FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EE 402 – Senior Project II

2019 / 2020 SPRING

FINAL REPORT

Design and Real-Time Simulation of an Energy-Efficient Control Algorithm on OpenECU with HIL Simulators for Plug-in Electric Vehicles

MEHMET BATU ÖZMETELER
S009924

SUPERVISOR:

PROF. H. FATİH UĞURDAĞ

Table of Contents

Table of Figures

Design and Real-Time Simulation of an Energy-Efficient Control Algorithm on OpenECU with HIL Simulators for Plug-in Electric Vehicles

1. Introduction

1.1. Problem Statement

Global warming emissions and air pollution have become major threats for our planet. According to NASA: Climate Change and Global Warming website, the current warming trend considered to be crucial due to the majority of it is quite likely (greater than 95 percent probability) to be the consequence of human activity since the mid-20th century and advancing at a rate that is remarkable over decades to millennia [1]. Since carbon dioxide emissions from the combustion of fossil fuels are one of the primary causes for both issues [2], reducing oil consumption can help avoid such hazards.

Reducing dependence on petroleum which has turned into a scarce resource is a difficult task. However, thanks to progresses in electrically powered transportation technology, electric vehicles (EVs) are conceived to be a relevant solution. As reported by Bloomberg New Energy Finance, electric vehicles are expected to discard two million barrels of oil a day by 2028 [3].

The success of electric vehicles can be connected to them having fewer moving parts meaning lower operating and maintenance costs. The weight is ideally distributed front-to-rear below the car's center of gravity making it more stable and safer to drive. Reduced noise pollution is another benefit. Additionally, the instantaneous torque response and full torque from standstill is practical. However, the long recharge time, limited speed and cruising range are probably the most critical drawbacks of EVs.

1.2. Project Description

For the purpose of increasing the popularity of EVs to rapidly shift into a greener era, this project aims to help enhance the driving range of PHEVs. Improving this competence can be achieved by only two possibilities: develop a battery technology which allows higher capacity and lower weight or come up with an energy management strategy that influences the driving style in order to optimize the energy consumption. Since soon improvements on batteries is uncertain [4], second option seems to be more suitable to address this challenge. But what is the ideal approach to develop such a strategy?

2. Literature Review

2.1. Energy Management and Driving Strategy for In-Wheel Motor Electric Ground Vehicles with Terrain Profile View

In 2014, Y. Chen, X. Li, C. Wiet and J. Wang [5] has presented a terrain-information- and actuator-efficiency-incorporated energy management driving strategy (EMDS) for maximizing the travel distance of in-wheel motor, pure electric ground vehicles (EGVs). In this paper, it is stated that such an approach is preferred because minimization of energy consumption for a certain trip with terrain preview based on the operating efficiencies of in-wheel motors and a traffic model is essential to maximize the total travel distances of an EGV.

In this paper, unlike conducting energy optimization under given vehicle speed profiles that are specified a priori in most literature, the optimally-varied vehicle velocity and globally optimal in-wheel motor actuation torque distributions are simultaneously obtained to minimize the EGV energy consumption by employing the dynamic programming method for the first time.

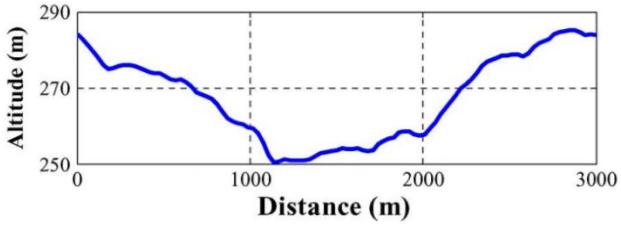


Figure 1: Altitude Profile of the Selected Real Road Terrain

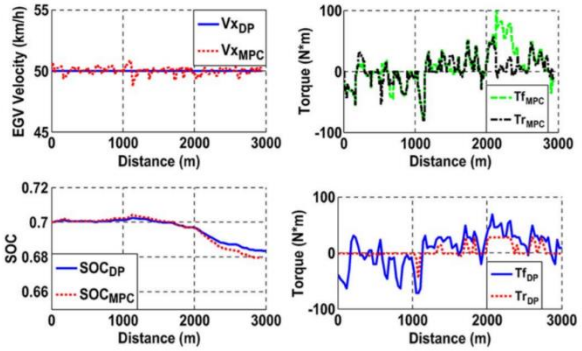


Figure 2: Comparison of Torque Distributions and SOC between the DP method and the MPC Method along the Terrain Profile

Category	Method	SOC end	SOC reduction	Improvement
Constant speed	DP	0.6835	0.0165	18.7%
v_x =50 km/h	MPC	0.6797	0.0203	
Optimal constant	DP	0.6865	0.0135	9.4%
v_x =29 km/h	MPC	0.6851	0.0149	
Optimal varied v_x	DP	0.6896	0.0104	29.3%
(fast preceding car)	MPC	0.6853	0.0147	
Optimal varied v_x	DP	0.6889	0.0111	23.5%
(slow preceding car)	MPC	0.6855	0.0145	

Figure 3: SoC Comparison between the DP and MPC Method

Hereby, although the optimization results may be sensitive to some key factors (e.g., terrain information), some common conclusions from the EMDS design can still be obtained:

1) The DP method shows a greater improvement for the EGV energy saving than the MPC method based on the terrain preview.

2) For the constant velocity cruise, the optimal constant velocity exists based on the given terrain. Higher or lower vehicle velocities will cost more energy for the EGV to finish the trip.

3) Compared with the constant velocity case, the optimally varied vehicle velocity makes a further improvement on energy saving. The EMDS design based on the globally optimal DP method in the paper provides a benchmark to evaluate other real-time implementable algorithms, such as the MPC method.

Complying with their conclusions, in Figure 1, altitude profile of the selected real road terrain, in Figure 2 comparison of torque distributions and SOC between the DP method and the MPC method along the terrain profile, in Figure 3, SoC comparison between the DP and MPC method can be seen.

2.2. Model Predictive Control Approach to Design Practical Adaptive Cruise Control for Traffic Jam

T. Takahama and D. Akasaka [6] has presented a design method of a Model Predictive Control (MPC) with low computational cost for a practical Adaptive Cruise Control (ACC) running on an embedded microprocessor. Generally, a problem with previous ACC is slow following response in traffic jams, in which stop-and-go driving is required. To improve the control performance, it is stated that it is important to design a controller considering vehicle characteristics which significantly changes depending on driving conditions.

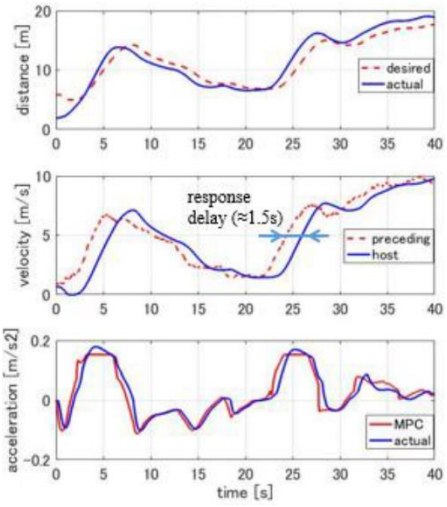


Figure 4: Experimental Results During Traffic Jam

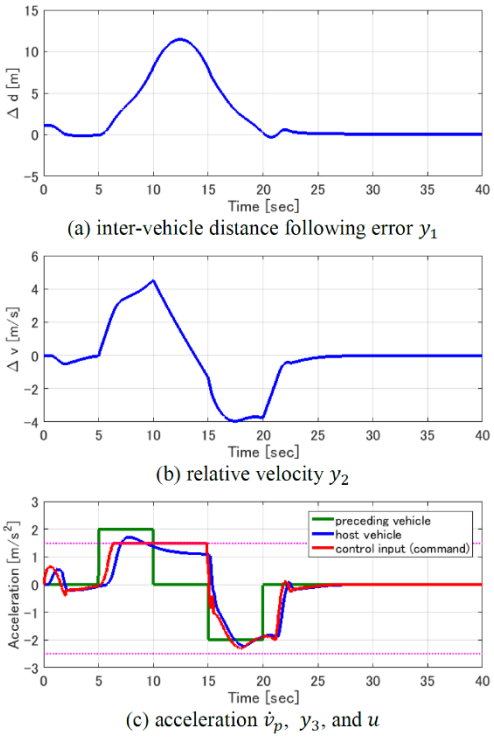


Figure 5: MPC Simulation Results

In this paper, they attempted to solve the problem by using MPC that can explicitly handle constraints imposed on, e.g., actuator or acceleration response. In addition, the computational load for the practical use of MPC is tried to be decreased by using low-order prediction model. In conclusion, computer simulation showed that the MPC is superior to conventional controllers because it treats actuator constraints and has higher response as ACC. Experimental verification results showed that the proposed MPC controller can be implemented in embedded microprocessors and can achieve high responsiveness and less discomfort.

Complying with their results, in Figure 4, experimental results during traffic jam and in Figure 5, MPC simulation results can be seen.

3. Methodology

3.1. Suggested Solution

The answer suggested in this project for this complication lies in having a specific energy-efficient control algorithm for longer driving ranges by manipulating the driving speed according to several factors. The goal of the first term was a reciprocal working demonstration of the interaction between an electronic control unit and a Hardware-in-the-Loop simulator. A modified PID control algorithm was run on an ECU which generated appropriate torque requests to drive the velocity output to the provided constant reference speed. Meanwhile, the process was being simulated by a real-time HIL simulator that mimics the plant behavior.

HIL simulation was used as a technique to show how one can ensure testing feasibility and enhance coverage in cases where testing is costly, unsafe, and not easily feasible. This was a beginning step that formed a basis where the spoken energy-efficient control algorithm can be designed, developed, tested, and implemented.

The main deliverable concerning the second term is numerically designing and implementing a model predictive control algorithm to install into the existing setup. Its performance and accuracy will be maximized by tuning weights, adjusting operating frequency, certain parameters, and constraints in order to get as close as possible to a real-time system. A block diagram of the suggested solution can be found below as Figure 6.

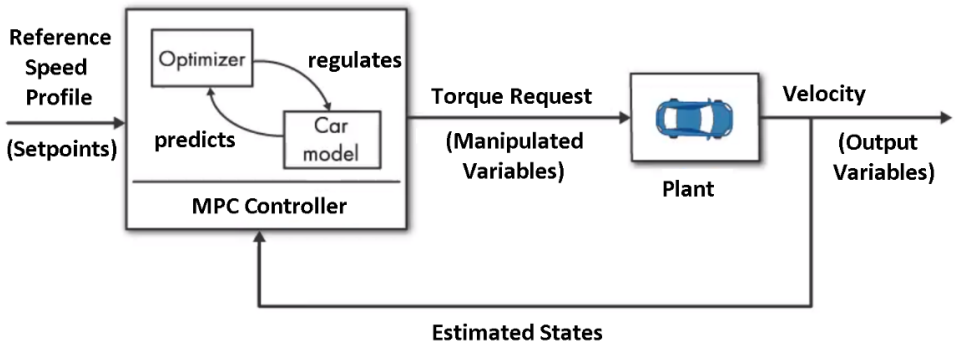


Figure 6: Block Diagram of the Suggested Solution

During this period, essential software tools and hardware technicalities will be grasped profoundly and experimented on. A methodology of trial and error for tuning and adjusting, I/O manipulation via a calibration tool for testing and debugging will be used to achieve the desired outcome from the simulation process. Since the general idea is to increase the cruising range of EVs, this project aspires to increase the acceptance of EVs which consequently provides a quicker shift to a greener era. It is safe to say that this solution is manufacturable and will raise awareness in our society by proving EVs to be secure, sustainable, economic and environment friendly.

