

# ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

**CENTRUM KOMPETENCE AUTOMOBILOVÉHO PRŮMYSLU JOSEFA BOŽKA**

## VÝSTUP ŘEŠENÍ PRACOVNÍHO BALÍČKU WP18

**TE01020020V049 WP18V004: Algoritmus řízení elektrického nebo hybridního pohonu vozidla**

**Výzkumná zpráva za rok 2014**

**Hlavní řešitel úkolu:**

**ČVUT – fakulta strojní**

Technická 4, Praha 6

Zástupce řešitele: Prof. Ing. Jan Macek, DrSc.

**Řešitel dílčího úkolu**

Doc. Ing. Pavel Mindl, CSc.

Spoluřešitelé dílčího úkolu

Ing. Tomáš Haubert

Ing. Pavel Mňuk, CSc.

Prof. Ing. Zdeněk Čerovský, DrSc.

Bc. Tomáš Hlinovský

Katedra elektrických pohonů a trakce

ČVUT FEL

Technická 2, 166 27 Praha 6

Počet listů:

VZ 415/K13114/2015

Praha, leden 2015

## Úvod

Předložená výzkumná zpráva shrnuje a dokumentuje výsledky řešitelského týmu, podílejícího se na řešení problematiky algoritmu řízení elektrického pohonu automobilu. Řízení elektrického pohonu lze rozdělit do několika rovin:

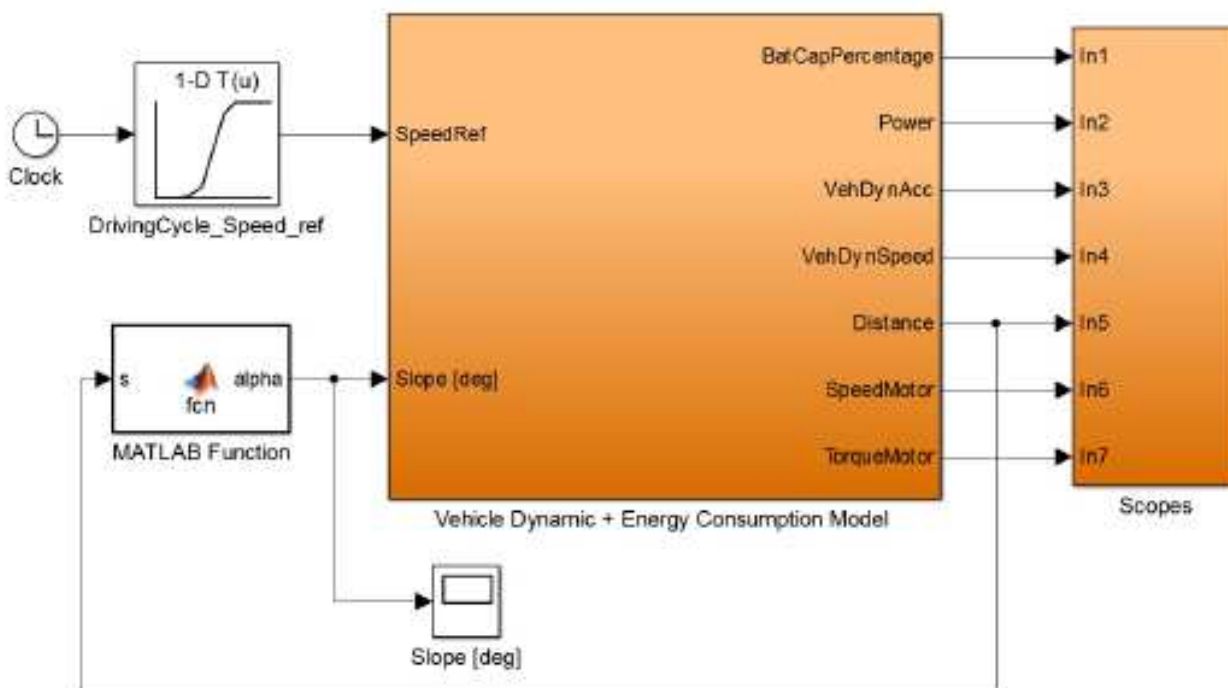
1. **Řízení elektrického stroje**, použitého pro pohon elektromobilu na úrovni elektronický měnič – stroj z hlediska generování požadovaných okamžitých hodnot napětí a proudů pro napájení elektrického stroje. Tento úkol je typicky řešen přímo v elektronickém měniči v závislosti na typu použitého elektrického stroje a jeho vzájemné okamžité poloze rotoru vůči statoru.
2. **Řízení elektrického pohonu** na úrovni nadřazený řídicí systém - elektronický měnič. Tato úroveň řízení je realizována na základě součinnosti spolupráce člověk – stroj (nadřazený řídicí systém), kdy chování pohonu a tím i celého vozidla je plně v kompetenci řidiče. Výsledná účinnost pohonu je silně ovlivněna znalostmi a zkušenostmi řidiče a vnějšími vlivy (přírodní podmínky v době jízdy, profil terénu, kvalita vozovky). U elektrických vozidel nezávislé trakce, kde zásoba energie pro pohon vozidla je velmi omezená, je důležitým faktorem optimalizace její spotřeby. Proto významnou roli hrají další prostředky, umožňující pokud možno bez zásahu lidského faktoru, optimalizovat spotřebu energie na palubě vozidla. A to jak z hlediska potřeb vlastního pohonu, tak z hlediska zajištění bezpečnosti jízdy a přiměřeného pohodlí posádky. K řešení tohoto problému slouží optimalizační algoritmy, umožňující minimalizovat energetické nároky vozidla během jeho provozu.
3. **Optimalizované řízení elektrického pohonu**. Optimalizované řízení elektrického pohonu lze provádět na základě rozsáhlého souboru znalostí a zkušeností o řízeném vozidle jako celku, detailní znalosti vlastností jeho pohonu a znalosti „okrajových podmínek“, jako je aktuální projížděná trasa, její budoucí vertikální a horizontální profil, požadované jízdní rychlosti, stav vozovky a meteorologická situace.

Prezentovaný výsledek řešení optimalizace řízení elektrického pohonu představují dva softwarové balíky pro nadřazený řídicí systém elektrického pohonu, které umožňují řešení problému na úrovni vozidlo – projížděná trasa.

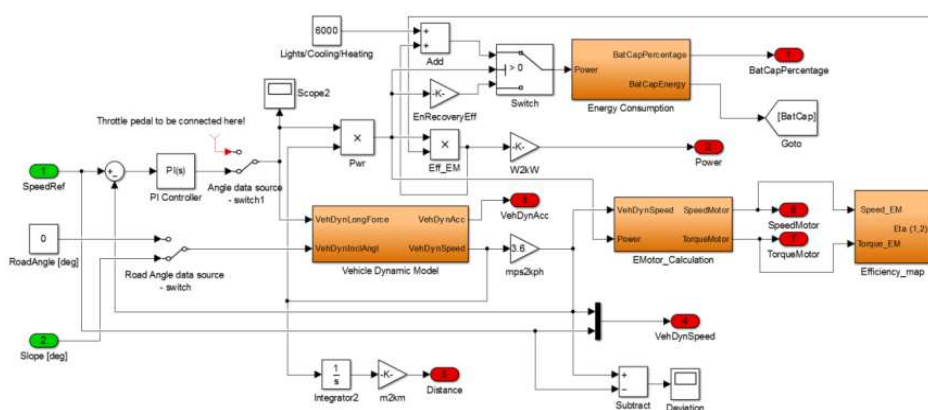
Prvý SW balík představuje matematický model elektrického vozidla, jehož vstupní informací je požadovaná rychlost pohybu, vyjádřená mírou sešlápnutí „plynového“ pedálu a doplňkové informace o aerodynamice, adhezi, sklonových a směrových poměrech projížděné trasy.

