Using Chip Simulation to optimize Engine Control

Matthias Simons, Mihai Feier, Jakob Mauss

Abstract

Calibration of engine controllers can greatly benefit from mathematical optimization. This requires however an executable model of the ECU functions. The following problem arises: Engine calibration is typically performed by an OEM, while the ECU code is owned by the supplier of the ECU. Therefore, the OEM is typically unable to set up an ECU simulation based on the original C code of the ECU. Instead, to set up optimization on PC, time consuming and error prone reverse engineering is needed to develop an 'equivalent model' of the ECU function of interest. To deal with this situation, we have implemented a novel method for automating the calibration of engine parameters. The method combines two ideas

- simulation of ECU program code on PC using chip simulation
- mathematical optimization based on the resulting executable model

The simulation requires only the hex, ASAP2/a2l and map file, that an OEM typically has access to, but not the source code of the ECU functions of interest.
This paper describes also a problem that we encountered when coupling chip simulation with optimization methods that require gradients to guide search for optima: partial derivatives of engine functions with respect to engine parameters are zero or infinite then, due to integer data types used by typical engine control code. As a result, gradient based optimization methods lack guidance and tend to terminate with sub-optimal solutions. The paper also sketches ideas how to overcome this problem and presents results of numerical experiments.

1. Simulation of ECUs on PC

Simulation has great potential to improve the development process for ECUs. Simulation helps to move development tasks to PC, where they often can be performed faster, cheaper or better in some respect [7]. To exploit these benefits, the ECU must first be ported to PC. This is typically done based on the C code of the ECU, which is either hand coded, or generated by tools such as Ascet (ETAS), TargetLink (dSPACE) or Embedded Coder (MathWorks). For example, QTronic's virtual ECU tool Silver [1] provides a framework to

- compile given ECU tasks for Windows PC,
- emulate the underlying RTOS and other services (CAN, XCP),
- run the resulting virtual ECU closed-loop with a simulated vehicle.

Typical applications are [2, 6], where a virtual ECU is used to develop the controller for an automatic transmission. For closed-loop simulation, vehicle models can be imported from many simulation tools into Silver, including MATLAB/Simulink, Dymola, SimulationX and MapleSim, e.g. through the FMI format for model exchange [4].

However, sometimes C code is not available for implementing a virtual ECU. There are two main sources for such a situation:

- *Protection of intellectual property*: All or major parts of the ECU have been developed by a supplier and the OEM interested in building a virtual ECU (e.g. to support calibration, a task typically performed by an OEM) has therefore no access to the C code.
- *Target-specific C code*: C code is available but the C code uses pragmas and other target or compiler specific constructs, which prevents compilation for other targets, such as the Windows x86 platform.

To deal with such situations, we have recently integrated a chip simulator into the virtual ECU tool Silver. This way, a virtual ECU can be build based on a hex file compiled for the target processor of the ECU. No access to C code is needed in this case. Instead of compiling C code for the Windows x86 platform, the chip simulator takes the binary compiled for the target processor and simulates the execution of the instructions by the target processor on Windows PC. Such a simulation requires

1. a hex file that contains program code and parameters of the simulated functions
2. start addresses of the functions to be simulated
3. an ASAP2/a2l file that defines the conversion rules for the involved inputs, outputs, and characteristics, as well as corresponding addresses

The start addresses of functions can e. g. be extracted from a map file generated together with the hex file. Silver uses the a2l file to automatically convert scaled integer values to physical values and vice versa during simulation. Such a chip simulation model can also be exported as SFunction (mexw32 file) for use in MATLAB/Simulink. On a standard PC, hex simulation runs with about 40 MIPS. If only simulating selected functions of an ECU, this is fast enough to run a simulation much faster than real-time.

The paper is structured as follows: Section 2 describes how to use chip simulation to build and run a virtual ECU on PC. In section 3, we report how the resulting ECU model has been coupled with numerical optimization to automate engine calibration.

