ON-LINE CONTROLLER TUNING BY MATLAB USING REAL SYSTEM RESPONSES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SEDA PEKTAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

NOVEMBER 2004

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Kemal İder
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Bülent E. Platin
Co-Supervisor

Prof. Dr. Tuna Balkan
Supervisor

Examining Committee Members

Prof. Dr. Y. Samim Ünlüsoy (METU, ME)

Prof. Dr. Tuna Balkan (METU, ME)

Prof. Dr. Bülent E. Platin (METU, ME)

Asst. Prof. Dr. İlhan Konukseven (METU, ME)

Y. Müh. Burak Gürcan (ASELSAN)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Seda Pektaş

ABSTRACT

ON-LINE CONTROLLER TUNING BY MATLAB® USING REAL SYSTEM RESPONSES

PEKTAŞ, Seda
M.Sc., Department of Mechanical Engineering
Supervisor: Prof. Dr. Tuna Balkan
Co-Supervisor: Prof. Dr. Bülent E. Platin
November 2004, 120 pages

This thesis attempts to tune any controller without the mathematical model knowledge of the system it is controlling. For that purpose, the optimization algorithm of MATLAB® 6.5 / Nonlinear Control Design Blockset (NCD) is adapted for real-time executions and combined with a hardware-in-the-loop simulation provided by MATLAB® 6.5 / Real-Time Windows Target (RTWT). A noise-included model of a DC motor position control system is obtained in MATLAB® / SIMULINK first and simulated to test the modified algorithm in some aspects. Then the presented methodology is verified using the physical plant (DC motor position control system) where tuning algorithm is driven mainly by the real system data and the required performance parameters specified by a user defined constraint window are successfully satisfied. Resultant improvements on the step response behavior of DC motor position control system are shown for two case studies.

Keywords: Controller tuning, Hardware-in-the-loop simulation, On-line tuning, Iterative feedback control

ÖZET

GERÇEK SİSTEM TEPKİLERİNİ KULLANARAK MATLAB® YARDIMIYLA GERÇEK ZAMANLI DENETLEÇ AYARLANMASI

PEKTAŞ, Seda
Yüksek Lisans, Makina Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Tuna Balkan
Ortak Tez Yöneticisi: Prof. Dr. Bülent E. Platin
Kasım 2004, 120 sayfa

Bu tez, esas olarak, matematik modeli bulunmayan bir sistemin denetim sisteminin ayarlanmasını amaçlamaktadır. Bu amaç için, MATLAB® 6.5 programının Nonlinear Control Design (NCD) biriminde var olan en iyileme algoritması gerçek zamanlı uygulamalara hazır hale getirilmiş ve diğer bir MATLAB® birimi olan Real Time Windows Target (RTWT) desteğiyle gerçek zamanlı yapılan benzetim tekniğiyle birlikte kullanılmıştır. Öncelikle gürültü içeren bir DC motor konum denetim sistemi modeli MATLAB® / SIMULINK yardımıyla hazırlanmış ve algoritmada yapılan modifikasyonlar belirli yönlerden test edilmiştir. Daha sonra, ayar algoritmasının kullanacağı verileri doğrudan alacağı fiziksel sistem (DC motor konum denetim sistemi) kullanılarak bahsi geçen metodun doğrulanması yapılmış ve kullanıcı tarafından tanımlanmış kısıtlamalarla belirlenen tasarım ölçütleri sağlanarak optimizasyon başarıyla sonuçlandırılmıştır. DC motor konum denetim sisteminin tepkilerindeki iyileşmeler iki ayrı durum çalışmasında gösterilmiştir.

Anahtar kelimeler: Denetim birimi ölçütlerinin ayarlanması, Gerçek zamanlı denetim, yinelemeli geribeslemeli denetim sistemleri.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor Prof. Dr. Tuna Balkan and co-supervisor Prof. Dr. Bülent E. Platin for their guidance, advice, criticism, encouragements and insight throughout the research.

A special thanks goes out to Prof. Dr. Samim Ünlüsoy, without whose crucial helps I would not have started my experimental work in the laboratory.

The friendship of Mr. Kamil Afacan and the technical assistance of Mr. İbrahim Sarı and Mr. Kerem Altun are gratefully acknowledged.

Many thanks to my patient and loving family, who have been a great source of strength all through this work. My mother deserves an award for her understanding during graduate school. I could not have done it without her.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background

This study involves two different branches of control engineering; controller tuning and hardware-in-the-loop simulation. A detailed explanation of all tuning methods is virtually impossible, because there are many tuning methods and many possible performance criteria. Also, field of hardware-in-the-loop simulation is rather vast but the method is straightforward. Only the milestones and main results of the previous work are presented in the following sections.

1.1.1 Controller Tuning

Controller parameters must be customized to a process or system to yield the best, or at least a minimally acceptable performance, called as the tuning a controller. To tune a controller, several critical factors must be taken into consideration. The stability of the system must always be assured over the entire operational conditions encountered. The smoothness of the response of the system to inputs or disturbances of varying magnitude must be maintained, such that there are no abrupt, disruptive or destructive changes to the system. There must be computational simplicity, so that controller computations are done quickly enough to send control signals to a real world system at an acceptable and efficient rate. Last of all, the controller must have the proper sensitivity, to be able to react to small control signals but resist and filter out the noise and disturbances [1]. However, for a variety of reasons optimal setting of the controller gains is difficult without a systematic procedure and as a result many tuning techniques were developed in the literature.

1.1.1.1 Standard Experimental PID Tuning Techniques

Despite huge advances in the field of control systems engineering, PID still remains the most common control algorithm in industrial use today. This is not only due to its simple structure, which is conceptually easy to understand making a manual tuning possible, but also to the fact that the algorithm provides an adequate performance in the vast majority of applications.
Transfer function of a PID controller is given as follows [2]:

$$G(s) = K_p(1 + \frac{1}{T_i s} + T_d s) = K_p + \frac{K_i}{s} + K_d s \tag{1.1}$$

In many practical control applications, a mathematical description of the plant is not available, and the controller has to be designed on the basis of measurements. This problem has attracted the attention of control engineers since the forties with the pioneering work by Ziegler and Nichols (1942), which focuses on the design of industrial PID controllers. After Ziegler and Nichols, many more techniques started to appear, partly as modifications and extensions of the Ziegler and Nichols method, partly as developments in new directions. Best known are the methods of Astrom and Hagglund, 1995 [3]; Chien, Hrones and Reswick, 1952 [4]; Dahlin, 1968 [5]; Haalman, 1965 [6]; McMillan, 1983 [7]. Here, however, only Ziegler-Nichols tuning rule (second method) will be introduced as an example to give a general idea on the basis of experimental PID tuning techniques.

Ziegler-Nichols tuning method [2] is straightforward. First, system is tested in closed loop with a proportional controller (integral and derivative modes are disconnected). The proportional controller gain is set to zero and increased until the system reaches its stability margin (oscillations). If there is no oscillation the set point is changed slightly in order to trigger any oscillation. The gain is adjusted so that the oscillation

is sustained, that is, continues at the same amplitude. If the magnitude of oscillations is increasing, gain is decreased slightly and vice versa. When oscillations with a constant amplitude and period are established, it is possible to determine the oscillations period (critical period) P_{cr} and controller (critical) gain K_{cr} with which oscillations where established. Based on experimentally obtained P_{cr} and K_{cr}, Ziegler and Nichols have given the following Table 1.1 for controller parameters (assuming quarter decay ratio criterion). One can use the set which corresponds with the desired configuration: P only, PI, or PID.

Table 1.1 Ziegler-Nichols optimal controller gains.

Controller Type	K_p	T_i	T_d
P	$0.5\ K_{cr}$	∞	0
PI	$0.45\ K_{cr}$	$0.833\ P_{cr}$	0
PID	$0.6\ K_{cr}$	$0.5\ P_{cr}$	$0.125\ P_{cr}$

The Ziegler–Nichols settings result in a very good disturbance response for integrating processes, but are otherwise known to result in rather aggressive settings, where oscillations and overshoot are usually not desired and also give poor performance for processes with a dominant delay.

The main characteristic of these techniques is that they were developed empirically through the simulation of a large number of process systems and provide simple tuning formulae to determine the PID controller parameters. However, since only a small amount of information on the dynamic behavior of the process is used, in many situations they do not provide good enough tuning or produce a satisfactory closed-loop response. The methods operate particularly well for simple systems and those which exhibit a clearly dominant pole-pair, but for more complex systems the PID gains may be strongly coupled in a less predictable way. For these systems, adequate performance is often only achieved through optimization based methods except manual and heuristic parameter variation.

1.1.1.2 Optimization Based Methods

1.1.1.2.1 Virtual Reference Feedback Tuning

M.C. Campi et. al [8] described a new controller tuning method called Virtual Reference Feedback Tuning (VRFT) for an unknown plant based on input/output measurements. This design method was direct (no model identification of the plant is needed) and can be applied using a single set of data generated by the plant. VRFT is a model reference control problem, where the user can specify his control objectives by a suitable selection of a reference model, M(s), i.e., desired transfer function of the closed-loop system. Such a reference is called "virtual" because it was not used to generate an output.

The basic idea of the virtual reference approach is to perform a wise selection of reference signal r(t) such that multiplication of reference signal and desired transfer function of the closed-loop system M(s) should be equal to measured system outputs y(t). After selecting reference signal the corresponding tracking error can be computed as e(t) = r(t) − y(t). Even though plant is not known, we know that when plant is fed by u(t) (actually measured input signal), it generates y(t) as an output. Therefore, a good controller is the one that generates u(t) when fed by e(t). Since both signals u(t) and e(t) are known, tuning task reduces to the identification problem of describing the dynamical relationship between e(t) and u(t). A controller parameter vector, θ, is selected such that it minimizes the following criterion:

$$J_{VR}^{N}(\theta) = \frac{1}{N}\sum_{t=1}^{N}(u_L(t) - C(s,\theta)e_L(t))^2 \tag{1.2}$$

where $C(s,\theta)$ represents the controller class.

However, in this procedure system is assumed to be noise-free. When the plant output y(t) is affected by an additive noise, it results in a significant deterioration of the performance. Also in general situations, testing the controller for stability is necessary before implementing the method.

1.1.1.2.2 PID Tuning Based on Genetic Algorithm

Genetic Algorithm (GA) is a stochastic global search method that mimics the process of natural evolution. The genetic algorithm starts with no knowledge of the correct solution and depends entirely on responses from its environment and evolution operators (i.e., reproduction, crossover and mutation) to arrive at the best solution. By starting at several independent points and searching in parallel, the algorithm avoids local minima and converging to sub-optimal solutions [9].

Three main stages of genetic algorithm are *reproduction, crossover* and *mutation*. During the reproduction phase the fitness value of each variable set (chromosome) is assessed. Just like in natural evolution, a fit chromosome has a higher probability of being selected for reproduction. Then, crossover operations swaps certain parts of the two selected strings in a bid to capture the good parts of old chromosomes and create better new ones. Finally by the introduction of a mutation operator, it is obtained enough diversity in the initial strings to ensure the GA searches the entire problem space. In literature, this technique is widely used for controller tuning, by defining the sets of controller gains as the chromosomes of Genetic Algorithm.

Genetic algorithms do not require derivative information or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the direction of the search. The main problem with the genetic algorithm, used to tune the controller online, is its computation time which is highly dependent on the speed of the hardware being used.

1.1.1.2.3 PID Tuning Based on Learning Action

In 2000, M.C. Best [10] introduced a formal approach to setting controller parameters, where the terms are adapted online to optimize a measure of system performance. The adaptation is conducted by a learning algorithm, using *Continuous Action Reinforcement Learning Automata* (CARLA). The control parameters are initially setted, then three separate learning automata are employed, one for each controller gain, to adaptively search the parameter space to minimize the specified cost criterion. Within each automata, each action has an associated probability density function $f(x)$ that is used as the basis for its selection. Action sets that produce an improvement in system performance invoke a high-performance "score", β, and thus through the learning sub-system have their probability of re-selection increased. This is achieved by modifying $f(x)$ through the use of a Gaussian neighborhood function centered on the successful action. The neighborhood function increases the probability of the original action, and also the probability of actions "close" to that selected; the assumption is that the performance surface over a range in each action is continuous and slowly varying. As the system learns, the probability distribution generally converges to a single Gaussian distribution around the desired parameter value.

M.C. Best made tests for engine idle-speed control, both in simulation and in practice. A Simulink hardware-in-the-loop system was designed, measuring engine speed and supplying a continuous control output to maintain idle at a constant rpm. PID parameters were set on-line via a MATLAB® program running the CARLA algorithm.

Their technique does not require a priori knowledge of the system dynamics, and it provides optimized control of complex nonlinear systems. One notable disadvantage of learning is its specificity to the individual test environment; plant variations can have significant implications for robustness.

1.1.2 Hardware-in-the-loop Simulations

The basic principle of hardware-in-the-loop simulation (HILS) is that some subsystems are physically embedded within a real-time simulation model. Real-time means the simulation of each component performed such that input and output signals show the same time dependent values as in real world dynamic operation. In HILS, the embedded system is fooled into thinking that it is operating with real-world inputs and outputs, in real-time. A computer software with real-time simulation capabilities and a computer with necessary communication abilities (A/D, D/A converters for communications with analog signals and digital ports for communication with digital signals) is necessary to perform hardware-in-the-loop simulation [11].

While performing HILS for a real system, control system hardware and software are usually the real system. The controlled process consisting of physical processes and sensors can then be either fully or partially simulated. Frequently, some actuators are real, and the process and sensors are simulated. The reason is that actuators and control hardware often form one integrated subsystem. Also, actuators are difficult to model precisely and to simulate in real-time. The use of real sensors together with the simulated process may require considerable realization efforts, because no real sensor input exists and it must be generated artificially.

1.2 Objective of the Study

The main goal of this thesis is to improve the controller tuning method of MATLAB® / Nonlinear Control Design (NCD) Blockset by doing a set of modifications on its optimization algorithm so that the algorithm will be applied to a hardware-in-the-loop simulation where the plant is real. Such a process will guarantee that the real system's output response will satisfy the required design specifications when the optimized controller parameter values are used.

NCD Blockset, MATLAB® is mainly used as user interface and its optimization algorithm is modified and adapted as being able to transfer input/output information

from/to a physical system which is provided by the usage of Real Time Windows Target, MATLAB®. Although it is not a perfect and final solution, it is a definite step toward reaching the most realistic results for the controller tuning process.

1.3 Scope of the Study

The method developed is intended to be used as a general real-time optimization tool whenever the model is unknown. Application area can be extended to any kind of optimization problem beside the controller parameters tuning. The systems that can be used with this algorithm are not limited to nonlinear, SISO, continuous time systems, also. A self-adapted controller tuning method against drastic set point changes is not aimed since this process would be too complicated with the lack of a mathematical model. One original feature of the method is that it is capable to use the physical plant instead of the mathematical model, and thus all the results are realistic. This feature does not exist in any of the optimization methods, best to our knowledge.

The thesis begins with an overview of NCD Blockset Version 1.1.6 of MATLAB® 6.5 describing the main idea and the working of the optimization algorithm behind, in Chapter 2. Related MATLAB® routines are explained in a logical order. Chapter 3 discusses the necessary modifications on the present NCD algorithm and the experimentation method to be able to use the real plant's inputs/outputs for the case studies. Additional details are provided related to the application of the algorithm before the example simulations are examined. As an example to the demonstration of the modified algorithm, a model-based simulation with a mathematical model of an inertia disc driven by a DC servo motor is made and the results are discussed in Chapter 4. This is followed, in Chapter 5, by hardware-in-the-loop simulation, which illustrates how well the algorithm works with the physical plant itself. Discussions, conclusions and future scope sections conclude the thesis report in Chapter 6. Additional details are provided in Appendix A, B and C about command window displays of case study II, flowchart of the algorithm and related original optimization m-files, respectively.

CHAPTER 2

MATLAB® NONLINEAR CONTROL DESIGN BLOCKSET

Nonlinear Control Design (NCD) Blockset of MATLAB® 6.5 Release 13 with Service Pack 1 is a tool that helps to tune design parameters in a nonlinear Simulink model by optimizing time-based signals to meet user-defined constraints by graphically placing constraints within a time-domain window.

To use the NCD Blockset, it only requires to include a special block, the NCD Outport block, in Simulink diagram and to connect that block to any signal in the model to signify that user wants to place some kind of constraint on the signal. NCD Outport block can be found under NCD within the Simulink Library Browser. Figure 2.1 shows an example usage of NCD Outport block in a Simulink model of the sample plant including a PID controller [12].

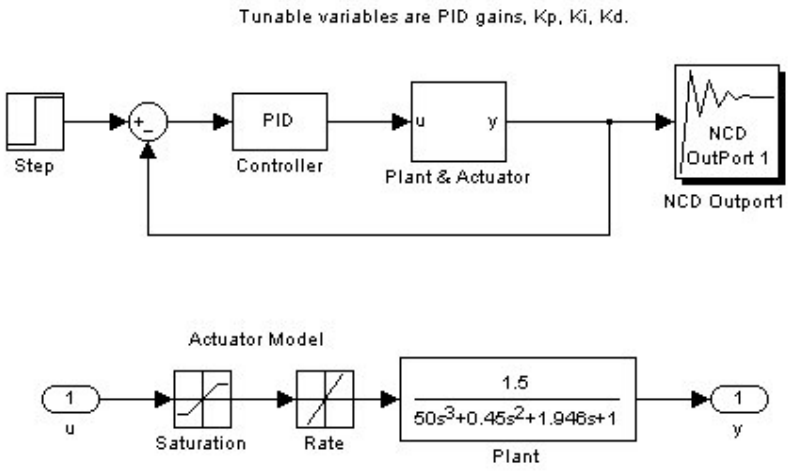


Figure 2.1 A Simulink model with NCD Outport Block

The NCD Blockset automatically converts time domain constraints into a constrained optimization problem and then solves the problem using state-of-the-art optimization routines taken from the Optimization Toolbox. The constrained optimization problem formulated by the NCD Blockset iteratively calls for simulations of the Simulink system, compares the results of the simulations with the constraint objectives, and uses gradient methods to adjust tunable parameters to better meet the objectives. The NCD Blockset allows to introduce uncertainty into plant dynamics, conduct Monte Carlo simulations, specify lower and upper limits on tunable parameters, and alter termination criterion. The progress of an optimization while the optimization is running can be followed from command window, and the final results are available in the MATLAB® workspace when an optimization is complete. Intermediate results are plotted after each simulation. It allows the user to terminate the optimization before it has completed, to retrieve the intermediate result or change the design.

2.1 Adjusting Constraints

NCD uses time-domain constraint bounds to represent lower and upper bounds on response signals, which appear as red bars in Figure 2.2. The lower and upper constraint bounds define a channel within which the ouput response should lie. NCD constraint window is opened by double-clicking on the NCD Outport block.

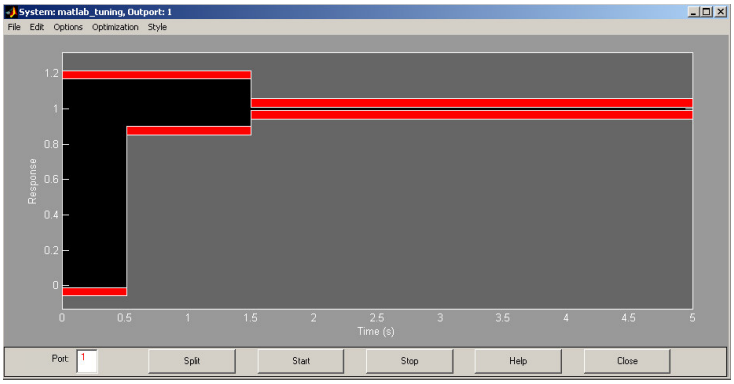


Figure 2.2 An example of NCD constraint window

These bounds must be changed to reflect the performance requirements proposed by the end user. To specify the desired output response range, it should be constrained by positioning (stretching, moving, splitting or opening) the constraint bound segments as shown in the Figure 2.3.

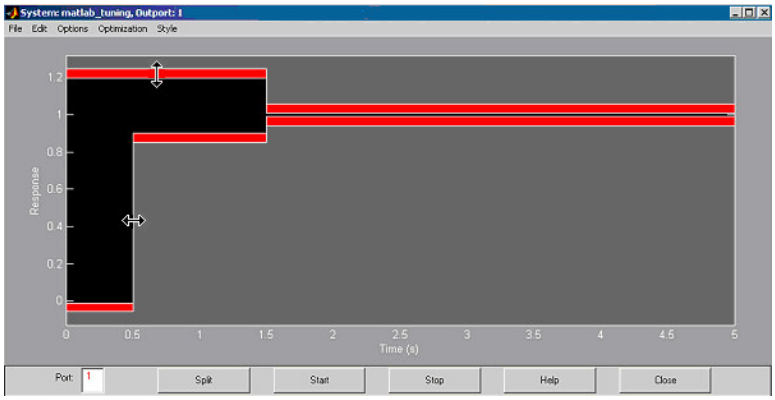


Figure 2.3. Adjusting output constraints

Alternatively, when optimizing the step response of the system, it is possible to specify the desired step response characteristics such as rise time, settling time, and overshoot by selecting **Step Response** from **Options** pane in the constraint window as shown in the Figure 2.4.

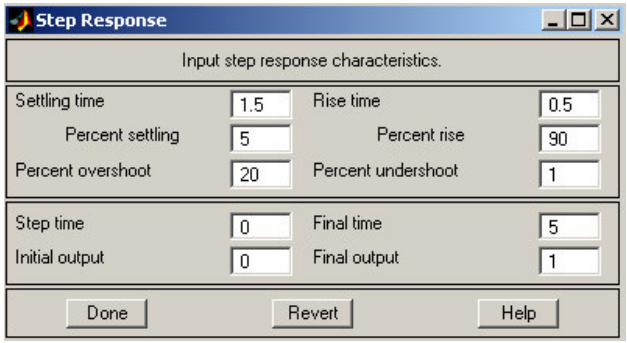


Figure 2.4. Example step response window

Three options specify the details of the step input:

- **Initial output**: Input level before the step occurs
- **Step time**: Time at which the step takes place
- **Final output**: Input level after the step occurs

The remaining options specify the characteristics of the response signal. Each of the step response characteristics is described in the Figure 2.5 below [12].

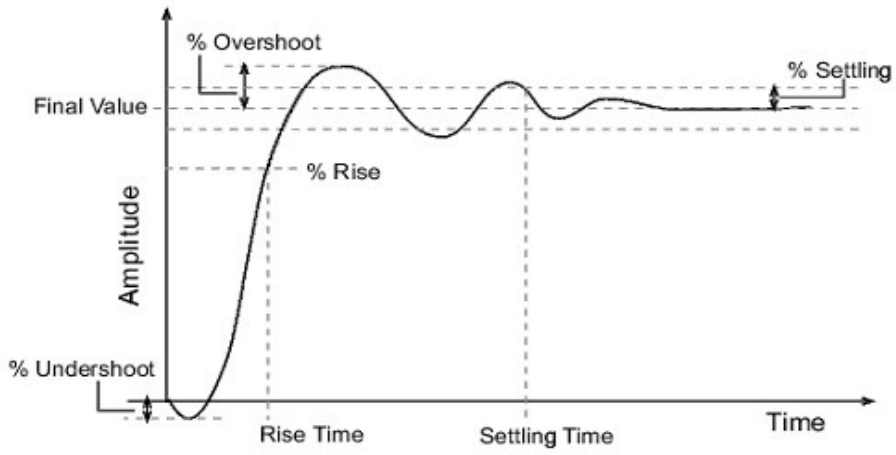


Figure 2.5. Step response characteristics

2.2 Specifying Tunable Variables and User Options

NCD attempts to reach the desired output response of the system by varying the user-defined parameters called as "tunable variables "[*]. Tunable parameters can be specified by the help of **Optimization Parameters** dialog box, by selecting **Parameters** from **Optimization** menu in constraint window and simply typing the name of the parameters into the **Tunable Variables** editable text field as shown in

[*] The name "**Tunable Variables**" is used by MATLAB®. Although the correct term should be "**Tunable Parameters**", we will use this name in the whole thesis to be compatible with MATLAB®.

Figure 2.6. If more than one tunable variable exists, variable names should be typed as separated by spaces.

