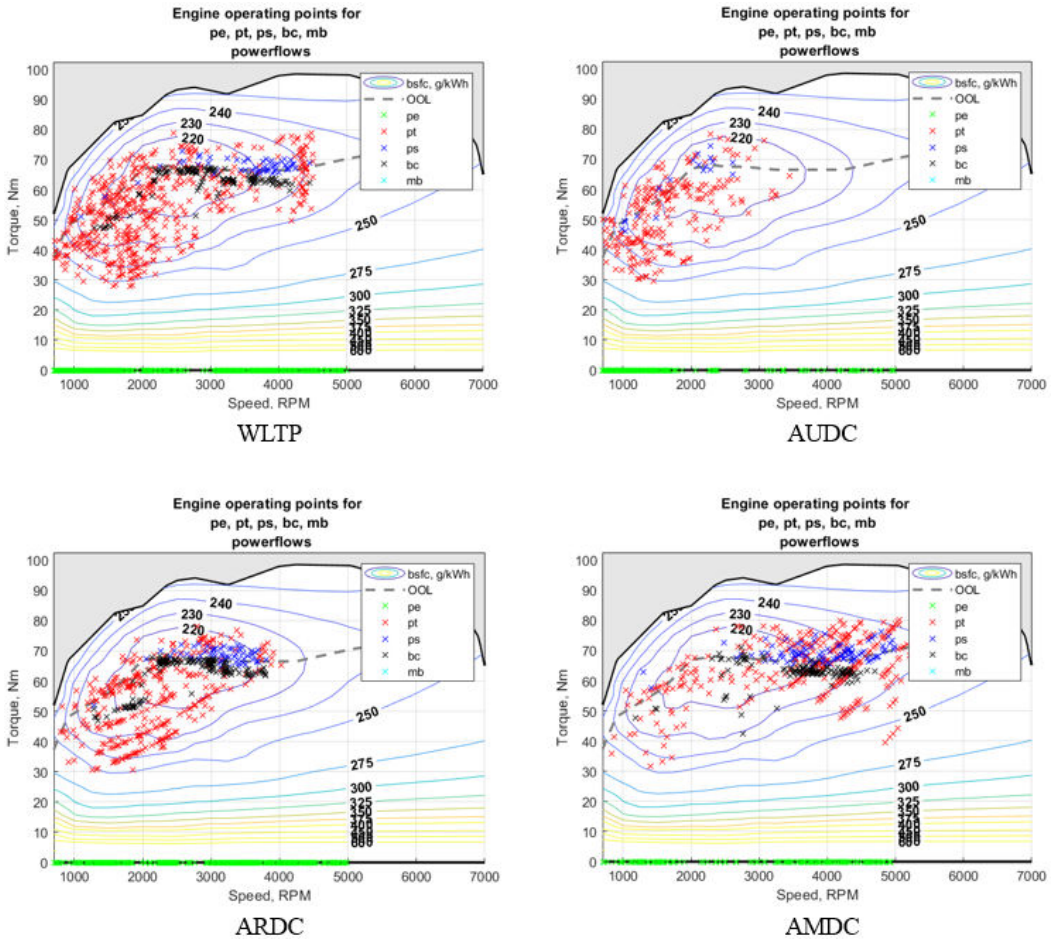


Fig.5: ICE maps with operating points for the 4 different driving missions with distance based strategy

EM operating points

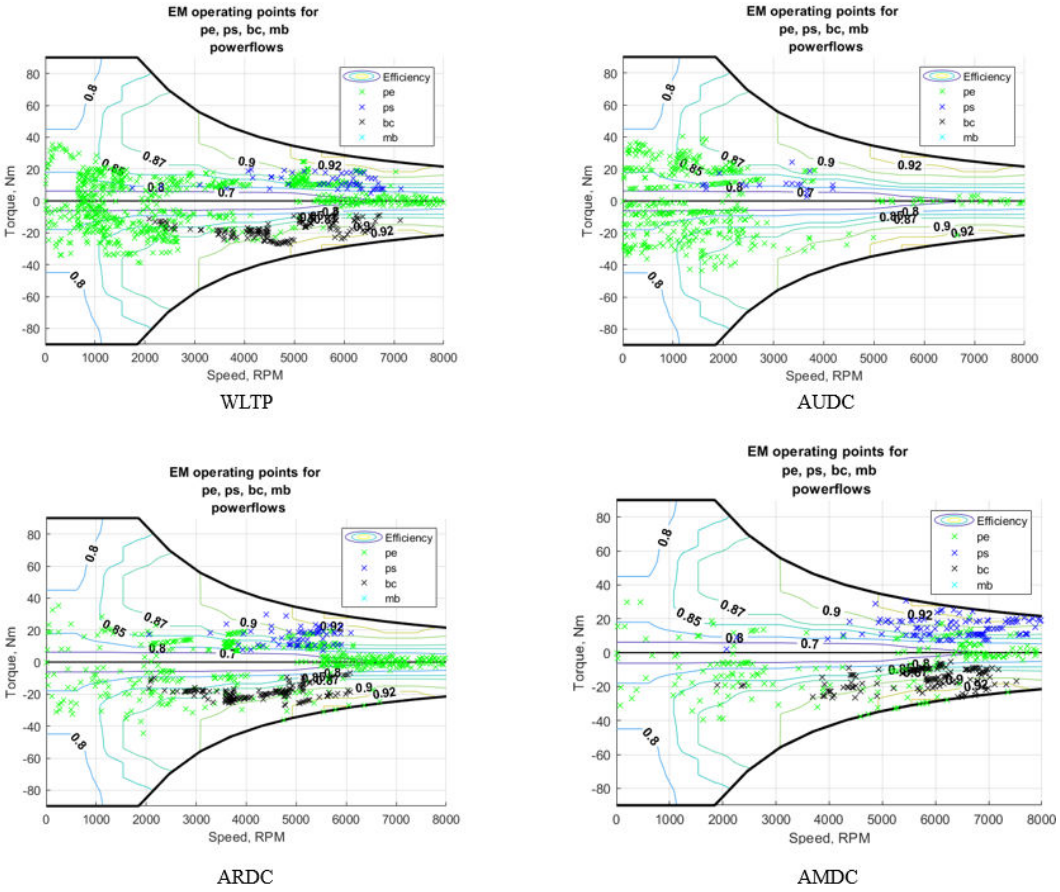


Fig.6: EM maps with operating points for the 4 different driving missions with distance based strategy

Power flow trends

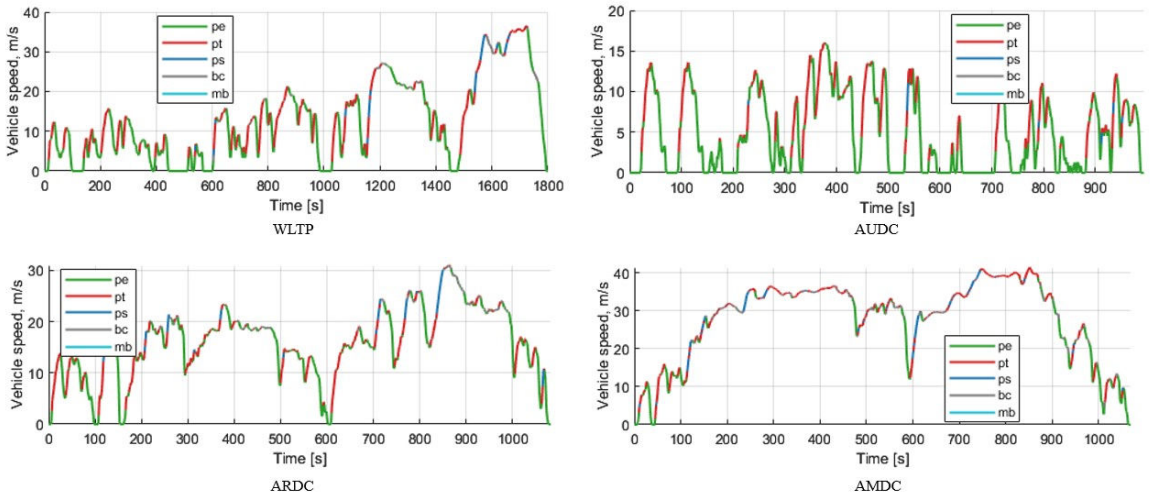
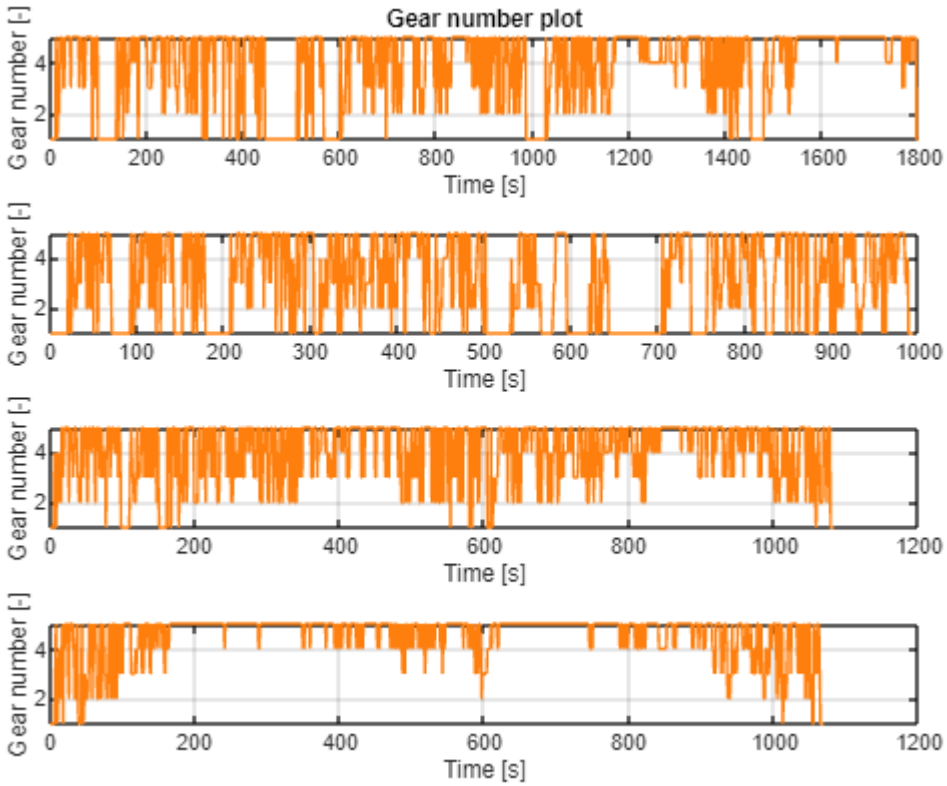


Fig.7: power flow trends for the 4 different driving missions with distance based strategy

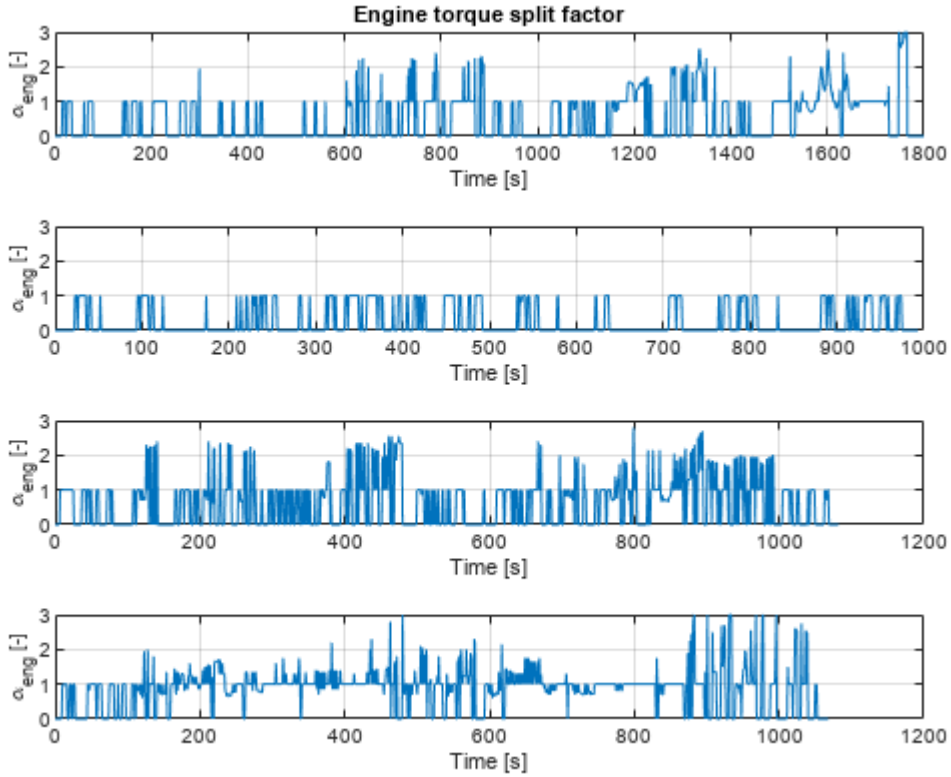
```
% Plots of the gear numbers
figure
tiledlayout(4,1)
% The 4 figures represent WLTP, AUDC, ARDC and AMDC
```

```
nexttile(1)
plot(mission_WLTP.time_s, prof_WLTP_dist.vehPrf.gearNumber, 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
ylabel("Gear number [-]")
ylim([1 5])
title('Gear number plot')
nexttile(2)
plot(mission_AUDC.time_s,
prof_AUDC_dist.vehPrf.gearNumber(1:length(mission_AUDC.time_s)), 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
ylabel("Gear number [-]")
ylim([1 5])
nexttile(3)
plot(mission_ARDC.time_s,
prof_ARDC_dist.vehPrf.gearNumber(1:length(mission_ARDC.time_s)), 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
ylabel("Gear number [-]")
ylim([1 5])
nexttile(4)
plot(mission_AMDC.time_s, prof_AMDC_dist.vehPrf.gearNumber, 'Color', '#ff7f0e')
grid on
xlabel("Time [s]")
ylabel("Gear number [-]")
ylim([1 5])
```




```
% Plots of the torque split factor
figure
tiledlayout(4,1)
% The 4 figures represent WLTP, AUDC, ARDC and AMDC
nexttile(1)
plot(mission_WLTP.time_s, prof_WLTP_dist.vehPrf.engAlpha)
grid on
xlabel("Time [s]")
ylabel("\alpha_{eng} [-]")
ylim([0 3])
title('Engine torque split factor')
nexttile(2)
plot(mission_AUDC.time_s,
prof_AUDC_dist.vehPrf.engAlpha(1:length(mission_AUDC.time_s)))
grid on
xlabel("Time [s]")
ylabel("\alpha_{eng} [-]")
ylim([0 3])
nexttile(3)
plot(mission_ARDC.time_s,
prof_ARDC_dist.vehPrf.engAlpha(1:length(mission_ARDC.time_s)))
grid on
xlabel("Time [s]")
ylabel("\alpha_{eng} [-]")
ylim([0 3])
nexttile(4)
```

```
plot(mission_AMDC.time_s, prof_AMDC_dist.vehPrf.engAlpha)
grid on
xlabel("Time [s]")
ylabel("\alpha_{eng} [-]")
ylim([0 3])
```



Save results

The following code saves the structure of the cycles time profiles, the fuel consumption (in kg), the fuel economy (in L/100km) and the final SOC in a .mat file for the three portions of the Artemis cycle. Two different files are created, one for the results of the time based controller and the other for the distance based one.

```
% Store results
save("results_adaptiveECMS_time.mat", "prof_WLTP_time",
"fuelConsumption_WLTP_time", "fuelEconomy_WLTP_time", "finalSOC_WLTP_time",
"prof_AUDC_time", "fuelConsumption_AUDC_time", "fuelEconomy_AUDC_time",
"finalSOC_AUDC_time", "prof_ARDC_time", "fuelConsumption_ARDC_time",
"fuelEconomy_ARDC_time", "finalSOC_ARDC_time", "prof_AMDC_time",
"fuelConsumption_AMDC_time", "fuelEconomy_AMDC_time", "finalSOC_AMDC_time");
save("results_adaptiveECMS_dist.mat", "prof_WLTP_dist",
"fuelConsumption_WLTP_dist", "fuelEconomy_WLTP_dist", "finalSOC_WLTP_dist",
"prof_AUDC_dist", "fuelConsumption_AUDC_dist", "fuelEconomy_AUDC_dist",
"finalSOC_AUDC_dist", "prof_ARDC_dist", "fuelConsumption_ARDC_dist",
"fuelEconomy_ARDC_dist", "finalSOC_ARDC_dist", "prof_AMDC_dist",
"fuelConsumption_AMDC_dist", "fuelEconomy_AMDC_dist", "finalSOC_AMDC_dist");
```

