Copyright © 1983, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

DELIGHT: AN INTERACTIVE SYSTEM FOR OPTIMIZATION-BASED ENGINEERING DESIGN

by

William T. Nye

Memorandum No. UCB/ERL M83/33

31 May 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research sponsored by the Joint Services Electronics Program
Contract F49620-79-C-0178.

M83/33
385 Pages
Velo Bind

DELIGHT: AN INTERACTIVE SYSTEM FOR OPTIMIZATION-BASED ENGINEERING DESIGN

by

William T. Nye

Memorandum No. UCB/ERL M83/33

31 May 1983

DELIGHT: An Interactive System for Optimization-Based Engineering Design

Ph.D. *William T. Nye* EECS Dept.

Signature: Committee Chairman

ABSTRACT

DELIGHT is an interactive system for applying optimization techniques to engineering design. With DELIGHT, designers can use optimization algorithms to improve the performance of their designs by automatically adjusting design parameters. DELIGHT offers the capability of optimizing arbitrary performance criteria as well as of studying complex tradeoffs between multiple competing objectives, while simultaneously satisfying multiple constraint specifications.

The use of optimization in engineering design was proposed in the sixties, especially for discrete electrical circuits such as filters. However, its use has not become as widespread as one would expect. We examine the shortcomings of several applications of classical optimization techniques to engineering design and develop design criteria for a computer-aided design system that should overcome these difficulties. These criteria, and the intention of serving a wide range of users from designers to optimization experts and system-support personnel, are used in the design of DELIGHT.

The various aspects of the DELIGHT system—the RATTLE language, RATTLE extensibility through defines and macros, the library of optimization algorithms, the problem description facilities, a set of high-level matrix macros, and

terminal-independent color graphics — are introduced and described in detail. A novel feature of the DELIGHT system is a new multiple objective problem formulation that provides a means of effectively classifying and conveying the relative importance of design specifications. A methodology for performing design tradeoffs when using this formulation is introduced that uses a new graphical display called the performance comb.

The DELIGHT system is intended to be used in many different areas of engineering design. Hence, we introduce a simulation interface methodology and other necessary features that facilitate the coupling of DELIGHT to existing simulation programs. We then take a look at several application areas and demonstrate the usefulness of the system by detailing the optimization of two electronic integrated circuits. Successful optimization of several industrial circuits and of systems in other engineering areas are reported that further show the effectiveness of the system.

Acknowledgements

I would like to express my deepest gratitude to Professor A. Sangiovanni-Vincentelli, my research advisor, and Professor L. Polak for introducing and giving me the opportunity to work in optimization and computer-aided design. They have supported me with enthusiasm and were available for numerous discussions throughout the course of my graduate study at Berkeley. Also, it is a pleasure to thank them, along with Professor K. Pister, for serving on my dissertation committee.

The special contributions by A. Sangiovanni-Vincentelli to cultivating my ability to make clear presentations, both verbally and in writing, will have lifelong value. I will always remember those occasions when, after expressing my excitement over something I had just written, he would peer over the top of his glasses and say, "let—me—see—it—first!"

The rewarding discussions I have had with fellow graduate students and others have contributed many useful ideas to this research. R. Balling, G. De Micheli, N. English, T. Essebo, Andrew Heunis, M. Karandikar, K. Keller, J. Kleckner, P. Labuhn, E. Lelarasmee, Sharad Nandgaonkar, P. Nicklin, R. Oliver, T. Quarles, Dave Riley, P. Siegel, James Spoto, Andre Tits, V. Visvanathan, Y. Wardi, and T. Wuu, are just a few of the names that come to mind. Andre Tits, in particular, has had a major influence on this work. Special thanks to the brave students who took EECS 241 in the fall of 1982 and provided their feedback on the results of this work.

My occasional acquaintances with W. Joy in Computer Science have inspired me to try to provide just a fraction of the benefit that he has given to mankind through his work with UNIX.

I take great pleasure in showing my appreciation of the friendship extended by G. Brand, C. Courcoubetis, J. Dorfman, E. Eschen, L. Guy, T. Hull, J. Jones, M. Loo, L. Gast, A. Neirynck, I. Ratiu, T. Salcudean, S. Sastry, R. Silva, D. Stimler, E. Szeto, A. Vladimirescu, professors P. R. Gray, D. A. Hodges, A. R. Newton, and D. O. Pederson, and many others, and also by the entire staff of the Electronics Research Laboratory and the Department of Electrical Engineering and Computer Science at Berkeley. Many other friends at Harris Semiconductor too numerous to mention are also appreciated. Special thanks to T. King for assisting with the figures.

My association with Professors A. Brodersen and S. Director during my early years in Electrical Engineering at the University of Florida created important foundations necessary for my doctoral work at Berkeley.

I would like to acknowledge research grants from Harris Corporation and thank, in particular, J. Cornell and J. Spoto of Harris Semiconductor for providing an ideal environment in which to demonstrate the industrial applicability of my work. The support of the Air Force Office of Scientific Research, the Joint Services Electronic Program, the National Science Foundation, and MICRO is also recognized.

Finally, I wish to express my gratitude to my parents, Sylvia and Thomas.

Table of Contents

CHAPTER 1

Introduction

1.1. The Nature of Engineering Design

For our purposes, engineering design can be considered as a three-phase iterative process. The designer must:

1. derive objectives and specifications from the requirements of the system being designed,
2. select a structure or configuration which has the best chance of meeting the specifications, and
3. determine values of system parameters that optimize the possibly competing design objectives while satisfying the constraint specifications.

For a particular design problem, the designer may go back and forth between the three phases many times until a satisfactory design has been achieved.

It is natural to try to use the computer to aid designers in performing the steps above. While there exist design procedures for limited classes of problems in which system structure and parameter values are synthesized simultaneously [149, 153], generally the selection of system configuration is a very creative and experience-based part of the design process and may not be very amenable to computer assistance. Similarly the derivation and evaluation of objectives and

specifications, in quantifying such questions as:

1. What quantities must be "specified"?
2. What criteria are used to define a "good design"?
3. What defines "acceptable performance"?

is also a very creative and subjective aspect of design which certainly requires the experience and intuition of the designer. For large complex systems, however, the selection of the values of a large number of system parameters is often time-consuming and is usually stopped short of the design's best performance. This is usually due to the difficulty of designers in predicting the effect of parameter changes on system performance without simplifying engineering approximations or numerous computer simulations.

That this parameter value determination phase can greatly benefit from computer assistance is the result of a declining cost of computing power and of significant advances, in the past 15 years, in two relatively separate research areas. The first of these advances is in the area of optimization algorithms for nonlinear programming. Algorithms have emerged which have many desirable properties that are particularly suited to engineering design. Many have guaranteed convergence properties which depend on rather mild conditions which are satisfied by many engineering design problems. There are several algorithms which optimize an objective subject to constraints and even some which solve problems involving *functional* constraints which must be satisfied over an interval of an independent parameter such as time, temperature, or frequency. Recently algorithms have been developed which allow multiple objectives to be optimized while simultaneously satisfying constraints.

The second area of advance is in system simulation or analysis programs.

Advances in accuracy, efficiency, and reliability of these programs lead, on the one hand, to the ability of engineers to simulate larger systems (or portions thereof). On the other hand, these advances also make it economically feasible to repeatedly simulate smaller portions of a system for the purpose of varying parameters to improve its performance. Computer simulation has become a popular and in many instances indispensable tool in engineering design.

This simulation power has lead many designers to believe that general analysis programs are all that is required, for with them designers may continue to use the popular cut-and-try methods as in the precomputer era—only faster, cheaper, and with greater accuracy. In the design of electronic circuits, the simulation program replaces the breadboard and the cut-and-try method is similar to "pot-tweaking". While such design methods have worked in the past, they are less likely to succeed in the future as system complexity and the number of parameters increases or as more performance is expected.

In this dissertation, the DELIGHT system, designed and implemented to combine the two areas of optimization and simulation, is presented to provide engineering designers with a powerful new computer-aided design (CAD) tool in which engineer and computer are complementary as they work together to optimize the performance of designs. In particular, DELIGHT uses *parametric optimization* in the numeric parameter determination phase of design in which the system configuration remains fixed. Used in this way, optimization algorithms may be viewed as a way of effectively managing the cut-and-try process by taking advantage of the computer's ability to assimilate much information about the current design from automatically performed simulations in order to determine the best set of parameter modifications. Optimization thus frees the designer from the difficult and tedious task of making these modifications and

allows him to concentrate on the more creative aspects of the design process.

There are several additional benefits of applying parametric optimization to engineering design. By freeing the designer from the highly repetitive aspects of the parameter selection process, he may apply his creativity, intuition, and experience in other ways:

1. He may devote more time to the derivation of objectives and constraint specifications and the definition of acceptable performance measures. This enhances the first phase of design by allowing a more rapid determination of whether they are suitable for meeting the requirements of the design.
2. He may concentrate on the structural aspects of the design by considering several alternate configurations without being bound by the usually prohibitive time required to compare the best performance of each.
3. He may provide a good initial guess for the design parameters. This may be obtained by other computer techniques as well as by manual design procedures.
4. Decisions about which constraints are most important to satisfy must be made. An indication of the admissible severity in violating each constraint must be given.
5. The designer may devote his time to evaluating and comparing the results of trading off competing objectives in a particular configuration. This exploration of tradeoffs is usually essential to any modern design methodology.
6. By monitoring the optimization process, the engineer can go through a learning experience about his system's performance that otherwise might be missing if manual design techniques were stopped short of the system's

best capabilities.

Another benefit is that in formulating the objectives and specifications of his design, the designer may address design problems more closely to their origin; the limitations of classical design approximation techniques may be avoided. Many design "tricks" which prior to the existence of an optimization-based approach were necessary in order to perform the mathematical manipulations of the design process, may no longer be considered a necessary part of the designer's repertoire. Instead, the designer's grab-bag contains, for example, his experience with the optimization process in knowing what type of problem formulations lead to rapid convergence of the optimization algorithms used.

1.2. Examples of Design Problems Addressed

Examples of design problems which fall into the class of problems handled by the DELIGHT system can be found in many areas of engineering. In the area of analog electronic integrated circuit design, a common problem in amplifier design is to maximize bandwidth subject to constraints on stability or on total power consumption. The design parameters whose values must be determined might be resistors, capacitors, or transistor device geometries. The structure of the network would be determined a priori by the designer and would remain fixed throughout the optimization process.

Another electronic circuit design problem involves digital cells of VLSI (very large scale integration) systems. One common scenario is to minimize circuit switching times subject to fan-in/fan-out requirements and again to constraints on power consumption. For these types of problems, the design parameters are usually device geometries only.

In the area of single input/single output (SISO) and multiple input/multiple

output (MIMO) control systems there are many different types of design problems which can benefit from optimization techniques. For example, one may wish to minimize the energy fed into the plant of a feedback system, subject to constraints. These constraints might be bounds on compensator parameters, overshoot and settling time constraints on the time-domain step response of the closed loop system, stability requirements that the poles of the closed loop system lie in certain regions in the complex plane, and requirements that the system be insensitive to output disturbances. Some of these requirements result in constraints which require very special optimization algorithms.

A final design problem example is found in the area of structural design in which braced frame buildings must withstand small earthquakes with no damage and large ones without collapse [18]. A common objective is to reduce construction cost by reducing the weight of the structure. This is done by minimizing the cross sectional area of the frame members subject to several groups of constraints. The designer may formulate one group of constraints corresponding to a static model subjected to gravity loads and another group corresponding to a dynamic model whose goal is to limit the relative floor displacements over the entire duration of a whole family of moderate and severe earthquake ground motions. The system structure remains fixed throughout the optimization process in that the permissible design variables may only affect element properties; thus, for example, the position of the frames and the distance between nodes must remain constant with changes in the design parameters. An example of element properties which may be design parameters are section area, strain hardening ratio, or section moment of inertia, for the beam-column element.

1.3. Dissertation Outline

The plan of this dissertation is as follows. We begin with an overview of optimization-based computer-aided-design in chapter 2. This leads from early efforts to recent design systems and ends with the limitations of both early and recent efforts. By examining the needs of various classes of users to be supported by the DELIGHT system; chapter 3 results in a set of design criteria for the system. Chapter 4 then surveys DELIGHT features which meet these design criteria. This includes DELIGHT's RATTLE programming language, the way design problems are specified to DELIGHT, a discussion of optimization algorithms and libraries, and other features which enhance the design process while using DELIGHT. In chapter 5 we present several applications of DELIGHT to engineering design. The foremost of these is DELIGHT.SPICE for electronic circuit design. We conclude in chapter 6 with a summary of the main contributions of this work followed by directions for future research. Appendices are included which explain several DELIGHT implementation issues and show the details of several DELIGHT features for developing application packages.

CHAPTER 2

Overview of Optimization-Based Computer-Aided-Design

2.1. Early Efforts

Since it is difficult to survey the early use of optimization in all areas of engineering design, we restrict ourselves for the most part to a few representative areas. One of the earliest uses of optimization in engineering design was to solve problems that had been formulated as matching or curve fitting problems [11, 26] in which the goal was to match a calculated and a desired system response. This was probably due to the fact that, by following Aaron [11] and defining "best" as the minimum of the sum of the squares of the errors between desired and calculated responses, matching problems could easily be formulated as linear or nonlinear least squares problems. These could then be solved using either specialized techniques or available general purpose algorithms and routines for simple unconstrained optimization, that were developed much earlier than ones for constrained or multiple objective optimization. Another error criterion was "equal ripple" approximation in which the absolute magnitude of the error was minimized.

The least squares matching problem was usually considered as follows. Defining $\mathbf{x}$ as the vector of design parameters, $f(\mathbf{x}, z_i)$ as the calculated response at the i'th value of an independent parameter z as i ranges from 1 to n (where z may be i itself), and $F(z_i)$ as the desired response (or measured in the case of model parameter determination), then the goal of having $f(\mathbf{x}, z_i)$

equal $F(z_i)$ for every i was formulated as the following least squares problem:

$$\underset{\mathbf{x}}{\text{minimize}} \sum_{i=1}^{n} (f(\mathbf{x}, z_i) - F(z_i))^2$$

In the special case that the number of parameters in $\mathbf{x}$ exactly equaled n, the number of z_i data points, ordinary Newton iteration was sometimes used [26]. Otherwise, either generalized Gauss-Newton methods (previously called Taylor Series Methods) [11], Marquardt enhancements to these methods [96], or other general unconstrained minimization techniques such as steepest descent [35] were used.

Early use of the above solution techniques is now illustrated. A typical computer program which combined the techniques above was SUPROX [48], an acronym for SUccessive apPROXimation program, developed at Bell Laboratories in the middle sixties. As explained by Golembeski [53], the program determined the unknown parameters via two "slope-following" techniques, the method of steepest descent and a generalized Newton-Raphson technique, which were used consecutively, although either technique could be used alone. The steepest descent method was used initially since it converged rapidly when the parameters were far from optimum. When the overall minimum was neared, i.e., there was less than 5 percent reduction in the error function per iteration, the generalized Newton-Raphson method was substituted because it had quadratic convergence when started with good initial parameter values. This program was used in a number of different applications of interest to designers.

Examples of engineering problems which were formulated as matching problems include model parameter determination for modeling system performance for computer simulation, black box techniques [102], and the design of electronic filters and microwave integrated circuits. The goal of the first of these,

modeling, is to represent physical structures or phenomena by idealized, but mathematically tractable models. Modeling becomes a matching problem once a fixed structure model is chosen and it is desired to minimize the error difference between responses calculated by the model and measured responses. The independent parameter in the least squares formulation (**x** above) are various external stimuli which bring into play the ways in which the model will be used.

Black box techniques, which have now evolved into present-day macro modeling, were methods which were required to cope with the system size limitations of early simulation programs [102]. In this approach, calculated or measured discrete data from a complex part of a system were replaced by standard functions whose parameters were determined by a curve fitting process. The data hopefully captured how external stimulus at input "ports" produced responses at output ports. The particular functions used were either determined by the application, for example hyperbolic functions for uniform transmission lines or orthogonal polynomials for certain mechanical systems, or chosen by the designer from several standard functions supplied by the curve fitting program. For the latter, the choice was a compromise between availability of function subroutines to do the fitting, system size, and efficiency and accuracy of the resulting black box.