3.2. Utilized Hardware

3.2.1 Electronic Control Unit (ECU)

Figure 7: Pi Innovo M250 Module

ECU is a name given to a device which controls one or more electrical systems in a vehicle. It provides instructions for various electrical systems, guiding them on what to do and how to operate. In this case, it will be carrying a distinct control algorithm as its set of rules. For this project, M250 module from Pi Innovo is selected due to its useful features instead of other products from the M-series. An image of the module can be seen on the left as Figure 7.

The M250 module is a compact electronics module that is suited to computationally intensive applications requiring a chassis mounted, sealed metal housing with IP67 environmental protection. It has a NXP MPC5534 as a 32-bit MCU (80Mhz), 2 CAN interfaces along with a 46-pin connector. The code space is 512 KB, RAM space is 64 KB and the calibration space is 256 KB. [7]

3.2.2 Hardware-In-The-Loop Simulator (HIL)

The instructions provided by the ECU will be followed by a HIL (Hardware-in-the-Loop) simulation which mimics the physical part of the system, the plant. It is a technique to test control systems, hereby representing the car that has many actuators and sensors. Hereby, Opal-RT OP4200 is used due to it offering Hardware-in-the-Loop (HIL), Rapid Control Prototyping (RCP), data acquisition and I/O expansion capabilities in a desktop-friendly package. [8] An image of the simulator can be seen on Figure 8.

Figure 8: OPAL RT OP-4200

Figure 9: AATech APS-3303DD

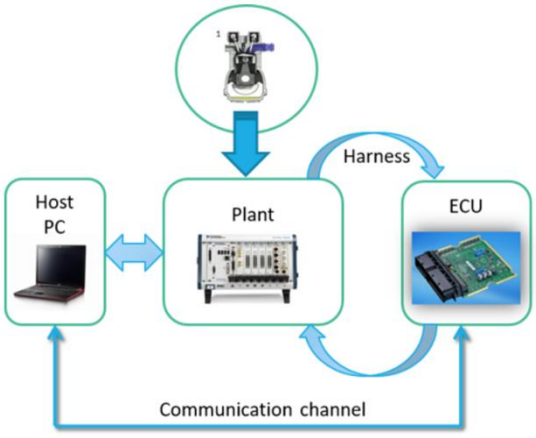
3.2.3Power Source

A power source of AATech APS-3303DD is used to power the whole system with the required energy. Figure 9 shows the utilized hardware.

3.2.4 USB-CAN Interface

In order to connect host PC(s) to the CAN interface, a KVASER Leaflight cable is used as an USB-CAN interface. The alternatives are CANable, PEAK-System etc. This product is preferred since its installation is quick and easy (plug-and-play). It supports both 11-bit (CAN 2.0A) and 29-bit (CAN 2.0B active) identifiers while being 100% compatible with applications written for other Kvaser CAN hardware with Kvaser CANlib. A high-speed CAN connection (compliant with ISO 11898-2), up to 1 Mbit/s is obtainable. Furthermore, it is fully compatible with J1939, CANopen, NMEA 2000R and DeviceNet. [9] Figure 10 depicts the used cable.

Figure 10: Kvaser Leaflight HS-V2



3.2.5 Host PC(s)

Host PCs have the required software installed on them. They run Win 7 SP1 64-bit operating systems for third-party tool compatibility. Figure 11 shows how ECU software development cycle with HIL testing works.

Figure 11: ECU Software Development Cycle with HIL Testing

3.3. Utilized Software

3.3.1 MATLAB, Simulink, and Other Related Toolboxes

MATLAB is a programming platform which is based on a matrix-based language that allows the most natural expression of computational mathematics. Implementing, testing and debugging algorithms is easy with the help of the large database of built-in functions. External libraries can be called, integrated to perform extensive data analysis and visualization. Working with OpenECU library which is the compatible software to our hardware is much simpler due to the effortless integration. Also, using Simulink for model-based design is advantageous. For these reasons, MATLAB is preferred instead of other high-level languages.

3.3.2 Calibration Tool and its Related Libraries

One of the essential third-party tool requirements for OpenECU is a calibration tool. This is used to interact with software while it runs in real-time on an embedded system or an ECU. It makes use of a communication link to access the memory for read/write operations, loads information (memory layout, variables, functions) about the application. Among other possible options like ETAS INCA, Vector CANape, ATI Vision; Pi Snoop Professional is chosen. Having a tool which helps for software debugging, testing, flash reprogramming, calibration editing, CAN bus work and diagnostics is more than enough. [10]

3.3.3 Real-Time Simulation Software

RT-LAB is OPAL-RT's real-time simulation software which is fully integrated with MATLAB/Simulink. It offers the most complex model-based design for interaction with real-world environments. In order to turn models into interactive real-time simulation applications for automotive industry, RT-LAB has the necessary flexibility and scalability. It handles everything, including code generation, with an easy-to-use interface. [11] Figure 12 visualizes the 4 steps to real-time simulation using RT-LAB.

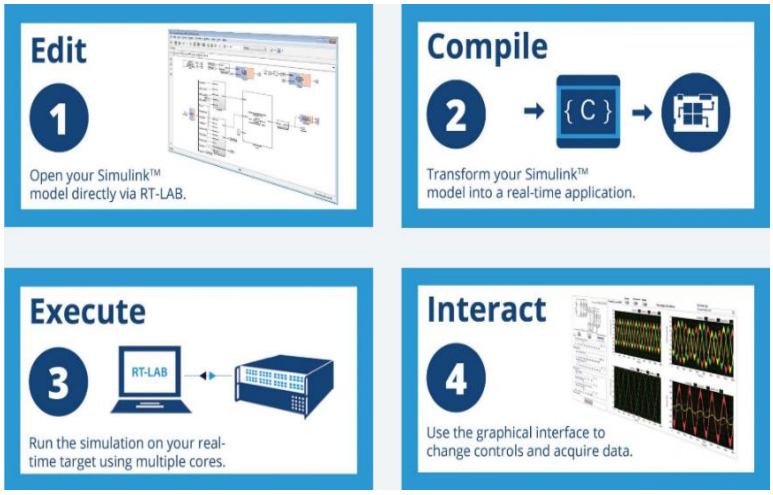


Figure 12: 4 Steps to Real-Time Simulation

3.3.4 CasADi

CasADi [12] is an open-source software framework for non-linear optimization and algorithmic differentiation. It is a general-purpose tool that can be used to prototype formulations, model and solve complex engineering problems with a large degree of flexibility. It facilitates rapid — yet efficient — implementation of different methods for numerical optimal control, both in an offline context and for nonlinear model predictive control (NMPC). It can be coupled with other software like MATLAB and the landscape of its academic and industrial applications is diverse.

3.4. Milestones

1. Profound Research on Related Subjects (papers, books, products etc.)
2. Getting Familiar with CasADi Through Examples in MATLAB
3. Numerically Designing the MPC using CasADi in MATLAB
4. Implementing the MPC controller in Simulink
5. Developing the Control Algorithm in Simulink
6. Installing the Control Algorithm into the Existing Setup
7. Fixing Bugs and Tuning the Control Algorithm for Best Performance
8. Testing Correctness and Accuracy of the Real-Time Behavior
9. Gathering Material and Writing the Final Report
10. Practicing for the Demonstration

3.5. Gantt Chart

Figure 13 illustrates the Gantt Chart.

TASK NAME	DURATION (Initially Planned)	DURATION (Actual Load)
Profound Research on Related Subjects (papers, books, products etc.)	14	10
Getting Familiar with CasADi Through Examples in MATLAB	7	12
Numerically Designing the MPC using CasADi in MATLAB	14	7
Implementing the MPC controller in Simulink	10	13
Developing the Control Algorithm in Simulink	10	13
Installing the Control Algorithm into the Existing Setup	5	11
Fixing Bugs and Tuning the Control Algorithm for Best Performance	7	10
Testing Correctness and Accuracy of the Real-Time Behavior	10	7
Gathering Material and Writing the Final Report	7	5
Practicing for the Demonstration	7	3

Figure 13: Gantt Chart

3.6. Deliverables

Figure 14 shows the deliverables for the mentioned project.

DELIVERABLES	IS IT ACHIEVED?
Functioning HIL Simulator (Real-Time) with a Vehicle Model	YES
Functioning ECU (Real-Time) with a Model Predictive Control Algorithm	YES
Synchronized Behavior of ECU and HIL Simulator	YES
Reference Speed Trajectory Tracking	YES

Figure 14: Deliverables

4. Work Done

The work done will be summarized progressively in the latter chapters. In the first place, a lot of profound reading and research was done about the associated domain. Coming next was to understand the capabilities of the CasADi software. Tutorials and lectures were followed on the subject to grasp the idea better.

4.1 The Theory Behind the Numerical Design of the MPC Strategy

Optimization is an act, process, or methodology of making something (such as a design, system, or decision) as fully perfect, functional, or effective as possible [13]. Optimization algorithms are used in many applications from diverse areas like allocation of resources in logistics, investments in business, estimation, and fitting of models to measurement data from experiments in science, design, and operation of technical systems in engineering etc. When optimization is discussed in engineering, MPC comes to mind as an effective method commonly used in various industries like automotive, energy, industrial manufacturing etc. This begs the question: What is MPC?

Model predictive control (MPC) is a feedback control algorithm that uses a model to make predictions about future outputs of a process. It can deal with multi-input multi-output (MIMO) systems that might have interactions between their inputs and outputs which is particularly difficult to achieve for PID controllers. In addition, it can handle constraints, has preview capability (prediction horizon N is a measure of how far ahead MPC looks into the future) and can easily incorporate future reference information into the control problem to improve controller performance. However, it requires a powerful, fast processor with a large memory since it solves an online optimization problem at each time-step to select the optimal control action from a sequence that drives the predicted output as close to the desired reference as possible. But what characterizes an optimization problem?

An optimization problem consists of three ingredients: an objective function $\phi(\mathbf{w})$, decision variables $\mathbf{w}$ and inequality/equality constraints. There are different mathematical formulations for an optimization problem, but the generic form is a non-linear programming problem (NLP) which can be mathematically formulated as follows:

- An objective function, $\phi(\mathbf{w})$, that shall be minimized or maximized by manipulating $\mathbf{w}$,
- Decision variables, $\mathbf{w}$, that can be chosen, and
- Constraints that shall be respected, e.g. of the form $\boldsymbol{g_1}(\boldsymbol{w}) = 0$ (equality constraints) or $\boldsymbol{g_2}(\boldsymbol{w}) \geq 0$ (inequality constraints).

