

## INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.
2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.
3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.\*
4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.\*

**\*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.**

**U·M·I** Dissertation  
Information Service

University Microfilms International  
A Bell & Howell Information Company  
300 N. Zeeb Road, Ann Arbor, Michigan 48106



1328516

**Tsao, Lu-Ping**

**INTERACTIVE NONLINEAR PROGRAMMING**

*The University of Arizona*

**M.S. 1986**

**University  
Microfilms  
International** 300 N. Zeeb Road, Ann Arbor, MI 48106



**PLEASE NOTE:**

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark .

1. Glossy photographs or pages \_\_\_\_\_
2. Colored illustrations, paper or print \_\_\_\_\_
3. Photographs with dark background \_\_\_\_\_
4. Illustrations are poor copy \_\_\_\_\_
5. Pages with black marks, not original copy \_\_\_\_\_
6. Print shows through as there is text on both sides of page \_\_\_\_\_
7. Indistinct, broken or small print on several pages
8. Print exceeds margin requirements \_\_\_\_\_
9. Tightly bound copy with print lost in spine \_\_\_\_\_
10. Computer printout pages with indistinct print \_\_\_\_\_
11. Page(s) \_\_\_\_\_ lacking when material received, and not available from school or author.
12. Page(s) \_\_\_\_\_ seem to be missing in numbering only as text follows.
13. Two pages numbered \_\_\_\_\_. Text follows.
14. Curling and wrinkled pages \_\_\_\_\_
15. Dissertation contains pages with print at a slant, filmed as received \_\_\_\_\_
16. Other \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

University  
Microfilms  
International



**INTERACTIVE NONLINEAR PROGRAMMING**

by

**Lu-Ping Tsao**

---

A Thesis Submitted to the Faculty of the  
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING  
In Partial Fulfillment of the Requirements  
For the Degree of  
MASTER OF SCIENCE  
WITH A MAJOR IN ELECTRICAL ENGINEERING  
In the Graduate College  
THE UNIVERSITY OF ARIZONA

1 9 8 6

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED *Lu-Ping Tsao*

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

*Francois E. Cellier*

Francois E. Cellier  
Associate Professor of  
Electrical and Computer Engineering

*6/30/86*  
Date

To my parents.

## ACKNOWLEDGMENT

The author wishes to express his gratitude to his advisor, Dr. F.E.Cellier, for his guidance and helps during the course of this study.

The author would also like to thank Dr. M.Sundareshan, Dr. L.P.Huelsman and Dr. D.G.Dudley for their suggestions and helps.

## TABLE OF CONTENTS

	Page
LIST OF ILLUSTRATIONS . . . . .	vii
ABSTRACT . . . . .	xi
<b>CHAPTER</b>	
1. INTRODUCTION . . . . .	1
2. THE OPTIMIZATION PROBLEM . . . . .	11
2.1 Mathematical Formulation . . . . .	11
2.2 The Unconstrained Optimization Problem . . . . .	16
2.2.1 Single Parameter Problem . . . . .	17
2.2.2 Multiple Parameters Problems Derivatives not Available . . . . .	20
2.2.3 Multiple Parameters Problems Derivatives Available . . . . .	21
2.3 The Constrained Problem . . . . .	23
2.3.1 Transformation Methods . . . . .	24
2.3.2 Direction Modification Methods . . . . .	26
3. NONLINEAR PROGRAMMING PACKAGE NLP . . . . .	28
3.1 General description . . . . .	28
3.1.1 Standard Description . . . . .	30
3.1.2 Special Application . . . . .	31
3.2 Parameter Identification . . . . .	33
3.3 New Version of NLP . . . . .	34
4. THE NLP DEVELOPMENT SYSTEM . . . . .	45
4.1 General Concept of a Development System . . . . .	45
4.2 Commands of the NLP Development System . . . . .	49
4.3 Running Problems in the NLP Development System . . . . .	50

**TABLE OF CONTENTS--Continued**

4.4	Graphic Development System . . . . .	63
4.5	Interactive Editing Programming . . . . .	67
5.	PRACTICAL EXAMPLES . . . . .	71
5.1	Introduction . . . . .	71
5.2	Boundary Value Problem 1 . . . . .	71
5.3	Boundary Value Problem 2 . . . . .	76
5.4	Balancing of Matrix . . . . .	78
5.5	DC-motor System . . . . .	83
5.6	Lotka-Volterra Model . . . . .	88
6.	CONNECTING NLP WITH DARE/INTERACTIVE . . . . .	101
6.1	Introducing DARE/INTERACTIVE . . . . .	101
6.2	Examples . . . . .	112
7.	DISCUSSION OF BOUNDARY VALUE PROBLEM . . . . .	123
7.1	Invariant Embedding Method . . . . .	124
7.2	Discussion . . . . .	126
8.	CONCLUSION AND OPEN QUESTIONS . . . . .	130
8.1	Conclusion . . . . .	130
8.2	Open Questions . . . . .	131
	APPENDIX A RUNNING NLP ON VAX/VMS . . . . .	138
	APPENDIX B RUNNING NLP ON VAX/UNIX . . . . .	139
	REFERENCES . . . . .	140

## LIST OF ILLUSTRATIONS

Figure		Page
1-1	System diagram . . . . .	2
1-2	Interaction between optimization and simulation program . . . . .	10
2-1	Nonlinear programming example . . . . .	13
2-2	Hierarchy of optimization problem . . . . .	15
3-1	Example showing how to use CONT4 as test problem . . . . .	32
3-2	Program and data flow of NLP's identification engine . . . . .	35
3-3	Identification algorithm used by NLP . . . . .	36
3-4	Structure of the program package: Subroutine NLID . . . . .	39
3-5	Structure of the program package: Subroutine NLP . . . . .	40
3-6	Structure of the program package: Subroutine UNIOP . . . . .	41
3-7	Structure of the program package: Subroutine GUNC . . . . .	42
3-8	Structure of the program package: Subroutine CONOP . . . . .	43
3-9	Structure of the program package: Subroutine UNCOP . . . . .	44
4-1	Welcoming display of the NLP development system . . . . .	51
4-2	Command menu of the NLP development system . . . . .	54

LIST OF ILLUSTRATIONS--Continued

4-3	Commands in group ADVANCED . . . . .	55
4-4	Main help message of the NLP development system . . . . .	56
4-5	Help message for STOREDIR . . . . .	57
4-6	Status display . . . . .	58
4-7	Status display for the command DIRECTORY . . . . .	58
4-8	Status display . . . . .	60
4-9	Editing NLP test program . . . . .	61
4-10	Command menu of the GRAPHIC development system . . . . .	63
4-11	Help menu of the GRAPHIC development system . . . . .	64
4-12	Status display . . . . .	65
4-13	Editing subroutine NAME1--(1) . . . . .	69
4-14	Editing subroutine NAME1--(2) . . . . .	69
5-1	NLP program of example 1 . . . . .	73
5-2	Results of the example 1 . . . . .	74
5-3	Temperature distribution in a radiating fin . . . . .	75
5-4	NLP program of example 2 . . . . .	77
5-5	Results of example 2 . . . . .	78
5-6	NLP program of example 3 . . . . .	80
5-7	Results of example 3 . . . . .	82
5-8	A DC-motor system with a gear box and a load consisting of a large rotation mass coupled to the motor by a rather stiff spring . . . . .	84

LIST OF ILLUSTRATIONS--Continued

5-9	NLID program of example 4 . . . . .	86
5-10	Input signal used during the optimization . . . . .	89
5-11	Simulation of plant with initial parameter values . . . . .	90
5-12	Simulation of plant with optimized parameter values . . . . .	91
5-13	Control input signal . . . . .	92
5-14	Results from control simulation run . . . . .	93
5-15	NLID program of example 5 . . . . .	96
5-16	Results of example 5 . . . . .	97
5-17	Simulated and measured trajectories of predator population using initial parameter values . . . . .	98
5-18	Simulated and measured trajectories of predator population using optimized parameter values . . . . .	99
6-1	Program structure of DARE/INTERACTIVE . . . . .	103
6-2	Connection of NLP and DARE/INTERACTIVE . . . . .	105
6-3	Measured input and output of a process . . . . .	113
6-4	DARE/INTERACTIVE program of example 1 . . . . .	114
6-5	Parameter identification in DARE/INTERACTIVE . . . . .	117
6-6	Parameter identification using NLID . . . . .	118
6-7	DARE/INTERACTIVE program of example 2 . . . . .	120
6-8	Graphic output of example 2 . . . . .	122
7-1	FORSIM program of the boundary value problem of example 1 . . . . .	125

LIST OF ILLUSTRATIONS--Continued

7-2	Results of the boundary value problem solved by FORSIM . . . . .	126
7-3	Results of three different approaches for the same boundary value problem . . . . .	126
8-1	Performance index varying versus parameter in uni-dimensional optimization . . . . .	135

## ABSTRACT

Optimization is the single most general technique available for both analysis and synthesis of engineering systems as well as scientific endeavor. In this thesis, we worked on a dynamic optimization system NLP that should ultimately be as easy to use as today's simulation software. We also connected NLP with the simulation software DARE/INTERACTIVE to demonstrate the possibility of integrating an optimization software into a simulation language. Several engineering problems have also been solved as examples in this thesis.

CHAPTER 1  
INTRODUCTION

Science means to determine the basic rules that govern our physical world. This is achieved by observing (measuring) data about some particular properties of our world. We then decide what are the causes, and which are the effects. Thereafter, we may start to experiment with our world, by generating series of causes, and observing the resulting effects. We now call the causes the INPUTS, and the effects the OUTPUTS. The relation between inputs and outputs is termed a SYSTEM.

The essence of engineering is to design what had never been. Our goal is to create a particular pattern of outputs. This can be achieved in two ways: we make use of sets of available inputs, and synthesize a system that maps these available inputs into the desired outputs, or we make use of an available system, and we try to synthesize an appropriate input that shall ultimately generate the desired output. The former approach can e.g. be realized by means of parameters identification, the latter by applying control theory.

A typical system is shown in Fig.1-1.

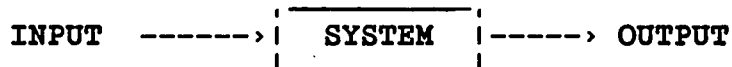


Fig.1-1: System Diagram

Basically, engineering design problems can be divided into three categories, namely:

- 1) ANALYSIS: input and system are known, output is unknown
- 2) SYNTHESIS: input and output are known, system is unknown
- 3) CONTROL: system and output are known, input is unknown.

Class (1) can be further decomposed into two subcategories: INITIAL VALUE PROBLEMS and BOUNDARY VALUE PROBLEMS. Initial value problems can be tackled by means of SIMULATION, and can be solved easily with currently available software. There exists a wide variety of different software systems on the market. These have recently been surveyed in [5]. For the solution of boundary value problems, there exist several different approaches. One is called INVARIANT EMBEDDING, a technique by which boundary value ordinary differential equations (ODE's) are transformed into initial value partial differential equations (PDE's) of the parabolic type, which are then solved again by means of simulation. The steady-state solution of the PDE constitutes the solution

of the boundary value problem. Again, there exist several simulation software systems for that purpose on the "software market", one of which is FORSIM-VI [4], a program currently available at the University of Arizona. Another approach is the so-called SHOOTING METHOD. Here, the unknown initial conditions are interpreted as PARAMETERS of an OPTIMIZATION study. As a PERFORMANCE INDEX we may for instance the sum of the squares of the distance between the evaluated boundary values and the required boundary values. The parameters are now modified in order to minimize the performance index. Each function evaluation involves an entire simulation run. We term this problem a DYNAMIC OPTIMIZATION as opposed to STATIC OPTIMIZATIONS where the performance index is an explicitly stated function that can be directly evaluated for any set of values of the parameter vector.

Class (2) deals with more difficult problems. Again we must distinguish between two subcategories: PARAMETRIC SYNTHESIS and NONPARAMETRIC SYNTHESIS. In the parametric synthesis problem, the structure of the system is assumed to be known up to a set of unknown quantities. These can be either parameters of the system in which case we have a PARAMETER ESTIMATION problem, or initial conditions of the state variables in which case we talk about a STATE IDENTIFICATION problem, or eventually a combination of both. The parameter estimation problem can

again be tackled by means of dynamic optimization. Here the performance index can be e.g. the integral of the square of the distance between the simulated output and the measured output. Again, the parameters are modified in order to minimize the performance index. The state identification problem is well understood in the case of linear deterministic systems (Luenberger observer), and in the case of some classes of linear stochastic systems (Kalman filter). In general terms, the state identification problem constitutes again a kind of boundary value problem where the boundary values are no longer lumped. Very little research has been done so far with respect to combined state identification and parameter estimation (e.g. observers for linear time-varying systems). Nonparametric synthesis problems (STRUCTURE IDENTIFICATION problems) are even less well understood. Some techniques (e.g. general system theory) have been implemented as software (SAPS-II [29]), but they produce results that are so coarse that they are often not directly useful for engineering problems. Other techniques (such as some pattern recognition techniques) are very specific to the particular problem, and hardly extendable to broader classes of problems.

Class (3) has been well studied for linear systems (pole placement, LQG design). The general non-linear case can again be reduced to a parametric dynamic optimization

problem by discretizing the input function over time. For the duration of one time interval, the input is assumed to remain constant. The set of amplitude values for the different time intervals is then considered as the parameter vector for the optimization problem. In case of linear systems with quadratic performance index, this leads to convex feedback design[23].

In this light, dynamic optimization is the single most general technique available for both analysis and synthesis of engineering systems, as well as for the analysis problems in scientific endeavor. However, eventhough there has been reported a lot of research with respect to individual optimization algorithms, and eventhough there is a wealth of individual optimization code available on the market, no general purpose optimization system has been developed so far. If we compare the situation of optimization software with that of simulation software as available on the software market, we find that we are with respect to optimization software now in a similar situation as we were with respect to simulation software prior to the invention of the first simulation language. There exist a good number of reliable optimization routines on the market (e.g. in IMSL), but the user has to "organize" his program by hand, e.g. by linking the optimization routine to an integration routine for dynamic optimization, or to obtain graphical

output, etc. Individual integration subroutines were available that were not compatible with each other with respect to the sequence of call parameters, and even with respect to the types of call parameters to be supplied.

It is the purpose of this thesis to lay the foundation for a dynamic optimization system that should ultimately be as easy to use as today's simulation software. With the advent of more powerful modern computers (virtual memory, greater number-crunching power), this task has now become feasible.

Optimization software adequate to our integrated software system should be flexible, general, robust, reliable, stable, accurate, efficient and easy to use. These characteristics are well defined by Moler and Van Loan [22]. An algorithm is said to be RELIABLE if it gives some warning whenever it introduces excessive errors. An algorithm is STABLE if it does not introduce any more sensitivity to perturbation than is inherent in the underlying problem. Usually, the ACCURACY of an algorithm refers primarily to the error introduced by truncating infinite series or terminating iterations. ROBUSTNESS means that an algorithm is insensitive to algorithmic parameters (sensitive only with respect to physical parameters). The EFFICIENCY is measured by the amount of computer time required to solve a particular problem. GENERALITY means that the method is applicable to wide

classes of problems. The EASE OF USE refers to the decoupling of the algorithmic properties from the physical properties. An optimization program that is easy to use is thus one that allows the user to concentrate on the physical parameters of his problem while he can ignore the algorithmic parameters of the underlying algorithm to a large extent.

As there does not exist a single integration algorithm that is equally useful for all applications, there does not exist a single optimization algorithm capable of minimizing all performance indices equally well. It is thus important to offer a variety of different algorithms in the optimization system. From these points of view, we chose the nonlinear programming subroutine package NLP as the optimization software for our system[26]. This package provides for a large set of different optimization algorithms. The simulation software we chose is DARE-INTERACTIVE, a new dialect within the DARE family[6]. Besides from ordinary simulations, DARE-INTERACTIVE already performs a set of further experiments, such as a complete range analysis (sensitivity analysis for nonlinear models) or a complete replication analysis. Moreover, new run-time experiments are easily integrated into this software. The DARE graphics postprocessor allows us to view families of trajectories either by use of

envelope graphics or by use of three-dimensional graphics with hidden lines removed [6].

Fig.1-2 depicts the interaction between the optimization program (NLP) and the simulation program (RK58). The (user coded) main program sets up the initial parameter vector ( $p_0$ ), and calls the NLP program.

In the case of a static optimization problem, NLP calls a (user coded) subroutine for evaluation of the performance index, passing the actual parameter vector  $p$  to the subroutine, and receiving the current value of the performance index back. NLP then varies the parameter vector until the performance index is minimized. Finally, the optimized parameter vector ( $p^*$ ) is passed back to the main program.

In the case of a danamic optimization problem, NLP calls a (system coded) subroutine for evaluation of the performance index. However, here each evaluation involves one entire simulation run, thus a (system coded) integration algorithm is called which in turn calls a (user coded) subroutine for the specification of the state-space model.

The procedure for developing our new software system is the following:

- 1) transfer NLP from Cyber to VAX/VMS machine
- 2) create a development system for NLP to interactively run the program

- 3) use the DARE postprocessor (and the GRAPHICS development system) to produce graphical output
- 4) integrate the development system with DARE-INTERACTIVE.

Several practical engineering design problems will be solved and shown as examples in this thesis. This thesis can also be considered as a user's manual for the integrated software system.



CHAPTER 2  
THE OPTIMIZATION PROBLEM

2.1 Mathematical Formulation

The problem of optimization deals with the minimization (or maximization) of a PERFORMANCE FUNCTION  $f(\mathbf{x})$ , with or without additional conditions of the form:

$$g_i(\mathbf{x}) \leq 0, \quad i=1, \dots, m$$

$$h_j(\mathbf{x}) = 0, \quad j=1, \dots, p$$

$g_i(\mathbf{x})$  and  $h_j(\mathbf{x})$  are so-called constraints.  $g_i(\mathbf{x})$  are called INEQUALITY CONSTRAINTS, whereas  $h_j(\mathbf{x})$  are called EQUALITY CONSTRAINTS.

If  $f(\mathbf{x})$ ,  $g_i(\mathbf{x})$  and  $h_j(\mathbf{x})$  are all linear equations, then the problem is referred to as LINEAR PROGRAMMING problem. When any of those functions is nonlinear, the problem is called NONLINEAR PROGRAMMING problem. In our program we shall always assume that the performance function (or performance index) is to be minimized. This poses no reduction in generality, as the maximization of any function  $f(\mathbf{x})$  corresponds simply to the minimization of the function  $-f(\mathbf{x})$ . Problems with additional conditions of either the type  $g_i(\mathbf{x})$  or  $h_j(\mathbf{x})$  or both are termed CONSTRAINED PROBLEMS. If a problem has no constraints present, it is called UNCONSTRAINED PROBLEM. Note that the

number of equality constraints must be less than the number of parameters, since each equality constraint reduces the degree of freedom (corresponding to the number of independent parameters) by one. If the number of equality constraints is larger than the number of parameters, there are no degrees of freedom left for optimizing. Such a problem is called OVERCONSTRAINED.

The following is an example of a valid nonlinear programming problem as presented in [9]:

Example.

$$\text{minimize } f(\mathbf{x}) = |x_1 - 2| + |x_2 - 2|$$

subject to the constraints:

$$g(\mathbf{x}) = x_1 - x_2^2 \geq 0.$$

$$h(\mathbf{x}) = x_1^2 + x_2^2 - 1 = 0.$$

Fig. 2-1 depicts this problem graphically. The dashed lines in Fig. 2-1 represent points at which the performance function  $f(\mathbf{x})$  has a constant value. The FEASIBLE REGION is the set of points that satisfy the constraints of the problem.

In this example, the feasible region is the arc of the circle lying within the parabola. A solution to the problem is any point in the feasible region with smallest performance function value. This is seen by inspection to be the point (0.707, 0.707). If the equality constraint is removed from the problem specification, the solution is seen to be at the point (2, 1.414). If both constraints are

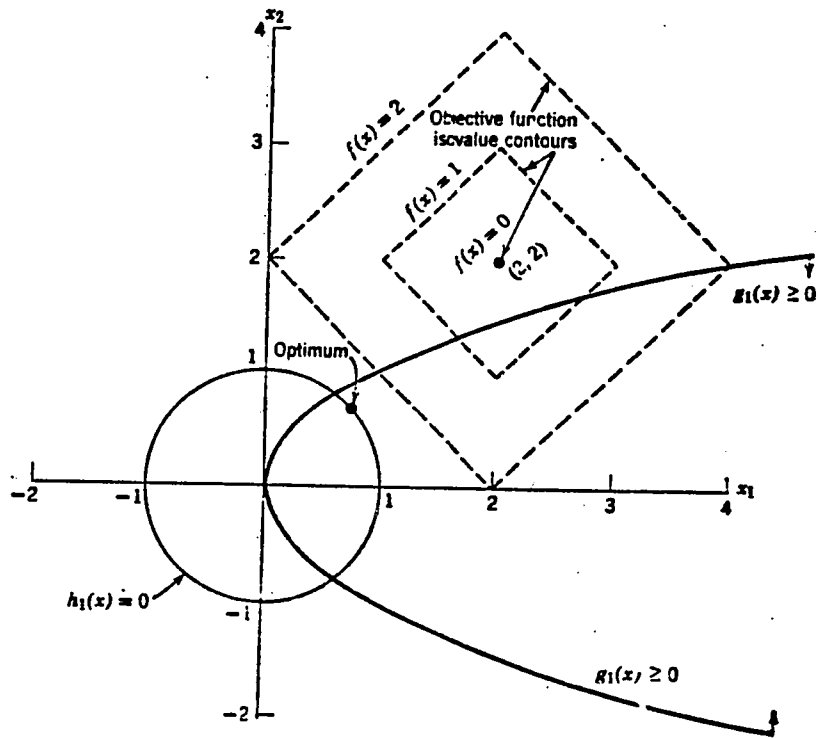


Fig.2-1 Nonlinear Programming Example

removed, the solution is at the point (2,2). In the latter case, the point (2,2) is called the unconstrained minimum.

Generally, the relations between the different types of optimization problems can be seen clearly from Fig.2-2.

Fig. 2-2 shows that constrained problems can be formulated as a set of unconstrained problems. Unconstrained problems can be solved by repeating several cycles of a unidimensional search. Each unidimensional search may involve many gradient computations. This onion-like hierarchy should be reflected in any robust general-purpose nonlinear programming package.

As can be seen from Fig. 2-2, the unidimensional search is the simplest problem. Optimization in many parameters can be thought of as solving several optimizations in one parameter only. Hence, we are going to introduce optimization methods starting from the inner part of Fig.2-2, and progress to the outer part. However, as these optimization techniques are well described in many books, and as the focus of our thesis is on software methodology rather than on numerical optimization techniques, we shall not spend too much emphasis on describing different optimization algorithms in detail. Our survey rather serves the purpose of classifying the different techniques into several groups for later reference. Readers interested in the details of these

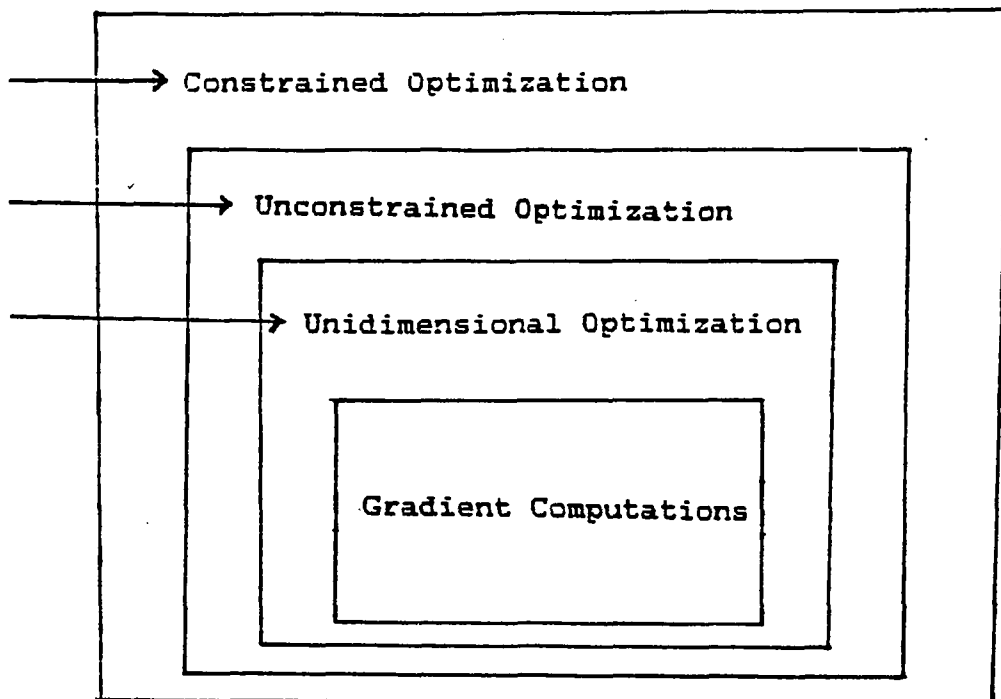


Fig. 2-2

Hierarchy of Optimization Problems

methods are provided with appropriate references in the references section of this thesis. Again, emphasis is placed on easy to understand comprehensive references rather than on a complete survey of the optimization literature.