Druhý SW balík řeší na základě mapových údajů o vertikálním a horizontálním profilu projížděné trasy optimalizované elementy dráhy, které jsou vstupními údaji pro model vozidla a výpočet požadovaného momentu na výstupu elektrického pohonu. Optimalizační algoritmus řeší jednak minimalizaci počtu linearizovaných úseků dráhy, aby potřebné následné matematické operace byly zvládnutelné v reálném čase a dále koriguje požadavky na moment motoru s ohledem na využitelnou naakumulovanou kinetickou energii vozidla. Pro řešení tohoto problému pracuje SW s predikcí dráhy vozidla v délce cca 2 km.

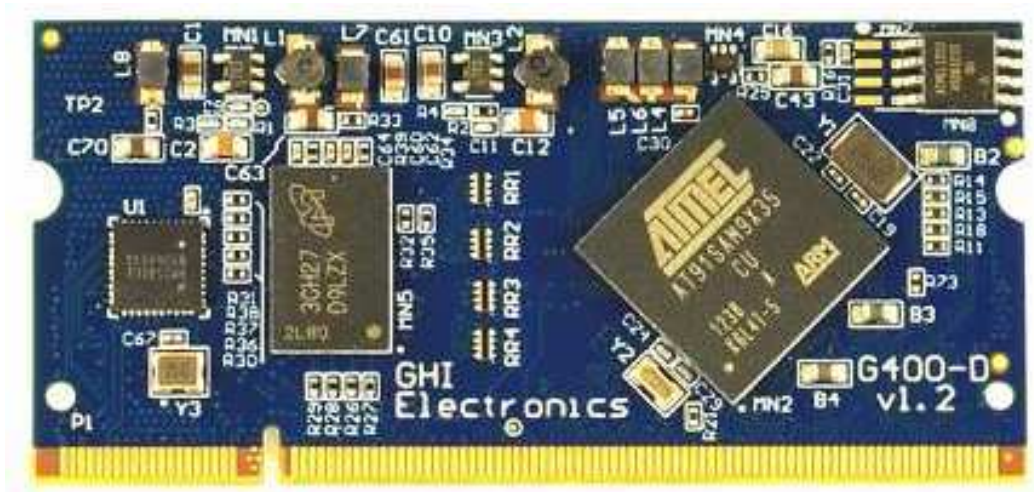
Součástí SW jsou i potřebné vizualizace indikačních přístrojů, obvyklých na palubních deskách vozidel. Softwarové vybavení je implementováno v notebooku a ve speciálním modulu G400 firmy GHI Electronics .



**Obr.1** Globalizované blokové schéma řízení pohonu elektromobilu s respektováním požadovaného jízdního cyklu a horizontálního i vertikálního profilu trasy.



**Obr.2** Subsystém Vehicle Dynamic + Energy Consumption model



Obr.3 C# .NET microFramework embedded modul G400 firmy GHI Electronics s implementovaným algoritmem pro řízení elektrického pohonu.

### Postup použití dynamického modelu EV

- Model je uložen ve složce EV\_DynModel. Název hlavního souboru je: VehDyn\_v14\_NEDC\_Heating\_Angle\_eff.slx
- Tento soubor je nutné spustit v programu Matlab/Simulink od firmy Mathworks (model byl vytvořen ve verzi 2013a a proto je důrazně doporučeno spouštět jej v této nebo novější verzi).
- Po otevření modelu je nutné otevřít 3 inicializační soubory s informacemi o referenční rychlosti (NEDC.m), účinnosti elektrického pohonu (Eff\_M\_n\_data\_table.m) a parametry simulovaného vozidla (VehDynInit\_v4.m). Po otevření je potřeba tyto soubory spustit a nechat nahrát data do workspace.
- Poté je možné nastavit parametry simulace (doba, krok apod.) a simulaci spustit. Referenční rychlost je dána standardním jízdním cyklem, naměřeným jízdním cyklem nebo daty z těchto cyklů upravenými optimalizačním algoritmem.

### Postup použití optimalizačního algoritmu

- Program je nahrán v modulu G400. Po zapnutí napájení program očekává SD kartu s CSV soubory.
- Pomocí tlačítka LDR1. Tlačítkem LDR0 dojde ke spuštění vybraného logu dráhy.
- Program se již dále chová autonomně o po zpracování dat a optimalizaci zobrazí výsledné grafy, které je možné přepínat pomocí tlačítek LDR0 a LDR1.

## Basic dynamic model of EV in Matlab/Simulink

### Basic equations, relations and variables

Model consists of 4 basic equations. The angle of the track (slope) is a function of trajectory, because it matters, in which part of the track the vehicle will be at the moment.

(1)

–

(2)

(3)

(4)

... weight of EV

... acceleration of EV

... air density

... frontal surface

... aerodynamic constant

... speed of EV

... rolling drag force coefficient

... gravity constant

... slope (function of trajectory)

According to general relation for power balance equation:

(5) it is possible to create following power

(6)

## Description of dynamic model subsystems

According to the mentioned equations subsystem Vehicle Dynamic Model has been created. The inputs of the subsystem are forward traction force (VehDynLongForce) and track slope (VehDynInclAngl) and

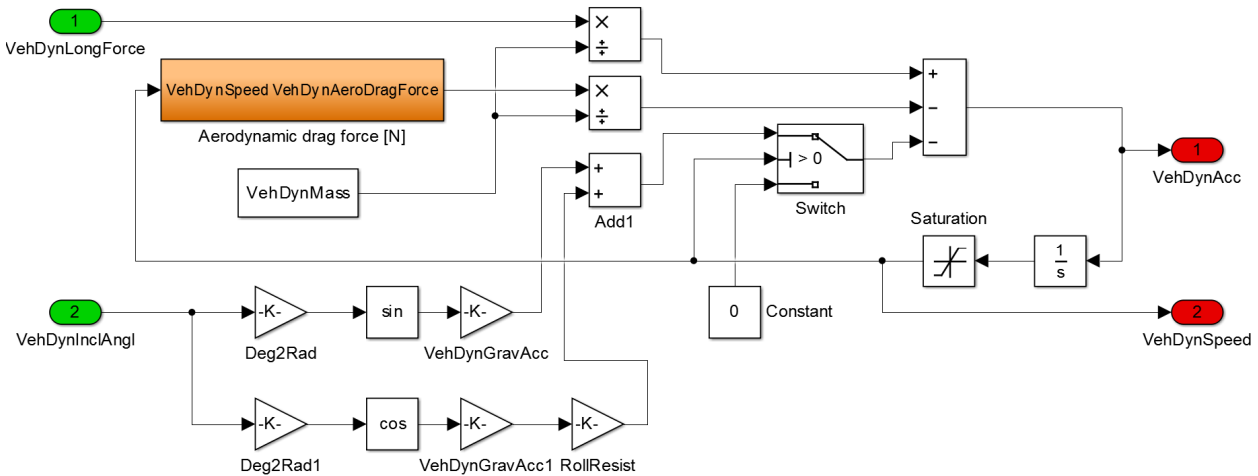


Fig. 4 Subsystem Vehicle Dynamic Model

outputs are the acceleration of the EV (VehDynAcc) and speed (VehDynSpeed), which is calculated as an integration of acceleration. Next subsystem is named Energy Consumption. In this subsystem, we add or subtract the consumed or added (recuperated) energy into the stack of energy which is represented as an accumulator. The outputs are just the current state of charge (in percent value) and a value of remaining energy stored in the battery.