In an NLP problem $\phi(\mathbf{w})$, $\boldsymbol{g_1}(\cdot)$, and $\boldsymbol{g_2}(\cdot)$ are usually assumed to be differentiable. In some cases, NLP problems can be reduced to:

- Linear Programming (LP) (when $\phi(\cdot)$, $\boldsymbol{g_1}(\cdot)$, and $\boldsymbol{g_2}(\cdot)$ are affine, i.e. these functions can be expressed as linear combinations of the elements of $\mathbf{w}$).
- Quadratic Programming (QP) (when $\boldsymbol{g_1}(\cdot)$, and $\boldsymbol{g_2}(\cdot)$ are affine, but the objective $\phi(\cdot)$ is a linear-quadratic function).

In this sense, MPC problems can be mathematically formulated as such:

- First, the running (stage) costs which characterizes the control objective (with weights Q and R):

$$\ell(\mathbf{x},\mathbf{u}) = \left\|\mathbf{x}_{\mathbf{u}} - \mathbf{x}^{r}\right\|_{\mathbf{Q}}^{2} + \left\|\mathbf{u} - \mathbf{u}^{r}\right\|_{\mathbf{R}}^{2}$$

- Secondly, cost function is the evaluation of the running costs along the whole prediction horizon:

$$J_N(\mathbf{x},\mathbf{u}) = \sum_{k=0}^{N-1} \ell(\mathbf{x}_{\mathbf{u}}(k),\mathbf{u}(k))$$

- Finally, the optimal control problem to find a minimizing control sequence:

$$\begin{aligned}
&\underset{\mathbf{u}}{\text{minimize}}\, J_N(\mathbf{x}_0,\mathbf{u}) = \sum_{k=0}^{N-1} \ell(\mathbf{x}_{\mathbf{u}}(k),\mathbf{u}(k)) \\
&\text{subject to}: \mathbf{x}_{\mathbf{u}}(k+1) = \mathbf{f}(\mathbf{x}_{\mathbf{u}}(k),\mathbf{u}(k)), \\
&\qquad \mathbf{x}_{\mathbf{u}}(0) = \mathbf{x}_0, \\
&\qquad \mathbf{u}(k) \in U, \ \ \forall k \in [0, N-1] \\
&\qquad \mathbf{x}_{\mathbf{u}}(k) \in X, \ \ \forall k \in [0, N]
\end{aligned}$$

An optimal control problem can be converted into a non-linear programming problem with different techniques namely: single-shooting, multiple shooting etc. In single shooting approach, a vector X keeps the optimal state trajectory is recursively filled with non-linear mapping function outputs considering $\boldsymbol{f}(\mathrm{w}, \mathrm{x}_0, \mathrm{t}_0) = \mathrm{x}_0$ as the only equality constraint which is the initial step. Basically, it is a forward-recursion of the initial step. However, with longer prediction horizons, the non-linearity in X propagates.
In multiple shooting, the key idea is to break down the system integrals into short-time intervals, i.e. use the system model as a state constraint at each optimization step. It is a lifted single shooting where the NLP solved is larger but sparser. Lifting in this manner means to reformulate a function with more variables to make it less non-linear. Overall, multiple shooting is considered to be more advantageous since it improves convergence and decreases computation time. Plus, MS allows the user to initialize the state trajectory with a known guess.

Having mentioned the conversion of an optimal control problem (MPC problem) into a non-linear programming problem, next to discuss is how MPC works.

For a single-input single-output system, $x(k+1) = f(x(k), u(k))$ is an equation which represents the next state of the system where f is the model equation (system RHS), $x(k)$ represents the state at instant k and $u(k)$ represents the control action at instant k. Figure 15 depicts the MPC strategy on a graph. In steps, MPC working principle can be summarized like so:

1. At decision instant k, measure $x(k)$
2. Based on $x(k)$, compute the optimal sequence of controls over a prediction horizon N:
3. $u^*(x(k)) := (u^*(k), u^*(k+1), \ldots, u^*(k+N-1))$
4. Apply the control $u^*(k)$ on the sampling period $[k\ k+1]$
5. Repeat the same steps at the next decision instant

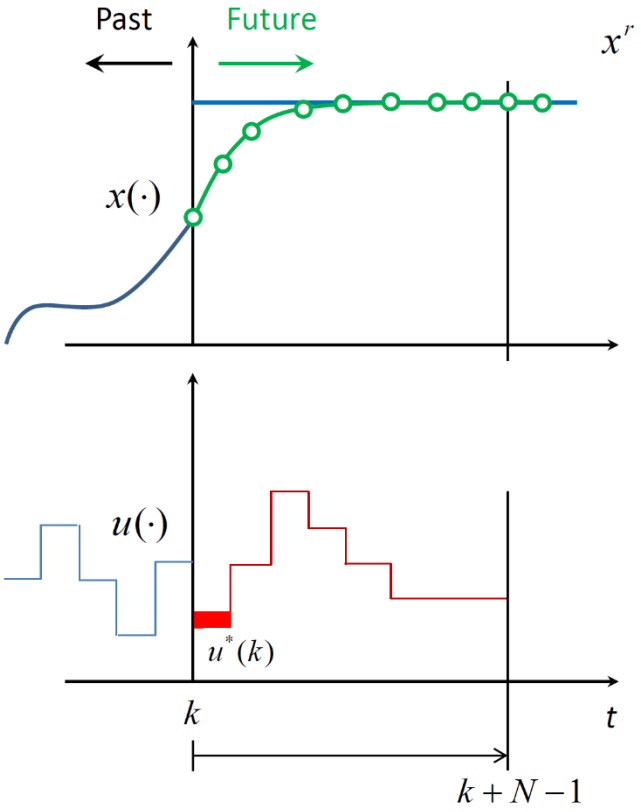


Figure 15: MPC Strategy on a Graph

All in all, there are a few principles to keep in mind while designing an MPC controller.

- The selected sample time **T** (meaning controller's execution rate) should be in between 5-10% of the system's rise-time (time it takes for the response to rise from 10% to 90% of the steady-state response). If it is bigger than this value, the system is vulnerable to disturbance. If smaller, then the computational load increases.
- The chosen prediction horizon **N** multiplied by sample time **T** must be bigger than the system's settling-time (time it takes for the error to fall within 2% of steady-state value). If it is smaller than this value, the system is vulnerable to disturbance. If bigger, then the computational power is wasted.
- The selected control horizon must be at least 2-3 time-steps. If too big, computational load augments, if too small accuracy decreases. Notably, the first time-steps of the control horizon has bigger impact than the rest.
- Having hard constraints (meaning that it cannot be violated) both at input and output should be avoided. Ideally, output constraints should be soft.
- Weights (present in the cost function) should be tuned precisely so that the controller is accurate and smooth. Otherwise, inaccurate behavior will not be penalized accordingly resulting in suboptimal control.

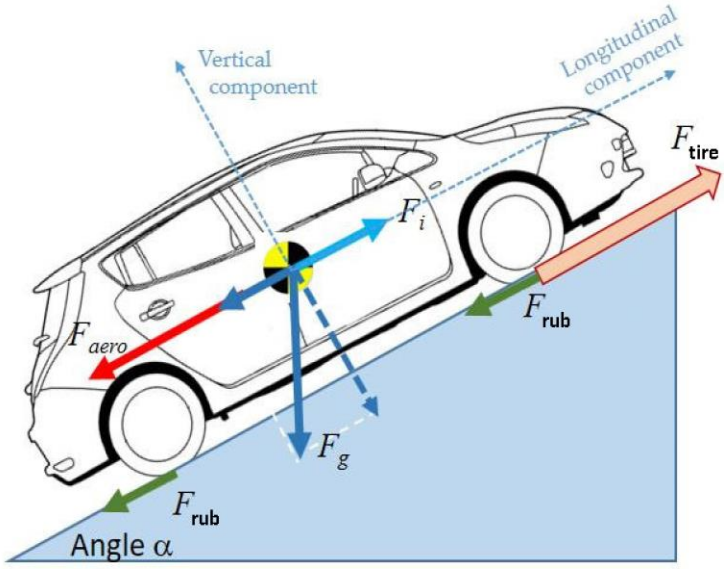
4.2 NLP Problem Formulation and MPC Implementation using CasADi

As mentioned in the previous chapter, optimal control problems can be discretized into non-linear programming problems using multiple shooting approach. In this chapter, the equations used to represent our model will be shared to formulate the problem. Later, a standalone MPC controller working in MATLAB will be presented.

Vehicle model dynamics were represented in equations to help with the numerical design of the MPC controller. Here are the parameters descriptions for the equations in Figure 16:

Symbol	Value	Unit	Description
R_{tire}	0.354	m	Radius of Tire
g	9.81	m/s^2	Earth Gravitational Constant
μ	0.009	N	Road Friction
C_d	0.55	-	Air Drag Coefficient
A	5.1	m^2	Vehicle Frontal Area
mss	4200	kg	Vehicle Mass
ng	0.85	-	Tire Specific Coefficient
Im	0.18	$kg.m^2$	Motor Inertia
ρ	1.225	kg/m^3	Density of Air
G_b	18.5	-	Total Gear Ratio

Figure 16: Vehicle Dynamics' Parameters Description



Vehicle dynamics represent the motion of a point mass which is based on Newton's second law of motion:

$$\sum f = m \cdot \frac{dv}{dt}$$

The forces which act on a vehicle moving in the longitudinal direction are represented in Figure 17 and given below:

- Aero-dynamic drag force $\mathbf{F_{aero}}$
- Force on tire $\mathbf{F_{tire}}$
- Rolling resistance force $\mathbf{F_{rub}}$

Figure 17: Forces Which Act on a Vehicle Moving in the Longitudinal Direction

Equations below determine how these forces can be calculated:

$$F_{aero} = \frac{1}{2} \cdot \rho \cdot C_D \cdot A \cdot V^2$$

$$F_{rub} = \mu \cdot mss \cdot g$$

$$F_{tire} = G_b \cdot Trq/R_{tire}$$

Subsequently, these values can be plugged in the equation below to compute the vehicle's acceleration:

$$\frac{dv}{dt} = \frac{(F_{tire} - F_{rub} - F_{aero} - F_{brake} - (mss \cdot G \cdot \sin\alpha))}{mss + \frac{Im \cdot {G_b}^2}{ng \cdot {R_{tire}}^2}}$$

In our case, the controller will try to manipulate torque requests to reach certain driving speeds along the speed profile. Therefore, our only state is the vehicle speed and our only decision variable is the torque request. In optimal control problems, there can be 2 control objectives: point stabilization or trajectory tracking. Point stabilization means that the reference values of the state vector are constant over the control period while trajectory tracking implies time-varying reference values of the state vector. Since the vehicle follows a particular speed profile, the speed setpoints update every single instance indicating a trajectory tracking problem. Also, during the journey, road slope will change which means that our constraints will update at every instance.