### 2.2 The Unconstrained Optimization Problem

Almost all of the unconstrained methods use the iteration

$$X_{r+1} = X_r + A_r * S_r$$

where  $S_r$  is the direction of search,  $A_r$  is the current step size,  $X_r$  is the last evaluated point in the parameter space, and  $X_{r+1}$  is the next point to be found.

Algorithms for unconstrained problems can be subdivided into two groups: DIRECT METHODS and INDIRECT METHODS. Direct search methods are characterized by the fact that they make use of the value of the performance function only. Indirect search methods are those that require first derivatives (first-order methods) or even second derivatives (second-order methods) of the performance function with respect to all parameters in addition to the value of the performance function itself.

When the derivatives are difficult or impossible to evaluate, direct search methods are the best choice. However, in cases where the derivatives of the performance function are readily available or easily to be obtained,

direct search methods are generally less efficient than indirect search methods.

Unconstrained algorithms form the basis of the constrained algorithms. Many constrained algorithms are constructed by the use of unconstrained algorithms. Constrained problems are thus often solved by transforming them into a series of unconstrained problems. The most efficient way to solve constrained problems is often to find a transformation that maps the original parameter space into a new parameter space in which the constraints are no longer present. In such a case, the constrained problem in one parameter space is reduced to one single unconstrained problem in another parameter space.

### 2.2.1 Single parameter problems

Two types of method will be introduced in this section, namely; polynomial approximation and Golden section type methods.

Polynomial approximation. The idea behind this class of methods is the following: The performance function is evaluated at  $N$  points, whereafter the performance function is reconstructed as a  $(N-1)$ -st order polynomial. We then minimize the reconstruction polynomial in place of the true performance index which can be achieved by ordinary calculus, hoping that the minimum of the true performance function lies in the neighborhood of the minimum of the approximating polynomial. Next we set up an iteration process which uses this minimum point to

replace one of the original points, and the procedure is repeated until a suitable convergence criterion is satisfied. Usually the number (N) of selected points is 3 or 4, hence the fitted polynomial is either quadratic or cubic. As indicated by Greig [16], this type of technique is usually more efficient than simple search techniques, as information of the shape of the performance index is taken into consideration. The price for this added efficiency is a slightly more complicated algorithm to be implemented.

Golden Section and Fibonacci Search. The name "Golden section" stems from its link with an ideal rectangle, for which

$$ls : ss = ss : (ls - ss)$$

where:

ls ::= long side

ss ::= short side .

From the above, we obtain a ratio between the short side and the long side of 0.618. If we want to find the minimum of a performance function within the interval [a,b], the following algorithm can be used:

- 1) Let  $h = b - a$  (assume  $b > a$ ), and evaluate the function values at the points  $a$ ,  $a+(1-0.618)*h$ ,  $a+0.618*h$ ,  $b$ , thus dividing the interval [a,b] into three subintervals.

- 2) Assuming that the performance function is strictly convex, we know that the minimum must lie in one of the two subintervals that form a valley. The third subinterval can thus be deleted.
- 3) The remaining three points form three of the supporting points of the next golden section step, thus we have to evaluate the performance function at one additional value only to obtain an iterative algorithm that can be repeated until the interval has become sufficiently small, or until the function values do no longer differ much from each other. In each golden section step, we are sure to reduce the interval by a factor of 0.618.

A slight improvement was suggested by Fibonacci. This technique is known as Fibonacci search. Although this algorithm is slightly more efficient, it is used rarely as it requires to decide upon the number of iteration steps beforehand. Moreover, it can be shown that the accuracy of any Fibonacci search with  $N$  iteration steps lies always between that of a golden section search with  $N$  iteration steps and that of a golden section search with  $(N+1)$  iteration steps. Thus, one single additional step of golden section suffices to obtain a result better than that found by the Fibonacci algorithm.

### 2.2.2 Multiple Parameters Problems Derivatives not Available

The most commonly used methods of this type are Simplex method and Pattern search type method.

The Simplex Method. This method was originally proposed by Spendley in 1962 and was modified by Nelder and Mead in 1965. The algorithm of this method, as summarized by Fletcher [11], is that "the performance function is first evaluated at a basis set or 'simplex' of  $(n+1)$  points, and the set is altered systematically --- dropping some points and adding others--- until the region of the minimum is reached. Its precise location is then found by interpolating a quadratic function at suitably chosen points."

This method, however, seems only to work well when the number of parameters is small.

#### Pattern Search and Rosenbrock's Modification.

Pattern search is a  $n$ -dimensional direct search method which was proposed by Hooke and Jeeves in 1961. Basically, this method contains two major parts that are: Exploration and Pattern Search. In the Exploratory part, the method starts at the initial point, and explores from there one direction at a time. If a direction that leads to a decreased function value has been found, a pattern of search is built. The function will be evaluated along this

direction with a larger and larger step size until the search fails, that is, no longer decreases the function value. At this moment, the step size will be decreased for a new search. When the minimum step size is reached without decreasing the function value any further, the algorithm will destroy the current pattern, and return to a new exploratory step. Aoki [2] says that "this method thus seeks to find the direction of the ravines of the objection function, and tries to follow them."

As a further development of the Hooke and Jeeves method, Rosenbrock [25] suggested a method which uses a set of  $n$  mutually orthogonal directions in each cycle of search instead of the exploratory moves.

### 2.2.3 Multiple Parameters Problems Derivatives Available

This type of techniques uses the gradient of the performance function, and hence it requires the partial derivatives of the function. The gradient vector points to the direction where the function value increases or decreases most rapidly. The length of the gradient vector is the rate of change in that direction.

Steepest Descent and Newton's Method. One of the oldest algorithms to solve the unconstrained optimization problem is the steepest descent method which was proposed by A. Cauchy in 1847. The algorithm is very simple, that is, to choose  $(-g)$  as the direction of search where  $g$  is the gradient of  $f(x)$ . The two main advantages of this

method are that it is simple to apply and that it is very reliable. However, a major disadvantage of it is that it converges slowly. As indicated by Burley [3], this algorithm performs a large number of very small steps.

Newton's method basically approximates the performance function  $f(\mathbf{x})$  by its Taylor series cut after the second order term, then finds the minimum of  $f(\mathbf{x})$  along the direction of search chosen as  $(-G^{-1}g)$ , where  $G$  is the Hessian matrix which contains the second derivatives of  $f(\mathbf{x})$ . The minimum can be found if and only if  $G$  is positive defined. Newton's method is particularly useful when  $f(\mathbf{x})$  is in a quadratic form. In this case,  $G$  is constant, and the minimum can be found in exactly one step. However, if  $f(\mathbf{x})$  is a general function other than a quadratic form,  $g$  and  $G$  need to be reevaluated for each iteration step. This is the first disadvantage of this method. The second disadvantage is that it is often unreliable. According to Burley [3,p.42], "this unreliability is usually produced by the quadratic approximation having a saddle point far from the required extremum".

There have been many methods suggested that base on steepest descent and Newton's method. Among those, the method proposed by Davidon is considered to be the most

successful one, and it is the one gradient method that is currently most commonly used.

Davidon's Method. The main idea of Davidon is very simple. It combines the best part of Newton's method and the steepest descent algorithms. That is, it uses only the first derivatives of the function, it converges rapidly, and most of all, it works reliably too. There exist several "dialects" of this algorithm. The most commonly used variety is the so-called Davidon-Fletcher-Powell algorithm which has been found to be most successful. This algorithm works as follows:

- 1) choose the initial condition  $\mathbf{x}$ , and set the initial inverse Hessian matrix  $H_0$  to unity.
- 2) let  $P_r = -H_r * g_r$  be the direction of search during the  $r$ -th iteration step,
- 3)  $\mathbf{x}_{r+1} = \mathbf{x}_r + a * P_r$
- 4)  $S_r = \mathbf{x}_{r+1} - \mathbf{x}_r$ ,  
 $Y_r = g_{r+1} - g_r$ , and  
 $H_{r+1} = H_r + (1/S_r' * Y_r) * S_r * S_r'$   
 $- (1/Y_r' * H_r * Y_r) * H_r * Y_r * Y_r' * H_r$
- 5) If the algorithm does not converge properly, the inverse Hessian can be reset to unity, and the algorithm is repeated by setting  $\mathbf{x}_0 = \mathbf{x}_n$ .

### 2.3 The Constrained Problem

As indicated before, unconstrained methods usually use the iteration

$$\mathbf{x}_{r+1} = \mathbf{x}_r + A_r * S_r$$

to find the minimum  $x$  of  $f(x)$ . The direction of search  $S$  is determined either by the performance function itself or by its gradient. For constrained problems, the constraints play also an import role in the choice of  $S$  .

In general, constrained methods are far more complex and difficult to implement than unconstrained methods. We will not discuss these methods here in great detail. These methods are covered well in [3,p.18]. There are many algorithms leading to the optimization of constrained problems. Most of these methods, however, can be categorized into one of two types: TRANSFORMATION METHODS and DIRECTION MODIFICATION METHODS.

### 2.3.1 Transformation Methods

There are two main categories of transformation methods. In one, the independent variables (that is: the parameter space) are transformed, whereas in the other, the performance function is transformed by use of penalty functions.

In the first type, the idea is to transform the parameter space such that in the new, modified parameter space the constraints are no longer present. We then can simply solve an unconstrained problem in these new parameters. This technique is not always applicable. It requires the geometry of the constraints to be simple enough such that a transformation can be found. The most frequently used transformation of this kind is the

logarithmic transformation. Let us assume that one parameter  $p_r$  had to be always positive. In that case, we can replace  $p_r$  by  $p_r^* = \log(p_r)$ , that is:  $p_r = \exp(p_r^*)$ . If we now consider  $p_r^*$  as the parameter to be optimized, obviously  $p_r^*$  can assume any value as the resulting true parameter  $p_{r+1}$  shall always turn out positive independent of how we choose  $p_r^*$ . Similar transformations exist for parameters that must lie in a prescribed interval  $[a,b]$ .

The penalty function methods transform the performance function by introducing penalty functions. Each modified performance function constitutes an entire unconstrained problem, and the algorithm is set up such that the penalty function is gradually improved such that the last unconstrained problem has a solution close to the solution of the constrained problem. Methods such as the Lagrange Multiplier techniques, and the Kuhn-Tucker method belong to this class of algorithms. Some techniques construct the penalty function inside the valid domain (interior penalty methods), while others add the penalty from the outside (exterior penalty methods). Exterior penalty methods have the advantage that the performance function is not altered at all within the valid domain, but they have the disadvantage that intermediate solutions may be produced that are entirely unphysical (as they may lie outside the valid domain, and are thus somewhat less robust. The major advantage of all penalty methods is that

the constraints are virtually ignored. These methods are applicable independent of the geometry of the constraints. However, these methods show also some disadvantages (as indicated by Davies and Swann [8]), in that some of the methods require function values outside the feasible region (which may not be possible), and in that instead of optimizing the true, original performance function, a modified performance function is optimized. If the program is terminated prematurely during the optimization, the values of the parameters obtained will probably be less satisfactory than we could wish, they may indeed be worst than even the initial values.

### 2.3.2 Direction Modification Methods

Methods of this type are characterized by the fact that they do not alter the performance function. "Some of these methods attempt to follow a constraint while others try to rebound from them and so continue the search in the feasible region." [28,p.190]. Methods such as the gradient-projection technique and the feasible direction methods are of this type. Generally, these algorithms are very complicated. If the reader wants to know more about them, [2,3,28] might be good sources of reference.

One disadvantage of this type of methods is that they don't work well on problems with highly nonlinear constraints. Hence, there does not exist a method that works well on all problems.

The problem of numerical optimization is by no means a new one. Mathematicians have been dealing with this problem since the time of Newton. However, no substantial progress could be made until the invention of the digital computer. Since then, many new algorithms have been introduced which solve unconstrained optimization problems successfully. On the other hand, constrained optimization based on unconstrained optimization methods has also seen a courageous development. However, the topic of optimization is large, and it is still continually developing. With the efforts made by many mathematicians, further developments can be expected in the near future.

However, we feel that the time is now ripe to consolidate the existing state-of-the-art algorithms into a robust and user-friendly software environment. This constitutes the major goal of this research.

CHAPTER 3  
NONLINEAR PROGRAMMING PACKAGE NLP

3.1 General Description

The FORTRAN subroutine package NLP can be used to solve nonlinear programming problems and nonlinear identification problems. In the application of solving nonlinear programming problems, it is desired to find a local minimum of a nonlinear function in one or several parameters, that is,

$$F(p) = \text{minimum} , \quad \text{dim}(p) = NP .$$

The minimization can also be subjected to nonlinear equality constraints and/or inequality constraints :

$$FE(p) = 0 , \quad \text{dim}(FE) = NFE ,$$

$$FI(p) < 0 , \quad \text{dim}(FI) = NFI .$$

If NFE and NFI both are zero, the problem is unconstrained. If NP = 1, a unidimensional optimization is to be performed.

NLP was originally designed by D.Rufer at ETH Zurich as part of his PhD dissertation. The package was developed on a CDC-6500 computer. We chose this program as the basis of our system as it exhibits the onion-like program structure that was presented in Fig.2-2. It comes thus closest to our ideas about a general purpose

optimization engine. Unfortunately, the code was not written in a very neat fashion. There existed many system dependencies, and part of the code is badly written. Meanwhile, there exist partly better routines for optimization (e.g. within IMSL), and it may be a good idea to replace one or the other of the algorithms currently embedded in NLP by better written software. We had to transfer the code to a VAX-11/750 system (the CAD-VAX) as it was our aim to make this batch operated code interactive. As a by-product, we also generated a VAX-UNIX version of NLP. Eventhough some modifications were made on the code itself, the major portion of the run-time code (optimization engine) is still the same as before. Hence, most of the material in this chapter is basically a digest from the original User's Guide of NLP [26].

It is important that, in order to guarantee the convergence of the multidimensional optimization, the functions  $F(p)$ ,  $FE(p)$ ,  $FI(p)$  and their first derivatives with respect to the parameter vector  $(p)$  must be continuous. In particular, NLP currently does not contain any algorithms for stochastic optimization. All algorithms offered within NLP shall perform extremely poorly when applied to a stochastic problem.

Many common engineering problems can be formulated and solved as parameter optimization problems. Practical examples will be shown in chapter five.

NLP contains three different user interfaces, two for static optimizations (parameter optimization), and one for dynamic optimizations (parameter identification).

Parameter optimization problems can be formulated in NLP in two different ways, namely; standard and special applications.

### 3.1.1 Standard Application

The user must not worry about the optimization method. Based on the input parameters, the program selects an appropriate optimization algorithm. To handle constraints, an exterior penalty function method is used. The unconstrained problem is iterated by using the conjugate direction method of Fletcher [12]. Depending on the problem, the method is combined with either a quadratic interpolation or a Golden-section search, and eventually with a numerical gradient approximation with optimized step length.

The user has to supply the performance function  $F(p)$  and eventually constraints  $FE(p)$  and/or  $FI(p)$  in subroutine NAME. If the derivatives of those functions are not available, the user needs to set the input parameter  $IDER=0$ . The description of the input and output parameters can be found in the User's Guide for NLP [26].

### 3.1.2 Special Application

In this application, the user has to manually select one of the optimization methods included in the NLP package. Unidimensional and multidimensional search algorithms and the technique for handling eventual constraints can be selected independently, and the user can define all parameters of these algorithms (step length, termination-criterion, etc.). In this way, the user can experiment with the available algorithms, and compare them easily with respect to their efficiency for the solution of his specific problem. Thus, NLP is here used as a toolbox. Usually, a manual selection of the most efficient algorithm is reasonable only if many similar optimization problems are to be solved afterwards with this algorithm or if the standard technique fails to converge. In most cases, the artificial intelligence component within the standard application engine of NLP is pretty smart with respect to determining a good combination of algorithms. However, more research is needed to ensure a higher degree of software robustness.

Default values are provided for each input parameter in the special application mode. Hence, the user needs to supply only those input parameters in the main program that he wishes to modify.

There are some test problems stored within NLP. With these test problems, the user can get a clear picture of how the subroutine NAME should look like. Subroutine UNIT1 is a unidimensional test problem. Subroutine UNCT1 to UNCT10 are multidimensional unconstrained test problems. Subroutine CONT1 to CONT7 are multidimensional constrained test problems. To run these test problems, the user simply calls them in place of subroutine NAME from within the main program (NAME is passed as EXTERNAL argument to NLP). Fig. 3-1 is an example showing how to use CONT4 as a test problem to be run in the special application mode of NLP:

```

COMMON/INPUT/IOUT,NPI,ND,NE,PI(40),PPERR(40),
*PPER(40),FFERR,FFERA,ICON,MAXCON,CCDE(40),
*F1CON,F2CON,F3CON,DEERR(40),IUNC,MAXUNC,
*IRESET,IUNI,MAXUNI,STPUNI,FACUNI,FFMIN,
*CONUNI,IGNU,GGSTEP(40),GERR(40),FRELER,ISCAL
COMMON/NLPP/NP,D1(42),J1(3),D2(40),J2,D3(42),
*J3,D4(81)
EXTERNAL CONT4
CALL DEF
NPI=3
PI(1)=10.
PI(2)=0.
PI(3)=0.
IGNU=0
IUNI=1
IOUT=5
ND=0
NE=2
NP=0
FFMIN=900.
CALL NLP(CONT4)
STOP
END

```

Fig. 3-1 Example showing how to use CONT4 as test problem.

The meaning of all these parameters is explained in the User's Guide of NLP [26]. However, we shall see later how we can get information interactively about every single parameter in our new VAX version of NLP.

### 3.2 Parameter identification

The NLP package includes subroutine NLID as yet another user interface for solving nonlinear identification problems, which can be formulated as unconstrained dynamic parameter optimization problems:

$$\sum_{i=1}^{MM} QZ(i) * \sum_{j=1}^{NM} | YM(i,j) - YMM(i,j) | \quad \begin{matrix} NZ \\ a \ b \ c \end{matrix} = \text{minimum} .$$

The  $QZ(i)$  represent weighting factors.  $YM(i,j)$  are the values of the  $MM$  output-variables of a continuous-time dynamic process measured at  $NM$  points  $t(j)$  of time.  $YMM(i,j)$  are the corresponding simulated output values of a process model of the form

$$\dot{x}(t) = f(x(t), u(t), t, a)$$

$$x(0) = \begin{matrix} | X(M) | \\ | \ b \ | \end{matrix}$$

$$YMM(i,j) = g(i, x(t(j)), u(t(j)), t(j), c)$$

$x(t)$  are the state variables of the model,  $X(M)$  are the measurable initial conditions and  $u(t)$  the input functions (values extracted from the measured trajectories of the process input). Therefore, the simulated output values  $YMM(i,j)$  are functions of the unknown model parameters  $a$ ,  $b$ , and  $c$ , which have to be optimized such that the output

variables of the process and of the model fit as well as possible. NLP is coded in FORTRAN, and does not make use of any memory management techniques. Consequently, NLP is limited with respect to the sizes of each and every vector and/or matrix it can digest. Currently, up to 10 input variables, up to 10 output variables, up to 30 state variables, up to 40 unknown parameters, and not more than 2100 measurement points can be specified. Practical examples of this application mode will also be shown in chapter five. Fig. 3-2 [27] shows the program and data flow of the identification engine within NLP, while Fig. 3-3 [27] depicts the algorithm used by this module.

### 3.3 New Version of NLP

NLP version 4.2 is the VAX-11/750 version of NLP. Many modifications from the CDC version were required in order to get the program running on the VAX. However, most of these modifications are internal to the code, and do not influence the application program. Those modifications shall not be described in this thesis. We shall limit our discussion to modifications that do influence the user program, namely:

- 1) The common block NLP has been renamed to NLPP.
- 2) Subroutine SECOND for the computation of the elapsed CPU-time was rewritten. This subroutine is of course highly machine dependent, and the user should expect

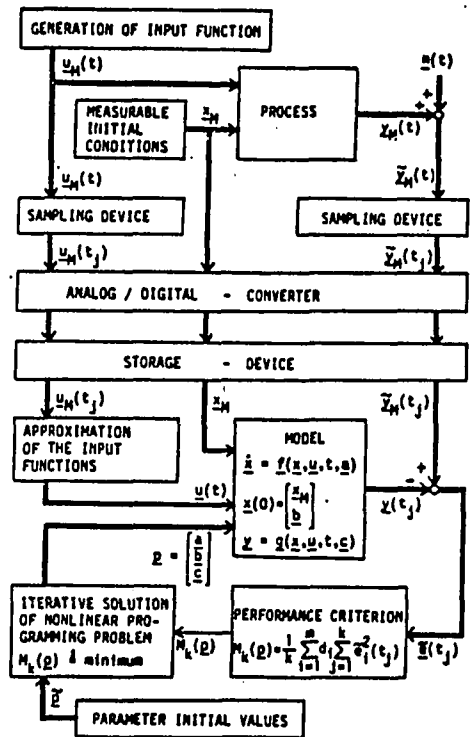


Fig. 3-2: Program and data flow of NLP's identification engine

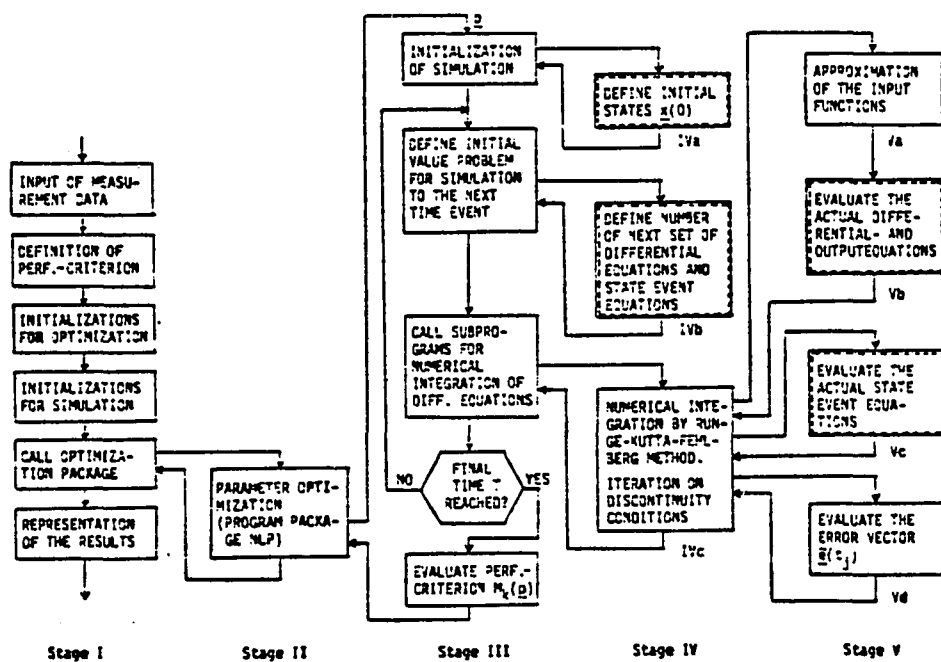


Fig. 3-3: Identification algorithm used by NLP

to obtain different answers when running the software on the VAX rather than on the CYBER.

- 3) We introduced new system dependent subroutines that allow the user to terminate unduly long optimization runs by hitting the CTRL/C key of the terminal without losing everything he has achieved so far. If the user presses the CTRL/C key of his terminal while executing NLP, a PARAMETER FILE containing the last optimized parameter values is created prior to the forced termination of the program. After ordinary termination, a parameter file is also generated.
- 4) Each time NLP is being executed, it tries to open the parameter file that may have been created by a previous run. If such a parameter file exists, NLP will read it and use those data as new starting points of the current execution rather than those supplied in the main program. If no such parameter file exists, the program will run by using the initial parameter values supplied in the main program.
- 5) After each run, the program will by default create four additional data files, MONIT.DAT, TIME.DAT, SAVE.DAT and CROSS.DAT for graphical output processing. These data files can later be used by the DARE/INTERACTIVE graphics engine to plot graphic output from the optimization on a graphical terminal, and/or route it to a printer/plotter. More details and examples of graphical output will be shown in chapter

six. If the user does not want these graphic data files to be created after each run, he must introduce a common block called GRAP in the main program. This common block has only one variable (IGRA) that can be set to (-1) for suppression of these data files.

Fig. 3-4 to Fig. 3-9 show the flow charts of NLP version 4.2. Appendix A describes how to run NLP in batch under VAX/VMS. Appendix B describes how to run NLP in batch under VAX/UNIX.

