2005 American Control Conference
June 8-10, 2005. Portland, OR, USA



# Optimal control-based powertrain feasibility assessment: A software implementation perspective

Ilya V. Kolmanovsky, Shiva N. Sivashankar, Jing Sun

*Abstract*— **With the increase in automotive powertrain complexity, an upfront assessment of powertrain capability in meeting its design targets is important early on in the development programs. The optimization of control policy based on powertrain simulation models can facilitate this assessment and establish limits of achievable performance for given powertrain configuration and parameters. The paper discusses several computational optimization and user interface solutions for deploying a numerical optimal control approach in a user-friendly software environment.**

## I. Introduction

An assessment of powertrain capability to meet its design targets and constraints (referred afterwards as *feasibility assessment*) is important early on in powertrain development programs. Both powertrain design parameters and its control policy influence the capability of powertrain to meet these targets and constraints. New powertrain systems, in particular, are characterized by an ever increasing number of control inputs as well as by strong static and dynamic interactions. Thus a synergistic treatment of design and control issues is especially important for the analysis of feasibility of such powertrains and in order to fully identify relevant design and control requirements.

The feasibility of *given values* of powertrain design parameters can be assessed on the basis of the *best achievable powertrain performance*, i.e., powertrain performance while operating under an optimal control policy. Due to multitude of objectives that powertrain systems must satisfy this is a *multi-objective* optimization problem. Thus the best achievable performance is characterized by a *Pareto* front, and the feasibility of the powertrain means that a subset of the Pareto front is located within the target set. See Figure 1. Different powertrain configurations or parameter values can be compared with each other based on selected points on the Pareto fronts (e.g., on the basis of best emission constrained fuel economy). By examining the behavior of trajectories corresponding to these selected points, subsystem level targets can be set and an insight into required control system architecture, actuator coordination and calibration can be gained.

In the early phases of the development programs, no hardware prototypes may yet exist or the technology may still be evolving. In this situation, the feasibility assessment can only be based on dynamic simulation models. The Pareto front can be computed via the numerical optimization applied to these models. Specifically, a weighted sum of the objectives can be minimized with respect to the control input trajectory for different values of the weights. A less computationally demanding approach is to only optimize certain parameters in an *a priori* defined parameter-dependent control policy. In the latter case, engineering judgement is necessary to define upfront a parameter-dependent control policy that is most likely to meet the targets. Also at times practical constraints make it not possible or desirable to change the control policy completely except for the values of a few parameters. In this case, the values of these parameters can be determined using parameter optimization while a complete optimal control policy can be used to understand the potential for further improvements.

In the paper we discuss the background and some of the features of a recently developed software environment for performing powertrain feasibility analysis. The software implements both the optimization based on dynamic programming and the optimization of parameters in parameter-dependent control policies. It provides automotive engineers, who may not be experts in optimization, with the capability to:

- interface with powertrain models in MATLAB/Simulink[1];
- define and solve state and control constrained optimal control problem via dynamic programming while taking advantage of fast/slow dynamics decomposition to deal with the curse of dimensionality;
- define parameter-dependent operating policies and perform parameter optimization;
- visualize solutions in different ways and analyze their sensitivity;
- automatically generate analysis reports;
- speed-up the optimization and parameter sweeps by taking advantage of parallel computations.

## II. Implementation of Dynamic Programming

The optimal powertrain control policies are computed using the dynamic programming. It is applied to discrete-time system models decomposed into a cascade of a static

I.V. Kolmanovsky is with Ford Motor Company, 2101 Village Road, Dearborn, MI 48124, E-mail: ikolmano@ford.com.

S.N. Sivashankar is with Emmeskay, Inc., Plymouth, MI 48170. E-mail: shiva@emmeskay.com.

J. Sun is with Department of Naval Architecture and Marine Engineering, the University of Michigan, Ann Arbor, MI 48109. E-mail: jingsun@umich.edu.