The Euler discretization equation of our OCP is as such: $v(k+1) = v(k) + \Delta T \cdot \frac{dv}{dt}$

By following this guideline, an MPC controller was developed (using CasADi) and was simulated in Simulink. Model screenshot as Figure 18 and simulation results as Figure 19, 20 and 21 can be found on the next pages. The model presented here is adapted for a quicker simulation purpose. The MATLAB System block which represents the MPC controller receives all the inputs as matrices from constant blocks and chooses the appropriate ones according to simulation time. The model utilized for the System RHS is identical to the model used in the MPC controller.

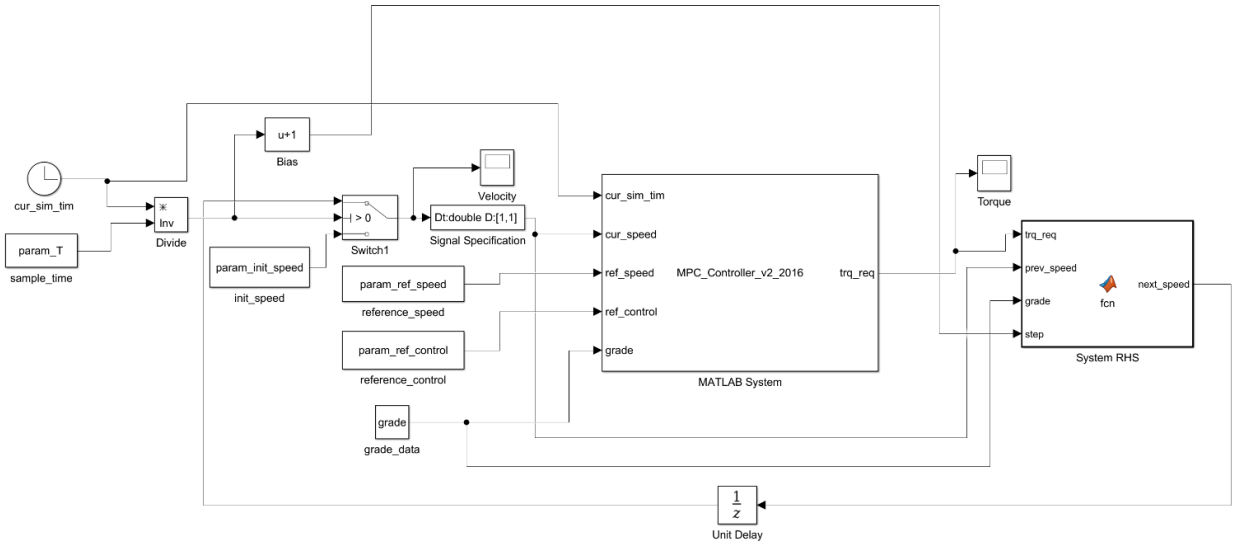


Figure 18: MPC Block Model

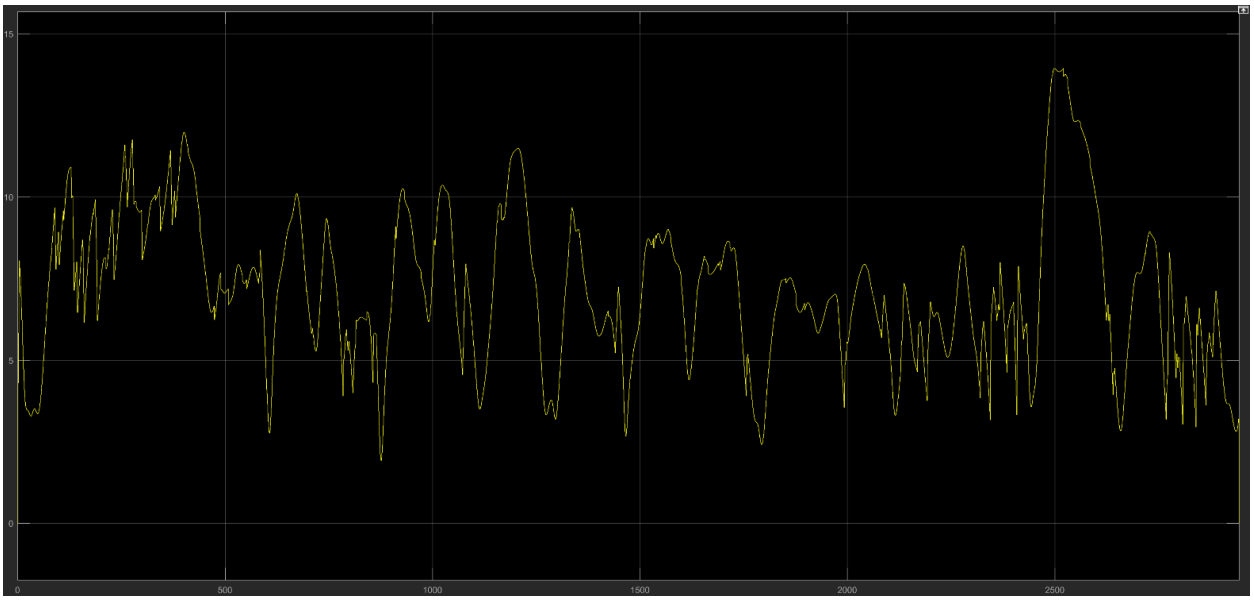


Figure 19: Reference Speed (MPC Simulation)

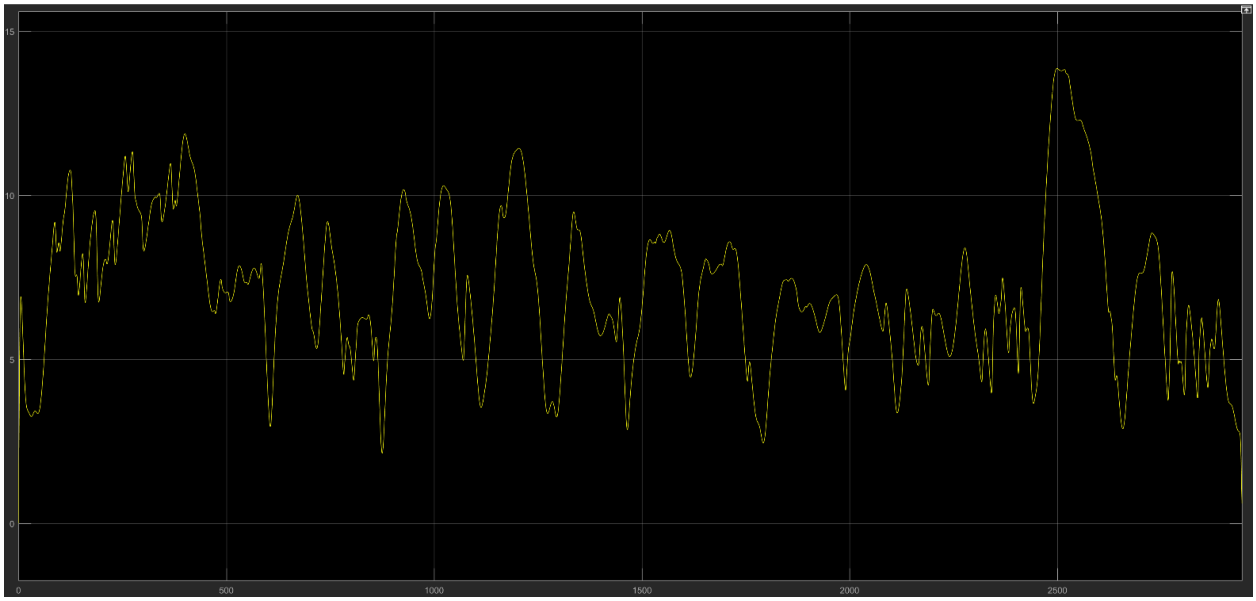


Figure 20: Vehicle Speed (MPC Simulation)

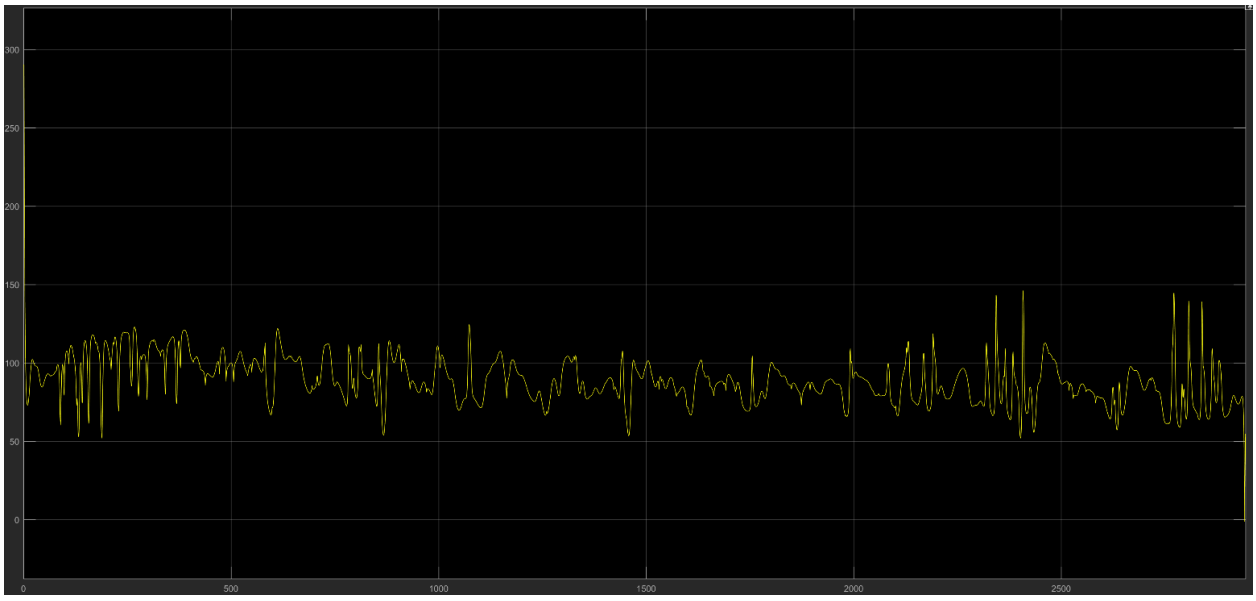


Figure 21: Torque Requests (MPC Simulation)

4.3 Building Process of the MPC Implementation using CasADi

Our target ECU is programmed via the calibration tool called PiSnoop using 2 files, namely: the binary image of the application (.hex) and an ASAP2 file (.a2l) which provides access to the application's C variables while the application runs on it. The application development software named OpenECU, has a highly automated build process to generate these files for our hardware. It is well-integrated with the Real-Time Workshop in MATLAB which is a tool for code generation for Simulink models, extensive model-based debugging support, extensible make processes and target support. The typical workflow for the automated build process is illustrated below as Figure 22. The steps are as follows:

1. **Model Compilation:** Real-Time Workshop analyzes the block diagram (and any models referenced by Model blocks) and compiles an intermediate hierarchical representation in a file called model.rtw.