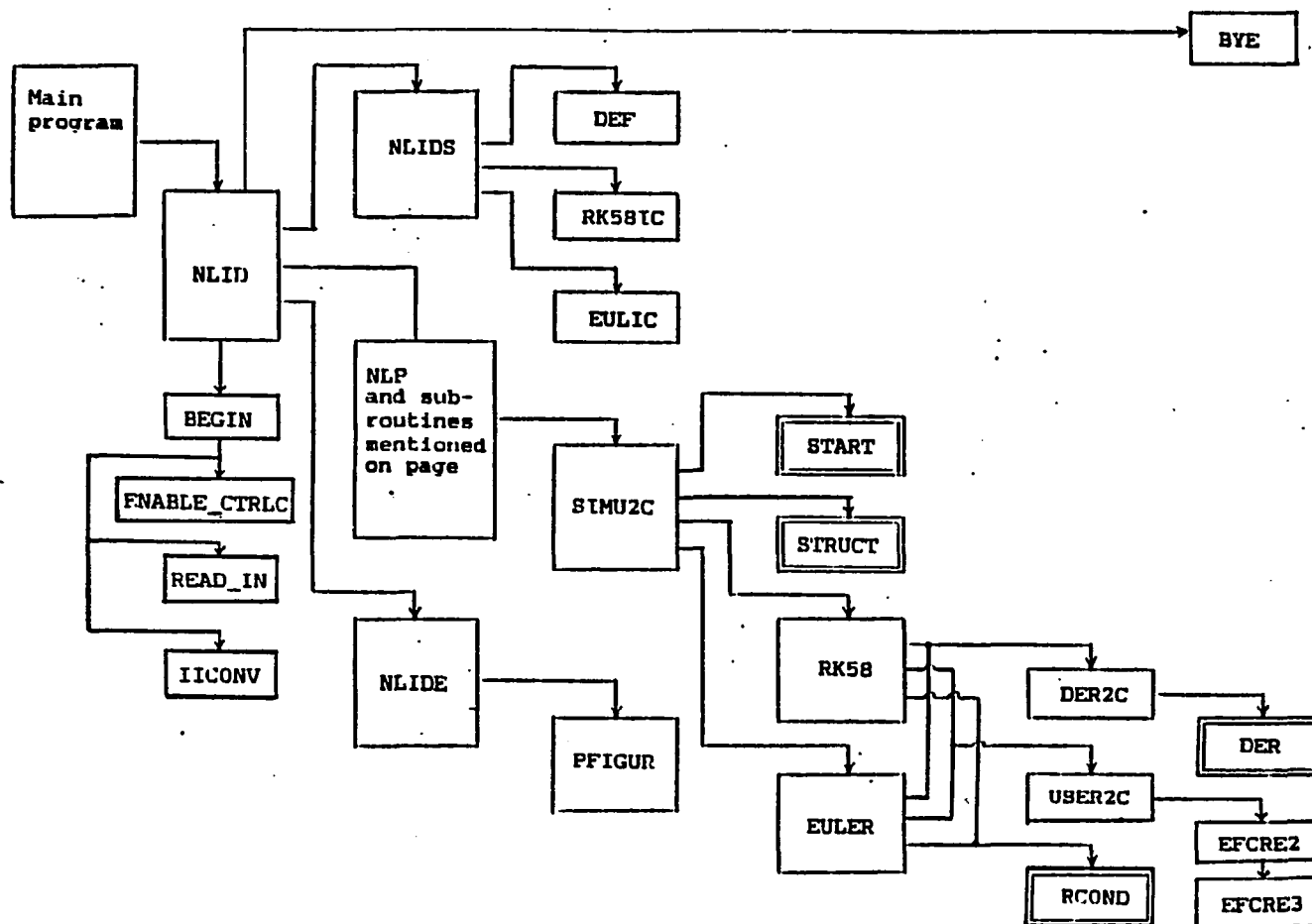


Fig. 3-4: Structure of the program package :  
Subroutine NLID.

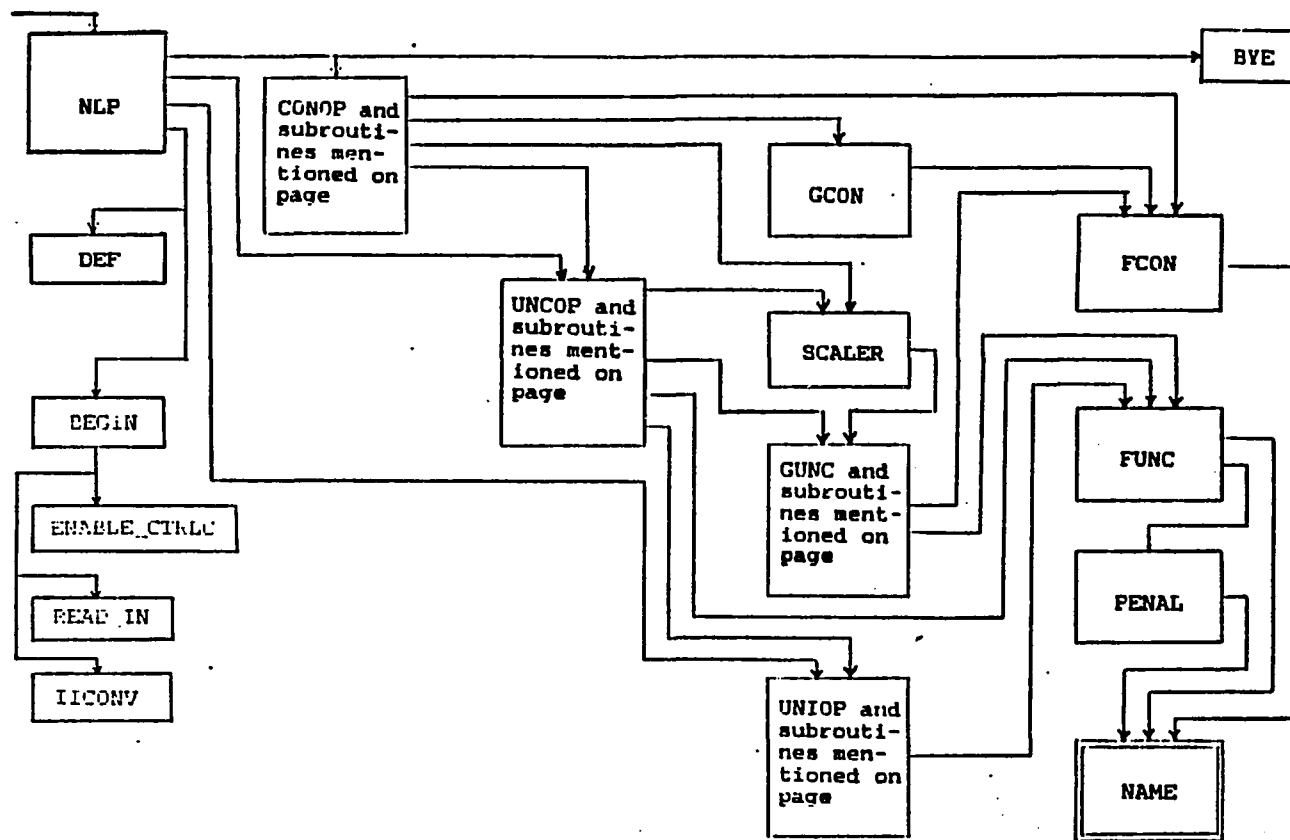


Fig. 3-5: Structure of the program package : Subroutine NLP.

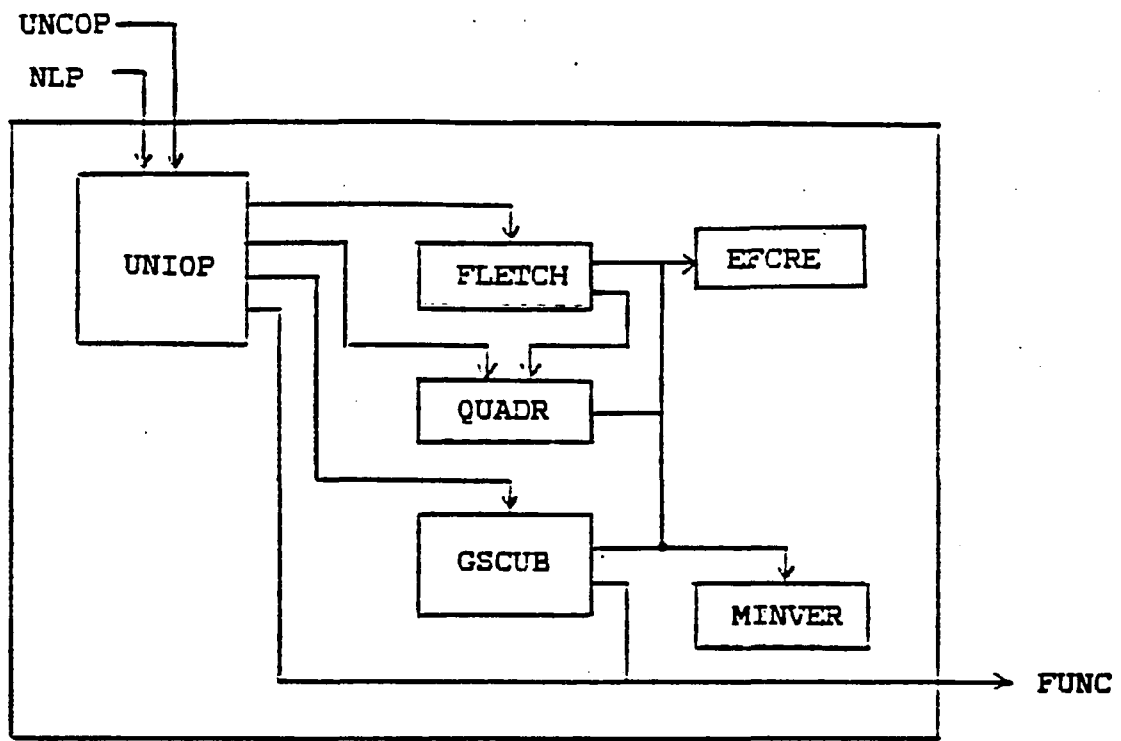


Fig. 3-6: Structure of the program package : Subroutine UNIOP.

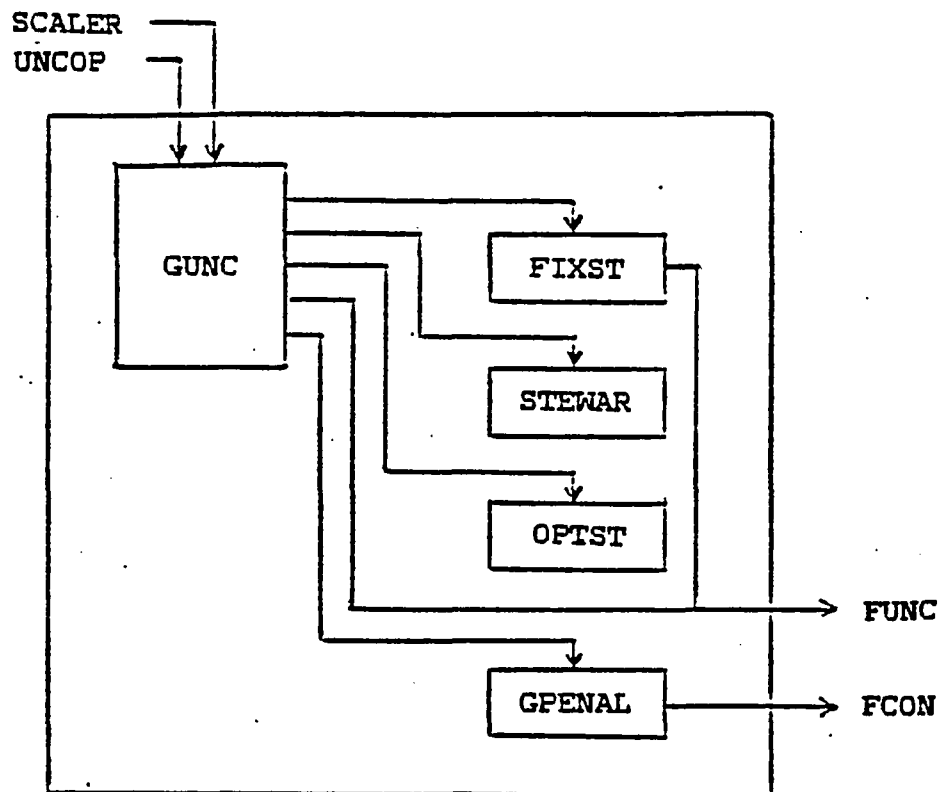


Fig. 3-7: Structure of the program package :  
Subroutine GUNC.

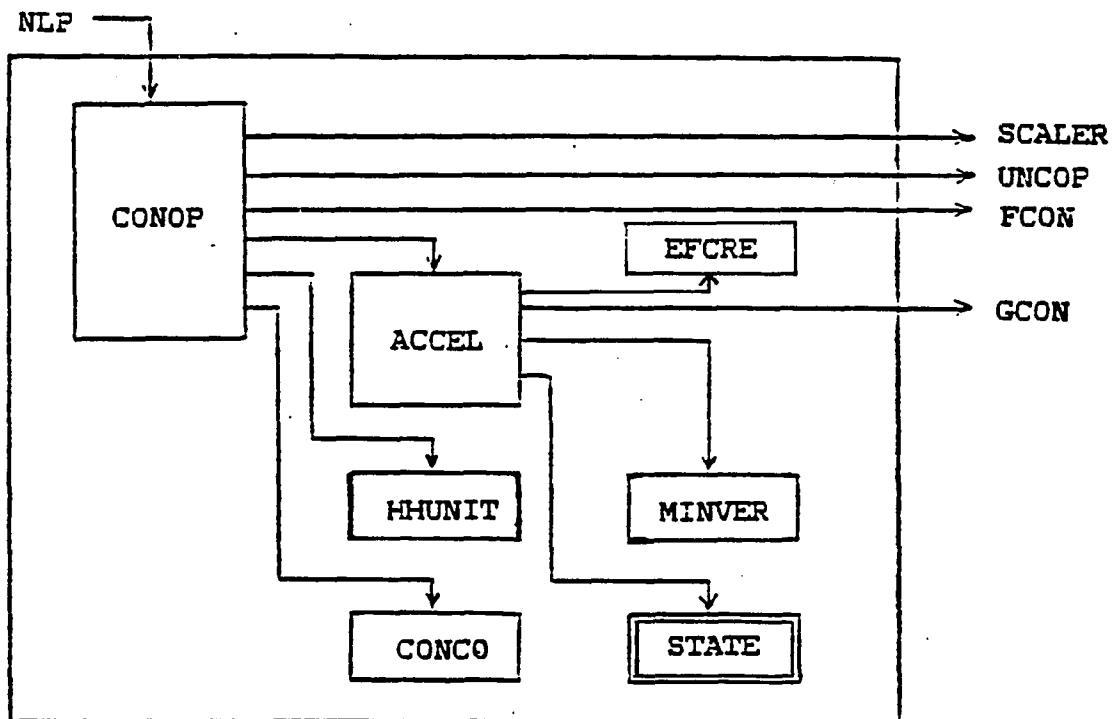


Fig. 3-8: Structure of the program package : Subroutine CONOP.

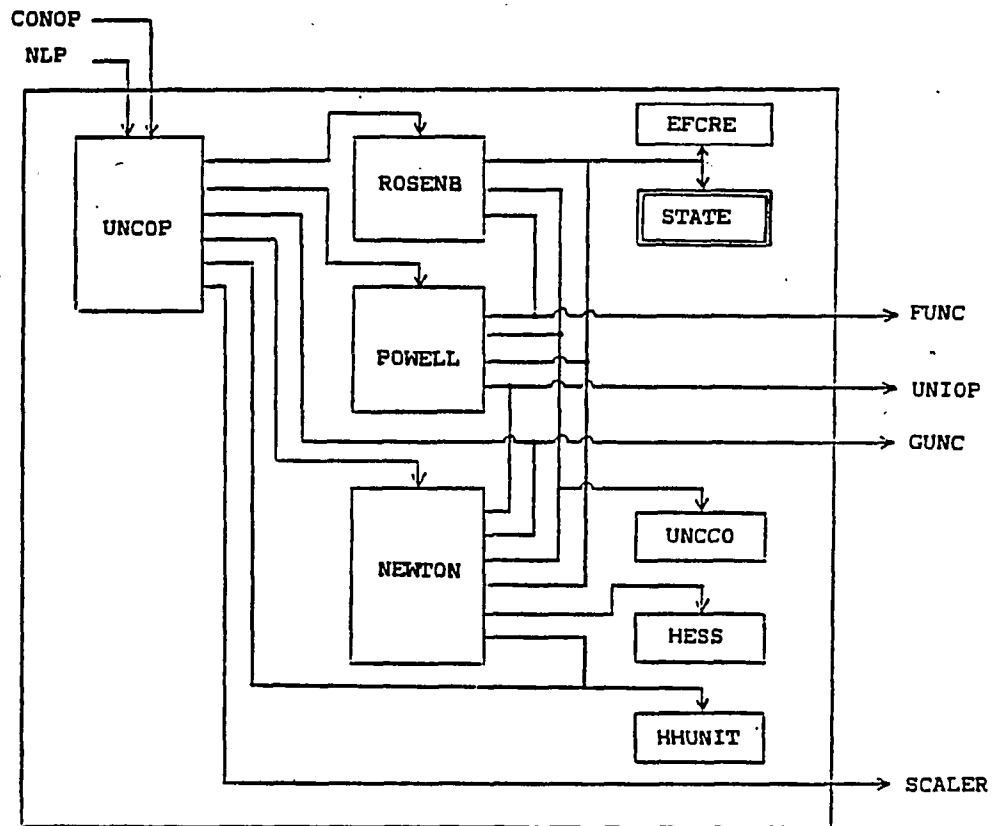


Fig. 3-9: Structure of the program package :  
Subroutine UNCOP.

CHAPTER 4  
THE NLP DEVELOPMENT SYSTEM

4.1 General Concept of a Development System

One of the major problems with using modern software systems lies in their sheer generality and flexibility. To provide this generality and flexibility, modern software systems are no longer monolithic pieces of software coded as one big chunk of code taking the user program as the one and only input file, and producing the result as one big output file to be sent to the line printer. Instead, they consist of many smaller programs that interface with each other through a bunch of different files, and eventually through a data base. The problem with this concept is that the user now requires a considerable amount of time to learn how to interact with the programming environment rather than being able to supply a card deck with his input statements to the "computer" which is basically considered a black box reading punched cards, and producing line printer output.

The immediate solution to this problem on a VAX/VMS is to provide a set of small command procedures that the user can call to compile, link, and execute his program, and eventually to do other things such as

reconfiguring the software in case his problem is larger than what the "standard" version of the software is able to handle. However, even with this concept, the file directories shall soon be swamped with virtually hundreds of different command procedures for one or the other purpose, and it may be quite difficult to remember which procedure to use for what purpose.

The operating system MIDGET [24], an add-on to VMS developed recently by M.Rimvall and F.Cellier, provides a solution to this problem in the form of a set of DEVELOPMENT SYSTEMS that can be considered to be special purpose operating environments for larger pieces of software. Each development system combines all command procedures needed for the operation of one software system (such as NLP) into one menu-driven operating environment that is extremely easy to use as INTERACTIVE HELP is provided for each of the commands available under the environment. Moreover, this menu-driven interface is highly standardized. Once the user has mastered any one of the available development systems, he will find it extremely easy to familiarize himself with any other development system.

To make this statement a little more clear: Almost every development system has a DIRECTORY command that displays the application programs currently written for that particular software system, a SELECT command to pick

one of them for further processing, and an EXECUTE command to compile, link, and run the application program, or whatever else needs to be done with it within the frame of the particular development system. E.g. the VMS command-language logic behind the EXECUTE command of different development systems may be drastically different, but the user does not need to know. Each command of the development system performs an action specific to the software system in question that may involve many commands of the underlying operating system. Each command basically executes one command procedure (eventhough MIDGET is not implemented in this way).

To make the creation of new development systems particularly easy, MIDGET provides for a COMPILER-COMPILER. The SYNTAX of the new development system (that is: the menu interface) is described in terms of a high-level syntactic language in a DEVELOPMENT SYSTEM DESCRIPTION FILE (NLP.DSD). The SEMANTICS of each command are described in an associated COMMAND FILE (NLP.COM). After this is done, the new development system can be "compiled" by use of the compiler-compiler, and the new development system is ready for action.

It is not the aim of this thesis to describe the syntax description language of the MIDGET development systems. However, we felt that this short introduction was needed to enlighten the description of our new NLP

development system, in particular as no paper describing the development system methodology of MIDGET is available yet, and as the only paper describing MIDGET as a whole has been published in German. However, a paper describing the MIDGET toolbox is currently under development by the authors of MIDGET.

The basic idea of the NLP development system is to provide the NLP user with a convenient, easy to use, and systematic environment to deal with NLP. In this environment, the user does not necessarily have to know much about the commands of the computer (VMS). When the user enters the NLP development system, information about the current status of the system is displayed, and then the user is prompted for further command input. If the user has any questions about any of the commands, interactive help information is provided in the system. The commands provided in the NLP development system should suffice for most applications. However, the user can still execute any VMS command directly from within the development system if needed. Any command starting with a '\$' is considered by MIDGET to be a VMS command, and is passed on to the underlying operating system for further processing.

The NLP development system contains the following files:

- 1) NLP.DSD: file describing the syntax (menu interface) of the NLP development system.

- 2) NLP.COM: main command file of the NLP development system.
- 3) NLPEXE.COM : command file to compile, link, and execute an NLP program running under the NLP development system. NLPEXE.COM is called indirectly (@NLPEXE) from within NLP.COM.
- 4) NLPBAT.COM: command file to submit an NLP program to the batch queue for detached operation. NLPBAT.COM is called indirectly (@NLPBAT) from within NLP.COM.

#### 4.2 Commands of the NLP Development System

In the following section, the commands and functions provided in the NLP development system are briefly described. For further information, refer to the interactive help facility of the NLP development system.

CURRENT: make old version current again.  
DATA: edit the user data file.  
DELETE: delete the currently selected problem.  
DEVELOP: switch to another development system  
DIRECTORY: list all defined NLP problems.  
EXECUTE: compile and run NLP problem.  
EXIT: save current settings and return to VMS  
FORGET: forget parameter values from last run.  
LINK: edit the link file.  
LISTFLAG: switches the listing flag on/off.  
LISTING: edit the NLP listing-file.

LISTMAP: edit the NLP loader map.  
LOAD: load new settings from a file.  
MAP: switches the loader map flag on/off.  
MENU: switches the menu display on/off.  
NOTEBOOK: edit the note book file.  
OUTPUT: edit NLP output file.  
PRINT: print a file on line printer.  
PROGRAM: edit NLP program.  
PURGE: purge all old versions.  
QUEUES: display status of queues.  
QUIT: return to VMS without saving the current settings.  
QUOTA: show disk quota.  
READ: read file from another problem.  
RUN: run an already compiled NLP problem.  
SAVE: save the current settings on a file.  
SELECT: select problem to be treated.  
STOREDIR: select storage directory.  
SUBMIT: submit the problem to the batch queue.  
WORKDIR: select working directory.  
WRITE: write file to another problem.

#### 4.3 Running Problems in the NLP Development System

In this section we want to show the procedure of how to run an NLP problem in the NLP development system. To enter the development system, we simply type the (MIDGET) command:

develop nlp

in response to the VMS prompter (\$). MIDGET takes over control from VMS, and the message shown in Fig. 4-1 could be displayed (the precise form of the message depends on the current settings):

```
*****
*****
*****
*****      WELCOME TO THE NLP DEVELOPMENT SYSTEM      *****
*****
*****
*****
*****
```

Message of the day:  
\*\*\*\*\*

January 5, 1986

- 1) This is a test version of the brand-new NLP development system. Please report ALL errors (however small) to Francois Cellier immediately. Please send a (VAX/VMS) MAIL to CELLIER and MIDGET.
- 2) Do not forget to report all errors with MAIL to MIDGET.

Dr. Francois Cellier  
ECE 205, phone: 621-6192

FLAGS: Menu-OFF FORTRAN listing-ON Loader map-OFF  
DEFAULTS: All Defaults - [SCRATCH.LUPING]

SYSPICS : Selected problem - \*\*\*UNDEFINED  
Problem type - \*\*\*UNDEFINED

Selected command :

Fig. 4-1 Welcoming display of the NLP development system.

The NLP development system operates on three FLAGS. The MENU-flag determines whether or not the available commands

should be listed after each command execution. This may be very useful in the beginning, but becomes soon cumbersome (especially when the software is operated from a terminal which is set to a low load rate). Thus, MENU will mostly be switched off. To see the list of commands once again, the user just needs to hit <CR> in response to "Selected command". The FORTRAN listing flag (currently switched on) determines whether or not a listing file is to be produced during the compilation of the selected NLP problem. The loader map flag (currently switched off) determines whether or not a loader map is to be produced during linkage.