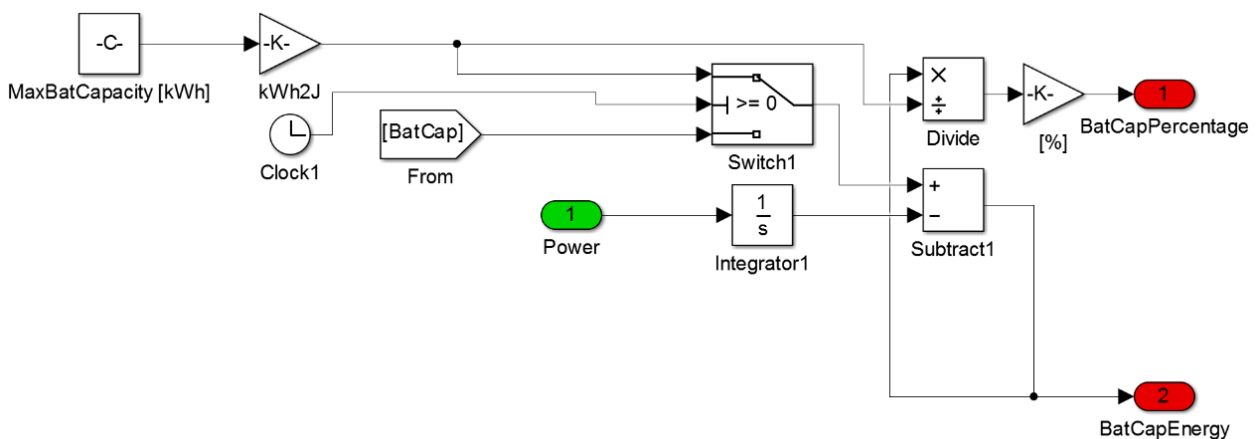


Fig. 5 Subsystem Energy Consumption

Model can also calculate with an efficiency map of the electric motor used in the EV. Because of the absence of model of synchronous motor with permanent magnets, it is necessary to get the torque values from the RPM of the motor. The speed of the EV is known and we have a defined type of tyres with specific

dimensions. Then we can calculate the torque generated by the motor on the shaft. The subsystem EMotor\_Calculation has two inputs, speed of EV and the current power generated by the motor.

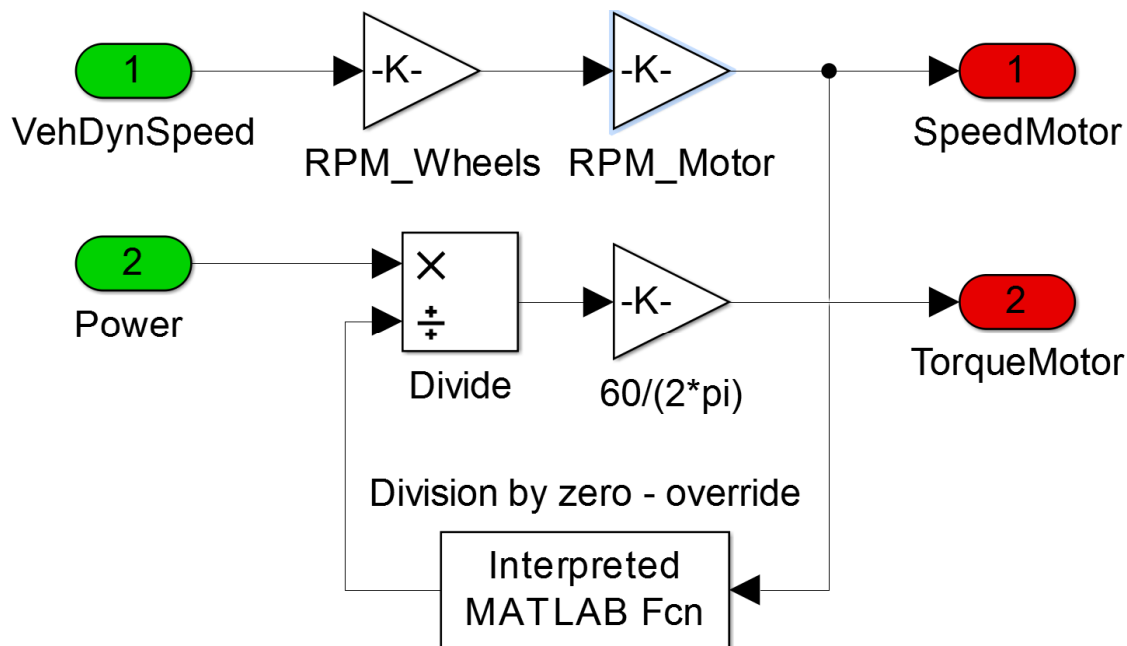


Fig. 6 Subsystem EMotor\_Calculation

Revolutions per minute of electric motor are calculated as:

$$\text{RPM\_Motor} = \text{RPM\_Wheels} \cdot \text{GearRedRatio} \quad (7)$$

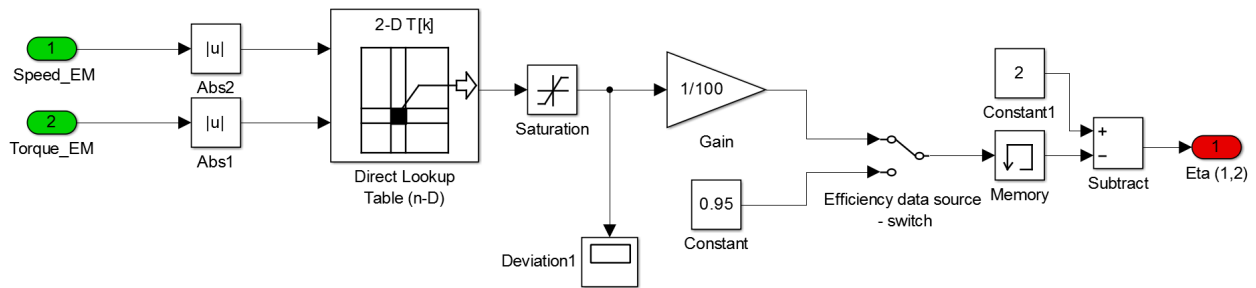
Variables TyrePerimeter and GearRedRatio are explained in the init. m.file (content of this file is displayed in this document later).

Torque generated by the electric motor is calculated as:

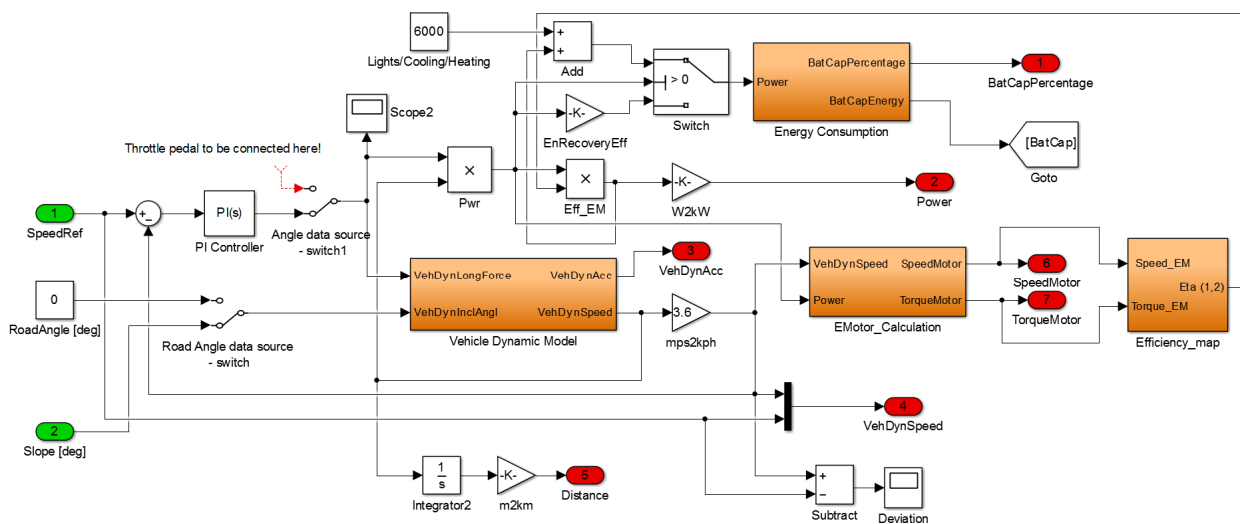
$$\text{TorqueMotor} = \text{Power} \cdot \text{Efficiency} \quad (8)$$

Efficiency map is a graphical representation of a relation  $\eta(M, n)$ . Since  $M$  (torque) and  $n$  (speed of the e. motor) are known, the model can respect the actual position of the operating point in this map. The data with the efficiency values are written in m.file, where it is stored with a certain accuracy and density of measured points. Block Direct Lookup table can perform an interpolation, so it is able to find a value of efficiency for every rpm in the given range. The subtraction (2-eff) has to be there, because the model then multiplies the value of power, needed for the motion of the EV, with this number and the power consumed from battery must be greater than the power generated by the el. motor. So it is necessary to subtract the value of efficiency, which lies in an interval (0, 1), from number 2.