2. **Code Generation:** The Target Language Compiler reads model.rtw, translates it to C code, and places the C file in a build directory within the working directory.

3. **Customized Makefile Generation:** Real-Time Workshop constructs a makefile from the appropriate target makefile template and writes it in the build directory.

4. **Executable Program Generation:** The system's make utility reads the makefile to compile source code, link object files and libraries, and generate an executable image (called model or model.exe). The makefile places the executable image in the working directory.

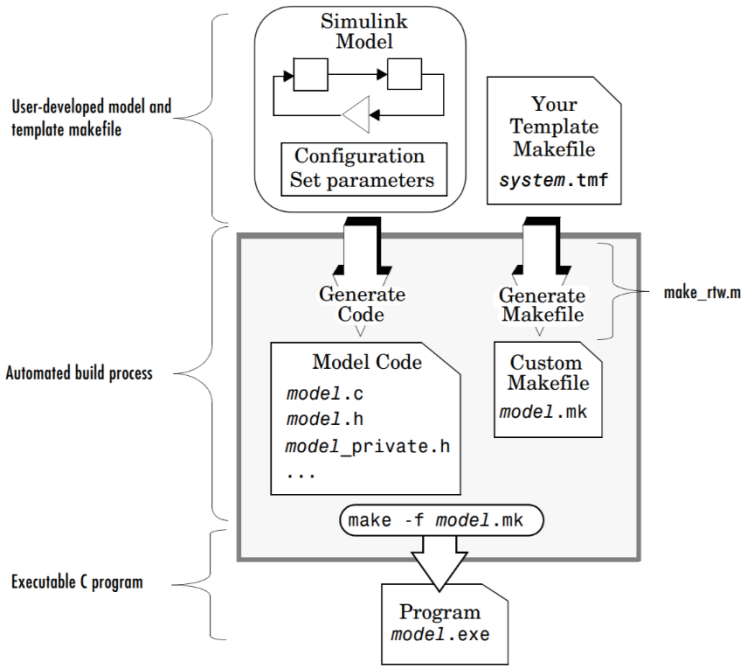


Figure 22: Real-Time Workshop Build Process

In the beginning, it was planned to implement an MPC strategy using CasADi software and embed this application in the target ECU for a real-time simulation. However, since the final product goes through this automated process, it was required to make the design compatible with it.

In order to be able to use CasADi code inside the application, import commands were included which aren't supported by MATLAB for code generation. Therefore, the algorithm was enveloped in a function and C/C++ code was generated from it by using CasADi's capabilities. Next, it was necessary to have an S-Function and its in-lining TLC file to integrate the existing C/C++ code into the model. The TLC files are ASCII files that explicitly control the way code is generated by Real-Time Workshop. By editing a TLC file, the way how code is generated can be altered for a particular block. Nonetheless, writing a TLC file from scratch is a very difficult task.

An alternative was to use the legacy_code function which creates a MATLAB structure for registering the specification for existing C or C++ code and the S-function being generated. In addition, the function can generate, compile and link, and create a masked block for the specified S-function. It can also create a TLC file for code generation or a rtwmakecfg.m file that can be customized to specify dependent source and header files that reside in a different directory than that of the generated S-function. Unfortunately, even though this approach seemed constructive, all the efforts were fruitless due to not knowing how to use this function effectively. As a result, methodology was changed, and the design of the MPC was decided to be done using built-in Simulink blocks along with MPC Designer App purely in MATLAB.

4.4 MPC Implementation using built-in Simulink Blocks and MPC Designer App

To design an MPC controller in Simulink, various types of MPC blocks can be used such as basic, gain-scheduled, adaptive, explicit, and non-linear. When we have a linear system with linear constraints and a quadratic cost function, LTI MPC controllers should be used which are the gain-scheduled and the adaptive MPC blocks. Gain-scheduled MPC is used when the number of states and constraints change across different operating conditions. The linearization of points is done offline and a linear MPC with different number of states and constraints are designed for each operating point. Since these MPCs are independent, an algorithm is used to switch between them which uses more memory. On the other hand, adaptive MPC is used when the structure of the optimization problem remains the same across different operating conditions. The linear model is computed on the fly as the operating conditions change while also updating the internal plant model used by the MPC.

Although, if we have a non-linear system with non-linear constraints and a non-linear cost function, non-linear MPC controllers should be used since they provide more accurate results with a higher computational cost. Modern system identification technique is based on data-driven optimization and machine learning to build models from data. Then, use those with MPC to stabilize and track reference trajectories for strongly non-linear systems.

Lastly, if a fast MPC solution is desired, explicit MPC can be used. It solves the optimization problem offline for all the states within a given range. Solutions consist of linear functions that are piecewise affine and continuous in x. Any constraint cuts the solution space in regions. Each region maps into a unique optimal solution. All MPC does online is to find the region current state lies in and evaluate the linear function to create the current control action.

In our case, it can be concluded that a basic MPC controller is enough since only reference values which are the speed setpoints and the slope of the road change at each time-step of the simulation while the number of states and constraints stay the same. Looking at our equation which gives us the acceleration rate after plugging in the parameter values, it can be concluded that this system is linearizable due to the coefficient of the quadratic term being quite small hence negligible.

Since the built-in MPC blocks require an internal plant model, a state-space representation of the plant was created. A state-space model is a mathematical representation of a physical system as a set of input, output, and state variables related by first-order differential equations. Discrete-time state-space model is of the following form:

$$x[n+1] = Ax[n] + Bu[n]$$

$$y[n] = Cx[n] + Du[n]$$

x vector represents the states, u vector describes the inputs while the y vector is for the output. First equation indicates how the system evolves state-by-state with the effect of inputs. Second equation is for expressing how the output is related to the current state with a possible inclusion of feedthrough of inputs. Here are the characteristics of the state-space model in Figure 23:

Discrete-Time State-Space Model Characteristics	
Sample Time	0.1 seconds
A	1
B	0.00010937
C	3.6
D	0
Manipulated Inputs	1
Measured Outputs	1

Figure 23: Discrete-Time State-Space Model Characteristics

Using the MPC designer app, the controller was tested and tuned with different scenarios. It was exported with the specifications below as Figure 24.

MPC Controller Design Characteristics	
Prediction Horizon	50
Control Horizon	4
Sample Time	0.1
Nominal Value for Inputs	0
Nominal Value for Outputs	0
Overall Adjustment Factor Applied to Weights	7.3891
Overall Adjustment Factor Applied to Estimation Model Gains	1.6596
Minimum Constraint for States	0
Maximum Constraint for States	51
Minimum Constraint for Inputs	-290
Maximum Constraint for Inputs	290

Figure 24: MPC Controller Design Characteristics

Using this MPC controller with the presented plant model, a control algorithm was developed to be embedded in our target ECU. Figure 25 illustrates the general schema of the control algorithm.

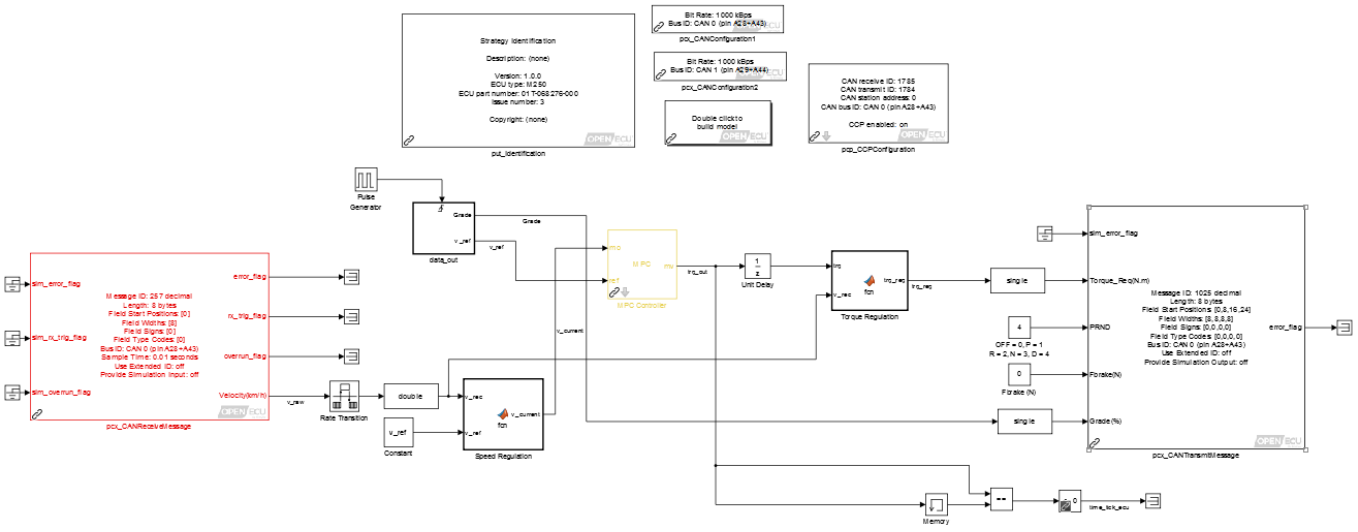


Figure 25: General Schema of the Control Algorithm

4.5 Method of Functioning of the Control Loop

Besides the battery, motor and inverter are the most important parts of an electric vehicle. The inverter block receives a torque request from the driver and generates a suitable amount of current to drive the electric motor so that the vehicle accelerates as required [14]. Here is a depiction of the EV model as Figure 26. Figure 27 shows the vehicle dynamics block in the EV model.

