Assignment #3: A-ECMS

Table of Contents

Group information....1
Load the cycle and vehicle data....1
Test the A-ECMS on the test cycles....3
Simulation for wide range of Kp....5
  Optimal Kp research....6
Result analysis....7
  Overall behavior analysis....7
    Kp vs SOC deviation....7
    Kp vs fuel consumption....8
  Simulation run for selected Kp....9
  Kp sensibility analysis....10
    Equivalence factor vs Time....10
    SOC vs Time....11
    Engine map....14
    Battery Power vs Time....15
  Kp variation result analysis....16
T update influence analysis....17
  Equivalence factor vs Time....17
  SOC vs Time....18
    Considerations on T_update influence....22
  Additional analysis on Kp - Tupdate correlation....22
    Considerations on Kp - T_update correlation and influences....24
Conclusion....25
Simulation Loop function....25
The A-ECMS controller....27
  Controller logic....28
  Controller implementation....28

Group information

Group number: 12

Students:

- Emanuele Landolina, s349706
- Giuseppe Maria Marchese, s348145
- Walter Maggio, s343988

Load the cycle and vehicle data

In this initial phase, the necessary folders are added to the MATLAB path to ensure that all required functions and tools are available for the script.

```
clear all
close all
clc
tic
addpath("utilities");
addpath("models");
addpath("data");
addpath("additional utilities");

% s_0 is the value of the equivalence factor obtained in Lab 02
s_0 = 2.6367;
bestConsumption = [4.455 6.686 3.2336 4.1861];

%load vehicle data
vehData_raw = load("vehData.mat");

%load mission data
missions_file = ["WLTP.mat","AMDC.mat","AUDC.mat","TurinTest.mat"];
missions_name = ["WLTP","AMDC","AUDC","TurinTest"];

for i = 1:length(missions_file)
    if missions_name(i) == "AUDC"
        AUDC = load(missions_file(i));
        missions.(missions_name(i)).time_s = 1:1:length(AUDC.speed_km_h)*5;
        missions.(missions_name(i)).speed_km_h = repmat(AUDC.speed_km_h,1,5);
        missions.(missions_name(i)).acceleration_m_s2 =
repmat(AUDC.acceleration_m_s2,1,5);
        clear AUDC;
    else
        missions.(missions_name(i)) = load(missions_file(i));
    end
end

%vehicle characteristic [Group 12]
engine_power = 66000;     %[W]
E_machine_power = 70000; %[W]
battery_energy = 945;     %[Wh]

%Scaled vehicle data
vehData = scaleVehData(vehData_raw,E_machine_power,engine_power,battery_energy);
```

This initial section of the code handles the setup and data loading required for simulating the hybrid vehicle’s performance. Several directories containing utility scripts, models, and data files are added to the MATLAB path, as well as the 'additional utilities' that contains the functions of all the plots we will discuss in this report.

The **equivalence factor** `s_0`, previously calculated in Lab 02, is defined along with a vector of optimal fuel consumption values for each driving missions that will be analyzed.

Vehicle data are loaded from `.mat` files and organized into the `missions` struct, which stores the complete information for each driving cycle. In the case of the AUDC cycle, speed and acceleration data are repeated to extend the profile duration before being stored in the struct.

The **vehicle’s characteristics** (engine power, electric machine power and battery energy) provided for the group are inserted and then the raw vehicle data are scaled to align with the specific parameters assigned to the group.

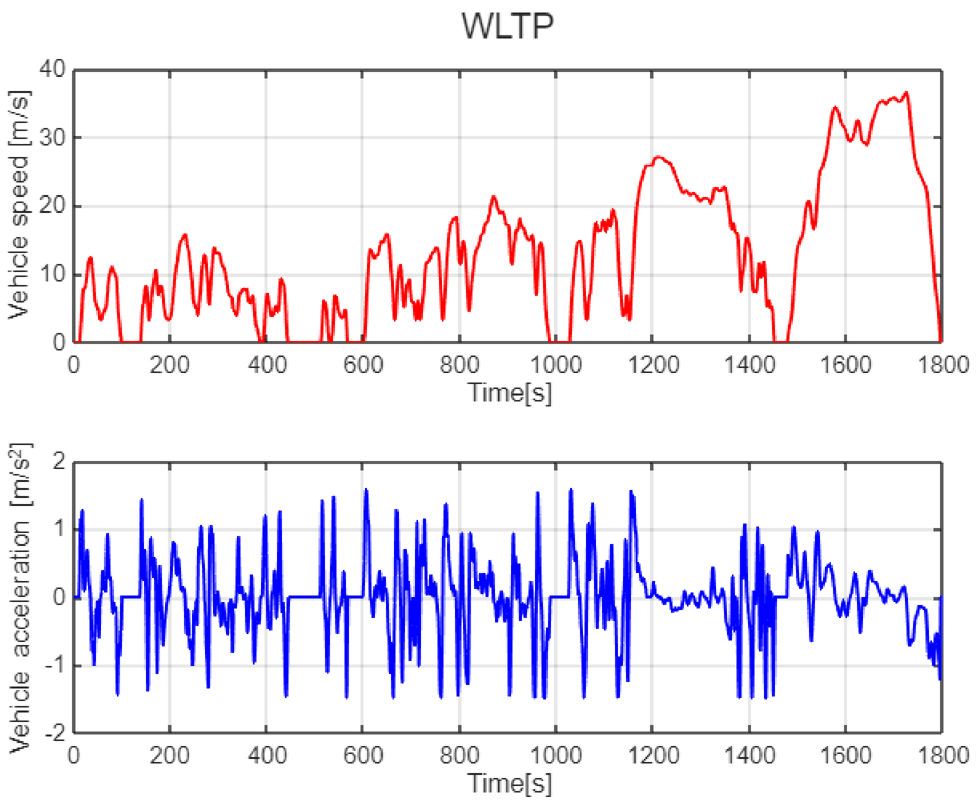
Test the A-ECMS on the test cycles

To test the adaptive ECMS the following four test cycles are used:

- The WLTP cycle.
- The Artemis Motorway cycle (AMDC).
- The Artemis Urban cycle (AUDC), repeated five times.
- The Turin Test cycle.

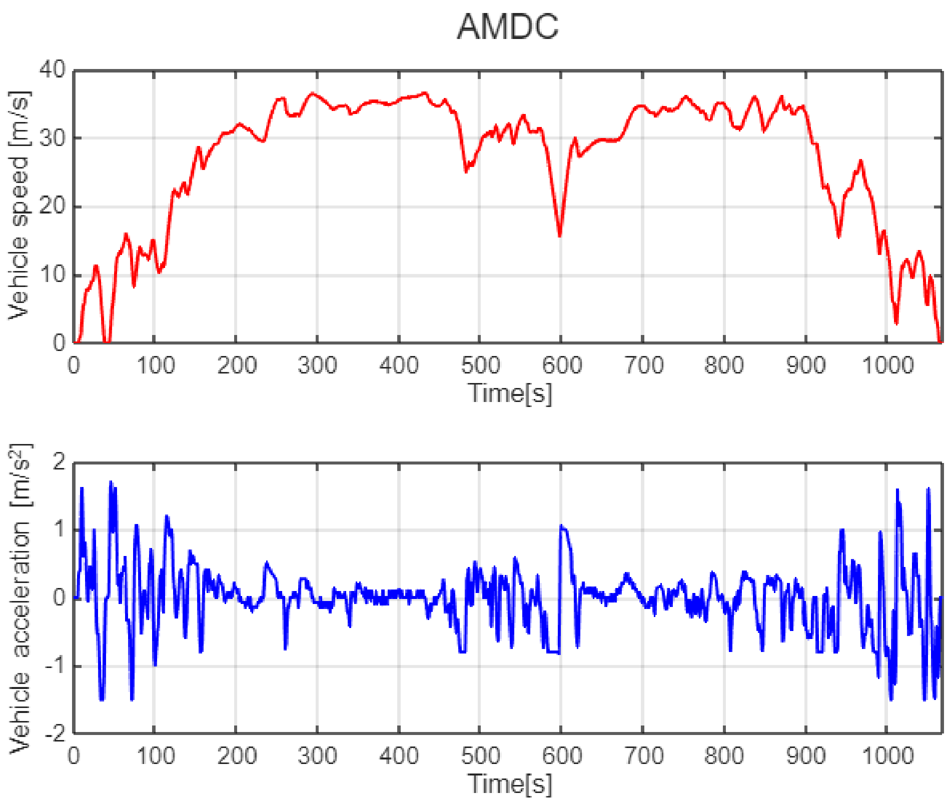
They are shown and briefly described, using plots, in terms of speed and acceleration vs time.

```
plot_function(missions,'WLTP');
```



The **WLTP** cycle is divided into several parts. The first phase simulates urban driving, characterized by low speeds and not so frequent accelerations, resulting in a relatively low power demand. As the cycle progresses into the extra-urban phase, both speed and acceleration increase, leading to a higher power requirement. The final phase represents highway driving, which is the most demanding in terms of power due to high speeds and rapid accelerations. It is noticeable that in none of the phases are present very high dynamics, with few stops and so not much energy recovered from braking manouvres.

```
plot_function(missions,'AMDC');
```



The **Artemis Motorway driving cycle** is a relatively short cycle with respect to the other ones. It emphazizes high speeds, running more than half of the cycle almost at the same speed.

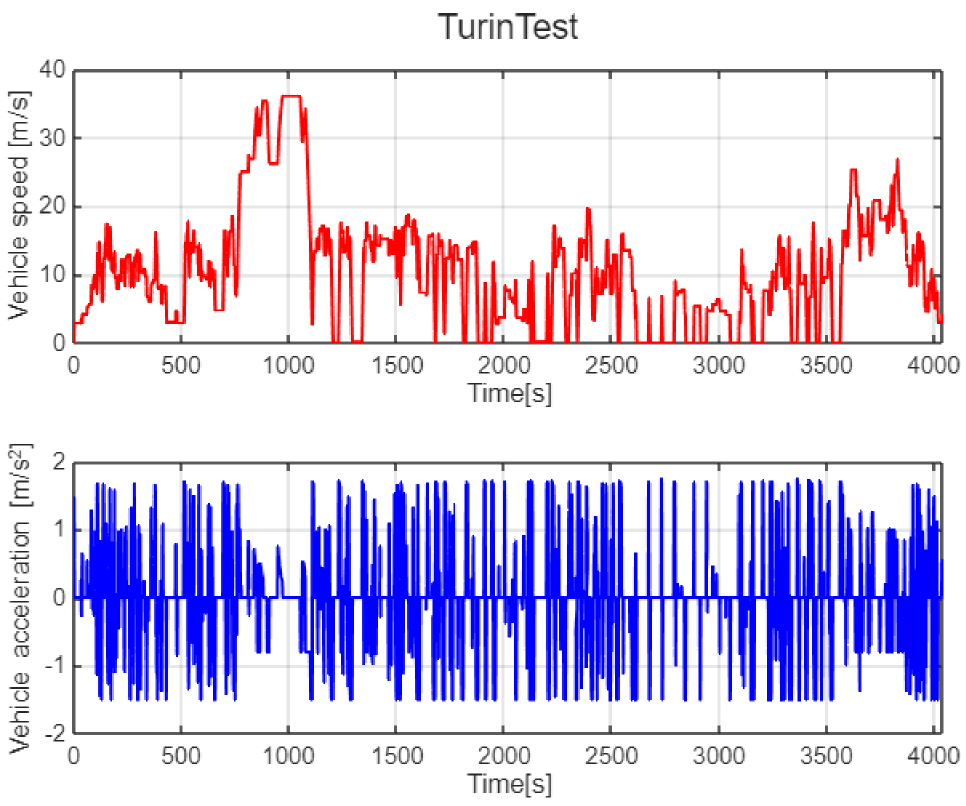
It shows a simmetrical behavior from the middle to the start/end part of the cycle. At the extremities there are the highest acceleration and deceleration phase, where the vehicle reaches the speed peaks (over 35 m/s). Then the vehicle maintains almost a constant speed, except for a brief phase in the central portion where it slighlty slows down and some acceleration/deceleration is visible. The dynamics are really slow, with almost no braking manouvres and consequently very little energy recovered through the brakes.

```
plot_function(missions,'AUDC');
```

The **Artemis Urban Driving Cycle** has been repeated 5 times, so it is way longer with respect to other cycles. It accounts for the urban zone, with very low speed (max 50 km/h) and very aggressive and frequent accelerations/decelerations.