Results comparison

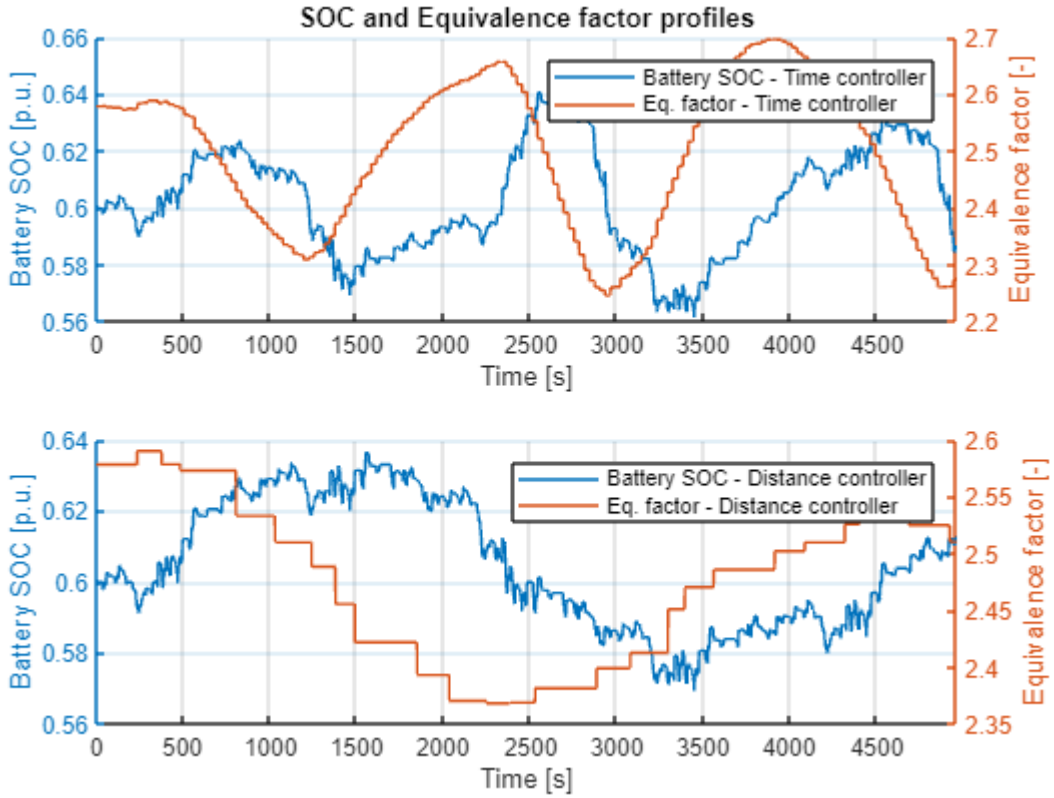
Time based vs Distance based adaptive ECMS

A comparison between the two controllers update logics can be useful to deeply understand and appreciate their differences. These are more evident in the observation of the behaviour of the SOC throughout the different cycles, which is strictly linked to the evolution of the equivalence factor. To observe the relation between these two variables, they are represented in the same figure, for different missions and controllers.

Battery SOC and equivalence factor

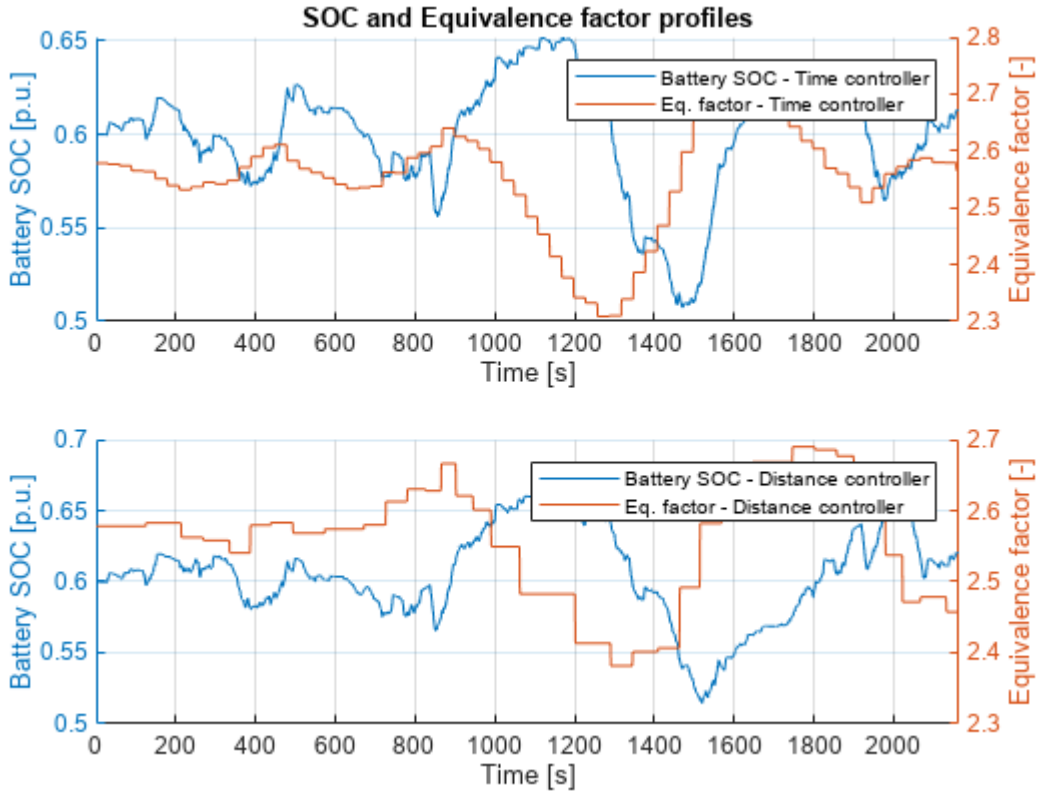
- AUDC cycle:

```
SOCvsS_profiles(mission_AUDC_5x, prof_AUDC_time, prof_AUDC_dist)
```



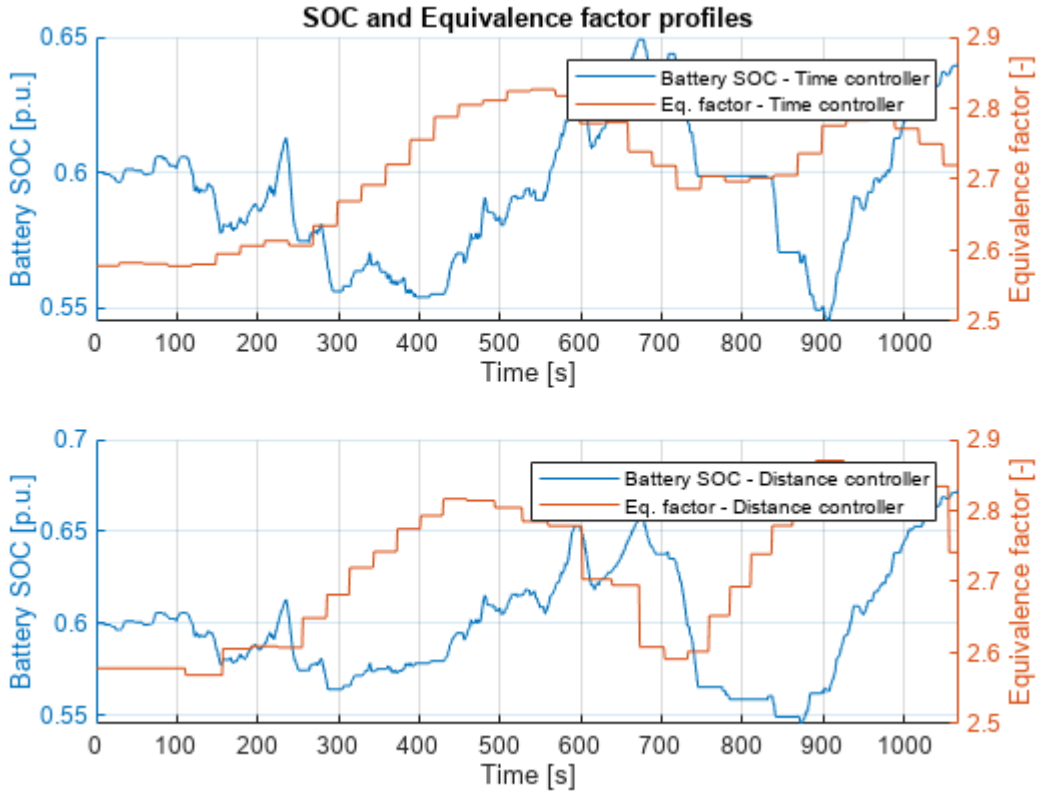
- ARDC cycle:

```
SOCvsS_profiles(mission_ARDC_2x, prof_ARDC_time, prof_ARDC_dist)
```



- AMDC cycle:

```
SOCvsS_profiles(mission_AMDC, prof_AMDC_time, prof_AMDC_dist)
```



As a general trend, it is observed how the behaviour of the equivalence factor is coherent with the controllers logic, since it reacts to the SOC evolution: when SOC overcomes 0.6 the equivalence factor starts decreasing, viceversa when SOC is lower, the equivalence factor increases; a certain delay is observed between the two signals, due to the dynamics of the equivalence factor imposed by the value of k_p, as well as the instantaneous vehicle power demand related to the particular mission. In general, the distance based controller shows a slower dynamic of SOC and equivalence factor evolution, related to the high selected update distance of 1 km; the behaviour is similar only in the motorway section of the cycle, in which the high average speed counteracts this effect.

To conclude the comparison, the values of fuel economy and total energy consumption of the controllers are reported: the energy takes into account not only the fuel, but also the electric energy used, related to the variation of SOC throughout the cycle. It is shown how the two logics do not provide significant differences for what concerns the consumption, with a slight advantage for the time based one.

	AUDC		ARDC		AMDC		Combined	
	Time	Distance	Time	Distance	Time	Distance	Time	Distance
Fuel consumption [L/100km]	3.5908	3.6148	3.6846	3.6934	6.3351	6.3788	4.5441	4.5688
Total energy consumption [kWh]	8.2844	8.3060	12.0192	12.0385	17.6522	17.7322	37.9558	38.0767

Tab.1: Fuel economy and energy consumption comparison, time vs distance based controllers

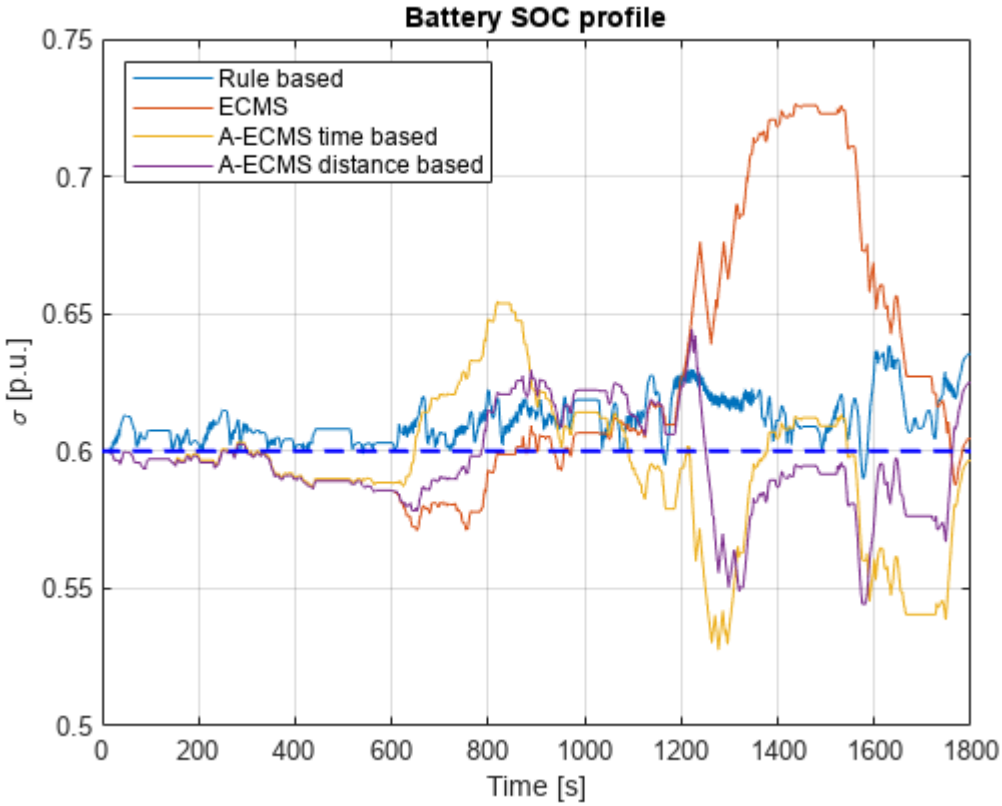
Controllers comparison: Rule based, ECMS, a-ECMS

To conclude the analysis, the adaptive ECMS controllers are put in comparison with the ones developed in the previous projects. This is done on the WLTP cycle since the data of the previous controllers are obtained from this mission. It is interesting to see how different controllers manage the power split strategy, leading to different SOC behaviours over the cycle, as well as the cumulative fuel consumption.

SOC profiles:

Starting from the SOC trend, the profiles of the rule based, ECMS and adaptive ECMS controllers are put in comparison.

```
% Extraction of previous projects data
load("resultsExtraFeature.mat")
prof_rule_based = prof;
load("results_ecms.mat")
prof_ecms = prof;

figure
plot(mission_WLTP.time_s, prof_rule_based.battPrf.battSOC), hold on
plot(mission_WLTP.time_s, prof_ecms.battPrf.battSOC), grid on
plot(mission_WLTP.time_s, prof_WLTP_time.battPrf.battSOC)
plot(mission_WLTP.time_s, prof_WLTP_dist.battPrf.battSOC)
plot(mission_WLTP.time_s, 0.6*ones(size(mission_WLTP.time_s)), LineStyle="--",
Color="b", LineWidth=1.2)
xlabel("Time [s]"); ylabel("\sigma [p.u.]")
title('Battery SOC profile')
legend('Rule based', 'ECMS', 'A-ECMS time based', 'A-ECMS distance based', location
='northwest')
xlim([0, 1800]);
```

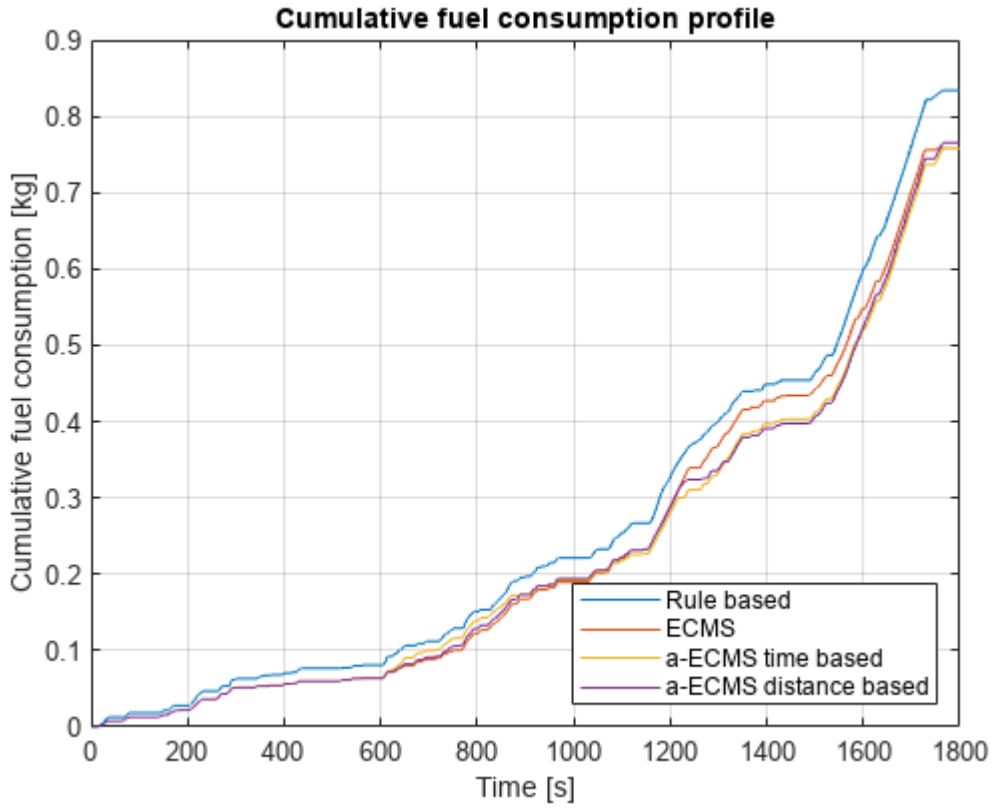


From a charge sustaining point of view, the rule based controller shows the most regular behaviour: in fact the state of charge is directly involved in the choice of the control logic, since the rules constantly compare the actual SOC with a target one. On the contrary, the non adaptive ECMS is simply based on an equivalent fuel minimization strategy, where only the SOC variation is taken into account, while the instantaneous SOC value has no influence. This results in a less controlled SOC profile, where the focus is only put on the final SOC value. Eventually, the adaptive ECMS shows an intermediate SOC trend, since the minimization function depends on the instantaneous equivalence factor, which is periodically corrected to bring the actual SOC close to a reference value. Finally, comparing the time and distance based controllers, no significant difference is evidenced, being the SOC variation in the same range.