For the design of electronic filters, the filter response functions to be matched against desired curves were usually the magnitude and phase of transfer functions (including input and output impedances) discretized over the independent parameter frequency. One interactive optimization program for designing electrical filters was TRANSFIT [103]. It was written in Fortran for the GE-605 time-shared computer system and was an adaptation of a program given

by Calahan [27] for fitting ratios of polynomials to frequency-domain characteristics. The fit was accomplished through the use of a Fletcher-Powell optimization method [49] with Fibonacci Search. The user could interact in the optimization process by changing the initial guess for the polynomial coefficients and the weights at different frequencies.

A more recent filter (and microwave) design system is COMPACT [6], a question/answer style interactive optimization program. It allows designers to minimize a weighted scalar error function consisting of the sum of the squared deviations between several frequency-domain properties of circuits. These properties include two-port scattering parameters, noise figure, and input port phase shift. Control of this fixed problem formulation is accomplished by adjusting the weights or simply setting them to zero to remove their respective terms from the error function. If the designer cannot provide an initial guess, COMPACT does a coarse search to find an approximate global minimum followed by the steepest descent method to find the actual minimum. This could be very time-consuming for a large number of design parameters. However, typically the circuits optimized are relatively small. Since COMPACT contains no DC or time-domain analysis and handles only bipolar transistors, its use in general circuit optimization is limited.

Another design program which emphasizes frequency-domain matching for electronic filters is OPNODE [3]. It uses the following adaptive search technique. The search begins in a purely random manner. Each error function evaluation updates stored probabilities for that direction. Thus, the search algorithm "learns" by trial and error how to vary parameters to minimize the error function. An interesting feature of OPNODE is how a designer interacts with the program. OPNODE runs on an HP minicomputer and allows a user to flip panel

switches to turn on or off performance plotting, parameter value display, and other algorithm controls during each trial error function evaluation. Also, since the program is written in Basic, it enjoys the interactive benefits of any interpreted language. One of these is that execution may be interrupted at any time, any program statements changed, and execution resumed from the same point.

The sheer number of references on the subject of computer-aided filter design suggests that in the electrical engineering area, it was the heaviest user of early optimization techniques.

An extension of the use of optimization in electronic circuit design to consider both DC biasing effects as well as frequency-domain matching was first accomplished by Dowell [44] at Berkeley. Shortly thereafter, his colleague McCalla [99] combined the automated DC biasing techniques of Dowell with the previous work of Walsh [151] and Wooley [161] in optimizing small-signal frequency responses without bias point variation. Thus, McCalla's optimization considered the direct AC dependence of design parameters as well as the indirect DC dependence through transistor biasing. The optimization performance function was again the sum of the squared errors between actual and desired responses summed over frequency. The Fletcher-Powell method [49] with cubic interpolation line search was added to the circuit simulator SLIC [68] and passive element values only were allowed as design parameters. An experiment tried by McCalla was to minimize the temperature sensitivity of his circuits by summing deviations at two different temperatures. After poor results, he concluded that optimization could not be effectively extended to optimizing temperature sensitivity due to the expense of computing the necessary second order derivatives. Another startling conclusion he made was that all of his optimization results could have been reached without the aid of the automated

approach, just more slowly.

The next step in the evolution of optimization in engineering design was the formulation of certain design problems as general nonlinear programming problems. This approach required the formulation of a single performance function (cost) to minimize subject to a set of inequality constraints, as in the following standard form:

$$\underset{\mathbf{x}}{\text{minimize}} \ \{ f(\mathbf{x}) \mid \mathbf{g}(\mathbf{x}) \leq 0 \}$$

There was little use, however, of what few constrained optimization algorithms existed; in many cases [136, 86] the constrained problem was transformed into an unconstrained minimization problem using, for example, the penalty function approach of Fiacco and McCormick [47]. Indeed, many researchers who described algorithmic procedures did not even mention how this important transformation might be carried out. An early review paper by Director [41] simply states that the first step of an automated network design procedure requires "a [scalar] performance function which embodies the design criterion." For the time-domain design problem, the integral performance function

$$\varepsilon = \int_0^T e(\mathbf{w}, \mathbf{q}, \mathbf{p}, t) \, dt$$

and the Fletcher-Powell [49] or Fletcher-Reeves [50] optimization methods were suggested.

In parallel with the increased use of nonlinear programming in design came great improvements in the efficiency of calculating the gradients necessary for the most useful optimization algorithms. For electronic circuits, Rohrer and Hachtel [132, 57] first showed how the computation of the gradient of a performance function at a single frequence point with respect to every design

parameter could be accomplished in just two network analyses; previously it had been thought that this computation required an additional analysis for each design parameter. Shortly thereafter, Director and Rohrer showed a similar result by introducing the *adjoint network approach* [39, 40]. It used a derivation that, although somewhat more general by not requiring a state variable formulation, was easily understood and the adjoint approach was used by many others as a means of calculating gradients for a wide variety of applications as well as for optimization.

These highly efficient methods for gradient computation have not been used as much as might be expected in other areas of engineering. Recently, however, there has been an interest in adding these methods to the ANSR [101] general purpose structural simulator [38].

2.2. Recent Design Systems

The maturation of simulation and optimization techniques have caused several computer-aided design systems to appear over the past few years. The first one to be discussed, ISPICE, provides only analysis functions whereas the remainder, A2OPT, INTEROPTDYN, and APLSTAP, are explicitly designed for engineering optimization applications.

ISPICE (for Interactive SPICE [5, 7]) is mentioned here because it was the first interactive design system used extensively by this author and its elegant interactive man-machine interface has had a significant influence on the design of DELIGHT. ISPICE is a user-friendly design tool which was created by combining the SPICE circuit analysis program [105] with the interactive and labor-saving aspects of AEDCAP [1]. One particularly interesting feature of ISPICE relevant to the design of DELIGHT is that it has no internal size limits. Rather

than allocating fixed size arrays or storage regions, ISPICE adapts to the circuit being simulated and obtains from the operating system whatever resources are required; simulations have been performed on circuits containing thousands of elements. An important novelty of ISPICE is the way circuit elements are assigned values in the circuit description file. A general expression capability allows almost any *VALUE* in the circuit file to be a constant, an arithmetic expression, a function reference, a variable parameter, or any combination of these. Another feature is the truly interactive way ISPICE communicates its results to the designer. Instead of following batch SPICE and dumping out reams of data on *all* the circuit elements, the program responds to user queries for various subsets of the available output data. The human-engineering of ISPICE interactive commands is seen in the following samples:

```
SWEEP VIN FROM -10V TO 10V BY 1V AT 10MEGHZ
PLOT VM(1) VS FREQ FROM 1MEGHZ to 100MEGHZ DEC 15
DISPLAY ELEMENTS R* Q101 QPAIR1 QPAIR2
```

The SLICE interactive program [9] in use at Harris Semiconductor was originally conceived by this author and was initially patterned after ISPICE. The capabilities have since then been considerably expanded and SLICE is currently used at Harris by many designers.

A2OPT [58] was an optimization-based CAD system for electronic circuits which added constrained optimization features to the ASTAP [2] network analysis program. It was probably the first system to allow designers to specify easily a wide range of objectives and constraints as arbitrary expressions of circuit outputs. While A2OPT was for the most part a batch program (ASTAP was strictly batch), an important feature was the limited interactive capability in

which the designer could interrupt and influence the course of the optimization. He could alter either (1) the optimization weights, (2) ASTAP run controls, or (3) parameters which control the optimization routine itself. The program combined the weighted sum of several objective or constraint functions, the latter using penalty functions, into a single scalar function which was minimized using the variable metric rank-one update method due to Cullum [32]. The program also included box constraints, i.e., upper and lower bounds, on the design parameters. The choice of weights was somewhat haphazardous, being adjusted experimentally by the designer after observing initial optimization performance. Because A2OPT constraints were actually the integral of the designer-specified expression, instantaneous violations might occur but be averaged out so that the constraint was actually satisfied; ways of circumventing this problem were suggested in [58].

In the digital circuit design experiments performed by the authors of [58] several points were made. One was the difficulty in initially adjusting the weights; often several optimization reruns were required. This could be very costly if the circuit size was large. Another point was their discovery that the design parameters and the objective functions were somewhat decoupled. This, they said, is inherent in the design of a mixed nonsaturating and saturating digital circuit in which the design specifications are a function of a single node. This point shows that designers must pay close attention to the specifications they formulate. Although A2OPT initially showed great promise, the lack of truly interactive features as well as the authors' limited drive to recruit users caused it to never be used extensively by practicing circuit designers.

Recently a research team at Berkeley put together the INTEROPTDYN design package [21] to enhance the use of optimization algorithms for engineering

design and to study the methodology needed for man-machine interaction and graphical display. It combined a particularly powerful optimization code OPTDYN [20], INTRAC-C, an extension of the INTRAC [155] language-interpreter construction module developed at the Lund Institute of Technology, and various application-dependent codes such as the CDP classical design package [33] from Imperial College in London. The INTEROPTDYN package allowed the user to write his own color or black-and-white graphical display programs as macro files. By inserting certain "call for interaction" INTRAC calls into the existing OPTDYN subroutine, the creators of INTEROPTDYN allowed the user to control the flow of computation by executing the optimization algorithm one step or one cycle of steps at a time. Computation could be interrupted and a matrix "scratch pad" used for diagnostic calculations in the middle of a run.

In the INTEROPTDYN-SISO version [124] for the design of single-input single-output linear feedback systems, the objective and constraints of the design problem formulation were fixed, with the designer only providing numeric parameter values for his performance goals. These parameters controlled: (1) an envelope on the closed loop step response, (2) frequency-domain criteria, (3) upper and lower bounds on the plant input and its derivative from a step input, and (4) design parameter box constraints. The built-in objective function to minimize was the integral squared error of the closed loop step response. The design parameters were limited to coefficients of the numerator and denominator polynomials of the compensator transfer function. INTEROPTDYN-SISO and other versions of INTEROPTDYN were used successfully for several different design applications but the difficulty in coding arbitrary objectives or constraints limited their use to academic circles at Berkeley. Of importance, however, is the fact that INTEROPTDYN helped to clarify ideas about what type of

interactive language and man/machine communication through graphics is necessary in such a design system.

The APLSTAP system [60] developed at IBM is an interactive design system which can operate in conjunction with an arbitrary existing simulation package. Presently APLSTAP algorithms written in APL exercise control over the ASTAP [2] circuit analysis program. They attempt to mathematically mimic the common practice by designers of trading off several performance objectives by linear extrapolation of their expected performance. Through interactive iterations with a novel computationally inexpensive linear programming (LP) step, key tradeoffs between multiple objective and constraint functions are revealed to the user. He uses this assistance and his knowledge and experience about the various functions to select a maximally effective LP step. Thus, the designer is constantly making tradeoff decisions in a design process in which emphasis is placed on getting the most from the first few optimization steps rather than on a completely convergent sequence of steps; in many design situations particularly involving large problems, convergence is computationally too expensive or not justified on the basis of model accuracy.

The APLSTAP system may be used for both optimizing multiple objectives and for improving the worst case performance of multiple objectives over the variations of a specified set of *worst case* statistically uncertain parameters. In each of these design problems the LP step may be used in either of two modes. *MIN-MAX mode* attempts to find the smallest values of all the objective functions over a user specified set of box constraints on the design parameters. These box constraints are an estimate by the designer of the range of linearity of the objective functions over each parameter. *MINBOX mode* attempts to find the smallest change in the design parameters which will achieve a user specified desired

improvement in each of the objective functions. The MINBOX LP step either produces the smallest change which achieves those improvements or states that they are not possible.

Since APLSTAP is a relatively recent system that has not yet been extensively used by practicing designers, the success of its approach is not yet known.

2.3. Limitations of These Systems

In some cases such as electronic filter design, early efforts were successful. But for more complex present-day design situations, both early as well as recent optimization-based design systems have many weaknesses which limit their use by practicing engineers. This section categorizes these weaknesses as those stemming from:

1. difficulties encountered by designers,
2. shortcomings of existing algorithms,
3. shortcomings of programs implementing algorithms,
4. difficulties encountered by optimization experts, and
5. shortcomings of simulation programs.

Recognition of these weaknesses leads to a set of goals and design criteria for DELIGHT in chapter 3.

Designers. The simplest reason why optimization was not used much for design is that many designers were (and still are) reluctant to use even CAD *analysis* tools. Those designers who did recognize the benefits a computer could provide were usually not trained in the areas of optimization or computer programming and thus had to use others' computer programs. Many times it just wasn't possible to access such programs, especially ones which contained the

algorithms they needed. When access *was* possible, it was often difficult for designers to formulate their design problems as well-posed optimization problems and to understand how to successfully use the algorithms which had been implemented. Brayton and Spence [23] emphasize these difficulties in listing the following objections design engineers might have to viewing their design problems as nonlinear programming problems:

1. "Design problems are not this simple."
2. "There is more than one design objective to be [improved]."
3. "It is not possible to state the design objectives in terms of mathematical functions."

Another point worth mentioning is that very often a "guru" of an institution would be the first to try to use a particular optimization program and if it did not quickly and easily yield success, he might make an initial conclusion which other engineers would accept as doctrine. This point is brought out by Paul Weil [152] about computer-aided design in general in an amusing article about his laments as a computer-aided design researcher.

Existing Algorithms. During the time period of most of the efforts described in sections 2.1 and 2.2, the best known optimization algorithms were usually inadequate to solve the complex design problems which faced many designers. Often multiple performance objective functions were combined into a scalar objective function to minimize using the *weighted sum* method. This was precisely the approach used by COMPACT and many other programs [6]. As pointed out by Lightner and Director [94], this may yield a poor design no matter what weights are chosen or what optimization technique is used for the minimization. As mentioned in sections 2.1 and 2.2, inequality constraints were often handled by penalty functions, an approach prone to have all kinds of problems. Penalty

functions are also far too primitive to solve design problems involving functional constraint specifications which must be satisfied for all values of an independent parameter over an interval.

Other difficulties with existing algorithms were that they often converged very slowly and sometimes not at all. It was not rare to find popular algorithms without any *guaranteed* convergence behavior whatsoever. Also, in cases of slow convergence, few algorithms were originally designed so that they had parameters which could be used to tune their performance to the particular class of problems being solved. In A2OPT [58] and others that did have such parameters, it was probably next to impossible for practicing engineers to understand how to adjust them.

Programs Implementing Algorithms. The main shortcoming of many early optimization programs was that they offered a fixed problem formulation which was inadequate to handle widely varying design requirements. Examples are the TRANSFIT [103] and COMPACT programs [6] mentioned in section 2.1. Designers could only supply desired curves to be matched by their filter or circuit transfer functions; they could not introduce additional constraints on, say, the pole locations of their desired filters. Another example is the fixed problem formulation of the INTEROPTDYN-SISO package [124] discussed in section 2.2. Although the formulation may have reflected the goals of many feedback system designs, the rigidity of the package not only made it difficult to add new constraints, but also, in early versions, forbid even a simple swap of a particular constraint and the cost. Thus, a designer could not minimize the rise time of his system subject to constraints on the integral square error of the closed loop step response.

Another deficiency of optimization programs was that they did not provide

any feedback to the user (especially powerful graphical feedback) on how well the algorithm was performing. Slow convergence can be caused by algorithm parameter values which cause sub-steps of the algorithm to run very poorly on a particular design problem. There are usually other values of these parameters which can greatly enhance algorithm efficiency. However, a designer usually had no way of knowing which ones to modify. Also, the algorithm might have contained adjustable parameters but the program implementation did not make them accessible to the user. Both of these shortcomings usually resulted in many optimization iterations or, in the case of batch programs, many reruns, costly both in computer resources and designer time.

Optimization Experts. Lack of emphasis by developers of optimization algorithms on certain aspects of real-world design problems may have caused their algorithms or programs implementing them to have less practical value to designers. For example, any program which did not put heavy emphasis on parameter and constraint scaling would probably perform poorly on real-world problems in which constraints or parameters sometimes take on values which may be orders of magnitude apart. The algorithms or programs reported in [21, 39, 136, 20, 58, 103] do not appear to have given this emphasis. Other things which algorithm developers may have lacked were (are) programming expertise and the time or interest to write a complete, user-oriented optimization package containing, for example, clear error messages and sufficient documentation. Even with programming expertise, the time required to code, debug, and test an algorithm was usually very long. Moreover, most optimization programs were usually written in Fortran, with its well-known deficiencies.