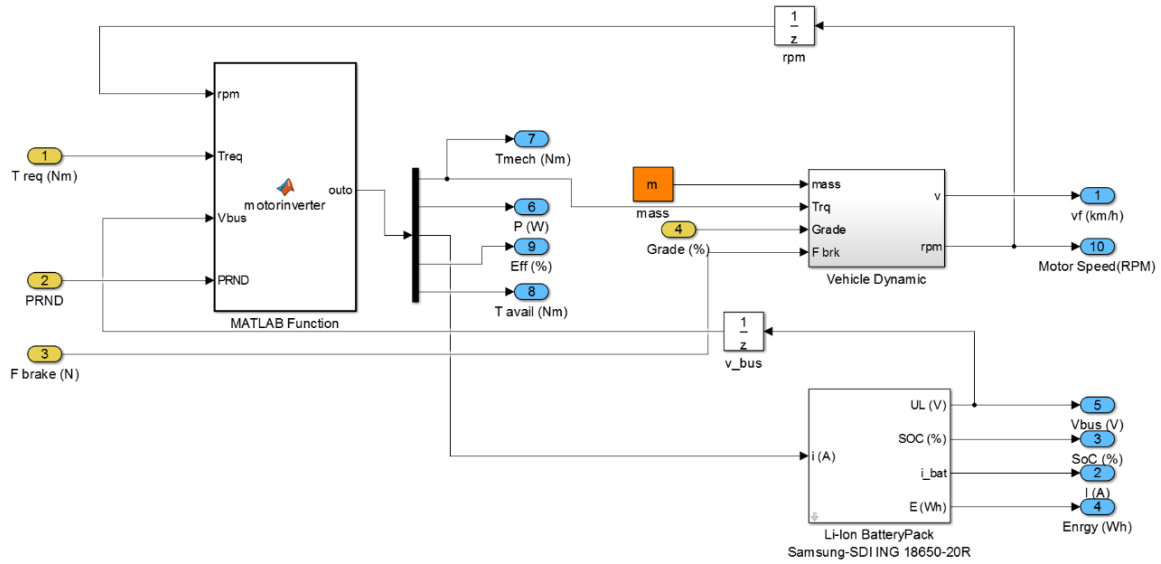


Figure 26: EV Model

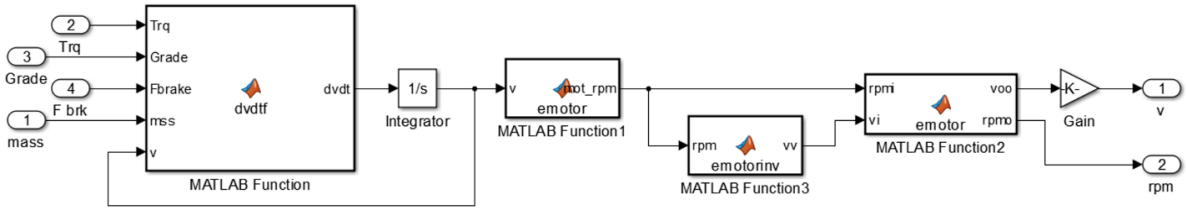


Figure 27: EV Model - Vehicle Dynamics Block

The Simulink model that consists of these blocks has a sample time of 0.001s. But the vehicle model execution rate is at 0.01s meaning that number of steps without overruns for this model must be 10. Figure 28 mentions the HIL model execution properties.

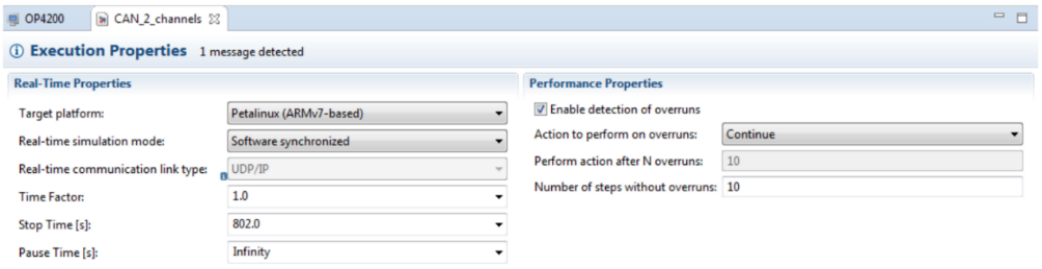


Figure 28: HIL Model Execution Properties

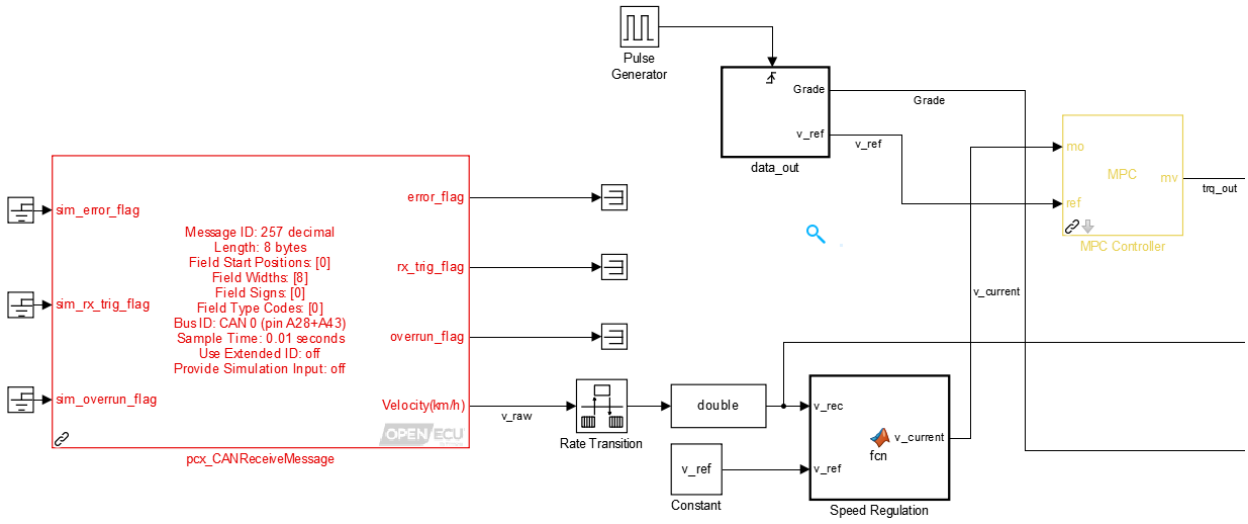


Figure 29: Control Algorithm - Part1

Having our vehicle speed updated every 0.01 seconds allows the HIL side to behave more like a real plant. Also, it means that our ECU should receive messages from CAN bus every 0.01s. But since our controller's sample time is 0.1s, desynchronization might occur. To allow, fast-to-slow transmission in the model, a rate transition block is added. This block acts as a zero-order hold to adjust data transmission rate and ensures data integrity during its transfer. Later, a speed regulation function is added to prevent any mismatches. Due to reference speed data and grade data having 1s periods, a pulse generator with a 1s period and a 10% pulse width is added to trigger the block at the right rate to provide reference speed and grade data input. Figure 29 and 30 shows a better image representation of the control algorithm.

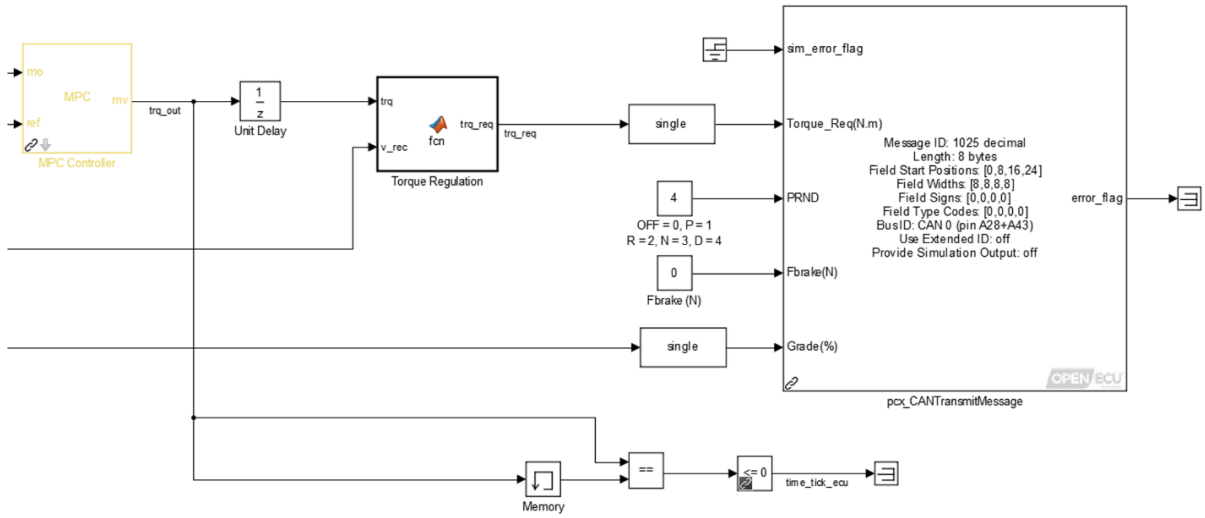


Figure 30: Control Algorithm - Part2

After the MPC block, the torque output is regulated as well with a function block. Finally, the ECU time tick is calculated, and the data is transmitted via the CAN bus.

4.6 Simulation Results

It is fair to say that the design is quite optimal and works well. The trajectory is decently tracked with a small margin of error. In addition, the behavior was observed to be synchronized by variable data logging on the diagnostics tool. The simulation was run for 500 seconds. The results are illustrated below as Figure 31 and Figure 32.