Fuel consumption along the cycle:

```
figure
fc_rb = cumtrapz(mission_WLTP.time_s, prof_rule_based.engPrf.fuelFlwRate)/1000;   %
[kg]
fc_ECMS = cumtrapz(mission_WLTP.time_s, prof_ecms.engPrf.fuelFlwRate)/1000;    %
[kg]
fc_aECMS_time = cumtrapz(mission_WLTP.time_s, prof_WLTP_time.engPrf.fuelFlwRate)/
1000;    % [kg]
fc_aECMS_dist = cumtrapz(mission_WLTP.time_s, prof_WLTP_dist.engPrf.fuelFlwRate)/
1000;    % [kg]

plot(mission_WLTP.time_s, fc_rb), hold on
plot(mission_WLTP.time_s, fc_ECMS), grid on
plot(mission_WLTP.time_s, fc_aECMS_time)
```

```
plot(mission_WLTP.time_s, fc_aECMS_dist)
xlabel("Time [s]"); ylabel("Cumulative fuel consumption [kg]")
title('Cumulative fuel consumption profile')
legend('Rule based', 'ECMS', 'a-ECMS time based', 'a-ECMS distance based', location
='southeast')
xlim([0, 1800])
```



Even if the rule based has a better control over the SOC, its simple strategy doesn't minimize the fuel consumption, while the ECMS controllers work considering the minimum equivalent fuel consumption. In the end, from the fuel consumption point of view, the three ECMS controllers return very similar result. To analyze deeper the behaviour, the total energy consumption, that takes into account also the SOC variation, can be considered.

	Project 1 – Rule based	Project 2 - ECMS	Project 3 – AECMS (time)	Project 3 – AECMS (distance)
Fuel consumption [L/100km]	4.5670	4.1524	4.1510	4.1925
Final battery SOC [p.u.]	0.6350	0.6045	0.5965	0.6248
Total energy consumption [kWh]	9.9940	9.1298	9.1369	9.1912

Tab.2: comparison of fuel consumption, final SOC and total energy consumption between different controllers

Among the proposed controllers, the non adaptive ECMS shows the lowest energy usage: this is obtained since the complete mission is known in advance and the equivalence factor 's' is consequently tuned; by the way,

this solution is not implementable online. The adaptive strategy for the ECMS represents a good solution, being suitable for online applications despite of a negligible difference on the total energy consumption.

The adaptive ECMS behaviour is summarized as follows:

PROs:

- Online implementation is possible: the knowledge of the driving mission is not anymore required for the equivalence factor calibration.
- more controlled SOC behaviour with respect to non adaptive ECMS, which remains closer to the target value;
- lower fuel and energy consumption with respect to the rule based controller;
- close-to-optimal hybrid system exploitation.

CONs:

- Driveability issues related to gear shifting;
- high computational cost.

Possible improvements:

- Implementation of a specific logic to provide smoother gear changes, limiting the shifts between far gears and avoiding too frequent shifts;
- Perform a sensitivity analysis on the variation of the update distance for the distance based controller to obtain an optimal value.

Functions implementation

In this section the functions 'adaptiveEcmsControl' and 'SOCvariation' used in the code are implemented. The first contains the logic for the choice of the control variables, while the second exploits the first to perform the driving cycle simulation, to retrieve the time profiles of the main quantities of interest.

The a-ECMS controller

For the adaptive ECMS control strategy, the function 'adaptiveEcmsControl.m' is designed, with the following sintax:

[GN, PowerSplit, eqFactorNext] = adaptiveEcmsControl(SOC0, SOCactual, EqFactorActual, EqFactorPrevious, k_p, counter, update_interval, vehSpd, vehAcc, veh)

Function inputs:

- **SOC0**: initial battery state of charge [p.u.];
- **SOCactual**: instantaneous battery state of charge [p.u.];
- **EqFactorActual:** value of equivalence factor s for the current iteration [-] ;
- **EqFactorPrevious:** value of equivalence factor s used before the update [-] ;
- **k_p**: proportional gain in the equivalence factor controller;

- **counter**: variable recording the time or distance elapsed since the last equivalence factor update [s] or [m];
- **update_interval**: threshold value for equivalence factor update [s] or [m];
- **vehSpd:** vehicle speed [m/s];
- **vehAcc:** vehicle acceleration [m/s^2];
- **veh:** data structure containing vehicle parameters.

Function outputs:

- **GN:** engaged gear [-];
- **PowerSplit:** engine torque-split factor [-].
- **eqFactorNext**: value of equivalence factor s for the next iteration [-] ;

Function logics:

The function structure is the same implemented in project 2, with a modification related to the variable equivalence factor.

The goal of the function is to create a matrix, in each time instant, of the equivalent fuel consumption values for all the combinations of α_{eng} and γ given the speed and acceleration cycle requests; the minimum value, among the computed ones, is considered as working point, retrieving the corresponding gear number and torque-split factor.

To avoid the usage of computationally demanding concatenate cycles, the 'hev_dp_model' function is used to directly deal with input vectors, computing the fuel consumption for all the combinations of (γ, α_{eng}) within a single function iteration, leading to a significant reduction in the simulation time.

The improved function has the following sintax:

[x_next, stageCost, unfeas, engPrf, emPrf, battPrf, vehPrf] = hev_dp_model(x, u, w, veh).

With respect to 'hev_model' the u input of the function is a cell containing two vectors: one is a column vector of the gear numbers (from 1 to 5 in this case) and the other one is a row vector containing all the discrete values of the torque-split factor chosen for the simulation: a range between 0 and 3 is adopted with a step of 0.1, since it is observed that, even trying with a wider range, values above 3 are never selected by the algorithm. In addition, increasing the discretization accuracy doesn't provide significant changes in the results.

The function provides the fuel consumption value, as well as the next SOC value, for each combination of control variables, collecting them into matrices.

With the obtained values of possible future SOC, the discrete time derivative is computed as difference with the actual value, divided by the time step. This, in combination with the fuel consumption data, allows to calculate the equivalent fuel consumption matrix, performing the calculations with the value of equivalence factor contained in the input variable 'EqFactorActual'.

Among these possible combinations, not all can be selected by the control. In particular, the situations that must be avoided are the following:

- Unfeasible working points of ICE or EM;
- $SOC_{NEXT} < 0.4$;
- $SOC_{NEXT} > 0.8$.

The first two cases can be easily managed by the application of a penalty to the equivalent fuel consumption value so that, once its minimum value is searched, these are automatically ignored.

The last presents a further criticity: in a case in which the actual SOC is already close to 0.8 and the torque demand becomes negative, the only feasible solution for the system is to choose pure electric (regenerative braking); indeed, all the other torque-split values would arise an unfeasible condition, requiring negative torque to ICE. However, this keeps increasing the SOC. Penalising also this solution would result in a penalisation of all the possible combinations, bringing the system to select a meaningless operating condition.

To avoid this, only overcharge conditions occurring during traction (SOCnext>0.8 & demTrq>0) are immediately penalized. For the ones occurring during braking, a specific strategy is developed to interrupt regeneration enabling only mechanical braking: the ECMS control returns a meaningless value (-1) of α_{eng}, specifically conceived to identify this situation. Then, when this value is subsequently detected by the function 'SOCvariation', a specific logic implemented in 'hev_dp_model' manages this condition.

This countermeasure is conceived with a particular attention for battery safety: it avoids that, during prolongued regenerative braking, the SOC increases uncontrolled going above the upper limit. With the adopted strategy the SOC can never exceed 0.8.

Eventually, the minimum equivalent fuel consumption is identified and, consequently, the corresponding α_{eng} and γ indexes. The values of the control parameters are so chosen and returned by the function.

In addition, the function performs a periodic update of the equivalence factor to keep the SOC behaviour bounded and close to a reference value. In this case, the reference coincides with the initial value, being the system working in charge sustaining strategy.

The update is not performed at every iteration; it occurs only when the value of 'counter' reaches the value of 'update_interval'.

The update law takes the average between the equivalence factor values used before and after the last update and adds the difference between the target SOC and the actual one, multiplied by the factor 'k_p' which is provided as input to the function. In this way the weight associated to electric energy usage is changed according to the actual SOC evolution.

In conclusion, the value of equivalence factor to be used for the following iteration is returned in the output variable 'eqFactorNext'.

```
function [GN, PowerSplit, eqFactorNext] = adaptiveEcmsControl(SOC0, SOCactual,
EqFactorActual, EqFactorPrevious, k_p, counter, update_interval, vehSpd, vehAcc,
veh)
    PowerSplit_vec = linspace(0,3,61); % alpha values [-]
    GN_vec = [1:length(veh.gb.spdRatio.Values)]'; % column vector of gear number [-]
    [SOCnext, stageCost, unfeas, engPrf, emPrf] = hev_dp_model({SOCactual},
{GN_vec,PowerSplit_vec}, {vehSpd, vehAcc}, veh);
```

```
    SOCnext = SOCnext{1,1}; % Conversion from cell to matrix [p.u.]

    sigma_dot = (SOCnext-SOCactual)/veh.dt; % discrete SOC derivative [p.u./s]
    fcEquivalent = stageCost - EqFactorActual*(veh.batt.nomEnergy*3600)*1000/
(veh.eng.fuelLHV)*sigma_dot; % equivalent fuel consumption matrix [g/s]
    demTrq = engPrf.engTrq + emPrf.emTrq; % Needed to determine if traction or
braking [Nm]
    fcEquivalent(unfeas==1 | SOCnext<0.4 | (SOCnext>0.8 & demTrq>0)) =
fcEquivalent(unfeas==1 | SOCnext<0.4 | (SOCnext>0.8 & demTrq>0)) + 100; % exclusion
of unfeasible solutions through penalty

    [~, idx] = min(fcEquivalent(:)); % returns scalar index identifying the minimum
position
    [GN_idx, PowerSplit_idx] = ind2sub(size(fcEquivalent), idx); % transforms the
scalar index 'idx' into row and column numbers
    GN = GN_idx; % gear number coincides with row index [-]
    PowerSplit = PowerSplit_vec(PowerSplit_idx); % [-]

    % Check if regenerative braking is charging over 0.8
    SOCnext = SOCnext(GN_idx, PowerSplit_idx);
    if SOCnext > 0.8
        PowerSplit = -1; % Sets a meaningless value to detect overcharge situation
[-]
    end
    % Next equivalence factor computation
    if counter >= update_interval
        eqFactorNext = (EqFactorActual+EqFactorPrevious)/2 + k_p*(SOC0 - SOCactual);
    else
        eqFactorNext = EqFactorActual;
    end
end
```

(Optional) The distance-based a-ECMS controller

The minimization strategy working with the distance based update logic exploits the same function 'adaptiveEcmsControl'. The only difference between the two controllers is related to the 'SOCvariation' function: for the sake of code compactness, a single function, that can work for both the considered strategies, is defined. To switch to this strategy, the input 'updateMode' is set equal to 'distance'. In this case the updating of the equivalence factor is performed every 1 km travelled by the vehicle.

SOCvariation

The following function performs the calculations related to the entire driving cycle and returns the SOC variation $\sigma(t_f) - \sigma(t_0)$; moreover, the time profiles of the main vehicle quantities are stored in the structure 'prof'. The function sintax is shown:

[psi, prof] = SOCvariation(SOC0,mission,k_p,veh,updateMode)

Function inputs:

- **SOC0:** initial value of the battery SOC [p.u.];
- **mission:** data structure containing cycle speed profile [km/h] and time vector [s] ;
- **k_p**: proportional gain used in the updating formula of equivalence factor [-];
- **veh:** data structure containing vehicle parameters.
- **updateMode**: string indicating the controller logic: 'time' for the time based and 'distance' for the distance based one.

Function outputs:

- **psi:** SOC variation between final and initial values ($\sigma(t_f) - \sigma(t_0)$) [p.u.];
- **prof:** structure containing time profiles of ICE and EM torque and speed, battery SOC, cumulative fuel consumption, α_{eng}, engaged gear number and unfeasibility vectors.

Function logics:

The starting point is the extraction of the speed and acceleration profiles characteristic of the driving cycle.

The function 'adaptiveEcmsControl' requires as input the two previous values of equivalence factor. Since at the first iteration only one initial value is present, the vector 'EqFactor_values' is initialized assuming two equal values: the number comes from the calibration process done in project 2. In this vector, the calculated values of equivalence factor are progressively stored only in the iterations in which this parameter is updated. Moreover, a vector saving the value of equivalence factor used in every cycle iteration is created ('EqFactor_profile'), to be able to track the time evolution of this parameter.

Subsequently, two counters are initialized: 'counter' is updated at every iteration and used by 'adaptiveEcmsControl' to decide wether to update or not the equivalence factor, while 'updateIndex' grows every time 'counter' reaches 'update_interval' and is used as index to identify the components of 'EqFactor_values' vector.

Then, by means of an iterative procedure, for each cycle time instant the actual gear, torque-split strategy and next equivalence factor value are computed by the adaptive ECMS controller; consequently, the evolution of the vehicle variables is evaluated by the 'hev_dp_model' function. Its working principle is the same as the previous project, including also the strategy for the mechanical braking condition ($\alpha_{eng} = -1$).

At the beginning of each iteration, the variable 'counter' is updated in a different way depending on the update strategy selected: in the 'time' logic, it is incremented by the time step (1), while in the 'distance' logic, considering that its meaning corresponds to the vehicle travelled distance since the last equivalence factor update, its increment equals the distance travelled between the past and the current iteration.

Before ending the iteration, a check is performed: if 'counter' reaches the value of 'update_interval', its value is reset to 0, 'updateIndex' is incremented by 1 and the new value of equivalence factor is added to 'EqFactor_values' vector.

To conclude, the variation between final and initial SOC is calculated and the time profiles of the main quantities are converted into monodimensional structures and consequently packed into a unique structure called 'prof'.

```
function [psi, prof] = SOCvariation(SOC0,mission,k_p,veh,updateMode)
    SOC(1) = SOC0; % [p.u.]
```

```
    time = mission.time_s; % [s]
    vehSpd = mission.speed_kmh./3.6; % [m/s]
    vehAcc = mission.vehAcc; % [m/s^2]

    EqFactor_values = [2.5781 2.5781]; % vector storing only once the used values
of equivalence factor [-]
    EqFactor_profile = [2.5781]; % vector cointaining equivalence factor time
evolution for each n [-]
    updateIndex = 2;
    counter = 0;
    % time based
    if strcmp(updateMode,'time')
        update_interval = 30; % [s]
    % distance based
    elseif strcmp(updateMode,'distance')
        update_interval = 1000; % [m]
    else
        disp('Uncorrect sintax: use ''time'' for time based update or ''distance''
for distance based.')
        return
    end
    % mission simulation
    for n = 1:length(time)
        % counter update
        if strcmp(updateMode,'time')
            counter = counter + 1;
        else
            if n>1
                counter = counter + trapz(time(n-1:n), vehSpd(n-1:n));
            end
        end
        [GN(n), PowerSplit(n), EqFactor_profile(n+1)] = adaptiveEcmsControl(SOC0,
SOC(n), EqFactor_values(updateIndex), EqFactor_values(updateIndex-1), k_p, counter,
update_interval, vehSpd(n), vehAcc(n), veh);
        [SOCnext, stageCost(n), unfeas(n), engPrf(n), emPrf(n), battPrf(n),
vehPrf(n)] = hev_dp_model({SOC(n)}, {GN(n), PowerSplit(n)}, {vehSpd(n), vehAcc(n)},
veh);
        SOC(n+1)= cell2mat(SOCnext);
        % counter reset and equivalence factor value saved
        if counter >= update_interval
            counter = 0;
            updateIndex = updateIndex + 1;
            EqFactor_values = [EqFactor_values, EqFactor_profile(n+1)];
        end
    end
    finalSOC = SOC(end); % [p.u.]
    psi = finalSOC-SOC0;

    % Transform the non-scalar structure, containing time profiles, into scalar
structures; this makes their manipulation easier.
```

```
    engPrf = structArray2struct(engPrf);
    emPrf = structArray2struct(emPrf);
    battPrf = structArray2struct(battPrf);
    vehPrf = structArray2struct(vehPrf);
    vehPrf.eqFactor_profile = EqFactor_profile(2:end)';
    % Pack profiles into a single structure
    prof.engPrf = engPrf;
    prof.emPrf = emPrf;
    prof.battPrf = battPrf;
    prof.vehPrf = vehPrf;
    prof.unfeas = unfeas;
end
```

Additional functions

In the following section some additional functions are implemented; some of them are integral part of the project functioning, while others are just meant to avoid repeated operations, to keep the code cleaner.

tuneKp

The following function performs the tuning of k_p by selecting a suitable value within a given vector. The tuning is performed considering the WLTP cycle and the Artemis ones in urban, rural and motorway scenarios. The goal is to find an optimal value of the proportional gain that leads to a final value of the SOC within 0.5 and 0.7 for each cycle; to chose among the values that satisfy the requirement, an objective function is implemented and minimized. Function sintax:

tunedK_p = tuneKp(k_p_values, SOC0, mission_WLTP, mission_AUDC_5x, mission_ARDC_2x, mission_AMDC, veh, updateMode)

Function inputs:

- **k_p_values**: vector containing all the tested values of proportional gain [-];
- **SOC0**: initial value of the battery SOC [p.u.];
- **mission_WLTP**: data structure, based on WLTP cycle, containing cycle speed profile [km/h], acceleration profile [m/s^2] and time vector [s];
- **mission_AUDC_5x**: assembled data structure, based on the Artemis Urban driving cycle, containing five times the speed profile [km/h], acceleration profile [m/s^2] and the time vector [s];
- **mission_ARDC_2x**: assembled data structure, based on the Artemis Rural driving cycle, containing twice the speed profile [km/h], acceleration profile [m/s^2] and the time vector [s];
- **mission_AMDC**: data structure, based on the Artemis Motorway driving cycle, containing the speed profile [km/h], acceleration profile [m/s^2] and the time vector [s];
- **k_p**: proportional gain used in the updating formula of equivalence factor [-];
- **veh:** data structure containing vehicle parameters.
- **updateMode**: string indicating the controller logic: 'time' for the time based and 'distance' for the distance based one.