In this cycle the dynamics are very fast, indeed the vehicle frequently stops and then starts again. So the power request and the energy from braking are very high with respect to the other cycles, making this a very challenging one.

```
plot_function(missions,'TurinTest');
```



The Turin test is a synthetic cycle, based on the city of Turin, so it mainly accounts for the urban segment and extra-urban segment.

The speed is almost everywhere bounded between 10 and 20 m/s, except for a brief highway segment where the speed approaches 37 m/s end the end of the cycle where it is slightly higher. Many stop/start events are present in the second half of the cycle, useful to analyze the performance of the hybrid system in terms of efficiency and fuel consumption. The acceleration values are the highest among all the cycles used on this assignment, translating in very high power requests, followed decelerations equally frequent and aggressive.

Simulation for wide range of Kp

To better understand how the proportional gain Kp influences system behavior and hybrid vehicle performance, a simulation is conducted across a range of Kp values. Specifically, the chosen range spans from 0.05 to 10, with an increment of 0.05. This choice reflects a **trade-off between accuracy and computational cost**:

- The **lower limit of 0.05** is selected to ensure that the effect of SOC deviation on the update of the equivalence factor is noticeable, while still maintaining a very smooth variation in the equivalence factor.
- The **upper limit of 10** is set to capture the system’s response when the SOC deviation has a strong influence on the equivalence factor update. However, it is kept low enough to avoid excessively rapid or drastic changes.
- A **step size of 0.05**, determined through several test runs, represents the optimal compromise between capturing performance trends with sufficient resolution and limiting simulation time.

To accelerate the simulation, **MATLAB’s Parallel Computing Toolbox** is employed. Specifically, the `parfor` function allows parallel execution, running a simulation for each Kp value on a separate core. For instance, with an 8-core processor, eight Kp values can be simulated simultaneously. Thanks to this optimization, the complete simulation of 200 Kp values is completed in approximately 25 minutes.

The results are **saved in two `.mat`** files: one containing the **`prof structure`** and the other **storing** the **SOC deviation** for each simulation. This allows the data to be reused without rerunning the entire simulation, significantly speeding up the result analysis.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ATTENTION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% It is recommended not to run this section, as it may take several %
% minutes to complete.                                              %
%                                                                   %
% REQUIREMENT: MATLAB’s Parallel Computing Toolbox                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Tupdate = 60;
% Kp_vect = 0.05:0.05:10;
% [tryDevTable,tryProf] = Kp_opt(s_0,missions,Tupdate,Kp_vect,vehData);
%
% save("tryDevTable.mat","tryDevTable")
% save("tryProf.mat","tryProf")
```

Optimal Kp research

To efficiently identify the best Kp values, a dedicated script has been developed.

The selected discriminant factor is the deviation of the State of Charge (SOC) between the initial and final values.

```
%load the result of dynamic programming
load("tryDevTable.mat")
load("tryProf.mat")

%Extract Kp vector
Kp_vect = tryDevTable.Kp';

% Initialize the vector
KpDevSum = zeros(1,length(Kp_vect));

% Compute the deviation sum between all the mission for each Kp
for i = 1:length(Kp_vect)
    devSum = 0;
    for k = 1:length(missions_name)
        devSum = devSum + abs(tryDevTable.(missions_name(k))(i));
    end
    KpDevSum(i) = devSum;
end

%take out the best Kp
[~, idx_min] = mink(KpDevSum, 15);
kp_opt_vect = Kp_vect(idx_min)
```

```
kp_opt_vect = 1×15
```

```
    0.6000    0.1000    0.6500    9.4000    8.6500    9.8000    8.6000    6.4500⋯
```

```
KpSelected = [0.60 5.45 9.4]; %manual selection for the comparison
KpSelIndex = [idx_min(1) idx_min(13) idx_min(4)];
profSel = [tryProf(KpSelIndex(1)); tryProf(KpSelIndex(2));
tryProf(KpSelIndex(3))];
```

The script computes the **sum** of the **absolute SOC deviations** across **all four cycles** for each `Kp` value, resulting in a vector where each row corresponds to a total SOC deviation and each column to a specific Kp value.

Subsequently, the **`mink`** function is used to **extract** the indexes from the vector, of the amount of **`Kp`** values requested (15 in this case), corresponding to the **smallest total SOC deviations.** Then the values are stored in a separate vector.

While this method does not guarantee that all deviations are individually minimized, it provides a useful starting point for further analysis. By examining these selected values in greater detail, it becomes possible to better understand their behavior and identify the optimal `Kp` value.

At the end, three Kp values are manually selected from those obtained in the previous function. To evaluate the influence of the Kp value on the system, the selection includes a small, a medium, and a large value.

Result analysis

In this section, the results are analyzed using plots.

The **first part** focuses on gaining a **general understanding** of the vehicle's behavior as a function of the `Kp` variation over a wide range.

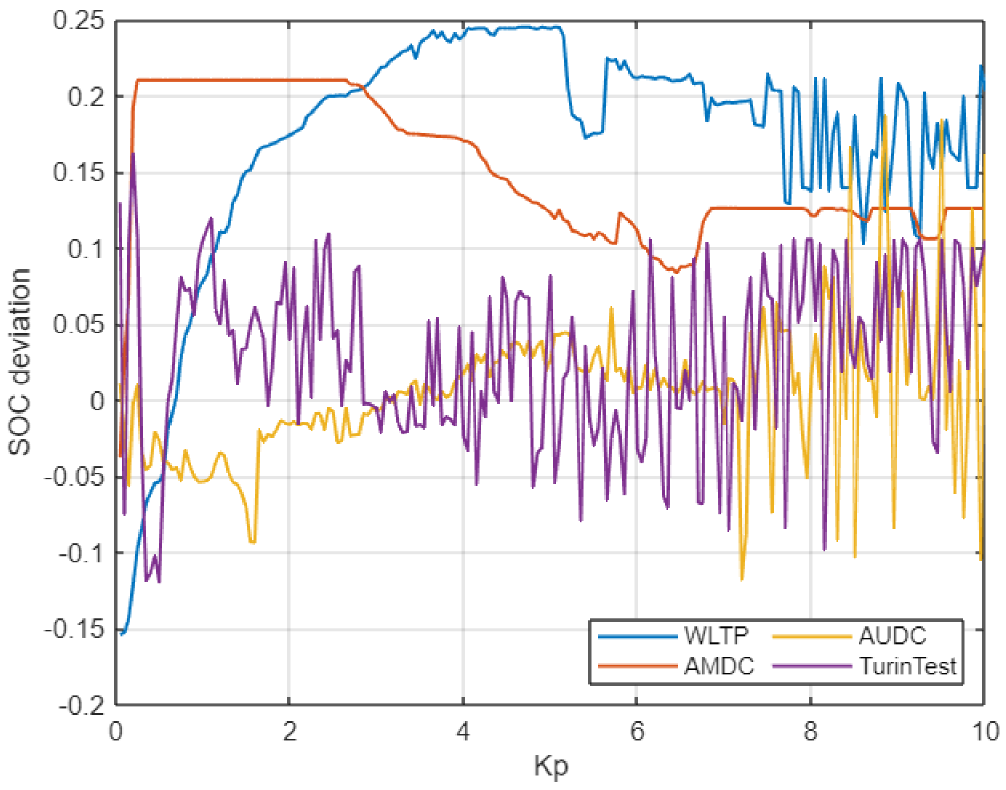
In the **second part**, a **comparison** is performed using **selected `Kp`** values, to gain deeper insights into how the magnitude of the parameter influences vehicle behavior.

Overall behavior analysis

Kp vs SOC deviation

This first plot aim is to show how the SOC deviation vary for all cycles in function of the `Kp`.

```
kp_soc_dev_plot(tryDevTable);
```



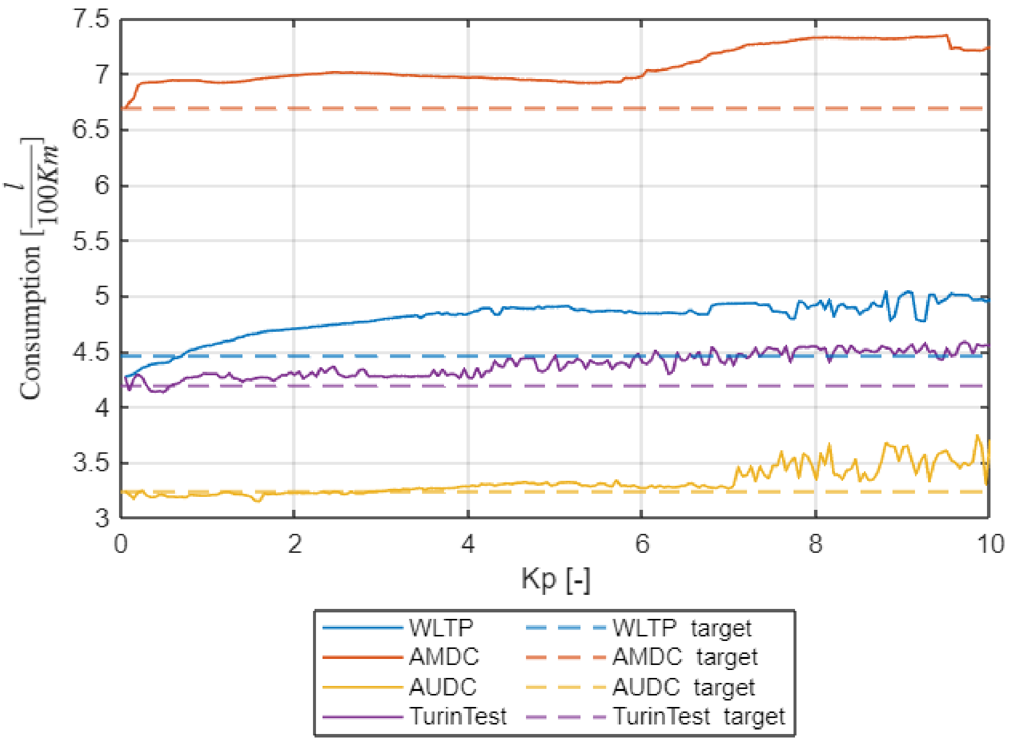
At first approximation, it is reasonable to state that finding a single `Kp` value that fits all driving cycles is very difficult, if not impossible. For instance, the **AMDC** and **WLTP** cycles clearly **demonstrate** that **achieving** proper operation in **charge-sustaining** mode **for both** simultaneously is **not feasible**. Optimizing the controller for one of the two cycles typically results in a deviation of at least 0.1 in the other. While this does not compromise the overall behavior of the controller, it already indicates a **failure** to maintain **charge-sustaining** mode **across all scenarios**.

Another important result emerging from this plot is that almost all cycles tend to stabilize around an average SOC deviation value at high `Kp`. However, this does not guarantee that the optimal `Kp` lies in that region. This observation suggests the need for further analysis, carried out in the second part of this section, to identify where the overall SOC deviation is actually minimized.

Kp vs fuel consumption

This plot **compares** the fuel **consumption** of each driving cycle evaluated **using** the previously developed **ECMS** (that computes `s_opt` knowing the mission in advance, offline) and the **A-ECMS** with a wide range of `Kp` values.