2. Chip simulation for TriCore targets

Many automotive controllers are based on processors of Infineon's TriCore family, in particular in the power train domain. Examples are engine controllers of the MED and EDC family by Bosch and transmission controllers by Continental. Since the initial release of TriCore AUDO (AUtomotive UnifieD-ProcessOr) in 1999, Infineon has released four updates of the TriCore architecture, named AUDO NG (e.g. TC1796), AUDO Future (e.g. TC1797), AUDO MAX (e.g. TC1798), and AURIX. All members of the TriCore family are based on the same instruction set. Individual chips differ in memory maps, kind of memory, on-chip devices, such as CAN controllers, and interfaces to external devices. This section describes the support for TriCore chip simulation as provided by Silver 2.5.

2.1 Turning a hex file into a virtual ECU

The software of an ECU consists of a real-time operating system (RTOS) that runs functions (tasks) at specified times, either initially, periodically or at certain events, such as angle positions reached by the crankshaft. Three kinds of tasks can be distinguished

1. tasks that generate signals, e.g. by reading sensors or CAN messages
2. tasks that compute output signals from input signals
3. tasks that use signals to command actuators or to create CAN messages

The tasks of categories 1 and 3 typically depend on details of the particular chip (such as hundreds of registers of on-chip devices), and on the ECU hardware. In contrast, tasks of category 2 are fairly independent from such hardware-specific details. To simulate ECU code, it is therefore convenient to run only tasks of category 2. The required inputs for these tasks can either be taken from measurement files (open-loop simulation), or they are computed online by some plant model (closed-loop simulation), bypassing the tasks of category 1. Likewise, the outputs of category 2 tasks can be directly compared to measurements (open loop) or fed into the plant model (closed loop), bypassing the category 3 tasks. The signal interface between categories 1-2 and 2-3 is typically well documented and available, e.g. from the CAN specification (DBC file) of the ECU.

This modelling strategy has a very good cost-benefit ratio. In order to simulate also the tasks of categories 1 and 3, one has to model hundreds or peripheral and chip specific registers, and to build state-machine models for low-level peripherals, such as CAN controllers. Technically, this is possible, e. g. with SystemC [5], but hardly justified by the added value, at least for the application considered here.

Silver 2.5 uses a specification file (similar to the OIL file used to configure OSEK) to specify, which tasks of a hex file to simulate. Silver automatically turns such a spec file into an executable Silver module (dll) or SFunction. A typical spec file looks as follows:

```
# specification of sfunction or Silver module
hex_file(m12345.hex, TriCore_1.3.1)
a2l_file(m12345.a2l)
map_file(m12345.map)      # a TASKING or GNU map file
frame_file(frame.s)       # assembler code to emulate RTOS
frame_set(STEP_SIZE, 10)  # Silver step size in ms
frame_set(TEXT_START, 0xa0000000) # location of frame code

# functions to be simulated, in order of execution
task_initial(ABCDE_ini)
task_initial(ABCDE_inisyn)
task_triggered(ABCDE_syn, trigger_ABCDE_syn)
task_periodic(ABCDE_20ms, 20, 0)
task_periodic(ABCDE_200ms, 200, 0)

# interface of the generated sfunction or Silver module
a2l_function_inputs(ABCDE)
a2l_function_outputs(ABCDE)
a2l_function_parameters_defined(ABCDE)
```

The hash `#` character starts a comment, which is ignored by Silver. The spec file first lists the required files (line 2-5). The map file is optional. If a map file is given, the spec file may use symbolic names for functions (such as `ABCDE_20ms`). Otherwise, addresses (such as `0x80081cde`) must be used. File frame.s (line 5) contains startup code and the generic part of the RTOS emulation used to run the tasks. As usual, the startup code sets up stacks, registers, timer and other resources.

Lines 10 - 14 lists the functions to run, and specifies when and in which order to run these functions. Silver uses this to generate the application-specific part of the RTOS emulation. For event triggered tasks, Silver offers two alternative event models. Line 12 shows a function that is executed *n* times at each Silver step, where *n* is the value of the input variable `trigger_ABCDE_syn` at the beginning of the step. Typically, *n* is 0 or 1 during simulation. Higher values occur only, when more than one trigger event occurs during one step. Silver also offers a more accurate event model, that allows execution of an event triggered task at exact event time, not just at the beginning of a step.

Finally, lines 17-19 define the inputs, outputs and parameters of the generated module or SFunction. In this case, we just reuse the interface of a FUNCTION element of the a2l file, for a function called ABCDE. It is also possible, to list individual variables here by name, as long as their properties (such as address, conversion rule, data type) are described in the a2l file.