The NLP development system operates on two WORKSPACES (usually directories). The WORKING AREA is the directory where transient files are to be stored whereas the STORAGE AREA is the place where the source code of the application programs is maintained. By default, all workspaces are set to the directory from where the development system was called (in this case: `_dua0:[scratch.luping]`), but they can be modified, and the modified specifications can be saved as part of the settings. It is usually a good idea to keep the working area (scratch area) the one from where the development system is called, while the source programs are stored on the next level of hierarchy (in our case: `_dua0:[scratch.luping.nlp]`). A good way of doing

would be to create a directory [username.develop], and directories for all development systems that are utilized by this particular user, such as: [username.develop.nlp], [username.develop.graphic], [username.develop.database], etc.. Then call any development system always from within the working area, that is: [username.develop], and save the underlying directories as the respective storage areas. In this way, the only files the working directory are ever to contain are directory files and a bunch of transient (scratch) files, whereas all the important stuff is maintained orderly in separate directories. It then becomes particularly easy to get rid of any garbage files after a session is completed to avoid any disk overflows.

The NLP development system operates on two SYSPICS. A SYSPIC is a name (symbol) of the operating system similar to a formal parameter of a subroutine. Many of the commands of the NLP development system refer to one or both of the SYSPICS. One of them denotes the SELECTED PROBLEM. Commands such as PROGRAM, DELETE, EXECUTE always refer to the selected problem. As a user usually wants to perform several activities relating to one problem in a row (such as editing, executing, re-editing, etc.), this makes sense as it prevents the user from having to tell again and again on which problem to operate. If no problem is currently selected, an error message will occur when any of these commands is executed. The second SYSPIC is an

indicator that tells the user which is the type of the currently selected problem (optimization with either preselected parameters (STANDARD) or user selected parameters (SPECIAL), or identification (NLID)). The application type is determined during the first problem definition, and is never changed thereafter. By default, all SYSPICS take a value of "\*\*\*UNDEFINED".

Now, the development system waits for the first command to be specified. First, we may want to have a look at the available commands. For this purpose, we simply hit <CR>, and obtain the message shown in Fig. 4-2.

```

FLAGS: Menu-OFF FORTRAN listing-ON Loader map-OFF
DEFAULTS: All Defaults - [SCRATCH.LUPING]

SYSPICS : Selected problem - ***UNDEFINED
             Problem type - ***UNDEFINED

COMMANDS :
  DIRECTORY <-- List all defined NLP problems
  EXECUTE <-- Compile and run NLP Problem
  HELP <-- Extensive help information.
  OUTPUT <-- Edit NLP output file
  PROGRAM <-- Edit NLP program
  SELECT <-- Select problem to be treated
             (also PROBLEM)
SUB-MENUS containing further commands:
  ADVANCED <-- Advanced commands to run NLP.
  DEFAULTS <-- Commands to change default values
  DEVICES <-- Commands to control/check external
             devices
  EDIT <-- Further editing commands
  FILES <-- File manipulation commands
  SYSTEM <-- Commands to leave current development
             system

```

Selected command :

Fig. 4-2 Command menu of the NLP development system.

The most commonly used commands are specified directly (in the main menu), whereas less used commands are collected into submenus. This is to prevent the screen from overflowing. However, submenus do not present a concept of hierarchy. All commands (also those belonging to submenus) can be entered at any time. True command hierarchies and even recursiveness is available in MIDGET, but there was no need to make use of these concepts within the NLP development system.

If the user wants to look at the commands of any submenu, he simply types the name of the group command. We can, for instance, type:

Selected Command :advanced

to obtain the following response from MIDGET:

```

FLAGS: Menu-OFF  FORTRAN listing-ON  Loader map-OFF
DEFAULTS: All Defaults      - [SCRATCH.LUPING]

SYSPICS : Selected problem      - ***UNDEFINED
          Problem type          - ***UNDEFINED

COMMANDS  in group ADVANCED
FORGET    <-- Forget parameter values
RUN       <-- Run an already compiled NLP Problem
STOREDIR  <-- Select storage directory.
SUBMIT    <-- Submit the problem to the batch queue.
WORKDIR   <-- Select working directory.
PRESS <CR> TO RETURN TO MAIN MENU

Selected command :
```

Fig. 4-3 Commands in group ADVANCED.

If the user wants to know more about any particular

command, he can always ask for help. In this example, we type

Selected command : help

## HELP

The HELP command invokes the VAX-11 HELP Utility to display information about a particular development system topic. The HELP utility retrieves help available on specific commands as well as general development system topics.

Format:

HELP [keyword [...]]

New users can display a tutorial explanation of HELP by typing TUTORIAL in response to the "HELP Subtopic?" prompt and pressing the RETURN key.

For a more detailed description of HELP, type COMMANDS and press the RETURN key.

Additional information available:

ADVANCED	CURRENT	DATA	DEFAULTS	DELETE
DEVELOP	DEVICES	DIRECTORY	EDIT	EXECUTE
EXIT	Fileassignments		FILES	FORGET
HELP	LINK	LISTFLAG	LISTING	LISTMAP
LOAD	MAP	MENU	MODE	NOTEBOOK
OUTPUT	Parameters	PRINT	PROBLEM	PROGRAM
PURGE	QUEUES	QUIT	QUOTA	READ
RUN	SAVE	SELECT	SPAWN	STOREDIR
SUBMIT	SYSTEM	WORKDIR	WRITE	

Topic?

Fig. 4-4 Main help message of the NLP development system.

The HELP command activates an ordinary VMS help library. Thus, HELP alone will display a list of all commands for

which help is available, while "HELP command-name" will display information about the command in question.

It is a general rule of MIDGET that, wherever possible, parameters can be supplied directly with the commands. However, if they are omitted, MIDGET enters a QUERY mode and asks for them. This is called the TYPE AHEAD feature of MIDGET. Input is buffered, and the system will omit all dialogue with the user as long as the input buffer is not empty, and simply satisfy its needs from the input buffer.

At the current moment in the dialogue, we could e.g. type:

Topic? storedir

to obtain the display:

STOREDIR

Lets you select the storage directory.  
This is an advanced command, should not be used  
by inexperienced users.

Topic?

Fig. 4-8 Help message for STOREDIR.

To exit form help library, simply press <CR>.

Now, let us change the storage area from [SCRATCH.LUPING] to [SCRATCH.LUPING.NLP]. We type:

Selected command : storedir \_dua0:[scratch.luping.nlp]

The display then becomes:

```

FLAGS: Menu-OFF  FORTRAN listing-ON  Loader map-OFF
DEFAULTS: Workspace      - [SCRATCH.LUPING]
          Storage Media   - [SCRATCH.LUPING.NLP]
SYSPICS : Selected problem - ***UNDEFINED
          Problem type    - ***UNDEFINED

```

Selected command :

Fig. 4-6 Status display.

If we now want to check which are the NLP problems that we currently have available for operation, we type:

Selected command : directory

to obtain the display:

NLP program files currently defined :

Directory DUAO:[SCRATCH.LUPING.NLP]

```

BALAN.NLP;6      BCP2A.NLP;3      BCP2B.NLP;4
DCMOT1.NLP;56   DCMOT2.NLP;8     DCMOT3.NLP;13
LOTKA.NLP;17    NLID1.NLP;14     PDE_1B.NLP;10
TEST1.NLP;7     TEST2.NLP;7      TEST3.NLP;5
TEST4.NLP;15    TEST6.NLP;5

```

Total of 14 files.

Users data files currently defined :

Directory DUAO:[SCRATCH.LUPING.NLP]

```

DCMOT1.NDA;6      DCMOT3.NDA;7      LOTKA.NDA;1
NLID1.NDA;2

```

Total of 5 files.

Link files currently defined :

Directory DUAO:[SCRATCH.LUPING.NLP]

Fig. 4-7 Status display for the command DIRECTORY.

BALAN.NLK;2

Total of 1 file.

Note book files currently defined :

Directory DUAO:[SCRATCH.LUPING.NLP]

LOTKA.NNB;1

Total of 1 file.

```

FLAGS: Menu-OFF  FORTRAN listing-ON  Loader map-OFF
DEFAULTS: Workspace      - [SCRATCH.LUPING]
          Storage Media   - [SCRATCH.LUPING.NLP]
SYSPICS : Selected problem - ***UNDEFINED
          Problem type    - ***UNDEFINED

```

Selected command :

Fig. 4-7 --Continued.

This shows all the NLP program files in the [SCRATCH.LUPING.NLP]. The extension .NLP denotes NLP user programs. .NDA files represent data files to be read from the user program. .NTY files are system maintained. They contain only one line indicating the problem type which will be recognized by the development system and then displayed as a SYSPIC. .NPA is the (system maintained) file containing the last optimized parameter values either after termination of a run, or after CTRL/C has been pressed during the execution of a run. .NLK is a (user maintained) file containing information about additional programs or libraries to be linked (cf. to the /OPTION feature of the VMS linker). An example of a correct .NLK

file would be the following:

```
_DUAO:[SIM.LINPACK.OBJECT]LINPACK/LIB .
```

The .NNB files denote notebook files containing any information the user wants to store about a particular problem. This is like a scratch pad in some modern operating systems. If we now want to choose the problem TEST4.NLP to be executed, we type:

```
Selected command : select test4
```

Note that no extension is needed here. This will give us the display:

```

FLAGS: Menu-OFF  FORTRAN listing-ON  Loader map-OFF
DEFAULTS: Workspace           - [SCRATCH.LUPING]
          Storage Media       - [SCRATCH.LUPING.NLP]
SYSPICS : Selected problem    - TEST4
          Problem type        - NLP-STANDARD

```

```
Selected command :
```

Fig. 4-8 Status display.

The problem type NLP-STANDARD indicates that the selected problem TEST4 is a "standard" application of NLP. If the problem type is still \*\*\*UNDEFINED, it means that the problem TEST4 does not exist. If we want to look at the program or make some changes, we type:

```
Selected command : program
```

if the program TEST4 does exist, the program will be taken into the editor (by calling one of the standard VMS

editors that is compatible with the terminal the user is operating from).

```

PROGRAM NLP1
COMMON/NLPP/NP, PA(40), FMIN, FTOL, IDER, IOU, NFE,
*FETOL(40), NFI, FITOL(40), FAC, IND, FRES, FERES(40),
*FIRES(40)
COMMON/GRAP/IGRA
EXTERNAL NAME1
IGRA=-1
NP=2
PA(1)--1.2
PA(2)-1.
FMIN=1.E10
FTOL=0.1
IDER=1
IOU=1
NFE=0
NFI=0
CALL NLP(NAME1)
STOP
END
SUBROUTINE NAME1
COMMON/FUNCT/IFUN, P(40), F, FIE(40), DF(40), DFIE(1600),
*IERR
Z1=P(2)-P(1)*P(1)
Z2=1.-P(1)
F=100.*Z1*Z1+Z2*Z2
IF(IFUN.GE.-1) RETURN
DF(2)=200.*(P(2)-P(1)*P(1))
DF(1)=-2.*(DF(2)*P(1)+1.-P(1))
RETURN
END

```

Fig. 4-9 Editing NLP test program.

After leaving the editor, the system automatically returns to the dialogue mode of the NLP development system.

Should the program TEST4 be a new problem for which no program exists yet, an interactive program will be executed to ask the user to input the parameters needed for the execution according to the application type of NLP (which is to be specified as the first piece of

information). This program will be discussed in more detail in one of the next sections.

Now we want to execute the problem. We type:

Selected command : execute

this command will compile, link and run the program TEST4. The system will respond to the command EXECUTE with the following display (which may take some time though):

FORTRAN COMPILER RUNNING ON TEST4.NLP

LINKER RUNNING

PROBLEM IS BEING EXECUTED

FORTRAN STOP

After the problem has been executed successfully, system will ask the user whether he wants a hardcopy and/or whether the transient files are to be cleaned up.

Do you wish to receive a hardcopy? (Y/N):

Do you wish to clean up your files? (Y/N):

If the second question is answered affirmatively, all transient files including the image file of the problem (TEST4.EXE) and all output files are gone. Thus, if the user still wants to look at his results, he may not allow the system to clean up at this point.

If the user did not clean up the files at this point, he can use the OUTPUT command to take the alphanumeric output file (TEST4.OUT) into the editor. Alternatively, the user may wish to view his results graphically. Usually, NLP prepares some data files for the

DARE-INTERACTIVE graphics processor. However, the NLP development system does not contain any commands to activate this program. Instead, we have to switch the development system entirely by typing:

Selected command : develop graphic

this command requests to switch to the GRAPHIC development system which will be discussed in the next section.

#### 4.4 Graphic development system

At the beginning of this chapter, we said that all development systems which are currently available on VAX/VMS operate in a very similar way. We would like to show this in this section by executing the GRAPHIC development system. As mentioned before, if we set IGRA-1 in the main program, there will be four data files generated by NLP. These data files can be used by the GRAPHIC development system to graph the simulation results.

After typing DEVELOP GRAPHIC, we enter the GRAPHIC development system which responds with a status display that is very similar to the one presented by the NLP development system:

```

FLAGS : Menu - ON
DEFAULTS: All defaults      - [SCRATCH.LUPING]
SYSPICS : Selected problem  - ***UNDEFINED
          Selected data base - ***UNDEFINED

```

Fig. 4-10 Command menu of the GRAPHIC development system.

```

COMMANDS :
CHANGE      <-- Create or modify data file
CONVERT     <-- Convert ASCII-files into graphics
              data files
DIRECTORY   <-- List all defined graphic problems
EXECUTE     <-- Run graphic processor
HELP       <-- Extensive help information.
SELECT     <-- Select problem to be treated
              (also PROBLEM)

SUB-MENUS containing further commands:
DEFAULTS   <-- Commands to change default values
DEVICES    <-- Commands to control/check external
              devices
EDIT       <-- Further editing commands
FILES     <-- File manipulation commands
SYSTEM    <-- Commands to leave current development
              system

```

Selected command :

Fig. 4-10 --Continued.

Again, there are commands in the submenus that work just the same way as in the NLP development system. There are also help messages available for this system. The topics of the help library are listed below:

Additional information available:

```

DEFAULTS   DEVELOP   EXIT       GRA:All_news
GRA:CHANGE GRA:CLEANUP GRA:CLEAR   GRA:CONVERT
GRA:CURRENT GRA:DBASE  GRA:DBDIR  GRA:DELETE
GRA:DEVICES GRA:DIRECTORY GRA:EDIT
GRA:EXECUTE GRA:FILES  GRA:INPUT  GRA:Intro
GRA:Message GRA:NOTEBOOK GRA:OUTPUT
GRA:PRINT   GRA:PROBLEM GRA:PURGE  GRA:QUEUES
GRA:QUOTA   GRA:SELECT  GRA:STOREDIR
GRA:SUBMIT  GRA:WORKDIR HELP       LOAD
MENU       MODE       QUIT      SAVE
SPAWN      SYSTEM

```

Fig. 4-11 Help menu of the GRAPHIC development system.

If we want to look at the graphic output of the problem we had mentioned in the last section (suppose we had set IGRA-1 in that program), we may e.g. specify:

Selected command : menu

then:

Selected command : storedir [scratch.luping.dare]

followed by:

Selected command : select test4

and finally:

Selected command : dbase test

This will result in the following display:

```

FLAGS      : Menu - OFF
DEFAULTS: Workspace           - [SCRATCH.LUPING]
          Storage media       - [SCRATCH.LUPING.DARE]
SYSPICS   : Selected problem  - TEST4
          Selected data base  - TEST

```

Selected command :

Fig. 4-12 Status display.

Next, we type:

Selected command : execute

If there do exist those four data files we mentioned before, the system will now activate the DARE-INTERACTIVE graphics processor which by itself enters an interactive dialogue mode. The prompt here is:

OUTPUT-MON>

Again, interactive help is available on all commands of

this hierarchy level (note that this now represents a true command hierarchy while the graphic development system was entered by calling the MIDGET DEVELOP command recursively from within the NLP development system). Type command DATA will display all variable names available for output. In the case of TEST4, they are

PI

PA01

PA02

which are the performance function and all parameters of the problem. It is worth mentioning here that the performance function and all parameters were automatically stored in the file CROSS.DAT. In the case of identification problems (problem type: NLID), state variables, input variables, measured output variables and simulated output variables are stored in addition in the file SAVE.DAT. Hence, in our case, we can e.g. type:

```
OUTPUT-MON> gras(cross:black:yellow) PI(:red),  
              PA01(:blue),PA02(:magenta)
```

which would give us (e.g. on the Tektronix 4105 terminal) a split screen graph of the three variables versus the run number. The entire graph will be displayed in black on yellow background, whereas the individual curves are displayed in the specified colors. If the user wants a hardcopy of the output, he must type the following command prior to the GRA command

OUTPUT-MON> set,file

this will, instead of displaying the output on the screen, put the output into a file. When the user exits from the graphic monitor, the output can then be printed out on an appropriate device (e.g. a laserpin printer). To exit from the graphic monitor, simply type EXIT. This will terminate the execution of the graphic processor, and the system returns to the dialogue mode of the GRAPHIC development system. It is important to notice that upon termination of any development system, all transient files will be automatically destroyed. This includes all image files, and also all output files (as disk space is always more scarce than CPU time). Hence, if the user wants to save any of these files (e.g. the graphic data files) for later reuse, he had better rename these files before exiting from the development system. As the development system self does not provide such a facility, use VMS in transparent mode, e.g. by typing:

Selected command : \$rename \*.dat \*.sav

The "\$" indicates transparent mode. MIDGET simply passes the command on to VMS after strapping off the "\$"-sign.

#### 4.5 INTERACTIVE EDITING PROGRAM

As mentioned before, if the user has selected a problem which does not exist, the command PROGRAM will activate an INTERACTIVE PROGRAM GENERATOR: EDINLP in the development system that supports the user in specifying his new problem.

EDINLP is a FORTRAN coded program designed specifically for NLP. EDINLP supports standard and special NLP applications, as well as NLID (that is: parameter identification) applications. Each application type has its own parameter set associated with it to be input by the user. EDINLP asks first what the application type is, and then asks the user for each parameter in a row that requires a user specified value. In the application type "NLP-special", all parameters have default values available. If the user wishes to use the default value with respect to a parameter, he can simply type "D" in response. The program will then omit that parameter from the user program. If the user types "D" for a parameter that has no default value associated with it, the program will tell him that 'No default value available !' and ask him again to input a value for that particular parameter.

Also EDINLP can activate a VMS help library that contains messages for every single parameter used in NLP. If the user does not understand the meaning of a particular parameter, he can simply type "?" in response to the request of entering a value. EDINLP will then search the help library for the specific parameter, and display the therein contained message. Users unfamiliar with all parameters of NLP, can type "Q" e.g. in response to the first parameter value request. EDINLP then enters a

QUERY mode, and help messages will be displayed automatically for every parameter. Type "Q" again when you wish to suppress further QUERY information. "Q" acts as a toggle parameter.

NLP requires the user to supply one or several subroutines. After the parameters for the main program are exhausted, EDINLP will generate skeleton subroutines one at a time, and ask the user to fill in the missing information. In particular, the required common blocks will already be provided by EDINLP, and do not need to be user coded.

For example, in the case of a NLP-standard problem, EDINLP will display:

```

SUBROUTINE NAME1
COMMON/FUNCT/IFUN,P(40),F,FIE(40),DF(40),DFIE(1600),
*           IERR

```

Fig. 4-13 Editing subroutine NAME1--(1).

It is important that the user must input the rest of the subroutine as normal FORTRAN input lines. The example shown below has the correct format.

```

SUBROUTINE NAME1
COMMON/FUNCT/IFUN,P(40),F,FIE(40),DF(40),DFIE(1600),
*           IERR
Z1-P(2)-P(1)*P(1)

```

Fig. 4-14 Editing subroutine NAME1--(2).

```
Z2=1.-P(1)
F=100.*Z1*Z1+Z2*Z2
IF(IFUN.GE.-1) RETURN
DF(2)=200.*(P(2)-P(1)*P(1))
DF(1)=-2.*(DF(2)*P(1)+1.-P(1))
RETURN
END
```

Fig. 4-14 --Continued.

EDINLP will "monitor" user input, but not check for correct FORTRAN syntax. It will use the "END" statement to know that the user considers the current subroutine as terminated. It will then proceed with the next step. Once all the required parameters and subroutines have been supplied, EDINLP will terminate, and control is returned to the dialogue mode of the NLP development system. Subsequent executions of the PROGRAM command result in a normal editing of the previously generated NLP program.

There are two things that the user should keep in mind, namely:

- 1) All input must be uppercase.
- 2) All parameter values must be real except for parameter names beginning with I,J,K,L,M,N.

CHAPTER 5  
PRACTICAL EXAMPLES

5.1 Introduction

As we mentioned before, it is interesting that many problems can be formulated and solved as parameter optimization problems. In this chapter, we show some practical examples solved by the NLP package.

5.2 Boundary Value Problem 1

Example 1. In this example we want to solve a boundary value problem which is used in [4] to calculate the temperature distribution in a radiating fin. The equation is

$$d^2T/dx^2 = c1*(T^4 - c2) .$$

where  $x \in [0,1]$ ,  $T(0)=100.0$ ,  $c1=1.0E-4$ ,  $c2=0$ ,  $dT/dx(1)=0$ . As mentioned before, there are several ways to solve this problem. Here we would like to use the optimization approach.

We first convert this ordinary differential equation (ODE) into a partial differential equation (PDE) of the parabolic type.

$$dT/dt = -d^2T/dx^2 + c1*(T^4 - c2)$$

In the steady state,  $dT/dt=0$ , thus, the temperature distribution of the PDE in its steady state represents the

solution of our previously stated boundary value ODE. This transformation is usually referred to as invariant embedding technique.

Now, we use the first order approximation for the spatial derivatives:

$$d^2T/dx^2 = [T(i+1)-2*T(i)+T(i-1)]/\Delta x^2, \quad x=x(i)$$

to transfer the PDE back into a set of first order ODE's

$$dT(i)/dt = -[T(i+1)-2*T(i)+T(i-1)]/\Delta x^2 \\ + c1*(T(i)^4 - c2) .$$

where  $x$  is the length of equispace. The ODE at the last point is

$$dT(n)/dt = -2*[T(n-1) - T(n)]/\Delta x^2 \\ + c1*(T(i)^4 - c2)$$

for symmetry reasons. (This can be shown easily to be true. Let us enlarge the space from the interval [0,1] to an interval [0,2], and set  $T(2)=T(0)$ . Then, for symmetry reasons, obviously  $dT/dx(1)=0$ . Thus, if "n" represents the central point of the new grid,  $T(n+1)=T(n-1)$ . If we plug this equation into the general ODE above, we obtain the special form for the boundary condition, and can now forget about the additional grid points again.) If we choose  $x=0.1$ , and let  $E(i) = dT(i)/dt$ , then the ODE's become

$$E(i) = -100*[T(i+1)-2*T(i)+T(i-1)] + c1*(T(i)^4 - c2), \quad i=1, \\ \text{and}$$

$$E(10) = -200*[T(9) - T(10)] + c1*[T(10)^4 - c2] .$$

In the steady state, all the  $E(i)$  functions must become equal to zero. Thus, instead of solving the set of ODE's over time, we can formulate this problem as a static optimization problem where the  $T(i)$  represent the unknown parameters, and a performance function, e.g.:

$$PI = \sum E(i)^2$$

is minimized. Obviously, we expect the PI to become close to zero in the neighborhood of the optimal parameter vector. We use the NLP-standard application for this problem. The main program together with subroutine NAME1 which codes the performance function are as follows:

```

COMMON/NLPP/NP,PA(40),FMIN,FTOL,IDER,IOU,NFE,
*      FETOL(40),NFI,FITOL(40),FAC,IND,FRES,
*      FERES(40),FIRES(40)
COMMON/GRAP/IGRA
EXTERNAL NAME1
IGRA = -1
NP = 10
DO 10 I=1,10
PA(I) = 100.
10 CONTINUE
FMIN = 0.100E+11
FTOL = -0.100E-07
IDER = 0
IOU = 1
NFE = 0
NFI = 0
CALL NLP(NAME1)
STOP
END
SUBROUTINE NAME1
COMMON/FUNCT/IFUN,P(40),F,FI(40),DF(40),
*      DFIE(1600),IERR
DIMENSION E(10)
C1 = 1.0E-4
E(1) = 100.*(P(2)-2.*P(1)+100.) - C1*P(1)**4

```

Fig. 5-1 NLP program of example 1.

```

F = 0.
DO 10 I=2,9
E(I) = 100.*(P(I+1)-2.*P(I)+P(I-1)) - C1*P(I)**4
F = F + E(I)**2.
10 CONTINUE
E(10) = 200.*(P(9)-P(10)) - C1*P(10)**4
F = F + E(10)**2
RETURN
END

```

Fig. 5-1 --Continued.