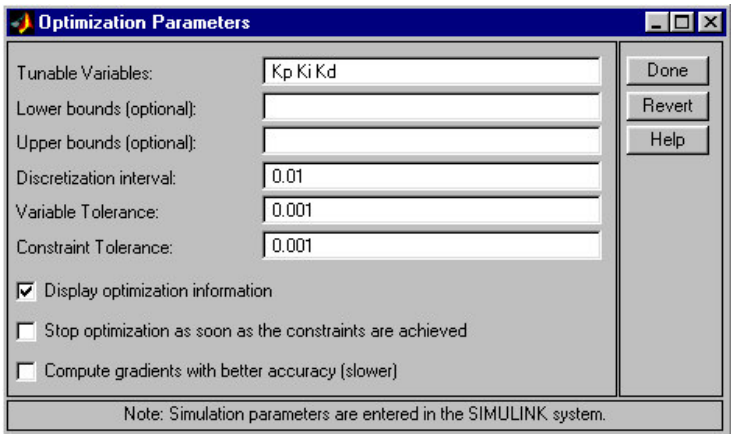


Figure 2.6. An optimization parameters window

User-defined **lower** and **upper bounds** limit the maximum and minimum values of tunable variables during the optimization process. **Variable** and **Constraint Tolerances** are the two terms related to termination criteria, which imply optimization will not terminate until all tunable variables (or constraints) converge to within the these values. One might also want to change the **Discretization Interval**. This number relates to the number of constraints generated by the optimization; the larger the discretization interval, the fewer constraints generated but the less rigorous the optimization. Typical discretization intervals range between one and two percent of the total simulation time. Normally NCD works with a variable step size chosen from simulation parameters in Simulink window. *In case a fixed step size is used for simulation, discretization interval should be equal to this fixed step size value*. By default, the optimization routine does not stop as soon as all the constraints are met, it tries to over achieve. **Stop optimization as soon as the constraints are achieved** check box prevents optimization process to go any further once the constraints are met. This is achieved, simply terminating the algorithm when the cost function is lower than zero. Also, one can specify the

optimization routine to use a separate routine for computing the gradients by enabling the **Compute gradients with better accuracy (slower)** check box. If this option is enabled, the gradient matrix of the constraints with respect to the tunable variables is computed by simulating the Simulink model with the original and perturbed values of the tunable variables, simultaneously. This procedure is slower, but may help the optimization in achieving the constraints for difficult problems.

2.3 Running the Optimization

After adjusting the constraint bounds in the constraint window and specifying the tuned parameters using the tunable parameters dialog box, NCD is ready to begin the optimization which can be started by clicking the **Start** button on the NCD Blockset Control panel or by selecting **Start** from the **Optimization** menu.

The NCD Constraint window, plots the responses at each iteration. Except the initial response plot, it overwrites the new plot over the previous one. So, the green line always shows the current, or final response while the white line shows the initial response. An example constraint figure window is shown in Figure 2.7 [12]. It can be seen that the final output response lies within the constraint bounds.

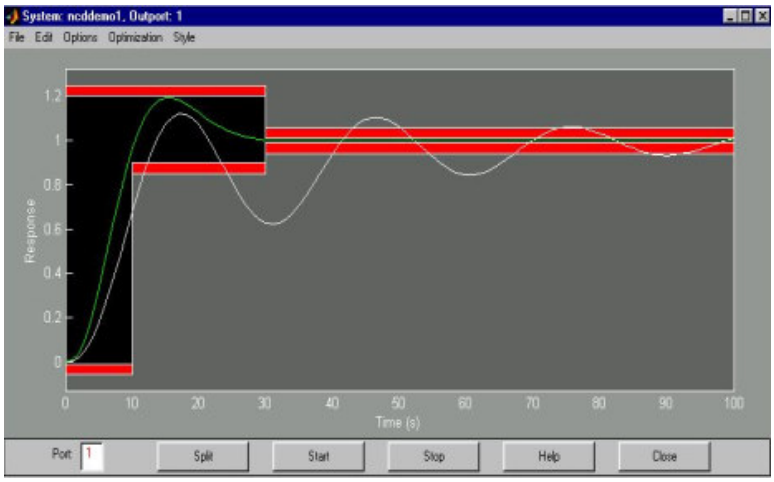


Figure 2.7. Initial and final output response plots

The result of each iteration appear in the command window shown in the following Figure 2.8 [12]. The new values of the tunable parameters appearing in the command window and is also changed in the MATLAB® workspace.

```
Processing uncertainty information.
Uncertainty turned off.
Setting up call to optimization routine.
Start time: 0 Stop time: 5.
There are 404 constraints to be met in each simulation.
There are 2 tunable variables.
There are 1 simulations per cost function call.
Creating Simulink model NCDmodel for gradients...Done
f-COUNT     MAX{g}         STEP  Procedures
    5     0.466256            1
   10    -0.421419            1   Hessian modified
   15    -0.392145            1
   20    -0.477709            1
   25    -0.478421            1   Hessian modified twice
   30    -0.473284            1   Hessian modified twice
   35    -0.473988            1   Hessian modified twice
   44    -0.473985       0.0625   Hessian modified twice
   51    -0.474014         0.25   Hessian modified
   60    -0.474371       0.0625   Hessian modified twice
   93    -0.474371   -3.73e-009   Hessian modified twice
  126    -0.474371   -3.73e-009   Hessian modified twice
  159    -0.474371   -3.73e-009   Hessian modified twice
  164    -0.473248            1   Hessian modified twice
  165    -0.477089            1   Hessian modified twice
Optimization Converged Successfully
Active Constraints:
    54
```

Figure 2.8. Sample command window display

During the optimization, the NCD Blockset first displays the information about plant uncertainty. Next, the blockset displays the information regarding the number of constraints per simulation and simulations conducted. To determine the total number of constraints to be met, one should multiply the constraints generated per

simulation by the number of simulation per cost function call. Then, information regarding the progress of the optimization follows.

The first column of output shows the total number of cost function calls. For one simulation per cost function call, this number gives the total number of simulations conducted. The second column (`max{g}`) shows the maximum (weighted) constraint violation. This number should tend to decrease during the optimization. When `max{g}` becomes negative, all constraints have been met. In the case above, a negative `max{g}` shows that all constraints were met after the ninth function call and the optimization then proceeded to overachieve. The third column `STEP`, displays the step size used by the line search algorithm. The last column shows special messages related to the quadratic programming sub problem. If the termination criteria are met, the optimization ends with the message `Optimization Converged Successfully`. Note that this does not imply that all constraints have been met.

Finally, the optimization displays an encoded list of the active constraints (i.e., which constraints prohibit further decrease in the cost function). The command window display can be disabled by unchecking the **Display optimization information** check box on the **Optimization Parameters** dialog box.

When the NCD Blockset begins the optimization, it plots the initial response in color white. To view the (initial) response without beginning the optimization, **Initial response** should be selected from **Options** menu. Viewing the initial response may help the user define better constraint bounds. At each iteration the optimization plots an intermediate response. Optimization can be terminated at any time and intermediate results can be recovered by clicking the **Stop** push button or selecting **Stop** from the **Optimization** menu.

The number of iterations necessary for the optimization to converge or terminate, will depend on the initial guess for the tuned parameters, the specific positioning of the constraints, and the optimization settings. In case the optimization does not converge, one might try a different initial guess or relax the constraints slightly.

2.4 Solving the Optimization Problem

NCD uses optimization algorithms to find parameter values that allow a feasible solution to the given constraints. NCD automatically converts the constraint bound data and tunable variable information into a constrained optimization problem.

Basically, the NCD Blockset attempts to minimize the maximum (weighted) constraint error. The NCD Blockset generates constraint errors at equally spaced time points (with spacing given by the **Discretization interval** defined in the **Optimization Parameters** dialog box) beginning at the simulation start time and ending at the simulation stop time. For upper bound constraints, it is defined the constraint error as the difference between the simulated output and the constraint boundary. For lower bound constraints, it is defined the constraint error as the difference between the constraint boundary and the simulated output.

When the optimization is started by the user pressing the **start** button, the Nonlinear Control Design Blockset invokes the routine `nlinopt`. `nlinopt` calls `simcnstr` function and it invokes the routine `nlconst` from the directory `~matlabR13root~\toolbox\simulink\simulink\private\nlconst.m`. Main calculations are done in `nlconst.m`. Necessary system output responses are obtained in `costfun.m` by the simulation of the model and then the related information are transferred into `nlconst`. The routine `nlconst` solves constrained optimization problems using a sequential quadratic programming (SQP) algorithm and quasi-Newton gradient search techniques.

The following pseudo code summarizes the optimization [12]:

% Begin nlinopt
% Process uncertain variable information (montevar)
% Expand constraint matrices (convertm)
% Initialize arguments for nlconst.m
% Begin nlconst

```
while ~(termination_criterion_met),
        for 1:Ntp, % Number of tunable parameters
        % Begin costfun
        % Calculate cost function (CostFunction)
        % Set tunable variables
            for 1:Npc, % Number of plants constrained
            % Assign plant uncertain variables
            % Call for simulation
            % Convert simulation time index
            % Draw necessary plots
            % Calculate constraints
            % Append constraints into vector, i.e., ConstraintError
            end % for Npc
         % End costfun
         % Tweak tunable variables in turn
         end % for Ntp
% Calculate gradient information
% Define search direction
% Perform line search
        % Begin costfun
        % Calculate cost function (CostFunction)
        % Set tunable variables
             for 1:Npc, % Number of plants constrained
             % Assign plant uncertain variables
             % Call for simulation
             % Convert simulation time index
            % Calculate constraints
             % Append constraints into vector, i.e., ConstraintError
             end % for Npc
        % End costfun
% Determine termination_criterion_met
end
```

% End nlconst

% End nlinopt

In the following sections, three vital functions of optimization process `nlinopt`, `costfun` and `nlconst` are explained in terms of their operations and interactions. They are explained mostly following the sequence of routines, but some parts such as loops related to plotting response graphs are not our concern and not discussed at all. Whole functions can be found in Appendix C. These sections have special importance for the future developments to understand, step by step, how the algorithm works. Also, a flowchart summarizing the whole process is given in Appendix B.

2.4.1 nlinopt.m

When the optimization is started by pressing the **start** button or when the initial response menu item is selected, NCD Blockset invokes the function `nlinopt`. Syntax definition of `nlinopt` function is

```
function nlinopt(sys,InitFlag)
```

Generation of the optimization problem involves mainly three steps:

1. Processing uncertainty data
2. Expanding the constraint matrices `ncdStruct.CnstrLB` and `ncdStruct.CnstrUB`.
3. Invoking the constrained optimization routine `nlconst`.

The NCD Blockset routine `montevar` processes uncertainty data input to the **Uncertain Variables** dialog box. It generates Monte Carlo plant data and performs certain error checks. The routine produces the Monte Carlo plants assuming a uniformly distributed probability density between the lower and upper bounds

entered into the **Uncertain Variables** dialog box. It will not be discussed in detail here, since uncertainty subject is out of scope of the thesis.

If `InitFlag` is 1, `nlinopt` gives a message "`Beginning simulations for initial response plots`" and plots the initial response by calling `initresp` routine. Although its algorithm seems very similar to that of `costfun`, the logic is different, since `initresp` does not deal with tunable variables, upper and lower bound data. So, it calculates neither the cost function nor constraint error. In the command window, it appears "`Done plotting the initial response`" and algorithm stops. If Initflag is zero, `nlinopt` gives a message "`Setting up call to optimization routine`", it also calls `initresp` routine, and starts the optimization process. As a first step, `nlinopt` vectorizes the tunable variables as `tvarvec` and also upper and lower bounds of tunable variables as `tvubvec` and `tvlbvec` in the same way.

The constraint bounds displayed in the NCD Blockset constraint window are for visualization purposes only. Two matrices, `ncdStruct.CnstrLB` and `ncdStruct.CnstrUB`, contain all the constraint information. The NCD Blockset routine `convertm.m` expands the constraint matrices, `ncdStruct.CnstrLB` and `ncdStruct.CnstrUB` using the discretization interval `Td` and converts them to `Ml` and `Mu`. Generally speaking, constraints are generated at an interval of `Td`, per constraint segment per constrained signal.

The matrix `ncdStruct.CnstrLB (ncdStruct.CnstrUB)` has the dimension 4 x 2L where L is the total number of line segments in all lower (upper) bounds. The first row of `ncdStruct.CnstrLB` and `ncdStruct.CnstrUB` contains the outport number for the constraint. All constraint bound segments for the same outport are grouped together. The second row contains the time axis values of the segment while the third row contains the response axis values. The time axis values for each output increase monotonically from optimization start time to optimization stop time. The time value end of one segment equals the time value beginning of the next segment. The fourth row of `ncdStruct.CnstrLB` (`ncdStruct.CnstrUB`) contains

information about the segment's weighting. As an example, consider the lower bound constraint matrix,

$$ncdStruct.CnstrLB = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 \\ 0 & 10 & 10 & 30 & 30 & 100 & 0 & 100 \\ 0 & 0 & 0.9 & 0.9 & 0.99 & 0.99 & -0.1 & -0.1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad (2.1)$$

First row of the matrix shows that two outputs are constrained. The first output is constrained by three line segments and the second by one line segment. Constraints on the second output are defined by the line segment from the (time, response) point (0,-0.1) to the point (100,-0.1). Constraints on the first output are defined by the line segments from (0,0) to (10,0), from (10,0.9) to (30,0.9), and from (30,0.99) to (100,0.99). Here, it is expected a simulation start time of zero and stop time of 100. The fourth row shows that all line segments are weighted equally, with weight of one.

Necessary command window displays are done in `nlinopt` before starting the optimization calculations, such as start time, stop time, number of constraints to be met in each simulation, number of tunable variables and number of simulations per cost function call. Also, some options related to optimization (variable tolerance, constraint tolerance, etc.) are assigned to values entered by the user from **Optimization Parameters** dialog box.

Finally, `nlinopt` invokes the helper function `simcnstr` by the following command:

```
x = simcnstr('ncdtoolbox','costfun',tvarvec,options,tvlbvec,
tvubvec,'',tvarmtx,tvarext,sys,timepts,Mu,Ml,offset,sims,
uvarmtx,uvarext,uvdata,SimOptions)
```

This helper function converts string `'costfun'` into an inline function `FUNfcn` and calls constrained optimization routine `nlconst`.

Routine `nlinopt` increases the number of tunable variables vector by 1, adding the cost function into the vector of tunable variables as well, while passing this vector to `nlconst`. Specifically, if `Kp Ki Kd` are entered as tunable variables into the **Optimization Parameters** dialog box, `nlinopt` passes the tunable variables vector `tvarvec` as `x = [Kp; Ki; Kd; gamma]` to `nlconst`, where gamma initially assigned to 1. By calling `nlconst` with a special option flag, it expects `gamma` to contain the value of the cost function. For the cost calculation `nlconst` invokes `costfun`. Here, first `costfun` will be explained and then `nlconst` will be discussed extendedly.

2.4.2 costfun.m

The NCD Blockset routine `costfun` inspects the output response of the system and returns the cost function and constraint errors as output. Syntax definition of `costfun` function is

```
function [CostFunction,ConstraintError] =
costfun(tvarvec,tvarmtx,tvarext,sysname,timepts,Mu,Ml,offset,
sims,uvarmtx,uvarext,uvdata,simoptions)
```

At the beginning of the routine, `costfun` recovers tunable variables from `tvarvec` (vector `x` coming from `nlconst`) and assign them to the appropriate tunable variables in the base workspace. Hereby, changed values of tunable variable in `nlconst` are transferred into base workspace before the simulation process.

Routine `costfun` calculates the cost function as follows:

```
CostFunction = tvarvec(end)
```

i.e., `gamma` which corresponds to the (weighted) maximum constraint violation. The routine then initializes the constraint vector to the empty matrix.

Next, it initiates a `for` loop according to the number of plants constrained, **Npc**. Specifically, **Npc** will be directly equal to 1, if any uncertainty and Monte Carlo simulations are not defined by the user. Within `for` loop, `costfun` calls for a simulation of system's Simulink model, by `sim` command as follows:

```
[SimTime, SimState, InterpOut]=sim(''' sysname ''', timepts,
simoptions)
```

where `InterpOut` is the simulation output linearly interpolated to the time basis `Tstart:Td:Tstop`. Depending on `OPT_STEP` value, `costfun` updates the plots in NCD Blockset constraint figures window and forces the updated plot to flush on the screen executing a `drawnow` command. Finally, at each pass through the `for` loop, it augments the constraint vector, `ConstraintError`, as

```
ConstraintError = [ConstraintError; ...
InterpOut(Mu(:,1)) - Mu(:,2) - Mu(:,3)*CostFunction; ...
Ml(:,2) - InterpOut(Mu(:,1)) - Ml(:,3)*CostFunction]
```

where upper and lower bound constraints, `Mu` and `Ml`, have three columns as follows:

`Ml` = [InterpOut_Index Constraint Bound Weight] (2.2)

`Mu` = [InterpOut_Index Constraint Bound Weight] (2.3)

2.4.3 nlconst.m

By the function `nlconst`, NCD Blockset transforms the constraint errors and simulated system output into an optimization problem of the form:

$$\min_{x,\gamma} \gamma \quad s.t. \quad \begin{cases} g(x) - w\gamma \le 0 \\ x_l \le x \le x_u \end{cases} \tag{2.4}$$

where variable **x** is a vectorization of the tunable variables while $\mathbf{x_l}$ and $\mathbf{x_u}$ are vectorizations of the lower and upper bounds on the tunable variables. The vector **g(x)** is a vectorization of the constraint bound error (absolute difference between output and constraint boundary) and **w** is a vectorization of weightings on the constraints. The scalar γ imposes an element of slackness (i.e., cost function calculated in `costfun` routine) into the problem, which otherwise imposes that the goals be rigidly met . Here, the term $g(x) - w\gamma$ implies the constraint error calculated in `costfun` routine.

Syntax definition of `nlconst` function is

```
function [x,OPTIONS,lambda,HESS] =
nlconst(FUNfcn,x,OPTIONS,VLB,VUB,GRADfcn,varargin)
```

Before starting the main loop, `nlconst` initializes necessary parameters and does some preparatory work. It defines `nvars` as the number of tunable variables and initializes `Hessian` and `CHG` for the first `costfun` call as follows:

```
HESS = eye(nvars,nvars)
CHG = 1e-7*abs(x)+1e-7*ones(nvars,1)
```

Then algorithm checks the upper and lower bounds on tunable variables which are entered by the user in **Optimization Parameters** window. In case lower bound is entered a value greater than upper bound, an immediate error message appears in command window as "`Bounds Infeasible`". If initial value of any tunable variable is lower than the lower bound, that variable is assigned to lower bound value. Reversely, if there exist any tunable variable exceeding the upper bound, then this variable is assigned to upper bound and direction of `CHG` is reversed with a sign

change. Upper and lower bounds are optional, so tunable variables remain same if there exist no bounds defined by the user.

After passing tunable variables with upper and lower bound filter, it requires to calculate cost function (`f`) and constraint error (`g`) by calling `costfun` routine as shown:

```
[f,g] = feval(FUNfcn{1},x,varargin{:})
```

Before starting iterations `nlconst` initializes number of function evaluations `OPTIONS(10)`, number of function gradient evaluation `OPTIONS(11)` and step length `OPTIONS(18)` to 1. Also, maximum number of function evaluations `OPTIONS(14)`, is defined as 100 times of `nvars`. Zero or missing values of `OPTIONS` vector are replaced with default parameters used by the optimization routines by `foptions` command. Descriptions and default values of related options are shown in Table 2.1. Note that if model-specific information is known (more sensible tolerances, minimum change in variable for finite difference gradients, etc.), then such information should always be used, since it may help to solve the model far more efficiently than by directly using defaults.

Table 2.1 Descriptions of optimization options

`OPTIONS(1)`	Display parameter. (Default:0). 1 displays some results.
`OPTIONS(2)`	Termination tolerance for X. (Default: 1e-4).
`OPTIONS(3)`	Termination tolerance on F. (Default: 1e-4).
`OPTIONS(4)`	Termination criterion on constraint violation. (Default: 1e-6)
`OPTIONS(5)`	Algorithm: Strategy: Not always used.
`OPTIONS(6)`	Algorithm: Optimizer: Not always used.
`OPTIONS(7)`	Algorithm: Line search algorithm. (Default 0)
`OPTIONS(8)`	Function value. (Lambda in goal attainment)
`OPTIONS(9)`	User-supplied gradients (1, for user-supplied gradients).

OPTIONS(10)	Number of function and constraint Evaluations.
OPTIONS(11)	Number of function gradient evaluations.
OPTIONS(12)	Number of constraint evaluations.
OPTIONS(13)	Number of equality constraints.
OPTIONS(14)	Maximum number of function evaluations.
OPTIONS(15)	Used in goal attainment for special objectives.
OPTIONS(16)	Minimum change in variables for finite difference gradients.
OPTIONS(17)	Maximum change in variable for finite difference gradients.
OPTIONS(18)	Step length. (Default 1 or less).

For using in the main loop, an initial GNEW value is defined by knowing CHG as follows:

```
GNEW = 1e8*CHG
```

Main loop is a while loop containing all the optimization processes. The state of loop is defined by a Boolean, status, which is assigned to 1, when the termination criteria are satisfied or maximum number of iteration is exceeded. For a better explanation, whole loop is divided into subsections according to their operational sequence.

2.4.3.1 Finite Difference Gradient Calculation

The idea is to obtain the first order gradients of cost function and constraint error by varying the tunable variables with a small CHG vector, which can be defined as

```
CHG = -1e-8/(GNEW+eps)
```

where GNEW is initially defined and will be obtained during the calculation of search direction. If this CHG is smaller than a minimum value specified as OPTIONS(16), or greater than a maximum value as OPTIONS(17), then CHG value is assigned to either

OPTIONS(16) or OPTIONS(17), respectively. If CHG is in between limits it will remain same.

```
CHG=sign(CHG+eps).*min(max(abs(CHG),OPTIONS(16)),OPTIONS(17))
```

The nlconst routine perturbs each tunable variable including cost function, as the amount of CHG (remaining the other variables unchanged) and evaluates the resulting cost function value (f) and constraint errors vector (g) by calling the function costfun.m for each time:

```
temp = x(i)

x(i)= temp + CHG(i)

[f,g] = feval (costfun, x, varargin{:})
```

Gradients of cost function and constraint error for each variable are calculated by the following equations:

```
gf(i,1) = (f-oldf)/CHG(i)

gg(i,:) = (g - oldg)'/CHG(i)

x(i) = temp
```

Here, it is very important to assign f and g to their values before the change application (oldf and oldg). There is no need to do an OLDX assignment on tunable variable vector, x, since all the tunable variables return to their original values at the end of gradient calculation by the usage of temp. These tentative changes on tunable variables should not be confused by the major steps in tunable variables, which will mainly occur in line search section.

2.4.3.2 Finding Search Direction

To find the search direction it is necessary to use a second order gradient, i.e., the Hessian, belonging to that iteration. For the first call, Hessian is not calculated. By using `gf` found in finite gradient calculation section and `OLDgf`, which is the gradient belonging to the previous iteration, `GNEW`, `GOLD` are calculated as follows:

```
GNEW=gf+AN'*NEWLAMBDA
GOLD=OLDgf+OLDAN'*LAMBDA
```

where `AN` is the transpose of `gg` and `LAMBDA` is one of the outputs of quadratic programming route, `qpsub.m`, which will not be discussed in detail here. Then `YL` is defined as

```
YL=GNEW-GOLD
```

With the same procedure `sdiff` can be found as the difference between `x` and `OLDX` as follows:

```
sdiff=XOUT-OLDX
```

Before finding the Hessian, algorithm should check its positive definiteness. A "`how`" variable is used to define the status of the Hessian, which is declared in the last column of the display output (the column labeled `Procedures`). Generally no display appears in the column meaning the Hessian is positive definite. For non-positive definite Hessians, two successive modifications can be performed to make the Hessian positive definite. If the first modification succeeds, the message `Hessian modified` appears in the `Procedures` column. The second modification always results in a positive definite Hessian and displays `Hessian modified twice` in the `Procedures` column. Often such messages imply that the optimization is far from a solution or that the problem is particularly sensitive to variations in some of the tunable parameters. Hessian is calculated as follows:

```
HESS=HESS+(YL*YL')/(YL'*sdiff)-(HESS*sdiff*sdiff'*HESS')/
(sdiff'*HESS*sdiff)
```

Before finding the search direction (SD), present f, g, gf and x values are stored as OLDF, OLDG, OLDgf and OLDX respectively, to use them in the next iteration. Then SD is found by using qpsub routine as follows:

```
[SD,lambda,howqp] =qpsub(HESS,gf,AN,-GT,[],[],XN,OPTIONS(13),
-1,'nlconst',size(AN,1),nvars,0,1)
```

The implementation of the Sequential Quadratic Programming (SQP) subproblem attempts to satisfy the Kuhn-Tucker equations, which are necessary conditions for optimality of a constrained optimization problem.

This section ends with command window displays.

```
disp([sprintf('%5.0f %12.6g ',OPTIONS(10),gamma),
sprintf('%12.3g  ',OPTIONS(18)),how, ' ',howqp]);
```

where OPTIONS(10) shows the total number of cost function calls (generally equals to number of simulation) and gamma implies the maximum constraint violation. In fact gamma contains both cost function and constraint error information as shown:

```
gamma = mg+f
```

where

```
mg=max(ga)
```

ga is the ordered version of constraint error (g) including the absolute of equality constraint errors, whose number is shown by OPTIONS(13) in the following equation:

```
ga=[abs(g((1:OPTIONS(13))'));g( (OPTIONS(13)+1:ncstr)')]
```

2.4.3.3 Line Search

After determining a search direction, `nlconst` performs a line search along the search direction in an attempt to simultaneously minimize the cost while satisfying constraint equations. Tunable variables are stored as `MATX` before starting the line search application and then the line search is performed using two merit functions. The line search determines a step length, `OPTIONS(18)`, ($0 <$`OPTIONS(18)` ≤ 1), such that the new set of tunable variables

```
x = MATX + OPTIONS(18)*SD
```

gives a sufficient decrease in merit functions.