In addition, the spec file offers means to specify

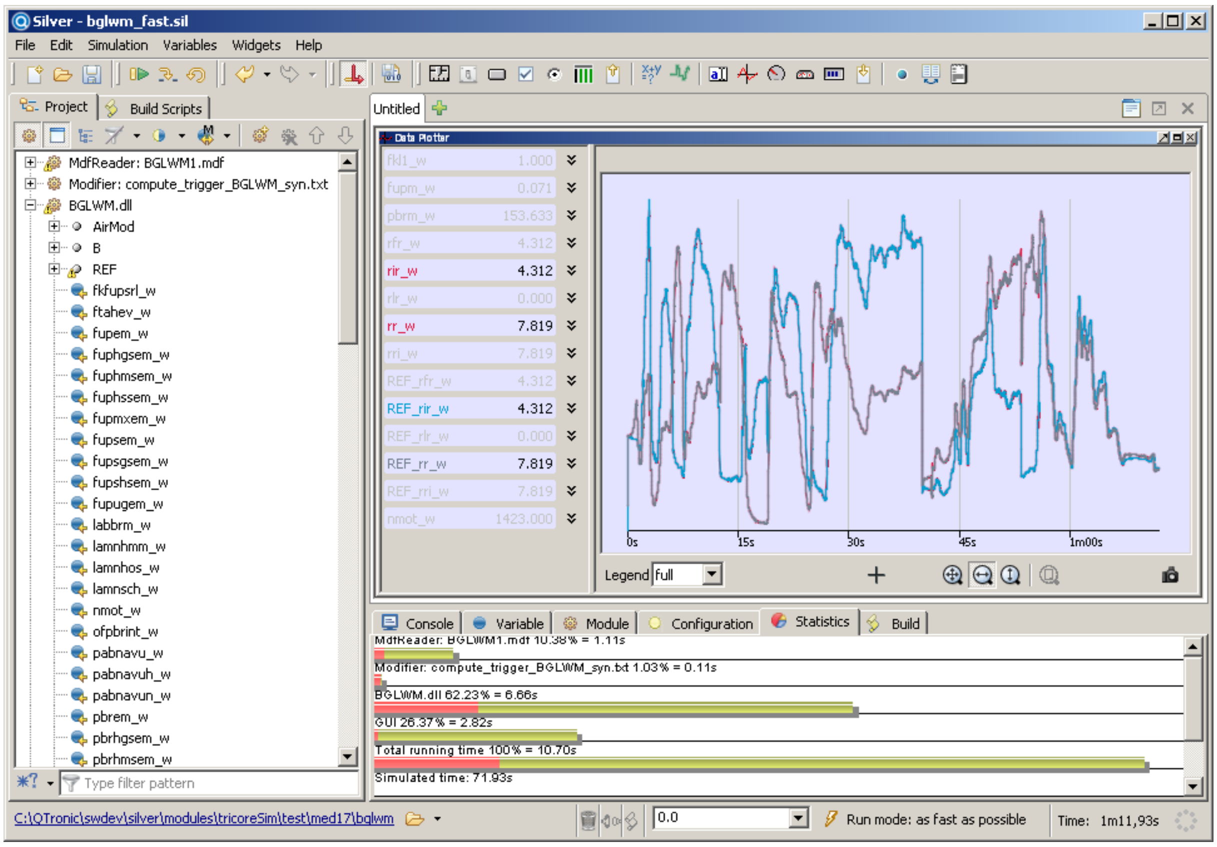
- properties of the XCP emulation, if any, to support online calibration and measurement using tools such as INCA and CANape
- data sections to be included into the generated Silver module or SFunction. This way, initial loading of the hex file into simulated memory can be avoided, to speed up simulation.
- memory areas to be copied to other (faster) memory by the start-up code
- functions to be replaced by other functions. This way, a function called by a task of category 1 or 3 to access sensors or actuators can be replaced by a function that directly accesses a plant model or measured values instead.
- logging options, e.g. to track memory access during simulation

The Silver module or SFunction generated this way performs exactly the same computations on PC, as on the real target, since the effect of every machine instruction on memory and chip registers is exactly simulated on PC. However:

- simulation is just instruction accurate, not cycle accurate. This means, the simulation on PC cannot be used to exactly predict execution time on the real target. For example, pipeline effects of different access times to memory (e.g. fast on-chip RAM vs. external RAM) are not modelled.
- conceptually, simulated tasks execute infinitely fast. This means that the emulated RTOS never interrupts a task. The corresponding effects cannot be analysed using the generated model.
- Silicon bugs are not simulated. If a compiler for the real target does not work around a silicon bug correctly, this is likely to be invisible in the simulation: simulated behaviour and behaviour on the real target might differ in such cases.

2.2 Debugging the virtual ECU

The spec file used to port selected parts of a hex file to PC might contain bugs. To locate bugs, Silver integrates a debugger based on the instruction set simulator tsim, developed by Infineon. This debugger is used whenever a simulation does not perform as expected, i.e. differs from measured behaviour. Silver can be switched to step mode. In this mode, Silver uses tsim to run just one TriCore instruction per step, allowing a user to inspect register content before and after execution of an instruction. It is also possible to set code and data breakpoints, for example to pause a simulation whenever a certain variable is accessed.



fkl1_w	1.000
fupm_w	0.071
pbrm_w	153.633
rfr_w	4.312
rir_w	4.312
rlr_w	0.000
rr_w	7.819
rri_w	7.819
REF_rfr_w	4.312
REF_rir_w	4.312
REF_rlr_w	0.000
REF_rr_w	7.819
REF_rri_w	7.819
nmot_w	1423.000

Legend: full

```
MdfReader: BGLWM1.mdf 10.38% = 1.11s
Modifier: compute_trigger_BGLWM_syn.txt 1.03% = 0.11s
BGLWM.dll 62.23% = 6.66s
GUI 26.37% = 2.82s
Total running time 100% = 10.70s
Simulated time: 71.93s
```

C:\QTronic\swdev\silver\modules\tricoreSim\test\med17\bglwm   0.0   Run mode: as fast as possible   Time: 1m11,93s

Fig 1: The BGLWM function running in Silver, driven by measurement file

2.3 Execution times

In order to measure the execution speed of chip simulation, we have ported a complex ECU function implemented by 5 different C functions that run initially, every 10 and 200 ms, and synchronous to the crankshaft. The spec file is very similar to the one shown in section 2.1. The function has 114 scalar inputs, 102 scalar outputs and 108 parameters (characteristics), many of them axes and maps. We have then measured all inputs and outputs of the function on an engine test rig for a scenario of 3.5 minutes and used the resulting measurement (mdf/dat) file to drive simulations in Silver, using either tsim or a generated Silver module. Each simulation executed 380.205256 million instructions (counted by tsim) and has been repeated 5 times on a Windows PC with Intel i5 processor at 2.4 GHz and 2.92 GB RAM. Average execution times found this way are shown in Table 1.

simulator	execution time on PC	MIPS
Infineon tsim	919.15 sec	0.41
Silver module	9.30 sec	40.80

Table 1: Performance of chip simulation for the BGLWM example

The ECU considered here (MED17 with TC1797) runs at 200 MHz and has a performance of about 300 MIPS. Nevertheless, on the ECU, the execution time for the 3.5 minutes scenario is of course exactly 3.5 minutes, due to the real time constraint. On a PC, this function runs 20 times faster.

2.4 Exporting a simulation to MATLAB/Simulink

Silver can also turn a spec file as described in section 2.1 into a SFunction, i.e. a mexw32 file that runs in Simulink. This is particularly interesting when using chip simulation to support automated optimization of parameters, because many optimization tools are implemented on top of MATLAB/Simulink. The generated SFunction accepts all characteristics listed in the spec file as SFunction parameters. This makes it easy to connect the generated SFunction with an optimization procedure. For example, the SFunction can be called with workspace variables that are then automatically varied by the optimization procedure between SFunction calls. The performance of a generated SFunction is again about 40 MIPS.