Function outputs:

- **tunedK_p**: chosen value of the proportional gain [-];

Function logics:

For each provided value of k_p contained in vector 'k_p_values' the different missions are simulated by means of the function 'SOCvariation', which provides the variation between initial and final SOC. If this variation is bounded within the desired range for all the cycles, the current value of k_p is stored in a vector called 'k_p_suitable' and the sum of the absolute values of SOC variation of each cycle is computed ('psiCostFunction'); instead, if the requirements are not met, simply a 'NaN' value is stored. Once these operations are performed for all the values in 'k_p_values', the value of k_p which provides the minimum of 'psiCostFunction' is selected and returned in 'tunedK_p'.

Moreover, a chart showing the trend of the objective function ('psiCostFunction') as a function of the suitable values of k_p is created, to track the behaviour of the calibration strategy.

```
function tunedK_p = tuneKp(k_p_values, SOC0, mission_WLTP, mission_AUDC_5x,
mission_ARDC_2x, mission_AMDC, veh, updateMode)
    k_p_suitable = [];
    for i = 1:length(k_p_values)
        % Cycles simulation
        WLTP_check = 0; AUDC_check = 0; ARDC_check = 0; AMDC_check = 0;
        % WLTP
        [psi_WLTP, ~] = SOCvariation(SOC0,mission_WLTP, k_p_values(i), veh,
updateMode);
        if abs(psi_WLTP)<0.1
            WLTP_check = 1;
        end
        if WLTP_check == 1
            % AUDC 5x
            [psi_AUDC, ~] = SOCvariation(SOC0,mission_AUDC_5x, k_p_values(i), veh,
updateMode);
            if abs(psi_AUDC)<0.1
                AUDC_check = 1;
            end
            if AUDC_check == 1
                % ARDC 2x
                [psi_ARDC, ~] = SOCvariation(SOC0,mission_ARDC_2x, k_p_values(i),
veh, updateMode);
                if abs(psi_ARDC)<0.1
                    ARDC_check = 1;
                end
                if ARDC_check == 1
                    % AMDC
                    [psi_AMDC, ~] = SOCvariation(SOC0,mission_AMDC, k_p_values(i),
veh, updateMode);
                    if abs(psi_AMDC)<0.1
                        AMDC_check = 1;
                    end
```

```
                end
            end
        end
        if WLTP_check && AUDC_check && ARDC_check && AMDC_check
            k_p_suitable = [k_p_suitable,k_p_values(i)];
            psiCostFunction(i) = abs(psi_WLTP)+abs(psi_AUDC)+abs(psi_ARDC)
+abs(psi_AMDC);
        else
            psiCostFunction(i) = NaN;
        end
    end
    [~, idx] = min(psiCostFunction);
    tunedK_p = k_p_values(idx);
    psiCostFunction = rmmissing(psiCostFunction);
    figure
    plot(k_p_suitable,psiCostFunction); grid on
    xlabel('k_{p} [-]');
    xlim([k_p_suitable(1) k_p_suitable(end)])
    ylabel('Final SOC deviation [p.u.]');
    title({updateMode 'based controller, objective function trend'});
end
```

missionProfiles

The following function performs the calculation of the acceleration vector starting from the given mission speed profile and plots the time profiles of these two quantities. The sintax is:

[mission_out] = missionProfiles(mission_in)

Function inputs:

- **mission_in:** data structure containing cycle speed profile [km/h] and time vector [s] ;

Function outputs:

- **mission_out:** input mission structure with the addition of vehicle acceleration vector [m/s^2] ;

Function logics:

Starting from the speed profile, the acceleration is computed performing a discrete derivative operation: each component of the vector is obtained dividing the difference between two consecutive speed values by the time step. Doing so, the obtained vector of acceleration has a length (N-1), where N is the number of time instants; a first component equal to 0 is added to the acceleration vector in order to work with vectors of the same length. This assumption is coherent with the cycles since the first terms of the speed vectors are constant and equal to 0. Moreover plots of the speed and acceleration as a function of time are performed.

```
function [mission_out] = missionProfiles(mission_in)
    cycleName = mission_in.cycleName;
    vehSpd = mission_in.speed_kmh./3.6; % [m/s]
    time = mission_in.time_s; % [s]
    dt = time(2) - time(1); % [s]
```

```
    vehAcc = (vehSpd(2:end)-vehSpd(1:end-1))./dt; % [m/s^2]
    % First component assumed 0 added to the acceleration vector
    vehAcc = [0;vehAcc]; % [m/s^2]
    mission_in.vehAcc = vehAcc;
    mission_out = mission_in;

    % The speed and acceleration profiles are plotted against time
    figure;
    t = tiledlayout(2,1);
    title(t, cycleName)
    nexttile(1)
    plot(time,vehSpd,'LineWidth',1)
    xlim([0 time(end)])
    grid minor
    title('Vehicle speed')
    xlabel('Time [s]')
    ylabel('Vehicle speed [m/s]')
    nexttile(2)
    plot(time,vehAcc,'LineWidth',1)
    xlim([0 time(end)])
    grid minor
    title('Vehicle acceleration')
    xlabel('Time [s]')
    ylabel('Vehicle acceleration [m/s^2]')
end
```

unfeasCheck

The following function performs a check over all the working points of the system and returns an error message if any unfeasibility is found. The sintax is:

unfeasCheck(prof,mission)

Function inputs:

- **prof:** structure containing all system unfeasibility vectors;
- **mission:** data structure containing cycle speed [km/h], acceleration profile [m/s^2] and time vector [s] as well as the name of the cycle;

Function logics:

If any unfeasibility is detected (unfeas == 1), all the vectors of EM and ICE speed and torque and battery unfeasibilities are checked and if any value equal to 1 is found a specific message is displayed, containing information about the cycle, the iteration number and the type of error encountered.

```
function unfeasCheck(prof,mission)
    for n = 1:length(mission.time_s)
        if prof.unfeas(n)==1
            if prof.engPrf.engSpdUnfeas(n)==1
                disp(['In ' mission.cycleName ' at iteration ' num2str(n) ' the
engine speed exceeds the limits.'])
```

```
            end
            if prof.engPrf.engTrqUnfeas(n)==1
                disp(['In ' mission.cycleName ' at iteration ' num2str(n) ' the
engine torque exceeds the limits.'])
            end
            if  prof.emPrf.emSpdUnfeas(n)==1
                disp(['In ' mission.cycleName ' at iteration ' num2str(n) ' the
electric motor speed exceeds the limits.'])
            end
            if  prof.emPrf.emTrqUnfeas(n)==1
                disp(['In ' mission.cycleName ' at iteration ' num2str(n) ' the
electric motor torque exceeds the limits.'])
            end
            if prof.vehPrf.battUnfeas(n) == 1
                disp(['In ' mission.cycleName ' at iteration ' num2str(n) ' the
battery current exceeds the limits.'])
            end
            if  prof.vehPrf.pwtUnfeas(n)==1
                disp(['In ' mission.cycleName ' at iteration ' num2str(n) '
uncoherent EM torque request.'])
            end
        end
    end
end
```

SOCvsSprofile

This function is implemented to show the chart of SOC evolution together with the equivalence factor throughout a driving mission, for the time and distance based adaptive ECMS controllers. The sintax is:

SOCvsS_profiles(mission, profTime, profDist)

Function inputs:

- **mission:** data structure containing cycle speed [km/h], acceleration profile [m/s^2] and time vector [s] as well as the name of the cycle;
- **profTime:** structure containing the SOC and equivalence factor time profiles for time based controller ;
- **profDist:** structure containing the SOC and equivalence factor time profiles for distance based controller;

Function logics:

The function returns two figures in one column comparing the SOC and equivalence factor time evolution of the time and distance based adaptive ECMS controllers for a selected mission.

```
function SOCvsS_profiles(mission, profTime, profDist)
    figure
    t = tiledlayout(2,1);
    nexttile(1)
    hold on; grid on;
    xlabel('Time [s]')
    xlim([mission.time_s(1),mission.time_s(end)])
```

```
    yyaxis left;
    plot(mission.time_s, profTime.battPrf.battSOC);
    ylabel('Battery SOC [p.u.]');
    yyaxis right;
    plot(mission.time_s, profTime.vehPrf.eqFactor_profile);
    ylabel('Equivalence factor [-]')
    title('SOC and Equivalence factor profiles')
    legend('Battery SOC - Time controller', 'Eq. factor - Time controller');
    nexttile(2)
    hold on; grid on;
    xlabel('Time [s]')
    xlim([mission.time_s(1),mission.time_s(end)])
    yyaxis left;
    plot(mission.time_s, profDist.battPrf.battSOC);
    ylabel('Battery SOC [p.u.]');
    yyaxis right;
    plot(mission.time_s, profDist.vehPrf.eqFactor_profile);
    ylabel('Equivalence factor [-]')
    legend('Battery SOC - Distance controller', 'Eq. factor - Distance controller');
end
```

energeticCalculations

The following function performs the calculation of cumulative fuel consumption, fuel economy and global energy conumption for a given driving mission. Function sintax:

Function inputs:

- **mission:** data structure containing cycle speed [km/h], acceleration profile [m/s^2] and time vector [s] as well as the name of the cycle;
- **prof:** structure containing the fuel flow rate and the battery voltage and current time profiles;
- **veh:** structure containing the main fuel parameters.

Function outputs:

- **fuelConsumption:** cumulative cycle fuel consumption [kg];
- **fuelEconomy:** cycle fuel economy [L/100km];
- **totalEnergyConsumption:** sum of electric and fuel energy used for the cycle [kWh].

Function logics:

The cumulative fuel consumption is calculated by integrating the fuel mass flow rate over the mission time; then, dividing by the cycle distance, the fuel economy is calculated. Eventually the energy used during the cycle is retrieved by summing the electrical energy with the fuel consumption one. It is also taken into account the battery charge/discharge with respect to the initial SOC; in fact the electrical energy consumption 'elEnergyConsumption' is negative when the final SOC is higher than the initial one, while it is positive viceversa: this comes as a consequence of integrating the battery power over the mission time.

```
function [fuelConsumption, fuelEconomy, totalEnergyConsumption] =
energeticCalculations(mission, prof, veh)
```

```
    fuelConsumption = trapz(mission.time_s, prof.engPrf.fuelFlwRate)/1000; %
Cumulative fuel consumption [kg]
    vehDist = trapz(mission.time_s, mission.speed_kmh/3.6); % Total cycle distance
[m]
    fuelEconomy = (fuelConsumption*10^5) / (veh.eng.fuelDensity*vehDist); % Fuel
consumption [L/100km]
    elEnergyConsumption = trapz(mission.time_s,
prof.battPrf.battVolt.*prof.battPrf.battCurr); % battery electrical energy [J]
    fuelEnergyConsumption = fuelConsumption*veh.eng.fuelLHV; % [J]
    totalEnergyConsumption = (elEnergyConsumption + fuelEnergyConsumption)/
1000/3600; % [kWh]
end
```

Assignment #4: dynamic programming

Table of Contents

Group information

Group number: 49

Students:

- Carlo Vittorio Colucci, s329703
- Riccardo Bressani, s323665
- Luca Marchetto, s323437

Project introduction

The aim of the project is to evaluate the optimal energy managment strategy in a p2 parallel HEV (represented in Fig.1) at each time instant of a the WLTP driving profile.

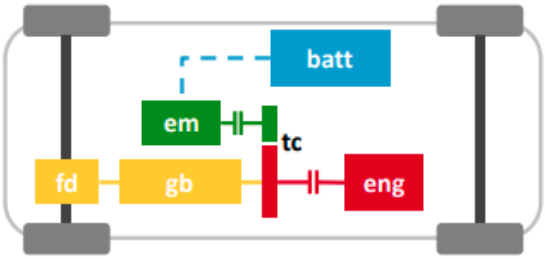


Fig.1: Scheme of a P2 parallel HEV

The procedure exploits the 'DynaProg' Matlab toolbox, used for the solution of both the backward and forward phases of a dynamic programming (DP) problem. The solution is performed considering the battery SOC as state variable, subject to charge sustaining constraints ($\sigma(0) = 60$ and $\sigma(end) = 60 \pm 1$), and as control variables the engaged gear (γ) and the engine torque-split factor (α_{eng}), defined accordingly:

$$\alpha_{eng} = \frac{T_{eng}}{T_{dem}}$$

The vehicle speed and acceleration are provided as exogenous inputs, while the cost function minimized by the algorithm is the fuel consumption.

Once the optimal solution is retrieved, some modifications are implemented in the vehicle model and in the cost function expression, with the aim of solving two issues related to all the controllers designed so far:

- too frequent gear shifts;
- repeated engine switch on;

To do so, penalties are introduced in the stage cost for the control strategies implying a gear shift or an engine switch on:

$$L_k = \dot{m}_f(\alpha_{eng}) + c_1 \cdot (\gamma \neq \gamma_{prev}) + c_2 \cdot (\alpha_{eng} > 0 \,\&\, \varepsilon = 0)$$

where L_k is the stage cost, γ_{prev} and ε are respectively the engaged gear and the engine state (1 = ON, 0 = OFF) in the previous iteration and the coefficients c_1, c_2 are weighting factors associated to each penalty, tuned in the project to obtain the desired system behaviour.

To trace the evolution of γ_{prev} and ε throughout the simulation, these quantities are added to the vector of the states in the vehicle model contained in 'hev_dp_model' function.

The aim of the project is to provide a control strategy able to realize a sub-optimal energy management, satisfying the following constraints on average:

- less than one gear shift per minute;
- less than one engine switch on per minute.

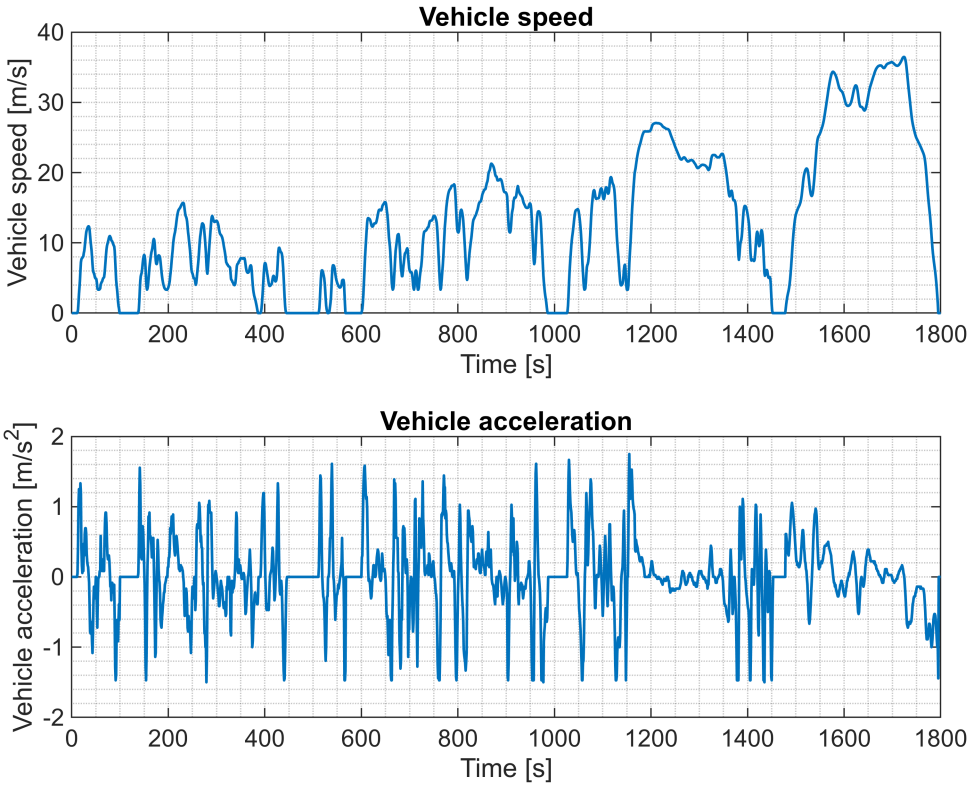
Load the cycle and vehicle data

The reference cycle which is used in this project is loaded in the following section: it consists in vectors representing speed profiles with respect to time. The data concerning the cycle are contained in .mat files stored in 'data' folder. The considered cycle is the WLTP.