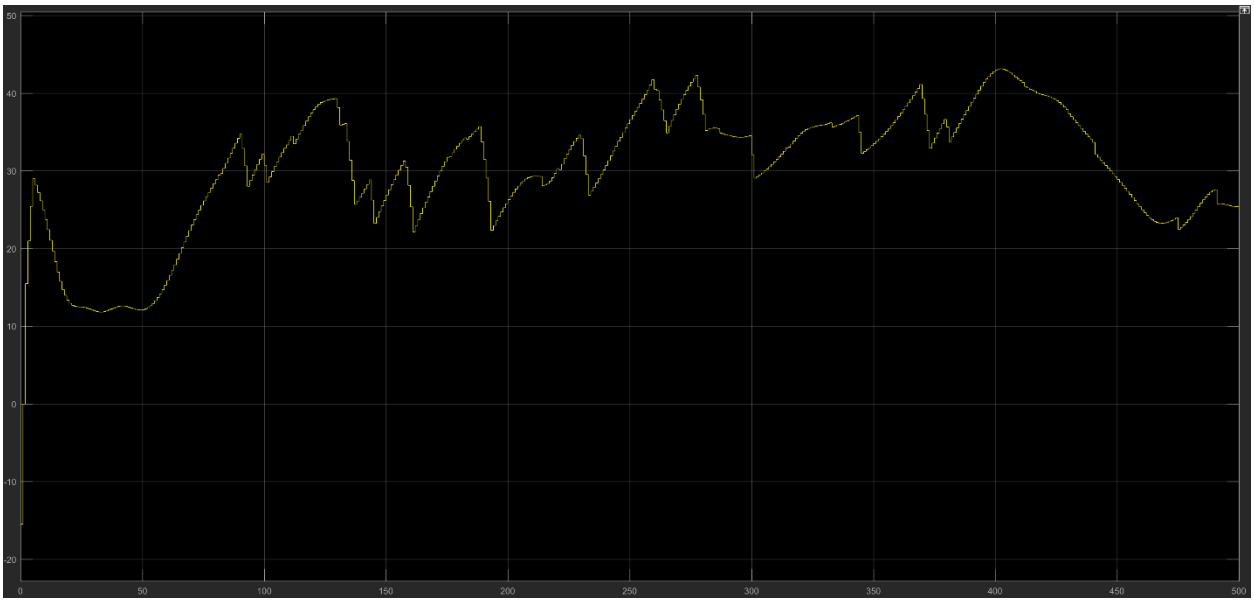


Figure 31: Reference Speed Trajectory

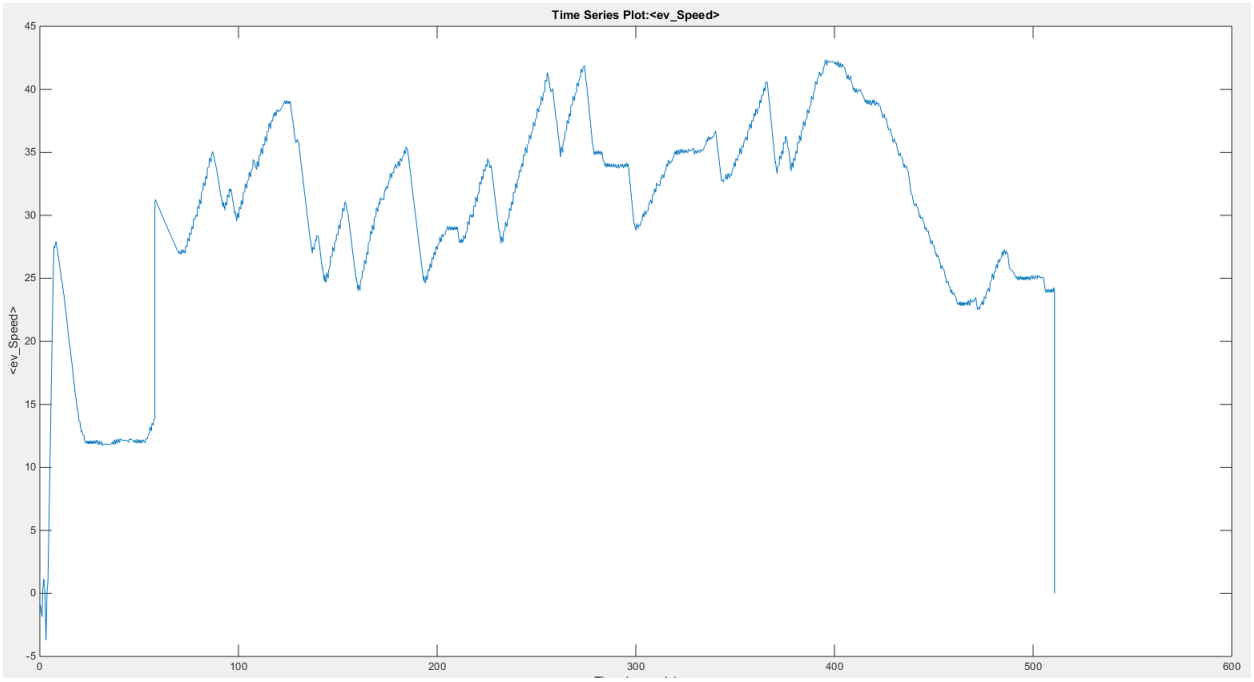


Figure 32: EV Speed(km/h)

Appendix

MPC Controller Design Using MATLAB & Simulink

```
% clear all
% clc
load('plant_EV.mat');
T = 0.1; %controller sample time
% %% load MAT files
% load('X'); load('Y'); load('Z'); load('Speed'); load('Torque');
% x = X(2,:); y = Y(2,:); z = Z(2,:);
%
% %% generate grade values over the trajectory
% traj_len = length(x);
% t = 1:1:traj_len;
% for i = 1:traj_len
%     grade_data(i) = acos(sqrt((x(i)^2) + (y(i)^2)) / sqrt((x(i)^2) + (y(i)^2) + (z(i)^2))) * 100;
% end
% grade = [t', grade_data'];
% %% create state space model of the plant
% A = 1.00; B = 0.00010937; C = 3.60; D = 0;
% plant_EV = ss(A, B, C, D, 0.01, 'InputName', 'trq', 'OutputName', 'v', 'StateName', {'v'},...
% 'StateUnit',{'m/s'}, 'InputUnit', {'N.m'}, 'OutputUnit', {'km/h'});
% plant_EV.InputGroup.Manipulated = 1;
% plant_EV.OutputGroup.Measured = 1;
%% create MPC controller object with sample time
mpc_controller_v1 = mpc(plant_C, T);
%% specify prediction horizon
mpc_controller_v1.PredictionHorizon = 50;
%% specify control horizon
mpc_controller_v1.ControlHorizon = 4;
%% specify nominal values for inputs and outputs
mpc_controller_v1.Model.Nominal.U = 0;
mpc_controller_v1.Model.Nominal.Y = 0;
%% specify overall adjustment factor applied to weights
beta = 7.3891;
%% specify weights
mpc_controller_v1.Weights.MV = 0*beta;
mpc_controller_v1.Weights.MVRate = 0.1/beta;
mpc_controller_v1.Weights.OV = 1*beta;
mpc_controller_v1.Weights.ECR = 100000;
%% specify overall adjustment factor applied to estimation model gains
alpha = 1.6596;
%% adjust default output disturbance model gains
setoutdist(mpc_controller_v1, 'model', getoutdist(mpc_controller_v1)*alpha);
%% adjust default measurement noise model gains
mpc_controller_v1.Model.Noise = mpc_controller_v1.Model.Noise/alpha;
```