## Completion of the model and initialization files



force is represented as a tuned PI controller for now. The input of the PI controller is reference speed and output is the above mentioned traction force. The other blocks surrounding the subsystems are mainly for unit conversions or a logic of charging/discharging the battery.

For testing of the model, we used basic drive cycles ECE, NEDC and Artemis. ECE cycle represents a drive in a city traffic conditions and NEDC cycle consists of 3 ECE cycles plus one cycle representing drive on a highway, expressway or city bypass. Artemis driving cycles are more complex ones as they are closer to the real driving scenarios. Data of these drive cycles consists of speed in relation on time. This is the reference speed for the PI controller. Furthermore it is needed to run the m.file with efficiency data.



Main file is the initialization file (below):

```
% Init m-file for VehDyn_model (version 5, 21/5/2014)

VehDynGravAcc = 9.81;

% Vehicle parameters
VehDynMass = 1200; %[kg]
VehDynAeroDragCoef = 0.3;
VehDynMassDensityOfAir = 1.3; %[kg/m^3]
VehDynFrontalArea = 2; %[m^2]
MaxBatCapacity = 7; %[kWh]
Cr = 0.01; % Rolling friction coefficient
% Tyre dimensions + Tyre perimeter calculation
TyreWidth = 185; %[mm]
TyreAspectRatio = 0.6; % [%] Height of the tire itself =
TyreWidth*TyreAspectRatio*0.01
TyreDiameter = 15; %[Inches] 1 inch = 2.54 cm
TyrePerimeter = ((TyreWidth*TyreAspectRatio)*2+TyreDiameter*25.4)*pi*0.001; %[m]

%Drivetrain parameters
GearRedRatio = 4.5; %
```

Fixed transmission has been set on value 4.5, because if we get the speed from relation (7), so at 7000 rpm the EV should be going forward with a speed of 175 kph on the defined tyre configuration. Breaking is only represented as recuperation so far, so if the EV is breaking, all the energy is getting stored in battery with efficiency about 25%.

Whole model is put in one last subsystem (figure 7), so one can clearly see the inputs and outputs. One can also notice the preparation for an optimization of the whole drive and inclusion of the track slope profile. However driving cycles ECE, NEDC and Artemis do not respect sloping of the track.

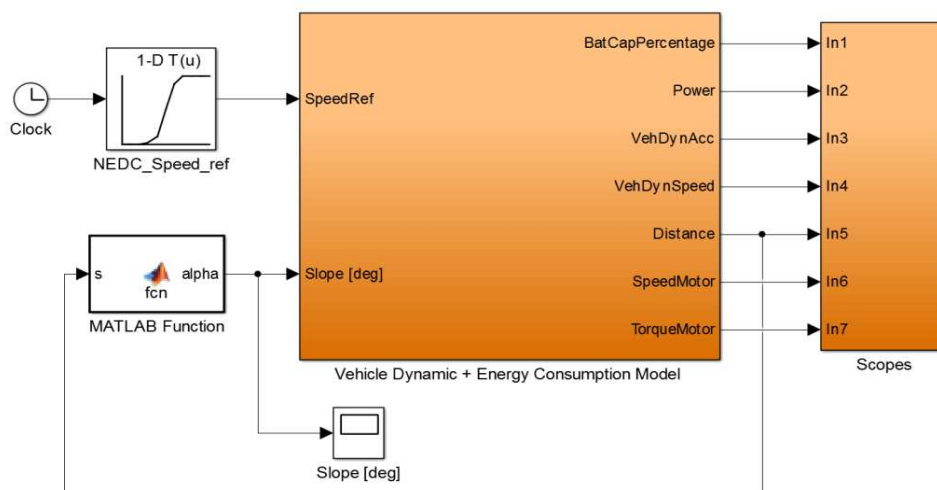


Fig. 9 Whole model with inputs and outputs

## Optimization algorithm description

### Hardware

The optimization algorithm is written in C# language using Visual Studio 2010. G400 module from GHI Electronics Company is used as a target platform. The G400 module is plugged in an interface board with sockets. LCD display and SD card modules are connected into the interface board with sockets. The power supply voltage is 5V and the interface board can be supplied by USB port from computer. Two buttons (LDR0 and LDR1) on interface board are used for GUI control.

### Software

The program architecture is Gadgeteer-based. The main function is `ProgramStarted()` in `Program` class. This function enables the main thread using `Start()` method. The first initialized class is `DisplayGUI` and then the program waits for SD card and its mount.

The user can choose only one function using push the button. The functionality of application is in `ProcessLogFile()` method which initializes `Processing` class. The SD card is unmounted after closing the application.

### Processing class

The methods for analyzing and processing data are in the `Processing` class. The user has to choose a CSV file with logged data. It is possible to use the LDR1 button for listing in the file structure and the optimization algorithm starts after push of LDR0 button.

The algorithm steps are:

#### 1. Read and parse CSV log using method `LoadLogFile()`

- This method returns `ArrayList` of `SensorsData` objects. These objects have Time and GPS data

Time ms	GPS Course	UTCTime	Latitude	Longitude	Alt	SpeedKMH
		2014-03-21				
40000	82.9	12:20:10.000	50.02719243	14.51178072	372.0	4.5
		2014-03-21				
41000	87.8	12:20:12.000	50.0271885	14.51182678	369.0	4.0
		2014-03-21				
42000	98.6	12:20:14.000	50.02718088	14.51186651	368.0	3.5
		2014-03-21				
43000	114.0	12:20:15.000	50.02716508	14.51189832	368.0	3.0
		2014-03-21				
44000	135.8	12:20:16.000	50.02714449	14.5119129	368.0	2.25
		2014-03-21				
45000	152.0	12:20:17.000	50.02712109	14.51192183	369.0	2.25

2. **Calculate a moving average** from altitude using `AnalysisIdentGPS.MovingAverage(ArrayList)`
  - This method is helpful when the data inaccurate
  - The filter parameter is 40 in default settings
3. **Data processing using RDP algorithm** using `AnalysisIdentGPS.RDPSimplification(ArrayList)` method
  - Recursive algorithm which finds the element where the position or altitude is changed
  - It is possible to change the tolerance. The default settings for position is 0.00005 (it is 55m in real) and for altitude is 3m
4. **The track segments** are calculated from the significant points using `AnalysisIdentGPS.SegmentIdentification(ArrayList)` method
5. The result is `ArrayList` of objects `LineSegment`

Segments:

			Distance			Elevation	
Trajectory	Elevation	Length	traveled	Speed	Angle	rate	Min speed
STRAIGHT	DOWN	0.277	0.00000	15	0	-3.8	50
CURVE	NONE	0.000	0.27657	3	157	0.0	40
STRAIGHT	UP	0.006	0.27657	0	0	2.8	50
CURVE	NONE	0.000	0.28276	0	165	0.0	40
STRAIGHT	DOWN	0.031	0.28276	5	0	-7.7	50
CURVE	NONE	0.000	0.31347	2	55	0.0	40
STRAIGHT	DOWN	0.022	0.31347	5	0	-3.2	50
CURVE	NONE	0.000	0.33596	6	5	0.0	40
STRAIGHT	UP	0.371	0.33596	41	0	0.8	50
CURVE	NONE	0.000	0.70731	5	10	0.0	40