Starting from the speed profile, the acceleration is computed performing a discrete derivative operation: each component of the vector is obtained dividing the difference between two consecutive speed values by the time step. Doing so, the obtained vector of acceleration has a length (N-1), where N is the number of time instants; a first component equal to 0 is added to the acceleration vector in order to work with vectors of the same length. This assumption is coherent with the cycle since the first terms of the speed vector are constant and equal to 0.

```
clear all
clc

% Addition of folders
addpath('data')
addpath('models')
addpath('utilities')
```

```
% Extraction of cycle velocities and accelerations
% WLTP
mission_WLTP = load('data\WLTP3.mat');
vehSpd = mission_WLTP.speed_kmh./3.6; % [m/s]
time = mission_WLTP.time_s; % [s]
dt = time(2) - time(1); % [s]
vehAcc = (vehSpd(2:end)-vehSpd(1:end-1))./dt; % [m/s^2]
% First component assumed 0 added to the acceleration vector
vehAcc = [0;vehAcc]; % [m/s^2]

% The speed and acceleration profiles are plotted against time
t = tiledlayout(2,1);
nexttile(1)
plot(time,vehSpd,'LineWidth',1)
grid minor
title('Vehicle speed')
xlabel('Time [s]')
ylabel('Vehicle speed [m/s]')
nexttile(2)
plot(time,vehAcc,'LineWidth',1)
grid minor
title('Vehicle acceleration')
xlabel('Time [s]')
ylabel('Vehicle acceleration [m/s^2]')
```



By using the following commands, the data related to the vehicle are loaded from the file 'vehData.mat'. These data are subsequently rescaled using the function 'scaleVehData.m' considering as additional inputs the values of ICE power [W], EM power [W], battery capacity [Ah] associated to the group number:

- **Group number:** 49
- **ICE power:** 60000 W
- **EM power:** 18000 W
- **Battery capacity:** 6.4 Ah

```
% Data loading and scaling
veh = load('data\vehData.mat');
engPwr = 60000; % [W]
emPwr = 18000; % [W]
battCap = 6.4; % [Ah]
veh = scaleVehData(veh,engPwr,emPwr,battCap); % scaleVehData(veh, engPwr[W],
emPwr[W], battCap[Ah])
```

Dynamic programming implementation

The following section uses the toolbox 'DynaProg'; to do this, both the state and control variables must be discretized. The discretization is a trade-off between two different aspects: a too rough mesh wouldn't be able to identify the optimal solution, on the other hand increasing the considered points makes the computation heavier and heavier. The mesh adopted in the project is simply identified by a trial and error procedure, to obtain a reasonable computational time. However, it is observed that a slight improvement in fuel consumption could still be achieved by further increasing the number of discretized values: dealing with a global optimization problem which is not suitable for online application, in the case that a more powerful calculator was available, it could be worth investigating even more refined grids to obtain, among the possible solutions, the closest to the optimal one.

The solution of 'DynaProg' exploits the function 'hev_dp_model', which takes as additional inputs the speed and acceleration profiles as well as the structure 'veh' containing the vehicle data; then, it solves the longitudinal dynamic equations of the vehicle to retrieve the evolution of the state as a function of the control variables and exogenous input. The structure of the function is the same adopted in the previous projects.

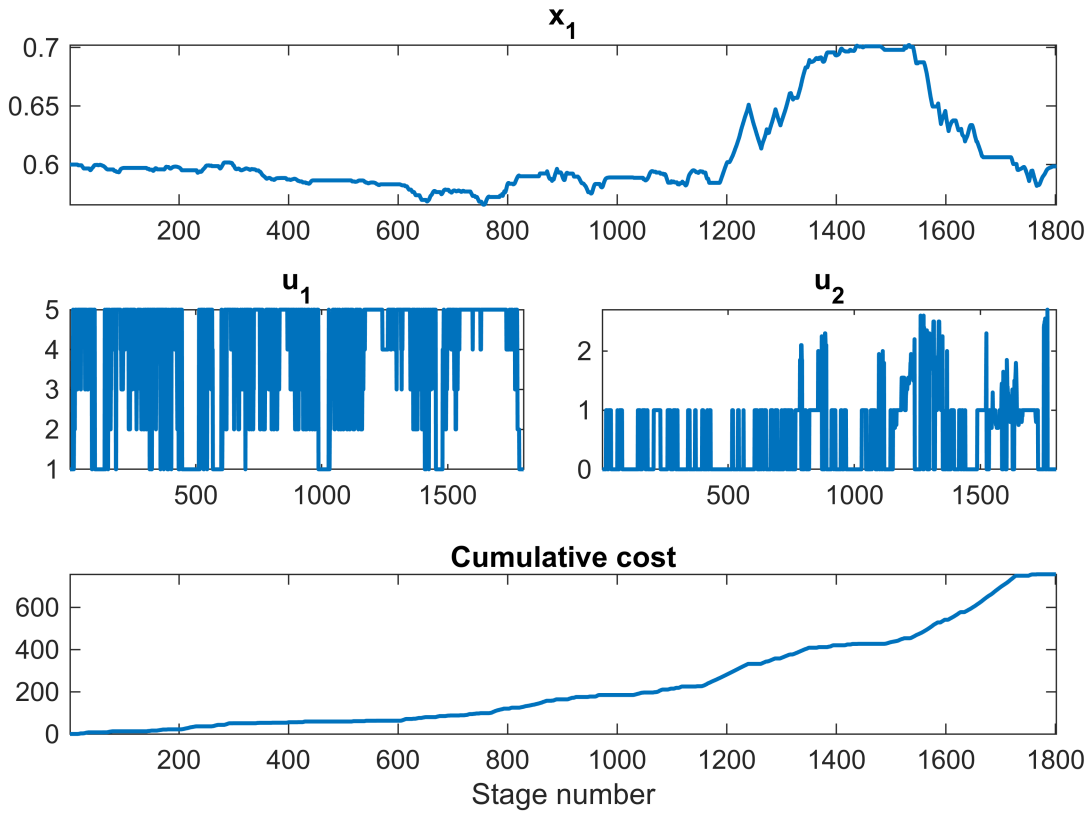
```
% Initial conditions
SOC0 = 0.6; % [p.u.]
Gear0 = 1; % [-]

% DP grid and inputs definition
StateGrid = {0.4:0.001:0.8}; % SOC [p.u.]
StateInitial = {SOC0}; % [p.u.]
StateFinal = {[0.59 0.61]}; % [p.u.]
GearGrid = [1 2 3 4 5]; % gear numbers [-]
AlphaGrid = 0:0.05:3; % Torque split factor [-]
ControlGrid = {GearGrid, AlphaGrid};
Nstages = length(time);
w = {vehSpd,vehAcc}; % exogenous input
```

```
% Additional inputs for hev_dp_model
SysName = @(StateGrid, ControlGrid, w) hev_dp_model(StateGrid, ControlGrid, w, veh);
% Definition and solution of the problem through DynaProg
prob = DynaProg(StateGrid, StateInitial, StateFinal, ControlGrid, Nstages, SysName,
'ExogenousInput', w);
prob = run(prob);
```

```
DP backward progress:100 %
DP forward progress:100 %
```

The algorithm also allows to track the time profiles of the state ($x_1 = SOC$) and control variables ($u_1 = \gamma, u_2 = \alpha_{eng}$), as well as the cumulative cost; in this case, since no penalties have been applied, the cumulative cost coincides with the cumulative fuel consumption [kg].

```
plot(prob);
```



As expected, the final battery SOC correctly falls in the range provided to 'StateFinal' in 'DynaProg'.

As anticipated, the average number of gear shifts and engine switch on [1/min] throughout the entire cycle is an indicator of the behaviour of the system. These quantities are calculated in the function 'DPprocessing', taking as input the structure 'prob', resulting from 'DynaProg', and the structure of the speed mission. In addition, the function creates the structure 'prof_DP' containing the main vehicle, EM, ICE and battery quantities time profiles.

```
[prof_DP, gearShiftAvg, engineStartAvg] = DPprocessing(prob, mission_WLTP);
gearShiftAvg % [min^-1]
```

```
gearShiftAvg = 15.9333
```

```
engineStartAvg % [min^-1]
```

```
engineStartAvg = 2.3667
```

The objective of the second part of the project is to bring these value below 1.

To conclude the calculation of the useful quantities for the results analysis, the fuel consumption [kg], fuel economy [L/100km] and total energy consumption [kWh] are retrieved by the function 'energeticCalculations', already introduced in the previous projects. The final SOC is instead taken from the vector of the SOC time profile, contained in 'prof_DP'.

```
[fuelConsumption_DP, fuelEconomy_DP, totalEnergyConsumption_DP] =
energeticCalculations(mission_WLTP, prof_DP, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_DP = 0.7568
fuelEconomy_DP = 4.1443
totalEnergyConsumption_DP = 9.1199
```

```
finalSOC_DP = prof_DP.battPrf.battSOC(end) % [p.u.]
```

```
finalSOC_DP = 0.5985
```

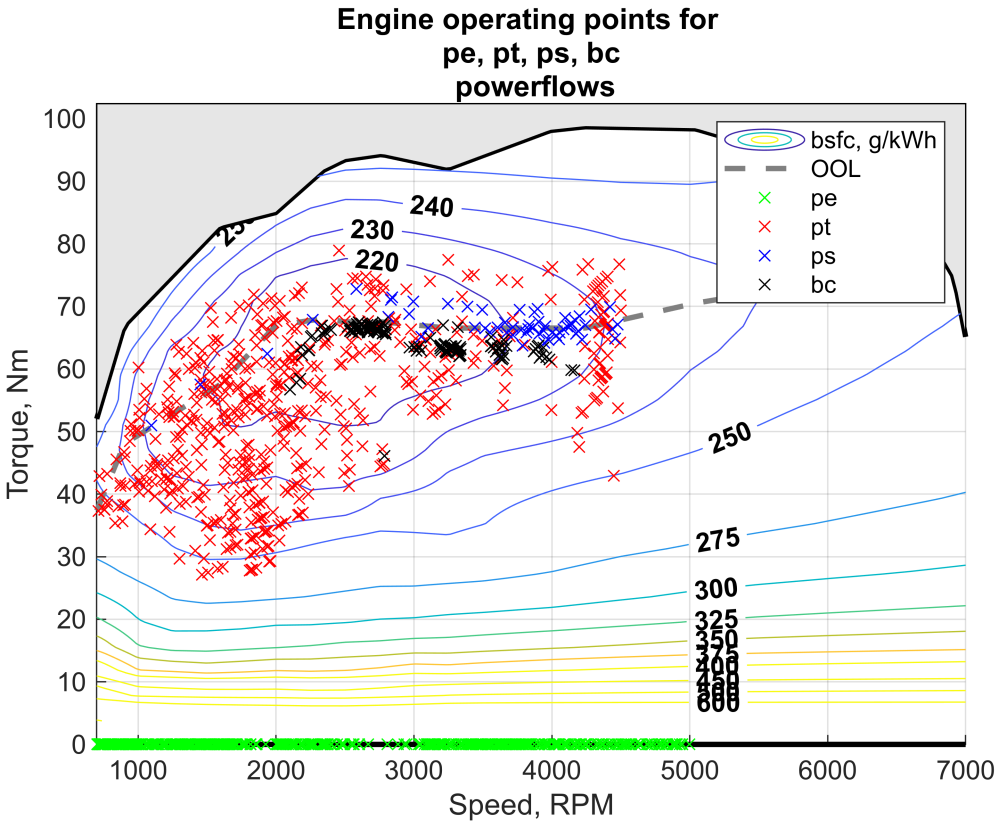
Post-processing

In this section, the behaviour of the dynamic programming solution is inspected by means of significative charts.

ICE map:

The engine operating points are shown together with the brake specific fuel consumption iso lines; the points highlight the torque-split mode selected by the controller.

```
% Engine operating points
engMapWithPF(veh.eng, prof_DP, "bsfc", 'all')
```

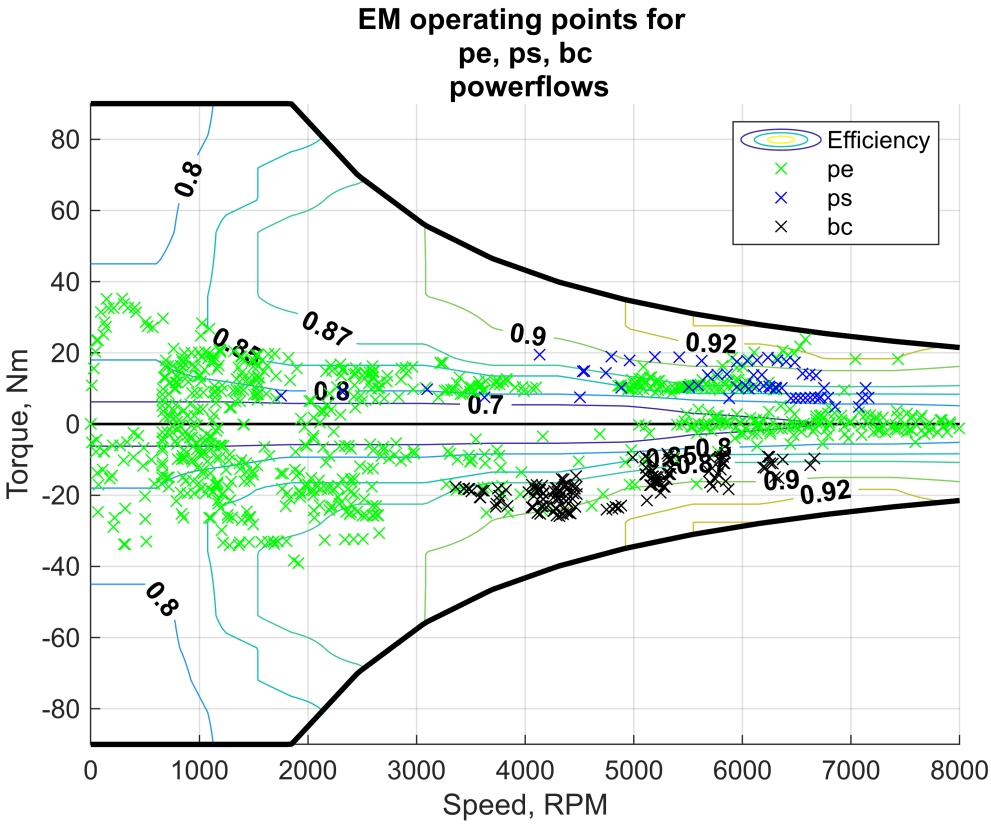


As expected, the majority of the ICE operating points selected in the optimal solution lie in a low brake specific fuel consumption region; the pure thermal operating points are a bit more spread throughout the map since, in that operating mode, the engine torque must exactly meet the vehicle requirement. By converse, in power split and battery charging, where the additional degree of freedom provided by the EM torque is added, the ICE can operate much closer to the optimal operating line.

EM map:

The EM operating points are shown in the map, together with the efficiency iso lines.

```
% EM operating points
emMapWithPF(veh.em, prof_DP, ["pe", "ps", "bc"]);
```

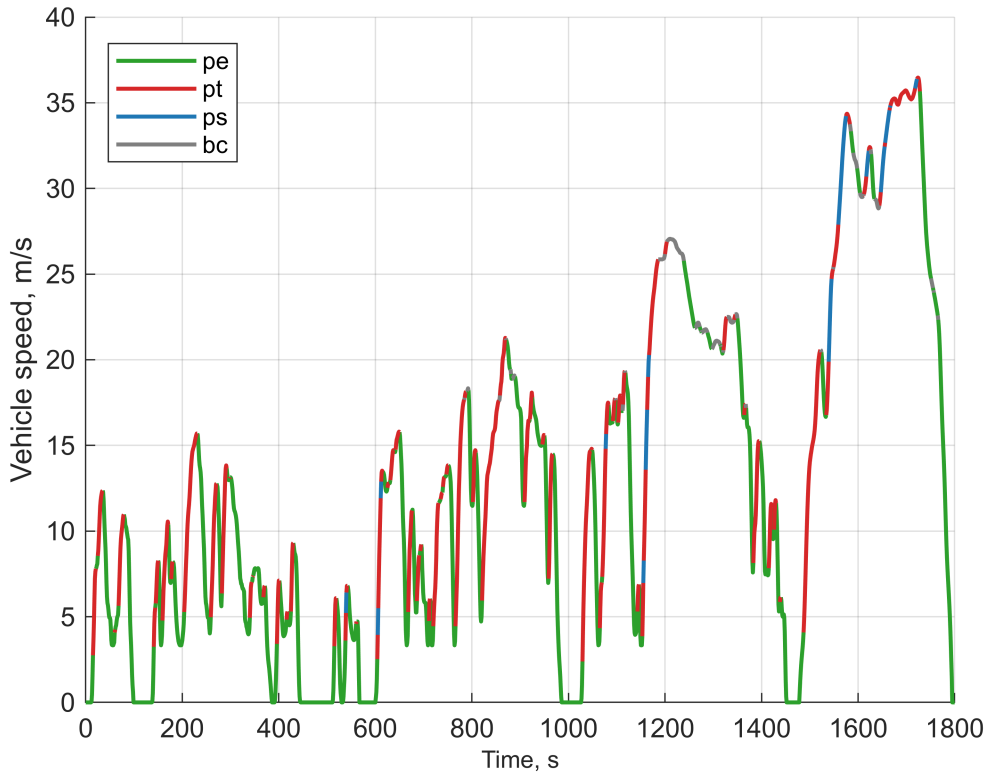


As already mentioned in the previous projects, it is more difficult to identify a trend of the system behaviour from the EM map, since the efficiency variation throughout the map of the EM is less significant with respect to the ICE one. However the logic of the dynamic programming algorithm guarantees that the selected operating points lead to the global optimal solution.