The results of the optimization are as follows:

```

FP          - 0.2109258E+1
NFUN        - 1410
CPU-TIME    - 0.447E+1
P( 1)      - 0.6589E+2
P( 2)      - 0.5064E+2
P( 3)      - 0.4197E+2
P( 4)      - 0.3641E+2
P( 5)      - 0.3261E+2
P( 6)      - 0.2995E+2
P( 7)      - 0.2810E+2
P( 8)      - 0.2687E+2
P( 9)      - 0.2617E+2
P(10)      - 0.2594E+2

```

Fig. 5-2 Results of the example 1.

where FP denotes the final value of the performance function, P(I) represents the temperature at the grid points, and NFUN denotes the number of function evaluations (that is: number of times, subroutine NAME1 was called). The graphical result for this problem is depicted in Fig. 5-3 which was obtained by use of the GRAPHICS development system and a Cannon laserpin printer.

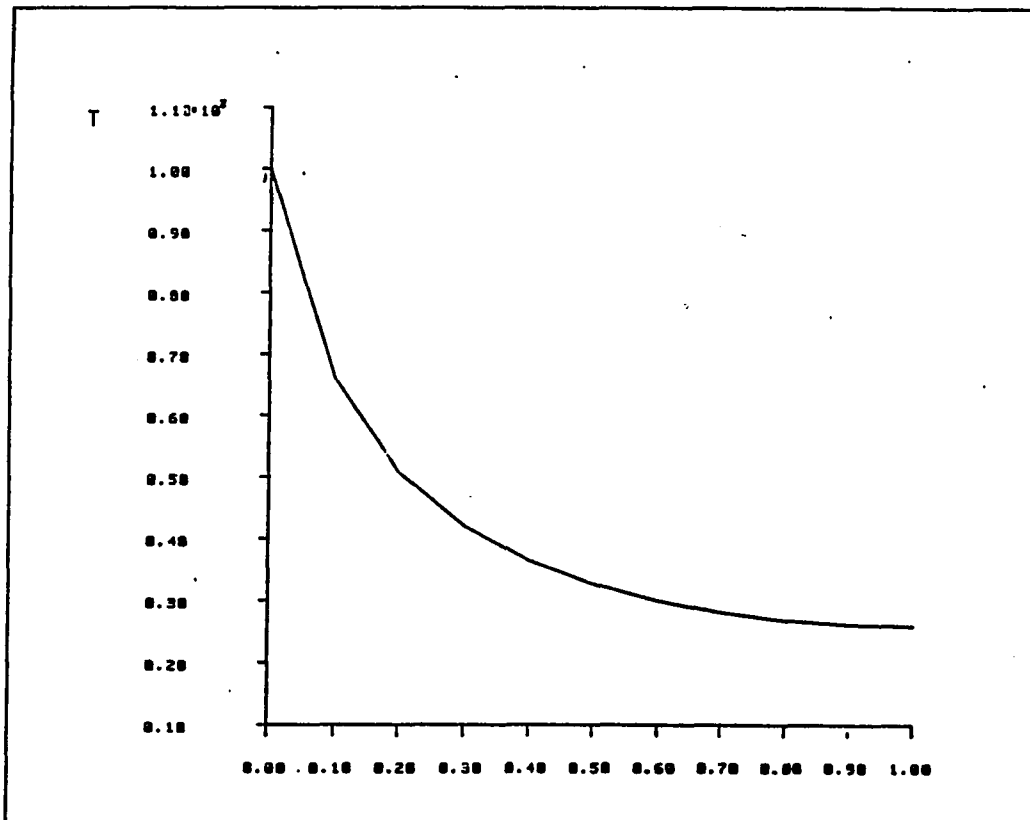


Fig. 8-3 Temperature distribution in a radiating fin.

T is the steady-state temperature distribution expressed in C, and graphed over Space x [0,1]m.

### 5.3 Boundary Value Problem 2

Example 2. In this example, we would like to find the steady state solution of the following PDE:

$$dU/dt = a*d^2U/dx^2 .$$

with  $x \in [0,1]$ ,  $a=0.25$ ,  $U(x,t=0)=0$ ,  $U(x=0,t)=1$ ,  $dU/dx(x=1,t)=0$ . This example has the advantage that we can compute the correct solution by hand. We expect all  $U(i)$  to become equal to 1 after sufficiently long time. Meanwhile, we learnt the trick. We obtain the following set of ODE's:

$$dU(i)/dt = a*[U(i+1)-2*U(i)+U(i-1)]/\Delta x^2$$

and

$$dU(n)/dt = 2*a*[U(n-1)-U(n)]/\Delta x^2 .$$

We also choose  $x=0.1$  and let  $E(i) = dU(i)/dt$  which leads to the following set of equations:

$$E(1) = 25*[U(2)-2*U(1)+1]$$

$$E(i) = 25*[U(i+1)-2*U(i)+U(i-1)] , i=2,9$$

$$E(10) = 50*[U(9)-U(10)]$$

This time, we used the NLP-special application for this problem. We chose the method of Powell (IUNC-10) for the unconstrained optimization. The NLP user program for this problem is shown below:

```

COMMON/INPUT/IOUT,NPI,ND,NE,PI(40),PPERR(40),
*          PPERA(40),FFERR,FFERA,ICON,MAXCON,
*          CCDE(40),F1CON,F2CON,F3CON,DEERR(40),
*          IUNC,MAXUNC,IRESET,IUNI,MAXUNI,STPUNI,
*          FACUNI,FRELER,ISCAL
COMMON/NLPP/NP,D1(42),J1(3),D2(4),J2,D3(42),J3,
*          D4(81)
COMMON/GRAP/IGRA
EXTERNAL NAME1
CALL DEF
IGRA = -1
NPI = 10
DO 10 I=1,NPI
PI(I) = 0.000E+00
10 CONTINUE
FFERA = 1.E-4
MAXUNC = 100
IRESET = 4
FRELER = 1.E-8
DO 20 I=1,10
PPERR(I) = 1.E-3
PPERA(I) = 1.E-4
20 CONTINUE
NE = 0
ND = 0
IOUT = 3
IUNC = 10
IGNU = 8
ISCAL = 0
NP = 0
CALL NLP(NAME1)
STOP
END
SUBROUTINE NAME1
COMMON/FUNCT/IFUN,P(40),F,FIE(40),DF(40),DFIE(1600),
*          IERR
DIMENSION E(10)
E(1) = 25.*(P(2)-2*P(1)+1.)
F = 0.
DO 10 I=2,9
E(I) = 25.*(P(I+1)-2*P(I)+P(I-1))
F = F + E(I)**2
10 CONTINUE
E(10) = 50.*(P(9)-P(10))
F = F + E(10)**2
RETURN
END

```

Fig. 5-4 NLP program of example 2.

The results obtained from NLP are the following:

```

FP           - 0.9616193361580E-02
NFUN        - 3249
CPU-TIME    - 0.31300E+01
PA(I)       - 0.9893833398819E+00
U(x-0.1)   - 0.9789869785309E+00
U(x-0.2)   - 0.9690473675728E+00
U(x-0.3)   - 0.9597477912903E+00
U(x-0.4)   - 0.9513475894928E+00
U(x-0.5)   - 0.9441356658936E+00
U(x-0.6)   - 0.9382637739182E+00
U(x-0.7)   - 0.9339659214020E+00
U(x-0.8)   - 0.9314922690392E+00
U(x-0.9)   - 0.9309771656990E+00

```

Fig. 5-5 Results of example 2.

#### 5.4 Balancing of Matrix

Example 3. In determining the eigenvalues of a matrix A, it has been shown by [15] that numerical errors satisfy the condition:

$$\text{ERROR}(\lambda_i) \leq S_1 \cdot E^{1/m}$$

Where  $S_1$  is the largest singular value (L2-norm of the matrix),  $m$  denotes the largest multiplicity of eigenvalues, and  $E$  is the numerical precision of the computer (machine resolution). In particular, matrices with many multiple eigenvalues thus can show large errors in the numerical results of an eigenvalue computation. Let us assume, we want to compute the eigenvalues of a 16x16 matrix in double precision that happens to have all eigenvalues at the same point ( $m=n-16$ ). The machine resolution is roughly  $10^{(-16)}$ , thus  $E^{(1/m)}$  is roughly

equal to 1. Thus, we cannot compute the multiple eigenvalue of this matrix more accurately than the L2-norm of the matrix which may be arbitrarily large. We call such a matrix ill-conditioned with respect to eigenvalue computation.

The trick here is to find another matrix  $AH$  which has the same eigenvalues but a smaller L2-NORM. From the theory, we know that we can apply any regular similarity transformation on  $A$  without changing the eigenvalues. Thus, we can choose an arbitrary non-singular matrix  $T$ , and compute:

$$AH = T \cdot A \cdot T^{-1}$$

Any such matrix  $AH$  is supposed to have the same eigenvalues as  $A$ . Thus, we can formulate the problem as a static optimization problem where the unknown elements of the matrix  $T$  are determined such that the L2-NORM of  $AH$  is minimized. A reduced precision of the optimization is harmless as it will only result in an  $AH$  with slightly larger norm. The only operation that is performed "on line" is the similarity transformation which by itself is numerically harmless as long as  $T$  is well-conditioned with respect to inversion, that is: the condition number of  $T$  (largest singular value divided by smallest singular value) should be sufficiently small).

We still have to consider  $n^2$  parameters which may be a lot. To simplify the problem, we consider  $T$

diagonal. This avoids all problems with respect to inversion as long as the ratio between the largest and the smallest  $|T(i,i)|$  is within reasonable bounds. Of course, this will give us a suboptimal solution only. Interestingly enough, it was found that we can predict beforehand what the L2-NORM of the suboptimal matrix AH shall be, as we shall explain further down in this example. The matrix we chose for our example is a 4x4 matrix:

$$A = \begin{vmatrix} 4 & 11 & 3 & 0 \\ 5 & 23 & 14 & 1 \\ 3 & 4 & 15 & 3 \\ 7 & 9 & 2 & 21 \end{vmatrix}$$

The NLP-standard application program presents itself as follow:

```

COMMON/NLPP/NP, PA(40), FMIN, FTOL, IDER, IOU, NFE,
*      FETOL(40), NFI, FITOL(40), FAC, IND, FRES,
*      FERES(40), FIRES(40)
COMMON/GRAP/IGRA
EXTERNAL NAME1
IGRA = 1
NP = 4
PA(1) = 0.100E+01
PA(2) = 0.100E+01
PA(3) = 0.100E+01
PA(4) = 0.100E+01
FMIN = 500.
FTOL = -1.E-8
IDER = 0
IOU = 2
NFE = 0
NFI = 0
CALL NLP(NAME1)
STOP
END

```

Fig. 5-6 NLP program of example 3.

```

SUBROUTINE NAME1
COMMON/FUNCT/IFUN,P(40),F,FIE(40),DF(40),
*   DFIE(1600),IERR
DOUBLE PRECISION RAX(4,4),SX(4),EX(4),UX(4,4),
*   VX(4,4),WORKX(4)
DIMENSION AX(4,4),AA(4,4),TX(4,4),TIX(4,4),AB(4,4)
DATA AX/4.,5.,3.,7.,11.,23.,4.,9.,3.,14.,15.,
*   2.,0.,1.,3.,21./
LDXX = 4
LDVX = 4
NP = 4
LDUX = 4
DO 20 I=1,NP
DO 10 J=1,NP
IF (I.EQ.J) THEN
    TX(I,J) = P(I)
    TIX(I,J) = 1./P(I)
ELSE
    TX(I,J) = 0.
    TIX(I,J) = 0.
ENDIF
10 CONTINUE
20 CONTINUE
CALL MULT(TX,AX,AA)
CALL MULT(AA,TIX,AB)
DO 40 I=1,4
DO 30 J=1,4
RAX(I,J) = DBLE(AB(I,J))
30 CONTINUE
40 CONTINUE
CALL DSVDC(RAX,LDXX,4,4,SX,EX,UX,LDUX,VX,LDVX,
*WORKX,11,INFOX)
ERROR = SNGL(SX(1))
F = ERROR**2
RETURN
END
SUBROUTINE MULT(A,B,C)
C THIS SUBROUTINE PROVIDES MULTIPLICATION OF TWO
C MATRICES C = A*B
DIMENSION A(4,4),B(4,4),C(4,4),D(4)
DO 30 I=1,4
DO 20 J=1,4
DO 10 K=1,4
D(K) = A(I,K)*B(K,J)
10 CONTINUE
C(I,J) = D(1) + D(2) + D(3) + D(4)
20 CONTINUE
30 CONTINUE
RETURN
END

```

Fig. 5-6 --Continued.

Here we assign the square of the largest singular value of  $AH = T^*A^*T^{**}(-1)$  as the performance function. Subroutine DSVDC is a subroutine from the LINPACK program library which was linked with NLP using the LINK option of the NLP development system. The reason for using the double precision version (DSVDC) in place of the single precision version (SSVDC) was simply that we currently have only the double precision version available on the CADVAX computer system. The output obtained is the following:

```
FP          - 0.1083668212891E+4
NFUN        - 191
CPU-TIME    - 0.15970E+2
PA(1)       - 0.8975525498390E+0
PA(2)       - 0.1015476822853E+1
PA(3)       - 0.1372430920601E+1
PA(4)       - 0.5358199477196E+0
```

Fig. 5-7 Results of example 3.

From the output we found that the largest singular value was reduced to 32.91911621 which happens to be identical with the largest eigenvalue of A. This seems to be a coincidence as we could not reproduce this phenomenon with another matrix. However in another case, we found that the largest singular value converged to about twice the largest eigenvalue. Thus, there may will exist a relation between these two numbers. Unfortunately, we were not able to find it.

### 5.5 DC-motor System

Example 4. A DC-motor system is shown in Fig. 5-8 [27]. The parameters and their values are the following:

PSI -	0.06 Vs,	flux
RA -	1.81 ohm,	armature resistance
LA -	14.8 mH,	armature inductance
F -	0.4 mN/rad,	spring constant
UE -	60 -,	ratio of PHI1/PHI2
J1 -	unknown,	inertia
D1 -	unknown,	friction
J2 -	unknown,	inertia
D2 -	unknown,	friction

The numerical values specified above had been found through measurements. The parameters J1, D1, J2 and D2 are unknown parameters to be identified. We are going to show the application of NLID (parameter identification) for this example. We first need to derive the mathematical model of the system. For the electrical subsystem of the motor, we find:

$$dI/dt = U/LA - RA*I/LA - UI/LA$$

or, with armature inductance being neglected (LA=0):

$$I = U/RA - UI/RA$$

where:

I(t)	-	armature current
U(t)	-	armature voltage

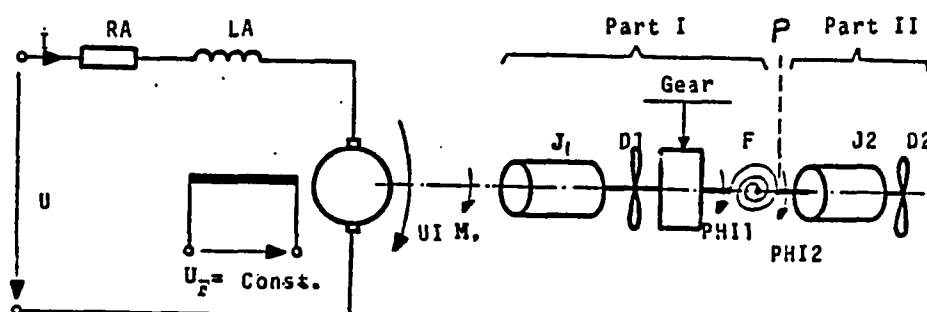
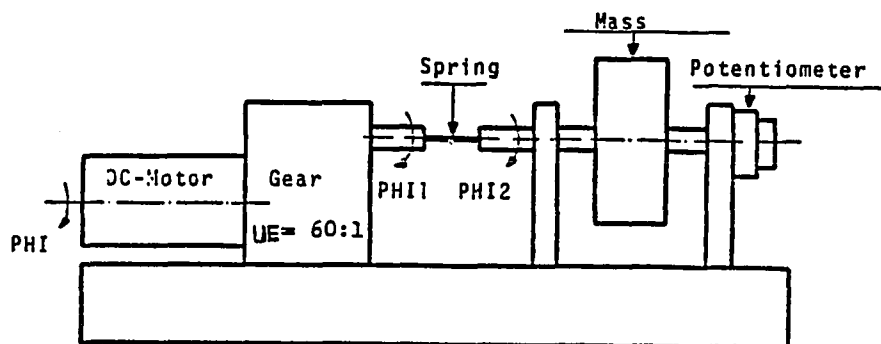


Fig. 5-8 A DC-motor system with a gear box and a load consisting of a large rotating mass coupled to the motor by a rather stiff spring.

$UI(t)$  - induced voltage .

For the mechanical subsystem of the motor, we find:

$$d^2 \text{PHI1}/dt^2 = T/J1 - D1*(d\text{PHI1}/dt)/J1 - F*(\text{PHI1}-\text{PHI2})/J1$$

where:

$T(t)$  - driving torque

$\text{PHI1}(t)$  - angular position of the motor

$\text{PHI2}(t)$  - angular position of the load .

These subsystems are coupled by the equations:

$$UI = \text{PSI} * U_E * d\text{PHI1}/dt$$

$$T = \text{PSI} * U_E * I$$

For the load, we find the equation:

$$d^2 \text{PHI2}/dt^2 = F*(\text{PHI1}-\text{PHI2})/J2 - D2*(d\text{PHI2}/dt)/J2$$

thus, depending on the electrical model, we end up with either a fourth or a fifth order model of this plant. We decided to neglect the armature inductance (which was justified by the fact that the electrical time constant was much smaller than the mechanical ones), and we chose as state variables the outputs of the remaining integrators:

$$X1 = \text{PHI1}$$

$$X2 = d\text{PHI1}/dt$$

$$X3 = \text{PHI2}$$

$$X4 = d\text{PHI2}/dt .$$

In this way, we obtain the mathematical model which describes the DC-motor system:

$$dX1/dt = X2$$

$$dX2/dt = -[ (PSI*UE)^2/(RA*J1) + D1/J1 ]*X2 - F*(X1-X3)/J1 \\ + (PSI*UE*U)/(RA*J1)$$

$$dX3/dt = X4$$

$$dX4/dt = F*(X1-X3)/J2 - D2*X4/J2$$

$$Y = C*X3$$

where C=0.159 is a constant factor that describes the ratio between the position (X3) itself, and the voltage measured from it by means of a rotational potentiometer.

We set:

$$J1 = PA(1)$$

$$D1 = PA(2)$$

$$J2 = PA(3)$$

$$D2 = PA(4) .$$

The NLID program coding this problem is shown below :

```
COMMON/CID2C/MS,MM,NM,XXX(2100),QZ(10),NZ,KI,II,EI,
* NP,SP(50),LP(50),IID,EOP,IDK,EMA(10),
* EMR(10),EMI(10),IERO
COMMON/USER/ R,PSI,UE,F,A21,A22,A41,A44,B2
COMMON/GRAP/ IGRA
IGRA = -1
MS = 1
MM = 1
NM = 25
QZ(1) = 1.
KI = -1
NZ = 2
II = 4
EI = 0.100E-02
NP = 4
SP(1) = 0.9
SP(2) = 4.0
```

Fig. 5-9 NLID program of example 4.

```

SP(3) - 0.02
SP(4) - 0.002
LP(1) - 2
LP(2) - 2
LP(3) - 2
LP(4) - 2
IID - 1
EOP - 0.100E-03
IDK - 0
***
R - 1.81
PSI - 0.06
F - 0.4
UR - 60.
***
DO 10 I-1,NM
10 READ(22,1000) XXX(I),XXX(I+NM),XXX(I+NM+NM)
1000 FORMAT(2X,3I2.4)
CALL NLID
STOP
END
SUBROUTINE START
COMMON/CST2C/PA(50),U(10),TI,XI(30),XMI(30),NORD,
* NM0D,NITR,XEND,XMM(10),IST,IKI
COMMON/USER/ R,PSI,UR,F,A21,A22,A41,A44,B2
NORD = 4
XI(1) - 19.23
XI(2) - 5.2
XI(3) - 3.5/0.159
XI(4) - -5.78
A21 - -F/PA(1)
A22 - -((PSI*UR)**2/(R*PA(1))) + PA(2)/PA(1))
A41 - F/PA(3)
A44 - -PA(4)/PA(3)
B2 - (PSI*UR)/(R*PA(1))
RETURN
END
SUBROUTINE DER
COMMON/DERCOM/T,X(30),XP(30),KEEP
COMMON/CST2C/PA(50),U(10),TI,XI(30),XMI(30),
* NORD,NMOD,NITR,XEND,XMM(10),IST,IKI
COMMON/USER/ R,PSI,UR,F,A21,A22,A41,A44,B2
XP(1) - X(2)
XP(2) - A21*X(1) + A22*X(2) - A21*X(3) + B2*U(1)
XP(3) - X(4)
XP(4) - A41*X(1) - A41*X(3) + A44*X(4)
XMM(1) - 0.159*X(3)
RETURN
END

```

Fig.5-10 depicts graphically the first input signal (IN01). This was used for the identification process. NLID found the following values for the unknown parameters:

J1 = 0.8698  
D1 = 5.2138  
J2 = 0.01631  
D2 = 0.00272

The results from simulation before and after optimization are depicted in Fig. 5-11 and Fig. 5-12 respectively. The first curve (1) depicts the simulated output (OS01), and the second curve (2) graphs the measured output (OM01). As can be seen, there exists a much better match between measured and simulated output after optimization has taken place. For control purposes, we applied the resulting parameter values to the same system once more, and simulated it (in this case, we set IID=0) using another input signal (IN02) as shown in Fig. 5-13.

The simulation results are graphed in Fig.5-14. They verify that the parameter values found are correct as there can still be noticed a very good match between measured (OM02) and simulated (OS02) output.

#### 5.6 Lotka-Volterra Model

Example 5. We want to identify the parameters of a Lotka-Volterra model applied to a population dynamics problem for the insect population of the larch bud moths

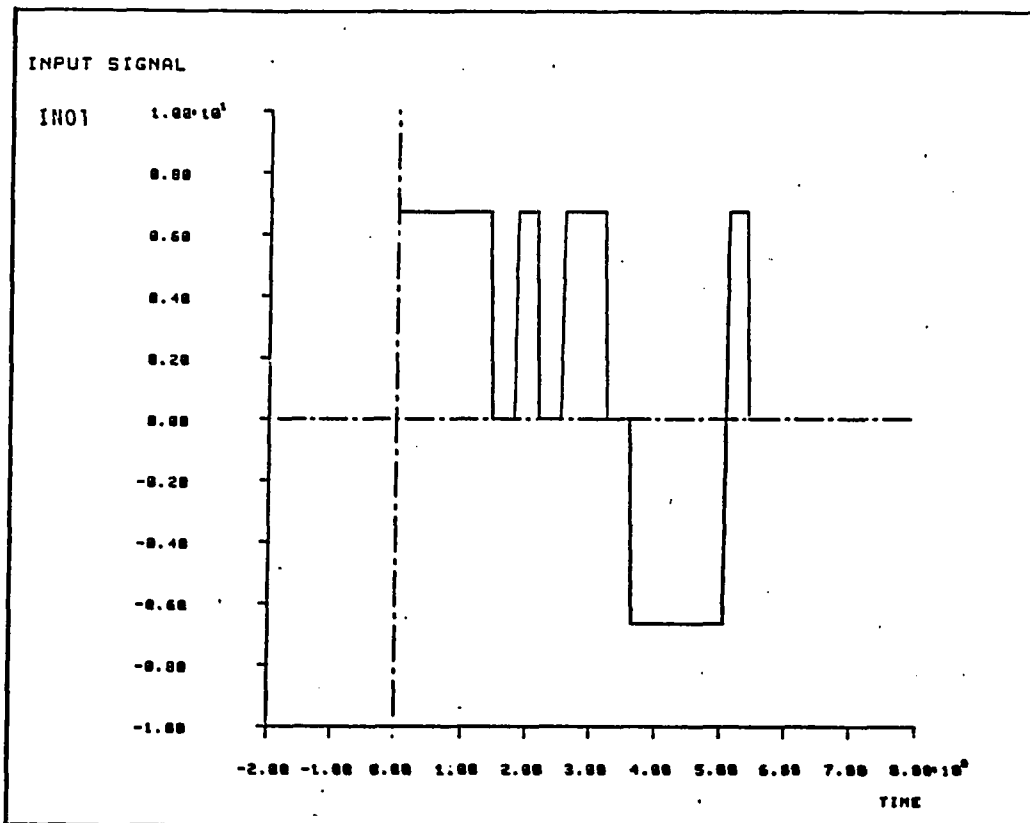


Fig. 8-10 Input signal used during the optimization.