For that purpose, a while loop is used to find new `x`, `f` , `g` and `mg` values by calling `costfun` routine and to check the merit functions, `MERIT` and `MERIT2` based on these new values. Here, `MERIT` and `MERIT2` are two criteria referring to cost function (`f`) and maximum constraint violation (`gamma`), respectively. Starting from "1", `OPTIONS(18)` is halved until `MERIT` and `MERIT2` are equal to or smaller than `MATL` and `MATL2`, which are merit functions corresponding to tunable variables set before the line search. Clearly, if line search brings up an improvement on cost function and constraint error (finding smaller values of them), it will go out from while loop at the first iteration with a step length equal to 1. An opposite situation implies a worse case, that means either optimization is in the wrong way or it is in the right way but has a larger step length than the necessary. In such a case, algorithm needs to resize the step length value. So, line search plays a very important role shaping the "destiny" of optimization process and it will be revised for the real-time applications in Chapter 3.

2.4.1.4 Finished Line Search

In the finished line search section, optimization is checked according to termination criteria. Four important values are considered in termination criteria. First of them is the maximum element of absolute search direction vector (`max(abs(SD))`. When the

maximum of SD is smaller than a limit value defined as two times of OPTIONS(2), which is entered by the user in **Variable Tolerance** field of **Optimization Parameters** window, this means tunable variables are changing very slowly. The other criteria is the absolute of multiplication of gf and SD vectors (abs(gf'*SD)), i.e., change in cost function. This implies that cost function changes very slowly for a smaller value than two times of OPTIONS(3), which is again entered by the user. Until now, parameters belonging to new tunable variables are not considered since gf and SD are calculated before the setting of new tunable variables. It is also important to check some parameters directly related to new set of tunable variables, such as maximum constraint violation, mg. A limit value for mg is defined as OPTIONS(4), **Constraint Tolerance** in **Optimization Parameters** window. Basically, a smaller mg means constraint equations are satisfied. Final stopping criterion is related to existence of an "infeasible solution" case while maximum constraint violation is greater than zero.

If the first three criteria are satisfied and the forth one is not satisfied during the algorithm, then command window displays the information related to the newly found tunable variables as the same procedure described before, with a message "Optimization Converged Successfully" and iterations stop. Alternatively if the first two criteria and, at the same time, the fourth one are satisfied, again algorithm stops, but this time a warning message "No feasible solution found" is displayed in the command window. Otherwise, algorithm checks the total number of iterations, if it exceeds the maximum permitted number of cost function call, OPTIONS(14), a message "Maximum number of function evaluations exceeded; increase OPTIONS(14)" appears. This times, x and f are returned to their values before line search, i.e., MATX and OLDF then the algorithm stops. In case the termination criteria are not satisfied by any of the above conditions and maximum number of cost function call is not exceeded, algorithm turns to the beginning of the while loop and starts the next iteration.

CHAPTER 3

MODIFICATIONS ON NCD BLOCKSET ALGORITHM

For the real-time hardware-in-the-loop simulation, a model as shown in Figure 3.1 should be prepared in Simulink **external** mode with Real-Time Windows Target (RTWT) Toolbox of MATLAB®. Although any type of controller can be used, in this model a PID controller is chosen as an example. The Real-Time Windows Target I/O blocks, Analog Input (A/D) and Analog Output (D/A), allow us to select and connect specific analog channels to our Simulink model through an I/O data acquisition board. In other words, they provide an interface to our physical I/O boards and our real-time application.

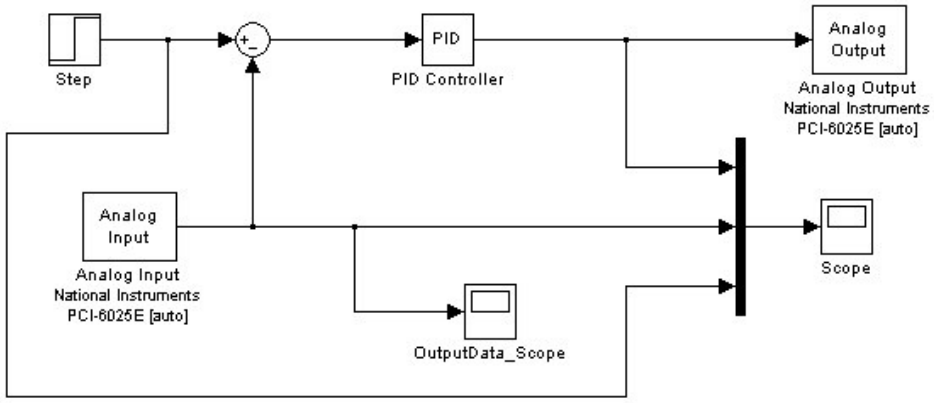


Figure 3.1 Simulink model for real-time application

Unfortunately, an attempt to use existing algorithm by attaching NCD Outport Block directly to plant output (Analog Input block) will be useless since it is not able to perform command line simulation of a model in external mode. (Still, NCD

Outport block will be attached into the above model later, just to create constraint figure window by double clicking on it). To achieve our goal, some short but vital interferences should be made on the algorithm.

Output response data created during real-time execution is saved to the base workspace through a Simulink scope block. Easiest way for saving a variable called as `OutputData` in the base workspace is to select **save data to workspace** check box in **Data History** menu of **'OutputData_Scope' parameters** and enter "OutputData" into **Variable name** text field as shown in Figure 5.3. **Array** is the most usable format type for this application. Default **Limit data points to last** property should be disabled.

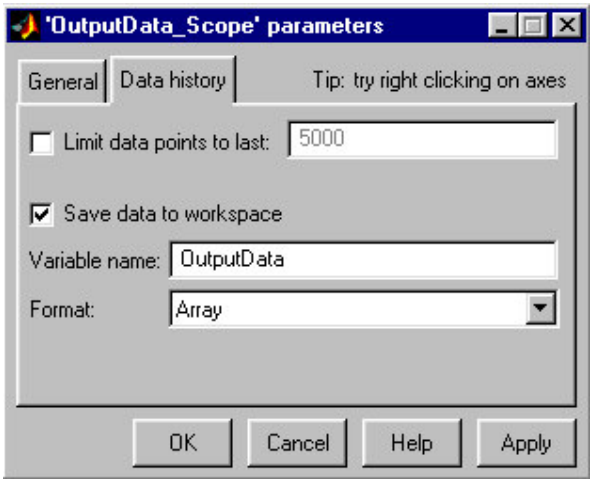


Figure 3.2 Saving output into workspace by using scope

Also, tunable variables changing in function local workspace during the optimization process need to be saved into base workspace before starting the real-time execution. However, this does not require any extra effort or modification on the algorithm since originally `costfun` routine fulfils this requirement as explained before.

3.1 Modifications on `costfun.m` Function

Some modifications are introduced necessary to feed the real plant's output data into `costfun` algorithm. This logic should be adapted and applied into `Initresp.m` routine to obtain the initial response plot in the same way.

3.1.1 Defining Output Response Data as Global Variable

Real plant's output response is obtained by the real-time execution and saved into the base workspace as described before. Ordinarily, each MATLAB® function, has its own local variables, which are separate from those of other functions, and from those of the base workspace. To introduce `OutputData` in `costfun` routine and to satisfy sharing a single copy of that variable, it should be declared as `global` both in `costfun` routine and in the base workspace.

Related part of `costfun` routine showing the modification in the line between the lines containing stars, appears as follows:

```
(….)
global OPT_STOP;
global OPT_STEP;
global ncdStruct;
%*************************
global OutputData
%*************************
atindx = 1;
for i=1:size(tvarmtx,1)
    siz = [atindx:tvarext(i,1)]';
    assignin('base','NCD_tmp',tvarvec(siz,1));
(….)
```

Also, a masked subsystem `ncd1init` is used to declare `OutputData` as global variable in the base workspace beside initializing the tunable variables. By double clicking on the subsystem box, it executes script file `ncd1init` and variables

created using scripts are considered to be in the base workspace. Initialization of tunable variables and global declaration of `OutputData` variable in base workspace should be done as a first step at the beginning of the optimization process. For an initial set of variables as Kp=1 Kd=1 Kd=1, `ncd1init` script appears as follows:

```
global OutputData
Kp=1;
Ki=1;
Kd=1;
```

There is a certain amount of risk associated with using global variables. One might unknowingly give a global variable in one function a name that is already used for a global variable in another function and may unintentionally overwrite the variables. Because of this and the difficulty to change the global variable name, it is recommended to use them sparingly. Alternatively, one can save the output response data as a mat-file and load this file into local workspace of `costfun` instead of using a global variable. To save output response, `OutputData`, as a mat-file, **enable archiving** property is activated in **External Data Archiving** window from **external mode control panel** of RTWT simulation window, as shown in Figure 3.3.

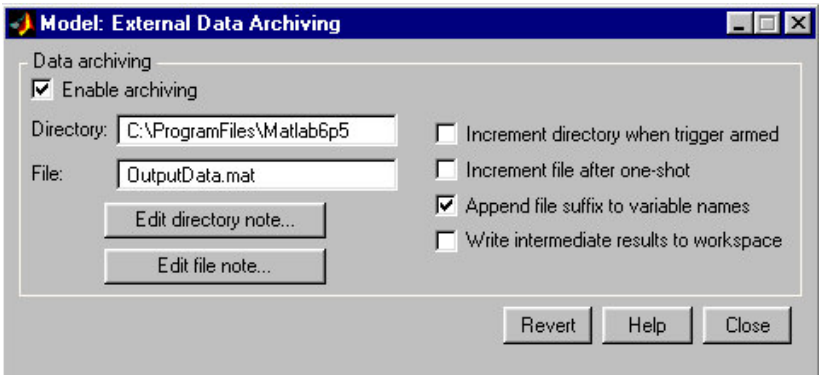


Figure 3.3. External Data Archiving window

To load this data from mat-file to function local workspace, following line should be added in `costfun` routine in place of command line simulation:

```
load('OutputData.mat')
```

3.1.2 Interrupting Algorithm to Run the Real-Time Simulation

To feed the output response data into the `costfun` routine externally, first it is necessary to cancel out the simulation process done by `sim` command inside the routine. Then it is added a `keyboard` statement, which stops m-file execution at the point where it appears and allows us input from keyboard to start the physical plant testing. Another advantage is when the program is in `keyboard` mode, local workspaces of each function can be examined by using the **Stack** field in the workspace browser. This mode is indicated by a special prompt as follows:

```
K>>
```

One can resume `costfun` execution by typing “return” in command window and pressing the **Return** key.

To let the user to track the values of tunable variables and gamma from command window at each `costfun` call, an extra line is added before `keyboard` statement by the help of `fprintf` command. These modifications will be combined and shown in the next section.

3.1.3 Converting Output Response Data into Suitable Name and Size

In scope parameters window by choosing the `array` format, real system’s output response, `OutputData`, is saved into base workspace as ((Tstop-Tstart)/Td+1) x 2 size matrix where first column has time data and second column has respective output data. However `costfun` routine requires a vector variable called as

`InterpOut`, containing output response data only. So, an `OutputData` to `InterpOut` conversion is compulsory to use output response information properly.

Related part of `costfun` routine including the modifications explained in section 3.1.2 and 3.1.3 (in the lines between the lines containing stars) is as follows:

```
(….)
atindx = 1;
   for i=1:size(uvarmtx,1)
      siz = [atindx:uvarext(i,1)]';
      assignin('base','NCDtmp',uvdata(siz,simindx));
      evalin('base',[uvarmtx(i,:) '(:) = NCDtmp;']);
      atindx = uvarext(i,1)+1;
   end
%************************************************************
%SimString=['sim('''sysname''',timepts,simoptions);']  CANCELLED
%lasterr('');                                          CANCELLED
%eval(['[SimTime,SimState,InterpOut]='SimString],' ')  CANCELLED
fprintf('tvarmtx(1,:)=%4.12f  tvarmtx(2,:)=%14.12f
tvarmtx(3,:)=%14.12f',tvarvec(1,1),tvarvec(2,1),tvarvec(3,1));
fprintf('\n');
keyboard;
InterpOut=OutputData(:,2);
%************************************************************
if ~isempty(lasterr),
      fprintf('\n      SL Error Message: %s\n     ',lasterr');
      fprintf('\n      COSTFUN: Error simulating %s',sysname);
(….)
```

3.1.4 Updating the Intermediate Response Plots

Originally, NCD algorithm updates the output response plots in the constraint window once for each major step, specifically after first tunable variable has been changed by a CHG value. This action can be seen with the following “if” statement of `costfun`:

```
(….)
if (ncdStruct.GradFlag == 0) & (OPT_STEP == 1),
      fighndls = allchild(0);
      fignames = char(get(fighndls,'Name'));
      prefix = ['System: ' sysname ', Outport: '];
      lnprefix = length(prefix)+1;
      fighndls = fighndls(strmatch(prefix,fignames));
      fignames = char(get(fighndls,'Name'));
      for figindx=1:length(fighndls)
           portnum = str2num(fignames(figindx,lnprefix:end));
           axs = get(fighndls(figindx),'CurrentAxes');
           MCSlns = get(axs,'UserData');
               if (~isempty(MCSlns))
              ln = MCSlns(max(sims)+simindx);
              set(ln,'YData',InterpOut(:,portnum));
              if (strcmp(get(ln,'Visible'),'off'))
                 set(ln,'Visible','on');
              end
          end
      end
 end
(….)
```

In the routines `nlconst` and `costfun`, `OPT_STEP` is defined as a global variable having a value either 0, 1 or 2 for different stages of optimization process. It is assigned to "1", for the `costfun` call following the intermediate (minor) change in first tunable variable. For the rest of the intermediate steps it is assigned to "0" and during the line search it is assigned to "2". Originally, NCD algorithm has very small `CHG` values. Also, it proceeds continuously and very fast so it does not make much difference where the plots are updated; in any case user will observe the improvements on plots. However, in our application it will be more useful to show the user the results of real-time simulation at each minor/major steps. For that goal, `OPT_STEP == 1` statement is basically removed from the `if` statement given above.

3.2 Modifications on `nlconst.m` Function

Mainly, necessity of modifications on `nlconst` routine arise from the knowledge of differences between the model simulation and real life execution. By these modifications, it is aimed to catch realistic step sizes for physical system and to avoid unnecessary iterations due to the combined effect of noise and non-repeatability of the real output data.

To be able to activate the modifications on `nlconst.m`, it is necessary to create a preparsed pseudocode file (p-file) of `nlconst.m`. This should be done by the command "`pcode nlconst`", which parses the m-file `nlconst.m` into the p-file `nlconst.p`.

3.2.1 Altering CHG value and Its Working Range

Change in variables for finite difference gradients, denoted as `CHG` in the algorithm requires to be amplified since the real system would not sense very small changes and in such a case, system would not give an appreciable variation in the output response. Depending on a larger `CHG`, upper and lower limit of `CHG`, `OPTIONS(16)` and `OPTIONS(17)`, should be enlarged. Logically, lower limit of `CHG` value should be chosen as the minimum change in tunable variables, which makes the real system produce a significant difference in output response. This requires a bit of specific system knowledge.

Related part of `nlconst` routine including the modifications (in lines between the stars) can be found below.

```
(….)
if ~analytic_gradient | OPTIONS(9)
        POINT = NPOINT;
        oldf = f;
        oldg = g;
        ncstr = length(g);
```

```
        FLAG = 0;
        gg = zeros(nvars, ncstr
        CHG = -1e-8./(GNEW+eps);
%************************************************************
        CHG=1e7*CHG
        OPTIONS(16)=0.1;
        OPTIONS(17)=1;
%************************************************************
CHG=sign(CHG+eps).*min(max(abs(CHG),OPTIONS(16)),OPTIONS(17));
        OPT_STEP = 1;
        for gcnt=1:nvars
            if gcnt == nvars,
              FLAG = -1
            end
            temp = XOUT(gcnt);
            XOUT(gcnt)= temp + CHG(gcnt);
            x(:) =XOUT;
            if strcmp(FUNfcn{4},'ncdtoolbox')
              [f,g] = feval(FUNfcn{1},x,varargin{:});
            else
              [f,g,msg] = opteval(x,FUNfcn,varargin{:});
              error(msg);
              g = g(:);
            end
(….)
```

3.2.2 Adding Merit Function Improvement Tolerance

By using the real system's output data in optimization process, it is necessary to avoid too stringent check for merit function improvement, since real systems have random noises and even for the same input, system may not give the same output response and the same merit function values. In such a case, strict values will be meaningless, instead, it is preferred to add an empirically found tolerance value into the line search algorithm. Notice that, giving this value very large will make the algorithm insensitive against the deterioration of optimization performance.

Related `while` loop of `nlconst` routine is given below and the modifications are shown by bold face characters.

(….)

```
while (MERIT2 > MATL2 + 0.1) & (MERIT > MATL + 0.1) & OPTIONS(10) <
OPTIONS(14) & ~OPT_STOP
```

(….)

3.2.3 Relaxing the Termination Criteria

Three termination criteria, related to change in tunable variables, change in cost function and maximum constraint violation should be relaxed for a real-time application by considering the same logic of section 3.2.1 and 3.2.2. As explained before, these criteria specified by `OPTIONS(2)`, `OPTIONS(3)` and `OPTIONS(4)` are adjustable by the user from **Variable** and **Constraint Tolerances** field of **Optimization Parameters** window. So, there is no need to make any modification inside the algorithm. Generally, enabling **Stop optimization as soon as the constraints are achieved** property will be the case in real-time applications, in which iterations take long time and overachieving is not the main target.

CHAPTER 4

MODEL BASED SIMULATION OF DC MOTOR SET-UP

4.1 DC Servomotor Experimental Set-up

The Feedback Control and Instrumentation MS150 Modular Servo system [13] is used as an experimental set-up to control the angular position of an inertia disc coupled to a DC servo motor by means of a reduction unit. By use of a potentiometer, position information about the inertia disk coupled to the DC-servomotor are available at feedback. The equipment used in the experimental set-up is listed and explained as follows:

Power Supply: This unit supplies a 24V direct current 2A unregulated supply to the motor through a multi way connector to the servo amplifier, as it is this unit that controls the motor.

DC Servomotor: A DC permanent magnet motor, which has an extened shaft, and onto which can be fixed the magnetic brake or inertia disc. The motor may also be attached to the Reduction Tacho Unit using the hexagonal coupling provided.

Pre-Amplifier: This unit provides the correct signals to drive the servo amplifier. The two inputs are effectively summed, allowing to signals to be applied e.g. a reference voltage and the tachogenerator voltage. A positive signal applied to either input causes the upper output terminal to go positive, the other output terminal staying near zero. A negative input causes the lower output to go positive, the upper one staying near zero. Thus bi-directional motor drive is obtained when these outputs are linked to the servo amplifier inputs.

Servo Amplifier: Transistors, which drive the motor in either direction, are contained in this unit.

Reduction Gear Tacho Unit: This unit contains a speed reduction gearbox with a ratio of 30/1 from the high speed input shaft to the low speed output shaft. A DC tachogenerator driven by the high speed shaft with an output on the top panel which can be used to display the tacho speed directly in rev/min or to monitor a DC voltage on another unit.

Input and Output Potentiometers: These are rotary potentiometers used for position control. Input potentiometer has ±150° of motion whilst the output potentiometer has no mechanical stops and so can not be damaged by continuous rotation. The input potentiometer is used to setup reference voltage and the output potentiometer is connected to the low-speed shaft by using the push-on coupling. Each unit has a buffer amplifier with a gain of one so that even if the output is shorted to a power supply or ground, the potentiometer will not be damaged by overloading. The buffer also ensures that the potentiometer wiper does not how to carry any current load during normal use.

Inertia Load: An aluminum disc can be mounted on the extended motor shaft and when rotated between the pulleys of the magnet of the loading unit, the eddy currents generated have the effect of the brake. The strength of the magnetic brake can be controlled by the position of the magnet. A heavy disc of the same diameter can also be mounted on the shaft instead of the aluminum disc to increase the inertia of the motor.

Figure 4.1 shows a schematic diagram of DC motor position control system. In Figure 4.2, a photograph of the experimental set-up is given.

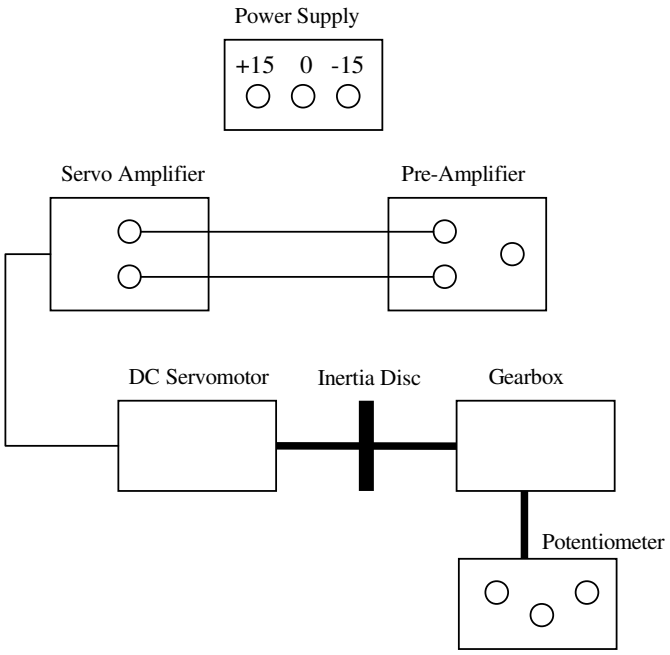


Figure 4.1 Schematic diagram of DC motor set-up

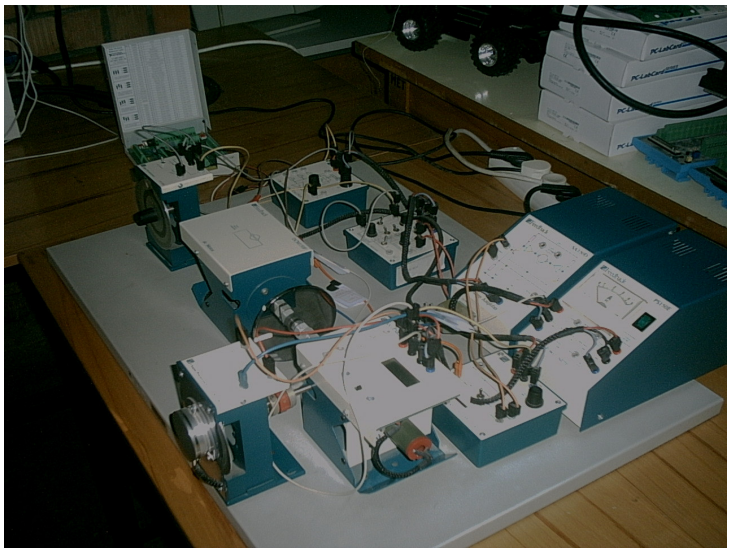


Figure 4.2 Photograph of DC motor set-up

4.2 Robustness Analysis of System with Non-repeatable Perturbations

A model of the control system implemented in MATLAB® /Simulink is used for the simulation and analysis of the dynamic system as shown in Figure 4.4. In **Simulation Parameters** dialog box of the model, simulation **Start time** is set to 0 and **Stop time** is set to 5. **Fixed-step** solver, **ode5 (Dormand-Prince)**, with a step size 0.01 is used. **Mode** is adjusted to **Auto** as shown in Figure 4.3.