[1]MATLAB and Simulink are registered trademarks of the MathWorks, Inc. of Natick, MA.

0-7803-9098-9/05/$25.00 ©2005 AACC

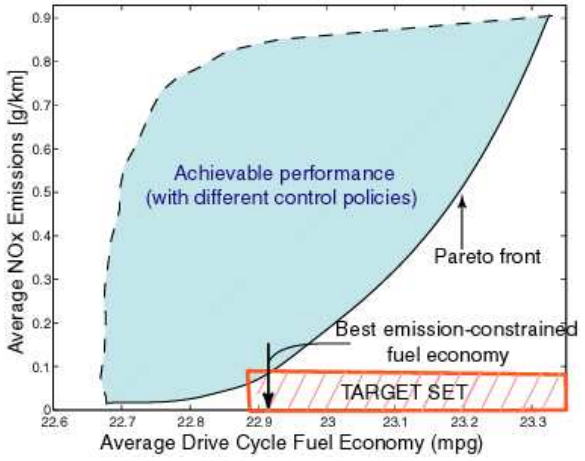


Fig. 1. Feasibility assessment: Achievable performance, best achievable performance and target performance.

nonlinear subsystem and a dynamic nonlinear subsystem,

$$\begin{aligned} x(t+1) &= f_d(x(t), u_d(t), w(t), v(t), p), \\ v(t) &= f_s(w(t), u_s(t), x(t), p), \end{aligned} \tag{1}$$

where $x$ is the state of the dynamic subsystem, $u_d$ is the control input of the dynamic subsystem, $w(t)$ is the vector of known operating variables of the powertrain (such as engine speed and engine torque or wheel speed and wheel torque), $p$ is the vector of powertrain design parameters, $v(t)$ is the output of the static subsystem, and $u_s(t)$ is the control input of the static subsystem. The decomposition (1) can be viewed as an approximation of a system with slow and fast dynamics. Such a time scale separation occurs frequently in powertrain applications. The fast dynamics can be approximated by the static subsystem for $v$ while the slow dynamics by the dynamic subsystem for $x$. Since dynamic programming based optimization becomes computationally demanding as the state dimension increases, the main advantage of the approximation (1) is keeping the effective state dimension low so that the attention can be focused on the behavior of key states of the powertrain.

For example, in the case study in [2], a Direct Injection Spark Ignition (DISI) engine behavior was approximated as a static subsystem so that $w(t)$ is the vector of engine speed and engine torque at time $t$; $v(t)$ is the vector of exhaust air-to-fuel ratio, exhaust temperature, oxides of nitrogen feedgas flow rate, hydrocarbon feedgas flow rate, carbon monoxide feedgas flow rate, exhaust mass flow rate and fueling rate at time $t$; and $u_s(t)$ is the vector of in-cylinder air-to-fuel ratio, exhaust gas recirculation (EGR) rate, spark timing, injection timing, fuel rail pressure, swirl control valve setting, and fueling rate at time $t$. The dynamic subsystem in [2] represented the behavior of the aftertreatment system and the state $x$ was a vector of stored oxides of nitrogen and oxygen in aftertreatment system, and aftertreatment system temperature. See Figure 2. The parameters $p$ define the properties of the aftertreatment system. In that case parametric models describing the sensitivity of performance variables existed and had been incorporated.

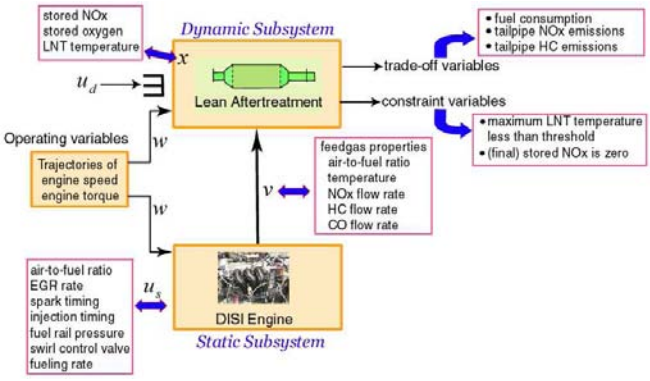


Fig. 2. Approximation of DISI engine and aftertreatment system behavior as a cascade of a static subsystem and a dynamic subsystem.