A review of the literature reveals the curious and popular notion by early optimization experts that optimization techniques were supposed to completely

automate the design process. In other words, they did not perceive the needs or the advantages of interactive computing, in which user and computer are complementary as they work together to carry out an optimization. This notion can be seen in the following partial quotations:

> (... will result in a) completely automated third (parameter adjustment) phase [28, page 242].
>
> ... methodology of "automated" or "hands-off" design, where "algorithm" replaces "insight" [27, page 139].
>
> Work is now in progress with the hope of constructing a set of *optimize* functions and weights so that the program may optimize from 300ns to 107ns in a single optimization run without user intervention [58, page 503].

Is it any wonder that early programs did not provide user feedback or a means of tuning algorithm performance ... their developers did not even expect the designer to be part of the optimization process!

One reason regarding "optimization experts" of why optimization has not "caught on" sooner is that after a few of them tried it in the early seventies with primitive algorithms that did not perform well, they consequently reached unfavorable conclusions[1].

Simulation Programs. The shortcomings of both past and present simulation programs are not a limitation of optimization-based design systems but are a reason why optimization has not been used more widely in design. The first shortcoming is that coupling to simulation programs is usually very difficult to achieve. Most simulators have not been designed to behave as function evaluation routines for an optimization process. Even when they have rerun capabilities for modified input parameter values, it is either through interaction as in

[1] See, for example, the conclusions of McCalla's dissertation [99].

ISPICE [5, 7] or through additional input "cards" as in ASTAP [2] or a recent version of SPICE [105], and not from a capability of receiving new input parameter values from a subroutine call by an algorithm. The already encountered difficulties in coupling DELIGHT to the SPICE [105] and ANSR [101] simulators bear out this point. Recently, however, work has begun on SPICE3 [129] with precisely the goal of having all access to a central "kernel" of analysis routines via a well-defined set of subroutines, including ones for parameter value update.

A further deficiency of most simulators is that they do not compute gradients needed by widely used optimization algorithms. As a result, gradient calculations are performed using finite differences, leading to a large increase in the time needed to perform an optimization.

Finally, the large number of different design environments in engineering means that there is a correspondingly large number of different simulation programs. Since the coupling of a successful algorithm subroutine to any particular simulator is usually very complex and interwoven, it can be difficult to extract the successful algorithm code for the purpose of using it with another simulator. Thus the spread of optimization to different areas is very slow.

In the next chapter, we shall consider the limitations discussed in this section in formulating a set of goals and design criteria for the DELIGHT system.

CHAPTER 3

Goals and Design Criteria of DELIGHT

3.1. Setting the Stage

In this chapter we discuss the goals of the DELIGHT system and the resulting set of design criteria intended to meet these goals. In creating these design criteria, we try to avoid the various limitations and pitfalls pointed out in the previous chapter.

The goals of DELIGHT fall generally into four broad categories. The first and foremost desire is that the system be *easy to use*. This goal is essential to the success of any large system intended to be used by many types of users. Computer operating systems such as UNIX[1] [131] and Interlisp [147] have long recognized the importance of a friendly user interface. Recently, many researchers have also been stressing this importance [89]. Singer et al. [140], in describing the design of a recent interactive environment for PASCAL programming, emphasizes this point. They recognized that human engineering must have top priority in the design of a system from the outset; it cannot be grafted on later. Our second goal is that DELIGHT be *versatile*. It must be useful in many different areas of engineering design, particularly in industrial environments. Third, the DELIGHT system must be relatively *efficient*, that is, make good use of limited computer resources. Our last goal is that the entire system be *portable* from one computer environment to another. The importance of this goal is

[1] UNIX is a Trademark of Bell Laboratories.

recognized more and more today and was one of the major underlying motivations for the design of the new programming language ADA [88, 8] by the Department of Defense.

One of the original DELIGHT objectives was to create a system that would facilitate its use by several different types of users. It is convenient to discuss the goals of DELIGHT by considering the needs of three classes of users. Our plan is to first describe the three classes in section 3.2. In section 3.3 we particularize the needs of each of these classes while section 3.4 summarizes the chapter by listing the resulting set of design criteria.

3.2. Classes of Users Supported

The DELIGHT system is intended to provide congenial support to the following three classes of users:

1. Optimization Experts
2. Engineering Designers
3. System Personnel and Application Program Developers

Optimization experts create or modify the optimization algorithms used by designers. They usually work with the actual algorithm implementations and peripheral user-interface routines which allow algorithm progress to be easily observed and controlled by designers. When a special problem arises that cannot be handled by an existing algorithm, they are usually the ones who are called upon to tailor it or create a new algorithm.

Engineering designers use the DELIGHT system to assist their day-to-day design efforts. The types of designers to be supported by DELIGHT include those who are not computer experts and want to use the system in the simplest

manner, as well as advanced users. Advanced DELIGHT users, who usually have a much greater understanding of system features and operation, often seek new capabilities from the support personnel. During an optimization they also try to insure that they are using the system in the most efficient manner.

There are several subclasses of system personnel. One class contains those whose main task is to support the enhancement of non-algorithmic aspects of the system. This includes meeting the requests of users and porting the system to new computer environments. Another class includes those extending DELIGHT into new areas of engineering by developing the required application-specific versions of the system. The easier this development is to accomplish, the greater the possibility of spreading the use of optimization to many areas.

3.3. Needs of the Various Users

In this section, we consider the needs of the three classes of DELIGHT users. Our purpose is to lead to the set of design criteria presented in section 3.5.

Optimization Experts. The needs of optimization experts and algorithm developers are many. The most important need is the ability to create new algorithms easily. This calls for an interactive high-level programming language that can execute user-written code *very soon* after it has been written. To be interactive, the language can either be interpreted or compiled into a machine-independent intermediate form. A programming language environment brings with it the need for test and debugging aids. There must be the ability to interrupt execution from the terminal, check the values of variables, etc., and resume execution from the point interrupted. Also, the system must be very forgiving to errors that will invariably occur during algorithm development. These include floating point overflows, out-of-bounds array subscripting, and

inversion of singular matrices. Many of these needs appear in general programming systems such as the Wilander [156] or the Carnegie Mellon GLIDE 2 [45] PASCAL systems.

Another wish of algorithm developers to ease the implementation of new algorithms is to have their program code be compact and resemble as much as possible the mathematical description of the algorithm being implemented. Consequently, most of the usual coding errors would be eliminated and the programming time shortened tremendously. This requires high-level *readable* access to an arsenal of common mathematical manipulation and numerical analysis software. Since the mathematics used in optimization algorithms is so diverse, however, a language that is *extensible* [141], i.e., can be extended to resemble new mathematical syntax, is needed.

A final desire is to have *practicing* engineering designers use their algorithms. This leads to the requirement that all DELIGHT application packages be able to use any general algorithms that may be developed.

Engineering Designers. The most pressing need of engineering designers is to gain access to optimization in the many different design environments in which they work. The resulting requirement that DELIGHT be able to interface easily to many different simulators is discussed below.

Other wishes of designers include the following. It must be easy to convey their design specifications to DELIGHT. There must be a way for them to categorize design goals as either objectives to improve or as constraints that must be satisfied. Also, it must be possible to indicate the relative importance of the various specifications. Lightner and Director [94] emphasize this requirement in one of their approaches to solving multiple criterion optimization

problems. The system must allow *arbitrary* formulation of these specifications. This calls for a general expression capability similar to that discussed in ISPICE [5, 7] and A2OPT [58] in section 2.2. Also, the problem formulation must be independent of any particular optimization algorithm selected. The need to be able to easily input their design problem to DELIGHT calls for a powerful *user-oriented problem entry facility*.

Engineering designers have several requirements that pertain to the optimization algorithms that they will use. The first is that there must be a way of selecting a "good" algorithm easily. This selection must be based on an interactive exploration of choices available in a library of algorithms. Moreover, these algorithms must be able to solve complex design problems, have guaranteed convergence behavior, and have parameters which can tune their performance to the particular problem. In order to carry out the tuning of the algorithm, designers must be given knowledge of how well the algorithm is performing. The performance can be very sensitive to the initial values of the design parameters and the conditioning of the problem formulation as well as to the values of the internal tuning parameters. Due to the complex information that must be conveyed, this necessitates graphical feedback on algorithm performance, most effective in color. Myers [104] discusses the perception of symbols, the avoidance of cluttering, character fonts, and other important considerations *needed* in such graphics design. After a designer detects that the algorithm is performing poorly he needs: (1) to be able to stop execution and adjust algorithm or design parameters, modify his problem formulation, or even select a different algorithm, (2) to know *how* to make these modifications, and (3) to be able to resume the optimization after any of these changes. These requirements call for graphical output to show *why* the algorithm is not performing well and,

as for algorithm developers, an ability to interrupt execution from the computer terminal.

Another reason designers need interaction is that typically they cannot specify a priori the relative importance of each of several design objectives until the best capabilities of the design have been determined. This exploration of tradeoffs, essential to any modern design methodology, is much more efficient in an interactive environment.

System Personnel and Application Program Developers. The final class of DELIGHT users consists of system personnel and application program developers. Along with the extensibility needs of system personnel for molding such things as problem entry facilities or application-specific commands of DELIGHT into different design environments, are other needs which pertain to the portability of DELIGHT. A logical requirement for the design of DELIGHT to follow is to force all machine dependencies to occur either in a small set of primitive routines that can be easily implemented on many computers, or in preprocessor substitution macros. These techniques have become quite popular recently. For example, achieving portability through the use of macroprocessors is reported by Brown [25], Hall et al. [61] list the primitive subroutines they used for implementing their portable UNIX-like shell in Ratfor[2], while Stewart [143] discusses both preprocessors and primitives. To be able to easily develop DELIGHT applications packages, it must be easy to *interface* DELIGHT to any existing simulation program. This requires a well-defined simulation interface methodology which at the least consists of two features. One is the ability to easily add existing routines to a set of DELIGHT built-in routines which are callable from the interactive programming language. The second is the ability to

[2] See appendix C for a list of DELIGHT primitives.

access variables from the built-in routines so that they can be manipulated in the same manner as ordinary variables of the interactive programming language.

3.4. Resulting Design Criteria

In this section we summarize the previous sections of this chapter by listing the set of design criteria for the DELIGHT system. These criteria are that an optimization-based computer-aided design system must contain:

- an interactive high-level programming language and associated test and debug aids,
- for easy extension, a language parser which is generated by an automated parser generator (or compiler-compiler),
- language extensibility which allows algorithms to resemble mathematical descriptions and eases the entry of the system into new areas,
- a built-in library of common mathematical software such as from LINPACK [43] or the Harwell Subroutine Library [10].
- a user-oriented problem entry facility allowing: (1) arbitrary problem formulation through a general expression capability, and (2) a means of conveying the relative importance of design specifications,
- a methodology for performing tradeoff of problem specifications,
- a library of optimization algorithms,
- the ability to tune or substitute algorithms in the middle of an optimization run,
- color graphics features for displaying algorithm and problem performance that are independent of the particular graphical display device used,
- a simulation interface methodology that allows easy interfacing to existing simulation programs, and
- machine dependencies which have been grouped to allow easily porting to other computer systems.

In the next chapter we survey various DELIGHT features which help it meet the above criteria.

CHAPTER 4

DELIGHT System Features

4.1. Introduction and Overview

We have considered the benefits of applying optimization to engineering design and identified many shortcomings of previous attempts to achieve this goal. In this chapter we survey various DELIGHT features that help to meet the design criteria given in section 3.4. In order to explain more easily other features of the system, the chapter begins with a top-to-bottom description of RATTLE, the built-in programming language of DELIGHT. This description begins with such fundamental notions as what it is to be used for and why it was created, followed by a description of the syntax of several statements that will be used many times throughout this dissertation. Additions needed to make RATTLE an interactive language include the ability to interrupt execution and to catch certain run-time errors such as numeric overflows. The *define* and *macro* extensibility features of RATTLE are presented and used to create powerful matrix operation macros. Such macros help achieve the goal that algorithm RATTLE code resemble its mathematical description.

The primary purpose of the DELIGHT system—to apply optimization techniques to engineering design— is presented in two parts: the problem side in section 4.4 and the algorithms side in section 4.5. The problem side is considered first because the mathematical programming formulations and their corresponding problem description facilities presented influence the algorithms

and graphical displays needed to solve them. First, a single-cost problem formulation is presented, capable of handling a broad class of optimization problems. However, to serve better our purpose in a design environment, a new multiple objective formulation is proposed that attempts to capture the essence of how a designer would like his design objectives to tradeoff. The crucial interaction between a designer and the optimization process as well as human-engineered graphics to support this interaction are described later in the chapter.

Optimization algorithms capable of solving complex design problems and having guaranteed convergence properties form a cornerstone of the DELIGHT system. We first introduce a library of optimization algorithms. After giving the structure of a typical algorithm found in the library, we demonstrate this structure by presenting an important class of algorithms known as *methods of feasible directions*. These methods are particularly good for engineering design problems. In fact, the algorithm we describe in section 4.5.2.2 for solving problems posed using our multiple objective formulation consists of several enhancements to the basic method of feasible directions. We conclude this part with how the optimization problem set up and the optimization algorithm chosen are coupled. This involves user aspects as well as a special software interface that allows algorithms to be clearly written while maintaining efficiency.

Due to the complex information that must be conveyed about both problem and algorithm performance during an optimization run, graphical displays are needed. General graphics features in DELIGHT are first presented in section 4.6.1. We then introduce several graphical displays, constructed from the basic graphics primitives, that are used to exhibit this performance. The *performance comb* display introduced is instrumental in allowing design problem

tradeoffs to be made and their effects to be observed. Using this display, we emphasize a tradeoff methodology that has been applied successfully to significant practical problems such as those covered in chapter 5. Furthermore, in engineering design it is usually far too costly to obtain a truly optimum solution. Whereas in the past, most uses of optimization in engineering design stressed finding such a solution, we strive for *performance improvement* in the first few optimization iterations. The methodology we introduce insures that a designer's wishes are accurately reflected during each and every iteration of an optimization run.

Since the DELIGHT system is intended for many different areas of engineering design, we close the chapter by introducing a simulation interface methodology and other necessary features that facilitate the coupling of DELIGHT to existing simulation programs. Part of the simulation interface exploits the compile-time macro feature mentioned above to carry out certain table lookup operations just once thus ensuring greater efficiency during the actual optimization execution.

4.2. RATTLE Language

In order to meet the various goals of DELIGHT, an interactive programming language is needed with very particular features. The subsections under section 4.2 give an overall description of the design and features of the *RATTLE* language. RATTLE is an acronym for "RATfor Terminal Language Environment". The functions of RATTLE in the DELIGHT design system are discussed in subsection 1. Subsection 2 explains why a new language is required by first considering the need for a non-standard language compiler and then discussing why existing languages are unsatisfactory. The question of execution efficiency is taken up at

the end of the subsection. In subsection 3, we discuss why RATTLE was designed to be similar to the existing language Ratfor [78]. Several preliminary decisions in the design of RATTLE are discussed in subsection 4. In subsection 5 we illustrate basic RATTLE language statements and other necessary and helpful language features. Arrays whose sizes may vary at run-time is a feature that facilitates the implementation of optimization algorithms. Interactively generated interrupts for stopping RATTLE execution are presented in subsection 6. Subsection 7 discusses the nature of RATTLE interactive execution and its importance to *incremental program development*. Finally, subsection 8 delves into the various aspects of DELIGHT extensibility, including Ratfor-like *defines* with extensions and compile-time *macros*.