Power flows:

The power flows chart shows the vehicle speed profile, highlighting the operating mode selected in each instant.

```
% Power flows
powerProfiles(prof_DP, 'all');
```



This chart could be useful as a first tool to identify some trends in the algorithm choices, which could be exploited, for instance, to develop a rule based controller for online implementation; it is shown for example how pure electric is always selected during the vehicle slow down and for low speed traction, while accelerations at higher speeds are carried out exploiting mainly pure thermal, with some contributions of power split.

Fuel-optimal EMS with driveabilty

The following section develops the solution of the dynamic programming, including the above mentioned driveability constraints:

- less than one gear shift per minute;
- less than one engine switch on per minute.

To do so, two penalties are added to the stage cost for the control strategies that imply a gear shift or an engine switch on with respect to the previous instant. The modifications to the model are carried out in a new function, called 'hev_dp_model_modified', which is based on the already used function 'hev_dp_model'. The syntax of the new version is:

[x_next, stageCost, unfeas, engPrf, emPrf, battPrf, vehPrf] = hev_dp_model_modified(x, u, w, gearPenalty, engPenalty, veh)

With respect to the base one, the updated function has the following changes:

- x and x_next (state vector in current and future iterations respectively) contain two additional state variables, the previous gear and engine state, so that it is possible to track these values, in order to attribute the penalties to the above mentioned situations. The state equations governing the new variables evolution are (given γ_k and $\alpha_{eng,k}$ the control variables at instant k):

$$\gamma_{\text{prev},\,k+1} = \gamma_k$$
$$\varepsilon_{k+1} = \left(\alpha_{\text{eng},k} > 0\right)$$

- the parameters 'gearPenalty' and 'engPenalty' are provided as additional inputs to the function and represent the weight attributed to the penalties in gear shift and engine switch on cases respectively;
- the stage cost contains, in addition to the fuel flow rate, the two penalties enforced to improve driveability.

The calibration of the penalties is carried out in a separate function, called 'penaltiesTuning'; in the function, the DP algorithm is applied adopting different values of 'gearPenalty' and 'engPenalty' which, starting from low values, are progressively increased until the driveability conditions are met; for a more detailed description and to visualize the code, please look at the file 'penaltiesTuning' in the 'models' folder. Since the calculations performed in the function iterate many values of penalty, its execution requires a lot of time. Therefore, they are not directly reported in the code, but the results are saved in the file 'penaltiesTuningResults' and loaded below.

```
% results of penaltiesTuning(mission_WLTP, veh)
load('penaltiesTuningResults')
gearPenaltyTuned
```

```
gearPenaltyTuned = 0.1300
```

```
engPenaltyTuned
```

```
engPenaltyTuned = 0.3200
```

Having added new state variables to the vehicle model, new grids must be provided for the additional variables; by the way, this is a quite trivial operation: indeed, being the new variables the previous gear and engine state, the discretizations are simply provided by the vector of the gear numbers and [0 1] for the engine state. For what concerns the final constraints, only the SOC is limited between 0.59 and 0.61.

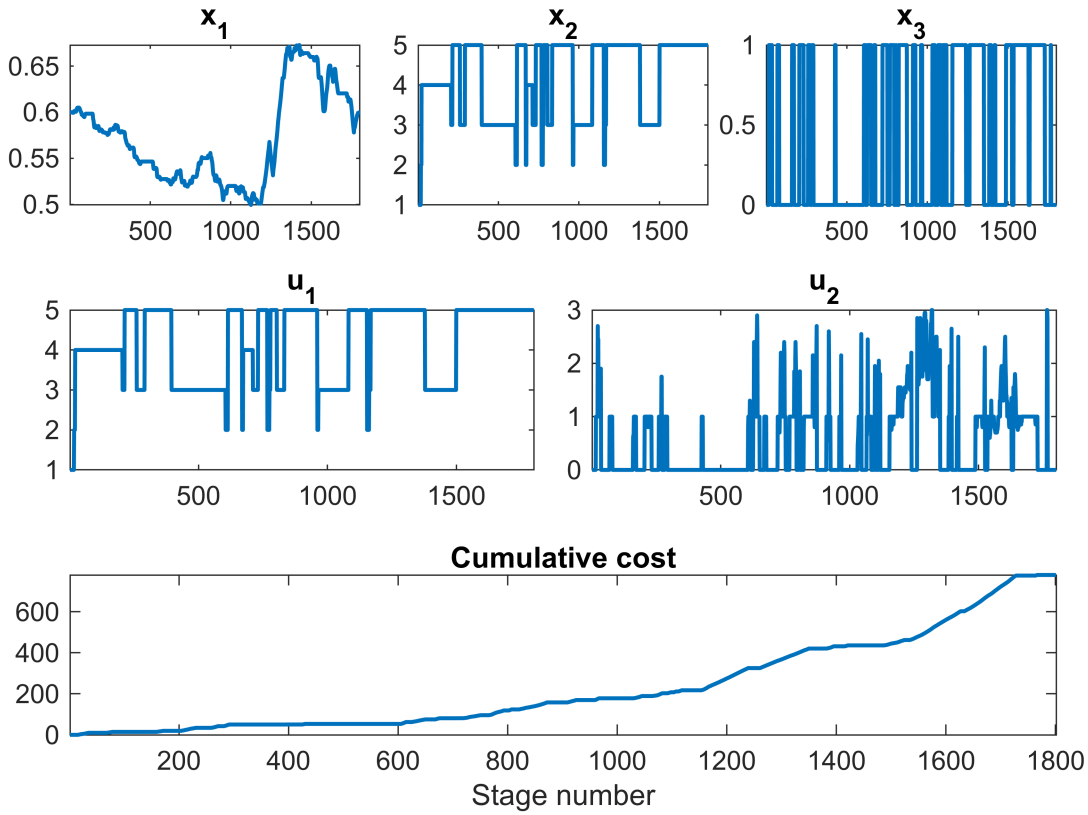
```
% New initial conditions
EngState0 = 0; % [-]

% New DP grid and inputs definition
StateGrid = {0.4:0.001:0.8, GearGrid, [0 1]}; % {[p.u.], [-], [-]}
StateInitial = {SOC0, Gear0, EngState0}; % {[p.u.], [-], [-]}
StateFinal = {[0.59 0.61],[],[]}; % {[p.u.], [-], [-]}
% Additional inputs for hev_dp_model_modified
SysName = @(StateGrid, ControlGrid, w) hev_dp_model_modified(StateGrid,
ControlGrid, w, gearPenaltyTuned, engPenaltyTuned, veh);
% Definition and solution of the problem through DynaProg
probWithPenalties = DynaProg(StateGrid, StateInitial, StateFinal, ControlGrid,
Nstages, SysName, 'ExogenousInput', w);
```

```
probWithPenalties = run(probWithPenalties);
```

```
DP backward progress:100 %
DP forward progress:100 %
```

As in the first case, the behaviour of the state ($x_1 = SOC$, $x_2 = \gamma_{prev}$, $x_3 = \varepsilon$) and control variables ($u_1 = \gamma, u_2 = \alpha_{eng}$) can be inspected, as well as the cumulative cost; in this case, the cumulative cost doesn't coincide anymore with the cumulative fuel consumption [kg], since the additional terms related to the penalties are present.

```
figure
plot(probWithPenalties);
```



From a graphical point of view it can already be observed that both the gear shifts and the passages of α_{eng} from 0 to 1 (engine switch on) are significantly less than the previous case. The average values are calculated below, in the function 'DPprocessing':

```
[prof_DP_WithPenalties, gearShiftAvgWithPenalties, engineStartAvgWithPenalties] =
DPprocessing(probWithPenalties, mission_WLTP);
gearShiftAvgWithPenalties  % [min^-1]
```

```
gearShiftAvgWithPenalties = 0.8667
```

```
engineStartAvgWithPenalties  % [min^-1]
```

```
engineStartAvgWithPenalties = 0.9667
```

The strategy reaches the goal to bring these values below 1.

Following the same structure of the previous section, the fuel consumption [kg], the fuel economy [L/100km] and the total energy consumption [kWh] are calculated in the function 'energeticCalculations'. The final SOC is instead taken from the vector of the SOC time profile, contained in 'prof_DP_WithPenalties'.

```
[fuelConsumption_DP_WithPenalties, fuelEconomy_DP_WithPenalties,
totalEnergyConsumption_DP_WithPenalties] = energeticCalculations(mission_WLTP,
prof_DP_WithPenalties, veh) % [kg],[L/100km],[kWh]
```

```
fuelConsumption_DP_WithPenalties = 0.7664
fuelEconomy_DP_WithPenalties = 4.1970
totalEnergyConsumption_DP_WithPenalties = 9.2333
```

```
finalSOC_DP_WithPenalties = prof_DP_WithPenalties.battPrf.battSOC(end) % [p.u.]
```

```
finalSOC_DP_WithPenalties = 0.5997
```

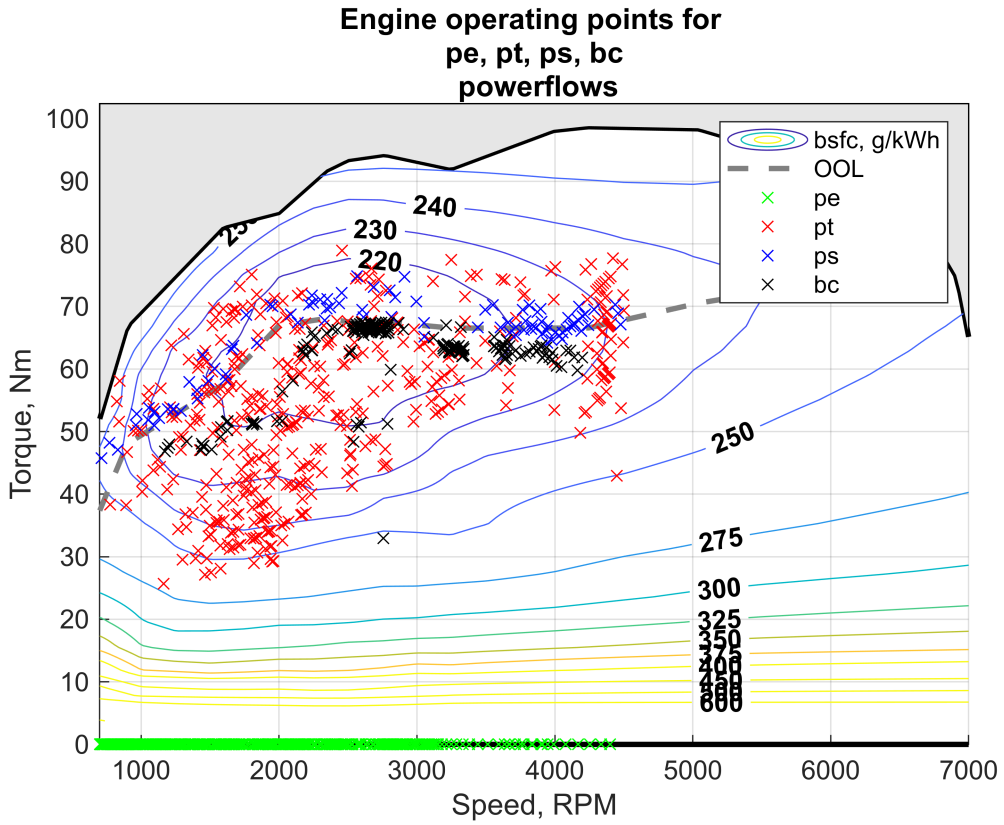
Post-processing with driveability

Similarly to the previous section, the behaviour of the solution is inspected by means of significative charts. By the way, the differences between the two strategies are hardly noticeable using only the machines maps and the powerflow; for this reason, the following charts are reported below for the sake of completeness, while a deeper comment on the two behaviours comparison is reported in the 'Results analysis' section.

ICE map:

The engine operating points are shown together with the brake specific fuel consumption iso lines; the points highlight the torque-split mode selected by the controller.

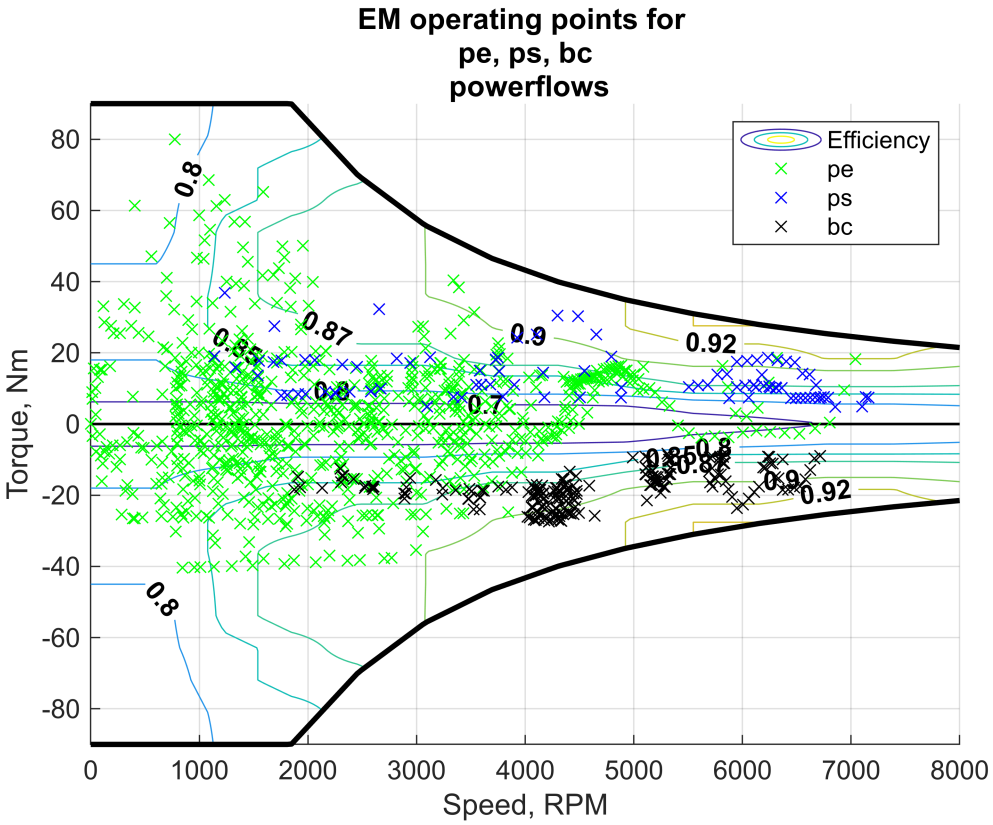
```
% Engine operating points
engMapWithPF(veh.eng, prof_DP_WithPenalties, "bsfc", 'all')
```



EM map:

The EM operating points are shown in the map, together with the efficiency iso lines.