6. **Optimization** is calculated using `Optimize.Run()` method in `Optimization` class
  - The length of vector `Constants.VECTOR_LENGTH` represents the number of second for one drive period
  - Two algorithms are implemented:
    - i. **Bobyqa** je mathematical algorithm which makes nonlinear optimization. The time for calculation is high because this algorithm scans the all state space.
    - ii. **Minimize** is mathematical algorithm designed for this problem. The time for calculation is about 100x less than **Bobyqa** but the result is not 100% optimal. This algorithm uses a knowledge of the optimal reactions on the future event. It is very important to set these reactions correctly in accordance with mathematical model of electric vehicle. This algorithm works as follows:
      1. The first step is to “drive” the track with initial values
      2. The second step is to set “driver’s pedal position” (0 – 1) in accordance with Min speed in `LineSegment`. The speed tolerance is in `Constants.SIGN_DIFFERENCE`.
      3. The third step is the “drive” the track and predicts the future segment. The algorithm will decide what is necessary to do after prediction. For example before hill the algorithm will increase the “driver’s pedal position”. There are three constants for algorithm settings:

- a. Constants.TIME\_OF\_PREDICTION: This constant represents the number of seconds for reaction for the future segment
  - b. Constants.INCREASE\_SPEED\_EFFECT: This constant represents the increase of “driver’s pedal position” before segment where the track goes up.
  - c. Constants.DECREASE\_SPEED\_EFFECT: This constant represents the increase of “driver’s pedal position” before segment where the track goes down.
4. The last step is to apply moving average filter on data. The result is these vectors:
- a. Driver’s pedal position
  - b. Vehicle speed
  - c. Consumption
  - d. Travelled distance

Ride:

i	Accelerator	Speed	Consumption
0		1	0
1	0.20999999999999996	23	400
2	0.084999999999999978	29	417
3	0.0037499999999999617	32	419
4	0.0037499999999999617	33	419

**Results:**

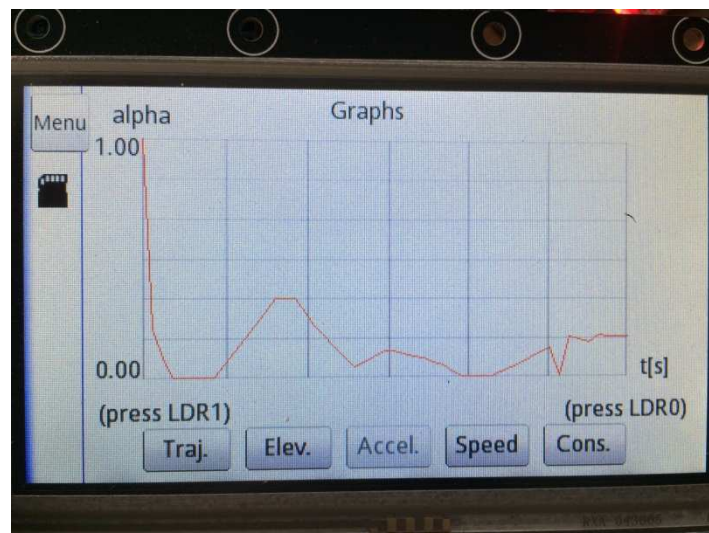


Fig. 10 Driver’s pedal position = function (time)

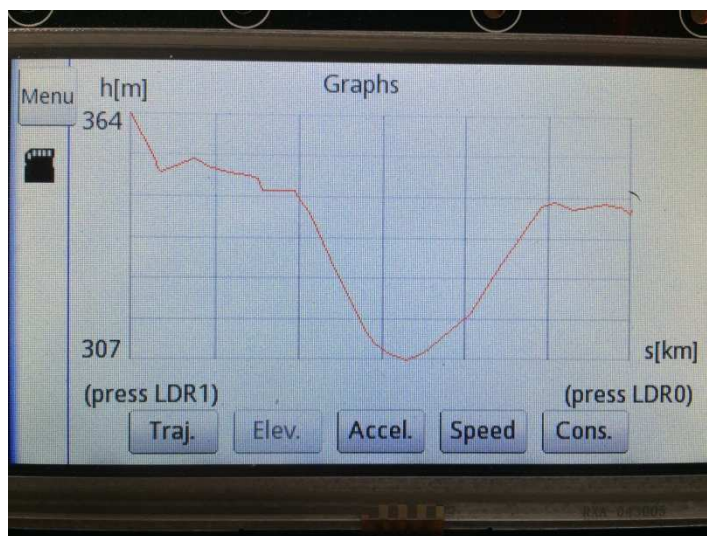


Fig. 11 Altitude = function (travelled distance)

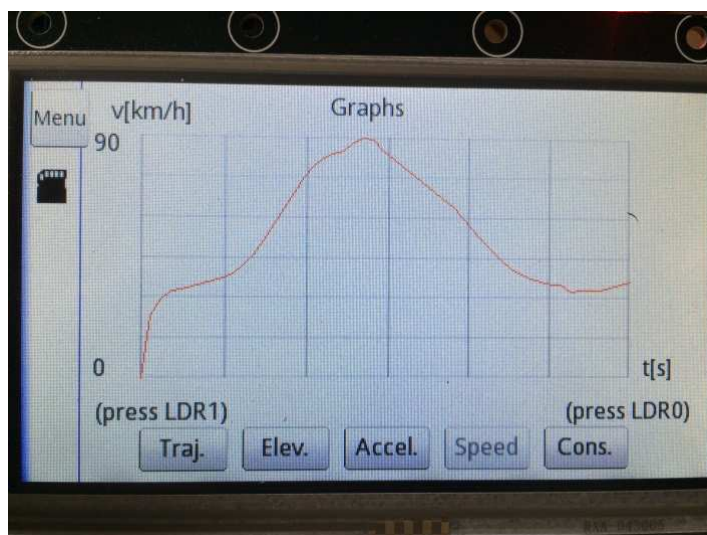


Fig. 12 Vehicle speed = function (time)

## Závěr

Výstupem této výzkumné zprávy jsou dva softwarové balíky simulující pohyb elektrického vozidla po trati včetně aktuálních stavů důležitých proměnných a optimalizační algoritmus minimalizující spotřebu elektrické energie. První algoritmus je realizován v prostředí MATLAB/Simulink a druhý prostřednictvím programovacího jazyka C# v prostředí Visual Studio 2010 na platformě G400. Tato platforma byla zvolena pro svůj vysoký výpočetní výkon a zároveň malou velikost s možností online optimalizace v reálném vozidle. Největším problémem optimalizačního algoritmu je citlivost na konstanty modelu a konstanty nastavení optimalizace. Při správném nastavení konstant (hlavně aktuální hmotnosti vozidla) je algoritmus schopen snížit spotřebu o 5%. Míra snížení spotřeby závisí na tom, jaký styl jízdy použil řidič vozidla pro referenční jízdu. Za předpokladu, že řidič velmi dobře znal trasu, hustotu provozu a další podmínky, byla spotřeba optimalizačního algoritmu nižší o cca 2% a naopak za nejhorších podmínek pro řidiče byla spotřeba optimalizačního algoritmu nižší o 10 a více procent.