4.2.1. Functions of the Interactive Language in DELIGHT

The interactive programming language must serve several functions in the DELIGHT computer-aided design system. The first is for describing design problem objectives and constraints. Since in engineering design these will often depend upon outputs of a simulation program, we may view them as composite functions—in which the "outer functions" are coded in the DELIGHT language and the "inner functions" are provided by the simulator. Since the parts of a design problem formulation to be coded in the DELIGHT language must be coded by designers, the language must be fairly easy to learn. One way of accomplishing this is to make it similar to a popular existing language. Being easy to learn is also a requirement of designers for the second function of the language—the implementation of problem-dependent output procedures. These procedures display on the terminal screen problem performance in whatever form desired

by the particular designer. They are automatically invoked after each major step in DELIGHT optimization algorithms. Finally, throughout an optimization run, designers will use the language to perform scratch pad calculations. The purpose of these may be to verify further the performance of their design or to determine how well the optimization algorithm itself is performing.

An important function of the programming language in DELIGHT is to implement optimization and other computer-aided design algorithms. This function embodies both the development of algorithms as well as their execution by designers. The requirements in this regard for both optimization experts as well as designers have been covered previously in section 2.3. Related issues are also discussed in the next section.

The final function of the interactive language is to create new user commands and features from a few built-in system features. In fact, although DELIGHT has a large number of commands and other user features, it has a relatively small number of intrinsic, "built-in" primitives (language statements and functions); most commands are simply composed of these basic primitives. This function of the language allows the DELIGHT system to meet the goal of being extended easily to any design environment with few or no modifications to DELIGHT (Ratfor) source code.

The following table summarizes the functions of the language:

Functions of the DELIGHT Interactive Language	
1	To describe the design problem.
2	To create problem-dependent output.
3	To perform side calculations.
4	To implement and run algorithms.
5	To create new features from old.

4.2.2. Why a New Language is Needed

This section shows why a new programming language is needed by considering each of the language functions given in the preceding section. There are two parts to this requirement. The first is the need for a new form of language compiler that operates *interactively*. The second is why existing programming languages, used without modification, are unacceptable and hence a *new* language is needed.

We first address why the language of DELIGHT must operate interactively. Consider the previous language functions of describing design problems, implementing algorithms, and running algorithms with the ability to exercise interactive control. Suppose we require that both problem description functions and algorithm procedures be written in a standard, non-interactive language such as Fortran[1]. This makes it very difficult and time-consuming to develop or modify algorithms or to change a problem formulation due to the lengthy load/linkage phase needed for a large program[2] and the inability to execute statements or groups of statements one step at a time. INTEROPTDYN used this approach,

[1] In this discussion, we often use "Fortran" to indicate any language whose normal working cycle consists of compile, link, and execute phases. In the context of DELIGHT, the language would probably be Ratfor [78] (as explained in section 4.2.3 below) although we, in particular, usually choose to avoid using this name due to the possible confusion between "Ratfor" and "RATTLE".

[2] This situation may soon change since Kleckner [81] has recently demonstrated a fast "on-the-fly" load/linkage capability for C procedures on the UNIX operating system.

which resulted in several different INTEROPTDYN versions in existence at Berkeley, each of which required a skillful and time-consuming Fortran programming effort. INTEROPTDYN demonstrated that this "all Fortran" approach *does* allow a user to exercise control over algorithm execution. An existing Fortran optimization algorithm subroutine was modified to make "call-for-interaction" subroutine calls at certain key points that separate major sub-blocks of the algorithm. The user could make interactive requests that caused specific sub-blocks to execute a certain number of times. Interactively generated interrupts could also be handled in this way; upon detecting an interrupt, the call-for-interaction subroutine would switch to an interactive command mode instead of returning to the optimization routine. Of course, modification of the interaction points would require recompilation of the optimization routine followed by re-linkage of the entire INTEROPTDYN program. These reasons, as well as the functions of performing side calculations and easily creating problem-dependent output routines, make it clear that the language in DELIGHT must operate interactively, without a necessary lengthy load/linkage phase.

We now discuss why existing programming languages, used without modification, are unacceptable for meeting the goals of DELIGHT. The foremost reason is that most languages offer many features which are simply not needed in the applications for which DELIGHT is intended. Hence, including these features may result in a compiler that is too large and more difficult to maintain. Another reason is that many present language features are not conducive to a language that is interactive. For example, shared variables such as in Fortran common blocks usually complicate the linkage of several routines into an executable program. Similarly, Fortran, Pascal, or C *goto* statements are gen-

erally difficult to handle in an interactive language processor[3]. Hence, these and other features should either be avoided in our language or redefined to maintain efficiency. We emphasize the idea once put forth by C. A. R. Hoare that one thing a language designer should not do is to "include untried ideas of his own" [79, page 318]. As shown in the next few sections, the RATTLE language borrows most of its features from several existing languages, most notably, Ratfor, C, and Modula.

Another reason why existing languages are unacceptable pertains to the language function of creating new features from old. This is related to the DELIGHT goal that the language should provide a convenient means for expressing algorithms in a manner which resembles mathematical descriptions, in that both have to do with *language extensibility*. Although a few languages exist that offer some extensibility, none offers it to the extent needed in DELIGHT. Thus, powerful language extensibility alone can justify the need for our new programming language. The extensibility of RATTLE is taken up further in section 4.2.8.

The problem of execution efficiency of the new language needs to be addressed. The word *compiler* has been used above rather loosely. As discussed in section 3.3, an interactive high-level programming language that can execute user-written code *very soon* after it has been written can either be interpreted or compiled into a machine-independent intermediate form. Although compilation was chosen for the language in DELIGHT in order to gain increased execution efficiency over interpretation, the language executes considerably slower then a compiled language such as Fortran. However, as discussed in the last section, in engineering design the problem description objectives and con-

[3] If language statements are forced to be numbered in a rigid manner as in Basic, *goto* statements can be handled more efficiently.

straints are often composite functions that depend upon outputs of a simulation program, whose execution time requirements are usually far greater than that needed to execute the "outer functions" coded in the interactive language. Similarly, optimization algorithms make calls on Fortran matrix manipulation and numerical analysis routines. Thus, the inefficiency of the language becomes less significant since it can be viewed as a high-level "controller" language which makes calls to more efficient (though time-consuming) Fortran routines.

4.2.3. Why RATTLE is Similar to Ratfor

The RATTLE interactive language of the DELIGHT system was designed to be similar to the existing programming language Ratfor [78], a language which is translated by a Ratfor preprocessor to Fortran and then compiled. There are several reasons for this:

1. The RATTLE language must be easy to learn. Ratfor is a fairly simple language and with a few extensions is adequate for the purposes of DELIGHT. Also, presently many engineers are familiar with the characteristics of Fortran, for example, its syntax rules for expressions.
2. The language must provide *structured programming* constructs for readability and to foster good programming style. The Ratfor control structures are based on the C language [80] and the large amount of maintainable code written in C under the UNIX operating system [131] demonstrates that C has adequate control structures.
3. After a RATTLE procedure is working and fully debugged, it can be converted to Ratfor for efficiency. This conversion is usually undertaken only for highly repetitive calculations without input or output which would run much faster in Ratfor.

4. DELIGHT itself is written in Ratfor. The Ratfor language was chosen for implementing DELIGHT because Ratfor shares Fortran portability since it is translated to standard Fortran. This, and the fact that a portable Ratfor preprocessor is available, are essential to meeting the DELIGHT goal of portability. Another language translated to Fortran that could have been chosen is the EFL language of Feldman [46]. It allows not only C-like control structures but also powerful data structures. However, to our knowledge, no portable EFL preprocessor written in Fortran exists; EFL itself is written in C and runs under UNIX.

4.2.4. Preliminary Design Decisions

The various DELIGHT design criteria which call for an interactive high-level programming language as well as the considerations in the last few sections have lead to the design and implementation of the RATTLE language. We first discuss a few important design decisions that eliminate the need for some features of Ratfor while adding a few others.

No Type Declarations. Probably the simplest usage of RATTLE is as an interactive "calculator" for performing side calculations while debugging or during an optimization run. The values of arbitrary expressions can be printed and variables and arrays can be created and assigned values simply by typing the appropriate RATTLE statements directly into the terminal. Due to this "calculator mode" usage, an early design decision made was to not require the *type* declaration of any scalar variables or arrays. This is directly opposite to the current trend for *strong typing* in programming languages such as Pascal [69]. However, strong typing is generally needed to catch programming errors in large programs, while most optimization algorithms to be implemented in

RATTLE consist of subprocedures that are small and contain very few variables. Moreover, as Raskin states in a recent letter [130], there is no justification that declaring all variables improves the reliability of computer programs; declarations separate the information about a variable from its use thus violating the idea of "locality". (Though this comment applies to languages such as Pascal, it does not seem to apply to nested block structure languages such as Algol [36] in which items may be declared at the places where they are required; Algol programs can thus display a high degree of locality.) In DELIGHT, variables created in calculator mode exist in a *pool* of double-precision floating-point scalar variables and arrays.

Import Statements. The next step in the design of the RATTLE language was the introduction of a facility for breaking a program up into a number of self-contained units that communicate with each other in a precisely defined way. *Procedures* provide this facility and in essence make the programming of large projects feasible. In DELIGHT, RATTLE procedures allow a group of RATTLE statements, the procedure *body*, to be called for execution as a unit from several places even though the procedure is compiled only once.

There is a requirement in optimization algorithms and other programs for procedures to share variables. User-settable algorithm parameters are an example of variables that need to be shared. One way of accomplishing this is simply to pass the shared variables as arguments to each procedure. But this can lead to very long argument lists. Another technique is to make all variables from the pool mentioned above global, that is, accessible to all procedures. Thus, shared variables would first be created in the pool, and *every* procedure would share them. This is similar to the *scope rules* in Pascal for two levels of procedures, the outer level being a fictitious procedure containing all of the pool

variables. However, this technique can lead to unsuspected variable name clashes with variables outside the procedure body; all procedure variables would have to be given unique names. This would make it very difficult to write procedures in a modular fashion, i.e., without knowing the precise context in which they were to be used. The approach adopted in RATTLE is borrowed from the programming language Modula [158]. Shared variables are still created in the pool. However, instead of having automatic access to all of the pool variables, procedures list those variables that they intend to use with an *import* statement. Other variables which are created inside the procedure *body* are called *local* variables and are not considered part of the pool. There is no possibility of having name clashes with local variables from other procedures or from the pool variables. RATTLE procedures and *import* statements are discussed further in the next section.

Binding of Local Variables. Another important consideration in the design of RATTLE is the *binding* of local procedure variables to physical memory locations as explained in chapter 4 of Barron [19]. How variables are bound is affected by the following consideration. *Recursive* procedures have not been necessary for the implementation of most reported optimization algorithms. Thus, RATTLE, like Ratfor, does not allow recursion. In other words, no procedure or function may call itself or any other procedure which calls itself. The absence of recursion allows *static allocation* to be used; local procedure variables are *bound* to fixed memory locations. This has the effect that local variables retain their values between procedure calls. This is particularly important in DELIGHT since it allows the values of local variables to be displayed after execution has been interrupted or during debugging without the need to add special debug print statements to procedures.

Variable Length Arrays. Optimization algorithms often involve many arrays, including work arrays, whose dimensions are related to the dimension of the problem being solved. The following possibilities exist for handling RATTLE work arrays that are needed by algorithm procedures:

1. Force locally declared arrays to be of fixed dimension. This then requires that the dimensions be large enough to handle the largest anticipated optimization problem. An obvious disadvantage is the waste of storage if many large arrays are declared in many procedures.
2. Declare space for work arrays outside of the procedures and pass these as additional arguments to the procedures. This is the alternative used in the Fortran Program Library for Optimization of Gill et al [51]. However, this has the disadvantage that it would take away some of the elegant simplicity of short argument lists; the argument list of a given procedure would include not only input/output variables, but also arrays whose sole purpose would be for temporary intermediate results.
3. Provide arrays whose dimensions may vary dynamically at run-time. In Algol [36], procedures may contain declarations of arrays whose dimensions depend on local scalar variables of the procedure. Similarly, the Carnegie Mellon GLIDE 2 engineering design system [45] allows variable length one-dimensional arrays of components of the same type. The decision made for RATTLE was to allow *all* arrays, both local and nonlocal pool arrays, to have any number of dimensions that depend on *arbitrary* RATTLE expressions (the syntax is shown in the next section). This avoids the disadvantages of alternatives 1 and 2 above. In particular, procedure work arrays need never be included in the procedure argument list.

An additional advantage of handling algorithm work arrays in this way is the flexibility it provides if new algorithm sub-block procedures are developed or existing ones are modified. If the new algorithm requires additional work arrays for temporary results, the argument list of the sub-block procedure is not changed and any other procedures that call that sub-block procedure need not be modified.

4.2.5. Basic Language Statements

The basic language statements and other features of RATTLE are displayed in this section. We start with the simplest statement for printing the value of numeric expressions, followed by more advanced features for controlling the format of what is printed. We then take a look at various other language statements including assignment, *if*, and various types of loop statements. The section ends with a discussion of RATTLE procedures and functions and two new statements for sharing variables between procedures or functions. Since RATTLE is an interactive language, the introduction to these statements is best accomplished by showing what would actually appear on the terminal screen. The **boldface** text in all of the examples shown here is what the user types or places in a file using an interactive editor[4]. The DELIGHT prompt string is "1>"; when this is seen on the terminal, DELIGHT is waiting for the user to input either a command or a RATTLE language statement.

Simple Unformatted Output. The simplest way to get the value of arbitrary numeric expressions is with the *print* statement. The print statement may be followed by any number of expressions as in the following examples:

[4] The text shown here in **boldface** can actually be typed directly into the terminal for "hands-on" experience with RATTLE and DELIGHT.

```
1> print 1.3
 1.300
1> print  1/3  sin(3.1416/2)  2**64
 .3333  1.000  1.845e+19
```

In the second example, the operator "**" stands for exponentiation. Thus, "2**64" means 2 to the power of 64.

Formatted Output Using printf. Whereas the *print* statement does not allow any control of the format of the numbers printed, the *printf* statement does. Patterned after the *printf* statement in the C programming language [80], it requires a quoted *format control string* followed by from 0 to 6 arguments which must be in one-to-one correspondence with *conversion specifications* in the control string. Unlike Fortran *write* and *format* statements, this syntax has the advantage of keeping both the format and the list of variables to print in the same statement. The following two examples show the output of one and then two real numbers:

```
1> printf '%r/n'  1/3
 .3333
1> printf 'min=%r  max=%r/n'  -2**8  2**9
min=-2.560e+2  max= 5.120e+2
```

The quoted control string contains two types of objects: ordinary characters, which are simply copied to the output, and *conversion specifications*, each of which causes conversion and printing of the next successive argument on the line. Each conversion specification is introduced by the character *%* and ended by one of the conversion characters *i*, *r*, *c*, *s*, or *p*. The meanings of the conversion specifications is adapted from those given in the C programming language manual [80]. For information on all the conversion characters, see the *DELIGHT Reference Manual* [110]; only the *r* conversion specification is discussed here.

To output a real number, *%r* may be used as in the above examples and has a

default of 4 significant figures. The number of significant figures printed may be controlled by, e.g., using *%.7r* to get 7 significant figures printed to the right of the decimal point. The */n* means output a *NEWLINE*, i.e., go to the next output line, at that point in the output; notice the results of the first example below: an extra blank line has been output due to the leading */n* in the format control string. The second example below shows that a / character is output by preceding it by another /; the / character is actually an *escape character* which changes the meaning of any character it precedes.

```
1> printf '/n  A=%.6r/n  B=%.2r/n'  1.0  2000/2

  A= 1.000000
  B= 1.00+3
1> printf 'Answer is 3//4/n'
Answer is 3/4
```

Number Conventions for Post-attached Units. To facilitate its use in an engineering environment, the RATTLE language follows SPICE [150] and ISPICE [5, 7] in supporting certain metric scale factor suffixes which may be attached to any number. In RATTLE, a number may be an integer such as 12 or -44, a floating point number such as 3.14159, either an integer or floating point number followed by an integer exponent such as 1e-14 or 2.65e3, or either an integer or a floating point number followed by one of the following scale factors:

$$p \triangleq 10^{-12} \quad n \triangleq 10^{-9} \quad u \triangleq 10^{-6} \quad m \triangleq 10^{-3} \quad k \triangleq 10^{3} \quad me \triangleq 10^{6} \quad g \triangleq 10^{9}$$

Letters immediately following a number that are not scale factors are ignored, and letters immediately following a scale factor are ignored. Hence, 10, 10*v*, 10*VOLTS*, and 10*hz* all represent the same number, and *M*, *mA*, *MSEC*, and *mwatts* all represent the same scale factor.