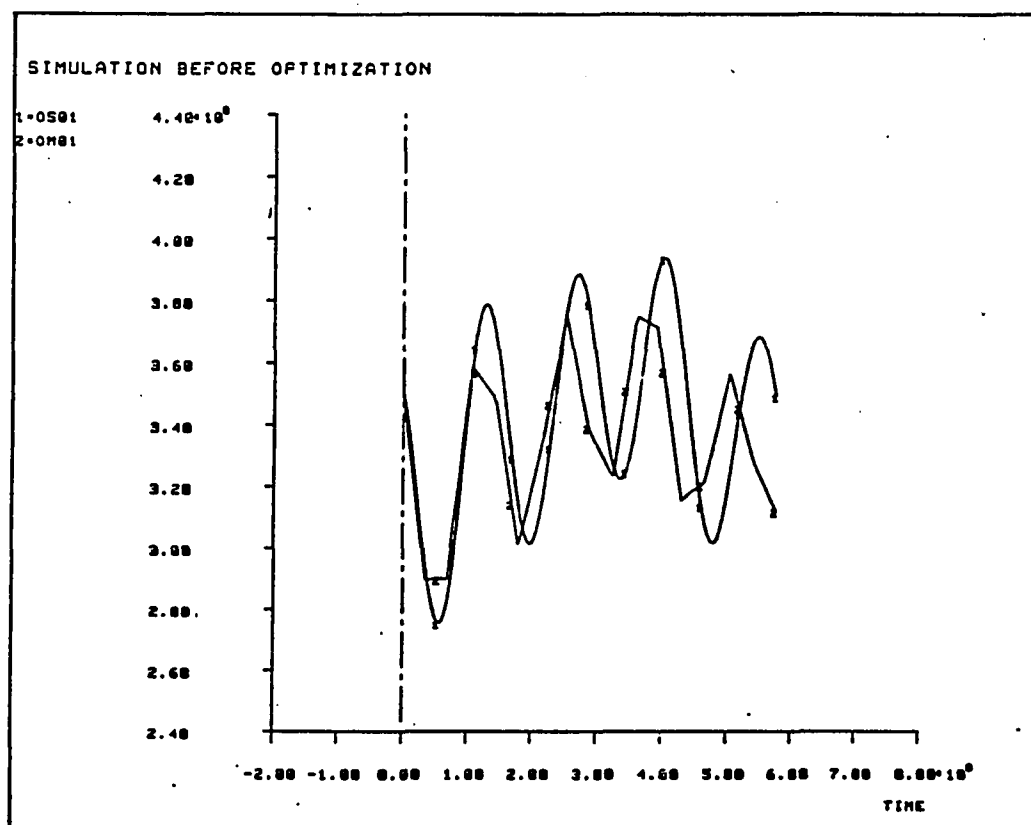


Fig. 5-11 Simulation of plant with initial parameter values.

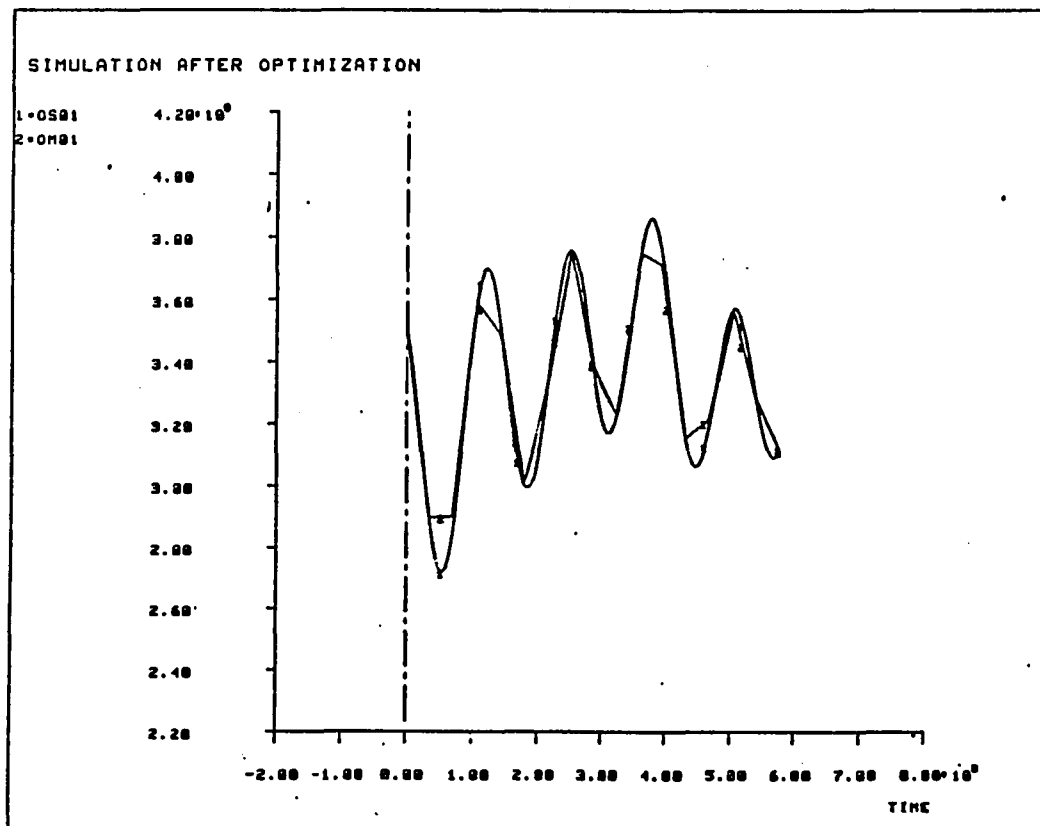


Fig. 5-12      Simulation of plant with optimized parameter values.

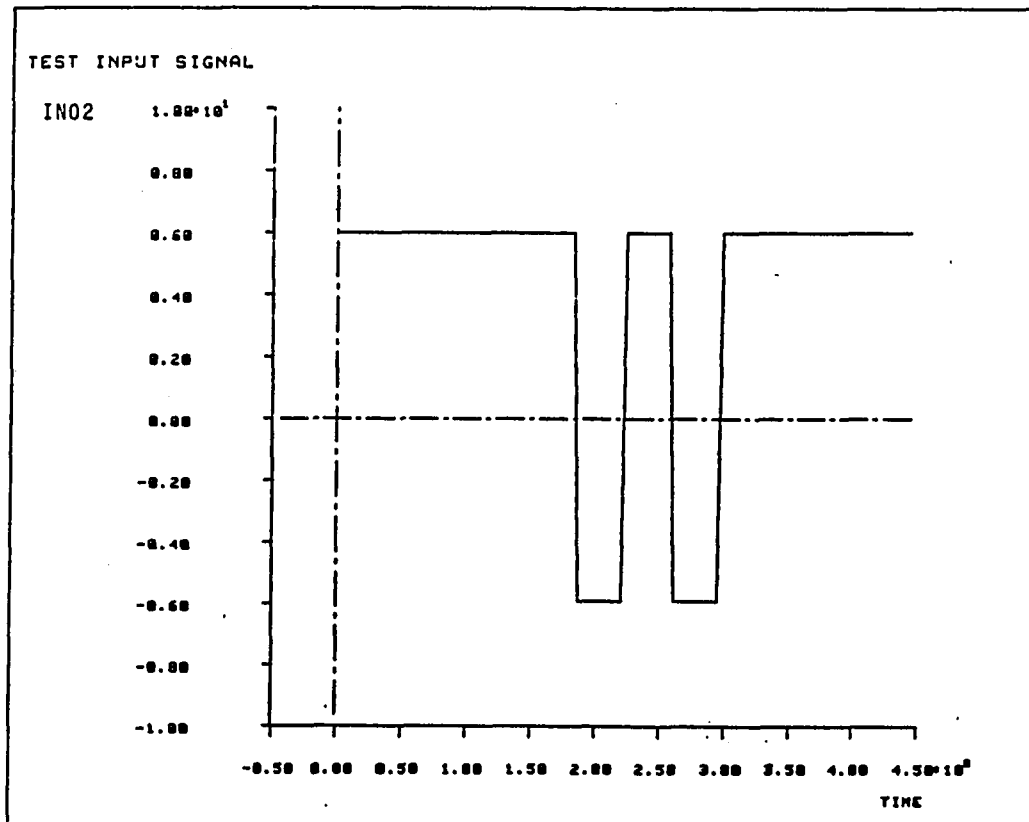


Fig. 5-13 Control input signal.

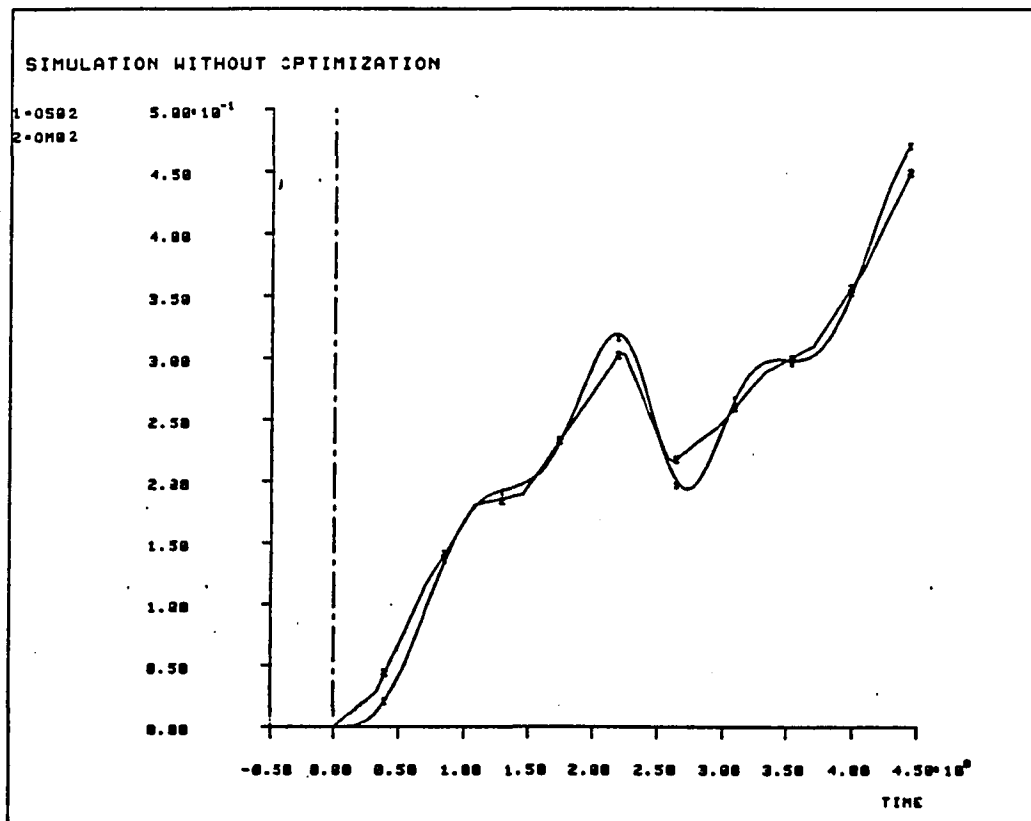


Fig. 5-14 Results from control simulation run.

in the Engadine Valley (Southern Switzerland).[7] The model takes the following form:

$$\begin{aligned} dX_1/dt &= aX_1 - bX_1X_2, & X_1(t=0) &= X_{10} \\ dX_2/dt &= -cX_2 + dX_1X_2, & X_2(t=0) &= X_{20} . \end{aligned}$$

where  $X_1$  describes the population of the prey (larch), and where  $X_2$  describes the population of the herbivorous consumer (larch bud moth). The parameter  $a$  describes the exponential growth of the prey population.  $c$  describes self-inhibition of the predator population. The parameters  $b$  and  $d$  describe the interaction between the two species (if predator eats prey, prey is eaten which decreases the prey population, and increases the predator population).

We want to use the identification procedure to determine the four parameters  $a$ ,  $b$ ,  $c$ , and  $d$  as well as the two initial conditions. The insects are expressed in number of larvae per kilogram branches, whereas the prey is expressed in kilograms of needed biomass.

However before doing so, we want to analyze our model a little more closely. We can apply a linear variable transformation:

$$\begin{aligned} X_1 &= (c/d)Y_1 \\ X_2 &= (a/b)Y_2 . \end{aligned}$$

This leads to a modified model:

$$\begin{aligned} Y_1 &= a(1 - Y_2)Y_1 \\ Y_2 &= -c(1 - Y_1)Y_2 . \end{aligned}$$

In this new model, we have only two parameters left.

Obviously, the parameters  $b$  and  $d$  do not influence the shape of the state trajectories. They are just multiplying factors of the output equations for computation of the physical variables  $X$  out of the internal state variables  $Y$ . With measurements of one (the predator) population alone, we can thus certainly not hope to identify all 4 parameters. Instead, we apply the modified model which reduces the number of parameters by one.

With this model we can then construct the program along with a data file containing measured data of the populations of larch and larch bud moths during some time period. For the optimization, we used the predator population data only. The prey population data were kept to check the results of the identification.

Note that the model is unstable if the two initial conditions are not in the first quadrant. As these initial conditions are parameters, we thus have a constrained optimization problem to solve. We chose the method of transforming the parameter space by applying a logarithmic transformation. In NLID, this can be achieved completely automatically, by setting both  $LP(1)$  and  $LP(2)$  equal to 2.

The program of this problem is shown below :

```

COMMON/CID2C/MS,MM,NM,XXX(2100),QZ(10),NZ,KI,II,EI,
* NP,SP(50),LP(50),IID,EOP,IDK,EMA(10),
* EMR(10),EMI(10),IERO
MS = 1
MM = 2
NM = 30
NZ = 2
II = 4
EI = 0.001
QZ(1) = 1.
QZ(2) = 1.
NP = 5
SP(1) = 0.2821
SP(2) = 0.02079
SP(3) = 2.653
SP(4) = 0.7869
SP(5) = 0.02394
DO 10 I=1,5
LP(I) = 2
10 CONTINUE
IID = 1
EOP = 1.E-4
IDK = 0
DO 20 I=1,NM
20 READ(22,1000) XXX(I),XXX(I+NM),XXX(I+2*NM),
*XXX(I+3*NM)
1000 FORMAT(2X,4E12.4)
CALL NLID
STOP
END
SUBROUTINE START
COMMON/CST2C/PA(50),U(10),TI,XI(30),XMI(30),NORD,
* NMOD,NITER,XEND,XMM(10),IST,IKI
NORD = 2
XI(1) = PA(4)
XI(2) = PA(5)
RETURN
END
SUBROUTINE DER
COMMON/DERCOM/T,X(30),XP(30),KEEP
COMMON/CST2C/PA(50),U(10),TI,XI(30),XMI(30),NORD,
* NMOD,NITER,XEND,XMM(10),IST,IKI
XP(1) = PA(1)*(1.-X(2))*X(1)
XP(2) = -PA(3)*(1.-X(1))*X(2)
XMM(1) = PA(1)/PA(2)*X(2)
RETURN
END

```

Fig. 5-15 NLID program of example 5.

In Fig. 5-17, the results of a simulation with the initial parameter values together with the measured data are depicted. Fig. 5-18 shows the same curves applying the optimized parameter values. The following parameter values were found:

```

FP      = 0.1375347E+5.
PA(1)  = 4.2748E-1
PA(2)  = 6.9623E-3
PA(3)  = 1.6977E0
PA(4)  = 5.9161E-1
PA(5)  = 5.1581E-2

```

Fig. 5-16 Results of example 5.

The minimized performance function (FP) is still very large although a satisfactory match between the simulated and the measured output curves can be noticed. This is alarming, and probably points to structural deficiencies of the model.

The model resulting from this identification process is:

$$Y1 = 0.42748*(1. - Y2)*Y1$$

$$Y2 = -1.6977*(1. - Y1)*Y2$$

Indeed, according to Fischlin and Baltensweiler [10], this model does not describe the tree population very well. This is because there exist additional factors influencing the behavior of the system. Hence, some other processes than grazing and physiological reactions would have to be included into the model to obtain more

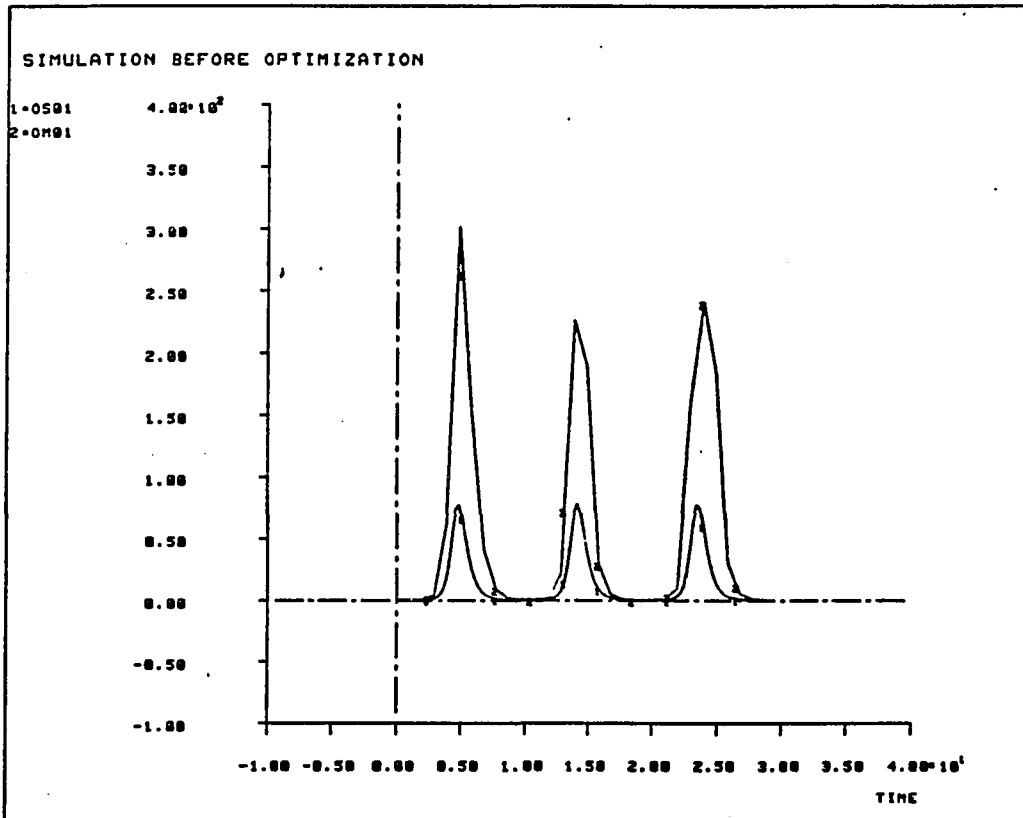


Fig. 5-17 Simulated and measured trajectories of predator population using initial parameter values.

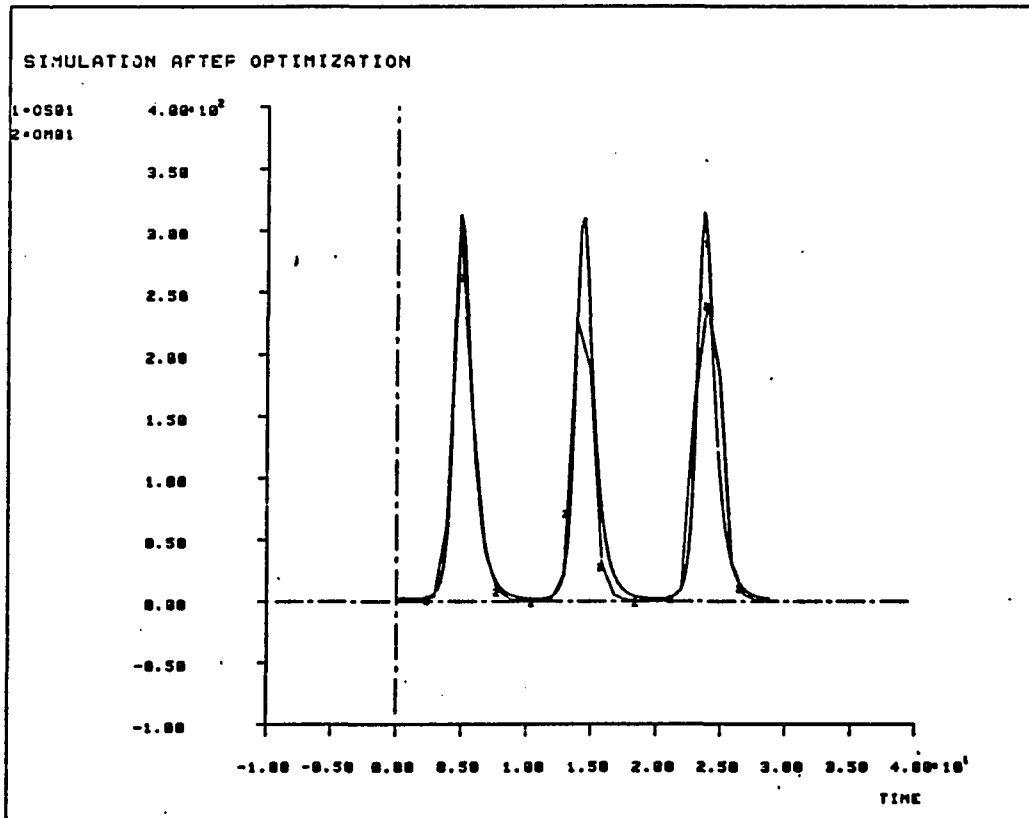


Fig. 5-18 Simulated and measured trajectories of predator population using optimized parameter values.

realistic behavior. One such model proposed by Fischlin and Baltensweiler [10] separates the valley into different regions (compartments), and includes the insect migration between the compartments as additional factor. This model is able to reproduce the system behavior much better, yet it is dangerous to conclude from there that the model is physically "correct", as it contains additional parameters. (It is a fact that a model can reproduce the behavior of almost any system satisfactorily if the number of parameters is chosen sufficiently large.)

In general, we face several types of problems with interpreting the outcome of an optimization study.

- (1) We can rarely be sure that the algorithm has led us to the global optimum. We could as well be stranded in a side valley.
- (2) Even if we have found the global optimum, it is often impossible to ensure that the model is truly correct, as the model could still have structural deficiencies (such as neglected parameters that would also be of influence).

CHAPTER 6  
CONNECTING NLP WITH DARE/INTERACTIVE

6.1 Introducing DARE/INTERACTIVE

None of the existing continuous system simulation languages has an optimization package associated with it that can be used as an integral part of the software. We thus try to combine NLP with the simulation language DARE/INTERACTIVE to create such an environment. This should enable us to enhance the experiment description capabilities of DARE/INTERACTIVE drastically.

As shown before, NLP allows us to do dynamical optimizations in the form of parameter identifications by itself. However, this tool is not powerful enough, as there exist other dynamic optimization problems as well (e.g. the solution of boundary value problems), and as the integration software within NLID is not very sophisticated (e.g. unsuited for stiff problems).

In the integration of NLP with DARE/INTERACTIVE, we shall make use of a subset of the NLP package only, in that the NLID entry is totally duplicated within DARE and thus useless, and in that the graphical file generation capabilities of NLP are also duplicated within DARE. In fact, as we used the DARE graphics processor even for NLP

alone, the graphics file generation of NLP and DARE interfere with each other. Thus, one should never use NLP together with DARE while persuading NLP to generate graphics files. The IGRA parameter of NLP must, therefore, always be set to (-1) when used together with DARE.

As a first step, we shall construct a loose interface between the two programs in that the user is asked to call the optimization software from within the description of his simulation experiment (LOGIC block of DARE). In a later step, it would be very feasible, to provide standard experiments for parameter identification, steady-state finding, boundary value problems, etc. that would make the user programs shorter, and thus the life of the simulation user easier.

Fig.6-1 shows the DARE/INTERACTIVE program structure. The software has three different modes in which the user can interact with the system [6]:

- 1) the modelling monitor: in which the user specifies his model to the system
- 2) the run-time monitor: in which the user describes his experiment (e.g. parameter values, run-time display, integration algorithm, type of experiment (normal simulation, range analysis, replication analysis).
- 3) the graphics monitor: in which the user interacts with the simulation data-base to view results of one or several previously performed simulations.