3. Application to the numerical optimization of engine parameters

We have combined chip simulation as described above with a procedure for numerical optimization to compute optimal values for certain engine parameters. These computations require an accurate and fast model of the engine function of interest. In the past, we have used hand-coded models of ECU functions, developed with MATLAB/Simulink. This has been time consuming and error prone. We have now partially replaced these hand-coded models with SFunctions generated automatically by Silver from a given hex file. The generated SFunctions proved to run as fast as their hand coded counterparts. The replacement of hand-coded floating-point models by generated fixed-point SFunctions raises the following problem: Some optimization procedures require gradient information to guide the search for optimal parameter values: When searching for an x that minimizes $f(x)$, the derivative df/dx is to be computed during optimization for different values of x. Finite differences are often used here: df/dx is computed as $(f(x + h) - f(x)) / h$ for small h, say $h = 10^{-6}$. If f is computed using chip simulation, x and $x+h$ are often both mapped to the same integer, resulting in a zero gradient. As a consequence, the optimization procedure is lacking guidance, and might return a suboptimal solution.
This section presents ideas how to overcome this problem and some results of numerical experiments. There are also so-called derivative-free procedures for optimization. Obviously, these are not affected by the above problem. This is exploited in [8].

3.1 Least squares optimization

Optimization in engine development can frequently be formulated as least-squares optimization. The objective is then to minimize a goal function

$$g(x)=\sum_{i=1}^{m} f_i^2(x) \tag{1}$$

where x is a vector of n real valued parameters. A typical application is curve fitting. The engine controller contains a function $model(x, t)$ that estimates a physical quantity that the controller cannot measure directly. This model needs to be calibrated by choosing parameters x such that a measured series of m data points is predicted by the model as good as possible, i.e. the squared sum of the m real-valued residuals

$$f_i(x)=model(x,t_i)-measurement(t_i) \tag{2}$$

gets minimized. In typical applications, there are hundreds of data points and parameters.

Algorithms typically used for least-squares optimization approximate for different choices of x the Jacobian

$$J_{i,j}(x)=\lim_{h\to 0}\frac{f_i(s(x,j,h))-f_i(x)}{h} \tag{3}$$
$$s_k(x,j,h)=if\,(j=k)\,then\,(x_k+h)\,else\,x_k$$

to determine at a given point x in parameter space the direction of steepest descent of $g(x)$. Each element of the above Jacobi matrix is typically approximated by a finite difference

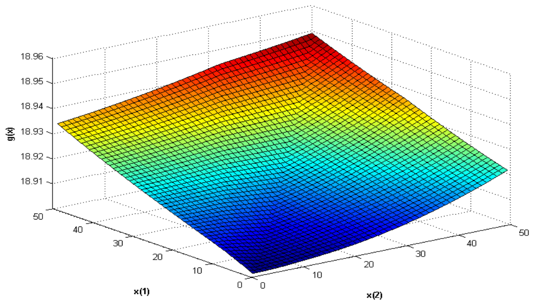
$$D_{i,j}(x)=\frac{f_i(s(x,j,h))-f_i(x)}{h} \tag{4}$$

with sufficiently small h, say h = 10^{-6}.

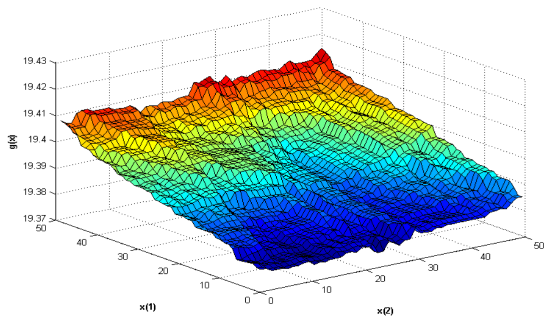
3.2 A problem with chip simulation

Engine controllers are frequently implemented using fixed-point code, i.e. all computations are performed using integers, not floating point numbers. As a consequence, when implementing the goal function g (or just the residuals f) using chip simulation, the elements of the Jacobi matrix (3) are either zero or undefined (infinite). This is illustrated in Fig. 2. for a certain engine control function.