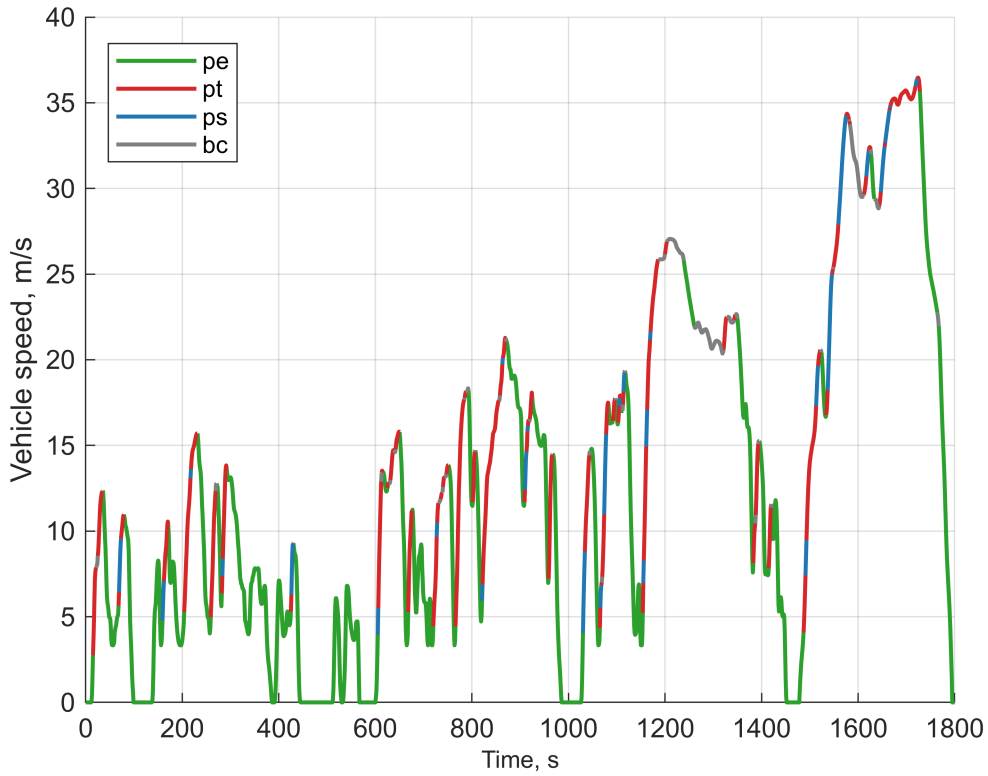
```
% EM operating points
emMapWithPF(veh.em, prof_DP_WithPenalties, ["pe", "ps", "bc"]);
```



Power flows:

The power flows chart show the vehicle speed profile, highlighting the operating mode selected in each instant.

```
% Power flows
powerProfiles(prof_DP_WithPenalties, 'all');
```



Save results

The results of the two analysis are saved below; two files are created, one containing the results of the simple dynamic programming, the other with the results of the problem involving driveability constraints.

```
% Store results
save("results_DP.mat", "prof_DP", "fuelConsumption_DP", "fuelEconomy_DP",
"finalSOC_DP", "gearShiftAvg", "engineStartAvg")
save("results_driv.mat", "prof_DP_WithPenalties",
"fuelConsumption_DP_WithPenalties", "fuelEconomy_DP_WithPenalties",
"finalSOC_DP_WithPenalties", "gearShiftAvgWithPenalties",
"engineStartAvgWithPenalties")
```

- `'prof_'`, `'fuelConsumption_'` , `'fuelEconomy_'`, `finalSOC_'` are the same quantities of the previous projects, calculated in the previous sections;
- `'gearShiftAvg_'` is the average number of gear shifts per minute in the whole mission;
- `'engineStartAvg_'` is the average number of engine starts per minute in the whole mission.

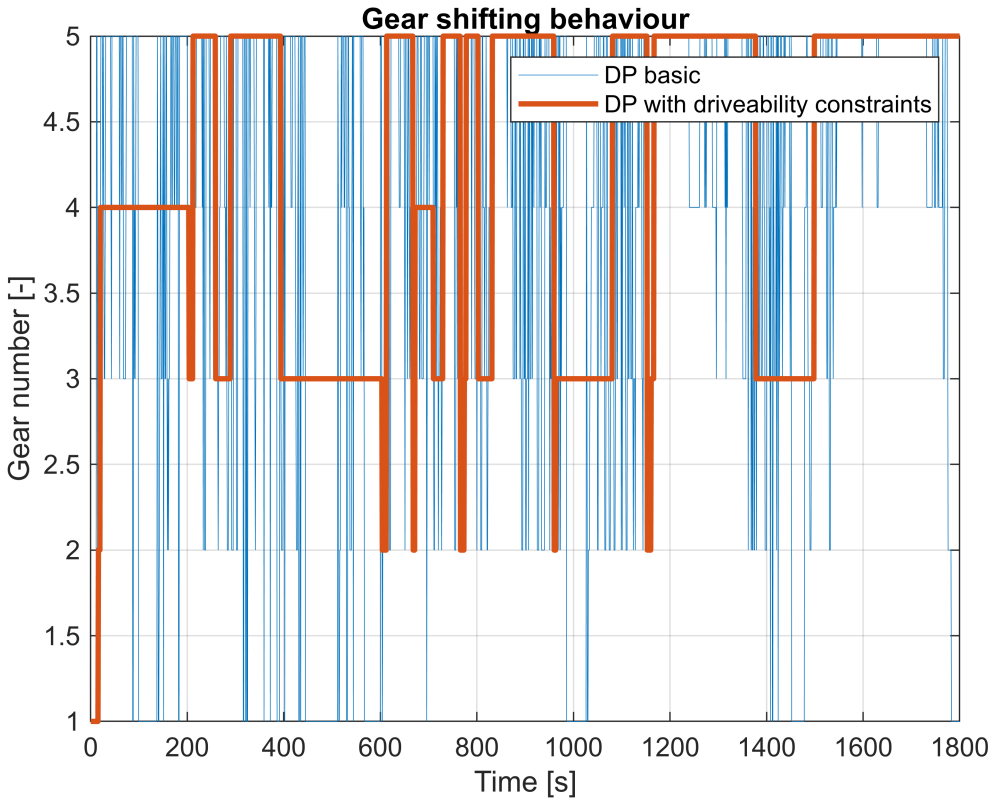
Results analysis

Effect of driveability constraints

In the following, the two different strategies calculated in this project are compared; to start, the two most significant aspects that differenciate the two cases are put in comparison: the gear shifting logic and the engine switch ons.

For this reason, the first diagram shown is the gear number evolution for the two solutions:

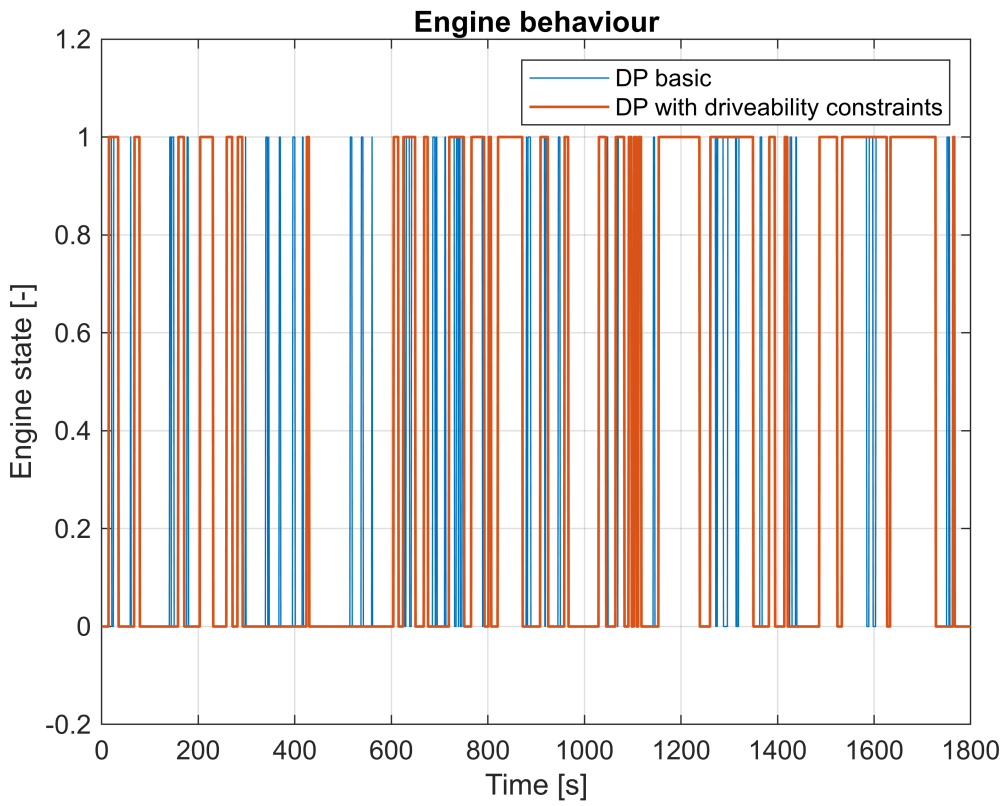
```
% gear number comparison
figure
plot(time, prof_DP.vehPrf.gearNumber, LineWidth=0.25); hold on, grid on
plot(time, prof_DP_WithPenalties.vehPrf.gearNumber, LineWidth=2)
xlabel('Time [s]')
ylabel('Gear number [-]')
title('Gear shifting behaviour')
legend('DP basic','DP with driveability constraints')
```



As already found from the results of the calculations, the control strategy with the driveability constraints allows to significantly reduce the number of the gear shifts throughout the cycle. The same consideration is carried out on the engine state (1=ON, 0=OFF):

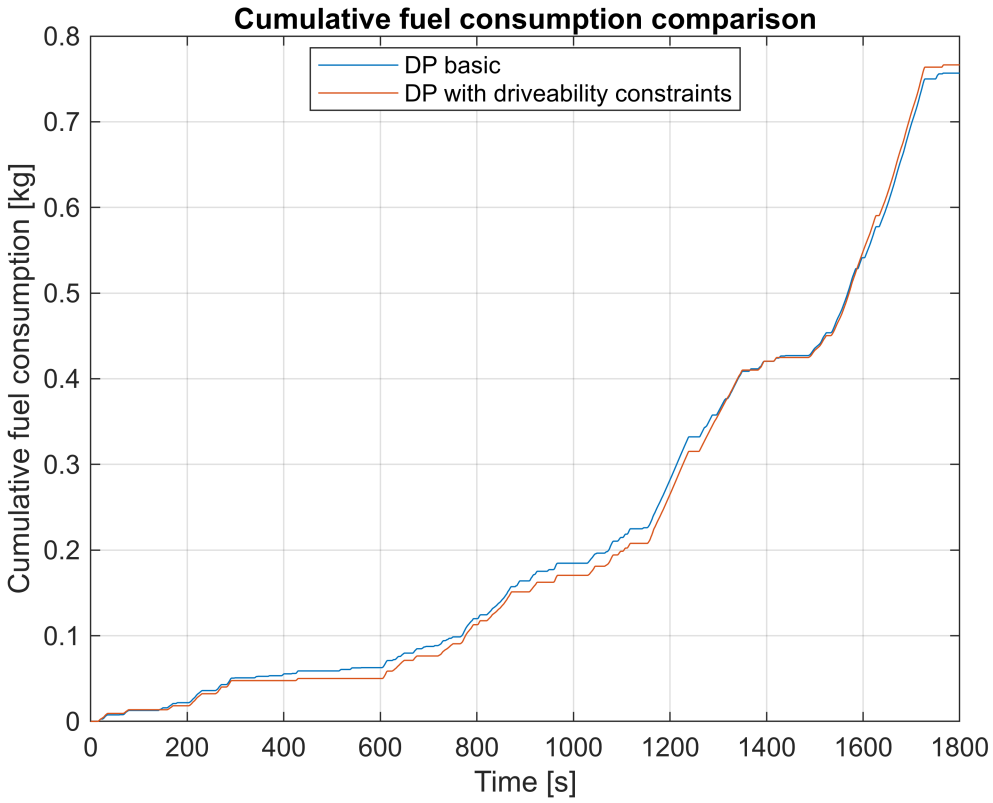
```
% Engine switch on comparison
figure
plot(time, prof_DP.engPrf.engineState, LineWidth=0.5); hold on, grid on
plot(time, prof_DP_WithPenalties.engPrf.engineState, LineWidth=1)
xlabel('Time [s]')
ylabel('Engine state [-]')
ylim([-0.2 1.2])
```

```
title('Engine behaviour')
legend('DP basic','DP with driveability constraints')
```



Also in this case, even if a bit less evident, it can be observed that the engine switch ons (passage from 0 to 1) are reduced with respect to the basic case; the solution found implementing the constraints on driveability is expected to be worse than the basic in terms of fuel consumption since, with respect to the global optimal solution, some modifications have been introduced. To quantify this degradation, the different cumulative fuel consumption are shown:

```
% Cumulative fuel consumption comparison
figure
fc_DP = cumtrapz(time, prof_DP.engPrf.fuelFlwRate)/1000;   % [kg]
fc_DP_driv = cumtrapz(time, prof_DP_WithPenalties.engPrf.fuelFlwRate)/1000;    %
[kg]
plot(time, fc_DP), hold on
plot(time, fc_DP_driv), grid on
xlabel("Time [s]");
ylabel("Cumulative fuel consumption [kg]")
title('Cumulative fuel consumption comparison')
legend('DP basic','DP with driveability constraints', Location='best')
```



Even if apparently the second solution has a lower consumption, in the first part of the cycle, the basic one is able to guarantee a lower value for the entire mission, being able to identify the global optimum. The overall difference, in pecentage, is given:

```
% Pecentage degradation in fuel consumption due to driveability
Delta_fc = ((fuelConsumption_DP_WithPenalties-fuelConsumption_DP)/
fuelConsumption_DP)*100 % [%]
```

```
Delta_fc = 1.2727
```

The result is quite appreciable, since a significant improvement in driveability is obtained, impacting less than 1,3% on the fuel consumption.

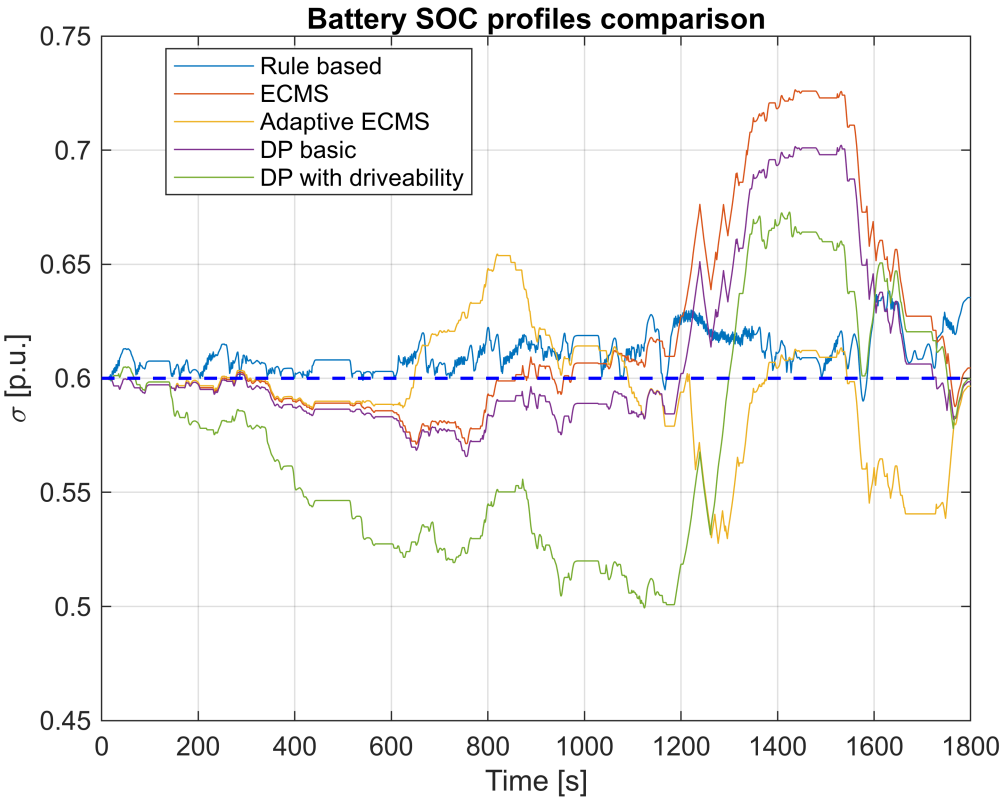
Comparison between all the controllers implemented

To draw some conclusions of the work carried out in the different projects, the main indicators of the controllers behaviour are compared in this section; for each one of the previous projects, one strategy is taken as reference for the global comparison:

- Rule based controller with extra feature for **Project 1**;
- ECMS for **Project 2**;
- Time based adaptive ECMS for **Project 3**;