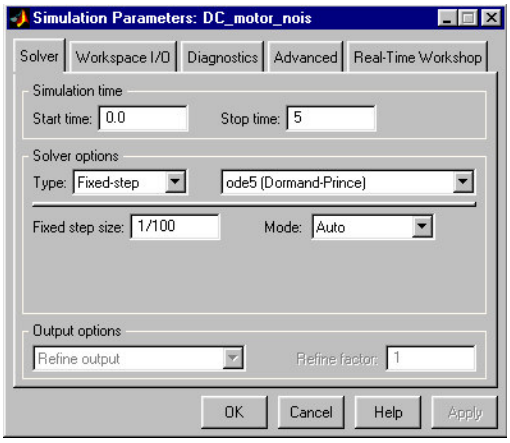


Figure 4.3 Simulation parameters window for robustness analysis

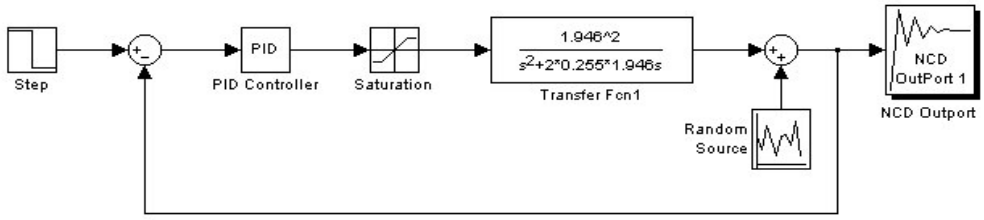


Figure 4.4 Simulink model including transfer function of DC motor setup

Transfer function of the plant is obtained experimentally [14] as shown in equation (4.1). Also, a saturation block with +10V upper limit and -10V lower limit should

be added into the model since in real-time applications data acquisition cards have a similar saturation process.

$$G(s) = \frac{1.946^2}{s^2 + 2\times 0.255\times 1.946 s} \tag{4.1}$$

To compensate for the system model inaccuracies, a random source generator is added into the Simulink model. Random source type was chosen as “uniform” and the minimum-maximum range for the amplitude of noise was ±0.1. Considering the noise in the real life would be completely random, repeatability term defined as “non-repeatable” in the block parameters. Also, sample time is same as the simulation step time, which was set to 0.01 before. Figure 4.5 shows properties of random source block diagram.

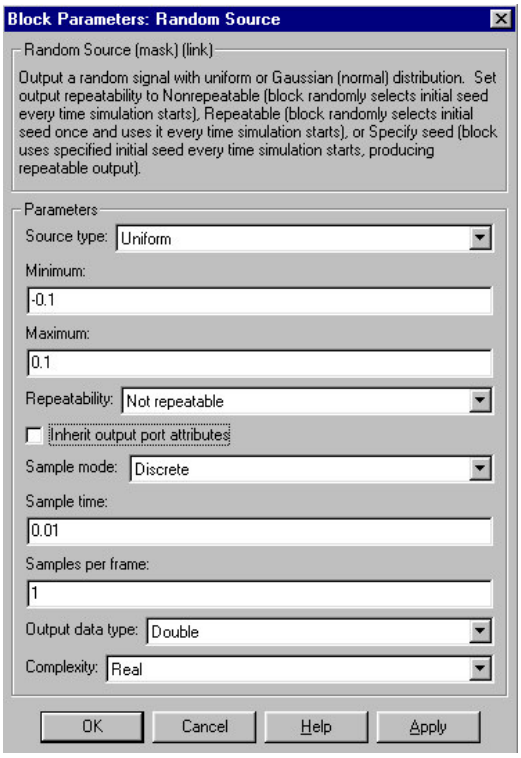


Figure 4.5 Random source block parameters

By having a random source generator, the model given in Figure 4.4 can be thought as a prior version of actual real-time applications. However, since this model is prepared in Simulink **normal** mode, it allows the command line simulation by `sim` command inside the `costfun` routine. So, there is no need to apply the modifications done in `costfun` routine, although the modifications in `nlconst` routine are still compulsory due to the random source effect.

For the initial values of Kp=1 Ki=1 Kd=1, one can see how the output response satisfies the given constraints in Figure 4.6.

Command window displays the following information during the process:

```
Setting up constraint window ........ done
Processing uncertainty information.
No uncertainty modeled.
Setting up call to optimization routine.
Done plotting the initial response.
Start time: 0      Stop time: 5.
There are 1005 constraints to be met in each simulation.
There are 3 tunable variables.
There are 1 simulations per cost function call.

 f-COUNT     MAX{g}     STEP        Procedures

    5       0.360424     1
   10       0.375529     1        Hessian modified
   15       0.385697     1        Hessian modified twice
   20       0.251793     1        Hessian modified twice
   25       0.214934     1        Hessian modified
   30       0.209884     1
   35       0.179054     1        Hessian modified
   40       0.181963     1
   45       0.167418     1
   50       0.146963     1        Hessian modified
   55       0.0848281    1        infeasible
   61       0.0976798    0.5
   66       0.0492157    1
   72       0.0668465    0.5      Hessian modified
```

```
   77        0.0273459     1         Hessian modified twice
   83        0.053649      0.5       Hessian modified; infeasible
   88        0.047488      1         Hessian modified; infeasible
   94        0.0382968     0.5       Hessian modified
   99        0.0566212     1         Hessian modified
  105        0.0493392     0.5
  111        0.0437437     0.5       infeasible
  117        0.0177746     0.5
  129        0.0793763     0.00781
  134        0.0441082     1         infeasible
  140        0.0400582     0.5       Hessian modified
  147        0.0396834     0.25
  152       -0.00120728    1         Hessian modified
  167       -0.00601804    -6e-005   Hessian modified

Optimization Converged Successfully
Active Constraints:
    504
    506
```

Finally, the optimized values of the controller gains are found as follows:

```
Kp= 3.4619
Ki= 0.37073
Kd= 0.7438
```

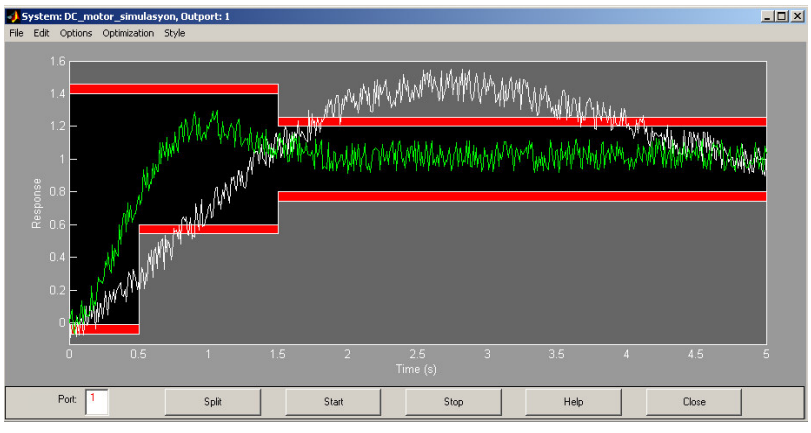


Figure 4.6 Noisy output response of DC motor Simulink model

CHAPTER 5

HARDWARE-IN-THE-LOOP SIMULATION ON DC MOTOR SET-UP

Once the algorithms have been developed and tested in software, the next step is to bridge the gap between software simulation and real world applications. Here, the method of hardware-in-the-loop simulation is applied by using DC servomotor experimental set-up introduced in previous chapter. A schematic diagram of this hardware-in-the-loop application can be shown in Figure 5.1. The angular position of the inertia disc is measured by the potentiometer and this information is passed to the computer environment, which consists of MATLAB® / (RTWT) and Simulink. In advance, MATLAB® optimization routine produces the required control signals for the pre-amplifier, then it provides the correct signals to servo amplifier which actually drives the servomotor. The information flow between the software and hardware environments, i.e., sensory signal from physical system to computer and command signal from software environment to the physical system, is acquired by means of National Instruments PCI 6025 Data Acquisition (DAQ) Cards. The whole process runs in real-time, which is controlled by MATLAB® / RTWT.

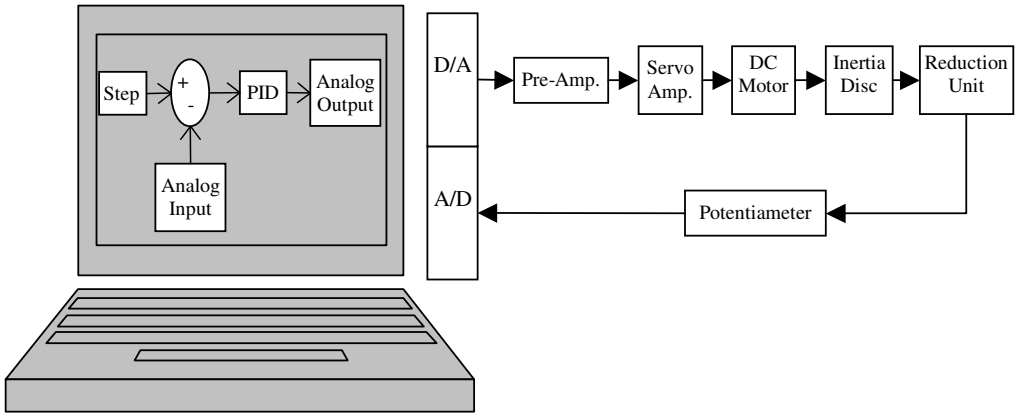


Figure 5.1. Schematic diagram of hardware-in-the-loop application

Simulink model for the real-time application is also given in Figure 5.2. **Simulation Parameters** of this real-time model are same as introduced in Section 4.2.

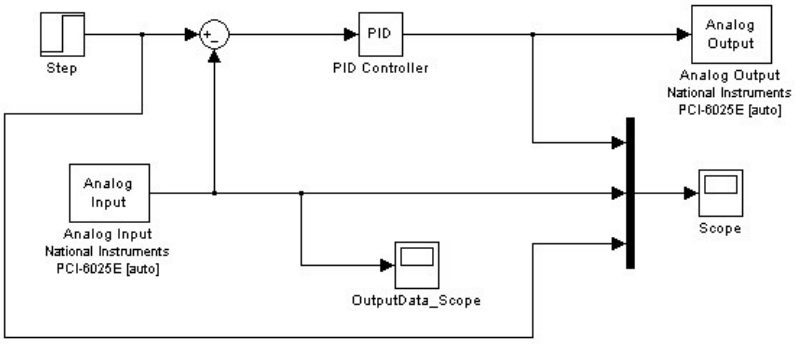


Figure 5.2. Simulink model for real-time application

5.1. Statistical Error Analysis

A statistical error analysis of the physical measuring methods or procedure employed in ascertaining the output response characteristics is prepared prior to implementation of the optimization algorithm with the real time execution.

Here the "repeatability" term occurs which is the difference in output values for the same input values. For DC motor set-up, repeatability error would be the difference in angular position of the inertia disc at each time points when an identical step input is applied on the model with same controller gains values. In order to calculate an average system response and other statistical values, exactly same experiments have been repeated for 25 times with the same initial values of tunable variables, specifically for Kp=1 Ki=1 Kd=1 and output response data have been plotted as shown in Figure 5.3. Scale factor of the system is 12°/V.

Differences on the collected data reflect that there is a certain amount of friction in mechanical parts and measurement device (potentiometer). Also A/D data

acquisition card is of finite resolution. It has 12 bit resolution and 19.012 mV absolute accuracy at full scale [15].

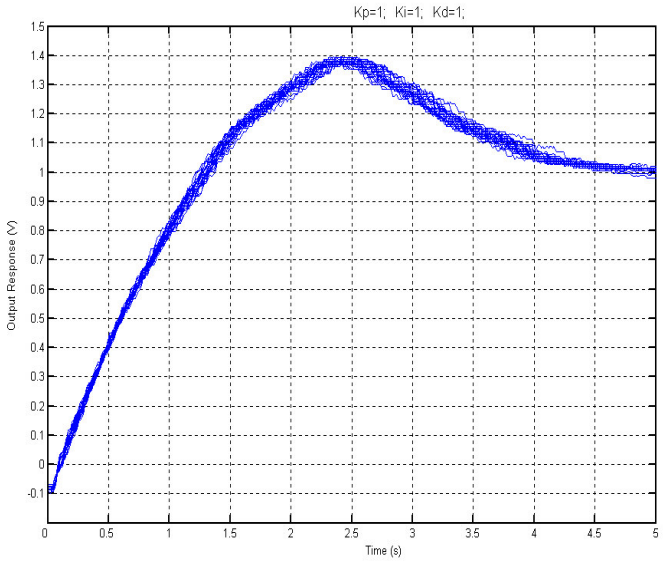


Figure 5.3. Repeated output response plots of DC motor set-up

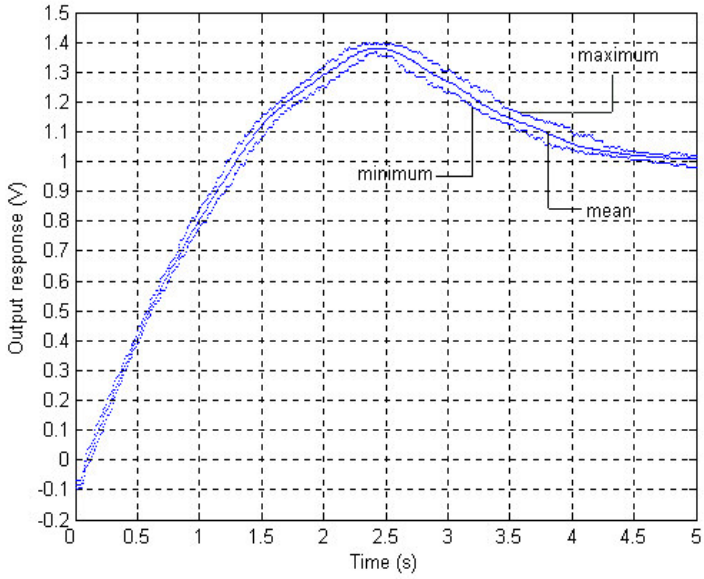


Figure 5.4. Max.-Min.-Mean plots for repeated output response

Maximum, minimum and mean values of output data at each time points is plotted in Figure 5.4.

In Figure 5.5, bandwidth of max.-min. output values and the standard deviation at each time step is shown by using the data obtained in statistical error analysis. According to this, maximum bandwidth of the distribution curve is 98 mV, which implies that almost 20 % of total error arise from specified data acquisition card.

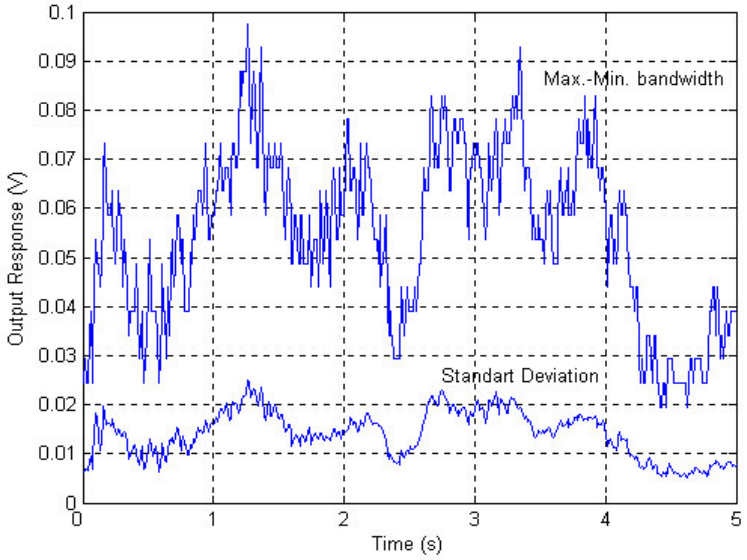


Figure 5.5. Max-Min bandwidth and standard deviation

5.2. Real-Time Application Method

Case studies are done with the following steps:

1) Open the following **DC_motor.mdl**.

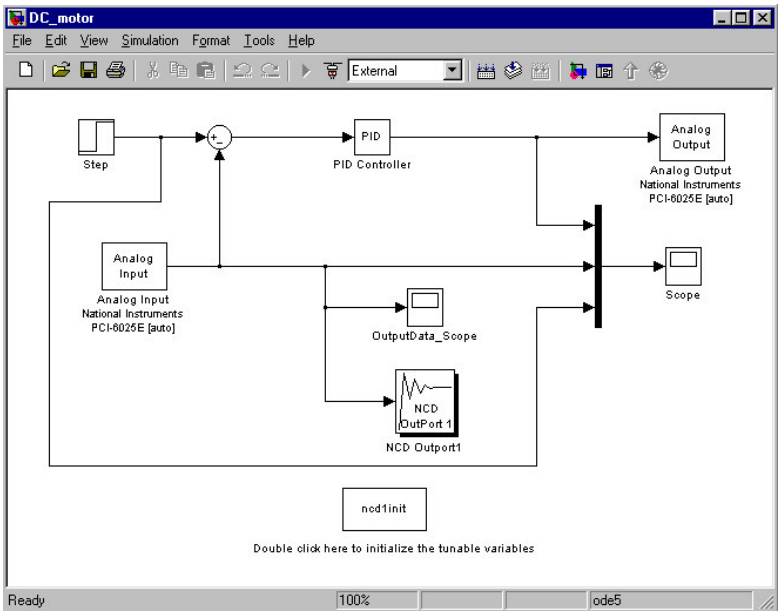


Figure 5. 6. DC motor RTWT control model

2) Double click `ncd1init` subsystem to initialize the tunable variables and to declare `OutputData` variable as global.

3) Double click NCD Outport to open the constraint figure. Default constraint bounds will be used in the experiments. Enter the name of tunable variables as "Kp Ki Kd" into optimization parameters window and 0.01 for **Discretization Interval**. Set variable and constraint tolerances to 0.01 again. Disable **Compute gradients with better accuracy** option and press **done**.

4) Press the **Start** button to start the optimization. Note that after a while, the algorithm will pause in keyboard mode and wait for an input from user by the command `K>>`.

   For an initial set of tunable variables, Kp=1 Ki=1 Kd=1, command window must appear as follows:

```
Setting up constraint window ........ done
Processing uncertainty information.
No uncertainty modeled.
```

```
Setting up call to optimization routine.
Kp=1.000  Ki=1.000  Kd=1.000
K>>
```

5) In Simulink window, do not forget to update PID parameters from **update diagram** in **edit** pane (or press Ctrl+D) before starting the real-time execution. Then press **connect to target** to get the system ready to run.

6) Set the inertia disc position to zero in potentiometer.

7) Press **Start real-time code** to execute the real system. Observe that the system stops after 5 seconds.

   Command window must appear as follows:

```
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
```

   Note that a global variable named `OutputData` is saved into the base workspace.

8) To turn back the optimization process, write "return" in command window and press **enter**. By this action, the system will exit from keyboard mode.

   Again for an initial set of tunable variables, Kp=1 Ki=1 Kd=1, command window must appear as follows:

```
K>> return
Done plotting the initial response.
Start time: 0      Stop time: 5.
There are 1005 constraints to be met in each simulation.
There are 3 tunable variables.
There are 1 simulations per cost function call.
Kp=1.000  Ki=1.000  Kd=1.000  f=1.000
K>>
```

Constraint figure window will appear as shown in Figure 5.7.

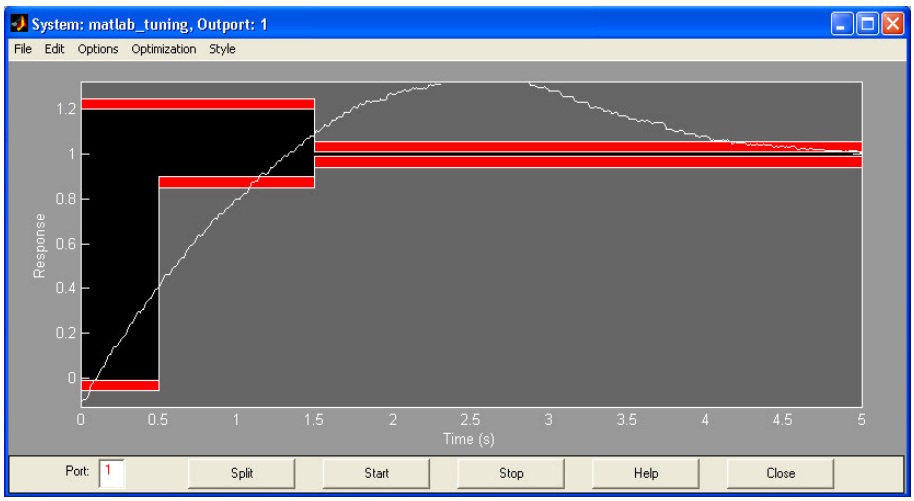


Figure 5.7. Example constraint figure window of initial response

9) Repeat the same procedure starting from step 5 for every new set of tunable variables Kp, Ki, Kd & f.

5.3. Case Study I

By following the described steps, first study is done with a set of initial values of tunable variables, Kp=1 Ki=1 Kd=1. Important parameters are displayed in command window and an appearance of constraint figure window is placed to let the user observe the output response improvement after each SD calculation i.e., major steps. For the whole process, command window appears as follows:

```
Setting up constraint window ........ done
Processing uncertainty information.
No uncertainty modeled.
Setting up call to optimization routine.
Kp=1.000  Ki=1.000  Kd=1.000
K>>
Model DC_motor_seda3 loaded
```

```
Model DC_motor_seda3 unloaded
K>> return
Done plotting the initial response.
Start time: 0   Stop time: 5.
There are 1005 constraints to be met in each simulation.
There are 3 tunable variables.
There are 1 simulations per cost function call.

Kp=1.000  Ki=1.000  Kd=1.000  f=1.000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

CHG =
   -0.1000
   -0.1000
   -0.1000
   -0.1000

Kp=0.900  Ki=1.000  Kd=1.000  f=1.000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.000  Ki=0.900  Kd=1.000  f=1.000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.000  Ki=1.000 Kd=0.900  f=1.000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.000  Ki=1.000  Kd=1.000  f=0.900
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
```

```
K>> return

mg = -0.5346

    f-COUNT      MAX{g}          STEP     Procedures
       5        0.46543            1

Step Length = 1

OPTIONS(11)= 2

SD =
    0.2985
    0.1568
   -0.1441
   -0.6830

Kp=1.29848  Ki=1.15679  Kd=0.85586  f=0.31703
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure 5.8.

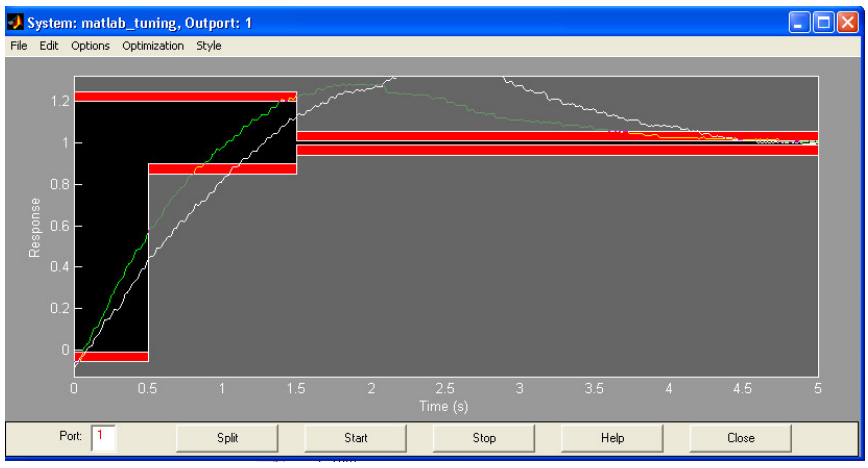


Figure 5.8. Constraint figure window at OPTIONS(11)=2

```
CHG =
   -0.1000
   -0.1000
   -0.1000
   -0.1000

Kp=1.19848  Ki=1.15679  Kd=0.85586  f=0.31703
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.29848  Ki=1.05679  Kd=0.85586  f=0.31703
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.29848  Ki=1.15679  Kd=0.75586  f=0.31703
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.29848  Ki=1.15679  Kd=0.85586  f=0.21703
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg = 0.0117

    f-COUNT     MAX{g}          STEP    Procedures
       10     0.328711            1

Step Length = 1

OPTIONS(11)=  3

SD =
    0.1238
```

```
   -0.0129
   -0.0806
   -0.0369

Kp=1.42223 Ki=1.14394  Kd=0.77528  f=0.28014
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure 5.9.

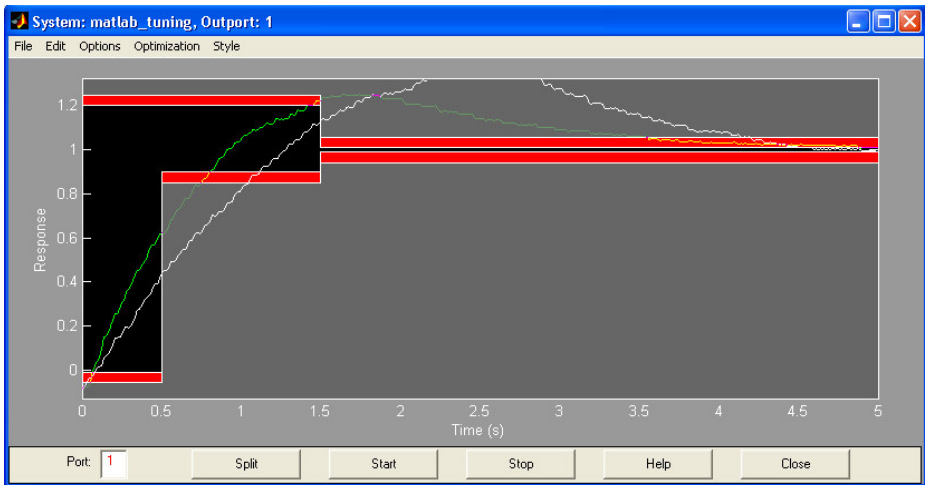


Figure 5.9. Constraint figure window at OPTIONS(11)=3

```
CHG =
    0.3930
    0.8394
   -0.9542
    0.3678

Kp=1.81522  Ki=1.14394  Kd=0.77528  f=0.28014
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

```
Kp=1.42223  Ki=1.98335  Kd=0.77528  f=0.28014
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.42223  Ki=1.14394  Kd=-0.17896  f=0.28014
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.42223 Ki=1.14394  Kd=0.77528  f=0.64792
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg = -2.5832e-004

    f-COUNT     MAX{g}          STEP    Procedures
      15       0.279883          1      Hessian modified

Step Length = 1

OPTIONS(11)= 4

SD =
    0.5761
   -0.0190
   -0.0046
   -0.1475

Kp=1.99832  Ki=1.12490 Kd=0.77069  f=0.13269
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure 5.10.