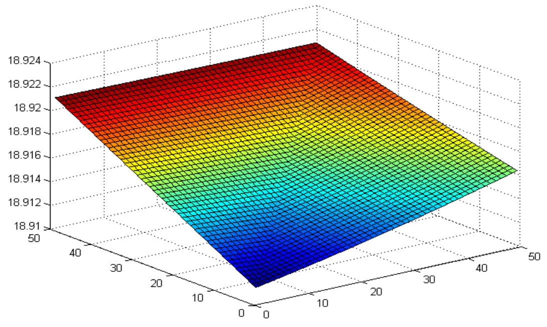
Figures 2*a* to *h* show how goal function $g(x)$ depends on two selected parameters x_1 and x_2. In cases *a*, *c*, *e, g*, the goal function is implemented in Simulink using a reverse-engineered model of the engine controller. In cases *b*, *d*, *f*, *h*, chip simulation has been used to implement $g(x)$ based on the original controller code. When ranges for x_1 and x_2 are chosen large enough (*a* and *e*), both implementations look fairly similar. At smaller scales, the goal function computed by chip simulation turns into a increasingly rugged landscape (*d* and *h*), while the Simulink model stays smooth at all scales (*c* and *g*).



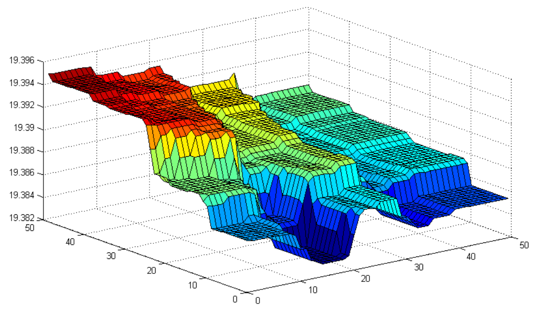
a) Simulink model



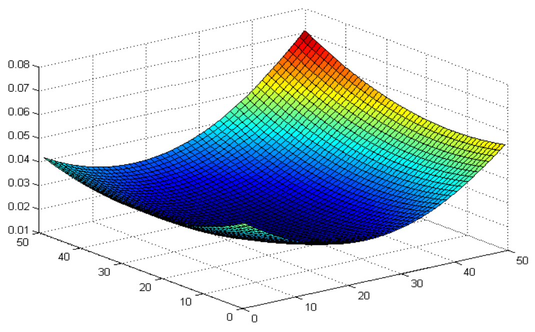
b) Chip simulation: same scale as a)



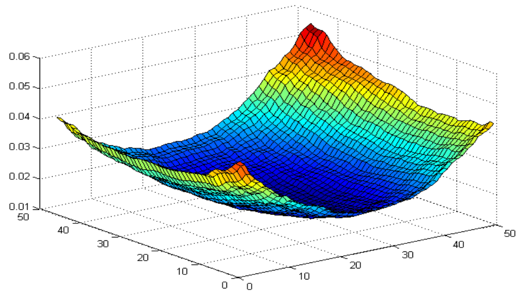
c) Simulink model: scale up



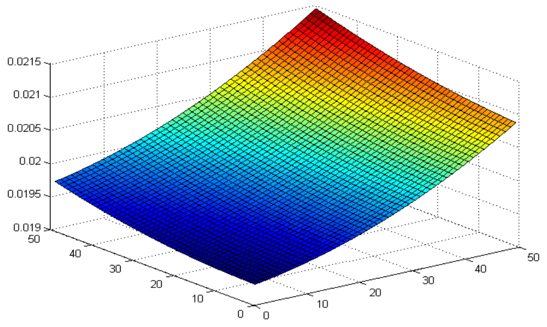
d) Chip simulation: same scale as c)



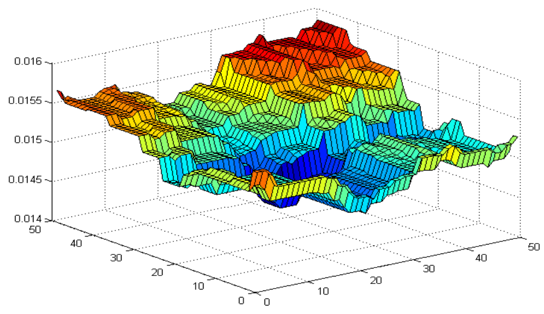
e) Simulink model, around the optimum



f) Chip simulation, same scale as e)



g) Simulink model, optimum scaled up



h) Chip simulation, same scale as g)

Fig 2: Goal function varied for two parameters x1 and x2

In general, when optimizing goal functions implemented using chip simulation with solvers that compute derivatives to guide the search, these solvers typically return early with a sub-optimal solution, since finite differences computed with (4) are nearly always zero and do not provide guiding information. Solvers tend to conclude, that they reached an optimum (since all derivatives are zero), and return a suboptimal solution.