## Příloha

Částečný výpis kódu optimalizačního algoritmu (Třída **AnalysisIdentGPS**)

```
using System;
using System.Collections;

namespace TrackOptimizer.Analysis
{
    /// <summary>
    /// Static class defines methods to analyze list of GPS coordinates
    /// </summary>
    class AnalysisIdentGPS
    {
        private static double distanceFromStart = 0.0;

        /// <summary>
        /// Method applies Ramer-Douglas-Peucker algorithm to given list of GPS points.
        /// </summary>
        /// <param name="points">List of GPS points</param>
        /// <returns>List of significant GPS points</returns>
        public static ArrayList RDPsimplification(ArrayList points)
        {
            // No need to analyze less than 3 points
            if (points == null || points.Count < 3) return null;

            // First and last point are kept (here pointIndexesToKeepCoordinates is list of
            point indexes only)
            ArrayList pointIndexesToKeep = new ArrayList();
            pointIndexesToKeep.Add(0);
            pointIndexesToKeep.Add(points.Count - 1);

            // Apply algorithms, firstly trajectory, then elevation
            RDPTrajectory(points, 0, points.Count - 1, pointIndexesToKeep);
            RDPElevation(points, 0, points.Count - 1, pointIndexesToKeep);

            // Sort list
            Sort(pointIndexesToKeep);

            // Build result list from list of indexes
            ArrayList returnPoints = new ArrayList();
            foreach (int index in pointIndexesToKeep)
            {
                returnPoints.Add(points[index]);
            }
            return returnPoints;
        }

        /// <summary>
        /// Implementation of Ramer-Douglas-Peucker algorithm applied considering trajectory
        /// </summary>
        /// <param name="points">List of GPS points</param>
        /// <param name="firstPoint">Index to first point</param>
        /// <param name="lastPoint">Index to last point</param>
        /// <param name="pointIndexesToKeep">List of point indexes to keep</param>
        private static void RDPTrajectory(ArrayList points, int firstPoint, int lastPoint,
        ArrayList pointIndexesToKeep)
        {
            double maxDistance = 0;
```



```

    int indexFarthest = 0;

    // Find maximum distance from line formed by first and last point
    for (int index = firstPoint; index < lastPoint; index++)
    {
        double distance = CoordinatesDistance((SensorsData)points[firstPoint],
        (SensorsData)points[lastPoint], (SensorsData)points[index]);
        if (distance > maxDistance)
        {
            maxDistance = distance;
            indexFarthest = index;
        }
    }

    // Point with maximum distance must be kept and RDP is called recursively to both
    sides other than the point
    if (maxDistance > Constants.RDP_TOLERANCE_COORDINATES && indexFarthest != 0)
    {
        pointIndexsToKeep.Add(indexFarthest);
        RDPTrajectory(points, firstPoint, indexFarthest, pointIndexsToKeep);
        RDPTrajectory(points, indexFarthest, lastPoint, pointIndexsToKeep);
    }
}

/// <summary>
/// Implementation of Ramer-Douglas-Peucker algorithm applied considering elevation
/// </summary>
/// <param name="points">List of GPS points</param>
/// <param name="firstPoint">Index to first point</param>
/// <param name="lastPoint">Index to last point</param>
/// <param name="pointIndexsToKeep">List of point indexes to keep</param>
private static void RDPElevation(ArrayList points, int firstPoint, int lastPoint,
ArrayList pointIndexsToKeep)
{
    double maxDistance = 0;
    int indexFarthest = 0;

    // Find maximum distance from line formed by first and last point
    for (int index = firstPoint; index < lastPoint; index++)
    {
        double distance = ElevationDistance((SensorsData)points[firstPoint],
        (SensorsData)points[lastPoint], (SensorsData)points[index]);
        if (distance > maxDistance)
        {
            maxDistance = distance;
            indexFarthest = index;
        }
    }

    // Point with maximum distance must be kept and RDP is called recursively on both
    sides other than the point
    if (maxDistance > Constants.RDP_TOLERANCE_ELEVATION && indexFarthest != 0)
    {
        pointIndexsToKeep.Add(indexFarthest);
        RDPElevation(points, firstPoint, indexFarthest, pointIndexsToKeep);
        RDPElevation(points, indexFarthest, lastPoint, pointIndexsToKeep);
    }
}

/// <summary>

```

```

    /// Coutns perpendicular distance between line formed by first two parameters and the
    third one
    /// </summary>
    /// <param name="A">First point of line</param>
    /// <param name="B">Last point of line</param>
    /// <param name="C">Examined point</param>
    /// <returns>Distance</returns>
    private static double CoordinatesDistance(SensorsData A, SensorsData B, SensorsData
C)
    {
        /// Perpendicular distance of point C and line AB, x = longitude, y = latitude
        double u =
            ((B.gps.Longitude - A.gps.Longitude) * (C.gps.Longitude - A.gps.Longitude) +
            (B.gps.Latitude - A.gps.Latitude) * (C.gps.Latitude - A.gps.Latitude))
            /
            (System.Math.Pow((B.gps.Longitude - A.gps.Longitude), 2) +
            System.Math.Pow((B.gps.Latitude - A.gps.Latitude), 2));
        if (u <= 0)
        {
            return System.Math.Sqrt(System.Math.Pow(System.Math.Abs(C.gps.Longitude -
A.gps.Longitude), 2) + System.Math.Pow(System.Math.Abs(C.gps.Latitude - A.gps.Latitude), 2));
        }
        else if (u >= 1)
        {
            return System.Math.Sqrt(System.Math.Pow(System.Math.Abs(C.gps.Longitude -
B.gps.Longitude), 2) + System.Math.Pow(System.Math.Abs(C.gps.Latitude - B.gps.Latitude), 2));
        }
        else
        {
            return System.Math.Sqrt(
                System.Math.Pow(
                    System.Math.Abs(
                        C.gps.Longitude - A.gps.Longitude -
                        ((B.gps.Longitude - A.gps.Longitude) * ((B.gps.Longitude -
A.gps.Longitude) * (C.gps.Longitude - A.gps.Longitude) + (B.gps.Latitude - A.gps.Latitude) *
(C.gps.Latitude - A.gps.Latitude)))
                        /
                        ((System.Math.Pow(B.gps.Longitude - A.gps.Longitude, 2) +
System.Math.Pow(B.gps.Latitude - A.gps.Latitude, 2))
                    ), 2)
                +
                System.Math.Pow(
                    System.Math.Abs(
                        C.gps.Latitude - A.gps.Latitude -
                        ((B.gps.Latitude - A.gps.Latitude) * ((B.gps.Longitude -
A.gps.Longitude) * (C.gps.Longitude - A.gps.Longitude) + (B.gps.Latitude - A.gps.Latitude) *
(C.gps.Latitude - A.gps.Latitude)))
                        /
                        ((System.Math.Pow(B.gps.Longitude - A.gps.Longitude, 2) +
System.Math.Pow(B.gps.Latitude - A.gps.Latitude, 2))
                    ), 2)
                );
        }
    }

    /// <summary>
    /// Coutns perpendicular distance between line formed by first two parameters and the
    third one, TODO test
    /// </summary>
    /// <param name="A">First point of line</param>

```

```

/// <param name="B">Last point of line</param>
/// <param name="C">Examined point</param>
/// <returns>Distance</returns>
private static double ElevationDistance(SensorsData A, SensorsData B, SensorsData C)
{
    // Perpendicular distance of point C and line AB, x = time, y = altitude
    double u =
        ((B.time / 1000 - A.time / 1000) * (C.time / 1000 - A.time / 1000) +
        (B.gps.Altitude - A.gps.Altitude) * (C.gps.Altitude - A.gps.Altitude))
        /
        (System.Math.Pow((B.time / 1000 - A.time / 1000), 2) +
        System.Math.Pow((B.gps.Altitude - A.gps.Altitude), 2));
    if (u <= 0)
    {
        return System.Math.Sqrt(System.Math.Pow(System.Math.Abs(C.time / 1000 -
        A.time / 1000), 2) + System.Math.Pow(System.Math.Abs(C.gps.Altitude - A.gps.Altitude), 2));
    }
    else if (u >= 1)
    {
        return System.Math.Sqrt(System.Math.Pow(System.Math.Abs(C.time / 1000 -
        B.time / 1000), 2) + System.Math.Pow(System.Math.Abs(C.gps.Altitude - B.gps.Altitude), 2));
    }
    else
    {
        return System.Math.Sqrt(
            System.Math.Pow(
                System.Math.Abs(
                    C.time / 1000 - A.time / 1000 -
                    ((B.time / 1000 - A.time / 1000) * ((B.time / 1000 - A.time /
                    1000) * (C.time / 1000 - A.time / 1000) + (B.gps.Altitude - A.gps.Altitude) * (C.gps.Altitude
                    - A.gps.Altitude)))
                    /
                    ((System.Math.Pow(B.time / 1000 - A.time / 1000, 2) +
                    System.Math.Pow(B.gps.Altitude - A.gps.Altitude, 2))
                    ), 2)
                +
                System.Math.Pow(
                    System.Math.Abs(
                        C.gps.Altitude - A.gps.Altitude -
                        ((B.gps.Altitude - A.gps.Altitude) * ((B.time / 1000 - A.time
                        / 1000) * (C.time / 1000 - A.time / 1000) + (B.gps.Altitude - A.gps.Altitude) *
                        (C.gps.Altitude - A.gps.Altitude)))
                        /
                        ((System.Math.Pow(B.time / 1000 - A.time / 1000, 2) +
                        System.Math.Pow(B.gps.Altitude - A.gps.Altitude, 2))
                        ), 2)
                    ), 2)
            );
    }
}

/// <summary>
/// Method filters list of GPS elevation points with simple moving average. Parameter
is in Constants class.
/// </summary>
/// <param name="points">List of GPS points</param>
public static void MovingAverage(ArrayList points)
{
    // Number of points must be larger than the parameter
    if (points.Count < Constants.MOVING_AVERAGE_PARAM) return;