Another computational simplification is to consider only a finite set of possible control policies for the static subsystem so that

$$u_s = g(w, x, m), \quad m = 1 \cdots, M, \tag{2}$$

where $m$ is an integer which identifies the control policy out of $M$ total control policies. Then the dynamic programming can be applied to the optimization of the control input

$$u = \begin{bmatrix} u_d \\ m \end{bmatrix}. \tag{3}$$

This procedure results in significant computational savings in those cases when the dimensionality of $u_s$ is large. A finite set of representative control policies can be generated in a number of different ways, depending on the application. In [2], following the approach of [3], a weighted sum of engine fuel consumption and emissions was optimized at each engine speed and engine torque operating point, $w$, for a finite set of weight combinations and for two different combustion regimes (stratified and homogeneous); each combination of a weight and a combustion regime would then define a separate policy for the static subsystem (engine). In effect, in that application $m$ may be viewed as an identifier for one of a finite number of engine maps optimized for different trade-off between fuel consumption and emissions. See Figure 3.

With (2) and (3), the system (1) can be written in the form

$$x(t+1) = f(x(t), u(t), w(t), p). \tag{4}$$

The optimization objective is to minimize a stage-additive cost functional,

$$\sum_{t=0}^{t=T} J(x(t), u(t), w(t), p) \to \min,$$

where the incremental cost, $J(x, u, w, p) = \lambda^{\mathrm{T}} y$, is a weighted sum of trade-off variables,

$$y(t) = h_y(x(t), u(t), w(t), p), \tag{5}$$

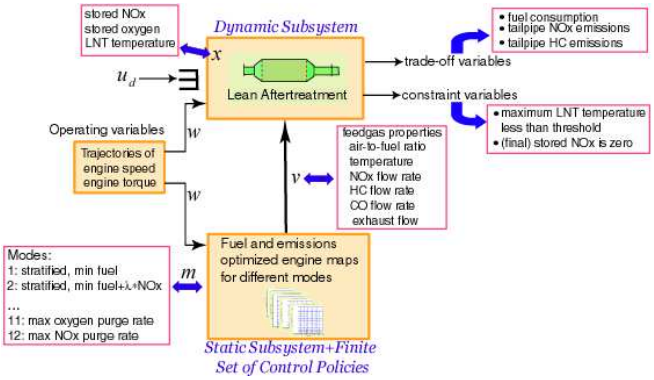


Fig. 3. Restricting static subsystem to a finite set of control policies.

and where $\lambda$ is the vector of the weights. The software provides flexibility to specify whether the weighting factors are applied at all time instants, at a final time instant or only at certain specified time instants. It can also select the weights automatically. The optimization is repeated for a specified sweep of the powertrain parameters, $p$.

The dynamic programming relies on backward-in-time value function iterations. Specifically, let $V(t,x,p)$ be the cost-to-go from the time instant $t$ and state $x$ to the end of the optimization horizon, $t=T$. Suppose $V(t+1,\cdot,p)$ has been already determined and let