3.3 Ideas for solving the problem

The main idea to overcome the above problem is to smoothen the discrete goal function and to pass the resulting smooth function to the solver used for optimization.

To construct such a smooth goal function, one might consider to replace the integer operations used by the chip simulator by the corresponding floating point operations. This does not work however, because it would either indifferently turn all arithmetic operations into floating point operations (even those that need to remain discrete, such as array index computations), or would require to distinguish these two cases for each operation in program code, which is impractical.

A mathematical rigorous idea is to explicitly construct a smooth goal function by interpolating the grid points of the discrete goal function g, and to determine the corresponding derivatives analytically. However, $g(x)$ is a function of n parameters. Each point x in n-dimensional parameter space has 2^n neighbouring grid points. Linear interpolation of a function value requires hence to compute a sum of 2^n terms. This seems infeasible, since n is about 100 here.

Instead, we replace in (4) the fixed step size h by the term $k\,H_{i,j}(x)$, which is sensitive to the grid size at point x

$$D_{i,j}(x)=\frac{f_i(s(x,j,k\,H_{i,j}(x)))-f_i(x)}{k\,H_{i,j}(x)} \qquad (5)$$

$H_{i,j}(x)$ is the discretisation grid size defined as follows:
$H_{i,j}(x) = max - min$, where max is the maximal and min is the minimal real number such that $h \geq min \wedge h \leq max \Rightarrow f_i(s(x, j, h)) = f_i(x)$.

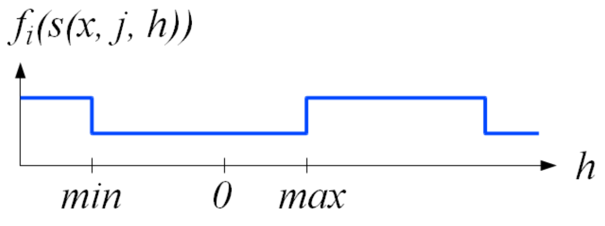


Fig 3: Discretisation grid size $H_{i,j}(x)$ for residual function f_i and parameter x_j

The use of $H_{i,j}(x)$ in (5) to compute a finite difference leads to non-zero differences in many cases. This provides guiding information for optimization, since it takes the function value beyond the next grid point into account. The discrete residual functions f_i are however subject to chaotic integer up and down rounding, which causes the rugged landscape seen in Fig. 2*d* and *h*. The constant factor k is introduced to compensate this. For example, choosing k = 10 averages the derivatives across 10 grid points, which reduces the noise generated by integer rounding.

For given x, each element of the matrix $H(x)$ can be computed by searching for the lower (min) and upper (max) grid point, e.g. using a binary search procedure. This is computational expensive. For the application reported here, the following simplifying assumption lead to good results: the matrix H does not change significantly during the optimization process, i.e. $H(x0) \approx H(xOptim)$, where $x0$ is the given start value for optimisation, and $xOptim$ is an optimal solution. It is sufficient then to compute H for the start vector $x0$ only, not for every intermediate point x considered during the optimisation process.

To avoid the expensive computation of all elements of H, one might also use a stochastic model of H as follows: for each of of the n parameters x_j, compute $H_{i,j}(x)$ for just a few, not all m time points t_i, and derive mean value μ_j and standard deviation σ_j of the discretisation grid size from that. During the entire solution process, the constant step width

$$h_j = k(\mu_j + 3\sigma_j) \qquad (6)$$

is then used in (4) to compute finite differences for parameter x_j, where $k \geq 1$ is again a smoothing factor to average across more than just one grid point.

3.4 Numerical results for a engine control function

Many optimization procedures that use derivatives provide also options to control the step size of finite differences. Such options are required in order to apply the ideas presented in 3.3. For the experiments reported here, we have used *lsqnonlin* from the MATLAB optimization toolbox. Procedure *lsqnonlin* provides options to control the step size h, either globally (option *DiffMinChange*), or with individual lower limit for each of the n parameters (option *FinDiffRelStep*). Alternatively, a user defined procedure for computing the Jacobian matrix can be provided (option *Jacobian*), which might then use its own step sizes. In all three cases, either the matrix H in (5), or the stochastic model (6) of H can be used to control the step size of finite differences.