```
Kp_consumption_plot(Kp_vect,tryProf,bestConsumption,missions,vehData);
```



As can be observed, each cycle exhibits a distinct behavior, highlighting that there is **not** a **single `Kp`** value **suitable for all** the analyzed **missions**.

The **AUDC** cycle shows a wide range of suitable `Kp` values (from very small values up to 6.5), indicating that the A-ECMS is able to obtain fuel consumption values well comparable to the mission-aware ECMS. This can likely be attributed to the fact that the AUDC cycle involves only one consistent driving condition.

Conversely, the **WLTP** cycle (which includes a mix of urban, suburban, and highway segments) demonstrates limits on the performance of many `Kp` values. In this case, only one value (0.7) achieves a fuel consumption comparable to that of the standard ECMS.

The **AMDC** is comparable to the AUDC, showing a nearly linear trend in fuel consumption for different values of `Kp`. However, in this case the fuel consumption does not converge, remaining approximately parallel to the optimal level.

The **Turin test**, on the other hand, is comparable to the WLTP, as it also includes various driving conditions. As a result, it exhibits a wider range of suitable `Kp` values (approximately between 0.01 and 4), beyond which fuel consumption increases, diverging from the optimal target.

Simulation run for selected Kp

This simulation has been done to collect all the results of the selected `Kp` in order to perform a sensibility analysis.

```
% Initialize controller
Tupdate = 60;

%Simulation run
parfor i = 1:length(KpSelected)
```

```
    Kp = KpSelected(i);
    results_struct(i) = simulationLoop(s_0,missions,Tupdate, Kp,vehData);
end
```

Kp sensibility analysis

The main objective of this project is to (attempt to) identify a value of Kp that is suitable across different driving cycles, without knowing them in advance. The goal is to operate the vehicle in **charge-sustaining mode** using the designed **A-ECMS** implemented on a HEV.

As stated in the previous paragraphs, since no single value of Kp consistently achieves the desired objective across all conditions, it has been decided to select three representative values of Kp, within the range of 0 to 10, and compare them to extract some considerations.

From the previous section (**Optimal Kp research**), the following values were selected:

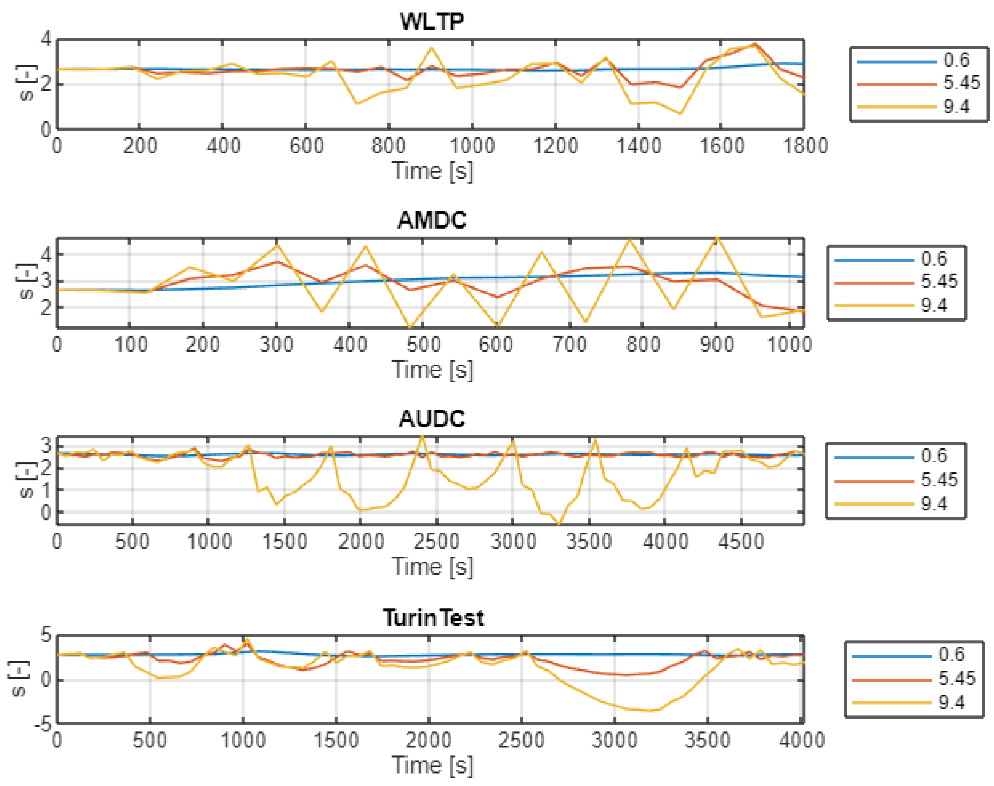
- **Kp = 0.60**, representing the **lower bound**;
- **Kp = 5.45**, an **intermediate** value;
- **Kp = 9.40**, representing the **upper bound**.

After that, several plots were analyzed to extract insights and assess the controller's behavior under the different conditions.

Equivalence factor vs Time

The plots show the evolution of the equivalence factor s under different Kp values across the four analyzed driving cycles.

```
eqFactorPlot_Kp(results_struct,KpSelected);
```



From the plots, it is clear that **increasing `Kp`** leads to **greater sensitivity** in the behavior of the equivalence factor.

This is consistent with theoretical expectations: higher values of Kp make the controller react more aggressively to deviations, resulting in noticeable fluctuations (especially in segments with high dynamics, such as motorway driving).

$$s_{n+1} = \frac{s_n + s_{n-1}}{2} + K_p \cdot [SOC_{REF} - SOC(t)]$$

A value of **`Kp=9.4`** corresponds to a controller that is **overly aggressive**, producing **strong** and **frequent oscillations** that may compromise overall energy management efficiency.

Conversely, a low value such as **`Kp=0.6`** results in a much **smoother response**. In this case, the equivalence factor evolves more **gradually** and remains **stable**, avoiding excessive oscillations and ensuring a more consistent trade-off between fuel and electric energy usage.

An intermediate value, such as **`Kp=5.45`**, tends to offer a **good compromise** between **responsiveness** and **stability**. These values generally perform well across all tested driving cycles, although **slight performance degradation** may still occur in more demanding conditions, such as those represented by the AMDC cycle.

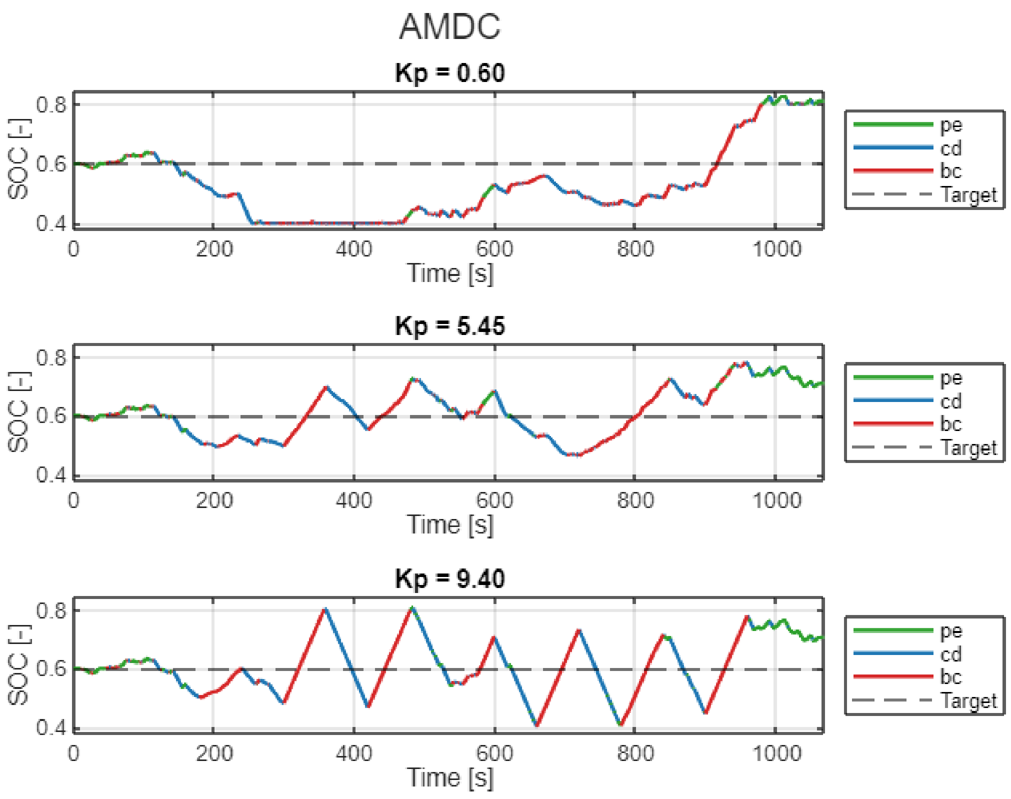
Based on these results, we conclude that the **optimal range** for `Kp` lies between **0** and **5**, with values **around 1** being **particularly robust** and preferable across a wide variety of driving conditions.

SOC vs Time

These plots show the variation of the SOC for every cycle for the selected `Kp`, in order to evaluate the difference in the controller behavior.

- **AMDC**