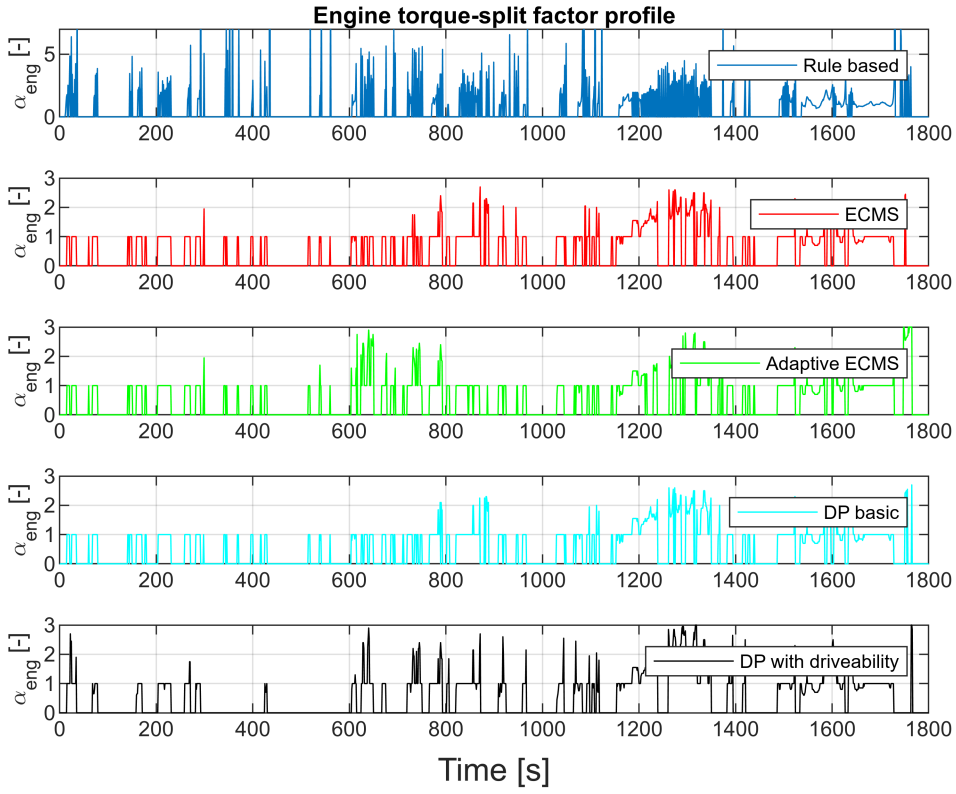
The quantities used for the comparison are the battery SOC evolution, the control parameter α_{eng} and the cumulative fuel consumption behaviour, while the refencence mission is the WLTP.

```
% Loading previous projects results
results_ExtraFeature =  load('resultsExtraFeature.mat');
results_ECMS = load('results_ecms.mat');
results_aECMS_time = load('results_adaptiveECMS_time.mat');

figure
plot(time, results_ExtraFeature.prof.battPrf.battSOC), hold on
plot(time, results_ECMS.prof.battPrf.battSOC), grid on
plot(time, results_aECMS_time.prof_WLTP_time.battPrf.battSOC)
plot(time, prof_DP.battPrf.battSOC)
plot(time, prof_DP_WithPenalties.battPrf.battSOC)
plot(time,0.6*ones(size(time)),LineStyle="--",Color="b",LineWidth=1.2)
xlabel("Time [s]"); ylabel("\sigma [p.u.]")
title('Battery SOC profiles comparison')
legend('Rule based','ECMS','Adaptive ECMS','DP basic','DP with
driveability',location ='best')
```

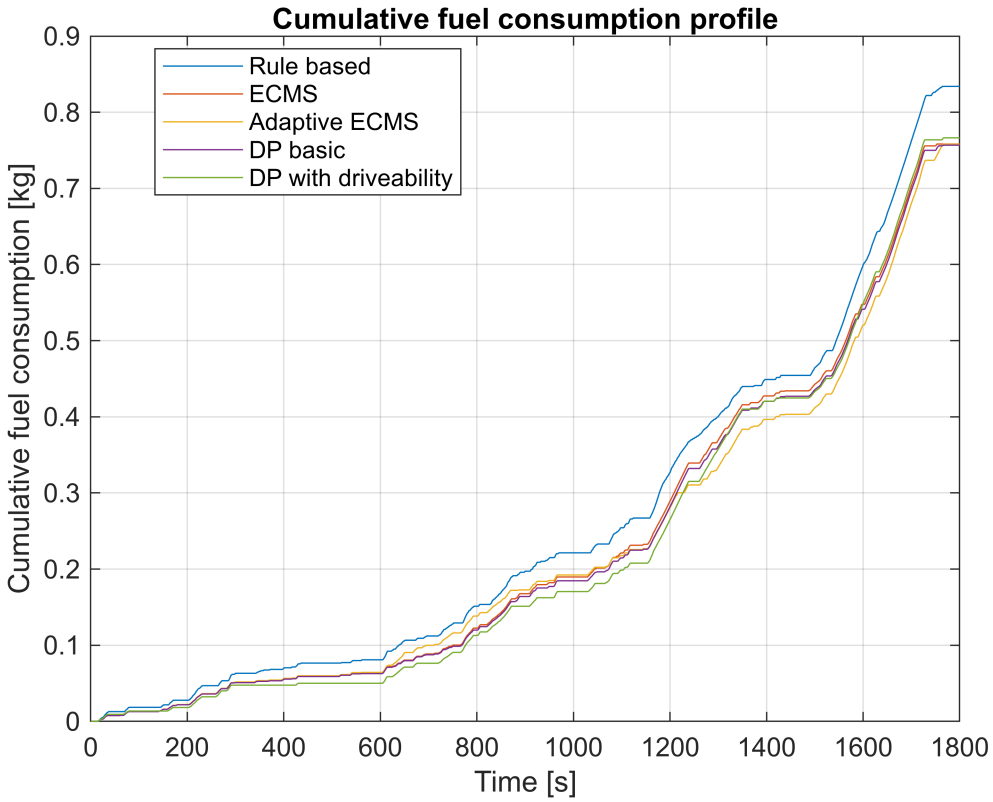



```
% Engine torque-split factor profile
figure
t = tiledlayout(5,1);

ax1 = nexttile;
plot(time, results_ExtraFeature.prof.vehPrf.engAlpha), grid on
ylabel("\alpha_{eng} [-]")
title('Engine torque-split factor profile')
legend('Rule based')
```

```
xlim([0, 1800]), ylim([0,7])

ax2 = nexttile;
plot(time, results_ECMS.prof.vehPrf.engAlpha, Color="r"), grid on
ylabel("\alpha_{eng} [-]")
legend('ECMS')
xlim([0, 1800]), ylim([0,3])

ax3 = nexttile;
plot(time, results_aECMS_time.prof_WLTP_time.vehPrf.engAlpha, Color="g"), grid on
ylabel("\alpha_{eng} [-]")
legend('Adaptive ECMS')
xlim([0, 1800]), ylim([0,3])

ax4 = nexttile;
plot(time, prof_DP.vehPrf.engAlpha, Color="c"), grid on
ylabel("\alpha_{eng} [-]")
legend('DP basic')
xlim([0, 1800]), ylim([0,3])

ax5 = nexttile;
plot(time, prof_DP_WithPenalties.vehPrf.engAlpha, Color="k"), grid on
ylabel("\alpha_{eng} [-]")
legend('DP with driveability')
xlim([0, 1800]), ylim([0,3])

xlabel(t, "Time [s]", 'FontSize', 12)
linkaxes([ax1 ax2 ax3 ax4], 'x')
xlim([0, 1800]), ylim([0,3])
```




```
figure
fc_rb = cumtrapz(time, results_ExtraFeature.prof.engPrf.fuelFlwRate)/1000;   % [kg]
fc_ecms = cumtrapz(time, results_ECMS.prof.engPrf.fuelFlwRate)/1000;    % [kg]
fc_Aecms = cumtrapz(time, results_aECMS_time.prof_WLTP_time.engPrf.fuelFlwRate)/
1000;    % [kg]
plot(time, fc_rb), hold on
plot(time, fc_ecms), grid on
plot(time, fc_Aecms)
plot(time, fc_DP)
plot(time, fc_DP_driv)
xlabel("Time [s]"); ylabel("Cumulative fuel consumption [kg]")
title('Cumulative fuel consumption profile')
legend('Rule based','ECMS','Adaptive ECMS','DP basic','DP with
driveability',location ='best')
xlim([0, 1800])
```



Interestingly, the behaviour of the basic versions of the ECMS and the DP solutions are very similar for all the three quantities analyzed. By the way, this is partially justified by the Pontryagin's principle, according to which the ECMS, despite being a local minimization strategy, is able to reach a solution that is very close to the global optimum. In addition, all the strategies involving the DP or the ECMS concepts, even with some modifications as in the case of adaptive ECMS and DP with driveability constraints, show quite similar results in terms of SOC behaviour and fuel consumption. The main results of the controllers are summarized below:

	Project 1 – Rule based	Project 2 - ECMS	Project 3 - AECMS (time)	Project 4 - DP (basic)	Project 4 - DP (driveability)
Fuel consumption [L/100km]	4.5670	4.1524	4.1510	4.1443	4.1970
Final battery SOC [p.u.]	0.6350	0.6045	0.5965	0.5985	0.5997
Total energy consumption [kWh]	9.9940	9.1298	9.1369	9.1199	9.2333

Tab 1: comparison of fuel consumption, final SOC and total energy consumption between different controllers

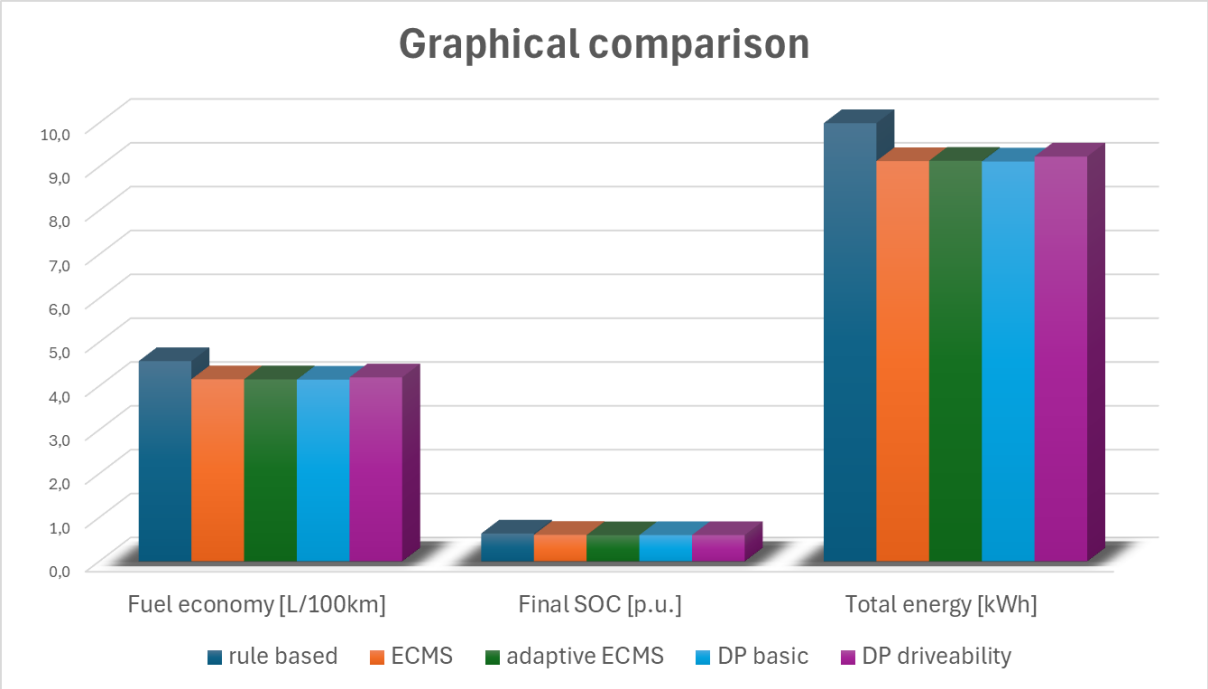


Fig 1: graphical comparison of controllers fuel consumption, final SOC and total energy consumption

Conclusions

It is observed that, with respect to the previously developed controllers, the dynamic programming solution is not suitable for online implementation on vehicles, since it is a global optimization method requiring the knowledge of the mission in advance; however, since its solution is the optimal one, given a driving cycle, it can be useful to understand how other controllers are behaving and how much margin of improvement is present. By converse, controllers from previous projects present severe downsides from the driveability point of view. A possible future development could be to implement the driveability constraints, that have proved to work on the DP, inside the logic of the adaptive ECMS controller, to provide a solution that combines the quasi optimal energy management of the ECMS with a better driveability.

In the following table, the main strenghts and weaknesses of each controller are highlighted:

	Project 1 – Rule based	Project 2 - ECMS	Project 3 - AECMS (time)	Project 4 - DP (basic)	Project 4 - DP (driveability)
Energy economy					
Controlled charge sustaining					
Online implementation suitability					
Driveability					

Tab 2: Summary of main controllers features

The judgement parameters selected are motivated as follows:

- **Energy economy:** the controller is able to provide a low value of total energy consumption;
- **Controlled charge sustaining:** the controller provides a final SOC close to the initial one, as well as a bounded behaviour throughout the entire mission, with active strategies to counteract unexpected SOC variations: in this way the controller can safely work also in missions that are different from the design one;
- **Online implementation suitability:** the calculations for the choice of the control variables can be computed in each instant directly on the vehicle, without the necessity to know future speed profiles or other non available data;
- **Driveability:** the system has a compliant behaviour with the customer's expectation, avoiding annoying frequent engine switch on and gear shifts.

Functions implementation

DPprocessing

The function analyzes the results of the dynamic programming algorithm, returning the number of average gear shifts and engine starts per minute, as well as a structure with the time evolution of the main vehicle, ICE, EM and battery variables. Function syntax:

[prof, gearShiftAvg, engineStartAvg] = DPprocessing(prob, mission)

Function inputs:

- **prob:** structure resulting from the application of 'DynaProg' tool;
- **mission:** data structure containing cycle speed [km/h], acceleration profile [m/s^2] and time vector [s] as well as the name of the cycle;

Function outputs:

- **prof:** structure with the time evolution of the main vehicle, ICE, EM and battery variables;
- **gearShiftAvg:** average gear shifts during the cycle [1/min];
- **engineStartAvg:** average engine switch ons during the cycle [1/min];

Function logics:

The function packs the additional outputs of 'prob' into the structure 'prof'. To do so, these are previously converted into monodimensional structures with the function structArray2struct. Then, the profiles of the gear number and the engine state are inspected to count the number of gear shifts and engine starts which, normalized on the cycle time, provide the averages values that are used to evaluate the system driveability.

```
function [prof, gearShiftAvg, engineStartAvg] = DPprocessing(prob, mission)
    % Additional outputs extraction
    engPrf = prob.AddOutputsProfile{1,1};
    emPrf = prob.AddOutputsProfile{1,2};
    battPrf = prob.AddOutputsProfile{1,3};
    vehPrf = prob.AddOutputsProfile{1,4};

    % Transform the non-scalar struct containing time profiles into scalar
    % structs; this makes their manipulation easier.
    engPrf = structArray2struct(engPrf);
    emPrf = structArray2struct(emPrf);
    battPrf = structArray2struct(battPrf);
    vehPrf = structArray2struct(vehPrf);

    % Pack profiles into a single structure
    prof.engPrf = engPrf;
    prof.emPrf = emPrf;
    prof.battPrf = battPrf;
    prof.vehPrf = vehPrf;

    % Gear shifts and engine switch on count
    gearShifts = 0;
    engSwitchOn = 0;
    for i = 2:length(mission.time_s)
        if prof.vehPrf.gearNumber(i) ~= prof.vehPrf.gearNumber(i-1)
            gearShifts = gearShifts + 1;
        end
        if prof.engPrf.engineState(i-1) == 0 && prof.engPrf.engineState(i) == 1
            engSwitchOn = engSwitchOn + 1;
        end
    end
    gearShiftAvg = (gearShifts/(mission.time_s(end)))*60;
    engineStartAvg = (engSwitchOn/(mission.time_s(end)))*60;
end
```

energeticCalculations

The following function performs the calculation of cumulative fuel consumption, fuel economy and global energy conumption for a given driving mission. Function sintax:

Function inputs:

- **mission:** data structure containing cycle speed [km/h], acceleration profile [m/s^2] and time vector [s] as well as the name of the cycle;
- **prof:** structure containing the fuel flow rate and the battery voltage and current time profiles;
- **veh:** structure containing the main vehicle parameters and the fuel lower heating value [J/kg].

Function outputs:

- **fuelConsumption:** cumulative cycle fuel consumption [kg];
- **fuelEconomy:** cycle fuel economy [L/100km];
- **totalEnergyConsumption:** sum of electric and fuel energy used for the cycle [kWh].

Function logics:

The cumulative fuel consumption is calculated by integrating the fuel mass flow rate over the mission time; then, dividing by the cycle distance, the fuel economy is calculated. Eventually the energy used during the cycle is retrieved by summing the electrical energy with the fuel consumption one. It is also taken into account the battery charge/discharge with respect to the initial SOC; in fact the electrical energy consumption 'elEnergyConsumption' is negative when the final SOC is higher than the initial one, while it is positive viceversa: this comes as a consequence of integrating the battery power over the mission time.

```
function [fuelConsumption, fuelEconomy, totalEnergyConsumption] =
energeticCalculations(mission, prof, veh)
    fuelConsumption = trapz(mission.time_s, prof.engPrf.fuelFlwRate)/1000; %
Cumulative fuel consumption [kg]
    vehDist = trapz(mission.time_s, mission.speed_kmh/3.6); % Total cycle distance
[m]
    fuelEconomy = (fuelConsumption*10^5) / (veh.eng.fuelDensity*vehDist); % Fuel
consumption [L/100km]
    elEnergyConsumption = trapz(mission.time_s,
prof.battPrf.battVolt.*prof.battPrf.battCurr); % battery electrical energy [J]
    fuelEnergyConsumption = fuelConsumption*veh.eng.fuelLHV; % [J]
    totalEnergyConsumption = (elEnergyConsumption + fuelEnergyConsumption)/
1000/3600; % [kWh]
end
```