$$Q(t,x,u,p) = J(x,u,w(t),p)+V(t+1,f(x,u,w(t),p),p). \tag{6}$$

Then the state and time-dependent optimal control at time $t$ satisfies

$$u^*(t,x,p) \in \min_u Q(t,x,u,p), \tag{7}$$

and $V(t,x,p)$ is determined as

$$V(t,x,p) = Q(x,t,u^*(t,x,p),p). \tag{8}$$

To perform iterations (6)-(8) numerically $x(t)$, $u(t)$ and $w(t)$ are gridded. The grids for $x(t)$ are defined via the *State Grid Set*, $X_{ij}$, and *State Grid Sequence function*, $S_X$. Specifically, $X_{ij}$, $i=1,\cdots,I_X$, is the $i$th element of the *State Grid Set* for the $j$th component of the vector $x(t)$; $S_X(t)=i$ implies that the $j$th component of the vector $x(t)$ assumes values from the grid $X_{ij}$. The grids for $u(t)$ are defined analogously with the help of *Control Sets*, $U_i$ and a *Control Set Sequence function*, $S_U$. Specifically, $U_{ij}$, $i=1,\cdots,I_U$, $j=1,\cdots,K(i)$, is the $j$th vector in the $i$th *Control Set*, $U_i$; $S_U(t)=i$ implies that $u(t) \in \{U_{i1},U_{i2},\cdots,U_{K(i)}\}$ (the number of vectors, $K(i)$, may be different for different $i$). Finally, the *Operating Variable Set*, $W$, and *Operating Variable Sequence function*, $S_W$, are introduced so that $W_i$, $i=1,\cdots,I_W$, is the $i$th vector in the *Operating Variable Set*, $W$, and $S_W(t)=i$ implies that at time $t$, $w(t)=W_i$.

If the user desires to use the same state grids for all $t$, then $I_X=1$ and $S_X(t)=1$; in this case the user can specify the grid for each state variable as a vector using the *Regular Grid* option, see Figure 5. The software can also pick the regular grids automatically. The *Advanced Grid* option is used if $I_X>1$, and in this case the *State Grid Set* and *State Grid Sequence* function are loaded from MATLAB workspace or from a file. The latter option can be used to implement iterative dynamic programming whereby the state grid at each time instant is centered around the optimal state trajectory from the previous iteration and reduced in size with each new iteration (the total number of the grid points remains the same for each iteration). The iterative dynamic programming may be used when the state dimension is high and the use of conventional dynamic programming is computationally prohibitive[2].

The *Control Sets*, $U_i$, and *Control Set Sequence function*, $S_U$, may be specified by the user in MATLAB workspace or in a file. They can also be generated as a finite set of control policies (2), (3) with the help of the special static optimization environment applied to the static subsystem in (1). The static optimization environment is supported by its own set of integrated graphic user interfaces, see Figure 4. The user can define multiple operating modes, where each mode may have its own objective function, equality and inequality constraints, and ranges for input variables. The objective function for each mode is specified as a weighted sum of the trade-off variables for this mode and is minimized with respect to $u_s$ for different values of the weights at different operating points (defined by $w$). Each $m$ in (2) then corresponds to a particular combination of an operating mode and a weight in the static optimization environment. The static optimization environment can be applied either to MATLAB/Simulink models of the static subsystem or to a static subsystem response data set (e.g., engine mapping data). A special interpolation procedure was implemented to enable the latter option. The *Control Set* reduction functionality performs a pairwise comparison of the vectors in each of $U_i$, and eliminates one in the pair of two if they deviate from each other by less than the specified tolerance.

The *Operating Variable Set*, $W$, and *Operating Variable Sequence function*, $S_W$, can be either directly specified by the user (in MATLAB workspace or in a file) or they may be generated automatically based on the built-in powertrain and drive cycle libraries. In the latter case, the user selects a vehicle type and a drive cycle type. Similarly to *Control Set* reduction, the *Operating Variable Set* reduction functionality can be applied to reduce the number of elements in $W$ based on the user-defined tolerances.

To improve the computational efficiency, the output map, $h_y(x,u,w,p)$ and the state update map $f(x,u,w,p)$ are first pre-computed for all values of $x$, $u$ and $w$ in the specified *State Grid Set*, *Control Sets* and *Operating Variable Sets*. Then the calculation of $J=\lambda^{\mathrm{T}}y$ and the updates (6)-(8) are made in the vectorized form which can be efficiently

[2]The resulting optimal control will, however, be specific to the selected initial condition.