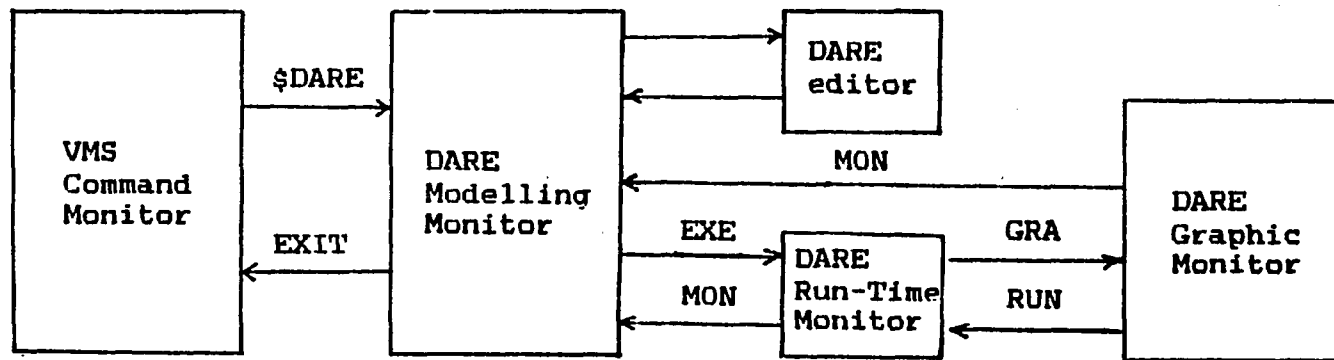


Fig. 6-1: Program structure of DARE/INTERACTIVE

Fig.6-2 shows how NLP is to be connected with DARE/INTERACTIVE. DARE/INTERACTIVE lends itself better to this type of an interface than other Continuous System Simulation Languages (CSSL's) such as ACSL or CSSL-IV. The reason for this is that DARE allows the user to specify explicitly from within the LOGIC block (which is basically an ordinary FORTRAN program) when a simulation is to be executed (by using the CALL RUN statement). In this way, the simulation is called as a subroutine which can easily happen deeply within NLP as well. In this way, the optimization program can call the simulation program from within the performance function evaluation (as it should be). Contrary to this situation, most of the CSSL's follow the original CSSL standard in which the simulation program is divided into three segments, an INITIAL segment to be performed prior to the simulation, a DYNAMIC segment which performs the actual simulation, and a TERMINAL segment which is executed following the simulation for final computations and output. Eventhough many CSSL's allow to jump from the TERMINAL segment back to the INITIAL segment for iterative execution of the simulation program, it is here actually the simulation that triggers an iteration, a very unnatural structure for optimization analysis. Eventhough we may still be able to perform optimizations, we would have to code our own optimization algorithms

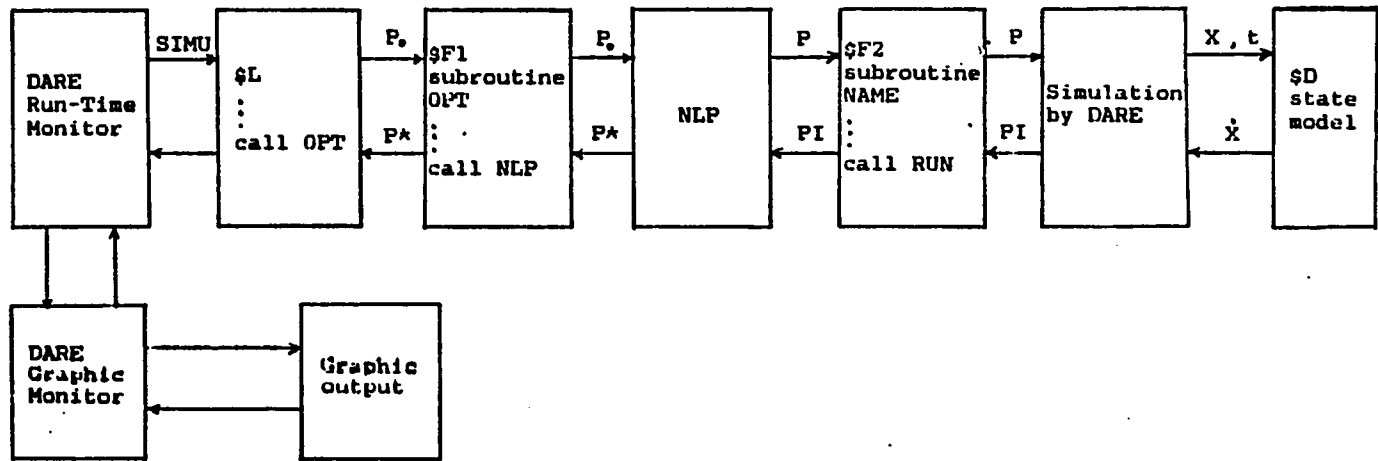


Fig. 6-2: Connection of NLP and DARE/INTERACTIVE

within the simulation program, and could not rely on a previously coded and tested optimization package.

Let us first discuss, how DARE/INTERACTIVE can be operated under VAX/VMS. The procedure of executing a problem in DARE/INTERACTIVE is the following:

- 1) Type DARE while being in the VMS command monitor mode. This will start DARE/INTERACTIVE in its modelling monitor mode. In this mode, the user can edit the problem to be executed. A DARE program is composed of several files with the same file name but different extensions. Each file contains one or several specific DARE blocks or information needed for the DARE system. The file types and formats are shown in the following:
  - a) filename.AFL: This is the "catalog" (directory) of the problem. It contains the extensions of all files belonging to the problem. This file is system maintained.
  - b) filename.DDI: This file contains information about the variables to be depicted in the run-time display. Also this file is system maintained.
  - c) filename.DDL: This user maintained file contains two of the blocks of the DARE/INTERACTIVE program, namely the \$D1-block (DYNAMIC block) which describes the system model through state equations, and the \$L-block (LOGIC block) describing the experiment of the problem.

Eventually, also a \$D2-block may be present. The format is exactly the same as in DARE-P, with the exception of the "CALL RESET" statement which was modified to "CALL REINIT", as RESET is a system subroutine. Each user maintained file is to be terminated by an END statement (in columns 1 .. 3).

- d) filename.DFi: where i stands for any alphanumerical character. These files allow the user to write his own FORTRAN functions and subroutines needed for the program (\$Fi-blocks). The format is the same as in DARE-P. The \$Fi-statement must also be present. Several subroutines can be placed in one file, by separating them by a \$Fi-statement. One new feature as compared to DARE-P is the introduction of the COMMON statement (in columns 1 .. 6 immediately following the subprogram header. This statement shall be interpreted as a comment by FORTRAN (as it starts with a "C" in column 1, but it shall trigger the preprocessor to insert the header information of the DARE program, thus making all the simulation variables (state variables, algebraic variables, parameters) available to the subroutine.
- e) filename.DIM: This system maintained file

remembers the integration method last used for execution of the problem. In DARE/INTERACTIVE, the integration method is determined at run time, thus no \$Mi-block exists in this system.

- f) filename.DOW: This user maintained file contains the \$O-block of DARE-P for coding a user integration algorithm.
- g) filename.DPA: This file contains the parameter values to be used. This is equivalent to the specifications in DARE-P following the first END statement. However, this file is system maintained. During the first compilation, the file is generated with values of UNDEFINED for all parameters. Prior to the first simulation, the system prompts the user for values of all parameters which can thereafter be modified interactively at any time.
- h) filename.DTi: These files contain one or several \$Ti-blocks each. In our applications, we shall use these files to store measurement data needed for the curve fitting process.
- i) filename.DMi: These user maintained files contain command macros, that is: series of DARE commands to be executed one after the other in the form of a canned experiment. The DARE/INTERACTIVE command monitor takes the role of the previously described

development systems, but is decoupled from the MIDGET operating system. (In fact, DARE/INTERACTIVE is older than MIDGET. It led conceptually to the design of the MIDGET development systems.) All the monitor commands are commands to manipulate files in one way or the other.

- j) filename.DNB: A notebook file to jot down anything that needs to be remembered about the particular problem (a scratch pad).
  - k) filename.DUT: A user maintained file containing output commands, corresponding to those of the DARE-P system, but largely extended. Such a file is optional, as graphics commands can be entered interactively in the graphics monitor mode.
- 2) To edit any of the above file, the user specifies the command: "EDIT extension" where the (common) "D" character can be left out, thus e.g. "EDIT F1" or "EDIT DL". The methodology of DARE/INTERACTIVE is such that at compilation, new files are produced which differ from the source code by the first character of the extension, thus: "filename.FDL" would denote the preprocessed version of the DL-file (that is: a set of FORTRAN subroutines, while "filename.OF1" would be the object code of the FORTRAN compiled F1-file. After the DARE program has been edited, the user can type the

command RUN to proceed to the run-time monitor by compiling all those files that need compilation, and by linking them together to an executable image file. DARE/INTERACTIVE maintains in its directory file (filename.AFL) information concerning the compilation level. In the run-time monitor mode, the user operates on the simulation experiment by modifying parameters, specifying variables to be included in the self scaling run-time display, and performing specific types of experiments such as ordinary simulation (execution of the \$L-block or more specific types of experiments such as RANGE analysis, REPLICATION analysis, ANTITHETIC VARIATES analysis, etc., experiments that consist of precoded \$L-blocks that replace the user specified \$L-block. The command SIMULATE starts the execution of the simulation, by calling the compiled version of the user specified \$L-block (or a precoded one, if the user has not specified any \$L-block at all). If some parameters have a value of UNDEFINED, the system first prompts the user for values of these parameters. During compilation, the parameter file is checked for consistency, thus if the user modifies his model, parameters no longer present in the model will automatically be deleted from the parameter file, while new parameters will be added with a value of

UNDEFINED.

- 3) Once the program has been executed, four data files have been generated by the DARE system (unless the user has specified the /FLY-flag on the experiment execution statement, e.g. "SIMULATE /FLY" or "RANGE /FLY, etc.). These files are, TIME.DAT, MONIT.DAT, SAVE.DAT, and CROSS.DAT. These data files contain the output data needed for the graphics processor. The run-time monitor command GRAPHIC allows you to switch from the run-time monitor mode to the graphics monitor mode.
- 4) The graphics monitor mode allows us to look at our simulation data by producing tables and plots. The commands are similar to those of the DARE-P system, but many new features have been added, e.g. an ENVELOPE command to graph the envelope of all trajectories found in a RANGE analysis, etc. If a "filename.DUT" file was specified, this file is executed automatically as a graphics command macro whenever the graphics monitor is entered. After completion of the last graph, the system returns to interactive mode to expect further output commands to be types in through the keyboard. After each (true) graph produced at the terminal, the system waits for a carriage return before it goes on to destroy the graph and draw the next one. To exit from the graphic

monitor, type RUN to return to the run-time monitor, or MONIT to return to the modelling monitor, or EXIT to return to VMS.

This introduction was needed, as there does not exist yet a manual for DARE/INTERACTIVE (although INTERACTIVE HELP information is available in the system in abundance).

### 6.2 Examples

In this section we want to use two examples to show how the DARE/INTERACTIVE system can be used in connection with NLP.

Example 1: This example is a parameter identification problem (NLID application) copied from the NLP user's guide [26]. The problem is stated as follow:

An input response of a single-input, single-output process has been measured as shown in Fig. 6-3. The system model is described by the following state and output equations with  $p_1$ ,  $p_2$  and  $p_3$  being the unknown parameters to be identified.

$$dx_1(t)/dt = p_3 \cdot 0.666666 \cdot U(t) - p_1 \cdot x_1(t) ,$$

$$dx_2(t)/dt = p_1 \cdot x_1(t) - p_2 \cdot x_2(t) ,$$

$$dx_3(t)/dt = p_2 \cdot x_2(t) - p_3 \cdot x_3(t) ,$$

$$y(t) = x_3(t) .$$

The initial conditions are:

$$x_1(0) = y(0) \cdot p_3 / p_1 ,$$

$$x_2(0) = y(0) \cdot p_3 / p_2 ,$$

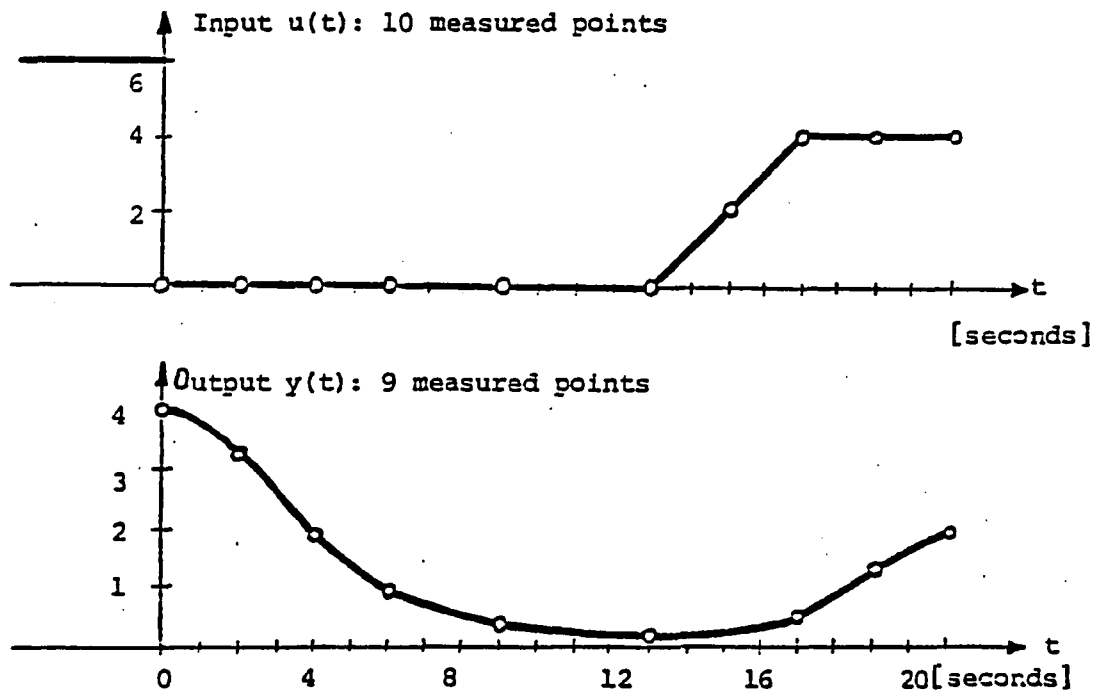


Fig. 6-3 Measured input and output of a process.

$$x_3(0) = y(0)$$

The DARE/INTERACTIVE program for this problem is as follows:

file : NLID01.DDL

```
$D1
X1. - PP3*0.666666666*UMEAS(T) - PP1*X1
X2. - PP1*X1 - PP2*X2
X3. - PP2*X2 - PP3*X3
Y - X3
YM - YMEAS(T)
PI. - (Y - YM)**2
$L
INPUT PPI1,PPI2,PPI3
CALL OPT
END
```

file : NLID01.DF1

```
$F1
SUBROUTINE OPT
COMMON
COMMON/NLPP/NP,PA(40),FMIN,FTOL,IDER,IOU,NFE,
*      FETOL(40),NFI,FITOL(40),FAC,IND,FRES,
*      FERES(40),FIRES(40)
COMMON/GRAP/IGRA
INTEGER*4 NAME1,NP,IDER,IOU,NFE,NFI,IND,IGRA
EXTERNAL NAME1
NP = 3
CALL REINIT
PA(1) = PPI1
PA(2) = PPI2
PA(3) = PPI3
CALL NAME1
CALL STROF
FMIN = 1.E10
FTOL = -1.E-5
IDER = 0
IOU = 2
NFE = 0.
NFI = 0.
IGRA = -1
CALL NLP (NAME1)
CALL STRON
```

Fig. 6-4 DARE/INTERACTIVE program of example 1.

```
CALL NAME1
RETURN
END
END

file : NLID01.DF2

$F2
SUBROUTINE NAME1
COMMON
COMMON/FUNCT/IFUN,P(40),F,FIE(40),DF(40),
*DFIE(1600),IERR
INTEGER*4 IFUN,IERR
PP1 = P(1)
PP2 = P(2)
PP3 = P(3)
CALL REINIT
XMI = YMEAS(0.)
X1 = XMI*PP3/PP1
X2 = XMI*PP3/PP2
X3 = XMI
CALL SAVE
CALL RUN
F=PI
CALL CROSS
RETURN
END
END

file : NLID01.DT1

$T1
UMEAS,10
0.0, 0.0
2.0, 0.0
4.0, 0.0
6.0, 0.0
9.0, 0.0
13.0, 0.0
15.0, 2.0
17.0, 4.0
19.0, 4.0
21.0, 4.0
END
```

Fig. 6-4 --Continued.

file : NLID01.DT2

```

$T2
  YMEAS,10
    0.0, 4.0
    2.0, 3.2
    4.0, 1.9
    6.0, 1.0
    9.0, 0.4
   13.0, 0.1
   15.0, 0.3
   17.0, 0.6
   19.0, 1.3
   21.0, 2.0
END

```

Fig. 6-4 --Continued.

Note the INTEGER\*4 statements declaring all integer variables of the NLP program. This is necessary as DARE/INTERACTIVE is compiled with the /NOI4 option (in which each INTEGER occupies only 16 bits) while NLP is compiled without option (with each INTEGER occupying 32 bits). The graphic output for this problem produced by the DARE/INTERACTIVE graphics monitor is shown in Fig. 6-5 which is very similar to the output form NLID shown in Fig. 6-6, the first curve (Y,OS01) represents the simulated, the second (YM,OM01) is the measured output.

Example 2: Here we want to solve the boundary value problem mentioned in chapter 5 by the approach of the shooting method. The problem as specified in chapter 5 was:

$$d^2T/dx^2 = c1*(T^4 - c2) ,$$

with  $x \in [0,1]$ ,  $T(0) = 100.$ ,  $c1 = 1.E-4$ ,  $c2 = 0.$ ,

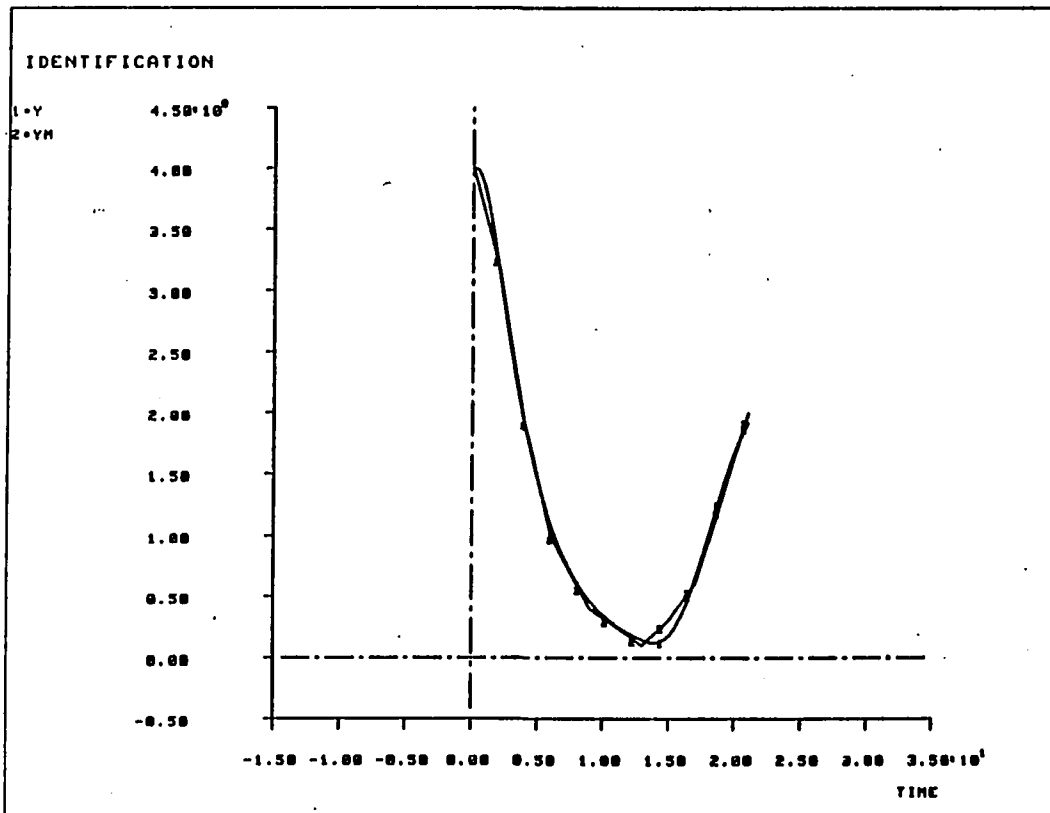


Fig. 6-5: Parameter identification in DARE/INTERACTIVE

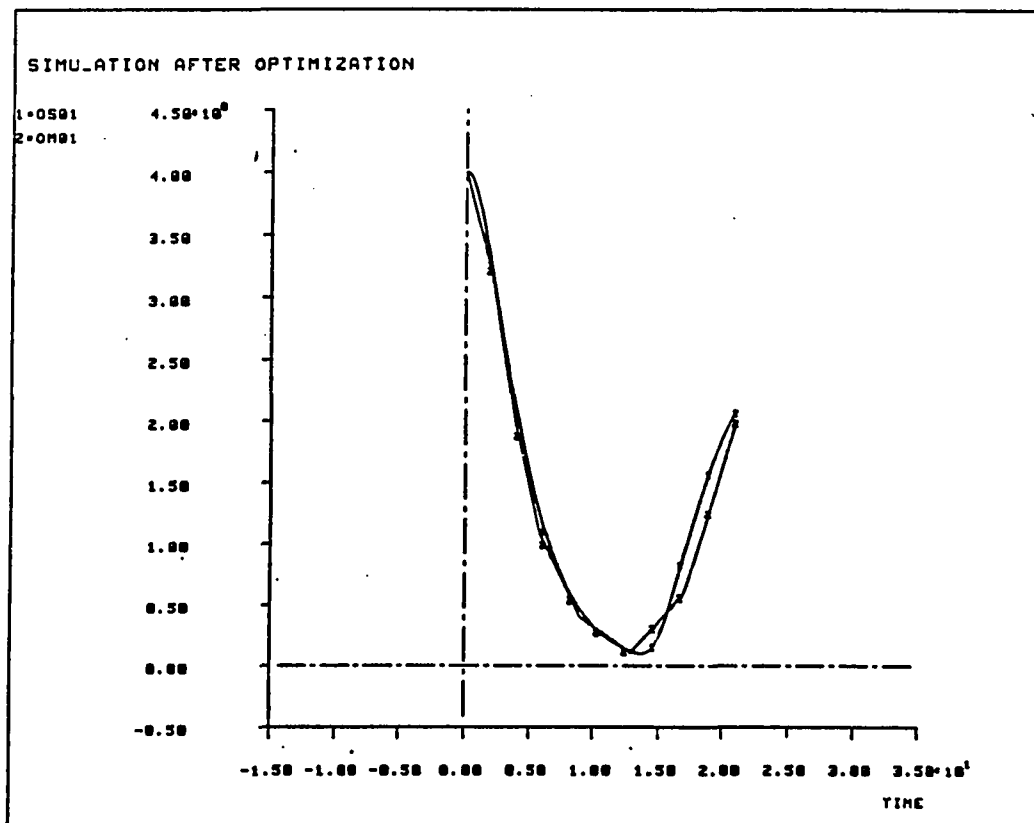


Fig. 6-6: Parameter identification using NLID

$dT/dx(1) = 0$ . The approach is the following. First, we transform the second order ODE into a set of two first order ODE's:

$$\begin{aligned}dT/dx &= V \\dV/dx &= c1*(T^4 - c2) .\end{aligned}$$

The initial conditions are now as follows:

$$\begin{aligned}T(0) &= 100 \\V(0) &= p(1) = \text{unknown} .\end{aligned}$$

We thus use the unknown initial condition as the one and only parameter of this problem. We then let the square of the variable  $V(1)$  be the performance function. We would like to have  $V(1)=0$ . Thus by minimizing  $V(1)$ , we optimize the performance of the system, and indirectly solve our posed boundary value problem.

When we try to run this program in DARE/INTERACTIVE, we realize that the differential equation is numerically unstable. Thus, we have to reformulate the problem slightly. We integrate backward from  $x=1$  to  $x=0$ . We use:

$$\begin{aligned}T(1) &= p(1) = \text{unknown} \\V(1) &= 0.0\end{aligned}$$

as the initial conditions, and then choose as the performance index:

$$PI = (T(0)-100.0)**2 .$$

The DARE/INTERACTIVE program for this problem is listed below:

```

                file : DARE01.DDL

$D1
  C1 = 1.E-4
  X1. = -X2
  X2. = -C1*X1**4

$L
  INPUT PO
  OUTPUT PI,PP1
  CALL OPT

END

                file : DARE01.DF1

$F1
  SUBROUTINE OPT
COMMON
  COMMON/NLPP/NP, PA(40), FMIN, FTOL, IDER, IOU, NFE,
  *          FETOL(40), NFI, FITOL(40), FAC, IND, FRES,
  *          FERES(40), FIRES(40)
  COMMON/GRAP/IGRA
  INTEGER*4 NAME1, NP, IDER, IOU, NFE, NFI, IND, IGRA
  EXTERNAL NAME1
  NP = 1
  CALL REINIT
  PA(1) = PO
  CALL NAME1
  CALL STROF
  FMIN = 1.E10
  FTOL = -1.E-5
  IDER = 0
  IOU = 2
  NFE = 0
  NFI = 0
  IGRA = -1
  CALL NLP (NAME1)
  CALL STRON
  CALL NAME1
  RETURN
  END

END

```

Fig. 6-7 DARE/INTERACTIVE program of example 2.

file : DARE01.DF2

```
$F2
SUBROUTINE NAME1
COMMON
  COMMON/FUNCT/IFUN,P(40),F,FIE(40),DF(40),
  * DFIE(1600),IERR
  INTEGER*4 IFUN,IERR
  PP1 = P(1)
  CALL REINIT
  X1 = PP1
  X2 = 0.0
  CALL SAVE
  CALL RUN
  PI = (X1-100.)**2
  F = PI
  CALL CROSS
  RETURN
END
END
```

Fig. 6-7 --Continued.