Assignment Statements and Continuation. Assignment statements in RATTLE are identical to those in Fortran. They obey the Ratfor continuation rule

which allows them to be continued on the next line if they end with a character which could not possibly legally end an assignment. In particular, they are continued if they end in any of the characters:

+ - * / , (| &

The last two characters above are logical operators, discussed later.

Several RATTLE assignment statements are shown below. Note that the prompt character changes to "}" after a partial statement has been typed in but before the complete (executable) RATTLE statement has been typed. Wilander [156], in the Pathcal program development system for Pascal, also relies on different prompt characters to indicate the state of the system. However, in Pathcal, the absence of a prompt string altogether indicates that the system is awaiting more input. In DELIGHT, the prompt "}" was chosen because in many cases, the character "}" itself is expected to close a *statement block* (see below).

```
1> Nparam = 2
1> array grad(Nparam)
1> grad(1) = 2 + 3*108
1> grad(2) = grad(1) -
1}           3 - 4/108
1> gradnorm = sqrt( grad(1)*grad(1) + grad(2)*grad(2) )
1> gradmax0 = max ( 0 ,
1}    grad(1), grad(2) )
```

In the above examples, an array has been declared with the *array* statement and the built-in RATTLE functions *sqrt* and *max* have been used.

If-statements. The purpose of an *if-statement* is to test the value of a logical expression and execute a RATTLE statement if the logical expression is *TRUE*. This statement is needed to allow conditional execution of parts of an optimization algorithm. The general form of an if-statement is:

```
if logical-expression
   RATTLE-statement
else
   RATTLE-statement
```

Unlike Fortran or Ratfor, the logical expression need not be surrounded by parenthesis. The *else-clause*, i.e., the word *else* and the associated RATTLE statement are optional; if not there and the logical expression is FALSE, execution just falls through to the next statement (or DELIGHT awaits further terminal input). The following are several if-statements:

```
1> if ( gradnorm == 0 )
1}    printf 'Gradient has become zero./n'
1} else
1}    printf 'Continuing with nonzero gradient./n'
Continuing with nonzero gradient.
1> if Nparam>0
1}    print Nparam
1} go
 2.000
```

Note that the second if-statement above does not execute immediately since DELIGHT is waiting for a possible else-clause; typing *go* forces it to execute. Of course, the else-clause may be given, as in the first example, and no *go* will be needed.

The first if-statement above also shows the RATTLE relational operator, ==, for checking for equality of two quantities. The logical expression *A==B* is true if variable A equals variable B. Other RATTLE arithmetic, relational, and logical operators along with their precedence are shown in the following table. The upper entries have *higher precedence* than the lower: *A+B*C* is automatically grouped as *A+(B*C)* since * has a higher precedence (lower numeric value) in column one of the table than +. Operators with the same precedence value in column one are grouped left to right: *A*B/C* is automatically grouped as *(A*B)/C*. As a final example, *a>=b|c!=d&e==f* is grouped as

(a>=b)|((c!=d)&(e==f)).

RATTLE Expression Operators		
Precedence	Operator	Meaning
1	**	Exponentiation
2	*	Multiplication
2	/	Division
3	+	Addition
3	-	Subtraction
4	<=	Less than or equal to
4	<	Less than
4	==	Equal to
4	!=	Not equal to
4	>=	Greater than or equal to
4	>	Greater than
5	!	Logical not
6	&	Logical and
7	|	Logical or

Statement Blocks. To make a RATTLE statement such as *if* act on more than one statement, the statements must be surrounded by curly brackets. This allows one to program the idea: "if something is true, do this group of things". The use of curly brackets for a statement block is shown in the following if-statement:

```
1> if ( gradnorm != 0 )  {
1}     printf 'Gradient is nonzero./n'
1}     gradnorminv = 1 / gradnorm
1}     }
1} go
Gradient is nonzero.
```

Barron [19] also points out that statement blocks encourage the production of programs displaying a high degree of locality that is important in the context of virtual memory computers: a program with good locality tends to have a small

working set and therefore performs well in a paged environment.

Separation of Statements by Semicolons. Statements (or commands) may be separated by semicolons on the same line as shown in the following examples:

```
1> print 1 ; print 3k/6 ; date
 1.000
 5.000e+2
Date: 11/04/82  Time: 03:13:12
1> if (gradnorm!=0) { printf 'Nonzero/n' ; gInv=1/gradnorm }
1} go
Nonzero
```

Loop Statements: While, Repeat-Until and For. These RATTLE statements, like the if-statement, take exactly one RATTLE statement as their body, unless several statements are surrounded in curly brackets. Their usage is seen by considering the following four ways to add up the entries in a one-dimensional array. Here, we use the DELIGHT comment convention: anything following a "#" character up to the end of the line is considered a comment and is discarded by the RATTLE compiler in DELIGHT[5].

```
1> array z(10)          # Create the array.
1> for i = 1 to 10      # Initialize the array: z(1)=1,
1}    z(i) = i          # z(2)=2, z(3)=3, etc.

1> sum1 = 0                              # METHOD 1
1> for i = 1 to 10
1}    sum1 = sum1 + z(i)

1> sum2 = 0                              # METHOD 2
1> for ( i=1 ; i<=10 ; i=i+1 )
1}    sum2 = sum2 + z(i)

1> sum3 = 0                              # METHOD 3
1> i = 1
1> while ( i <= 10 )  {
1}    sum3 = sum3 + z(i)
1}    i = i + 1
1}    }
```

[5] When typing in any of the examples shown here, comments need not be typed; they are there for clarification and indeed are not in **boldface** type.

```
1> sum4 = 0                          # METHOD 4
1> i = 0
1> repeat  {
1}     i = i + 1
1}     sum4 = sum4 + x(i)
1}     }
1} until ( i == 10 )

1> ## Now check the results.
1> printf '%i %i %i %i/n' sum1 sum2 sum3 sum4
55 55 55 55
```

Additional information about these loop statements is found in the *DELIGHT Reference Manual* [110], especially the not-so-obvious Ratfor-like for-loop used in METHOD 2.

Breaking Out of Loops with the break statement. The *break* statement allows any RATTLE loop to be exited before the normal loop termination. Execution resumes with the statement following the last statement of the body of the loop. In the example below, the inner *j* loop normally would execute 6 times but due to the if...break statement it only executes 3 times, as seen in the output:

```
1> for i = 1 to 2  {
1}     printf 'i=%i/n'  i
1}     for j = 1 to 6 {
1}        printf '     j=%i/n'  j
1}        if ( j == 3 )  break
1}        }
1}     }
i=1
     j=1                    (Note, here, that j is never
     j=2                     printed greater than 3.)
     j=3
i=2
     j=1
     j=2
     j=3
```

Arrays. As mentioned in the previous section, arrays in RATTLE are all dynamic, i.e., may change size at any time. The array statement is an *executable* statement and may appear *anywhere* in a RATTLE procedure, not just at the top. Arrays may have any number of dimensions and *the values given for the sizes of the dimensions may be arbitrary expressions*. The following are exam-

ples of array statements:

```
1> array y(100)
1> array y2(3,3), y3(10,20,30)
1> k = 4
1> array var(k,2*k,k**2)
```

The implementation of dynamic arrays with the DELIGHT *dynamic memory manager* is discussed in appendix A.1.2.

Procedures and Functions. Procedures in RATTLE are analogous to subroutines in Fortran or Ratfor. They allow one to execute as a unit a group of RATTLE statements, the body of the procedure, which are compiled only once. A function is identical in structure except that it contains one or more *return* statements to specify the value to be returned as the "function value". Also, the function *call* or invocation appears in an expression as in *print 2+funval(5)*.

Procedures and functions can have zero or more arguments. If a function has no arguments, it can still be called in any expression by following its name with a set of empty parenthesis as in *print 5+fval()/2*.

The body of a procedure consists of one RATTLE statement. If more than one statement is desired in the procedure body, such statements must be surrounded by curly brackets, similar to the body of loop statements. Exit from a procedure body is automatic when "hitting the bottom", i.e., after the last statement in the body has been executed. To exit from any other place, a *return* statement may be used. For a function, the function value to return is the expression value following the keyword *return*.

Several function and procedure examples are shown below, each example separated by a blank line:

```
1> function foo
1|    return 5+3
1> print foo()
 8.000

1> function foo (x)
                  ^
WARNING(1) Number of arguments changed on an existing procedure
1|    return ( x + 4 )
1> z = 1
1> print foo(1)  foo(z)  foo(-4)  foo(-2*2*z)
 5.000  5.000  0.000  0.000

1> procedure doit (a,b)  {
1|    if ( a == 1 )  print b
1|    else           print -b
1|    printf 'Leaving doit/n'
1|    }
1> doit(1,5)
 5.000
Leaving doit
1> doit(2,5)
-5.000
Leaving doit
```

Note that when function *foo* is defined a second time above, the new function body completely supersedes the previous one. This is the way that optimization algorithm sub-blocks are substituted by the designer; he gives a command which causes a procedure of the same name but with a different body to be compiled, and thus supersede the previous. More will be said about sub-block substitution later.

Imported and Global Variables. As mentioned in section 4.2.4, members of the pool of variables that are not local to any procedure may be "imported" or made known to a procedure by listing the variables or arrays in an *import* statement. The following procedure contains several *import* statements:

```
1> procedure demo  {
1|    import y, y2, y3
1|    import var
1|    # (Remainder of procedure body)
1|    }
```

Outside of procedures, variables may be made global, i.e., known automatically to all procedures without each having to import them, using the

global statement. This statement has the same syntax as the *import* statement above, as shown in the following example:

```
1> global y, y2, var
```

In the DELIGHT system, variables should be (and have been) made global with care; this avoids unsuspecting name clashes with local procedure variables created, for example, by designers.

4.2.6. Interrupts and Run-Time Errors

Interrupts and run-time errors are two mechanisms that cause an executing RATTLE program to suspend execution. *Interrupts* are signals from the external environment that are generated by the user depressing a special key on his terminal. Their ability to interrupt an execution or lengthy calculation is essential in an interactive environment. A user might interrupt in order to examine the progress of an algorithm by displaying the values of certain variables or plotting computed curves. Based on these observations, he might then want to adjust some algorithm parameters and resume execution or start another subcalculation (which might also need to be interrupted).

DELIGHT recognizes two kinds of interrupts generated at the terminal, *hard* interrupts and *soft* interrupts. A hard interrupt is generated when a user presses the special interrupt key on the terminal twice in succession. A soft interrupt results when the key is pressed just once. A hard interrupt causes immediate suspension of RATTLE execution. A soft interrupt may be used to suspend execution at a "major stopping point" instead of at some arbitrary statement, or to alter program flow. This is done by testing in an if-statement the special RATTLE keyword *interrupt*; it becomes *TRUE* after a soft interrupt

has been generated. The body of the if-statement can be a *suspend* statement to suspend execution immediately[6] or simply any RATTLE statement. As we shall see later, in each of the optimization algorithms used in DELIGHT, there is a "major stopping point" where *interrupt* is tested that allows designers to press the special interrupt key once and complete the current optimization iteration before suspending.

In DELIGHT, there is a multi-level interrupt feature that allows execution suspended by an interrupt or run-time error to be subsequently resumed for execution levels up to five deep; the DELIGHT prompt string is the current execution level followed by ">". When first entering DELIGHT, the system shows that it is ready to accept commands by displaying on the terminal the prompt string "1>". After one (hard) interrupt or run-time error, execution suspends and DELIGHT again accepts commands by displaying the prompt string "2>". This may occur up to five levels deep[7]. To resume execution, the user types *resume*; when the current execution is finished, DELIGHT again accepts commands by displaying the prompt string with the number in the prompt decremented by one[8].

The following terminal session demonstrates hard and soft interrupts and the multi-level interrupt feature. First, two different loops are (hard) interrupted to show three levels of interrupt and execution resumption.

---

[6] *suspend* is actually a mechanism for generating a hard interrupt through software.

[7] If the suspension occurs while execution is in a procedure (as opposed to just a set of RATTLE statements which have been typed in at the terminal), then a traceback of nested called procedure names may be obtained by typing the command *trace*.

[8] There is also a command, *reset*, which may be used to immediately set the execution level back to one. After a *reset*, execution cannot be resumed.

```
1> for i = 1 to 5k
1}    k = i                  (After 2 seconds the user   )
                             (generates a hard interrupt.)
Interrupt...
2> print i
 5.920e+2
2> for j = 1 to 5k
2}    k = j                  (After 2 seconds the user   )
                             (generates another hard interrupt.)
Interrupt...
3> print j k
 1.319e+3  1.319e+3
3> resume                    (Resume j loop.)
2> print j k
 5.001e+3  5.000e+3
2> resume                    (Resume i loop.)
1> print i
 5.001e+3
```

In the following procedure, the if-statement in the for-loop tests for a soft interrupt. If one is detected, a message is printed and execution is suspended using the *suspend* statement.

```
1> procedure catch  {
1}   for i = 1 to 20k
1}      if interrupt  {
1}         printf 'Got the interrupt when i = %i/n'  i
1}         suspend
1}         }
1}   }

1> catch()                        (After 2 seconds, the user  )
Got the interrupt when i = 2077   (generates a soft interrupt.)

Interrupt ...
2> reset
1>
```

After the execution of procedure *catch* is suspended, *reset* is typed to leave the interrupted state.

Run-time errors is the second mechanism that affects RATTLE execution. When certain errors which occur during execution are detected by DELIGHT, execution is suspended immediately. This allows DELIGHT to be very forgiving to errors that invariably occur during algorithm development or design problem formulation. Run-time errors can originate either externally or internally. Ille-

gal floating point operations such as numeric overflow are *external* run-time errors; through a machine-dependent Ratfor primitive, DELIGHT receives the overflow signal from the actual computer hardware. Errors such as an attempt to invert a singular matrix or to multiply two matrices that are not conformable are *internal* run-time errors; these errors are discovered by software checks in DELIGHT.

To be more specific, DELIGHT run-time errors include:

Floating-point exceptions such as division by zero, numerical overflow, or illegal arguments to built-in Fortran-like functions such as the logarithm of a negative number, etc.

Out-of-bounds array subscripting, i.e., if the "net" array subscript for an array goes beyond the total array size or is less than one. For example, *array y(5); print y(6)* would suspend along with *array y(2,2); print y(3,3)*. But *array y(3,5); print y(4,2)* would not suspend since the net subscript of 7 [9] is still within the total array size of 3·5=15.

Matrix manipulation errors such as nonconformable matrices being multiplied, inversion of a singular matrix, unbounded solution in a linear or quadratic program, or attempting to find the real eigenvalues of a supposedly symmetric matrix that is not in fact symmetric.

After a run-time error, DELIGHT awaits further command input by printing the prompt string with the interrupt level increased by one, just as if a hard interrupt had occurred.

---

[9] This is Fortran-like column major order array addressing: 4 + (2-1)*3 = 7. See, for example, page 178 of Gries [56].

4.2.7. Incremental Program Development

The RATTLE language supports *incremental program development* [156], that is, the ability to test, by just typing it in, a single statement, procedure, or section of an algorithm, without having to write and load/link a whole program. It is thus possible to construct and test pieces of a program one at a time and later combine them into the whole system. It is possible to test procedures that contain calls to other procedures that have been declared as "dummy", i.e., do not yet exist. This allows the system to be developed either "top down", "bottom up", or the more critical portion first according to the preference of the programmer.