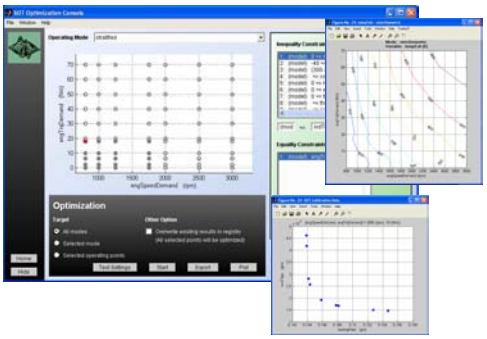

Fig. 4. Optimization console of the static optimization environment shows the status of static optimization at different operating points ($w$) after its completion. If circle is 'filled' then a feasible solution has been found, otherwise it is 'hollow'. The user can analyze the optimal responses using trade-off, contour and surface plots which can be open by clicking on operating points (examples shown).

and rapidly[3] handled by MATLAB [2]. The user is provided with several options to trade-off available RAM versus the speed of computations. If memory is limited, the pre-computed output map and state update map can be stored in multiple files and are loaded into the memory on "as needed" basis.

The pointwise-in-time constraints, $z_{min} \leq z(t) = h_z(x(t), u(t), w(t), p) \leq z_{max}$, are handled by augmenting a penalty term,

$$\sum_{t=1}^{t=T} \lambda_z^{\mathrm{T}} \phi(z(t)),$$

to the cost function. Here $\phi$ is a built-in penalty function applied to each component of $z$, which is non-zero only if $z(t)$ violates the constraints while $\lambda_z$ is the vector of weights. The software provides flexibility to handle constraints that are either persistent (i.e., applied at all time instants), final (applied only a final time instant) or intermittent (applied only at specified time instants). It prebuilds the tables for $z$ in the same way as for $y$.

The software environment leads the user through a series of integrated Graphic User Interfaces (GUIs) that load the model and model information file; define *State Grid Set*, *Control Sets* and *Operating Variable Set* as described above; define objective function, constraints and desired parameter sweeps; and, finally, provide means to visualize and analyze the results in numerous ways. One of the user interfaces for running the dynamic programming is illustrated in Figure 5. Prior to running the dynamic programming (which is a computationally intensive and time-consuming task) the user can quickly test settings to check for errors and warnings; in particular, the software checks that it can calculate all expressions that the user has defined. The user-specified initialization and termination scripts are executed before the start of the optimization and after its completion; the initialization script may also be specified to run after each parameter or weight run.

[3]This property helps reduce computing times in a situation when new weights are added iteratively based on the analysis of past dynamic programming results.

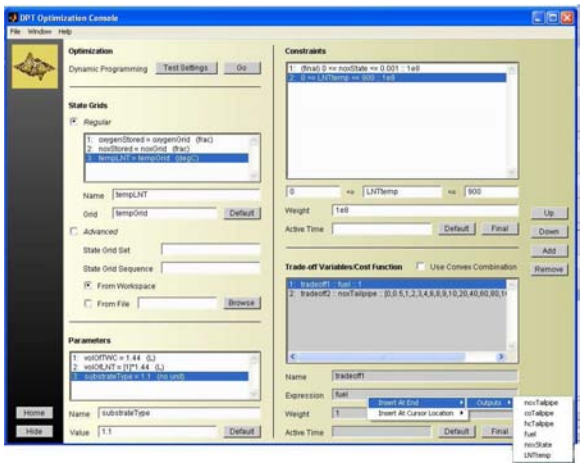

Fig. 5. The appearance of Graphic User Inteface for running dynamic programming. The insertion menu at the bottom is opened with a right mouse click and enables the user to enter a previously defined variable name into a trade-off variable expression.