The optimization returns a value of:

$$P(1) = 25.7$$

The graphic result is shown in Fig.6-8 which is comparable to the output in example 1 (Fig.5-1,chapter 5). However, the graphic output from this example looks smoother than the other two. This is due to the fact that we discretized the x-axis using 10 intervals in those previous two examples, whereas we did not need to discretize it in this example. Hence, the results obtained are more accurate.

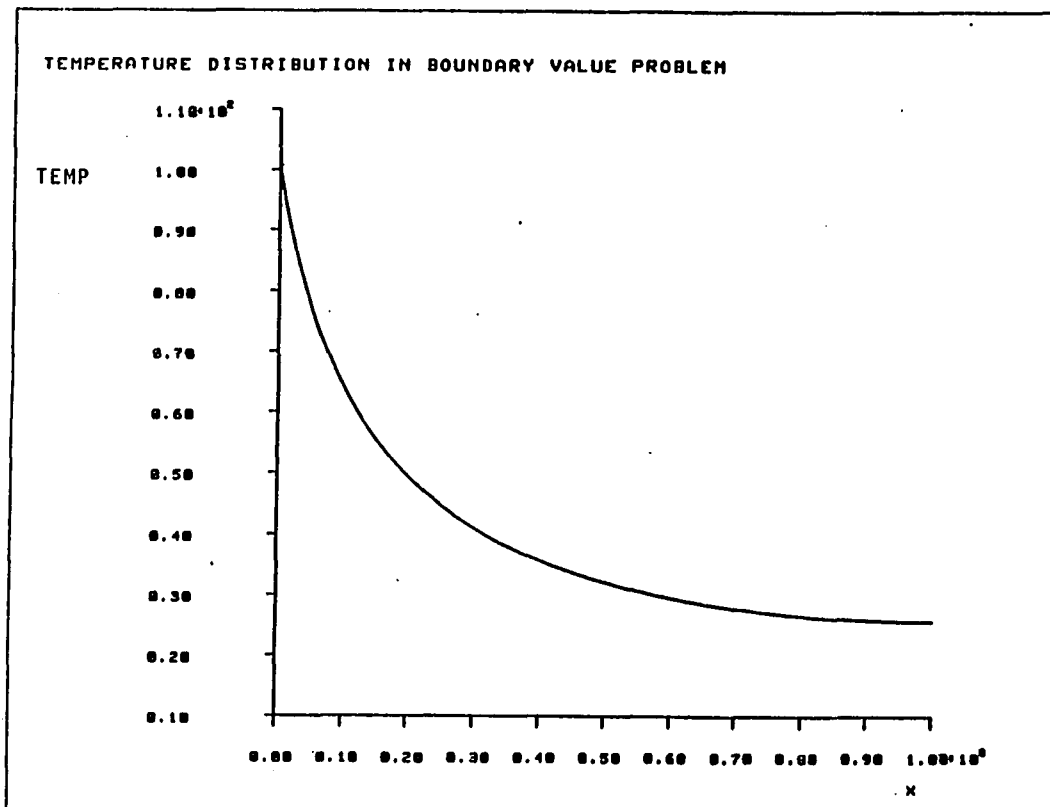


Fig. 6-8 Graphic output of example 2.

## CHAPTER 7

### DISCUSSION OF BOUNDARY VALUE PROBLEMS

As mentioned before, we usually have several different methods available for solving a boundary value problem. We have already solved the same boundary value problem twice, namely by use of a shooting technique (this was done in Example 2 in Chapter 6), whereas a static optimization approach has been shown in Example 1 of Chapter 5. A third technique would be to use the result from invariant embedding directly to solve the resulting PDE problem to steady-state.

Thus, the distinction between ANALYSIS and SYNTHESIS problems is not as clear cut as most people like to see it. One and the same problem can often be viewed either as an synthesis problem (inverse problem) or as a (more complicated) analysis problem (direct problem). The conversion between these two ways of viewing the problem is called INVARIANT EMBEDDING.

In our example, we can solve this boundary value problem either as:

- 1) A static optimization problem in many parameters,
  - 2) A dynamic optimization problem in only one parameter,
- or
- 3) A pure simulation problem in distributed parameters.

It seems now worthwhile to investigate which of the three approaches is numerically best suited. In this chapter, we want to answer this question, by solving the same problem once more, this time using the third technique. We apply the FORSIM-VI program (remember that this problem actually stems from the FORSIM manual). Of course, we are going to use the FORSIM DEVELOPMENT SYSTEM which is yet another one of these user interfaces provided in MIDGET. FORSIM is designed to solve PDE's by applying the method of lines. The space axes are discretized, and spatial derivatives are approximated by finite differences, while the time axis is kept continuous. The resulting state equations are integrated over time until they reach their steady-state.

### 7.1 Invariant Embedding Method

Example 1: The problem we deal with is the following:

$$d^2T/dx^2 = c1*(T^4 - c2) .$$

To solve this problem, we introduce a time derivative  $dT/dt$  to convert the equation into a partial differential equation of the parabolic type, and integrate this PDE until it converge to its steady state. Hence, we have:

$$dT/dt = -d^2T/dx^2 + c1*(T^4 - c2) .$$

Using this partial differential equation, we can write a FORSIM program as follows:

```

SUBROUTINE UPDATE
COMMON /RESERV/ TIME,STEP
*      /CNTROL/ INOUT
*      /INTEGT/ T(11)
*      /DERIVT/ DT(11)
*      /DERVX/  TX(11)
*      /DERVXX/ TXX(11)
*      /PARTB/  BC(4,2,1)
IF (TIME.NE.O.ODO) GO TO 100
NPPOINT = 11
NPDE = 1
C = 1.0D-4
DO 50 I=1,NPOINT
50 T(I) = 100.ODO
   BC(3,1,1) = 100.ODO
   BC(1,2,1) = 1.ODO
   BC(2,2,1) = 0.ODO
100 CALL PARSET (NPDE,NPOINT,T,DT,TX,TXX)
   DO 120 I=1,NPOINT
120 DT(I) = TXX(I) - C*T(I)**4
   CALL PARFIN (NPDE,NPOINT,T,DT)
   IF (INOUT.EQ.0) RETURN
   CALL RITER (X,10HCOORDINATE)
   CALL RITER (T,4HTEMP)
RETURN
END

```

Fig. 7-1 FORSIM program of the boundary value problem of example 1.

The following data file is needed for this program:

```

TEST
*NOPARS*
*FINIT*

```

The result of this problem run by FORSIM was found as follows:

```

TEMP(x=0.1) = 0.6590D+2
TEMP(x=0.2) = 0.5065D+2
TEMP(x=0.3) = 0.4199D+2
TEMP(x=0.4) = 0.3643D+2
TEMP(x=0.5) = 0.3264D+2
TEMP(x=0.6) = 0.2997D+2
TEMP(x=0.7) = 0.2812D+2
TEMP(x=0.8) = 0.2689D+2
TEMP(x=0.9) = 0.2619D+2
TEMP(x=1.0) = 0.2595D+2
CPU-TIME    = 0.814D+1

```

Fig. 7-2 Results of the boundary value problem solved by FORSIM.

### 7.2 Discussion

We have used three different methods to solve the same boundary value problem. All results are listed below again.

	OPTIMIZATION APPROACH	INVARIANT EMBEDDING	SHOOTING METHOD
TEMP(x=0.1)	0.6589E+2	0.6590D+2	0.6414E+2
TEMP(x=0.2)	0.5064E+2	0.5065D+2	0.4935E+2
TEMP(x=0.3)	0.4197E+2	0.4199D+2	0.4105E+2
TEMP(x=0.4)	0.3641E+2	0.3643D+2	0.3573E+2
TEMP(x=0.5)	0.3261E+2	0.3264D+2	0.3210E+2
TEMP(x=0.6)	0.2995E+2	0.2997D+2	0.2955E+2
TEMP(x=0.7)	0.2810E+2	0.2812D+2	0.2777E+2
TEMP(x=0.8)	0.2687E+2	0.2689D+2	0.2659E+2
TEMP(x=0.9)	0.2617E+2	0.2619D+2	0.2592E+2
TEMP(x=1.0)	0.2594E+2	0.2595D+2	0.2570E+2
CPU-TIME(sec)	4.47	8.14	20.44

Fig. 7-3 Results of three different approaches for the same boundary value problem.

We would like to compare the results with respect to convenience, robustness, accuracy, and efficiency.

With respect to CONVENIENCE, the shooting technique is certainly the most easy one to apply, as it does not require any reformulation of the problem. Once NLP shall be completely integrated into DARE/INTERACTIVE, the user program will be also the shortest one.

With respect to ROBUSTNESS, all three methods have the same deficiency. They all are unstable if not formulated carefully. The shooting problem had to be integrated backward in time, as the forward integration was unstable. The PDE has to be created with the correct sign of  $dT/dt$ . Otherwise, the PDE would have been unstable. The static optimization problem required an intelligent guess with respect to the initial conditions of the (10) parameters which was easy in this example, but this is not always the case.

With respect to ACCURACY, again the shooting method seems the best, as it does not require any discretization. The problem can be solved directly as posed. Looking at the results above, it seems that the two other methods are in better agreement with each other, but that does not mean anything, as we have used the same discretization in both methods. Indeed, when we use the value  $T(1) = 25.95$  as the initial value for an ordinary simulation, we obtain  $T(0) = 108.$ , that is: 8% error on the second boundary value.

With respect to EFFICIENCY, the two discretized problems seem to be equal whereas the shooting method is somewhat more expensive. However, it is difficult to make a final judgement here for the following reasons:

- 1) FORSIM computes in double precision, whereas NLP uses single precision only. Of course, double precision is more expensive in computation, thus, we do not compare apples to apples. It really would make sense to convert NLP to double precision as well.
- 2) In the shooting technique, we have not taken advantage of the fact that the integration needs not to be very accurate as long as we are far away from the optimum, that is: as long as the gradient is sufficiently large. We should modify the combined NLP/DARE system such that the integration tolerances are made dependent on the optimization gradient. More research is needed to find out how to do this in an optimal manner.
- 3) We really can only compare the efficiency of CODES rather than the efficiency of ALGORITHMS. This is a standard shortcoming of all benchmarking tests. Thus, if we find that one particular "solution" was obtained

more efficiently, it may simply mean that the programmer of that tool was more clever than the others, and not necessarily, that the technique is superior. This is at least true as long as the differences in time do not vary by orders of magnitude which was not the case in our example.

CHAPTER 8  
CONCLUSIONS AND OPEN QUESTIONS

8.1 Conclusion

As mentioned in chapter 1, the goal of this research was to develop an integrated software system capable of solving engineering design problems. We also mentioned that in order to achieve this goal, the following steps were needed:

- 1) transfer NLP from the CYBER to the VAX/VMS machine
- 2) create a development system for NLP which interactively runs the program
- 3) use the DARE postprocessor to produce graphic output
- 4) integrate the NLP software with DARE/INTERACTIVE. Of these four steps, we have finished the first three. We also connected NLP with DARE/INTERACTIVE externally, that is: we can use the two software systems together, but the user interface for this combined software system is not yet optimal. The code is meanwhile executing satisfactorily.

We were able to prove the applicability of the chosen path at hand of a number of practical problems that we were able to solve. The software as implemented places a powerful tool into the hand of the engineer and

scientist by use of which many important, and currently unsolved problems can be brought to a solution. One such study, curve fitting of random data to distribution functions, has already been conducted, and led to a very successful senior project report [1]. Another application, optimization of electro-optical lenses has meanwhile begun. Due to time limitations, our work has to stop here. The following problems still need to be tackled.

### 8.2 Open Questions

1) We have already connected NLP with DARE/INTERACTIVE externally. However, as indicated by Cellier [6]: "we believe that a nonlinear programming package should be an intrinsic part of any simulation system." Hence, it would be an important step to completely integrate NLP into the DARE/INTERACTIVE system. As suggested by Cellier, the idea of this work is the following:

1) Create a subroutine for the DARE run-time system to retrieve and interpolate data stored on a STASH-file. "A STASH file is the DARE/INTERACTIVE data base which can hold any amount of data records stemming from various sources" [6]. In the current work, we used several times tabular data (e.g. a \$T-block of DARE) to describe measurement data. This is not realistic for practical purposes. There may be hundreds or thousands of data points required. It is nonsensical to expect

the user to retype them in form of a \$-T-block. Even a preprocessor that would automatically generate the \$T-block would not be the right answer, as the data would still have to be held as program code in memory. It is important that these data points can remain on the mass storage device, while interpolation is performed directly on those data.

2) Modify the interface between DARE/INTERACTIVE and NLP such that the measured data may come from the STASH file rather than from a memory-maintained table.

3) Integrate (2) into DARE/INTERACTIVE and introduce a new run-time command: "FIT #n" where n stands for a STASH record number. With this command being executed, the system will search the STASH record #n to find a match between names of variables on the STASH record, and names expressed as state or algebraic variables in the model. It then will fit those variables with the data stored in the respective columns of the STASH record. The parameters to be modified are those parameters in the parameter file, whose current value is "UNKNOWN". The performance function used will be:

$$PI. = \sum (x_i - \hat{x}_i)$$

but it will be system maintained rather than having

to be specified by the user in the \$D1-block. The execution of the program will be the same as was demonstrated in chapter 6, however, the user program will become much shorter (the currently needed \$F1- and \$F2-blocks will simply vanish.

4) Eventually, an additional keyword will be needed in the DARE language to denote optimization constraints. Constraints involving only one parameter such as "par1>0," or "10<par2<25" can already now be specified as tolerances in the parameter file. These are very common types of constraints. However, constraints may eventually be expressed as complicated function involving several parameters, and for this purpose, we currently do not provide for a concept. Eventually, we may introduce a new keyword similar to the TERMINATE keyword into the \$D1-block, such as:

```
CONSTRAINT par1*par2 < 100.
```

2) As long as we are far away from the optimal point, the precision of the program is really not a problem. However, when we approach the solution, we may require more digits than we have currently available. A double precision version would be very meaningful. For reasons of time constraints, NLP was not converted yet from single precision to double precision. The original CYBER version was coded in single precision

as the CYBER offers a larger word length anyhow (60 bits).

- 3) Fig. 8-1 shows the performance function varying over a parameter in a one dimensional search. The goal of optimization is to find  $p^*$  such that PI takes its minimum value within some given interval. Ideally, the exact value can be found by repetitive evaluation of the performance index, and iteration over the optimization algorithm. However, there exist always some numerical errors in the solution. One of them is the truncation error which is introduced during an integration step. As indicated by Gear [13], it is "the difference between the value given by the method and the value of the solution of the differential equation which passes through the value at the beginning of the step." Another error is the round-off error which is due to the finite number of digits in the calculation. All these errors can be regarded as "noise" of the system. The effect of the noise becomes very serious when there are integrations in the algorithm. For static optimization problems, the current noise level in NLP is about  $10.E-6$  (which is the machine resolution of the VAX/VMS in single precision, that is: the smallest number that can be distinguished from 1 in an addition:  $(1+\text{eps} > 1)$ ). For dynamic optimizations (e.g. identification problems),

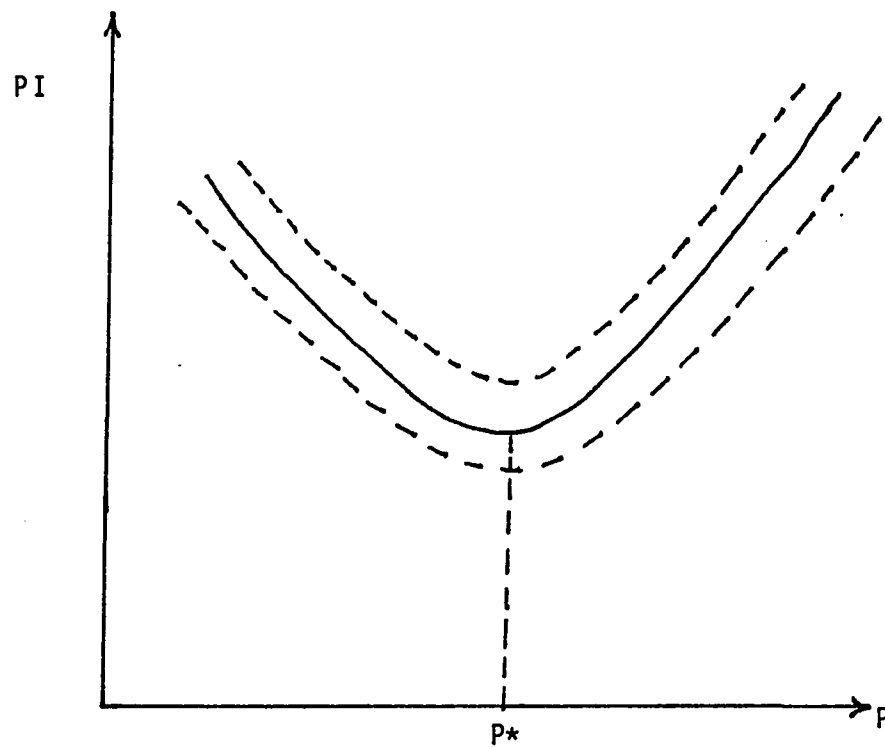


Fig. 8-1 Performance index varying versus parameter in uni-dimensional optimization.

the current noise level is about  $10.E-4$  as each performance function evaluation involves an entire simulation with its round-off and truncation errors. (The global integration error is roughly one order of magnitude larger than the local integration error introduced during one integration step for most stable systems.) As Fig. 8-1 illustrates, this noise has little influence on the optimization as long as the solution is far away from its optimum (the gradient of the performance function still being sufficiently large). However, in the neighborhood of the optimum solution, this noise shall hamper convergence drastically. On several occasions, we had to increase the termination criterion of the optimization algorithm, as the solution approached the optimal point to a certain degree, then started to stochastically "hop around" the true solution point for ever. One way to overcome this problem is to switch to double precision as mentioned earlier. Then, the static noise level will be roughly  $10.E-16$ , and the dynamic noise level will be roughly  $10.E-14$  on VAX/VMS. This is sufficient for most applications. However, we do not need all that (expensive) precision as long as we are far from the optimum. Hence, it is necessary to develop a new algorithm which will vary the accuracy requirements as a function of the

gradient of the performance function. That is, the error tolerance at the bottom of the performance function should be smaller than at the initial point. The proper relation may be :

$$\text{Tolerance} = \min (0.01 * \text{Gradient}, 1.E-2) .$$

- 4) Sometimes, we need to optimize parameters of a stochastic system. All the remarks made under (3) apply here again, but much aggravated. We need new algorithms that take over once the optimum is sufficiently close. None of the algorithms implemented in NLP at this date will do for this task. We may eventually have to turn to some type of Monte Carlo technique. It is important to study this situation, and come up with some answers.

The first two points are simple implementation tasks, and we would have implemented them ourselves if time constraints had allowed it. The latter two points (3 and 4) are much more serious, and require additional basic research. They were never intended to be tackled in this limited study.

APPENDIX A  
RUNNING NLP ON VAX/VMS

Here we want to show how to run the NLP package on the VAX/VMS system without the development system. Assume we have the NLP in the directory [SCRATCH.LUPING]. Following are the procedures to run the package.

```
**** create the NLP library ****  
  
$FOR NLP.FOR  
$LIB/CREATE NLP.OLB NLP.OBJ  
  
**** edit the test problem ****  
  
$EDIT TEST1.FOR  
omit  
  
**** compile the TEST1.FOR and link it  
**** with NLP/LIB  
  
$FOR TEST1.FOR  
$LINK TEST1,NLP/LIB  
  
**** assign the output to TEST1.OUT for hardcopy **  
  
$ASSIGN TEST1.OUT FOR006  
  
**** run TEST1.EXE ****  
  
$RUN TEST1  
  
**** after the problem has been executed,  
**** print out the result  
  
$PRINT TEST1.OUT  
$DEASSIGN FOR006  
  
**** end ****
```

APPENDIX B  
RUNNING NLP ON VAX/UNIX

The procedures of running NLP on VAX/UNIX are the following:

```
**** compile NLP and the test problem TEST2 ****  
$ f77 -c NLP.f  
$ f77 -c TEST2.f  
**** link TEST2.o with NLP.o ****  
$ f77 -o NLP.out TEST2.o,NLP.o  
**** run the executable file NLP.out ****  
$ NLP.out  
**** end ****
```

## REFERENCES

1. Alali, A. "A Random Number Generation and Distribution Fitting with an Interface to the CONTROL-C System.", Senior Project Report, University of Arizona, 1986.
2. Aoki, M. Introduction to Optimization Techniques., Macmillan, New York, 1971.
3. Burley, D.M. Studies in Optimization., Halsted Press, New York, 1974.
4. Carver, M.B. Stewart, D.G. Blair, J.M. and Selander, W.N. The FORSIM VI Simulation Package for the Automated Solution of Arbitrarily Defined Partial and/or Ordinary Differential Equation System., Chalk River Nuclear Laboratories, Chalk River, Ontario, 1978.
5. Cellier, F.E. "Simulation Software : Today and Tomorrow.", Simulation in Engineering Sciences., North-Holland, 1983, pp.3-19.
6. Cellier, F.E. "Enhanced Run-Time Experiments for Continuous System Simulation Languages.", Proc. of the 1986, SCS Multiconference on Language for Continuous System Simulation, San Diego, 1986, pp.78-83.
7. Cellier, F.E. and Fischlin, A. "Computer-assisted Modeling of Ill-defined Systems.", Progress in Cybernetics and Systems Research., Vol.VIII, 1982, pp.417-429.
8. Davies, D. and Swann, W.H. "Review of Constrained Optimization.", Optimization., Academic Press, London and New York, 1969, pp.187-202.
9. Fiacco, V.F. and McCormick, G.P. Nonlinear Programming: Sequential Unconstrained Minimization Techniques., Wiley, New York, 1968, pp.1-2.
10. Fischlin, A and Baltensweiler, W "Systems Analysis of the Larch Bud Moth System. Part 1: the Larch-Larch Bud Moth Relationship"., Mitteilungen der Schweizerischen Entomologischen Gesellschaft., No.52, 1979, pp.273-289.

11. Fletcher, R. "A Review of methods for Unconstrained Optimization.", Optimization., Academic Press, London and New York, 1969, pp.1-12.
12. Fletcher, R. "A New Approach to Variable Metric Algorithms.", Computer Journal., Vol.13, 1970, p.317.
13. Gear, C.W. Numerical Initial Value Problems in Ordinary Differential Equations., Prentice-Hall, Inc..
14. Gerald, C.F. and Wheatley, P.O. Applied Numerical Analysis., Addison Wesley, 1984, pp.359-365.
15. Golub, G.H. and Wilkinson, J.H. "Ill-conditioned Eigensystem and the Computation of the Jordan Canonical Form.", SIAM REVIEW., Vol.18, No.4, October 1976, pp.578-618.
16. Greig, D.M. Optimization., Longman, London and New York, 1980.
17. Hugnenin, F. "Zustandsregelung eines Elektromechanischen Systems mittels Mikrorechner", Elektroniker., No.10, 1979.
18. Kowalik, J. and Osborne, M.R. Methods for Unconstrained Optimization Problems., American Elsevier Publishing Company, Inc., New York, 1968.
19. Kuhn, H.W. "Nonlinear Programming : A Historical View.", Nonlinear Programming., Vol.IX, SIAM-AMS Proceedings, 1976, pp.1-26.
20. Leon, A. "A Comparison among Eight Known Optimizing Procedures.", Recent Advances in Optimization Techniques., Wiley, New York, 1965, pp.23-46.
21. McCormick, G.P. and Pearson, J.D. "Variable Metric Methods and Unconstrained Optimization.", Optimization., Academic Press, London and New York, 1969, pp.307-326.
22. Moler, C. and Van Loan, C. "Nineteen Dubious Ways to Compute the Exponential of a Matrix.", SIAM REVIEW., Vol.20, No.4, October 1978, pp.801-836.
23. Nour Eldin, H.A. "Optimale Stenerung Linearer Regelsysteme mit Quadratischer Zielfunktion.", Regelungstechnik., No.18, 1970, pp.164-169.

24. Rinvall, M. and Cellier, F.E. "MIDGET, Ein Flexibles, Simulationstechnisches Entwicklungssystem.", Proc. of ASIM '84, Viena, Austria, Springer Verlag, Informatik-Fachberichte, 1984.
25. Rosenbrock, H.H. Computer Journal., Vol.3, 1960, p175.
26. Rufer, D.F. User's Guide for NLP--A Subroutine Package to solve Nonlinear Optimization Problems., Dept. of Automatic Control, Swiss Federal Institute of Tech., ETH-Zentrum, CH-8092 Zurich, 1978.
27. Rufer, D.F. "Implementation and Properties of a Method for the Identification of Nonlinear Continuous Time Models.", A Link Between Science and Application of Automatic Control., Vol.3, IFAC, 1978, pp.1919-1926.
28. Wismer, D.A. and Chattergy, R. Introduction to Nonlinear Optimization., North-Holland, New York, 1978.
29. Yandell, D. SAPS II :User's Manual and Progress Report., 85-06, Computer Eng. Research Lab., 1985.