```
SOCwithPF(results_struct,"AMDC",KpSelected,"Kp");
```



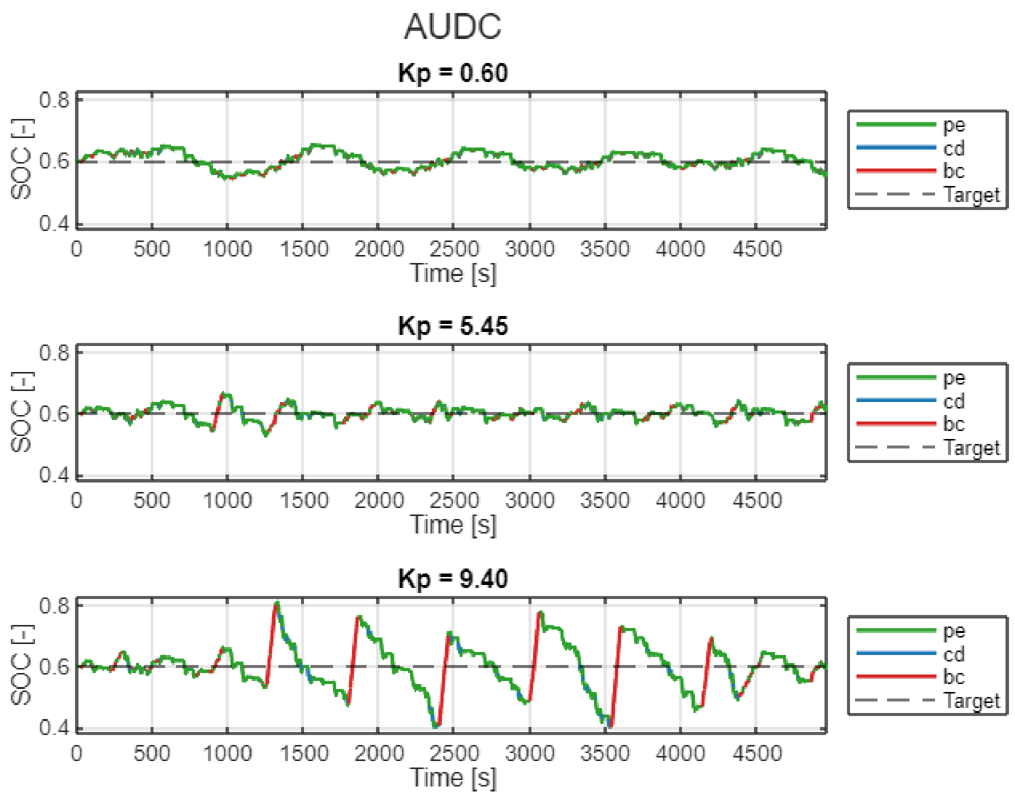
The **AMDC** cycle is very **challenging**, due to its high power requests. The consequence is that the selected values **fail** to **achieve** a proper **charge sustaining** behavior, with the final SOC deviation exceeding 0.1 in every case (the only one among the analyzed cycles).

It can be observed that **increasing `Kp` speeds up** the system **dynamics**, causing the **controller** to **oscillate more** between the SOC boundaries. On the other hand, it tends to **maintain** a **single operating mode** for **longer periods** compared to lower `Kp` values, where mode switching is more frequent. This obviously worsen the behavior, because the charge depleting and battery charge modes are very steep, indicating very high power requests.

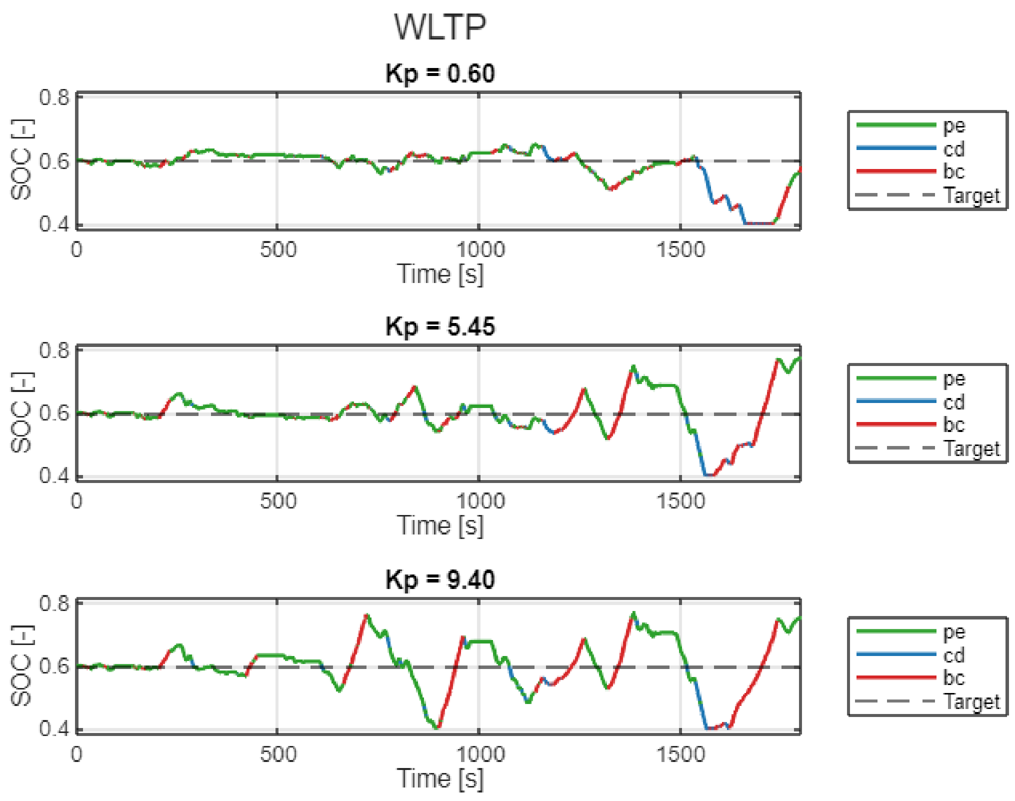
The best performance is clearly observed with the first two `Kp` values. Beyond that, it becomes a matter of **compromise**, **whether** a **larger final SOC deviation** from the desired value is **acceptable**, or if minimizing this deviation is a priority. If such a deviation is considered tolerable, the value Kp=0.6 exhibits the best overall behavior throughout the cycle, maintaining SOC within a consistent range without too much oscillations, enabling smooth charging and discharging of the battery.

- **AUDC**, **WLTP** & **Turin Test**

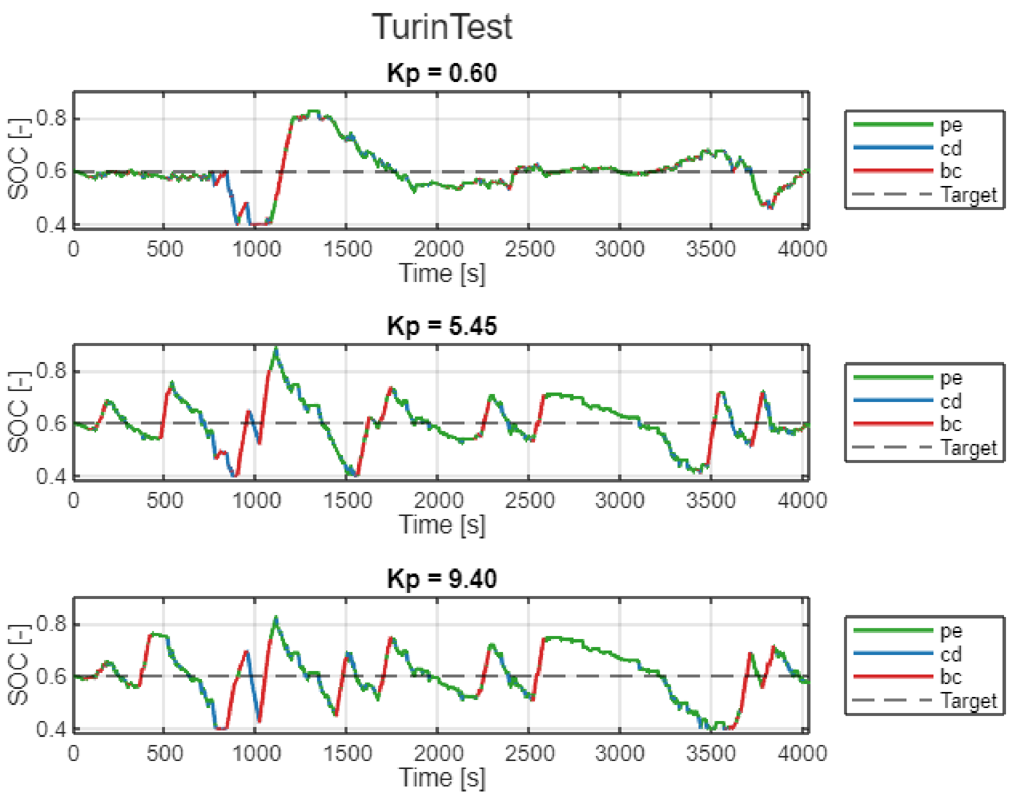
```
SOCwithPF(results_struct,"AUDC",KpSelected,"Kp");
```



```
SOCwithPF(results_struct,"WLTP",KpSelected,"Kp");
```



```
SOCwithPF(results_struct,"TurinTest",KpSelected,"Kp");
```



In the **AUDC**, **WLTP** and **Turin test** a **full charge-sustaining** behavior is **achieved** at least for one `Kp`.

As previously noted, **higher `Kp`** values lead to **increased oscillations**, with **steeper** SOC **trends** and often **failing** to achieve the **target**.

The **best** overall **SOC performance** appears to belong to the **`Kp=0.6`**, which minimizes oscillations and smooths the behaviors, confirming the idea that **lower `Kp`** values are the **best fit**. So it can be assumed that a smaller `Kp` is **better** in **mixing** the **charge depleting** and the **battery charge** modes, helping the controller manage the sub-urban and motorway segments with reduced number of oscillations between the SOC limits.

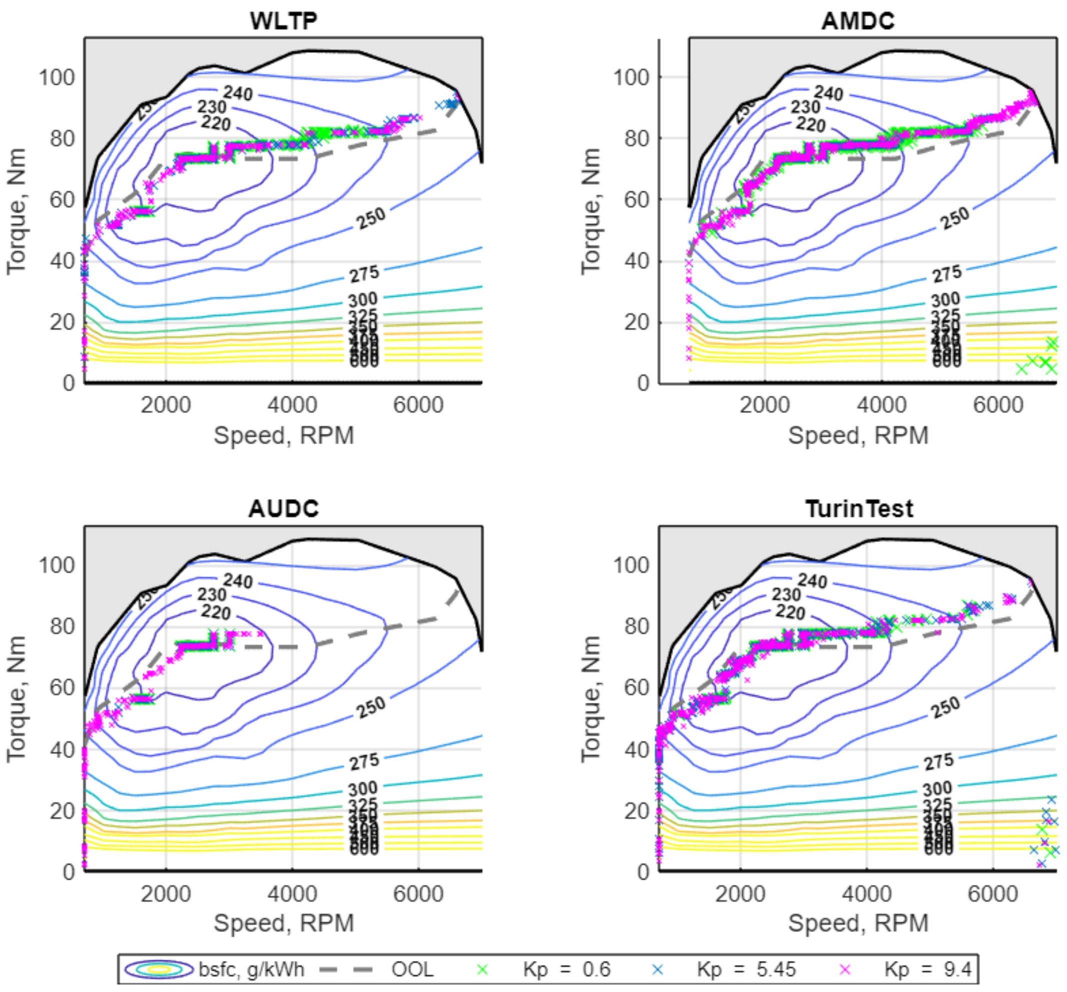
An important observation could be that the **controller works better in the urban area**, worsening in the sub-urban and even more in the motorway segments. This is likely **due to increased power requests** over a bigger durations and **decreased regenerative braking maneuvers**.

This last consideration has been a starting point for further investigations to improve the controller, discussed in paragraph T update influence.

Engine map

The following plot clearly illustrates the influence of the **`Kp`** parameter on engine management and its operating points.