Once the dynamic programming phase has been completed, the user can analyze and visualize optimal trajectories in numerous ways. Since the dynamic programming provides an optimal control *function*, $u^*(t, x)$, the optimal trajectory corresponding to any initial condition can be generated rapidly via simulations of the model (either of the same model as was used for optimization or, if necessary, of a higher fidelity model). Figure 6 shows a trade-off plot where by clicking on any point the user can plot the trajectories of optimal inputs, outputs and states corresponding to that point. A smoothing utility permits the user to graphically change the control trajectory (e.g., smooth it out if it shows excessive chattering) and compare the resulting cost and constraint violation with the original optimal solution. The original solution is referred to as *Parent* while the solutions derived via this smoothing are referred to as *Children*; the software keeps track of *Children* and graphically shows them on the trade-off plot connected to *Parent* via dashed lines. The smoothing can be particular effective when it is applied to the mode $m$ in (3). Since sometimes optimal solutions may appear non-intuitive, smoothing provides a mechanism for checking if certain features of optimal trajectories are really required, or if they may be caused by model irregularities. The software implements a database to avoid recomputing the optimal dynamic programming solutions or simulated trajectories if they have been already computed and the problem formulation did not change.

## III. Implementation of Parameter Optimization Approach

In the parameter optimization approach [3] only certain parameters, $q$, in a specified control policy, $u^0(t, x, q)$, are optimized. To specify a control policy within the software

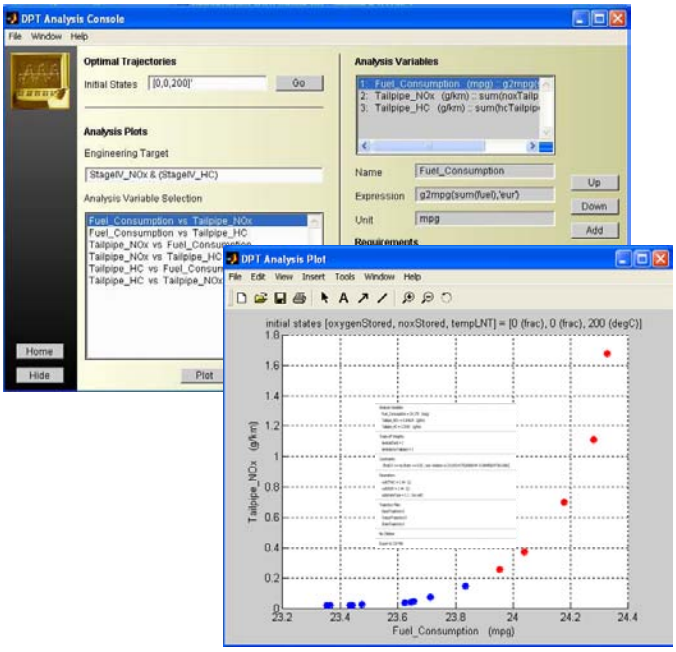


Fig. 6. The appearance of trade-off plots and trajectory analysis GUI. Analysis variables used to form the trade-off plots can be defined from trajectory data using valid MATLAB expressions and functions. The points are color-coded and shape-coded depending on whether they satisfy the specified requirements and pointwise-in-time constraints. The user can left click on individual points to access the Trajectory Plotting GUI, Trajectory Comparison GUI and Smoothing Utility GUI or right-click to get a summary information about the solution (shown).

environment, the user is provided with a mechanism to define different operating modes. For each mode, an objective function and constraints are defined to select the control input at time $t$, i.e., select $u^0(t,x,q) \in U_{ij}$, $j = 1, \cdots, K(i)$, $S_U(t) = i$. The objective functions and constraints can be functions of parameters and these parameters can be included into the vector $q$ for the subsequent optimization. For example, for the DISI engine case study [2], different modes may correspond to stratified, homogeneous and purge operation.

To complete the definition of the control policy, the user defines the mode transition logic. Specifically, the logic-based conditions that enable or disable mode transitions to specified destination modes based on states, control inputs, and operating variables of the powertrain are defined. These transition rules may depend on threshold parameters which can be included into the vector $q$ and subsequently optimized. Figure 7 illustrates a user interface for defining these parameter-dependent mode transition rules.