```

```

for (int idx = 0; idx < (points.Count - Constants.MOVING_AVERAGE_PARAM); idx++)
{
    double accumulator = 0.0;
    for (int i = 0; i < Constants.MOVING_AVERAGE_PARAM; i++)
    {
        accumulator += ((SensorsData)points[idx + i]).gps.Altitude;
    }
    accumulator /= Constants.MOVING_AVERAGE_PARAM;
    ((SensorsData)points[idx]).gps.Altitude = accumulator;
}
// Reverse last x points
for (int idx = points.Count - 1; idx > (points.Count -
Constants.MOVING_AVERAGE_PARAM - 1); idx--)
{
    double accumulator = 0.0;
    for (int i = 0; i < Constants.MOVING_AVERAGE_PARAM; i++)
    {
        accumulator += ((SensorsData)points[idx - i]).gps.Altitude;
    }
    accumulator /= Constants.MOVING_AVERAGE_PARAM;
    ((SensorsData)points[idx]).gps.Altitude = accumulator;
}

/// <summary>
/// Since result list might be disordered, we need to sort the indexes again. Simple
bubblesort used.
/// </summary>
/// <param name="list">List of point indexes</param>
private static void Sort(ArrayList list)
{
    object temp;
    int length = list.Count;
    bool change = true;

    while (change)
    {
        change = false;
        for (int sort = 0; sort < length - 1; sort++)
        {
            if ((int)list[sort] > (int)list[sort + 1])
            {
                temp = list[sort + 1];
                list[sort + 1] = list[sort];
                list[sort] = temp;
                change = true;
            }
            else if ((int)list[sort] == (int)list[sort + 1])
            {
                list.RemoveAt(sort);
                length--;
                sort--;
            }
        }
    }
}

/// <summary>
/// Counts distance in KM between two GPS points.

```

```

/// </summary>
/// <param name="previous">First GPS point</param>
/// <param name="current">Second GPS point</param>
/// <returns>Distance in KM</returns>
public static double GPSCoordsDistanceKM(SensorsData previous, SensorsData current)
{
    double dlong = (current.gps.Longitude - previous.gps.Longitude) *
Constants.DEGREES_TO_RAD;
    double dlat = (current.gps.Latitude - previous.gps.Latitude) *
Constants.DEGREES_TO_RAD;
    double a = System.Math.Pow(System.Math.Sin(dlat / 2.0), 2) +
System.Math.Cos(previous.gps.Latitude * Constants.DEGREES_TO_RAD) *
System.Math.Cos(current.gps.Latitude * Constants.DEGREES_TO_RAD) *
System.Math.Pow(System.Math.Sin(dlong / 2.0), 2);
    double c = 2 * System.Math.Atan2(System.Math.Sqrt(a), System.Math.Sqrt(1 - a));
    return 6367 * c;
}

/// <summary>
/// Counts angle clutching vector defined by points A, B and vector defined by points
B, C
/// </summary>
/// <param name="previous">GPS point A</param>
/// <param name="current">GPS point B</param>
/// <param name="next">GPS point C</param>
/// <returns>Angle in degrees</returns>
private static int GPSCoordsAngleDegrees(SensorsData previous, SensorsData current,
SensorsData next)
{
    double[] a = { current.gps.Longitude - previous.gps.Longitude,
current.gps.Latitude - previous.gps.Latitude };
    double[] b = { current.gps.Longitude - next.gps.Longitude, current.gps.Latitude -
next.gps.Latitude };
    double theta = System.Math.Acos((a[0] * b[0] + a[1] * b[1])
/
System.Math.Sqrt((a[0] * a[0] + a[1] * a[1]) * (b[0] * b[0] + b[1] * b[1])));
    return (int)(180 - (theta * Constants.RAD_TO_DEGREES));
}

/// <summary>
/// Method identifies each segment and counts its parameters.
/// </summary>
/// <param name="coordinatesPoints">Simplified list of GPS points</param>
/// <returns>List of LineSegment objects</returns>
public static ArrayList SegmentIdentification(ArrayList coordinatesPoints)
{
    if (coordinatesPoints == null) return null;

    ArrayList records = new ArrayList();
    for (int i = 1; i < coordinatesPoints.Count; i++)
    {
        SensorsData previous = (SensorsData)coordinatesPoints[i - 1];
        SensorsData current = (SensorsData)coordinatesPoints[i];

        // previous segment, always straight with possible altitude changes
        LineSegment previousSegment = new LineSegment();
        previousSegment.CurrentTrajectory = LineSegment.Trajectory.Straight;
        previousSegment.DistanceTraveled = distanceFromStart;
        previousSegment.Length = GPSCoordsDistanceKM(previous, current);
        TimeSpan timeDiff = current.gps.TimeUTC.Subtract(previous.gps.TimeUTC);
    }
}

```

```

        previousSegment.Velocity = (int)((previousSegment.Length * 1000) /
(timeDiff.Seconds + (timeDiff.Minutes * 60) + (timeDiff.Hours * 3600))) * 3.6);
        double elevationPercent = ((current.gps.Altitude - previous.gps.Altitude) /
(previousSegment.Length * 1000)) * 100;
        if (elevationPercent < -Constants.ELEVATION_PERCENTAGE_TOLERANCE)
            previousSegment.CurrentElevation = LineSegment.Elevation.Down;
        else if (elevationPercent > Constants.ELEVATION_PERCENTAGE_TOLERANCE)
            previousSegment.CurrentElevation = LineSegment.Elevation.Up;
        else
            elevationPercent = 0;
        previousSegment.ElevationRate = elevationPercent;
        previousSegment.MinSpeed = 50;

        distanceFromStart += previousSegment.Length;
        records.Add(previousSegment);

        // current segment, always curve
        if (i < coordinatesPoints.Count - 1)
        {
            LineSegment currentSegment = new LineSegment();
            currentSegment.CurrentTrajectory = LineSegment.Trajectory.Curve;
            currentSegment.Velocity = (int)(current.gps.SpeedKMH);
            currentSegment.DistanceTraveled = distanceFromStart;
            SensorsData next = (SensorsData)coordinatesPoints[i + 1];
            currentSegment.TrajectoryAngle = GPSCoordsAngleDegrees(previous, current,
next);

            currentSegment.MinSpeed = 40;
            records.Add(currentSegment);
        }
    }
    return records;
}
}
}
}