MPC Controller Design Using CasADi

```
classdef MPC_Controller_v2_2016 < matlab.System & matlab.system.mixin.Propagates

% This MATLAB System Block formulates the non-linear programming problem for the MPC Implementation.
% It initializes and outputs the solver once later to be evaluated.

% This template includes the minimum set of functions required
% to define a System object with discrete state.

properties
%By default, these are public, tunable properties meaning
%their value can change at any time of the simulation.

end

properties(Nontunable)
% Nontunable attribute for a property is used when the algorithm
% depends on the value being constant once data processing starts.

%S = 0.01;   % S*(Next Control - Current Control)^2

% 1. Constants for MPC Design %
% 2. Weighing Matrices for States and Controls %
% 3. Constants for Vehicle Dynamics %

    T = 1;             % Sample Time (s)
    N = 5;               % Prediction Horizon (# of steps)

    Q = 5;               % Q*(Current State - Reference State)^2
    R = 0.0001;          % R*(Current Control - Reference Control)^2

    Fbrake = 0;          % Braking Force (N)
    Trajlen = 2500;      % Trajectory Length (# of steps)
end

properties (DiscreteState)
%If your algorithm uses properties that hold state,
%you can assign those properties the DiscreteState attribute.

% Updated Variables at Each Time-Step
    X0;          % previous states over the horizon   (predicted)
    u0;          % previous controls over the horizon (predicted)
end
```

```
properties (Access = private)
% Conversion Metrics
        % 1 rad/s = 1/2*pi Hz = 60/2*pi RPM
        % 1 m/s   = 3.6 km/h
        % 1 hP    = 745.7 W

% Constants for Vehicle Dynamics
    Rtire;       % radius of tire(m)
    G;           % earth's gravitational constant(m/s2)
    mu;          % road friction(N)
    rho;         % density of air as a fluid(kg/m3)
    Cd;          % air drag coefficient
    A;           % vehicle frontal area(m2)
    mss;         % vehicle mass(kg)
    ng;          % tire-specific constant
    Im;          % motor inertia(kg*m2)
    Gb;          % total gear ratio
% Symbols for States, Controls and Constraints
    states;      % velocity(m/s)
    controls;    % torque(N*m)
% Variables for Vehicle Dynamics
    alpha;       % degree of slope(rad)
    Faero;       % drag force against vehicle(N)
    Frub;        % rubbing force against tires(N)
    Ftire;       % driving force(N)
    dvdt;        % acceleration(m/s2)
% Objective Function and Constraints Vector
    J;           % objective function
    g;           % constraints vector
% Non-Linear Mapping Function f(x,u) and System R.H.S
    f;       % non-linear mapping function
    U;       % decision variables (controls)
    X;       % a vector that represents the states over the optimization problem
    P;       % parameters (which include the initial state and the reference along
             % the predicted trajectory (reference states and reference controls))
% Non-Linear Programming Problem Initialization
    opt_variables  % optization variables
    nlp_prob       % non-linear problem structure initalization
    opts           % simulation properties
% Pre-computed Functions.
    rpm_calc
    solver
end
```

```
methods (Access = protected)
    function [sz, dt, cp] = getDiscreteStateSpecificationImpl(obj, name)
        if strcmp(name, 'X0')
            sz = [obj.N + 1, 1];
            dt = 'double';
            cp = false;
        elseif strcmp(name, 'u0')
            sz = [obj.N, 1];
            dt = 'double';
            cp = false;
        else
            error(['Error: Incorrect State Name: ', name.']);
        end
    end
    function num = getNumInputsImpl(~)
        num = 5; %cur_sim_tim, cur_speed, ref_speed, ref_control, grade
    end
    function num = getNumOutputsImpl(~)
        num = 1; %trq_req
    end
    function [dt1] = getOutputDataTypeImpl(~)
        dt1 = 'double'; %torque request
    end
    function [dt1, dt2, dt3, dt4, dt5] = getInputDataTypeImpl(~)
        dt1 = 'double'; %current simulation time
        dt2 = 'double'; %current speed
        dt3 = 'double'; %reference speed
        dt4 = 'double'; %reference control
        dt5 = 'double'; %grade
    end
    function [sz1] = getOutputSizeImpl(~)
        sz1 = [1, 1];
    end
    function [sz1, sz2, sz3, sz4, sz5] = getInputSizeImpl(obj)
        sz1 = [1, 1];
        sz2 = [1, 1];
        sz3 = [1, obj.Trajlen];
        sz4 = [1, obj.Trajlen];
        sz5 = [1, obj.Trajlen];
    end
```

```
function [cp1, cp2, cp3, cp4, cp5] = isInputComplexImpl(~)
    cp1 = false;
    cp2 = false;
    cp3 = false;
    cp4 = false;
    cp5 = false;
end
function [cp1] = isOutputComplexImpl(~)
    cp1 = false;
end
function [fz1, fz2, fz3, fz4, fz5] = isInputFixedSizeImpl(~)
    fz1 = true;
    fz2 = true;
    fz3 = true;
    fz4 = true;
    fz5 = true;
end
function [fz1] = isOutputFixedSizeImpl(~)
    fz1 = true;
end
```

```
function setupImpl(obj)
    % Implement tasks that need to be performed only once,
    % such as pre-computed constants.

    % CasADi v3.5.1
    addpath('C:\Program Files\casadi-windows-matlabR2016a-v3.5.1')
    import casadi.*

    % Updated Variables at Each Time-Step
    obj.u0 = zeros(obj.N, 1);
    obj.X0 = zeros(obj.N + 1, 1);

    % Constants for Vehicle Dynamics
    obj.Rtire  =  0.354;  obj.G      =  9.81;
    obj.mu     =  0.09;   obj.rho    =  1.225;
    obj.Cd     =  0.55;   obj.A      =  5.1;
    obj.mss    =  4200;   obj.ng     =  0.85;
    obj.Im     =  0.18;   obj.Gb     =  18.5;

    % Symbols for States, Controls and Changing Constraints
    obj.states = SX.sym('v'); n_states = length(obj.states);
    obj.controls = SX.sym('trq'); n_controls = length(obj.controls);

    % Variables for Vehicle Dynamics
    obj.alpha     =  atan(0 / 100);
    obj.Faero     =  0.5*obj.rho*obj.A*obj.Cd*(obj.states^2);
    obj.Frub      =  obj.mu*obj.mss*obj.G;
    obj.Ftire     =  obj.controls*obj.Gb / obj.Rtire;
    obj.dvdt      =  (obj.Ftire - obj.Frub - obj.Faero - obj.mss*obj.G*sin(obj.alpha)...
        - obj.Fbrake)/(obj.mss + obj.Im*obj.Gb^2 / (obj.ng*obj.Rtire^2));

    % Objective Function and Constraints Vector
    obj.J = 0; obj.g = [];

    % Non-Linear Mapping Function f(x,u) and System R.H.S
    obj.f = Function('f', {obj.states, obj.controls}, {obj.dvdt});
    obj.U = SX.sym('U', n_controls, obj.N);
    obj.X = SX.sym('X', n_states, (obj.N + 1));
    obj.P = SX.sym('P', n_states + obj.N*(n_states + n_controls));
```

```
    % Computation of the Objective Function

    % Caching of the System Object Data in Local Variable Before
    % Multiple Access in Each Iteration in a Loop

    for k = 1:obj.N
        current_state = obj.X(k); current_control = obj.U(k);

        %if k == obj.N
        %    next_control = U(k);
        %else
        %    next_control = U(k+1);
        %end

        obj.J = obj.J + ((current_state - obj.P(2*k))'*obj.Q*(current_state - obj.P(2*k))) + ...
                  ((current_control - obj.P(2*k+1))'*obj.R*(current_control - obj.P(2*k+1))); %+ ...
                  %((next_control-current_control)*obj.S*(next_control-current_control));
    end

    % Non-Linear Programming Problem Initialization
    obj.opt_variables = [reshape(obj.X, obj.N+1, 1);reshape(obj.U, obj.N, 1)];
    %obj.nlp_prob = struct('f', obj.J, 'x', obj.opt_variables, 'g', obj.g, 'p', obj.P);

    obj.opts = struct;
    obj.opts.ipopt.max_iter = 2000;
    obj.opts.ipopt.print_level =0;%0,3
    obj.opts.print_time = 0;
    obj.opts.ipopt.acceptable_tol =1e-8;
    obj.opts.ipopt.acceptable_obj_change_tol = 1e-6;

    rpm = (obj.states/obj.Rtire)*(60/(2*pi))*obj.Gb; % revolutions per minute(rpm)

    obj.rpm_calc  = Function('rpm_calc', {obj.states},{rpm});
    %obj.solver    = nlpsol('solver', 'ipopt', obj.nlp_prob, obj.opts);
end
```

```
function [trq_req] = stepImpl(obj, cur_sim_tim, cur_speed,...
        ref_speed, ref_control, grade)
    import casadi.*
    inst = uint16((cur_sim_tim/obj.T) + 1); % kth instance of simulation

    if inst > length(ref_speed) - obj.N
        xs = ref_speed(inst:end);
        cs = ref_control(inst:end);
        gs = grade(inst:end);

        check = (length(ref_speed) - inst + 1); %to check how many
        %reference velocity points are supposed to be added to have
        %exact N points at every step

        for iter = check:obj.N-1 %array completion
            xs(length(xs)+1) = xs(end);
            cs(length(cs)+1) = cs(end);
            gs(length(gs)+1) = gs(end);
        end
    else
        xs = ref_speed(inst:(inst+obj.N-1));   % reference speed
        cs = ref_control(inst:(inst+obj.N-1)); % reference control
        gs = grade(inst:(inst+obj.N-1));       % grade data
    end

      motor_inv_max_trq_output   = [250,250,249,243,215,162,132.5,112.5,97,84,79.6];
      motor_inv_max_rpm_input    = [0,4000,5000,5500,6000,7000,8000,9000,10000,11000,12000];

      maxAvailableTrq = interp1(motor_inv_max_rpm_input,motor_inv_max_trq_output,full(obj.rpm_calc(cur_speed)));

      if  isnan(maxAvailableTrq)
          maxAvailableTrq = 250;
      end

    maxAvailableTrq = 290;

    v_max    = 55;  v_min   = 0;
    trq_max =  maxAvailableTrq; trq_min = -maxAvailableTrq;

    calc_constraints_vect(obj,gs);
    obj.nlp_prob = struct('f', obj.J, 'x', obj.opt_variables, 'g', obj.g, 'p', obj.P);
    obj.solver   = nlpsol('solver', 'ipopt', obj.nlp_prob, obj.opts);
```

```
    args = struct;

    %Equality Constraints
    args.lbg(1:obj.N+1) = 0;
    args.ubg(1:obj.N+1) = 0;

    %States Upper and Lower Bounds
    args.lbx(1:obj.N+1) = v_min; %velocity lower bound
    args.ubx(1:obj.N+1) = v_max; %velocity upper bound

    %Controls Upper and Lower Bounds
    args.lbx((obj.N+1)+1:(2*obj.N)+1) = trq_min; %torque lower bound
    args.ubx((obj.N+1)+1:(2*obj.N)+1) = trq_max; %torque upper bound

    args.p(1) = cur_speed; % initial condition of the robot posture
    for k = 1:obj.N %new - set the reference to track
        args.p(2*k)   = xs(k);
        args.p(2*k+1) = cs(k);
    end

    args.x0  = [reshape(obj.X0',(obj.N+1),1);reshape(obj.u0',obj.N,1)];

    sol = obj.solver('x0', args.x0, 'lbx', args.lbx, 'ubx', args.ubx,...
    'lbg', args.lbg, 'ubg', args.ubg,'p',args.p);

    u = reshape(full(sol.x((obj.N+1)+1:end))',1,obj.N)'; %control actions for the given prediction horizon
    obj.X0 = full(sol.x(1:(obj.N+1))); %predicted states
    trq_req = u(1); %first control action is output to be applied
    obj.u0 = [u(2:size(u,1));u(size(u,1))]; %shift controls to initialize the next step
    obj.X0 = [obj.X0(2:end);obj.X0(end)]; % shift trajectory to initialize the next step
end
```

```
    function calc_constraints_vect(obj, grade_data)
        import casadi.*
        g_rep = [];              Ftire_rep = obj.Ftire;  Frub_rep  = obj.Frub;   Fbrake_rep = obj.Fbrake;
        ng_rep = obj.ng;         Faero_rep = obj.Faero;  mss_rep = obj.mss;      U_rep = obj.U;
        G_rep = obj.G;           Im_rep = obj.Im;        Gb_rep = obj.Gb;        N_rep = obj.N;
        Rtire_rep = obj.Rtire;   X_rep = obj.X;          T_rep = obj.T;          P_rep = obj.P;

        init_state = X_rep(1);                  % initial state
        g_rep = [g_rep; (init_state - P_rep(1))]; % initial condition constraints
        for k = 1:N_rep
            alpha_rep = atan(grade_data(k) / 100);
            dvdt_rep = (Ftire_rep - Frub_rep - Faero_rep - mss_rep*G_rep*sin(alpha_rep) - Fbrake_rep)/(mss_rep + Im_rep*Gb_rep^2 / (ng_rep*Rtire_rep^2));
            func = Function('func', {obj.states, obj.controls}, {dvdt_rep});

            current_state = X_rep(k); current_control = U_rep(k);
            f_value  = func(current_state,current_control);
            next_state = X_rep(k+1);
            next_state_pred = current_state + (T_rep*f_value);
            g_rep = [g_rep; next_state - next_state_pred];
        end
        obj.alpha = alpha_rep;
        obj.dvdt  = dvdt_rep;
        obj.g = g_rep;
    end

    function resetImpl(~)
        % Initialize discrete-state properties.
    end
end
end
```

References

[1] Climate Change Evidence: How Do We Know? (2020, May 27). Retrieved November 11, 2019, from https://climate.nasa.gov/evidence/

[2] Global Emissions. (2020, January 07). Retrieved November 11, 2019, from https://www.c2es.org/content/international-emissions/

[3] Retrieved November 11, 2019, from https://www.bloomberg.com/features/2016-ev-oil-crisis/

[4] Bassett, Mike & Brods, Bruno & Hall, Jonathan & Borman, Stephen & Grove, Matthew & Reader, Simon. (2015). GPS Based Energy Management Control for Plug-in Hybrid Vehicles. SAE Technical Papers. 2015. 10.4271/2015-01-1226.

[5] Chen, Yan & Li, Xiaodong & Wiet, Chris & Wang, Junmin. (2014). Energy Management and Driving Strategy for In-Wheel Motor Electric Ground Vehicles with Terrain Profile Preview. IEEE Transactions on Industrial Informatics. 10. 10.1109/TII.2013.2290067.

[6] Takahama, T. & Akasaka, Daisuke. (2018). Model Predictive Control Approach to Design Practical Adaptive Cruise Control for traffic jam. International Journal of Automotive Engineering. 9. 99-104. 10.20485/jsaeijae.9.3_99.

[7] M250. (2019, November 29). Retrieved November 11, 2020, from https://www.pi-innovo.com/product/m250/

[8] RCP System | HIL System | Rapid Prototyping Equipment. Retrieved November 11, 2019, from https://www.opal-rt.com/op4200/

[9] Kvaser Leaf Light HS v2 - Kvaser - Advanced CAN Solutions. (2020, June 05). Retrieved November 11, 2019, from https://www.kvaser.com/product/kvaser-leaf-light-hs-v2/

[10] PiSnoop. (2018, April 30). Retrieved November 11, 2019, from https://www.pi-innovo.com/product/pisnoop/

[11] Software simulation | Real Time applications | RT labs. Retrieved June 11, 2019, from https://www.opal-rt.com/software-rt-lab/

[12] Andersson, Joel & Gillis, Joris & Horn, Greg & Rawlings, James & Diehl, Moritz. (2018). CasADi: A Software Framework for Non-linear Optimization and Optimal Control. Mathematical Programming Computation. 11. 10.1007/s12532-018-0139-4.

[13] Optimization. Retrieved April 25, 2020, from https://www.merriam-webster.com/dictionary/optimization

[14] Gozukucuk, Mehmet & Akdogan, Taylan & Hussain, Waqas & Kargar Tasooji, Tohid & Sahin, Mert & Celik, Mert & Ugurdag, H. Fatih. (2018). Design and Simulation of an Optimal Energy Management Strategy for Plug-In Electric Vehicles. 1-6. 10.1109/CEIT.2018.8751923.

ISO Standards Taken into Account

ISO 11898-2:2016. (2016, December 14). Retrieved November 11, 2020, from https://www.iso.org/standard/67244.html
ISO 26262-1:2018. (2018, December 17). Retrieved November 11, 2020, from https://www.iso.org/standard/68383.html