The parameter optimization of $q$ can be performed in a number of different ways including exhaustive parameter sweeps, designed experiments, search algorithms, etc. The approach provides the user with a flexible and powerful framework to optimize parameters in specified control policies and analyze the results while keeping the overall computational effort containable even for large simulation models. In fact, at this stage the use of parameter optimization by internal customers has exceeded that of the dynamic programming. Still as Figure 8 shows the dynamic programming based solution can be useful in delineating opportunities for improvement.

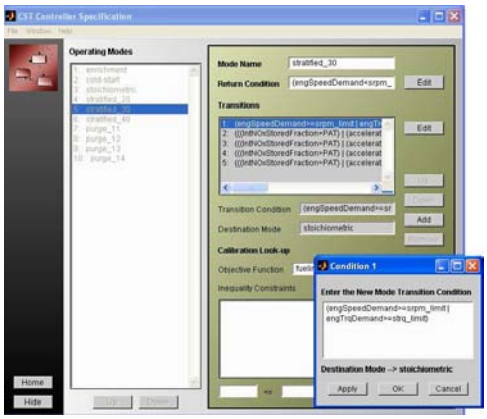


Fig. 7. User interface for defining mode transition logic.

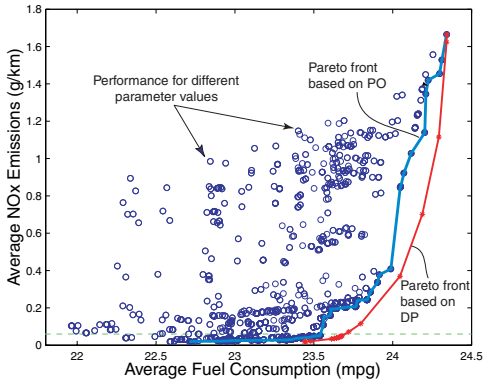


Fig. 8. Pareto fronts (Fuel consumption versus emissions) for DISI engine based on parameter sweeps for fixed structure policies (PO) and based on dynamic programming (DP). Parameters being swept are purge activation threshold and feedgas NOx emission index (see [3]).

## IV. Interface to MATLAB/Simulink Models

To correctly interface MATLAB models with the optimization software environment, the model must be accompanied by a model information file. The model information file identifies key variables and information important for the optimization such as states, inputs, outputs, parameters, units, variable ranges, default values, and constraints. The supported model types are the m-file (MATLAB function), the mdl-file (Simulink) and the mat-file (MATLAB mat files) models. The model information file is generated automatically based on the information the user enters into a model information file editor. Alternatively the model information file can be created manually. For the mdl-files and with the model information file editor open, the user can click directly on the outputs of Simulink blocks; then the model information editor record the path to the relevant variable into the model information file. The mat-files

TABLE I

EXECUTION TIMES IN SEC FOR S-FUNCTIONS AND MULTIPLE PARALLEL SESSIONS FOR A MODEL WITH NATIVE SIMULINK BLOCKS AND FOR AUTOMATICALLY GENERATED S-FUNCTION.

| Matlab sessions | Native Simulink Blocks | S-function |
|---|---|---|
| 1 | 1028 | - |
| 2 | 524 | 188 |
| 3 | 360 | 135 |

models are used in conjunction with the static optimization environment for generation of a finite set of control policies (2).

In order to run optimizations faster, the plant model can be implemented in the form of an s-function or in C (as a C-MEX file called from MATLAB wrapper). The software environment can generate s-function automatically from a model with native Simulink blocks (with certain restrictions) using Real Time Workshop. Table 1 illustrates the computational speed improvements with this approach.

## V. PARALLEL COMPUTATIONS

Parallel computations can reduce the time to complete weight sweeps and parameter sweeps for the dynamic programming-based optimization and for the parameter optimization. Additionally, parallel computing can be employed to faster build the output and state update maps. Either single processor computers connected over a network or a single multi-processor workstation can be used to implement this approach.