```
mapWithPF(results_struct,KpSelected,"Kp",vehData);
```



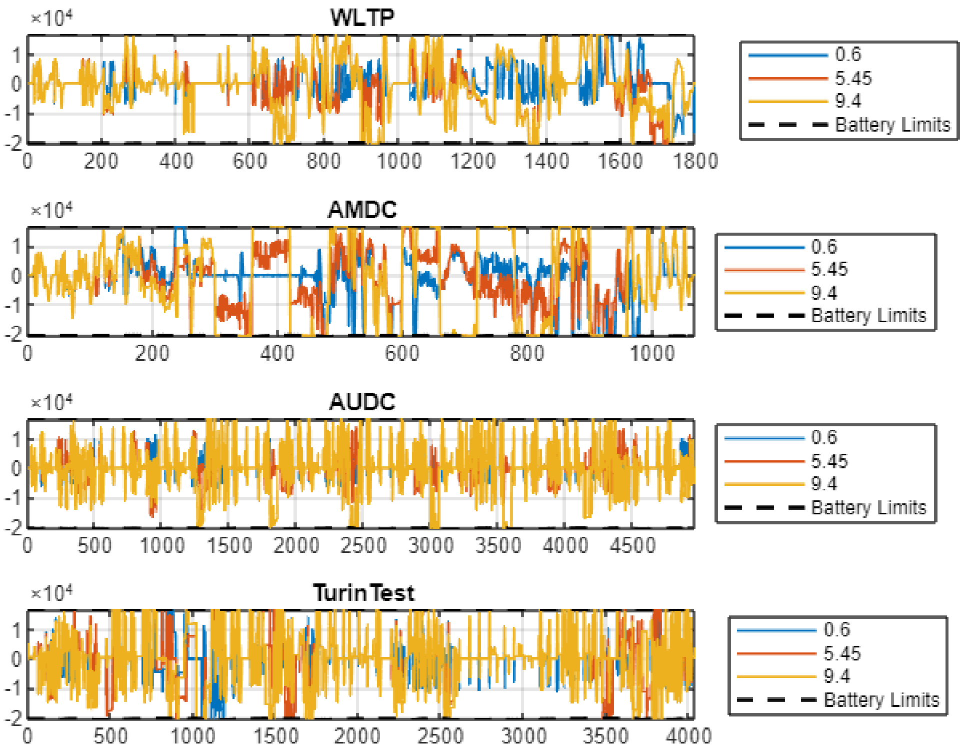
At **low Kp** values, the engine tends to operate predominantly within its **maximum efficiency** zone. In the contrary, as **Kp increases**, the A-ECMS adopts a more **aggressive strategy** for managing the engine. This typically results in **higher power levels**, with operating points shifting toward higher engine speeds and torques, in **less efficient regions**.

This shift is likely due to greater fluctuations in the equivalence factor, which increases the cost of battery energy and makes it more advantageous to operate the engine even at lower efficiency points. This trend is particularly evident in the Kp vs fuel consumption plot.

Notably, certain operating points fall outside the optimal range in both the **AMDC** test with **Kp = 0.6** and the **TurinTest** with **Kp = 0.6** and **5.54**. These points, located at low torque and high speed, clearly lie in a low-efficiency zone. This likely reflects **boundary operating condition**: for example, a **SOC** near **0.8** combined with **power demands** that **exceed** the **battery**'s **capacity**. Under such constraints, the engine is required to provide a specific amount of power to prevent overcharging the battery, even if that means operating at a sub-optimal efficiency.

Battery Power vs Time

```
battPwrPlot(results_struct,KpSelected,vehData);
```



From the previous plot, a common trend can be observed across the different driving cycles: a **higher `Kp`** leads to **greater battery usage**, resulting in **increased battery power** both during charging and discharging phases.

Focusing on the case with **Kp=9.4**, in all driving cycles the **battery power frequently reaches** its **operational limits** in both directions. This behavior can contribute to accelerated **battery aging** and may **compromise** its **long-term reliability**. This trend is particularly evident in the **AMDC cycle**, where, as seen in the SOC analysis, from around 300 seconds onward, the battery alternates between charging and discharging near its maximum limits.

Conversely, the lowest tested value, **Kp=0.6**, results in a **more conservative** use of the battery, with **power levels** remaining **low** and **rarely reaching** the **limits**. From a battery health perspective, this represents the most favorable condition, maximizing reliability over time.

Mid-range values of **`Kp`** represent a **compromise** between the two extremes: **battery power levels** are **elevated** but generally remain within **safe** operating limits, offering a **balance** between **performance** and **longevity**.

Kp variation result analysis

The sensitivity analysis confirmed that the choice of `Kp` significantly affects the A-ECMS controller's behavior. **Higher values** result in **aggressive responses**, with **large oscillations** in the equivalence factor, **unstable** SOC **behavior**, and **heavy battery usage**, often compromising energy efficiency and component durability. In contrast, **lower values** like `Kp=0.6` ensure **smoother control**, better **SOC stability**, and more **conservative battery usage**, making them preferable for robustness and long-term reliability. **Intermediate values** such as `Kp=5.45` offer a **compromise**, **but** can still show **instability** in demanding cycles.

Overall, the **optimal range** for `Kp` lies between **0** and **5**, with values **around 1** providing the **most consistent** performance across unknown driving conditions. So it can be stated that it is **not possible** to **identify** a **single** optimal **`Kp`** value, capable of ensuring charge-sustaining behavior across all driving cycles.

T update influence analysis

The **A-ECMS** controller operates by adjusting the **equivalence factor `s`**, which is **periodically updated** based on the deviation of the SOC from a predefined reference value (feedback control). These updates occur at **fixed intervals**, defined by the parameter **`T_update`**.

Selecting an appropriate update interval is therefore crucial for achieving the control objectives. If the **updates** are **too frequent** or **too infrequent**, the system may either **overreact** or **respond too slowly** to changes, affecting energy management performance.

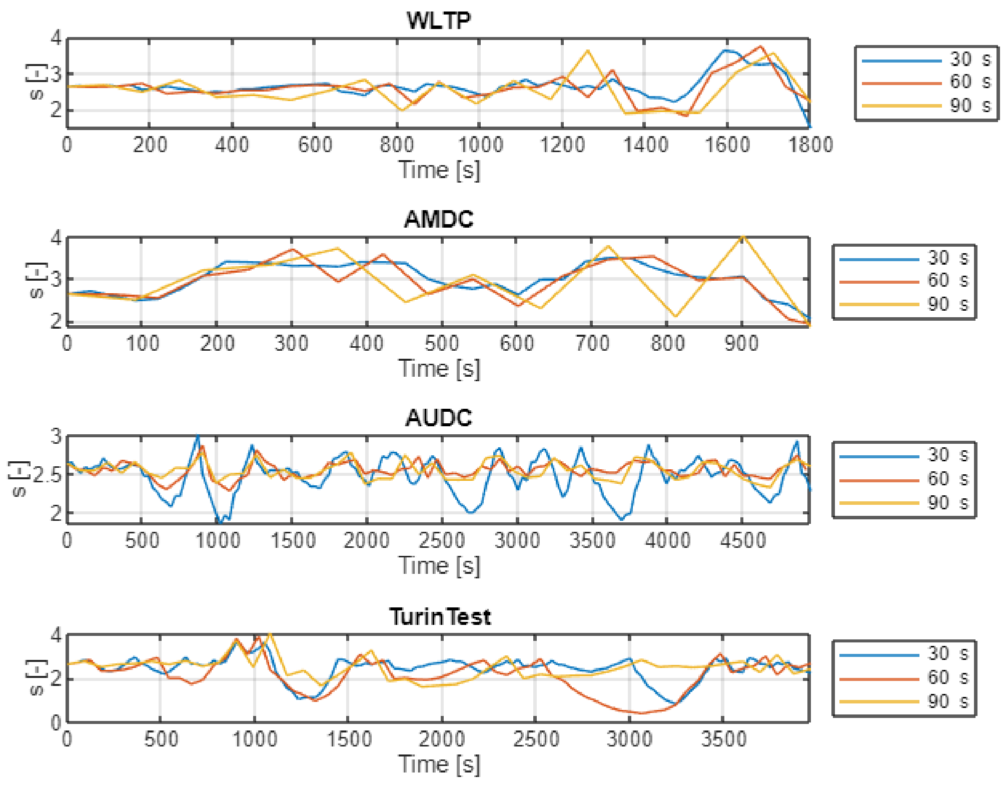
To **assess** the **impact** of this parameter, after initially developing the controller with `T_update=60s`, we performed **additional analyses** by varying this value. Specifically, we considered both **increased** and **decreased update intervals** to evaluate how the controller's behavior changes and extract eventual conclusions.