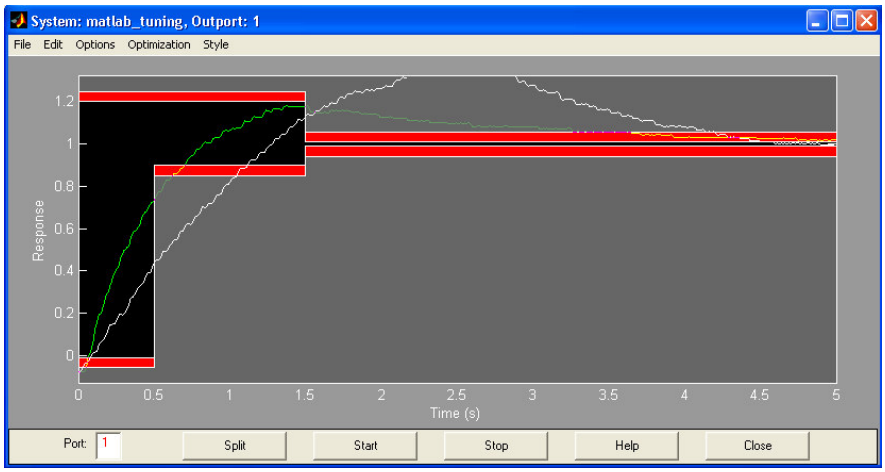


Figure 5.10. Constraint figure window at OPTIONS(11)=4

```
CHG =
    0.5241
    1.0000
   -0.1000
   -1.0000

Kp=2.52244  Ki=1.12490  Kd=0.77069  f=0.13269
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.99832  Ki=2.12490  Kd=0.77069  f=0.13269
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.99832  Ki=1.12490  Kd=0.67069  f=0.13269
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=1.99832  Ki=1.12490  Kd=0.77069  f= -0.86731
K>>
```

```
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg = 0.0390

    f-COUNT     MAX{g}          STEP    Procedures

       20      0.171641           1     Hessian modified

Step Length = 1

OPTIONS(11)= 5

SD =
    1.8837
   -0.0639
   -0.0489
   -0.0727

Kp=3.88198  Ki=1.06104  Kd=0.72176  f=0.05999
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure 5.11.

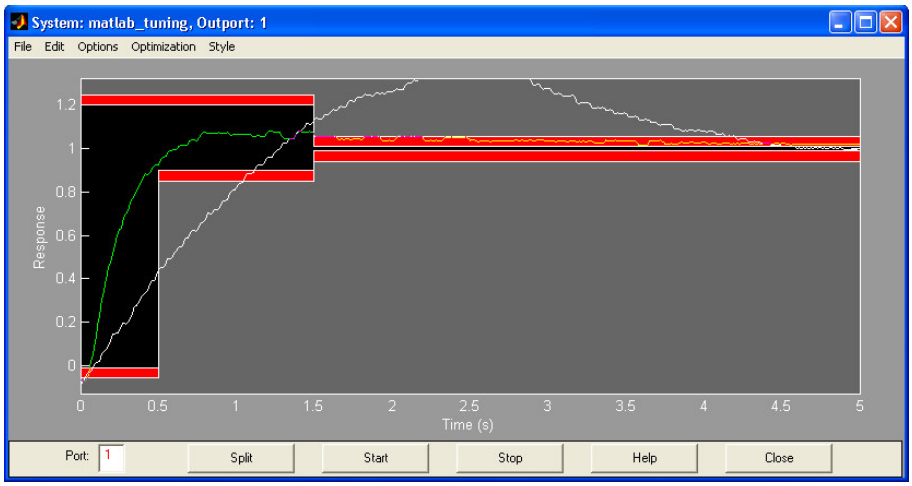


Figure 5.11. Constraint figure window at OPTIONS(11)=5

```
CHG =
    0.6518
    1.0000
   -0.1643
   -1.0000

Kp=4.53382  Ki=1.06104  Kd=0.72176  f=0.05999
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.88198  Ki=2.06104  Kd=0.72176  f=0.05999
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.88198  Ki=1.06104  Kd=0.55749  f=0.05999
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.88198  Ki=1.06104  Kd=0.72176  f= -0.94000
K>>
Model DC_motor_seda3 loaded
```

```
Model DC_motor_seda3 unloaded
K>> return

mg = 0.0032

 f-COUNT      MAX{g}          STEP     Procedures
    25      0.0632422            1      Hessian modified twice

Step Length = 1

OPTIONS(11)=  6

SD =
    0.0274
   -0.0039
   -0.0075
    0.0031

Kp=3.90938  Ki=1.05711  Kd=0.71421  f=0.06305
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure 5.12.

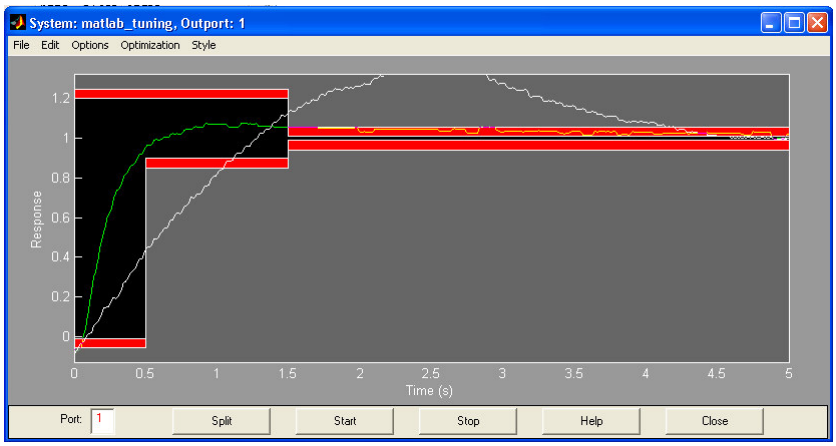


Figure 5.12. Constraint figure window at OPTIONS(11)=6

```
CHG =
     1
     1
    -1
    -1

Kp=4.90938  Ki=1.05711  Kd=0.71421  f=0.06305
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.90938  Ki=2.05711  Kd=0.71421  f=0.06305
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.90938  Ki=1.05711  Kd= -0.28579  f=0.06305
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.90938  Ki=1.05711  Kd=0.71421  f= -0.93694
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
mg = 1.8597e-004

    f-COUNT     MAX{g}          STEP    Procedures

       30      0.0632422          1     Hessian modified

Step Length = 1

OPTIONS(11)=  7

SD =
    0.0129
    0.0013
```

```
    0.0033
    0.0001

Kp=3.92224   Ki=1.05838   Kd=0.71750   f=0.06320
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure 5.13.

Figure 5.13. Constraint figure window at OPTIONS(11)=7

```
    f-COUNT      MAX{g}          STEP     Procedures
       31      0.0544531          1      Hessian modified

Optimization Converged Successfully

Active Constraints:
   288
   504
```

While the initial conditions are Kp=1 Ki=1 Kd=1, final results are found as follows:

Kp = 3.9222
Ki = 1.0584
Kd = 0.7175

Trend of tunable variables, cost function and termination criteria parameters during the whole process can be observed by below Table 5.1.

Table 5.1. Results of case study I

Kp	Ki	Kd	gamma	max |SD|	|gf'*SD|	mg
1.0000	1.0000	1.0000	0.4654	-	-	-0.5346
1.2984	1.1568	0.8558	0.3287	0.6830	0.6830	0.0117
1.4222	1.1439	0.7753	0.2799	0.1238	0.0369	-0.0003
1.9983	1.1249	0.7707	0.1716	0.5761	0.1475	0.0390
3.8819	1.0610	0.7217	0.0632	1.8837	0.0727	0.0032
3.9094	1.0571	0.7142	0.0632	0.0274	0.0031	0.0002
3.9222	1.0584	0.7175	0.0544	0.0129	0.0001	-0.0087

Plots of each tunable variables and cost function are given in Figure 5.14 and Figure 5.15. Also, in Figure 5.16 one can observe the improvements on output response behavior during the optimization process.

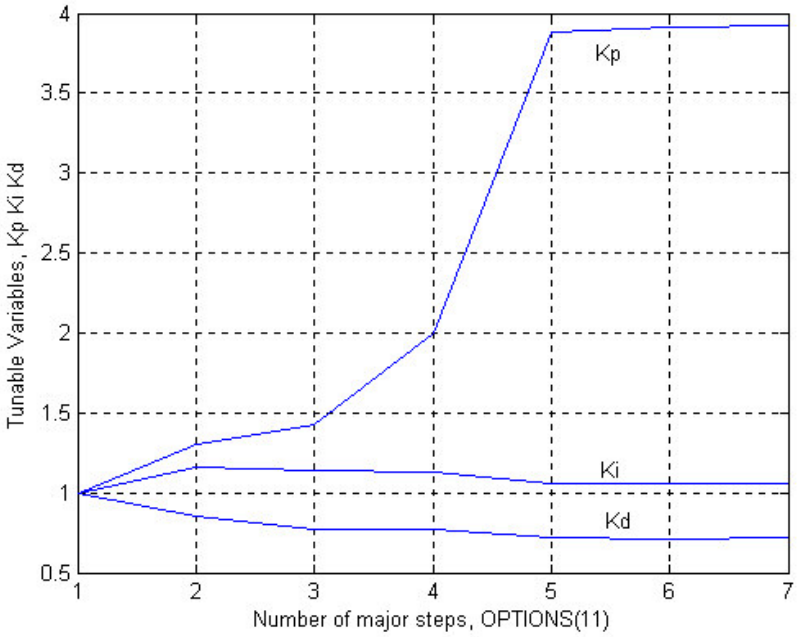


Figure 5.14.Plots of tunable variables for case study I.

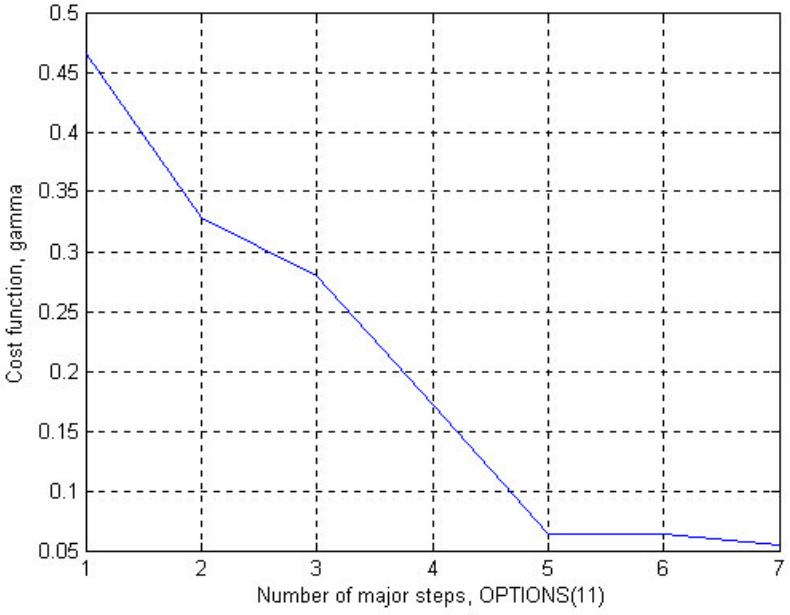


Figure 5.15. Plots of cost function for case study I.

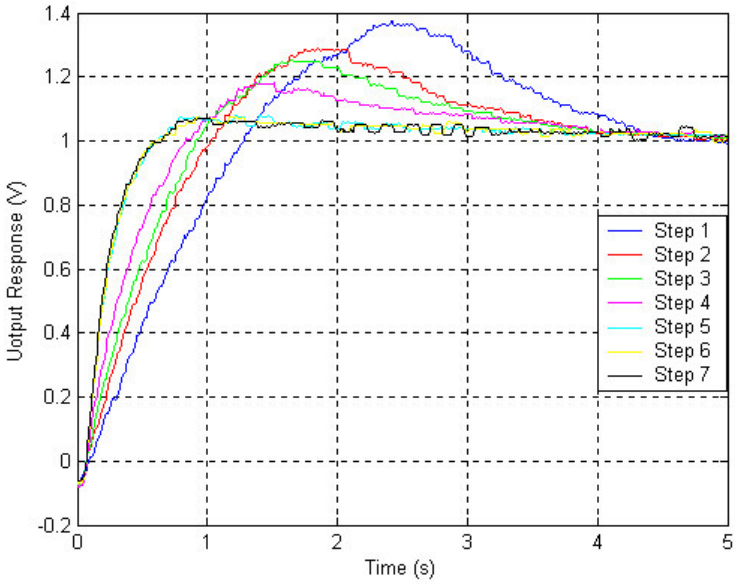


Figure 5.16. Output response improvement during case study I.

5.4. Case Study II

This study is done with a set of initial values of tunable variables, Kp=3 Ki=1 Kd=0, which gives an oscillatory initial output response and makes system very close to marginally stable condition. To ensure the stability of the system, each of tunable variables are bounded by zero in **Lower bound** from **Optimization Parameters** window. Also considering that the initial output response is very far from the given constraints, **Stop optimization as soon as the constraints are achieved** property is enabled to decrease the number of iteration. Complete command window display and related constraint figure windows after each major step are given in Appendix A. Here, only the necessary plots and summary information will be shown.

For a set of initial conditions, Kp=3 Ki=1 Kd=0, final results are found as follows:

Kp = 3.4635

Ki = 0.1650

Kd = 0.2001

At the end of the process, constraint figure window displays the system's output response plots belonging to initial and optimized values of controller gains as shown in Figure 5.17. Trend of each tunable variables, cost function and termination criteria parameters during the whole process can be observed by below Table 5.2.

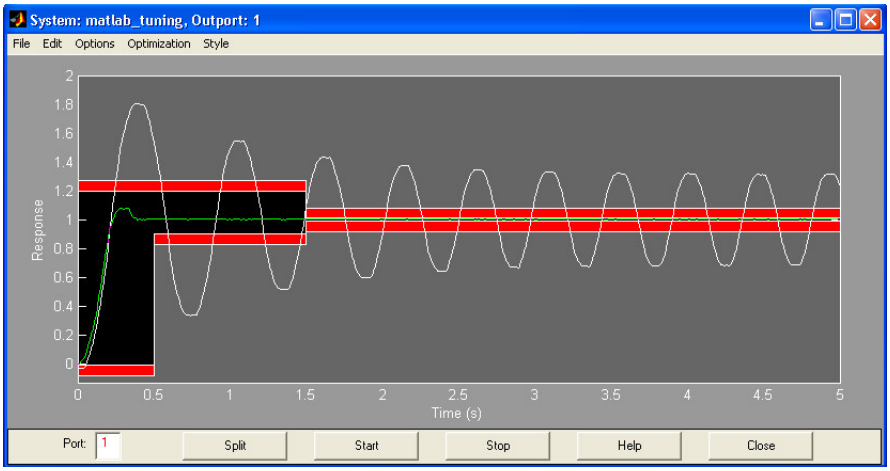


Figure 5.17. Constraint figure window at the end of case study II.

Table 5.2. Results of case study II.

Kp	Ki	Kd	gamma	max |SD|	|gf'*SD|	mg
3.0000	1.0000	0.0000	0.6018	0.0000	0.0000	-0.3982
3.1555	0.8238	0.0000	0.5773	0.3165	0.3165	-0.1062
3.0192	0.7623	0.0022	0.5383	0.1514	0.1514	0.0061
2.9834	0.7338	0.0058	0.5285	0.0358	0.0096	0.0060
2.3617	0.7156	0.2220	0.0301	0.6217	0.3462	-0.1463
2.3241	0.4993	0.1943	0.0203	0.8333	0.8333	0.0203
3.5074	1.4421	0.1937	0.0252	1.1833	0.5535	0.0251
3.4659	0.2416	0.1997	0.0007	1.2005	0.7511	0.0007
3.4635	0.1650	0.2001	-0.0041	0.0766	0.0000	-0.0041

Plots of each tunable variables and cost function are given in Figure 5.18 and Figure 5.19. Also, in Figure 5.20 one can observe the improvements on output response behavior during the optimization process following the graphs by row-wise from left to right.

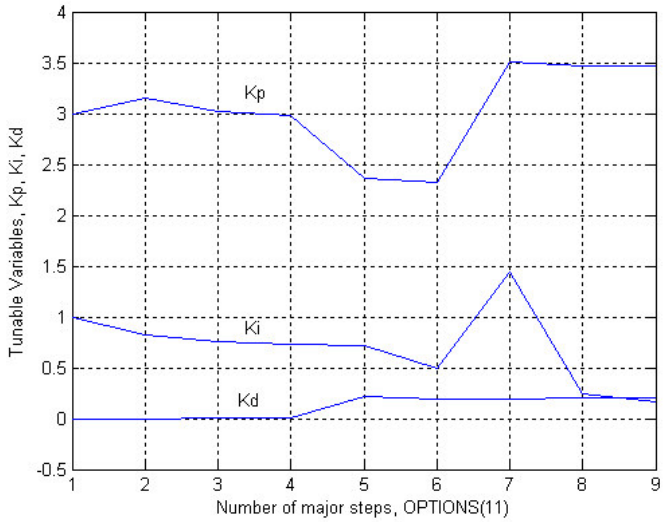


Figure 5.18. Plots of tunable variables for case study II.

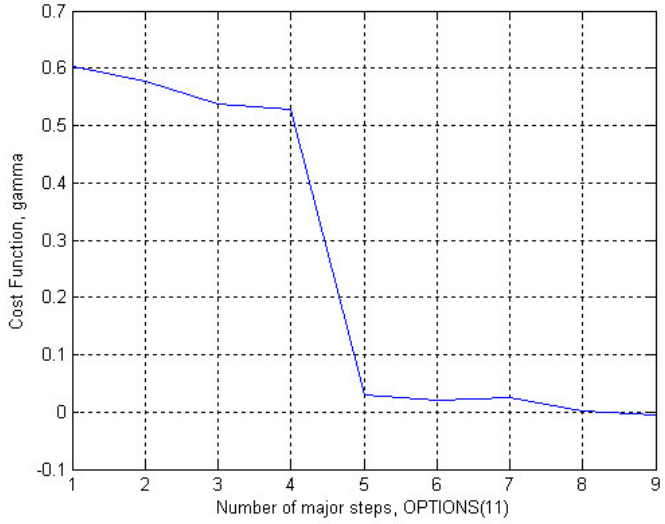


Figure 5.19. Plots of cost function for case study II

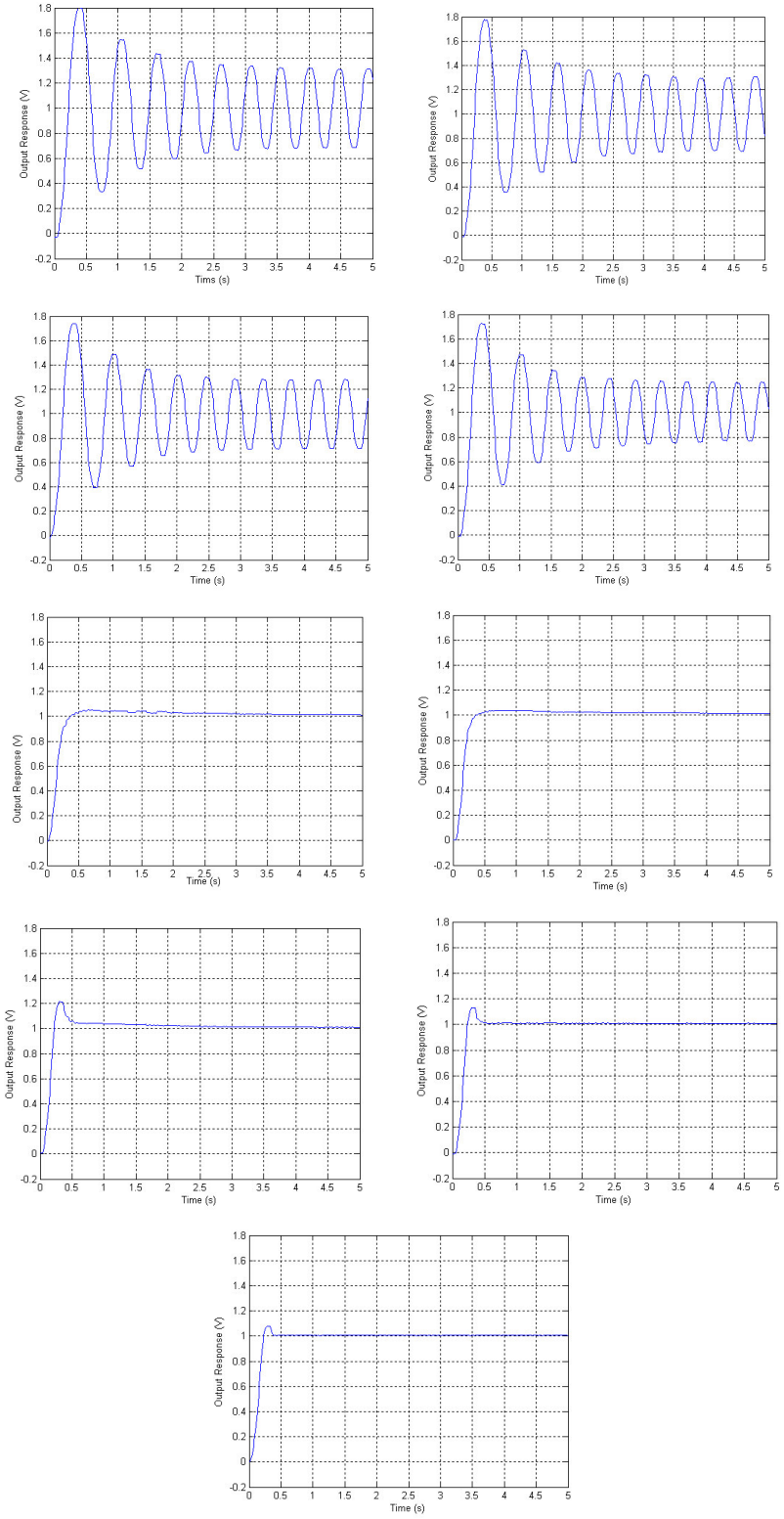


Figure 5.20. Output response improvement during case study II

CHAPTER 6

DISCUSSION AND CONCLUSIONS

6.1. Discussion and Conclusions

The aim of this thesis is to develop an on-line strategy which will lead to the determination of optimum control system parameters, based on presently available algorithm of MATLAB® 6.5 R13 (SP1) / Nonlinear Control Design Blockset Version 1.1.6.

The basic idea behind NCD algorithm is introduced in a logical sequence. Processes of three vital optimization routines and interactions with each other are analyzed in detail. This requires a dedicated study on the algorithm and has a special importance not only by being the milestone of this thesis but also a guide for further improvements on the algorithm. A summarizing pseudo-code is given for a better understanding.

Then an illustrative model in "external" mode prepared by the help of MATLAB® / Real Time Windows Target (RTWT) is introduced and present NCD algorithm is modified so that it could be used in such a real-time application. Modifications are necessary mainly for two reasons: To transfer input/output data between the physical system and the algorithm and also to amplify some parameters such as change in tunable parameters according to physical plant responsivity. The first necessity arise from the fact that it is not possible to perform command line simulation of a model in external mode.

A demonstrative PID tuning process is realized by using the model-based simulation of a DC servomotor set-up (including the transfer function of the plant) under the

effect of non-repeatable perturbations (random source) on the response signal. This study can be thought as an intermediate step before starting the real-time case studies, which actually contain noises. The satisfactory result as shown in Figure 4.6 is promising for the real-time application.
Finally, the hardware-in-the-loop simulation on previously defined DC servomotor set-up is done by using the RTWT model which is also discussed in Chapter 3. A set of statistical error analysis results are given at that point to determine the non-repeatability of real system output response data. For both case studies an experimentation method is fixed. During the case studies, example tuning processes are presented to show some of the potential uses of the model.

When compared to the trial and error method, which is used widely in industry, this method offers a more scientific and logical approach to a difficult problem of tuning control systems. Also, when the control strategy is not well-known, unlike a PID controller, tuning by trial and error method will be a time-consuming process or almost insoluble. The strategy is unique in the sense that it is the one and only control system tuning method applying an iterative optimization algorithm with directly physical plant's input/output data usage, known to the author.

6.2 Future Scope

This study can help engineers to design controllers by a systematic and progressive approach with the proposed "tuning by hardware-in-the-loop simulation" strategies. Although no mathematic formula is used to describe this approach, it has sound philosophic background, and could be a very easy and powerful tool for some extended projects.

The course of work took more than three years because of the fact that such an application has never been done before. To the author's knowledge, this study proposes one of the few algorithms to ensure output response obtained by the tuned results will satisfy the constraints of the real system and is the only one that applies the hardware-in-the-loop simulation concept to the problem of finding optimized

controller gain values in a control system. Depending on the slackness of the constraints, more than one solution set can be obtained. This method aims to find an optimum solution set, although it might not be the "best" one. This is because NCD finds the local minimum and does not guarantee that it's a global minimum. For being closer to a global minimum, algorithm might be forced to find more than one solution set for the specific tuning process and the program might choose the best one as the final solution set.

This thesis deals with minimizing maximum error method but there are many more methods to solve multi objective nonlinear problems. More work could be directed toward that area where many routines are used on the same problem and the best optimization method can be chosen by the user. Also, a better method for choosing the initial parameter values should be developed. Minor step length, `CHG`, used during the finite difference gradient calculation is one of the key values for the real-time application and should be determined by the user with some amount of pre-knowledge of the system. A user-interface will be helpful for a better `CHG` value specification depending on the specific real plant characteristics.

Due to the complex nature of real-time simulation within an iterative area, a simple model was taken for the study. In future methods more work could be done with more complicated systems, and also a possible automatic application of that process will have a great contribution to the study in terms of time and effort saving for the future works.

REFERENCES

[1] **Nguyen, H.T., Sugeno, M., (1998),** "Fuzzy Systems Modeling And Control", *Kluwer Academic Publishers*.

[2] **Ogata, K., (1997),** "Modern Control Engineering", Prentice-Hall, 3rd ed.

[3] **Åström, K.J., Hägglund, T., (1995),** "PID Controlllers: Theory, Design and Tuning", *Instrument Society of America*, Research Triangle Park, NC, 2nd ed.

[4] **Chien, K.L., Hrones, J.A., Reswick, J.B., (1952),** "On the Automatic Control of Generalized Passive Systems", *Transactions of the ASME*, 74, pp. 175-185.

[5] **Dahlin, E.B., (1968),** "Designing and Tuning Digital Controllers*", Instruments and Control Systems*, 42, pp. 77-83.

[6] **Haalman, A., (1965),** "Adjusting Controllers for a Dead-time Process", *Control Engineering*, July-65, pp. 71-73.

[7] **McMillan, G.K., (1983),** "Tuning and Control Loop Performance", *Instruments Society of America*, North Carolina, USA.

[8] **Campi, M.C., Lecchini, A., Savaresi, S.M., (2002),** "Virtual Reference feedback tuning: A Direct Method for the Design of Feedback Controllers", *Automatica*, 38, pp. 1337 – 1346.

[9] **O' Mahony, T., Downing,C.J., Klaudiuz, F.,** "Genetic Algorithms for PID Parameter Optimization: Minimising Error Criteria", [online], URL: http://www.pwr.wroc.pl/~i-8zas/kf_glas00.pdf

[10] **Best, M.C., Howell, M. N., (2000),** "On-line PID Tuning for Engine Idle-s-Speed Control Using Continuous Action Reinforcement Learning Automata", *Control Engineering Practice* 8, pp. 147-154.

[11] **Kocijan, J., Karba, R., (1997),** "The Chemical Process Application of Multivariable Control Hardware and Algorithm Testing by Means of Simulation". *Simulation Practice and Theory*, 5, pp.153–165.

[12] **MathWorks Inc., (2002),** MATLAB® Nonlinear Control Design Blockset User's Guide, Version 1.1.6, July 2002 .

[13] **Feedback Instruments Ltd.,** "Modular Servo System MS150 DC, Synchro & AC Basic Assignments", User Manual, pp. 16-23.

[14] **Middle East Technical University, ME 304, (2003),** " Closed loop position control of a DC motor", Experiment Manual, Spring 2003.

[15] **National Instruments**, **(2004)**, The Measurement and Automation Catalog.

APPENDIX A

COMMAND WINDOW DISPLAY OF CASE STUDY II