The following annotated terminal session shows the development of a program to perform Newton-Raphson iteration for finding the roots of the two equations in two unknowns $y = x^2$ and $x = y^2$. The program is developed by creating three procedures in succession. We first create and test a procedure that returns the function values:

```
1> procedure Func (x,funval)  {
1}     array x(2), funval(2)
1}     funval(1) = x(1)**2 - x(2)
1}     funval(2) = x(2)**2 - x(1)
1}     }

1> array x(2), f(2)      (Create unknown and function )
1> x(1) = 3              (value test vectors (arrays).)
1> x(2) = 3
1> Func (x,f)            (Check function value procedure.)
1> printv f
Column f(2):
  6
  6
```

As an example of incremental program development, a procedure that computes the Jacobian is now created and various errors are detected and corrected. Instead of typing this procedure directly into DELIGHT as we did above, we first

place it into a file and then *include* it, that is, have its contents read by DELIGHT as if they had been typed in directly. Here a text editor is used to place the Jacobian procedure into the file *Jfile*. Instead of showing the editing session, we simply list the file and then *include* it:

```
1> list Jfile
------------------- Begin Jfile -------------------
procedure Jacobian (x,J)  {
   array x(2), J(2,2)
   J(1,1) = 2 * x(1)
   J(1,2) = -1
   J(2,1) = 2 * x(1,1)
   J(2,2) = -1
   }
-------------------- End Jfile --------------------
1> include Jfile
J(2,2) = 2 * x(1,1)
              ^
ERROR(1) LINE(5) Wrong no. args to array, function or procedure
```

Now, we edit file *Jfile*, change *x(1,1)* on line 5 to *x(1)*, and re*include* the file[10]. Each time the file is *include*d, the new procedure body completely supersedes the old:

```
1> list Jfile
------------------- Begin Jfile -------------------
procedure Jacobian (x,J)  {
   array x(2), J(2,2)
   J(1,1) = 2 * x(1)
   J(1,2) = -1
   J(2,1) = 2 * x(1)
   J(2,2) = -1
   }
-------------------- End Jfile --------------------
1> include Jfile
1>
```

Using the text editor, we now create file *Nfile* containing the Newton-Raphson procedure and RATTLE compile the procedure by including the file. Usage of the *matop* statements below should be clear from the comments; see section 4.3 for more details:

---

[10] Note that this Jacobian procedure has two other errors: *x(1)* on the fifth line of the file should have been changed to *x(2)* and the *J(2,1)* and *J(2,2)* right-hand sides need to be swapped. These errors will be noticed later.

```
1> list Nfile
-------------------- Begin Nfile -------------------
procedure Newton (x)  {
   array x(2), f(2), J(2,2)
   repeat  {
      Func (x,f)                        # Get function value.
      Jacobian (x,J)                    # Get Jacobian.
      printf 'x = %-12.5r %-12.5r ||f|| = %r/n' x(1) x(2) ||f||
      matop Jinv      = inv (J)         # Get Jacobian inverse.
      matop deltax    = Jinv * f        # Compute Newton step.
      matop x = x - deltax              # Update unknown vector.
      }
   until ( ||f|| <= 1.0e-14 )           # Repeat until small norm.
   }
--------------------- End Nfile --------------------
1> include Nfile
```

Since the file was *included* without errors, procedure Newton is ready to execute:

```
1> Newton(x)
x =  3.00000      3.00000      ||f|| =  8.485

RUN-TIME ERROR: Singular matrix in inv ... arg(s): J

Interrupt...
 > trace                          (Print function call traceback.)
Interrupted IN procedure
         errprocmess_      line 33  of file <Merrmess>
called by invproc_         line 26  of file <Minvproc>
called by Newton           line 27  of file Nfile
2> enter Newton                   (Enter procedure Newton so that  )
e> display local arrays           (local variables may be accessed.)
4 arrays:
   J                (2,2)
   Jinv             (2,2)
   deltax           (2)
   f                (2)
e> printv J                       (Print the Jacobian matrix.)
Matrix J(2,2):
  6 -1
  6 -1
e> leave
2> reset
1>
```

We now decide to try a different initial guess:

```
1> x(1) = 4
1> printv x
Column x(2):
  4
  3
```

```
1> Newton(x)
x =  4.00000       3.00000       ||f|| =  1.393e+1

RUN-TIME ERROR: Singular matrix in inv ... arg(s): J

Interrupt...
2> enter Newton
e> printv J
Matrix J(2,2):
  6 -1
  6 -1
e> reset
1>
```

Since the Jacobian, array *J*, seems to be always singular, we examine file *Jfile* and discover the errors in the Jacobian computation. After editing the file to correct the error, the development continues:

```
1> include Jfile
1> x(1) = 3
1> x(2) = 3
1> Newton(x)
x =  3.00000       3.00000       ||f|| =  8.485
x =  1.80000       1.80000       ||f|| =  2.036
x =  1.24615       1.24615       ||f|| =  .4338
x =  1.04060       1.04060       ||f|| =  5.975e-2
x =  1.00152       1.00152       ||f|| =  2.160e-3
x =  1.00000       1.00000       ||f|| =  3.278e-6
x =  1.00000       1.00000       ||f|| =  7.598e-12
x =  1.00000       1.00000       ||f|| =  0.000
1>
```

The Newton-Raphson program now appears to be working as seen by the quadratic convergence in the rightmost column above.

4.2.8. Extensibility: Defines and Macros

A scientific programming language should provide a user with a convenient means for expressing his algorithms in a manner which is similar to his mathematical descriptions or matches his personal programming style. One approach is the idea of a *universal language*—one that tries to provide all programming needs—with the language PL/I [31] providing an example of one of the few practical implementations. For the language needed in DELIGHT to be universal, features would have to be provided for many diverse areas, e.g.,

numerical analysis, matrix manipulation, computer graphics, engineering problem entry, etc. Hence, the compiler would be unavoidably large causing it to be difficult to write and maintain.

A more serious drawback of a universal language from the user's point of view is its lack of "open-endedness". In other words, the syntax and "semantics" of such a language implemented with a conventional compiler would be fixed at the time of implementation, and all users would be bound by the decisions of the language designer. Since it is impossible to foresee all the demands and applications that the RATTLE language in DELIGHT might be required to meet, the idea of a universal language for DELIGHT is abandoned.

Another approach to the design of the RATTLE language is to design an *extensible language*, which starts off with a few features, but which can be extended by users or system personnel who can define necessary features as they arise. Thus the compiler for such a language would have a built-in open-endedness which would allow users to tailor the language to their specific needs. Another advantage of an extensible language is that language features which are never used in a particular version or at a particular installation site need never be implemented, so the size of the compiler can be kept under control.

Solntseff and Yezerski present an excellent survey of extensible programming languages in [141]. They present a classification scheme for extensible languages which is an extension of a macro classification facility proposed earlier. The extensibility mechanisms are grouped on the basis of the stage in the language-translation process during which they are processed. The six clearly defined stages they present are:

1. lexical analysis,
2. syntactic analysis,
3. production of intermediate language,
4. analysis of intermediate language,
5. machine code generation, and
6. machine code conversion.

In DELIGHT, the extensibility mechanisms were chosen to be of the *Type-A extension* class from [141] in which the conversions and substitutions occur strictly during the lexical-analysis stage of the translation process. The reasons for this choice are: (1) this type extensibility should be conceptually easier to understand by the potential users of DELIGHT, (2) it is adequate to meet the extensibility goals presented earlier, and (3) such a "preprocessor" scheme has a more straight-forward implementation. A disadvantage of Type-A extensibility is that substitutions are performed without checking the surrounding syntax, as would be the case if the extension occurred during syntactic analysis.

The extensibility of DELIGHT has probably contributed the most to its success among advanced DELIGHT users. This capability takes two forms, *defines* and *macros*, both of which can be used to create new language constructs or new commands from existing ones, to express an algorithm in a manner which looks like a mathematical description, and to adapt DELIGHT to a particular user's personal programming or design style. Defines and macros thus ease the entry of DELIGHT into new design areas. Before discussing defines in section 4.2.8.2 and macros in section 4.2.8.3, section 4.2.8.1 presents the DELIGHT view of character I/O (input/output) and the concept of *pushback*.

4.2.8.1. Character Stream View of I/O and Pushback

Before discussing RATTLE defines and macros, an understanding of how DELIGHT views character I/O (input/output) and the associated *pushback*

mechanism is necessary. Both concepts are used in DELIGHT *internally* to implement defines and macros. However, these concepts are also important to designers or other DELIGHT users who want to take advantage of the extensibility features offered.

In DELIGHT, I/O is considered as a stream of characters that are read or written one character at a time in sequence; there is no concept of reading the next input *line* as in languages such as Fortran. Similar to the complementary *getc/putc* functions in the C programming language [80], DELIGHT provides the functions *getcpb*[11] and *outchr* for reading the next input character and writing the next output character. At the end of an input line, a *NEWLINE* character is returned by *getcpb*. Similarly, a NEWLINE character must be output using *outchr* to cause output being written to start on the next line.

The *pushback mechanism*, following that of Kernighan and Plauger [79], allows the RATTLE compiler in DELIGHT to receive input that was not actually typed into the terminal and does not come from a file being *include*d. This is accomplished by *pushing back* characters or strings "onto the input" (internally they are simply stored in a stack implemented as a fixed length array). When any DELIGHT routine tries to read an input character (using the built-in function *getcpb*), any characters that have been pushed back are read first. Commands that a user may place in RATTLE macros to push back arbitrary text are shown later in section 4.2.8.3.

The VP/CSS Executive Language [4] contains a similar means of providing responses to a program or command, that would normally be given in response to interactive questions, but instead are known and provided in advance. These

[11] The "pb" stands for "pushback", as explained shortly.

responses are placed by the user into a special area called the *Css stack*. This stack is checked for content before every keyboard release. If the stack is not empty, the first line is taken out and interpreted by whatever environment the system is in, i.e., as a command, program input, editor input, etc. Lines are removed until the stack is empty. Then input is read from the terminal.

Pushback in DELIGHT is similar to the CSS stack in that the pushback stack is checked for content before going to the terminal keyboard or to a file being *included*. However, in DELIGHT, what is pushed back is considered on a character by character basis, instead of on a line by line basis. This allows more types of substitutions to be performed by DELIGHT macros.

To understand conceptually how pushback works, consider the following table in which the left column contains RATTLE code executed, the middle column contains a comment concerning the effects of the execution, and the right column shows characters remaining to be read by the RATTLE compiler, that were either typed in by the user or previously pushed back. Here, as for the *printf* statement in section 4.2.5, */n* indicates a NEWLINE character, the character at the end of an input line. DELIGHT built-in routine *gtoken*, used below, returns the character string of the next *token* or item read from the input in its first argument[12]. If the input token is a number, the second argument of *gtoken* contains the numeric value.

[12] A *token* is either an integer, a real number, a name (sequence of letters, digits, or underscores starting with a letter or underscore), a quoted string (quoted by either " or '), or a single character that is none of the preceding such as "(", "#", NEWLINE, etc.

RATTLE Pushback Mechanism		
RATTLE Code Executed	Effect of Execution	Remaining Input Characters
		1+PI/n
gtoken(TokenString,Value)	TokenString now contains "1" and Value equals 1.	
		+PI/n
gtoken(TokenString,Value)	TokenString now contains "+".	
		PI/n
gtoken(TokenString,Value)	TokenString now contains "PI".	
		/n
PushBack('3.1416')	Push back the character string "3.1416".	
		3.1416/n
gtoken(TokenString,Value)	TokenString now contains "3.1416" and Value equals 3.1416.	
		/n
gtoken(TokenString,Value)	TokenString now contains "/n". Now, the next line of input would be read in.	

This example shows how the word "PI" can be substituted by the characters "3.1416" and is quite similar to *define* substitution of the next section. The determination that a substitution should be made for *PI* is not shown above, but would occur just before the call to (fictitious) routine *PushBack*.

4.2.8.2. Defines and Extensions

RATTLE *defines*, patterned after those in Ratfor [78], allow users, in the simplest usage, to substitute one piece of text for another. For example, *define(TWOPI,6.283185307)* allows users to use the value of the mathematical

constant 2π in any RATTLE expression without actually writing the whole number out. Another important use of defines is to create DELIGHT commands. Indeed, most commands in DELIGHT are actually defines. Examples of such defines are user-oriented commands for invoking RATTLE procedures for graphics. For example, *define(erase,greras())* allows *erase* to be typed to call built-in routine *greras* to erase the graphics screen. Similarly, *window* is a define which allows users to type *window wname* to specify a particular set of world coordinates and corresponding viewport coordinates [106], the latter in the (0,0)-(1,1) coordinate system of the terminal screen (see section 4.6), which have been previously associated with the window *wname*. Defines used in this way create commands that are easy to use and that make RATTLE code more readable by hiding unimportant details or complex constructs.

Several extensions to the simple Ratfor define include:

1. arguments,
2. literal strings, which must appear when using the define,
3. optional arguments,
4. default values for optional arguments,
5. automatic argument quoting, and
6. multi-line defines.

The following terminal dialogue and discussion introduces each of these features individually by showing the creation and use of simple example defines. Also shown are several invalid uses of some of the defines and the error messages that result.

```
1> define (a,4)                          ## Simple defines.
1> define (b,5) define (c,6)
1> print a b c**2
 4.000   5.000   3.600e+1
```

```
1> define (p x,print x**2)          ## 1. Arguments.
1> p 5
 2.500e+1

1> define (p x 'over' y,print x/y)  ## 2. Literal strings.
1> p 5
ERROR: expecting "over " for define "p"
1> p 5 over
ERROR: missing arguments after "p"
1> p 5 over 2
 2.500
```

Next we show the extensions which allow optional arguments and default values for optional arguments. Consider the following examples:

```
1> define (debug ; x , print k x)  ## 3. Optional arguments.
1> k = 3
1> debug
 3.000
1> debug k**2
 3.000  9.000

1> define (debug ;x=4 ,print k x)  ## 4. Default values for
1>                                 ## optional arguments.
1> debug
 3.000  4.000
1> debug k**2
 3.000  9.000
```

In the first example, the argument x comes after the semicolon and is thus an optional argument. If no argument is typed after *debug*, any occurrences of x in the definition are substituted by a null string, i.e., the x's are eliminated. In the second example, if no argument is typed after *debug*, any occurrences of x in the definition are substituted by the default value of 4. The rule for default values, explained more fully in the *Argument Conventions* subsection of section DEFINES(4b) of the *DELIGHT Reference Manual* [110], is that in a define declaration, any optional argument may be followed by "=" and the default substitution string, which is all characters up to the next blank.

In a define declaration, any argument which is preceded by two consecutive single quotes as *arg* in *define(p ''arg,printf arg)*, during definition substitution,

is surrounded by quotes before substituting the argument into the definition. This means that for the define just given, typing *p xyz* is the same as typing *printf 'xyz'*; the argument value *xyz* is surrounded by quotes before substituting it into the definition *printf arg*. This *Double Quote Convention* (see section DEFINES(4b) of [110]) is similar to the QUOTE form in the LISP programming language [154] which receives and transmits (as an S-expression value) its argument unevaluated; here, the argument string being quoted is prevented from being evaluated as a RATTLE expression.

In the following example of the Double Quote Convention, assume *ListProc* is a procedure that lists the contents of any file whose name is passed to it as a quoted string. Then the following define allows one to list the file *tmpfile* by typing *list tmpfile* instead of the more awkward *ListProc('tmpfile')*:

```
1> define (list ''fn , ListProc(fn))  ## 5. Automatic
1>                                    ## argument quoting.
```

Whereas the syntax of an ordinary define is:

```
define ( DefineName [arguments] , DefineDefinition )
```

a *multi-line* define has the following syntax:

```
define  DefineName  [arguments]
   Definition Line 1
   Definition Line 2
   ...
   end
```

The purpose of multi-line defines is to allow defines to have an arbitrary length definition. In the next piece of terminal dialogue, a multi-line define is shown. It does not have a leading left parenthesis and it ends with the keyword *end*.

```
1> define p x                    ## 6. Multi-line defines.
   print x
   print -x
   end
1> p 5
 5.000
-5.000
1>
```

Note that each time *p* above is redefined, the new definition completely supersedes the previous definition.

A define which uses several of these extensions is shown below:

```
define (vector x1 y1 x2 y2 ; 'in' ''C='white' ,grcolp(C);grvect(x1,y1,x2,y2))
```

This define, *vector*, has four required arguments, *x1*, *y1*, *x2*, and *y2* (which appear before the first semicolon), and one optional argument, *C*. In the above define, *in* is quoted and thus has to be typed explicitly when using the define. The definition, appearing after the comma, contains two calls to built-in DELIGHT routines, *grcolp* and *grvect*. The color name that is typed for argument *C* will be surrounded by quotes before being substituted into the definition since *C* on the left is preceded by two quotes. However, since *C* is optional, not typing any color will cause *C* to default to white.