```

Částečný výpis kódu optimalizačního algoritmu (**Třída Custom**) = Navržený optimalizační algoritmus

```
using System;
using System.Collections;

namespace TrackOptimizer.Analysis
{
    /// <summary>
    /// Static class represents own version of vehicle consumption optimization function. It
    performs very well on wise
    /// choice of parameters, which may (and should) be set with knowledge of tests results.
    Algorithm is about 100 times
    /// faster than BOBYQA, is way more configurable and performs almost equally. Feel free
    to set all the
    /// constants to fit your testing and estimates, but always do it carefully, otherwise
    you spend a lot of time to
    /// set it back and working right.
    /// </summary>
    static class Custom
    {
        private static double[] _accelerator;
        private static int[] _speed;
        private static int[] _consumption;

        private static int[] _optimSpeed;
        private static double[] _betas;

        private static ArrayList _data;

        /// <summary>
        /// The only public function prepares data space and does all the black box magic.
        /// </summary>
        /// <param name="caller"></param>
        /// <param name="data"></param>
        public static void FindMin(Optimization caller, ArrayList data)
        {
            _data = data;

            _accelerator = caller.Accelerator;
            _speed = caller.Speed;
            _consumption = caller.Consumption;
            _optimSpeed = new int[Constants.VECTOR_LENGTH];
            _betas = new double[Constants.VECTOR_LENGTH];

            Minimization();

            caller.Accelerator = _accelerator;
            caller.Speed = _speed;
            caller.Consumption = _consumption;
        }

        /// <summary>
        /// Do the minimization.
        /// </summary>
        private static void Minimization()
        {
            // first iteration, ride the entire trip with reference data
            Acc();

            // gain appropriate speed

```



```

GainRequiredSpeed();

// act on changes of elevation rates
PredictChanges();

// apply moving average on accelerator for comfortable ride
MovingAverageComfort();

int resultVectorLength = Acc();

// copy arrays of results
double[] resAcc = new double[resultVectorLength];
Array.Copy(_accelerator, resAcc, resultVectorLength);
_accelerator = resAcc;
int[] resSpeed = new int[resultVectorLength];
Array.Copy(_speed, resSpeed, resultVectorLength);
_speed = resSpeed;
int[] resCons = new int[resultVectorLength];
Array.Copy(_consumption, resCons, resultVectorLength);
_consumption = resCons;
}

/// <summary>
/// Sets accelerator to hold required speed at each segment. No need to edit
anything, it just makes you
/// follow the optimal speed at first point.
/// </summary>
private static void GainRequiredSpeed()
{
    for (int iter = 0; iter < 80; iter++)
    {
        for (int i = 0; i < Constants.VECTOR_LENGTH; i++)
        {
            int diff = Difference(_speed[i], _optimSpeed[i]);
            switch (diff)
            {
                case -1:
                    _accelerator[i] += 0.01;
                    break;
                case 1:
                    _accelerator[i] -= 0.01;
                    break;
            }
            if (_accelerator[i] > 1) _accelerator[i] = 1;
            else if (_accelerator[i] < 0) _accelerator[i] = 0;
        }
        Acc();
    }
}

/// <summary>
/// Do the action based on elevation profile changes prediction.
/// </summary>
private static void PredictChanges()
{
    ArrayList changes = new ArrayList();
    for (int i = 0; i < Constants.VECTOR_LENGTH - 1; i++)
    {
        if (_betas[i] != _betas[i + 1])
        {

```

```

        changes.Add(i);
    }
}

// preparing the accelerator for changes
double previousDiff = 0;
for (int i = 0; i < changes.Count; i++)
{
    int idx = (int)changes[i];
    double val1 = _betas[idx];
    double val2 = _betas[idx + 1];
    double diff = System.Math.Abs(val1 - val2);
    // this is just when data are extremely unlucky and new change occurs (means
    extremely short track segment)
    if (diff == previousDiff) continue;
    previousDiff = diff;

    // less climb, climb to descent or greater descent - decrease speed
    if (val1 > val2)
    {
        for (int j = idx; j >= (((idx - Constants.TIME_OF_PREDICTION) > 0) ? (idx
        - Constants.TIME_OF_PREDICTION) : 0); j--)
        {
            _accelerator[j] -= Constants.DECREASE_SPEED_EFFECT *
            Constants.TIME_OF_PREDICTION * diff;
            if (_accelerator[j] < 0) _accelerator[j] = 0;
        }
        // less descent, descent to climb or greater climb - increase speed
        else
        {
            for (int j = idx; j >= (((idx - Constants.TIME_OF_PREDICTION) > 0) ? (idx
            - Constants.TIME_OF_PREDICTION) : 0); j--)
            {
                _accelerator[j] += Constants.INCREASE_SPEED_EFFECT *
                Constants.TIME_OF_PREDICTION * diff;
                if (_accelerator[j] > 1) _accelerator[j] = 1;
            }
        }
        Acc();
        // now the changes are on wrong indexes, so get them right
        changes.Clear();
        for (int x = 0; x < Constants.VECTOR_LENGTH - 1; x++)
        {
            if (_betas[x] != _betas[x + 1])
            {
                changes.Add(x);
            }
        }
    }
}

/// <summary>
/// Applies moving average smoother on accelerator for comfortable and nice smooth
ride as well as low consumption.
/// </summary>
private static void MovingAverageComfort()
{
    if (Constants.VECTOR_LENGTH > Constants.MOV_AVG_COMFORT)

```

```

    {
        for (int idx = 1; idx < (Constants.VECTOR_LENGTH -
Constants.MOV_AVG_COMFORT); idx++)
        {
            double accumulator = 0.0;
            for (int i = 0; i < Constants.MOV_AVG_COMFORT; i++)
            {
                accumulator += _accelerator[idx + i];
            }
            accumulator /= Constants.MOV_AVG_COMFORT;
            _accelerator[idx] = accumulator;
        }
    }
}

/// <summary>
/// Counts significance of difference between two integer numbers.
/// </summary>
/// <param name="a">First number</param>
/// <param name="b">Second number</param>
/// <returns>-1 if a is less than b, 0 if similar, 1 if a is greater than b</returns>
private static int Difference(int a, int b)
{
    if (a + Constants.SIGN_DIFFERENCE <= b) return -1;
    else if (a - Constants.SIGN_DIFFERENCE >= b) return 1;
    else return 0;
}

/// <summary>
/// Objective function Acc.
/// </summary>
/// <returns>VECTOR_LENGTH or shorter vector size when set too high</returns>
private static int Acc()
{
    int vk = Constants.v0;
    int J = 0;
    double s = 0;

    // accelerator every second
    for (int i = 0; i < Constants.VECTOR_LENGTH; i++)
    {
        _speed[i] = vk;
        _consumption[i] = J;

        double beta = 0; // beta is slope of the road, negative=down
        int vref = 0;
        s += vk / 3600.0; // s is covered distance in km/h

        if (s >= ((LineSegment)_data[_data.Count - 1]).DistanceTraveled +
((LineSegment)_data[_data.Count - 1]).Length)
        {
            return i;
        }

        // gain slope and vref
        for (int j = 0; j < _data.Count; j += 2)
        {
            LineSegment now = (LineSegment)_data[j];
            if (s >= now.DistanceTraveled - Constants.CURVE_LENGTH && s <=
now.DistanceTraveled && j > 0)

```

```
{
    vref = ((LineSegment)_data[j - 1]).MinSpeed;
}

else if (s >= now.DistanceTraveled && s <= now.DistanceTraveled +
Constants.CURVE_LENGTH && j < _data.Count - 1)
{
    vref = ((LineSegment)_data[j + 1]).MinSpeed;
}

if (s >= now.DistanceTraveled && s <= now.DistanceTraveled + now.Length)
{
    beta = now.ElevationRate;
    if (vref == 0)
    {
        vref = now.MinSpeed;
    }
    _optimSpeed[i] = vref;
    _betas[i] = beta;
    break;
}
}

// current speed
int vkk = (int)(Constants.k1 * vk - Constants.k2 * vk * vk *
System.Math.Sign(vk) - Constants.k3 * beta + Constants.k4 * _accelerator[i]);

// consumption and speed
J += (int)(Constants.a2 * _accelerator[i] * _accelerator[i]);
vk = vkk;
}
return Constants.VECTOR_LENGTH;
}
}
```