```
Setting up constraint window ........ done
Processing uncertainty information.
No uncertainty modeled.
Setting up call to optimization routine.
Kp=3.0000000   Ki=1.0000000   Kd=0.0000000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Done plotting the initial response.
```

Constraint figure window appears as shown in Figure A.1.

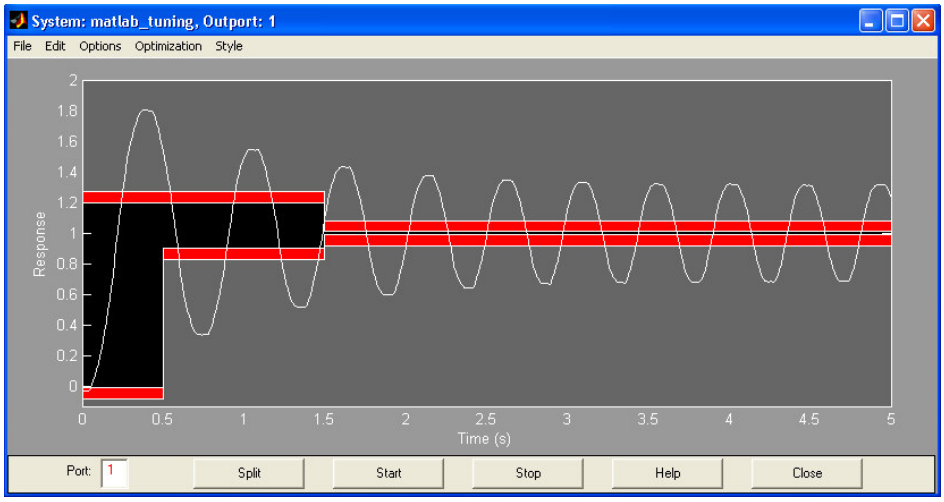


Figure A.1. Constraint figure window at initial response

```
Start time: 0      Stop time: 5.
There are 1005 constraints to be met in each simulation.
There are 3 tunable variables.
There are 1 simulations per cost function call.
OPTIONS(11)=1
Kp=3.0000000  Ki=1.0000000  Kd=0.0000000  f=1.0000000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

CHG =
   -0.1000
   -0.1000
   -0.1000
   -0.1000

Kp=2.9000000  Ki=1.0000000  Kd=0.0000000  f=1.0000000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.0000000  Ki=0.9000000  Kd=0.0000000   f=1.0000000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.0000000  Ki=1.0000000  Kd= -0.1000000   f=1.0000000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.0000000  Ki=1.0000000  Kd=0.0000000     f=0.9000000
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg = -0.3982
```

```
      f-COUNT      MAX{g}          STEP     Procedures

           5       0.601758           1

OPTIONS(11)= 2

Step Length =  1

SD =
    0.1555
   -0.1762
   -0.0000
   -0.3165

Kp=3.1554726  Ki=0.8237977  Kd=0.0000000  f=0.6835372
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.2.

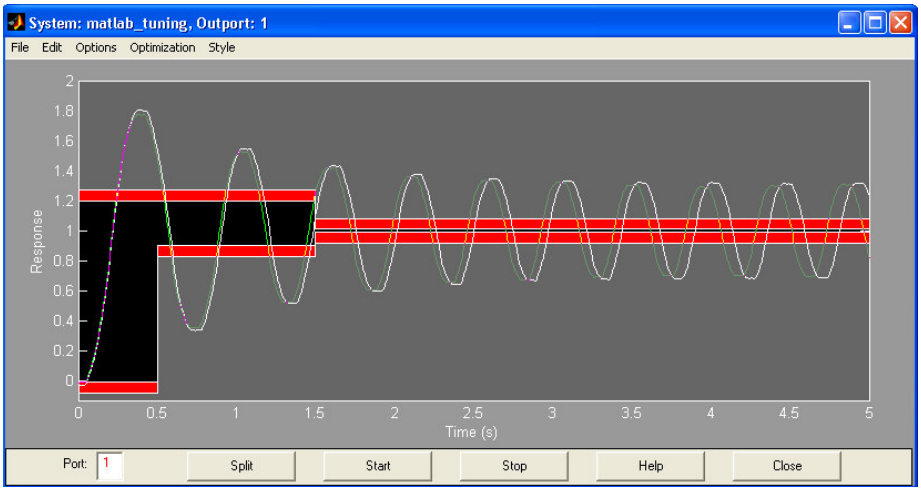


Figure A.2. Constraint figure window at OPTIONS(11)=2

```
CHG =
   -0.1000
   -0.1000
   -0.1000
   -0.1000

Kp=3.0554726  Ki=0.8237977  Kd=0.0000000  f=0.6835372
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.1554726  Ki=0.7237977  Kd=0.0000000  f=0.6835372
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.1554726  Ki=0.8237977  Kd= -0.1000000  f=0.6835372
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.1554726  Ki=0.8237977  Kd= 0.0000000  f=0.5835372
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg =  -0.1062

     f-COUNT     MAX{g}          STEP    Procedures

        10     0.577344            1     Hessian modified

OPTIONS(11)= 3

Step Length = 1

SD =
   -0.1362
```

```
   -0.0615
    0.0022
   -0.1514

Kp=3.0192227  Ki=0.7622811  Kd=0.0022436  f=0.532146
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.3.

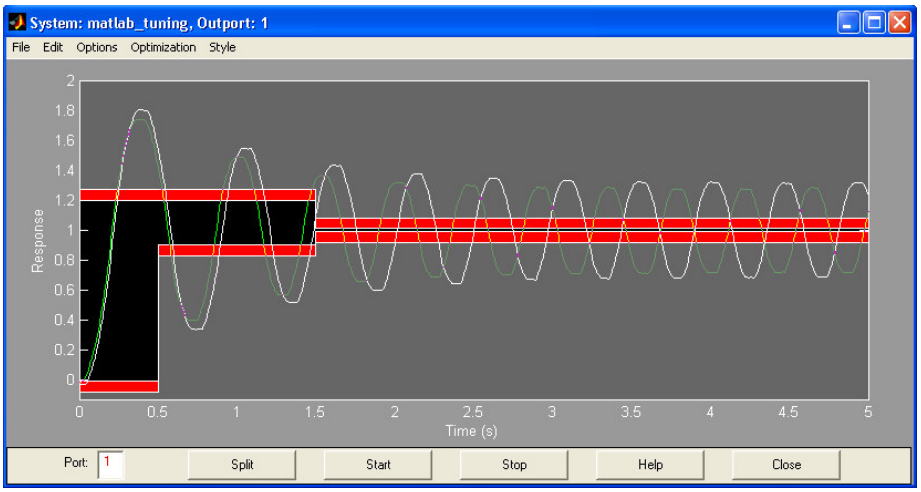


Figure A.3. Constraint figure window at OPTIONS(11)=3

```
CHG =
    1.0000
    1.0000
   -0.1000
   -0.1286

Kp=4.0192227  Ki=0.7622811  Kd=0.0022436  f=0.5321466
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
```

```
K>> return
Kp=3.0192227  Ki=1.7622811  Kd=0.0022436  f=0.5321466
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.0192227  Ki=0.7622811  Kd= -0.0977564  f=0.5321466
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.0192227  Ki=0.7622811  Kd=0.0022436   f=0.4035066
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg = 0.0061

     f-COUNT     MAX{g}          STEP    Procedures

        15      0.538281           1

OPTIONS(11)= 4

Step Length = 1

SD =
   -0.0358
   -0.0285
    0.0035
   -0.0096

Kp=2.9833888  Ki=0.7337870  Kd=0.0057774  f=0.5225005
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.4.

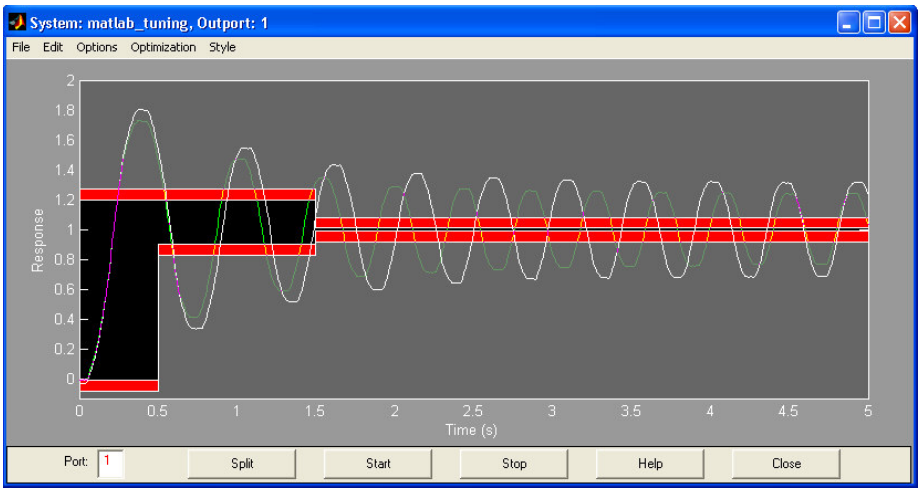


Figure A.4. Constraint figure window at OPTIONS(11)=4

```
CHG =
   -1.0000
   -1.0000
    0.4476
    1.0000

Kp=1.9833888  Ki=0.7337870  Kd=0.0057774  f=0.5225005
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.9833888  Ki= -0.2662130  Kd=0.0057774  f=0.5225005
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.9833888  Ki=0.7337870    Kd=0.4534203   f=0.5225005
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
```

```
K>> return
Kp=2.9833888  Ki=0.7337870  Kd=0.0057774  f=1.5225005
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg =  0.0060

     f-COUNT     MAX{g}          STEP    Procedures

        20     0.528516            1     Hessian modified

OPTIONS(11)=  5

Step Length = 1

SD =
   -0.6217
   -0.0182
    0.2163
   -0.3462

Kp=2.3617083  Ki=0.7155824  Kd=0.2220607  f=0.1763051
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.5.

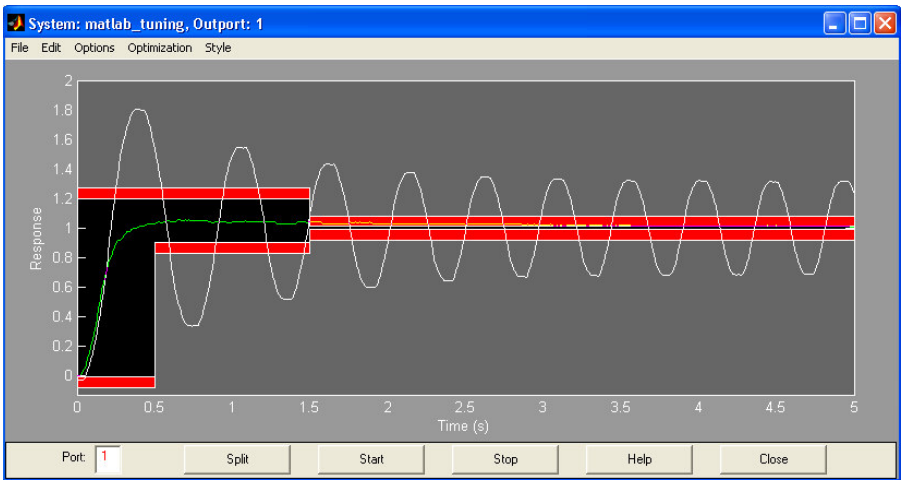


Figure A.5. Constraint figure window at OPTIONS(11)=5

```
CHG =
   -1.0000
   -1.0000
    0.1000
   -0.8920

Kp=1.3617083  Ki=0.7155824  Kd=0.2220607  f=0.1763051
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.3617083  Ki= -0.2844176  Kd=0.2220607  f=0.1763051
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.3617083  Ki=0.7155824  Kd=0.3220607  f=0.1763051
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.3617083  Ki=0.7155824  Kd=0.2220607  f= -0.7156958
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
```

```
K>> return

mg = -0.1463

     f-COUNT      MAX{g}          STEP     Procedures

        25      0.0300391            1

OPTIONS(11)= 6

Step Length = 1

SD =
   -0.0376
   -0.2163
   -0.0277
   -0.8333

Kp=2.3240594   Ki=0.4992767   Kd=0.1943178   f= -0.6569536
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.6.

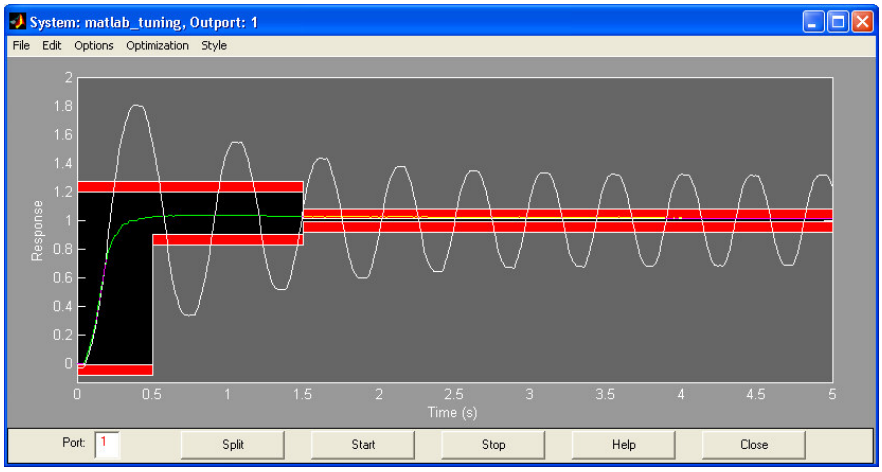


Figure A.6. Constraint figure window at OPTIONS(11)=6

```
CHG =
    -1
     1
    -1
    -1

Kp=1.3240594  Ki=0.4992768   Kd=0.1943179  f= -0.6569536
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.3240594  Ki=1.4992767  Kd=0.1943178  f= -0.6569536
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.3240594  Ki=0.4992767  Kd= -0.8056822  f= -0.6569536
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=2.3240594  Ki=0.4992767  Kd=0.1943178  f= -1.6569536
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg = 0.0203

 f-COUNT     MAX{g}         STEP    Procedures

    30     0.0202734          1    Hessian modified;infeasible

OPTIONS(11)= 7

Step Length = 1

SD =
    1.1833
```

```
    0.9429
   -0.0006
   -0.5535

Kp=3.5074001   Ki=1.4421329   Kd=0.1936892   f= -1.2104203
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.7.

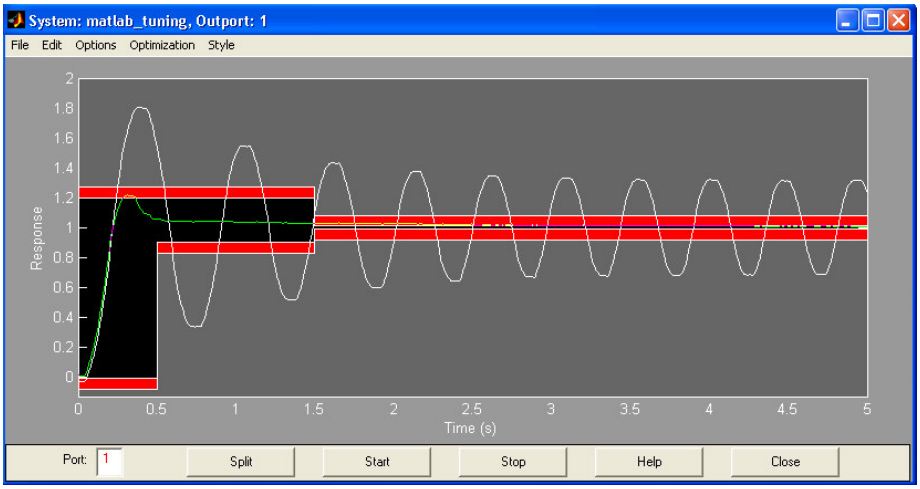


Figure A.7. Constraint figure window at OPTIONS(11)=7

```
CHG =
    1.0000
   -1.0000
   -0.1000
    1.0000

Kp=4.5074001   Ki=1.4421328   Kd=0.1936892   f= -1.2104203
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

```
Kp=3.5074001  Ki=0.4421328  Kd=0.1936892  f= -1.2104203
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.5074001  Ki=1.4421328  Kd=0.0936892  f= -1.2104203
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.5074001  Ki=1.4421328  Kd=0.1936892  f= -0.2104203
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return

mg = 0.0252

     f-COUNT     MAX{g}          STEP    Procedures

        35    0.0251562            1     infeasible

OPTIONS(11)= 8

Step Length = 1

SD =
   -0.0415
   -1.2005
    0.0061
   -0.7511

Kp=3.4658971  Ki=0.2416506  Kd=0.1997486  f= -1.9614853
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.8.

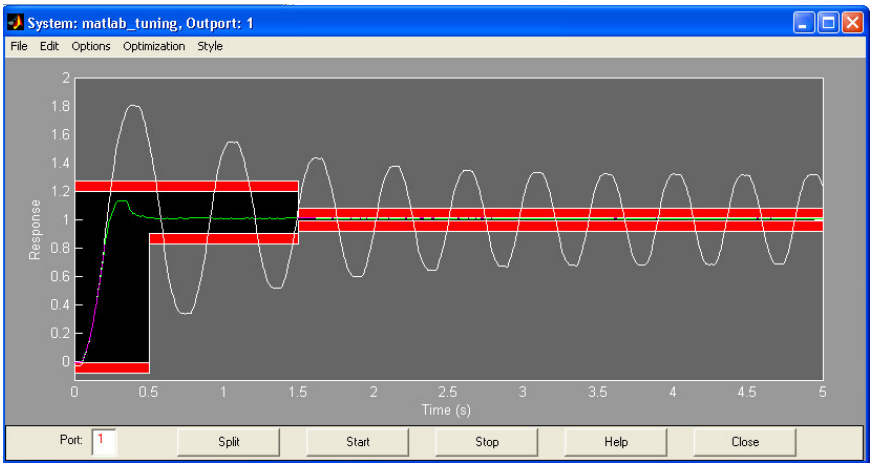


Figure A.8. Constraint figure window at OPTIONS(11)=8

```
CHG =
     -1
     -1
     -1
     -1

Kp=2.4658971   Ki=0.2416506   Kd=0.1997486   f= -1.9614853
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.4658971   Ki= -0.7583494   Kd=0.1997486   f= -1.9614853
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.4658971   Ki=0.2416506   Kd= -0.8002514   f= -1.9614853
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
Kp=3.4658971   Ki=0.2416506   Kd=0.1997486   f= -2.9614853
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
```