```
Tupdate_vect = [30 60 90];
Kp_sim = 5.45;

parfor h = 1:length(Tupdate_vect)

    Tupdate = Tupdate_vect(h)
    results_TupKp0_6(h) = simulationLoop(s_0,missions,Tupdate, KpSelected(1),
vehData);
    results_TupKp5_45(h) = simulationLoop(s_0,missions,Tupdate, KpSelected(2),
vehData);
    results_TupKp9_4(h) = simulationLoop(s_0,missions,Tupdate, KpSelected(3),
vehData);

end
```

Equivalence factor vs Time

These plots show how the equivalence factor `s` evolves over time for different values of the update interval `T_update`, while keeping the sensitivity coefficient fixed at the intermediate value **`Kp=5.45`**.

```
eqFactorPlot_Tup(results_TupKp5_45,Tupdate_vect);
```



The influence of `T_update` has been analyzed across all four driving cycles, following the same methodology adopted in the paragraph dedicated into the `Kp` sensitivity analysis.

Short intervals (like `T_update=30s`) lead to **more frequent updates** of the equivalence factor `s`, resulting in a **more responsive** and **adaptive** controller. However, in highly dynamic conditions (such as those found in the AUDC) this can introduces greater variability in `s`, as the controller reacts promptly to every SOC deviation, **potentially** leading to a **oscillatory** and **unstable behavior**.

Conversely, **longer update intervals** (like `T_update=90s`) **reduce** the controller's **reactivity**. This results in a **smoother evolution** of `s`, with **fewer fluctuations**, but also in **slower correction** of deviations. In segments with rapidly changing loads, this delayed response can cause temporary mismatches between actual and reference SOC trajectories.

When **comparing** these values to the baseline `T_update=60s`, it appears that for **highly dynamic driving cycles**, a **longer update time** can be beneficial. It prevents the controller from reacting too quickly to short-term SOC fluctuations, thus promoting a smoother and more stable evolution of the equivalence factor. This helps avoid over-corrections and unnecessary oscillations.

On the other hand, for **more stable cycles** with gradually evolving driving phases ,such as the WLTP which includes urban, suburban, and motorway segments with relatively smooth transitions, a **shorter update interval** may be advantageous. In these cases, more frequent updates allow the controller to better track the optimal energy management strategy without causing instability.

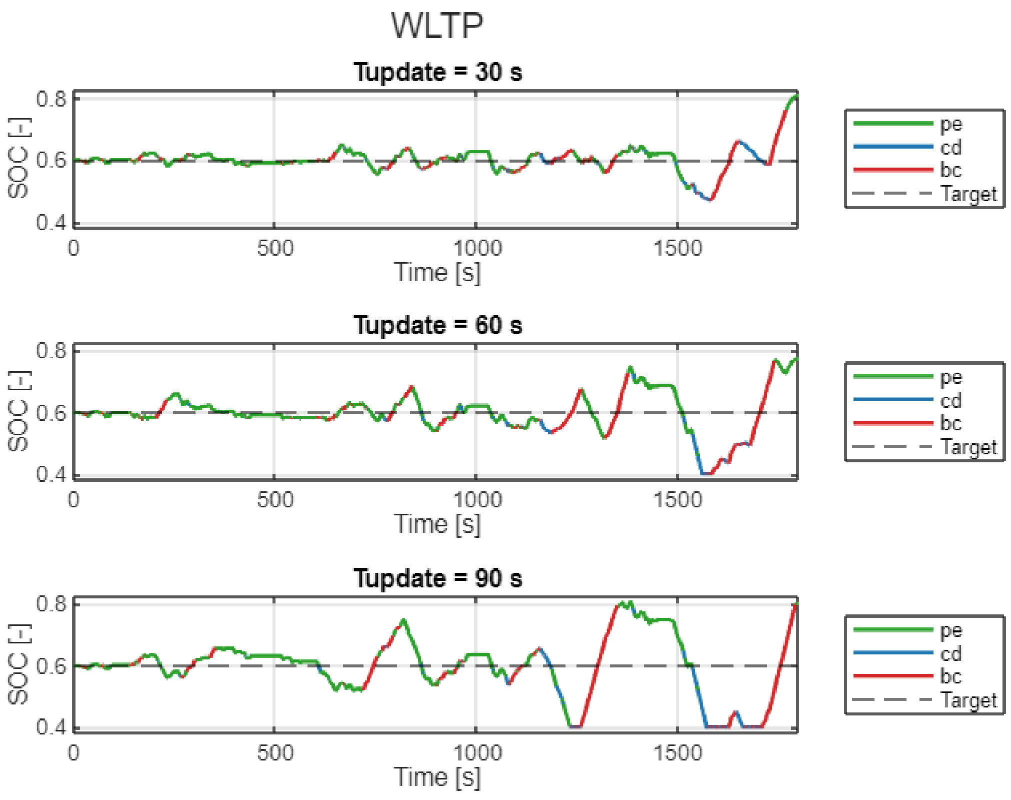
This behavior is particularly evident in the **AUDC cycle**, which, due to its highly dynamic nature, clearly benefits from longer `T_update` values: as the update interval increases, the equivalence factor becomes significantly less oscillatory and more stable.

SOC vs Time

These plots show the variation of the SOC for every cycle for the selected `Kp`, in order to evaluate the difference in the controller behavior.

- **WLTP**

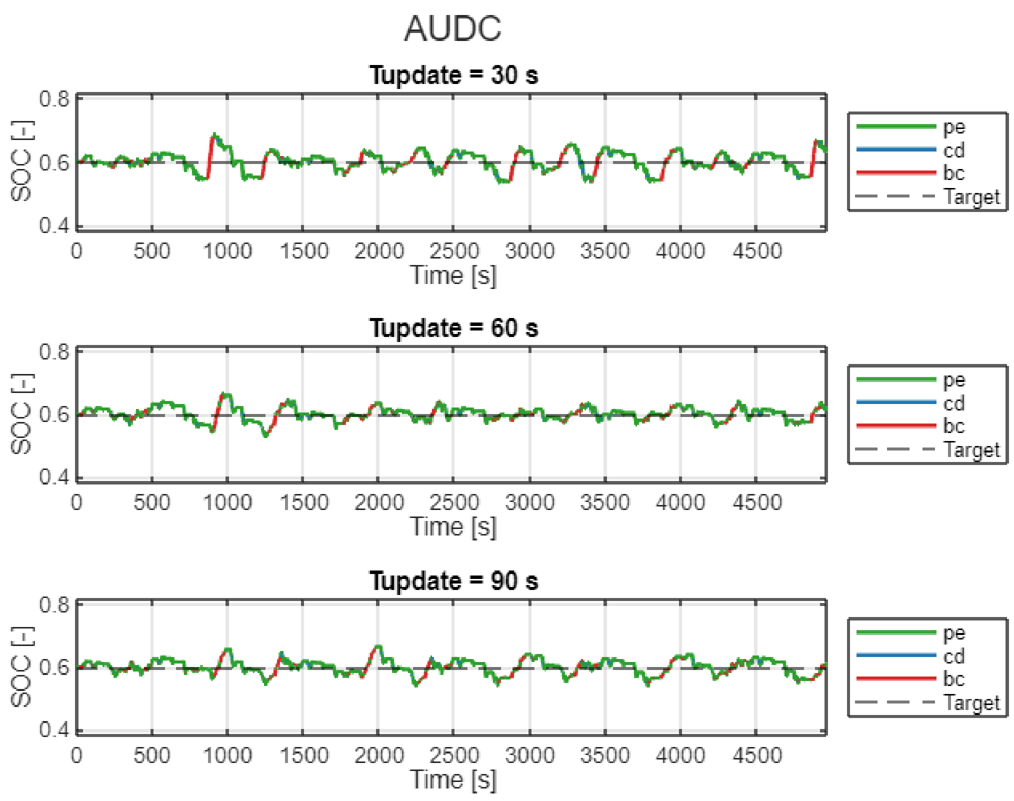
```
SOCwithPF(results_TupKp5_45,"WLTP",Tupdate_vect,"Tup");
```



For the **WLTP** cycle, the best performance is clearly achieved with **`Tupdate=30`**, pointing out that our **previous results** can be **significantly optimized**, even if the SOC deviation increases slightly, reaching 0,2.

- **AUDC**

```
SOCwithPF(results_TupKp5_45,"AUDC",Tupdate_vect,"Tup");
```

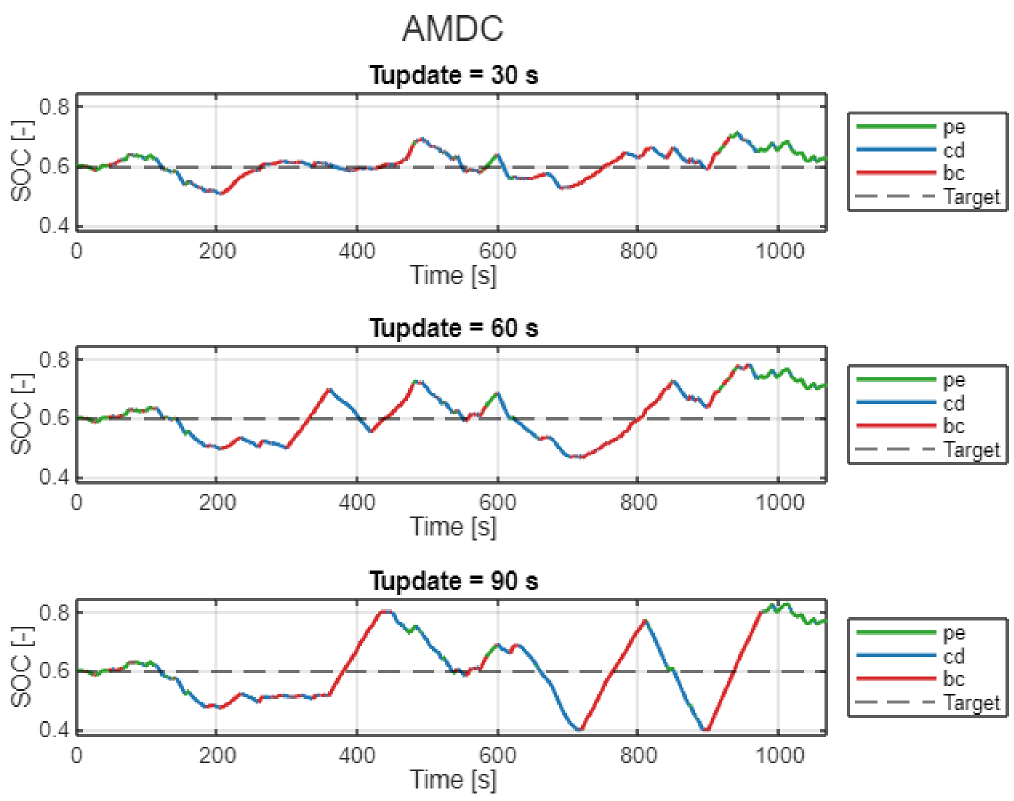


In the **AUDC**, which consists solely of the urban segment, the best behavior could belong to both the 60s and 90s update time. The second one exibits more oscillations but also lower peaks in the SOC.

This suggest we understand that also increasing `Tupdate` can benefit our simulations.

- **AMDC**

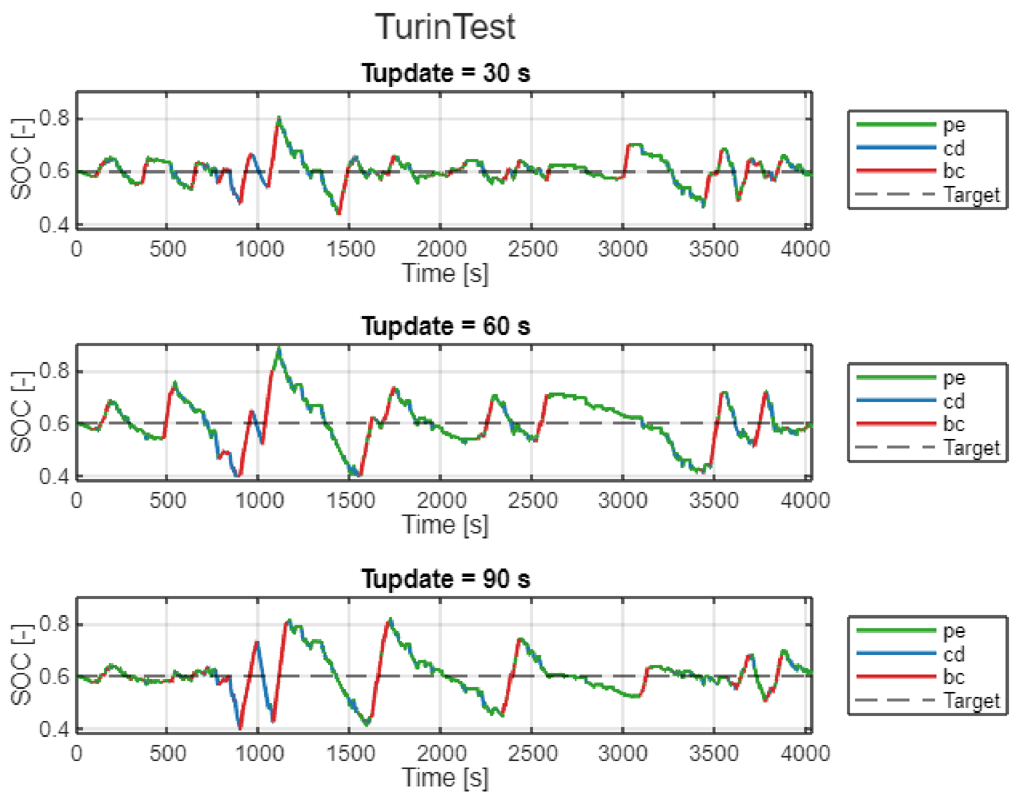
```
SOCwithPF(results_TupKp5_45,"AMDC",Tupdate_vect,"Tup");
```



In the **AMDC** cycle, which contains mainly the motorway segment, the best behavior clearly belongs to `Tupdate=30`, similarly to the WLTP with which share many aspects.

- **Turin Test**