This define could be used in any of the following ways:

```
vector  0  0  1  1
vector  0  0  .5  .5  in  red
vector  sin(x)  1+sin(x)  cos(x)  1+cos(x)  in orange
```

If the second example above had been placed in a procedure, it would be, using the definition above, as if the following had been placed there:

```
grcolp('red') ; grvect(0,0,.5,.5)
```

4.2.8.3. Compile-Time Macros

DELIGHT has extensibility needs that cannot be handled by the simple define substitution mechanism discussed in the previous section. These have to do with making conditional substitutions, that is, substitutions that are based on the arguments that are used with the define. For example, there is no way to make a define called *MatrixFunc* which will allow the statement *MatrixFunc A=inv(B)* to substitute the procedure call *Inverse(A,B)* but the statement *MatrixFunc A=adj(B)* to substitute *Adjugate(A,B)*. The definition substituted when a define is encountered is fixed in structure; only arbitrary argument values can be "dropped" into the appropriate places in the definition.

One approach to conditional substitution was suggested in 1966 by Leavenworth [87], who was one of the first to apply extensibility to high level languages. Leavenworth described a scheme in which different substitutions could occur based on which one of several keyword-introduced clauses appeared in the input; "the substitutions could be qualified by block number." As an example, let us use Leavenworth's scheme to provide the Fortran language with a new and powerful *for* statement. We wish to perform the three text substitutions shown in the table below:

Source Text	Substituted Text
for v1 = F to T by I BODY	v1 = F 100 if (v1 .gt. T) go to 101 BODY v1 = v1 + I go to 100 101 continue
for v1 = F to T times I BODY	v1 = F 100 if (v1 .gt. T) go to 101 BODY v1 = v1 * I go to 100 101 continue
for v1 = F to T log I BODY	v1 = F 100 if (v1 .gt. T) go to 101 BODY v1 = v1 * (T/F)**(I-1) go to 100 101 continue

The first substitution is for a linear variation of a variable from *F* to *T* by adding the increment *I* each pass through the loop, the second multiplies by *I* during each pass, while the third is for a logarithmic variation from *F* to *T* with the total number of points specified as *I*. The substitution for the third case computes a multiplier that will give *I* total points.

Using Leavenworth's scheme we could implement the above substitutions with the *for* declaration:

```
for v1 = F to T { by I | times I | log I }   B1

    v1 = F
100 if ( v1 .gt. T )  go to 101
    B1
    {1 v1 = v1 + I }
    {2 v1 = v1 * I }
    {3 v1 = v1 * (T/F)**(I-1) }
    go to 100
101 continue
```

in which {*n* ...} is substituted if clause *n* appeared in the input. For the above declaration, the clauses *by I*, *times I*, and *log I* are numbered, respectively, 1, 2, and 3.

A second approach to conditional substitution, given by Brown [25] while reporting on achieving portability through the use of macroprocessors, shows an *#if* construction for making the substitution performed dependent on the context or the arguments given. In this approach, *metakeywords*, that usually begin with a special character such as the "#" in *#if*, *#while*, *#elseif*, etc., would be used in a define definition to control the substitution performed. This form of conditional substitution is widely used, notably the "&if" syntax of the VP/CSS Executive Language [4] and the "*if" syntax of Mark Bales' *csubst* text preprocessor [17].

But, as emphasized by Wilander in the Pathcal program development system for Pascal [156], a *one-language system*, in which the substitution control syntax and the original programming language have exactly the same syntax, has the advantage that one can write conditional substitutions or loops over commands without having to learn an additional syntax.

Since the two above approaches are unsatisfactory, a third approach to conditional substitution emerges. In this approach, used in DELIGHT, RATTLE *compile-time macros* are written in the RATTLE language in exactly the same

way as procedures, except the keyword *procedure* is replaced by *macro* (see the example below). This idea is not entirely new. Teitelman [147], in describing the Interlisp programming environment, mentions two kinds of macros. *Substitution macros* associate a template (composed of existing commands) with a new command; this is analogous to RATTLE defines. *Computed macros* are basically LISP expressions, evaluated to produce a new list of operators/commands/expressions; this is analogous to RATTLE macros pushing back a new RATTLE statement.

As mentioned above, macros are written as ordinary RATTLE procedures. However, they are not executed at run time, but rather when their name is encountered during the compilation of RATTLE statements such as procedures or statements read from the terminal keyboard. They can act as filters in the stream of input characters being received by the RATTLE compiler. For example, one can write a macro to scan the next few input tokens, which need not be valid RATTLE code (they are never parsed by the RATTLE compiler), make decisions based on what is found, and then push back valid RATTLE code to be sent to the compiler.

As an example, let us implement the *MatrixFunc* statement discussed in the first paragraph of this section. The comments (after a "#") in the following macro describe the operation of the macro in converting *MatrixFunc A=inv(B)* to *Inverse(A,B)*. All error checking has been excluded for simplicity:

```
macro MatrixFunc  {
   array OutputMatrix(80), InputMatrix(80),
         Function(80), Dummy(80)
```

```
gtoken (OutputMatrix,Value)     # Read "A".
gtoken (Dummy,Value)            # Discard "=".
gtoken (Function,Value)         # Read function.
gtoken (Dummy,Value)            # Discard "(".
gtoken (InputMatrix,Value)      # Read "B".
gtoken (Dummy,Value)            # Discard ")".
if ( Function == 'inv' )
   PushBackF 'Inverse(%s,%s)' Outputmatrix InputMatrix
else if ( Function == 'adj' )
   PushBackF 'Adjugate(%s,%s)' Outputmatrix InputMatrix
else
   printf 'ERROR: illegal function for MatrixFunc/n'
}
```

The various arrays declared above are each for holding a character string containing the next input item (token) returned in the first argument of built-in routine *gtoken*. *PushBackF* is a (fictitious) command for pushing back the output of a printf-like formatted control string (see section 4.2.5). The "%s" fields above are for the character strings returned by *gtoken*.

If *MatrixFunc* were to be used, when the macro name *MatrixFunc* was read by DELIGHT, the macro would execute immediately and use gtoken to read the next few items from the input; these items would be the arguments following *MatrixFunc*. Then, the macro would push back one of the two procedure calls, *Inverse* or *Adjugate*, based on which of the functions, *inv* or *adj*, was read by the macro.

To clarify the difference between a macro and a procedure, consider the following procedure, where *p2* is the name of a precompiled procedure:

```
procedure p1 (C,D)  {
   p2 (C,D)
   MatrixFunc C=inv(D)
   }
```

When *p1* is being compiled, the detection of *MatrixFunc* causes the macro of the same name to be executed, substituting *Inverse(C,D)* for the second line in *p1*. Of course, procedure *p2* is not called at compile time but instead when *p1* is exe-

cuted (after it is finished compiling) via the occurrence of the statement *p1(Arg1,Arg2)*.

As shown in the following section, DELIGHT compile-time macros have been used to provide the RATTLE language with a rich set of easy to use matrix manipulation statements.

4.3. Matrix Macros

Probably the most commonly occurring operations in optimization algorithms are linear algebraic matrix manipulations. To meet the goal of expressing algorithms in a manner which resembles mathematical descriptions as closely as possible, high-level statements for matrix manipulation are needed. Besides enhancing readability, these high-level statements can also hide details of their implementation from a programmer so that his coding task is greatly simplified.

This idea is not new; simple matrix operations exist in other programming languages in which the name of an array stands for the entire aggregate of values [19]. For example, in PL/I, APL, and Algol 68, arrays can be assigned, respectively, using $A=B$, $A \leftarrow B$, and $A:=B$. These make A a copy of B, i.e., there is an element-by-element assignment. PL/I also has a more general facility since it will accept array names as members of the right-hand side of an array assignment. Thus, if A, B, and C are arrays of the same size and shape, $A = 2 * B + C$ is a legal PL/I assignment. Assignments such as B = *ABS(A)*, defined on an element-by-element basis, are also allowed. However, operations such as matrix inversion or singular value decomposition are not allowed since they cannot be substituted by simple nested loops of a scalar operation. In Algol 68 [160], however, the value of a function can be an array and thus a matrix inversion

function could be implemented that allowed the statement $B = inv(A)$. In APL [52], all elementary operators are defined to operate element-by-element on arrays. However, the lack of operator precedence in APL expressions is so different from the precedence rules of RATTLE scalar expressions that APL syntax is excluded from being a model for DELIGHT matrix operations.

The DELIGHT compile-time macro feature of the previous section has been used to enable DELIGHT users to carry out very complex matrix computations—far beyond those of the languages of the previous paragraph—by means of very simple statements. For example, using the macro *linprog*, one can solve a linear programming problem with the following statement:

```
linprog z = argmin { c'*x | x>=0 , x<=d , A*x<=b }
```

where the array z is assigned the minimizing value of x. The macro *linprog* scans ahead, determines what is being requested, and pushes back onto the push-back stack a normal RATTLE procedure call, similar to the macro *MatrixFunc* of section 4.2.8.3. In addition, the *linprog* macro sets up all the necessary input arguments and work arrays for a call to a built-in Harwell Library [10] linear programming Fortran routine. Thus, this macro not only enhances readability but also relieves the programmer of the need to create work arrays and to master the other usually complicated requirements of program library routines.

Presently, there is an arsenal of matrix operation software available to the user through macros. The following table shows examples of a sample of these macros[13]:

[13] Many of the RATTLE macros in this table were initially written by Tommy Essebo from the Lund Institute of Technology while visiting Berkeley one summer.

DELIGHT Matrix Macros	
Computation	Example Macro Usage
Create Identity Matrix	matop I5 = identity(5)
Create Constant Matrix	matop B = array(3,5) of 0.0
Transpose a Matrix	matop A = B'
Add Matrices	A = B + C
Multiply Matrices	A = B * C'
Scalar Times a Matrix	matop A = (3.4) * C
Fill a Submatrix	fill A(3:4,2:8) = B
Clip Out a Submatrix	clip A = B(3:4,2:6)
Determinant of Matrix	det(A) (in any expression)
Condition Number of Matrix	rcond(A) (in any expression)
L2-norm of Vector	||v|| (In any expression)
Inner Product	<<x,y>> (In any expression)
Matrix Inversion	matop Ainv = inv(A)
Solve Linear Equations	lineq A*x = B
Eigenvalues/Vectors	matop Lambda,Ev = eigen(A)
QR Decomposition	matop Q,R = qr(A)
Singular Values/Vectors	matop S,U,V = svd(A)
Linear Programming	linprog Z = argmin { C'*x | x>=0, A*x=0 }
Quadratic Programming	quadprog Z = argmin { x'*Q*x + B*x | A*x<=C }

Most of the nontrivial macros above use LINPACK routines [43]. For the *linprog* and *quadprog* macros, arbitrary linear equality and inequality constraints may be given to the right of the vertical bar, and they may appear in any order. Several of the above macros, including simple binary matrix operations, eigenvalues/eigenvectors, and singular value decomposition, have been implemented for the complex matrices needed in the DELIGHT-MIMO version of DELIGHT [97] for multivariable control system design. They have the same syntax as above except that the word *matop* becomes *cmatop*[14].

[14] This syntax is not yet firm.

4.4. Problem Input Language

The use of DELIGHT for optimization requires a designer to formulate his design problem as a certain standard mathematical programming problem. There are optimization algorithms in DELIGHT that support both a classical problem formulation as well as a recent multiobjective formulation, particularly well suited to engineering design. These are discussed, respectively, in sections 4.4.1.1 and 4.4.1.2 . Corresponding to these two problem formulations are user-oriented problem description facilities that meet the DELIGHT design goal of allowing full freedom[15] in describing a design problem through the general expression capability of RATTLE. We discuss these problem description facilities in sections 4.4.2.1 and 4.4.2.2 .

4.4.1. Mathematical Programming Formulations

Once early optimization efforts escaped from the quagmire of formulating optimization problems as unconstrained, least squares, matching problems as discussed earlier in section 2.1, much more useful problem formulations emerged. The problem formulations presented in the next two sections are discussed from the following simple mathematical point of view. The engineering system being designed consists of design parameters (or unknowns) to be automatically varied by an optimization algorithm. These parameters are referred to as a point $\mathbf{x} = (x_1, \cdots, x_n)^T$ in Euclidean n-space, denoted by $\mathbb{R}^n$ (throughout this thesis, "^{T}" is used to indicate matrix transpose). Each different point in $\mathbb{R}^n$ represents a different system but with the same structure. The idea of fixed structure was emphasized in the discussion of *parametric optimization* in section 1.1 . The responses of the engineering system, be they from whatever

[15] Subject to certain requirements on continuity and differentiability, discussed later.

type of computer simulation, depend on this design parameter vector $\mathbf{x}$. Thus, if v_i, $i=1,\cdots,m$ represent m response functions, we may consider the vector $\mathbf{v} = (v_1,\cdots,v_m)^T$ as a mapping from the parameter space to the response space:

$$\mathbf{v}: \mathbb{R}^n \rightarrow \mathbb{R}^m.$$

Viewed in this way, the simulation program is simply a means of evaluating the vector function $\mathbf{v}(\mathbf{x})$ given a design parameter vector $\mathbf{x} \in \mathbb{R}^n$. The objective and constraint functions presented in the next problem formulations may be directly a function of the design parameter vector $\mathbf{x}$ or indirectly a function of $\mathbf{x}$ by being a function of the system response functions $\mathbf{v}(\mathbf{x})$. In engineering design problems such as electronic circuit design, the latter is more common.

4.4.1.1. Classical Single Cost with Constraints

The simplest optimization problem we will consider is the *unconstrained* nonlinear programming problem:

$$\underset{\mathbf{x}}{\text{minimize:}}\ cost(\mathbf{x})$$

in which the minimum value of some scalar function of the design parameters is sought. In engineering design, however, it is much more likely that there are additional constraints that must be met. An example is a constraint on the power dissipated in an electronic circuit. We then have the *constrained* nonlinear programming problem:

$$\underset{\mathbf{x}}{\text{minimize:}}\ cost(\mathbf{x})$$

$$\text{subject to: } \mathbf{eq}(\mathbf{x})=0 \quad \text{and} \quad \mathbf{ineq}(\mathbf{x})\leq 0$$

in which **eq** and **ineq** are, in general, nonlinear vector-valued functions of the

design parameter vector $\mathbf{x}$, i.e. we have the *Neq* equality constraints $\mathbf{eq}(\mathbf{x}) \triangleq (eq_1(\mathbf{x}), \cdots, eq_{Neq}(\mathbf{x}))^T$ and the *Nineq* inequality constraints $\mathbf{ineq}(\mathbf{x}) \triangleq (ineq_1(\mathbf{x}), \cdots, ineq_{Nineq}(\mathbf{x}))^T$. Since in general, engineering design problems do not involve equality constraints that cannot be handled in a simple manner, we will ignore them in most of this dissertation.

Unfortunately, many engineering design problems cannot be formulated as the standard mathematical programming problem given above. For example, many commonly occurring constraints require that some specification be met *over a range of an independent parameter* such as time, temperature, or frequency. For example, one common temperature constraint is that a system behave in a certain way for every temperature from $-55°C$ to $125°C$. These constraints are called *functional* inequality constraints and must be handled in a special way by optimization algorithms [122, 98]. By adding functional inequality constraints to the above problem formulation (and excluding equality constraints), we arrive at the following *semi-infinite*[16] nonlinear programming problem:

$$\text{minimize}_{\mathbf{x}}: \; cost(\mathbf{x})$$

$$\text{subject to: } \mathbf{ineq}(\mathbf{x}) \leq 0 \quad \text{and} \quad \mathbf{fineq}(\mathbf{x}, w) \leq 0 \quad \forall w \in [W_0, W_c] \, .$$

As illustrated in several engineering design examples by Polak [126], many commonly occurring parametric optimization problems are usually expressed as semi-infinite programming problems.

[16] This name stems from the fact that the second variable of **fineq** in the formulation may be viewed as leading to an infinite number of constraints. See, for example, Gonzaga et al. [54] or Hettich [63].