```
K>> return

mg = 7.4220e-004

     f-COUNT      MAX{g}          STEP     Procedures

        40     0.000742187          1      Hessian modified

OPTIONS(11)= 9

Step Length = 1

SD =
   -0.0024
   -0.0766
    0.0003
   -0.0000

Kp=3.4635016  Ki=0.1650299  Kd=0.2000949  f= -1.9614942
K>>
Model DC_motor_seda3 loaded
Model DC_motor_seda3 unloaded
K>> return
```

Constraint figure window appears as shown in Figure A.9.

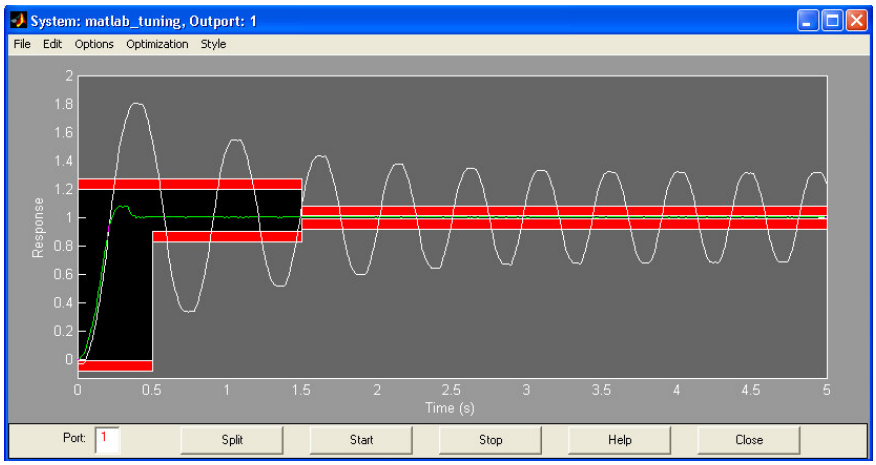
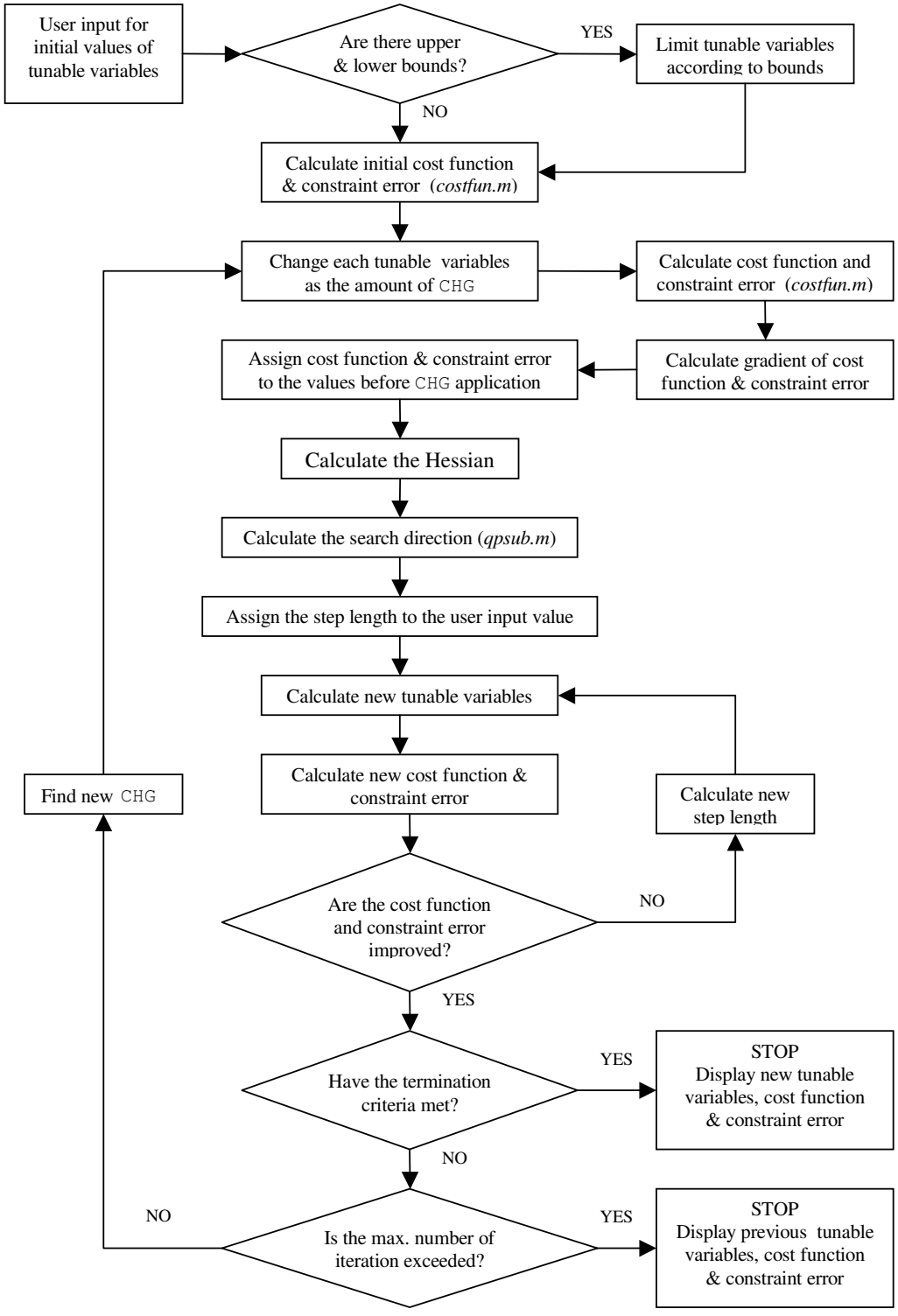


Figure A.9. Constraint figure window at OPTIONS(11)=9

```
mg = -0.0041

     f-COUNT     MAX{g}          STEP    Procedures

        45    -0.00414063          1    Hessian modified twice

   Optimization Converged Successfully

   Active Constraints:
      268
```

APPENDIX B

FLOWCHART



APPENDIX C

RELATED ORIGINAL OPTIMIZATION M-FILES

nlinopt.m

```
function nlinopt(sys,InitFlag)
%NLINOPT Runs the optimization algorithm.
%
%       NLINOPT(SYS,InitFlag) is called when the Start button is
%       pushed or when the Initial response menu item is selected.
%       It calls another routine to initialize any Monte Carlo
%       simulations.  It calls a routine to convert
ncdStruct.CnstrLB and
%       ncdStruct.CnstrUB into constraints used by the optimization
routine.
%       Finally, if InitFlag=1, it plots the initial response,
%       otherwise it calls the optimization routine.
%
%       See also MONTEVAR, CONVERTM, CONSTR, COSTFUN, GRADFUN.

%   Author(s): A. Potvin, 12-1-92
%              M. Yeddanapudi, Sept. 24, '96
%   Revised  : K. Subbarao 10-30-2001
%   Copyright 1990-2002 The MathWorks, Inc.
%   $Revision: 1.28 $
%   $Date: 2002/06/06 15:37:39 $

% OPT_STOP is global and must be empty to continue
global OPT_STOP;
if isempty(OPT_STOP),
   % Setting OPT_STOP to zero allows optimization
   % to continue and tells the dialog  boxes that
   % the parameters can no longer be changed.
   OPT_STOP = 0;
else
   fprintf('\nNLINOPT: First Click on the Stop push button to stop
optimization\n');
   fprintf('         that might be already running. If that is not
the case, try\n');
   fprintf('         setting OPT_STOP=[] to enable the Start push
button.\n');
   return;
end


% load the model into memory
loadCmd = [sys '([],[],[],0);'];
lasterr('');
```

```
evalin('base',loadCmd,'');
if ~isempty(lasterr),
   fprintf('\nNLINOPT: Error loading model: %s\n',sys);
   fprintf('%s\n',lasterr);
   OPT_STOP = [];
   return;
end

%% Declare Global
global ncdStruct;

% Want to keep one argument option so
% user can easily invoke optimization
if (nargin==1), InitFlag = 0; end

% MONTEVAR checks the uncertain parameters and
% initializes the variables:  SIMS,  UVARMATX,
% UVAREXT and UVDATA that are used in  COSTFUN

fprintf('\nProcessing uncertainty information.\n');
[sims,uvarmtx,uvarext,uvdata] = montevar;

if ((isempty(sims)) & (~InitFlag))
   error(['NLINOPT: No simulations constrained. ' ...
         'Check Uncertain Variable dialog box.']);
   OPT_STOP = [];
   return;
end

tstart = get_param(sys,'Start time');
if (isstr(tstart)), tstart = eval(tstart); end
tfinal = get_param(sys, 'Stop time');
if (isstr(tfinal)), tfinal = eval(tfinal); end

if InitFlag,
   if isempty(sims),
      fprintf('No simulations constrained. Plotting nominal.\n')
      sims = 1;
   end
   fprintf('Beginning simulations for initial response plots.\n')
else



   fprintf('Setting up call to optimization routine.\n')
end

if isempty(ncdStruct.Tdelta),
    ncdStruct.Tdelta = 1/100;
   %ncdStruct.Tdelta = (tfinal-tstart)/100;
end
timepts = [tstart:ncdStruct.Tdelta:tfinal]';
if timepts(end) < tfinal,
   timepts = [timepts; tfinal];
end

% NCD_OutPorts contains the port numbers of the NCD
% Masked Outport blocks. Only the entries of ncdStruct.CnstrLB
```

```
% and ncdStruct.CnstrUB that correspond to these outports will
% be used in the optimization.

NCD_OutPorts=[];
tmpcell = find_system(sys,'SearchDepth',1,'MaskType','NCD Outport');
NumNCDoutports = length(tmpcell(:,1));
for i=1:NumNCDoutports

NCD_OutPorts=[NCD_OutPorts;str2num(get_param(tmpcell{i},'Port'))];
end
tmpcell = find_system(sys,'SearchDepth',1,'BlockType','Outport');
NumOutPorts = length(tmpcell(:,1));

% Initialize the time out flag.
% tmpnum = Inf;
% tmpcell = find_system(sys,'MaskType','Sim TimeOut Block');
% for i=1:length(tmpcell(:,1))
%    tmpstr = get_param(tmpcell{i},'maskvaluestring');
%    dumstr = tmpstr(1:find(tmpstr == '|')-1);
%    dum = evalin('base',dumstr,'Inf');
%    if dum < tmpnum,
%       tmpnum = dum;
%       ncdStruct.TimeOutFlag = tmpstr(find(tmpstr == '|')+1:end);
%    end
% end
% The evalin('try','catch') in the above for loop ignores
% errors, so we need to clear lasterr, which may be nonempty.
% lasterr('');
% if isempty(ncdStruct.TimeOutFlag) | ~isstr(ncdStruct.TimeOutFlag),
%    ncdStruct.TimeOutFlag = '';
% end



% Check for constraint figures on the screen
% and initialize the handles for the initial
% and intermediate plots, in all the figres.

xdata = timepts(:,[ones(1,2*max(sims))]);
zdata = ones(size(xdata));

FigHndls = allchild(0);
FigNames = get(FigHndls,'Name');

prefix = ['System: ' sys ', Outport: '];
lnprefix = length(prefix)+1;

FigHndls = FigHndls(strmatch(prefix,FigNames));
FigNames = char(get(FigHndls,'Name'));

AllLines = [];
for indx=1:length(FigHndls)

   axs = get(FigHndls(indx),'CurrentAxes');

   % delete exisiting lines from the axs userdata
   delete(findobj(get(axs,'Children'),'Type','line'));
```

```
   % MCSlns is a vector of Monte Carlo simulation plot
   % handles. May create a couple of extra lines since
   % nominal, upper bound, and  lower bound plants are
   % not always constrained.
   % Note: MCSlns is never empty.

   MCSlns = line(xdata,xdata,zdata, ...
             'Parent',axs,'Color','green', ...
             'Visible','off','Clipping','on', ...
             'EraseMode','xor');
   AllLines = [AllLines; MCSlns];

   % The initial response is in MCSlns(1:max(sims))

   set(MCSlns(1:max(sims)), ...
      'Color','white','EraseMode','background');

   % Put the line handles in axis UserData
   set(axs,'UserData',MCSlns);

end


SimOptions = simset('SrcWorkSpace','base', ...
                    'DstWorkSpace','current', ...
                    'OutputPoints','specified');

% simulate the SL model and plot the initial response
if initresp(sys,timepts,sims,uvarmtx,uvarext,uvdata,SimOptions) ==
1,
   % error in initresp, clean up and bail out
   OPT_STOP = [];
   return;
end
fprintf('Done plotting the initial response.\n')

if (InitFlag == 0)

   % Lot of processing to do before we begin optimization

   % first parse ncdStruct.TvarStr and setup the following variables
   % tvarmtx: str2mat2(ncdStruct.TvarStr)
   % tvarvec: vectorized tunable variables
   % tvarext: vector containig the sizes of the tuneable variables

   atindx = 0;
   tvarext = [];
   tvarmtx = ''; tvarvec = [];

   [tvarmtx,error_str] = str2mat2(ncdStruct.TvarStr);
   if (~isempty(error_str))
      fprintf(['\nNLINOPT: ' error_str]);
      fprintf('\n          error parsing ncdStruct.TvarStr');
      fprintf('\n          cannot start optimization.\n');
      OPT_STOP = [];
      return;
   end
```

```
   lasterr('');
   for i=1:size(tvarmtx,1)
      tmpvar = evalin('base',tvarmtx(i,:),['''' tvarmtx(i,:) '''']);
      if ~isempty(lasterr),
         fprintf(lasterr);
         fprintf(['\nNLINOPT: error accessing: ' tmpvar ' in the
base workspace']);
         fprintf(['\n         cannot start optimization.\n']);
         OPT_STOP = [];
         return;
      end



      tmpint = prod(size(tmpvar));
      if (tmpint == 0)
         fprintf('\nNLINOPT: %s is empty',tmpvar);
         fprintf('\n         cannot start optimization.\n');
         OPT_STOP = [];
         return;
      end
      tvarvec = [tvarvec;tmpvar(:)];
      tvarext(i,1) = atindx+tmpint;
      atindx = tvarext(i,1);
   end

   % ncdStruct.TvarMtx is required in TVARSET
   ncdStruct.TvarMtx = tvarmtx;

   % Done with ncdStruct.TvarStr.

   % Now process the lower bounds in ncdStruct.TvlbStr
   % if successful tvlbvec will contain the
   % vectorized values of the lower bounds.

   tvlbmtx = ''; tvlbvec = [];

   if ~isstr(ncdStruct.TvlbStr),
      if ~isempty(ncdStruct.TvlbStr),
         fprintf('\nNLINOPT: ncdStruct.TvlbStr is not a string!');
         fprintf('\n         setting ncdStruct.TvlbStr to empty');
         fprintf('\n         proceeding without lower bounds.\n')
      end
      ncdStruct.TvlbStr = '';
   elseif ~isempty(ncdStruct.TvlbStr),
      [tvlbmtx,error_str] = minipars(ncdStruct.TvlbStr);
      if ~isempty(error_str),
         fprintf('\nNLINOPT: error parsing the lower bound string:
%s',ncdStruct.TvlbStr);
         fprintf('\n         proceeding without lower bounds.\n')
         tvlbmatx = '';
      elseif size(tvarmtx,1) ~= size(tvlbmtx,1),
         fprintf('\nNLINOPT: sizes of ncdStruct.TvarStr and
ncdStruct.TvlbStr should be equal');
         fprintf('\n         proceeding without lower bounds.\n')
         tvlbmatx = '';
      else
```

```
         atindx = 1;
         lasterr('');
         for i=1:size(tvarmtx,1)
            siz = tvarext(i)-atindx+1;
            tvlbtmp = evalin('base',tvlbmtx(i,:),['''' tvlbmtx(i,:)
''''']);
            if ~isempty(lasterr),
               fprintf(lasterr); lasterr('');
               fprintf('\nNLINOPT: error evaluating %s in the base
workspace',tvlbtmp);
               fprintf('\n         setting the lower bound of %s to
-Inf\n',deblank(tvarmtx(i,:)));
               tvlbtmp = repmat(-Inf,[siz 1]);
            elseif isempty(tvlbtmp),
               fprintf('\nNLINOPT: the lower bound of %s -> %s is
empty', ...

deblank(tvarmtx(i,:)),deblank(tvlbmtx(i,:)));
               fprintf('\n         setting the lower bound of %s to
+Inf\n',deblank(tvarmtx(i,:)));
               tvlbtmp = repmat(-Inf,[siz 1]);
            elseif prod(size(tvlbtmp)) ~= siz,
               fprintf('\nNLINOPT: size of tunable variable %s and
its lower bound %s are inconsistent', ...

deblank(tvarmtx(i,:)),deblank(tvlbmtx(i,:)));
               fprintf('\n         setting the lower bound of %s to
-Inf\n',deblank(tvarmtx(i,:)));
               tvlbtmp = repmat(-Inf,[siz 1]);
            end
            tvlbvec = [tvlbvec;tvlbtmp(:)];
            atindx = tvarext(i,1)+1;
         end
      end
   end

   % Done with ncdStruct.TvlbStr.

   % Now process the upper bounds in ncdStruct.TvubStr
   % if successful tvubvec will contain the
   % vectorized values of the upper bounds.

   tvubmtx = ''; tvubvec = [];

   if ~isstr(ncdStruct.TvubStr),
      if ~isempty(ncdStruct.TvubStr),
         fprintf('\nNLINOPT: ncdStruct.TvubStr is not a string!');
         fprintf('\n         setting ncdStruct.TvubStr to empty
and');



      fprintf('\n         proceeding without upper bounds.\n')
      end
```

```
      ncdStruct.TvubStr = '';
   elseif ~isempty(ncdStruct.TvubStr),
      [tvubmtx,error_str] = minipars(ncdStruct.TvubStr);
      if ~isempty(error_str),
         fprintf('\nNLINOPT: error parsing the upper bound string:
%s',ncdStruct.TvubStr);
         fprintf('\n         proceeding without upper bounds.\n')
         tvubmatx = '';
      elseif size(tvarmtx,1) ~= size(tvubmtx,1),
         fprintf('\nNLINOPT: sizes of ncdStruct.TvarStr and
ncdStruct.TvubStr should be equal');
         fprintf('\n         proceeding without upper bounds.\n')
         tvubmatx = '';
      else
         atindx = 1;
         lasterr('');
         for i=1:size(tvarmtx,1)
            siz = tvarext(i)-atindx+1;
            tvubtmp = evalin('base',tvubmtx(i,:),['''' tvubmtx(i,:)
''''']);
            if ~isempty(lasterr),
               fprintf(lasterr); lasterr('');
               fprintf('\nNLINOPT: error evaluating %s in the base
workspace',tvubtmp);
               fprintf('\n         setting the upper bound of %s to
+Inf\n',deblank(tvarmtx(i,:)));
               tvubtmp = repmat(Inf,[siz 1]);
            elseif isempty(tvubtmp),
               fprintf('\nNLINOPT: the upper bound of %s -> %s is
empty', ...

deblank(tvarmtx(i,:)),deblank(tvubmtx(i,:)));
               fprintf('\n         setting the upper bound of %s to
+Inf\n',deblank(tvarmtx(i,:)));
               tvubtmp = repmat(Inf,[siz 1]);
            elseif prod(size(tvubtmp)) ~= siz,
               fprintf('\nNLINOPT: size of tunable variable %s and
its upper bound %s are inconsistent', ...

deblank(tvarmtx(i,:)),deblank(tvubmtx(i,:)));
               fprintf('\n         setting the upper bound of %s to
+Inf\n',deblank(tvarmtx(i,:)));
               tvubtmp = repmat(Inf,[siz 1]);
            end
            tvubvec = [tvubvec;tvubtmp(:)];
            atindx = tvarext(i,1)+1;


         end
      end
   end

   % Done processing ncdStruct.TvarStr, ncdStruct.TvlbStr and
ncdStruct.TvubStr.

   % Begin processing the constraint bounds defined
   % in the global bound matrices ncdStruct.CnstrLB and
ncdStruct.CnstrUB
```

```
   % Convert ncdStruct.CnstrLB  and ncdStruct.CnstrUB to Mu and Ml,
where
   %     Mu - upper bound constraints
   %     Ml - lower bound constraints

   lb = ncdStruct.CnstrLB;

   irow = lb(ones(length(NCD_OutPorts),1),:);
   icol = NCD_OutPorts(:,ones(size(lb,2),1));
   indx = find(sum(irow == icol) == 0);

   if ~isempty(indx),
      fprintf('\nIgnoring the following lower constraints\n');
      fprintf('     in ncdStruct.CnstrLB for Non-NCD Masked
Outports\n');
      lb(:,indx)
      lb(:,indx) = [];
   end

   ub = ncdStruct.CnstrUB;
   irow = ub(ones(length(NCD_OutPorts),1),:);
   icol = NCD_OutPorts(:,ones(size(ub,2),1));
   indx = find(sum(irow == icol) == 0);
   if ~isempty(indx),
      fprintf('\nIgnoring the following upper constraints\n');
      fprintf('     in ncdStruct.CnstrLB for Non-NCD Masked
Outports\n');
      ub(:,indx)
      ub(:,indx) = [];
   end

   Ml = convertm(lb,timepts);
   Mu = convertm(ub,timepts);

   % Determine how many constraints are to be met

   if ((isempty(Mu))&(isempty(Ml)))
      RunFlag = 0;
      fprintf('\nNLINOPT from CONVERTM: no constraints generated');


      fprintf('\n                     start Optimization
ignored\n');
      OPT_STOP = [];
      return;
   end

   % Tell user start and stop times and how many constraints are to
be met

   fprintf(['Start time: ' num2str(tstart) '\t Stop time: '
num2str(tfinal) '.\n']);
   fprintf(['There are ' int2str(size(Mu,1)+size(Ml,1)) ...
                  ' constraints to be met in each simulation.\n']);
   fprintf(['There are ' int2str(length(tvarvec)) ' tunable
variables.\n']);
   fprintf(['There are ' int2str(length(sims)) ' simulations per
cost function call.\n']);
```

```
   if (ncdStruct.GradFlag == 1)
      % create model for  simulating the actual
      % and the perturbed models simultaneously

      gradsys = strrep(tempname,tempdir,'');
      fprintf(['Creating a temporary SL model ' gradsys ' for
computing gradients...\n']);

      lasterr('');
      eval(['new_system(''' gradsys ''')'],'');
      if ~isempty(lasterr),
         % may be gradsys is already open, try close_system.

         error_str = lasterr; lasterr('');
         eval(['close_system(''' gradsys ''',0)'],'');
         if ~isempty(lasterr),
            % rats! even close_system caused an error, give up.

            fprintf('\nNLINOPT: error creating %s\n',gradsys);
            fprintf('%s\n',error_str);
            OPT_STOP = [];
            return;
         else
            % close_system worked, so try new_system once again.
            % no need to set lasterr(''), because it is still empty

            eval(['new_system(''' gradsys ''')'],'');
            if ~isempty(lasterr),
               % even after close_system, new_system still
               % results in an error, this time error out.



               fprintf('\nNLINOPT: error creating %s\n',gradsys);
               fprintf('%s\n',lasterr);
               OPT_STOP = [];
               return;
            end
         end
      end

      % create original and perturbed copies of the
      % tunable variables which will be used in gradfun

      for j=1:size(tvarmtx,1)
         varname = deblank(tvarmtx(j,:));
         evalin('base',[varname '_original  = ' varname ';']);
         evalin('base',[varname '_perturbed = ' varname ';']);
      end

      eval(['copymdl(''' sys ''',''' gradsys ''')'],'');
      if ~isempty(lasterr),
         fprintf('\nNLINOPT: error while copying into
%s\n',gradsys);
         fprintf('%s\n',lasterr);
         close_system(gradsys,0);
         OPT_STOP = [];
```

```
         return;
      end

      % Make sure gradsys is properly loaded in memory.
      loadCmd = [gradsys '([],[],[],0);'];
      lasterr('');
      evalin('base',loadCmd,'');
      if ~isempty(lasterr),
         fprintf('\nNLINOPT: Error loading model: %s\n',gradsys);
         fprintf('%s\n',lasterr);
         close_system(gradsys,0);
         OPT_STOP = [];
         return;
      end

      fprintf(['Creating simulink model ' gradsys ' for
gradients...Done\n']);

   end

   % initialize the cost: gamma




   gamma = 1;
   tvarvec = [tvarvec; gamma];

   % Call optimization routine.

   if isempty(ncdStruct.OptmOptns),
      % Default optimization options
      ncdStruct.OptmOptns = [1 0.001 0.001];
   end

   options(1) = ncdStruct.OptmOptns(1); % display on/off
   options(2) = ncdStruct.OptmOptns(2); % variable tolerance
   options(3) = ncdStruct.OptmOptns(2); % function tolerance
   options(4) = ncdStruct.OptmOptns(3); % constraint tolerance
   options(7) = 1; % Line search modified for slack variable

   offset = NumOutPorts*length(timepts);

   if (ncdStruct.GradFlag == 1)

      % Debug mode to check the gradients and open gradsys
      % To enable this mode, declare NCDdebuggingON =1 in base
      % workspace.

      % REMARK: lasterr after the eval('try this','otherwise')
      % is needed so that we can ignore errors and reset lasterr
      % to empty, in case 'try this' did not work. This usage
      % is a bit different from the eval('try','catch')

      dum = evalin('base','NCDdebuggingON','0'); lasterr('');
      if dum == 1,
         disp('Will stop in graderr to check the gradients.');
         path2graderr=which('graderr','simcnstr');
```

```
         evalin('base',['dbstop in ' path2graderr]);
         options(9) = 1; % gradient check
         open_system(gradsys);
      end

      % save the name of gradsys in ncdStruct for access in gradfun
      ncdStruct.GradSysName = gradsys;

      x =
simcnstr('ncdtoolbox','costfun',tvarvec,options,tvlbvec,tvubvec,...