```
SOCwithPF(results_TupKp5_45,"TurinTest",Tupdate_vect,"Tup");
```



Finally the **Turin Test** shows that none of the simulations can be assumed as better, because at different portions of the cycle correspond different optimal update times. For example in the first part, until the 800s, `Tupdate=90` appears to be the best. In the contrary, from 800 to 3000s `Tupdate=90` outmatch the others. Finally from 3000 to 4000s there is not an evident different between them.

Considerations on T_update influence

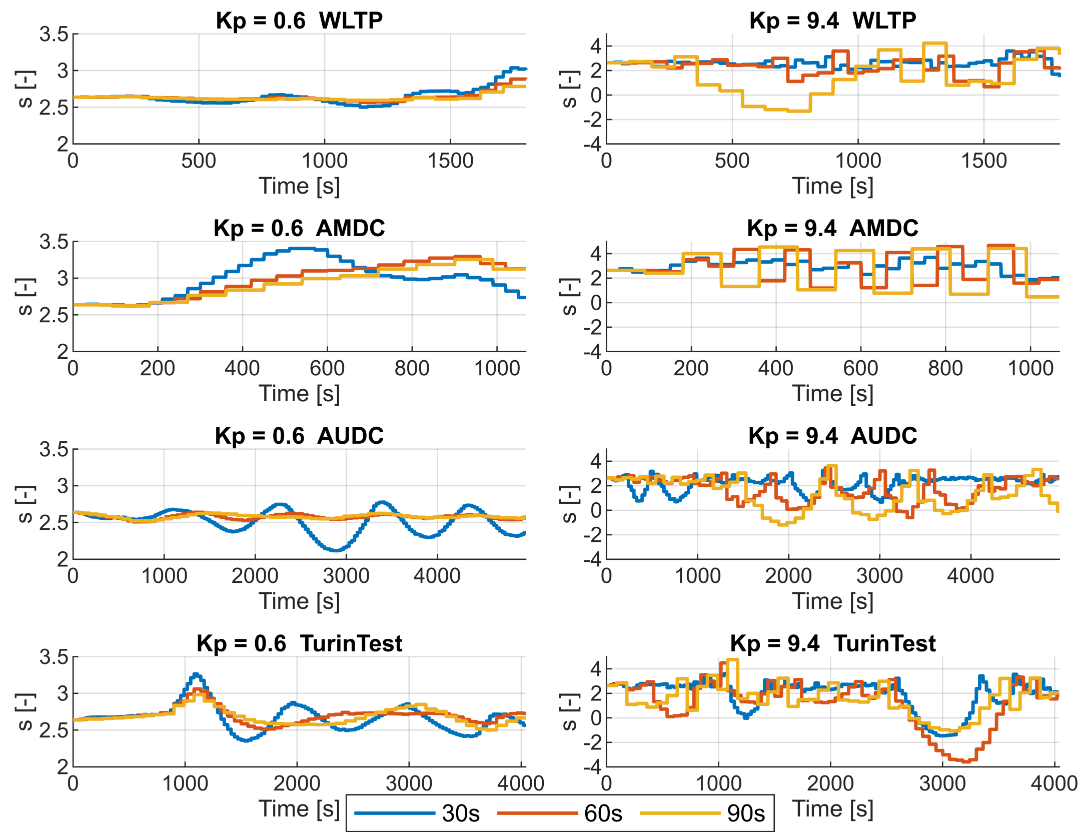
The results appear to align with those presented in the previous paragraph, indicating that segments with high dynamic cycles benefit from a larger observation window, while those with low dynamics perform better with a smaller one. However, the findings of this study do not fully confirm this assertion, suggesting that further investigation is needed to gain a deeper understanding of the controller's behavior.

Additional analysis on Kp - Tupdate correlation

In this section, the previous analysis is extended to verify the correctness of the results and to assess the correlation between `Kp` and `Tupdate`, ultimately leading to the final outcome of the analysis. In order to do this, the same plots of the previous paragraph are reproduced with the other two selected `Kp` values, already used in chapter `Kp` analysis.

```
% Extract the figure from the function
f1 = eqFactorPlot_Tup(results_TupKp0_6,Tupdate_vect);
f2 = eqFactorPlot_Tup(results_TupKp9_4,Tupdate_vect);

% Find all the graphics objects and flip them
axes1 = flipud(findall(f1, 'Type', 'axes', 'Visible', 'on'));
axes2 = flipud(findall(f2, 'Type', 'axes', 'Visible', 'on'));

% Initialize the figure and the layout
fFinal = figure;
```

```
t = tiledlayout(4,2, 'Padding','compact', 'TileSpacing','compact');

% Plot the fist column
for i = 1:4

    % Create a new box plot
    ax = nexttile(t,(2*i+1)-2);

    % Copy the needed plot object from the function figure
    p1(i,:) = copyobj(allchild(axes1(i)), ax);

    % Finalize the plot
    grid on
    xlabel("Time [s]")
    ylabel("s [-]")
    xlim([0 missions.(missions_name(i)).time_s(end)])
    ylim([2 3.5])
    if isgraphics(axes1(i).Title)
        title(ax, join([  "Kp = 0.6 ",axes1(i).Title.String]));
    end
end

% Plot the second column
for i = 1:4

    % Create a new box plot
    ax = nexttile(t,2*i);

    % Copy the needed plot object from the function figure
    p(i,:) = copyobj(allchild(axes2(i)), ax);

    % Finalize the plot
    grid on
    xlabel("Time [s]")
    ylabel("s [-]")
    xlim([0 missions.(missions_name(i)).time_s(end)])
    ylim([-4 5])
    if isgraphics(axes2(i).Title)
        title(ax,join([ "Kp = 9.4 ",axes2(i).Title.String]));
    end
end

% Close the figures
close(f1)
close(f2)

% Plot the legend
ax1 = axes('Position', [0 0 1 1], 'Visible', 'off');
lg = legend(ax1, flip(p(1,:)),["30s","60s","90s"], ...,
            'Location', 'southoutside', 'Orientation', 'horizontal');
```



As can be observed, the **most significant result** is that the **optimal combinations** of `Kp` and `T_update` occur when **small `Kp`** is paired with **bigger `T_update`** and vice-versa (**large `Kp`** with **smaller `T_update`**).

This behavior can be explained considering the A-ECMS formulation, which uses a proportional gain to compute the equivalence factor s: when `Kp` is increased (in our case to 9.4, which is close to the upper limit) the sensibility of the controller also increases, requiring a higher update frequency of the SOC variation. If the observation window is too big and the SOC undergoes significant changes, the equivalence factor in the following interval could become too aggressive, leading to noticeable oscillations.

A similar (but opposite) thing happens when the proportional gain is decreased (in our case 0.6), where a smaller sensitivity with a larger observation window allow a smooth behavior without many oscillations. This is likely due to a higher difficulty of the `Kp` part of the equation to follow the SOC variation, leaving more importance to the averaging factor.

Considerations on Kp - T_update correlation and influences

The analysis confirms that the optimal combination of `Kp` and `T_update` depends strongly on the characteristics of the driving cycle, and that these two parameters must be tuned in coordination to ensure good performance.

In **highly dynamic cycles**, such as the AUDC, **longer update intervals** (like `T_update = 90 s`) combined with **lower `Kp`** values tend to **improve stability**. A wider observation window helps the controller to interpret SOC variations over a longer horizon, reducing the impact of rapid but temporary deviations caused by individual maneuvers. In these conditions, a less sensitive controller (low `Kp`) updated less frequently avoids aggressive corrections and excessive oscillations of the equivalence factor *s*, resulting in smoother behavior.

In the contrary, with **less dynamic cycles**, such as WLTP and AMDC, the controller **benefits** from **shorter update intervals** (like `T_update = 30 s`) and **higher `Kp`** values. In these scenarios, where speed and load evolve more gradually, more frequent updates and a more responsive gain enable the controller to follow the SOC trajectory more closely and maintain better alignment with the reference. The increased sensitivity does not introduce instability due to the smoother nature of the cycle, and allows better exploitation of the energy management potential.

A special case is represented by **mixed or irregular profiles**, such as the Turin Test, where the optimal `T_update` may vary significantly across different sections of the same cycle. In these situations, no single `T_update` value clearly outperforms the others across the whole duration, suggesting that an adaptive tuning strategy could further improve performance.

This confirms that **`T_update`** should **not** be treated as an **independent tuning** parameter **but** rather **co-optimized** with **`Kp`**, depending on the expected characteristics of the driving cycle.

Conclusion

The analysis confirms that the behavior of the A-ECMS controller is highly dependent on the value of the proportional gain Kp, which governs how the equivalence factor `s` is adjusted in response to deviations in the state of charge (SOC). Across the four evaluated driving cycles (WLTP, AMDC, AUDC, Turin Test), it is evident that high `Kp` values result in an aggressive controller response, characterized by frequent oscillations in sss, unstable SOC trends, and increased battery usage. This often leads the system to operate outside the desired SOC range [0.4, 0.8] and pushes the engine toward low-efficiency regions.

Conversely, low Kp values, such as 0.6, promote a much smoother controller behavior. The equivalence factor evolves gradually, the SOC stays more stable, and the battery is used more conservatively, reducing stress and improving long-term reliability. However, this comes at the cost of reduced responsiveness, especially in highly dynamic cycles like AMDC, where the system struggles to maintain charge-sustaining operation.

There is therefore a clear trade-off between responsiveness and stability. A single `Kp` value that performs well across all cycles does not exist. While intermediate values (around `Kp=5.45`) represent a reasonable compromise, slight performance degradation still occurs under certain operating conditions.

Furthermore, the analysis shows that relying solely on SOC feedback is effective but not always sufficient —particularly in mixed or high-load cycles. Improvements could be achieved by combining `Kp` tuning with additional strategies, such as predictive control or adaptive gain scheduling, although these would increase system complexity.

In conclusion, the A-ECMS controller demonstrates good potential for real-world applications, but careful tuning of `Kp` (and `Tupdate`) is essential. While a low `Kp` ensures stability and battery health, achieving optimal fuel efficiency and charge-sustaining behavior across all drive cycles may require more advanced adaptive control strategies.

Simulation Loop function

The simulation loop function from the previous assignment has been adapted to handle multiple driving cycles and implement an Adaptive ECMS strategy. To achieve this, both the input and output structures have been modified as follows:

Inputs:

- `s_0`: the initial value of the equivalence factor.

- `missions`: a structure containing all mission data, where each field corresponds to a different driving cycle (a 1×1 structure with a number of fields equal to the number of missions).
- `Tupdate`: the time interval at which the equivalence factor is updated.
- `Kp`: the proportional gain used by the controller to adjust the equivalence factor based on SOC deviation.

Output:

- `results`: a structure that stores all simulation outputs, organized by mission (also a 1×1 structure with a number of fields equal to the number of missions).

At the end of each driving cycle simulation, all output variables are cleared using the `clear` function to prevent conflicts with the subsequent simulation.