In the case of a multi-processor workstation, operating systems such as Windows 2000 automatically launch a new MATLAB session on a different processor if there is a MATLAB process running on one of the processors. So, multiple parameter sweeps can be completed faster using parallel sessions on a multi-processor machine as compared to a single processor machine. Tests were conducted on an eight processor Windows 2000 machine with 512MB of shared RAM. Standard demo example with settings for 5 parameter sweeps were run using 1, 2 and 3 MATLAB sessions on this machine. The results for an example with 5 parameter sweeps are illustrated in Table I.

## VI. EXAMPLE

We consider a brief example of an engine operating near idle (in neutral) for 2 sec. The engine speed is constrained between 800 rpm and 1200 rpm. At the end of the time interval the engine speed must be between 980 and 1020 rpm. The control inputs are the indicated torque command to the engine and spark retard. There is a rate limit on the indicated torque that the engine can develop, which is no more than 5 Nm within 0.01 sec. The rate limit is handled by introducing an auxiliary state $z(t)$ which stores the past value of the indicated torque; then the rate limit becomes a pointwise-in-time state/control constraint that the software environment can handle.

To achieve best fuel economy, it is clear that the engine speed must stay close to 800 rpm for as long as possible. On the other hand, for the maximum exhaust heat generation the engine speed must stay close to 1200 rpm for as long as possible. However, if we minimize a weighted sum of total fuel consumption and minus the exhaust heat the optimal speed trajectory can be oscillatory, see Figure 9.

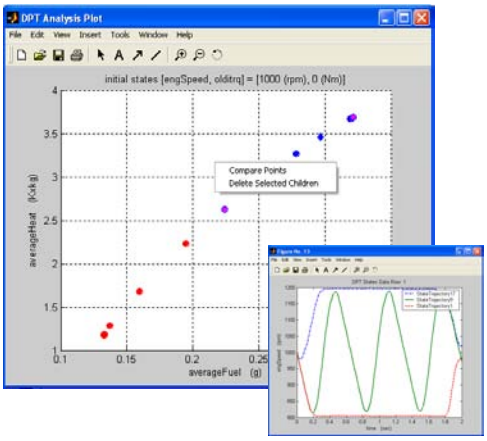


Fig. 9. Optimal speed control example with state constraints: Fuel consumption versus exhaust heat trade-off plot. The plot on the bottom shows engine speed trajectories corresponding to minimum fuel consumption, maximum exhaust heat generation and a minimum of weighted sum of fuel consumption and minus exhaust heat.

## VII. CONCLUSIONS

The paper discussed at a high level several computational and user interface solutions for implementing an optimal control based powertrain system assessment in a user-friendly software environment. A potential for future extensions exists and includes implementation of additional optimization algorithm and analysis options, ways to address variability and uncertainty in the inputs and model parameters, as well as automatic generation of parameter-dependent control policies from the optimal control solutions.

## VIII. ACKNOWLEDGEMENT

Alongkrit Chutinan and Gopalan Raghavachari of Emmeskay, Inc. have contributed to the development and implementation of the software environment discussed in this paper.

## REFERENCES

[1] D.P. Bertsekas. Dynamic Programming and Optimal Control. Athena Scientific. 2nd Edition. 2001.

[2] J.M. Kang, I. Kolmanovsky, and J.W. Grizzle. Dynamic optimization of lean burn engine aftertreatment. Journal of Dynamic Systems, Measurement, and Control, vol. 123, pp. 153-160, June 2001.

[3] I. Kolmanovsky, M. van Nieuwstadt, and J. Sun. Optimization of complex powertrain systems for fuel economy and emissions. Proceedings of the 1999 IEEE International Conference on Control Applications, vol. 1, pp. 833-839, Kohala Coast, Hawaii, 22-27 August 1999.