We validated the ideas presented in 3.3 using a curve fitting problem of an engine control function with n = 20 parameters and m = 202 time points. The goal function has been implemented using chip simulation and - for reference - also using a hand coded Simulink model. The goal function implemented by chip simulation and passed to *lsqnonlin* with option *FinDiffRelStep* in combination with step size limits (6) generated solutions of roughly the same optimality and number of required function evaluations as the reference set up based on the hand-coded Simulink model. Optimization using chip simulation took however a factor 2 longer, due to the additional initial computation of step size limits (6) from the given start value $x0$.

One interesting point is cross-comparison of found solutions: The hand coded Simulink model generated a solution *xOptSimulink* with

$gSimulink(xOptSimulink) = 0.0148$

while optimisation with chip simulation generated a slightly different solution *xOptChipsim* with

$gChipsim(xOptChipsim) = 0.0149$

Cross-comparison shows that both goal functions define slightly different optima:

$gSimulink(xOptChipsim) = 0.0200$

$gChipsim(xOptSimulink) = 0.0217$

The goal function *gChipsim* is however a bit accurate model of the computation of the real engine controller, while *gSimulink* is a hand-coded model with a certain modeling error. We therefore believe that on the real engine controller, the solution found by chip simulation performs effectively better (0.0149) than the one found by the hand-coded Simulink model (0.0217).

4. Conclusions

As demonstrated above, an ECU hex file compiled for some target processor can be executed by the virtual ECU tool Silver on Windows PC, either open-loop driven by measurements or in closed-loop with a vehicle model. Depending on the application, selected ECU functions are simulated, or nearly the entire ECU. As shown in section 3, such chip simulations can be coupled with optimisation procedures.

This kind of simulation opens new possibilities to move development tasks from road, test rig or HiL to PCs, where they can be processed faster, cheaper or better in some respect, without requiring access to the underlying C code. Daimler currently uses this innovative simulation approach to support controls development for gasoline and diesel engines, see also [8]. Other applications, such as online calibration on PC via XCP seem to be doable as well.

References

[1] A. Junghanns, R. Serway, T. Liebezeit, M. Bonin: Building Virtual ECUs Quickly and Economically, ATZ elektronik 03/2012, Juni 2012. See www.ATZonline.de or http://qtronic.de/doc/ATZe_2012_en.pdf

[2] H. Brückmann, J. Strenkert, U. Keller, B. Wiesner, A. Junghanns: Model-based Development of a Dual-Clutch Transmission using Rapid Prototyping and SiL. International VDI Congress Transmissions in Vehicles 2009, Friedrichshafen, Germany, 30.06.-01-07.2009. http://qtronic.de/doc/DCT_2009.pdf

[3] K. Röpke (ed.): Design of Experiments (DoE) in Engine Development - Innovative Development Methods for Vehicle Engines. Expert Verlag, 2011.

[4] T. Blochwitz, M. Otter et. al.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. 9th International Modelica Conference, Munich, 2012.

[5] SystemC, Language for System-Level Modeling, Design and Verification, see www.systemc.org

[6] M. Tatar, R. Schaich, T. Breitinger: Automated test of the AMG Speedshift DCT control software. 9th International CTI Symposium Innovative Automotive Transmissions, Berlin, 2010. http://qtronic.de/doc/TestWeaver_CTI_2010_paper.pdf

[7] J. Mauss, M. Simons: Chip simulation of automotive ECUs. 9. Symposium Steuerungssysteme für automobile Antriebe, 20.-21.09.2012, Berlin. In: Nietschke und Predelli (Hrsg.): Steuerungssysteme für Automobile Antriebe, expert Verlag Renningen 2012.

[8] D. Rimmelspacher, W. Baumann, P. Klein, R. Linssen: Transient Calibration Process using Chip Simulation and Dynamic Modeling. 7th Conference on Design of Experiments (DoE) in Engine Development, Berlin, 2013.