```
function [results] = simulationLoop(s_0,missions,Tupdate, Kp,vehData)


for i = 1:length(fields(missions))

    % Mission name is extracted from the structure
    mission_name = fields(missions);
    mission_name = mission_name{i};

    %time vector
    time_vector = missions.(mission_name).time_s;

    % Time discretization
    dt = vehData.dt;

    %Velocity and acceleration
    vehAcc = missions.(mission_name).acceleration_m_s2;       %[m/s^2]
    vehSpd = missions.(mission_name).speed_km_h/3.6;        %[m/s]

    %Initializing SOC, engine state and time value
    time = 0;
    SOC(1) = 0.6;
    engState(1) = 0; % engine is off

    %Initialize equivalence factor of the controller
    s_vect = [s_0 s_0];

    % Simulation Loop
    for n = 1:length(time_vector)
        %Time
        time = n*dt;

        % Control variables computed using ECMS control
        [engSpd(n), engTrq(n),s_vect] =
adaptiveEcmsControl(SOC(n),vehSpd(n),vehAcc(n),vehData,s_vect,Kp,Tupdate,time,SO
C(1));

        % Using the vehicle model the states variables are evaluated
        [SOC(n+1), fuelflwRate(n), unfeas(n), prof(n)] = hev_model(SOC(n),
[engSpd(n), engTrq(n)], [vehSpd(n), vehAcc(n)], vehData);

        % engine state for each instant is saved
        engState(n+1) = engSpd(n) >= vehData.eng.idleSpd & engTrq(n) > 0;

        if unfeas(n) == 1
            fprintf("unfeas at %d s mission %s \n",[n,mission_name])
        end

    end

    % Save results of simulation in a structure
    results.(mission_name).SOC = SOC;
    results.(mission_name).fuelflwRate = fuelflwRate;
    results.(mission_name).unfeas = unfeas;
    results.(mission_name).prof = structArray2struct(prof);
    results.(mission_name).engState = engState;
    results.(mission_name).s_vect = s_vect;

    % Used variables are cleared to avoid intersection problems with
    % previous cycle
    clear SOC; clear engState; clear unfeas; clear engSpd; clear engTrq;
    clear prof; clear fuelflwRate;


end
end
```

The A-ECMS controller

The ECMS controller developed in Lab 2 used a fixed equivalence factor *s*, optimized for a specific mission to ensure an SOC deviation of only 0.01. However, this approach assumed the mission was known in advance and did not incorporate any form of present a look-ahead capability.

The objective of the current assignment is to develop an **Adaptive ECMS** (A-ECMS) that maintains **charge-sustaining operation without prior knowledge of the mission**, while more importantly minimizing fuel consumption.

The new A-ECMS controller retains the same inputs and outputs as the ECMS controller from the previous lab, with some additions. On the input side:

- **`s_vect:`** stores all equivalence factors computed from the start of the simulation up to the current time.
- **`Kp:`** the proportional gain applied to the SOC deviation to calculate the new equivalence factor when it's time to update it.
- **`Tup:`** defines the observation window and, accordingly, the update interval for the equivalence factor.
- **`time:`** the current simulation time.
- **`SOC_0:`** the initial state of charge, used to compute the SOC deviation during updates.

On the output side:

- **s_vect:** remains unchanged and is passed along to maintain continuity across updates.

Controller logic

The core logic of the controller remains the same as in Lab 2, with the addition of a new section dedicated to updating the equivalence factor *s* during the simulation.

Since the driving conditions are not known in advance, the equivalence factor must be adapted dynamically to ensure charge-sustaining behavior. To achieve this, a SOC feedback strategy is implemented. The equivalence factor is updated periodically at fixed intervals, defined by the update time `Tup`, which establishes the observation window. During this window, the controller monitors the system's evolution—particularly the state of charge (SOC).

The next equivalence factor is computed using the following formula

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

In this equation:

- The second term can be compared to a proportional controller, where the SOC error $(\sigma_0 - \sigma(t))$ (i.e: the deviation from the target SOC, set to 0.6) drives the adjustment. The proportional gain K_p determines the aggressiveness of the controller (i.e: how quickly the equivalence factor responds to changes in SOC).
- The first term introduces a stabilizing effect by averaging the last two values of the equivalence factor, helping to smooth out rapid fluctuations and ensure more gradual transitions.

Controller implementation

At the beginning of the function, the equivalence factor for the current iteration is determined within an `if` condition. Specifically, the update time (`Tup`) and the current mission time (`time`) are compared using the `mod` function, which returns the remainder of the division. If the remainder is zero (i.e., the current time is a multiple of the update interval), then the equivalence factor is updated.

In this case, the new value of sss is computed using the formula described earlier. This value is then appended to the `s_vect` vector and set as the current equivalence factor for the simulation step.

If the remainder is not zero, meaning the current time is not aligned with the update interval, the equivalence factor remains unchanged and is set equal to the last value in `s_vect`.

```
function [engSpeed, engTorque, s_vect] =
adaptiveEcmsControl(SOC,vehSpd,vehAcc,vehData,s_vect,Kp,Tup,time,SOC_0)


% Equivalence factor computation

% if time is a multiple of Tupdate the equivalence factor updated
if mod(time, Tup) == 0  % mod returns the rest of the division
    s_vect(end+1) = (s_vect(end)+s_vect(end-1))/2+Kp*(SOC_0-SOC);
    s = s_vect(end);
else
% else the last equivalence factor is used
    s = s_vect(end);
end
```

```
% Define the higher threshold of the SOC
SOChi = 0.8;

% Define the lower threshold of the SOC
SOClo = 0.4;

% Time discretization of the simulation
dt = vehData.dt;  %[s]

% Define energy capacity of the battery
batteryEnergy = vehData.batt.nomEnergy*3600; %[W*s]

% Define lower heating value of the fuel
Q_LHV = vehData.eng.fuelLHV/1000;  %[J/g]

% Define the resolution of the grid (discretization of the maps)
res = 250;

% Create the speed vector that swipe from idle to max speed
engSpd_vect = linspace(vehData.eng.idleSpd,vehData.eng.maxSpd, res); %[rad/s]
% Zero speed is included in the vector to consider engine off case
engSpd_vect = [0,engSpd_vect];

% Define the maximum torque that can be exploited by the engine
engMaxTrq = max(vehData.eng.maxTrq(engSpd_vect));  %[Nm]

% Create the torque vector that swipe from 0 to maximum torque
engTrq_vect = linspace(0,engMaxTrq,res);  %[Nm]

% Zero is added to the torque vector in correspondence to the zero speed
% point
engTrq_vect = [0,engTrq_vect];

% Grid of engine speed and torque points is created
[engSpd_grid, engTrq_grid] = ndgrid(engSpd_vect,engTrq_vect);

% The vehicle model is used to define SOC, fuel consumption and eventual
% unfeas after each working point of the grid
[SOC_next,mf_eng,unfeas] = hev_cell_model({SOC},{engSpd_grid,engTrq_grid},
{vehSpd,vehAcc},vehData);

% Conversion cell to array
SOC_next = SOC_next{1};

% The derivative of the SOC is computed for each point of the matrix
SOC_dot = (SOC_next-SOC)./dt;

% Equivalent consumption formulation is implemented
mf_eq = mf_eng -s.*(batteryEnergy/Q_LHV).*SOC_dot; %[g/s]

% mf_eq -> matrix dimension (res+1 x res+1)
% SOC_next -> matrix dimension (res+1 x res+1)
```

```matlab
% unfeas -> matrix dimension (res+1 x res+1)

% For a given point penalty is given in case of SOC
% overcame the higher SOC threshold
mf_eq(SOC_next > SOChi) = 2000;

% For a given point penalty is given in case of SOC
% go below the lower SOC threshold
mf_eq(SOC_next < SOClo) = 2000;

% For a given point penalty is given in case of an unfeas is present
mf_eq(unfeas == 1) = inf;

% Extract the minimum equivalent fuel consumption
minEqMf = min(mf_eq(:));

if min(unfeas) == 1
    fprintf("All unfeas")
end

if minEqMf == inf
    fprintf("Warning! Non feasable condition \n")
end

prova = unfeas(mf_eq == minEqMf);
if prova(1) == 1
    fprintf("Controller unfeas \n")
end

% Engine speed corresponding to the minimum equivalent fuel consumption is
% extracted
engSpeed = engSpd_grid(mf_eq == minEqMf);

% Engine torque corresponding to the minimum equivalent fuel consumption is
% extracted
engTorque = engTrq_grid(mf_eq == minEqMf);

% In case of multiple minimum points only the first, at lower speed, is
% considered
engSpeed = engSpeed(1);
engTorque = engTorque(1);


end
```