'gradfun',tvarmtx,tvarext,sys,timepts,Mu,Ml,offset,sims, ...
                    uvarmtx,uvarext,uvdata,SimOptions);



     close_system(gradsys,0);

      % clean up the workspace

      for j=1:size(tvarmtx,1)
         varname = deblank(tvarmtx(j,:));
         evalin('base',['clear ' varname '_original;']);
         evalin('base',['clear ' varname '_perturbed;']);
      end

   else
      x =
simcnstr('ncdtoolbox','costfun',tvarvec,options,tvlbvec,tvubvec,...
 '',tvarmtx,tvarext,sys,timepts,Mu,Ml,offset,sims, ...
   uvarmtx,uvarext,uvdata,SimOptions);
   end
end

% Reset plant to nominal

atindx = 1;
for i=1:size(uvarmtx,1)
   siz = [atindx:uvarext(i,1)]';
   assignin('base','NCD_tmp',uvdata(siz,1));
   evalin('base',[uvarmtx(i,:) '(:) = NCD_tmp;']);
   evalin('base','NCD_tmp = [];');
   atindx = uvarext(i,1)+1;
end
evalin('base','clear NCD_tmp;');

% Set the EraseMode of all plotted lines to normal

eval('set(AllLines,''EraseMode'',''normal'')','');lasterr('');

% Setting OPT_STOP to be empty enables dialogs
fprintf('\n');

OPT_STOP = [];

% end nlinopt
```

costfun.m

```
function [CostFunction,ConstraintError] =
costfun(tvarvec,tvarmtx,tvarext,sysname, ...
timepts,Mu,Ml,offset,sims,uvarmtx,uvarext,uvdata,simoptions)


%COSTFUN Cost function for NCD Blockset optimization.
%
%       [CostFunction,ConstraintError] = COSTFUN(TVARVEC, ...
%            VARMTX,TVAREXT,SYSNAME,TIMEPTS,MU,ML,OFFSET, ...
%            SIMS,UVARMTX,UVAREXT,UVDATA,SIMOPTIONS)
%       calculates the CostFunction and ConstraintError given:
%
%       Inputs:
%          TVARVEC    -- vectorized tunable parameters at this
iteration
%          DTUNEVAR   -- suggested perturbations to the tunable
parameters
%          SYSNAME    -- SIMULINK system name
%          TVARMTX    -- tunable parameter names formatted as a
padded string matrix
%          TVAREXT    -- vector of (vectorized) tunable parameter
dimensions
%          TIMEPTS    -- time vector: [tstart:tdelta:tfinal]
%          MU         -- [<vectorized output index> <upper
constraint> <weight>]
%          ML         -- [<vectorized output index> <lower
constraint> <weight>]
%          OFFSET     -- NumOutPorts * length(TIMEPTS)
%          SIMS       -- vector of simulations to be constrained
%          UVARMTX    -- uncertain parameter names formatted as a
padded string matrix
%          UVAREXT    -- vector of (vectorized) tunable parameter
dimensions
%          UVDATA     -- matrix of uncertain parameter values in
each simulation
%          SIMOPTIONS -- simulation options
%
%       See also NLINOPT.

%   Author(s): A. Potvin, 12-1-92
%              M. Yeddanapudi, Sept 16, '96
%   Copyright 1990-2002 The MathWorks, Inc.
%   $Revision: 1.16 $
%   $Date: 2002/03/22 14:11:47 $


% Declare globals

global OPT_STOP;
global OPT_STEP;
global ncdStruct;
```

```
% ncdStruct.TimeOutFlag holds the name of the
% global variable that is set by the simstop block.
% We use eval('try this','otherwise') in case
% there is a simstop block sysname SL model
% eval(['global ' ncdStruct.TimeOutFlag],''); lasterr('');

% Recover tunable variables  from tvarvec and
% assign them to the appropriate tunable
% variables in the  base workspace.  No error
% checking needed here, because in NLINOPT we
% made sure everything was ok.

atindx = 1;
for i=1:size(tvarmtx,1)
   siz = [atindx:tvarext(i,1)]';
   assignin('base','NCD_tmp',tvarvec(siz,1));
   evalin('base',[tvarmtx(i,:) '(:) = NCD_tmp;']);
   atindx = tvarext(i,1)+1;
end


% Initialize constraint vector and output CostFunction


ConstraintError = [];
if ~isempty(tvarvec),
   CostFunction = tvarvec(end);
   if ncdStruct.CostFlag == 1,
      CostFunction = max(CostFunction,-1.0e-8);
   end
end


% Set up backward for loop


for simindx=sims

   % Try to better process button and break
   % out of  the loop in case OPT_STOP == 1

   drawnow;

   if OPT_STOP == 1,
      fprintf('.');
      CostFunction = 1e10;
      ConstraintError =
repmat(CostFunction,length(sims)*(size(Mu,1)+size(Ml,1)),1);
      return;


   end


   % At each Monte Carlo run  set the uncertain
   % parameters to the  values specified in the
```

```
   % simindx^th column of uvdata which has been
   % initialized in MONTEVAR


   atindx = 1;
   for i=1:size(uvarmtx,1)
      siz = [atindx:uvarext(i,1)]';
      assignin('base','NCDtmp',uvdata(siz,simindx));
      evalin('base',[uvarmtx(i,:) '(:) = NCDtmp;']);
      atindx = uvarext(i,1)+1;
   end


   % Simulate the  model and abort
   % if any errors are encountered



   SimString = ['sim(''' sysname ''',timepts,simoptions);'];
   lasterr('');
   eval(['[SimTime,SimState,InterpOut]=' SimString],' ');



   if ~isempty(lasterr),
      fprintf('\n      SL Error Message: %s\n     ',lasterr');
      fprintf('\n      COSTFUN: Error simulating %s',sysname);
      CostFunction = 1e10;
      ConstraintError =
repmat(CostFunction,length(sims)*(size(Mu,1)+size(Ml,1)),1);
      if OPT_STEP == 2,
         fprintf('\n            Error occured during line search
...');
         fprintf('\n            Continuing Optimization ...\n');
      else
         fprintf('\n            Error occured during a major update
...');
         fprintf('\n            Stopping Optimization...\n');
         OPT_STOP = 1;
      end
      return;
   end


   % code to time out a simulation.
   % use eval('try this','otherwise')
   % TimeOutFlag = eval(ncdStruct.TimeOutFlag,'[]'); lasterr('');
   % if isequal(TimeOutFlag,1),
   %    fprintf('\nCOSTFUN: Simulation Timed Out');
   %    CostFunction = 1e10;
   %    ConstraintError =
repmat(CostFunction,length(sims)*(size(Mu,1)+size(Ml,1)),1);
   %    if OPT_STEP == 2,
   %       fprintf(' during line search ...');
   %       fprintf('\n            Continuing Optimization ...\n');
   %    else
   %       fprintf(' during a major update ...');
   %       fprintf('\n            Stopping Optimization ...\n');
```

```
%       OPT_STOP = 1;
%    end
% end

if (ncdStruct.GradFlag == 0) & (OPT_STEP == 1),

   %%% Update The Plots in the NCD Figure Windows %%%

   % get the handles and names of all the open constraint
   % figure windows and update the intermediate response plots

   fighndls = allchild(0);
   fignames = char(get(fighndls,'Name'));
   prefix = ['System: ' sysname ', Outport: '];
   lnprefix = length(prefix)+1;
   fighndls = fighndls(strmatch(prefix,fignames));
   fignames = char(get(fighndls,'Name'));

   for figindx=1:length(fighndls)

      portnum = str2num(fignames(figindx,lnprefix:end));
      axs = get(fighndls(figindx),'CurrentAxes');
      MCSlns = get(axs,'UserData');

      if (~isempty(MCSlns))
         ln = MCSlns(max(sims)+simindx);
         set(ln,'YData',InterpOut(:,portnum));


         if (strcmp(get(ln,'Visible'),'off'))
            set(ln,'Visible','on');
         end
      end
   end


   drawnow;


   if (OPT_STOP)
      fprintf('.');
      CostFunction = 1e10;
      ConstraintError =
repmat(CostFunction,length(sims)*(size(Mu,1)+size(Ml,1)),1);
      return;
   end


   % Form ConstraintError


   if (~isempty(Mu))
      ConstraintError = [ConstraintError; ...
         InterpOut(Mu(:,1))-Mu(:,2)-Mu(:,3)*CostFunction];
   end

   if (~isempty(Ml))
```

```
      ConstraintError = [ConstraintError; ...
         Ml(:,2)-InterpOut(Ml(:,1))-Ml(:,3)*CostFunction];
   end


   % Remark: This may abstract away too much information.
   %         For example, user  may desire more info  on
   %         limiting constraints.

end

% end costfun
```

nlconst.m

```
function
[x,OPTIONS,lambda,HESS]=nlconst(FUNfcn,x,OPTIONS,VLB,VUB,GRADfcn,...
                                          varargin)
%NLCONST Helper function for SIMCNSTR.
%   NLCONST is a helper function for SIMCNSTR to find the
constrained minimum
%   of a function of several variables.
%
%
[X,OPTIONS,LAMBDA,HESS]=NLCONST('FUN',X0,OPTIONS,VLB,VUB,'GRADFUN',.
..
%   varargin{:}) starts at X0 and finds a constrained minimum to the
function
%   which is described in FUN. FUN is a four element cell array set
up by
%   PREFCNCHK.  It contains the call to the objective/constraint
function, the
%   gradients of the objective/constraint functions, the calling
type (used by
%   OPTEVAL), and the calling function name.

%   Copyright 1990-2002 The MathWorks, Inc.
%   $Revision: 1.10 $
%   Andy Grace 7-9-90, Mary Ann Branch 9-30-96.

%   Calls OPTEVAL.


% Expectations: GRADfcn must be [] if it does not exist.
global OPT_STOP OPT_STEP;
OPT_STEP = 1;
OPT_STOP = 0;
% Initialize so if OPT_STOP these have values
lambda = []; HESS = [];

% Set up parameters.
XOUT=x(:);

VLB=VLB(:); lenvlb=length(VLB);
VUB=VUB(:); lenvub=length(VUB);
bestf = Inf;

nvars = length(XOUT);

OPTIONS(10)=1;
OPTIONS(11)=1;


CHG = 1e-7*abs(XOUT)+1e-7*ones(nvars,1);

if lenvlb*lenvlb>0
      if any(VLB( (1:lenvub)' ) > VUB), error('Bounds Infeasible'),
end
```

```
end
for i=1:lenvlb
       if lenvlb>0,if XOUT(i)<VLB(i),XOUT(i)=VLB(i)+1e-4; end,end
end
for i=1:lenvub
       if lenvub>0,if XOUT(i)>VUB(i),XOUT(i)=VUB(i);CHG(i)=-
CHG(i);end,end
end

% Used for semi-infinite optimization:
s = nan; POINT =[]; NEWLAMBDA =[]; LAMBDA = []; NPOINT =[]; FLAG =
2;
OLDLAMBDA = [];

sizep = length(OPTIONS);
OPTIONS = foptions(OPTIONS);
if lenvlb*lenvlb>0
      if any(VLB((1:lenvub)') > VUB), error('Bounds Infeasible'),
end
end
for i=1:lenvlb
       if lenvlb>0,if XOUT(i)<VLB(i),XOUT(i)=VLB(i)+eps; end,end
end
OPTIONS(18)=1;
if OPTIONS(1)>0
   if OPTIONS(7)==1
        disp('')
        disp('f-COUNT     MAX{g}         STEP  Procedures');
   else
    disp('')
        disp('f-COUNT   FUNCTION       MAX{g}         STEP
Procedures');
   end
end
HESS=eye(nvars,nvars);
if sizep<1 |OPTIONS(14)==0, OPTIONS(14)=nvars*100;end

x(:) = XOUT;  % Set x to have user expected size
% Compute the objective function and constraints
if strcmp(FUNfcn{4},'ncdtoolbox')
  [f,g] = feval(FUNfcn{1},x,varargin{:});
else
  [f,g,msg] = opteval(x,FUNfcn,varargin{:});


  error(msg);
  g = g(:);
end
if isempty(f)
  error('FUN must return a non-empty objective function.')
end
ncstr = length(g);

GNEW=1e8*CHG;
% Evaluate gradients and check size
if isempty(GRADfcn)
  analytic_gradient = 0;
else
```

```
  analytic_gradient = 1;
  if strcmp(FUNfcn{4},'ncdtoolbox')
     [gf_user,gg_user,OPTIONS] =
feval(GRADfcn{1},x,GNEW,OPTIONS,varargin{:});
     gf_user = gf_user(:);
  else
     [gf_user,gg_user,msg] = opteval(x,GRADfcn,varargin{:});
     error(msg);
     gf_user = gf_user(:);
  end
  % Both might evaluate to empty when expression syntax is used
  if isempty(gf_user) & isempty(gg_user)
    analytic_gradient = 0;
  else  % Either gf or gg is defined
     if length(gf_user) ~= nvars
       error('The objective gradient is the wrong size.')
     end
     if isempty(gg_user) & isempty(g)
       % Make gg compatible
       gg = g';
     else % Check size of gg
       [ggrow, ggcol] = size(gg_user);
       if ggrow ~= nvars
          error('The constraint gradient has the wrong number of
rows.')
       end
       if ggcol ~= ncstr
          error('The constraint gradient has the wrong number of
columns.')
       end
     end % isempty(gg_user)
  end % isempty(gf_user) & isempty(gg_user)
end % isempty(GRADfcn)



OLDX=XOUT;
OLDG=g;
OLDgf=zeros(nvars,1);
gf=zeros(nvars,1);
OLDAN=zeros(ncstr,nvars);
LAMBDA=zeros(ncstr,1);


%-------------------------------- Main Loop -----------------------
-----
status = 0;
first_iter=1;
while status ~= 1

%--------------- GRADIENTS ---------------

    if ~analytic_gradient | OPTIONS(9)
% Finite Difference gradients (even if just checking analytical)
        POINT = NPOINT;
        oldf = f;
        oldg = g;
        ncstr = length(g);
```

```
        FLAG = 0; % For semi-infinite
        gg = zeros(nvars, ncstr);  % For semi-infinite
% Try to make the finite differences equal to 1e-8.
        CHG = -1e-8./(GNEW+eps);
        CHG =
sign(CHG+eps).*min(max(abs(CHG),OPTIONS(16)),OPTIONS(17));
        OPT_STEP = 1;
        for gcnt=1:nvars
            if gcnt == nvars,
              FLAG = -1;
            end
            temp = XOUT(gcnt);
            XOUT(gcnt)= temp + CHG(gcnt);
            x(:) =XOUT;

            if strcmp(FUNfcn{4},'ncdtoolbox')
              [f,g] = feval(FUNfcn{1},x,varargin{:});
            else
              [f,g,msg] = opteval(x,FUNfcn,varargin{:});
              error(msg);
              g = g(:);
            end
            OPT_STEP = 0;



            if OPT_STOP
               break;
            end
            % Next line used for problems with varying number of
constraints
            if ncstr~=length(g),
               diff=length(g);
               g=v2sort(oldg,g);
            end

            gf(gcnt,1) = (f-oldf)/CHG(gcnt);
            if ~isempty(g)
              gg(gcnt,:) = (g - oldg)'/CHG(gcnt);
            end
            XOUT(gcnt) = temp;
            if OPT_STOP
                break;
            end
        end % for
        if OPT_STOP
            break;
        end

% Gradient check
        if OPTIONS(9) == 1 & analytic_gradient
            gfFD = gf;
            ggFD = gg;
            gg = gg_user;
            gf = gf_user;

            disp('Function derivative')
            if isa(GRADfcn{1},'inline')
```

```
              graderr(gfFD, gf, formula(GRADfcn{1}));
            else
              graderr(gfFD, gf, GRADfcn{1});
            end
            if ~isempty(gg)
              disp('Constraint derivative')
              if isa(GRADfcn{3},'inline')
                graderr(ggFD, gg, formula(GRADfcn{3}));
              else
                graderr(ggFD, gg, GRADfcn{3});
              end
            end
            OPTIONS(9) = 0;
        end % OPTIONS(9) == 1 & analytic_gradient


        FLAG = 1; % For semi-infinite
        OPTIONS(10) = OPTIONS(10) + nvars;
        f=oldf;
        g=oldg;
    else % analytic_gradient & options(9)=0
    % User-supplied gradients
        % gf and gg already computed first time through loop
        if ~first_iter
          gg = zeros(nvars, ncstr);
          if strcmp(FUNfcn{4},'ncdtoolbox')
             [gf,gg,OPTIONS] =
feval(GRADfcn{1},x,GNEW,OPTIONS,varargin{:});
          else
             [gf,gg,msg] = opteval(x,GRADfcn,varargin{:});
             error(msg);
          end
          gf = gf(:);
          if isempty(gg) & isempty(g)
            gg = g';
          end
        else
          % First time through loop
          gg = gg_user;
          gf = gf_user;
          first_iter=0;
        end

        if OPT_STOP
           break;
         end

    end  % if ~analytic_gradient | OPTIONS(9)
    AN=gg';
    how='';
    OPT_STEP = 2;

%------------ SEARCH DIRECTION --------------

    for i=1:OPTIONS(13)
        schg=AN(i,:)*gf;
        if schg>0
            AN(i,:)=-AN(i,:);
```

```
            g(i)=-g(i);
        end



    end

    if OPTIONS(11)>1  % Check for first call
% For equality constraints make gradient face in
% opposite direction to function gradient.
        if OPTIONS(7)~=5,
            NEWLAMBDA=LAMBDA;
        end
        [ma,na] = size(AN);
        GNEW=gf+AN'*NEWLAMBDA;
        GOLD=OLDgf+OLDAN'*LAMBDA;
        YL=GNEW-GOLD;
        sdiff=XOUT-OLDX;
% Make sure Hessian is positive definite in update.
        if YL'*sdiff<OPTIONS(18)^2*1e-3
            while YL'*sdiff<-1e-5
                [YMAX,YIND]=min(YL.*sdiff);
                YL(YIND)=YL(YIND)/2;
            end
            if YL'*sdiff < (eps*norm(HESS,'fro'));
                how=' Hessian modified twice';
                FACTOR=AN'*g - OLDAN'*OLDG;
                FACTOR=FACTOR.*(sdiff.*FACTOR>0).*(YL.*sdiff<=eps);
                WT=1e-2;
                if max(abs(FACTOR))==0; FACTOR=1e-5*sign(sdiff); end
                while YL'*sdiff < (eps*norm(HESS,'fro')) & WT <
1/eps
                    YL=YL+WT*FACTOR;
                    WT=WT*2;
                end
             else
                    how=' Hessian modified';
            end
        end

%--------- Perform BFGS Update If YL'S Is Positive ---------
        if YL'*sdiff>eps
            HESS=HESS+(YL*YL')/(YL'*sdiff)-
(HESS*sdiff*sdiff'*HESS')/(sdiff'*HESS*sdiff);
% BFGS Update using Cholesky factorization  of Gill, Murray and
Wright.
% In practice this was less robust than above method and slower.
%   R=chol(HESS);
%   s2=R*S; y=R'\YL;
%   W=eye(nvars,nvars)-(s2'*s2)\(s2*s2') + (y'*s2)\(y*y');
%   HESS=R'*W*R;



   else
            how=' Hessian not updated';
        end
```

```
    else % First call
          OLDLAMBDA=(eps+gf'*gf)*ones(ncstr,1)./(sum(AN'.*AN')'+eps)
;
    end % if OPTIONS(11)>1
    OPTIONS(11)=OPTIONS(11)+1;

    LOLD=LAMBDA;
    OLDAN=AN;
    OLDgf=gf;
    OLDG=g;
    OLDF=f;
    OLDX=XOUT;
    XN=zeros(nvars,1);
    if (OPTIONS(7)>0&OPTIONS(7)<5)
    % Minimax and attgoal problems have special Hessian:
        HESS(nvars,1:nvars)=zeros(1,nvars);
        HESS(1:nvars,nvars)=zeros(nvars,1);
        HESS(nvars,nvars)=1e-8*norm(HESS,'inf');
        XN(nvars)=max(g); % Make a feasible solution for qp
    end
    if lenvlb>0,
       AN=[AN;-eye(lenvlb,nvars)];
       GT=[g;-XOUT((1:lenvlb)')+VLB];
    else
       GT=g;
    end
    if lenvub>0
       AN=[AN;eye(lenvub,nvars)];
       GT=[GT;XOUT((1:lenvub)')-VUB];
    end
    [SD,lambda,howqp] = qpsub(HESS,gf,AN,-GT,[],[],XN,OPTIONS(13),-
1, ...
                         'nlconst',size(AN,1),nvars,0,1);
    lambda((1:OPTIONS(13))') = abs(lambda( (1:OPTIONS(13))' ));
    ga=[abs(g( (1:OPTIONS(13))' )) ; g( (OPTIONS(13)+1:ncstr)' ) ];
    if ~isempty(g)
      mg=max(ga);
    else
      mg = 0;
    end

    if OPTIONS(1)>0
           if strncmp(howqp,'ok',2); howqp =''; end


           if ~isempty(how) & ~isempty(howqp)
             how = [how,'; '];
           end
           if OPTIONS(7)==1,
              gamma = mg+f;
              disp([sprintf('%5.0f %12.6g ',OPTIONS(10),gamma),
sprintf('%12.3g  ',OPTIONS(18)),how, ' ',howqp]);
           else
        disp([sprintf('%5.0f %12.6g %12.6g ',OPTIONS(10),f,mg),
sprintf('%12.3g  ',OPTIONS(18)),how, ' ',howqp]);
           end
    end
    LAMBDA=lambda((1:ncstr)');
```

```
    OLDLAMBDA=max([LAMBDA';0.5*(LAMBDA+OLDLAMBDA)'])' ;

%-------------- LINESEARCH --------------------

    MATX=XOUT;
    MATL = f+sum(OLDLAMBDA.*(ga>0).*ga) + 1e-30;
    infeas = strncmp(howqp,'i',1);
    if OPTIONS(7)==0 | OPTIONS(7) == 5
% This merit function looks for improvement in either the constraint
% or the objective function unless the sub-problem is infeasible in
which
% case only a reduction in the maximum constraint is tolerated.
% This less "stringent" merit function has produced faster
convergence in
% a large number of problems.
        if mg > 0
            MATL2 = mg;
        elseif f >=0
            MATL2 = -1/(f+1);
        else
            MATL2 = 0;
        end
        if ~infeas & f < 0
            MATL2 = MATL2 + f - 1;
        end
    else
% Merit function used for MINIMAX or ATTGOAL problems.
        MATL2=mg+f;
    end
    if mg < eps & f < bestf
        bestf = f;
        bestx = XOUT;
    end
    MERIT = MATL + 1;
    MERIT2 = MATL2 + 1;


    OPTIONS(18)=2;
    while (MERIT2 > MATL2) & (MERIT > MATL) & OPTIONS(10) <
OPTIONS(14) & ~OPT_STOP
        OPTIONS(18)=OPTIONS(18)/2;
        if OPTIONS(18) < 1e-4,
            OPTIONS(18) = -OPTIONS(18);

        % Semi-infinite may have changing sampling interval
        % so avoid too stringent check for improvement
            if OPTIONS(7) == 5,
                OPTIONS(18) = -OPTIONS(18);
                MATL2 = MATL2 + 10;
            end
        end
        XOUT = MATX + OPTIONS(18)*SD;
        x(:)=XOUT;
        if strcmp(FUNfcn{4},'ncdtoolbox')
           [f,g] = feval(FUNfcn{1},x,varargin{:});
        else
          [f,g,msg] = opteval(x,FUNfcn,varargin{:});
          error(msg);
```

```
        end
        g = g(:);
        if OPT_STOP
           break;
        end

        OPTIONS(10) = OPTIONS(10) + 1;
        ga=[abs(g( (1:OPTIONS(13))' )) ; g(
(OPTIONS(13)+1:length(g))' )];
        if ~isempty(g)
           mg=max(ga);
        else
           mg = 0;
        end

        MERIT = f+sum(OLDLAMBDA.*(ga>0).*ga);
        if OPTIONS(7)==0 | OPTIONS(7) == 5
            if mg > 0
                MERIT2 = mg;
            elseif f >=0
                MERIT2 = -1/(f+1);
            else
                MERIT2 = 0;
            end
            if ~infeas & f < 0


                MERIT2 = MERIT2 + f - 1;
            end
        else
            MERIT2=mg+f;
        end
    end

%----------- Finished Line Search ------------

    if OPTIONS(7)~=5
        mf=abs(OPTIONS(18));
        LAMBDA=mf*LAMBDA+(1-mf)*LOLD;
    end
    if max(abs(SD))<2*OPTIONS(2) & abs(gf'*SD)<2*OPTIONS(3) & ...
          (mg<OPTIONS(4) | (strncmp(howqp,'i',1) & mg > 0 ) )
        if OPTIONS(1)>0
                  if OPTIONS(7)==1,
                     gamma = mg+f;
                     disp([sprintf('%5.0f %12.6g
',OPTIONS(10),gamma),sprintf('%12.3g  ',OPTIONS(18)),how, '
',howqp]);
                  else
            disp([sprintf('%5.0f %12.6g %12.6g
',OPTIONS(10),f,mg),sprintf('%12.3g  ',OPTIONS(18)),how, '
',howqp]);
                  end
            if ~strncmp(howqp, 'i', 1)
                disp('Optimization Converged Successfully')
                active_const = find(LAMBDA>0);
                    if active_const
                      disp('Active Constraints:'),
```

```
                    disp(active_const)
                  else % active_const == 0
                    disp(' No Active Constraints');
                  end
            end
        end
        if (strncmp(howqp, 'i',1) & mg > 0)
            disp('Warning: No feasible solution found.')
        end
        status=1;

    else
    %   NEED=[LAMBDA>0]|G>0
        if OPTIONS(10) >= OPTIONS(14) | OPT_STOP
            XOUT = MATX;
            f = OLDF;
            if ~OPT_STOP


                disp('Maximum number of function evaluations
exceeded;')
                disp('increase OPTIONS(14)')
            end
            status=1;
        end
    end
end

% If a better unconstrained solution was found earlier, use it:
if f > bestf
    XOUT = bestx;
    f = bestf;
end
OPTIONS(8)=f;
x(:) = XOUT;
if (OPT_STOP)
    disp('Optimization terminated prematurely by user')
end
```