4.4.1.2. Multiple Objectives with Constraints

In engineering design, the mathematical problem solved by an optimization algorithm may be remote from the real world problem a designer is facing. This is due to the rigidity of the classical nonlinear programming problem given in the previous section. While that problem formulation does encompass the general idea of optimizing some design objective (cost) while meeting various constraints specifications, it fails to take into account several important characteristics of a large class of engineering design problems[17].

First, it is rarely the case that a single objective (cost) has to be optimized. In most applications, various cost functions, hereinafter called *performance objectives*, compete against each other and a compromise has to be reached. The literature in several diverse areas abounds with results on multiobjective (or multicriteria) optimization. One simple technique used quite often for handling such problems is known as the *weighted sum approach*—the weighted sum of the performance objectives is formed in order to achieve a single cost function to minimize. However, this has the disadvantage, particularly acute in an interactive environment, of hiding the physical significance of these objectives when viewing the value of the cost during an optimization. Also, adjusting the weights to vary the emphasis given to each objective is usually cumbersome. Furthermore, Lightner et al. in [94] point out the well-known problem with the weighted sum approach that certain possibly desirable optimization solution points are not achievable no matter what choice of weights is used.

Second, a solution to a problem formulated using the previous mathematical formulation does not allow any violations of the constraints **ineq** or **fineq**. In

[17] As mentioned below, these characteristics were indeed observed during several months of close interaction with circuit designers in an industrial environment.

design applications, constraint specifications are often relatively flexible and do not have to meet any precise numeric values. Hence, moderate violation of a constraint should be allowed in the solution of an optimization problem; often this will permit an optimization algorithm to achieve a better value of the performance objectives. Put in another way, the constraints may often be *traded off* against the objective functions in much the same way tradeoffs occur in all engineering design. Of course, there may be constraints that *cannot* be violated; both types should be handled. (Obviously, it should be more important to an optimization algorithm to satisfy the constraints that cannot be violated. Thus, we are led to designers classifying their constraints into two levels of importance.) Another shortcoming is that the formulations **ineq**($\mathbf{x}$)$\leq$0 and **fineq**($\mathbf{x}$,w)$\leq$0 give no way of estimating the importance to the designer of a given constraint violation.

In the fall of 1981, when DELIGHT was first applied in an industrial environment, we were made painfully aware of these first two difficulties. At that time, using an algorithm that handled the previous single-cost problem formulation, we were forced to formulate all but one of a set of multiple objectives as constraints and to keep interactively modifying their desired values as the optimization algorithm satisfied them. The way tradeoffs were performed was also difficult. To indicate the relative importance of our constraints, they were weighted with multiplicative scale factors. Constraints whose scale factors were large would be "considered first" by the algorithm and would become less violated after a few iterations. Constraints whose scale factors were small had less relative importance and might become more violated. However, the trial-and-error procedure required to adjust the scale factors caused this to be a

painful way of performing tradeoffs[18].

A third and more general deficiency of the previous formulation is that it expresses only partially the knowledge a designer has about his problem. Some of this knowledge, built on experience and physical intuition, is often impossible to express numerically. However, this formulation does not allow the designer to express some easily quantified intuitive knowledge such as the degree of confidence he has in the initial guess provided for each design parameter or what change in each parameter value he considers reasonable to use in an attempt to improve the performance of the design. Such knowledge can be used by optimization algorithms to aid in parameter scaling and thus enhance speed of convergence.

Last, this formulation does not allow design parameter ***box constraints*** such as $-2 \leq x_4 \leq 2$ to be easily given, i.e., without considering them part of the general nonlinear constraints in **ineq**.

The following list summaries the requirements of a new problem formulation. A new problem formulation must allow:

1. multiple objectives,
2. two types of constraints and the ability to specify constraint violation importance,
3. a designer to provide his intuitive knowledge, and
4. design parameter box constraints to be easily expressed.

[18] This author and Ekachai Lelarasmee, Giovanni De Micheli, and Alberto Sangiovanni-Vincentelli spent three months during the fall of 1981 at Harris Semiconductor in Palm Bay, Florida. In spite of the difficulties mentioned above, all was not painful that fall. The four of us stayed in a beautiful 2-story townhouse ***directly on the beach***: right out our back door was the Atlantic Ocean.

Obviously, there is no unique way to meet the requirements mentioned above. The problem formulation described next makes use of some ideas and concepts which seem particularly well suited to designer intuition. The four requirements are handled one at a time.

Multiple Objectives. We address the first difficulty by defining a new problem formulation that allows multiple objectives and by extending the Combined Phase I - Phase II Method of Feasible Directions algorithm of Polak et al. [123] to include the capability of handling multiple objectives (instead of just a single cost function). This algorithm is described later in section 4.5.2.2 .

Two Types of Constraints and Violation Importance. The second enhancement is to define the new formulation in such a way that at a problem solution, violations of certain of the **ineq** or **fineq** constraints can occur if this allows some particularly good performance to be achieved by the objectives. As shall be seen in section 4.5.2.2, another extension of Polak's Feasible Directions algorithm handles this feature by allowing performance objectives and certain unsatisfied constraints to compete equally for improvement. The new problem formulation must also allow a way for a designer to estimate the importance of a given constraint violation. A novel way of capturing the intents of a designer, that is, of having him specify which constraints allow violation and the importance of these violations, is presented next. However, it is important to note that sometimes these intents cannot be completely specified at the beginning of the design process; adjustments in the specifications may appear to be necessary only after the problem has been solved and has yielded an inadequate design. Furthermore, a designer may wish to adjust the specifications quite often during an optimization run to insure that they are continually being traded off in a satisfactory manner or to obtain designs for which various

performance objectives have been emphasized. Thus, the DELIGHT system must provide ways of changing these specifications while an optimization is being performed.

How the new problem formulation allows two types of constraints to be specified is as follows. While we were at Harris Semiconductor, it was observed that not all constraints are perceived by designers in the same way. Therefore, the constraints of an optimization problem are broken into two categories, *hard* and *soft*. Hard constraints consist of those which the designer gives the utmost priority in having the algorithm satisfy and which, once satisfied, the designer wishes to remain satisfied and not take part in any subsequent design tradeoffs. Obviously, any constraint whose satisfaction is necessary for physical realizability, such as a certain quantity being non-negative, should be treated as a hard constraint. Soft constraints, on the other hand, are those which the designer is interested in conveniently trading off against one another and against the performance objectives during intermediate iterations of an optimization run. Of course, the satisfaction of soft constraints should not be required for the system being designed to be feasible. Otherwise, design tradeoffs involving these constraints would be meaningless. As shall be seen later, when entering a constraint, the user has to specify if the constraint is considered hard or soft, the default being hard.

Since objectives and soft constraints may be traded off by the designer, it is important to specify their relative importance to an optimization algorithm. For example, in integrated circuit design, a constraint on power dissipation in a circuit such as $power \leq 1.5watts$ might be very important to prevent chip overheating whereas the constraint $Zin \geq 10megohm$ for high input impedance may be less important since often a considerably lower input impedance is acceptable.

The literature abounds with techniques for specifying the relative importance of objectives in a multiple objective optimization. Our approach, being targeted at "practicality" in the sense that we are interested in having designers use it, is not the result of a comprehensive survey of this entire field of literature. However, we do consider a few relevant approaches below.

Lightner and Director [94] handle the relative importance of multiple objectives by choosing the weights in a weighted ∞-norm minimax optimization in various ways[19]. They define the *canonic weight for the max norm* associated with a particular *noninferior solution* $\mathbf{f}^*$ of the problem

$$\min_{\mathbf{x}\in\Omega} \max_i \{w_i f_i(\mathbf{x})\}$$

as $W \triangleq \text{diag}(w_i^*)$ where $w_i^* \triangleq 1/f_i^* \ \forall i$. Briefly, a *noninferior point* or *Pareto point* for a multiple objective optimization problem is one for which there is no possible change in the unknown vector $\mathbf{x}$ which will cause at least one of the objective functions to decrease and none of these functions to increase. Then, given a canonic weight vector $\mathbf{w}^*$, $\mathbf{f}^*$ solves $\min_{\mathbf{x}\in\Omega} \max_i \{w_i^* f_i(\mathbf{x})\}$ with $\max_i \{w_i^* f_i(\mathbf{x})\}=1$. Of course, since the solution $\mathbf{f}^*$ is not generally known, the weights w_i^* to use in an optimization cannot be determined. However, the concept of a canonic weight for the ∞-norm allows these authors to develop various heuristics for weight selection. One heuristic is to ask the designer to specify a desired solution, say $\mathbf{f}^d$. Then a minimization can be performed with the weight $W^d=\text{diag}(w_i^d)$ where $w_i^d=1/f_i^d$. This approach can be used to explore the *tradeoff surface* of a multiobjective problem by having a designer adjust the desired solution and hence the weights, possibly using the weight selection strategy of these authors.

---

[19] The techniques discussed here are also presented in [23] and [24].

A weakness of these approaches is that only near the end of a *complete* optimization for a particular weight vector does the optimization emphasize the various objectives in a way that corresponds to the designer's desired tradeoff. To see this point, consider the graph in figure 4.1, which plots a hypothetical designer's satisfaction versus the objective function values for f_1, f_2, and f_3. This plot indicates that it is desired to *decrease* all three objectives. In general, these curves might be arbitrary monotonically decreasing (or increasing) curves but for simplicity, let us assume that they are straight lines as shown.

Figure 4.2 shows the three desired objective values f_1^d, f_2^d, and f_3^d that correspond to a level of satisfaction marked by **D**. In figure 4.3, the weighted functions $w_1^d f_1$, $w_2^d f_2$, and $w_3^d f_3$, whose maximum is to be minimized, are shown (the weights are defined above as $w_i^d \triangleq 1/f_i^d$). Suppose that at some point, during the initial part of the optimization suggested by Lightner and Director, the three objectives have values indicated by the dotted line at point **P**. Then the

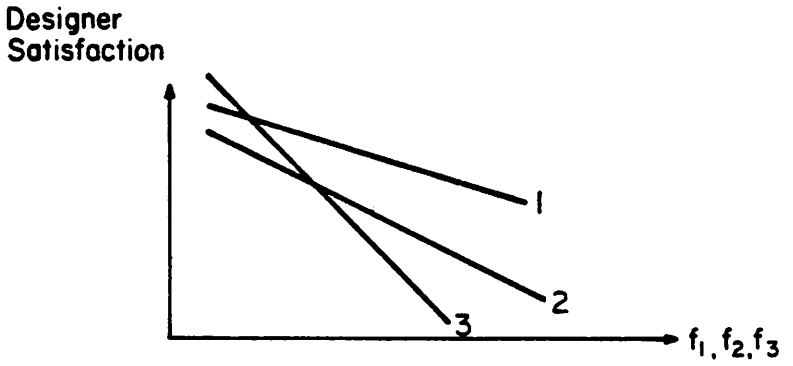


Figure 4.1. Designer Satisfaction Versus Objective Function Values.

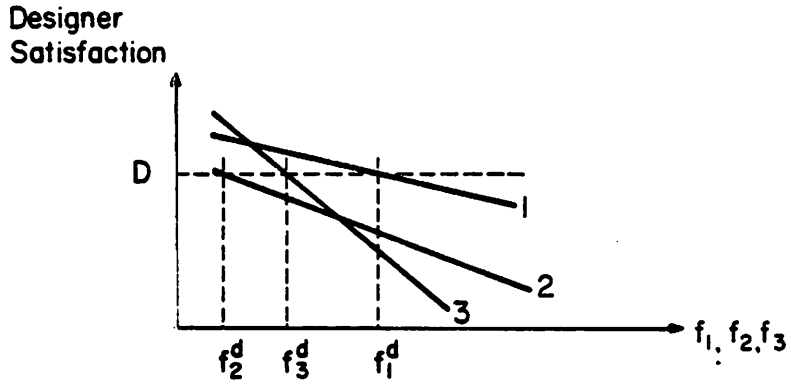


Figure 4.2. Objective Function Values at Particular Level of Designer Satisfaction

effect of the minimax optimization is to move that dotted line to the left. But here lies the culprit: the optimization is trying to decrease the three weighted functions equally even though, as shown, the designer is less satisfied with the value of $w_3^d f_3$ than with the value of $w_1^d f_1$; if the desired values cannot be reached, the resulting design is obviously not very satisfactory. As just stated, the optimization is not emphasizing the various objectives in a way that corresponds to the way the designer would like them to trade off. (Near the desired point $\mathbf{f}^d$, however, the optimization does emphasize according to the desired tradeoff.) The problem is that the functions have been normalized with only a *single* desired function value.

An alternative and quite natural way of indicating the relative importance of performance objectives and soft constraints is by having the designer specify two values for each: a *good* value and a *bad* value. The meaning of these values is limited to the following understanding, referred to as the *uniform*

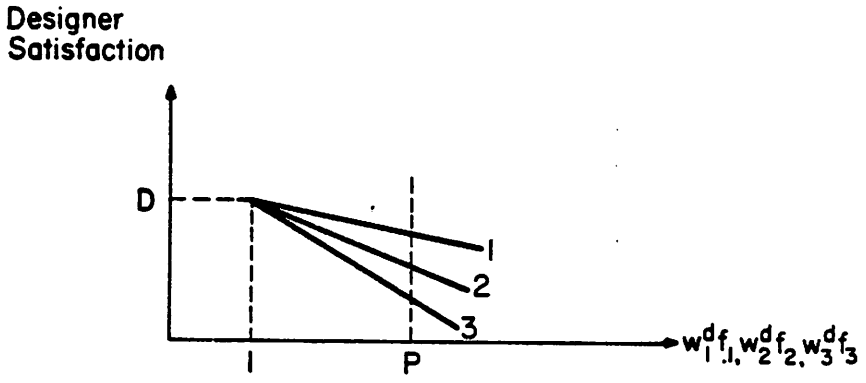


Figure 4.3. Designer Satisfaction Versus Weighted Objective Function Values.

satisfaction/dissatisfaction rule: having all of the various objectives and soft constraints achieve their corresponding good values should provide the same level of "satisfaction" to the designer for each, while achieving the bad values should provide the same level of "dissatisfaction". Furthermore, the good value of a soft constraint must be the value considered by the designer to be satisfactory, in the sense that an algorithm need not try to improve the constraint any further. The use of good and bad values in this way provides a very simple way to do tradeoff analyses: if a designer is unsatisfied with the performance level achieved by a particular objective or constraint, he simply changes what he considers to be satisfactory or unsatisfactory by adjusting the good or bad values and then resumes execution of the optimization. DELIGHT commands to modify these values are discussed later.

The good and bad values are used to produce the normalized objectives and constraints that are "seen" by the optimization algorithm. They generally have

values between 0 and 1 or nearby, where 0 and 1 correspond respectively to the designer's good and bad values, as seen in the transformation:

$$f_i' \triangleq \frac{f_i - good_i}{bad_i - good_i}. \tag{4.1}$$

In this formula, f is either an objective or a constraint function. A minimax related optimization, as described later in section 4.5.2.2, is then performed on the normalized function values. The above formula is similar to the normalization

$$f_i' \triangleq \frac{f_i - f_{SPEC_i}}{f_i^0 - f_{SPEC_i}}. \tag{4.2}$$

given by Hachtel et al. [59] in describing the methods used in the APLSTAP [60] design system. In their formula f_{SPEC_i}, the function value desired at the end of a completed sequence of LP steps, is analogous to the good value in equation (4.1). However, f_i^0, the objective function value prior to an LP optimization step, is used in place of our fixed (at least until manually changed) bad value in (4.1). Hence, they refer to this normalization as the "relative amount out of spec (compared with the original)". Since this has the same effect as stating the unlikely condition that all objective and constraint values prior to taking an optimization step are equally bad, we believe that the normalization in equation (4.2) violates the uniform satisfaction/dissatisfaction rule given above and that the design process is more correctly served by (4.1).

Figure 4.4 shows how the designer's satisfaction versus the normalized functions coincide for the normalization in equation (4.1); throughout an optimization run the objectives and constraints improve in a way that corresponds to the way in which he would like them to trade off. (This assumes that designer satisfaction is linear in the objective function values. This will be approximately true

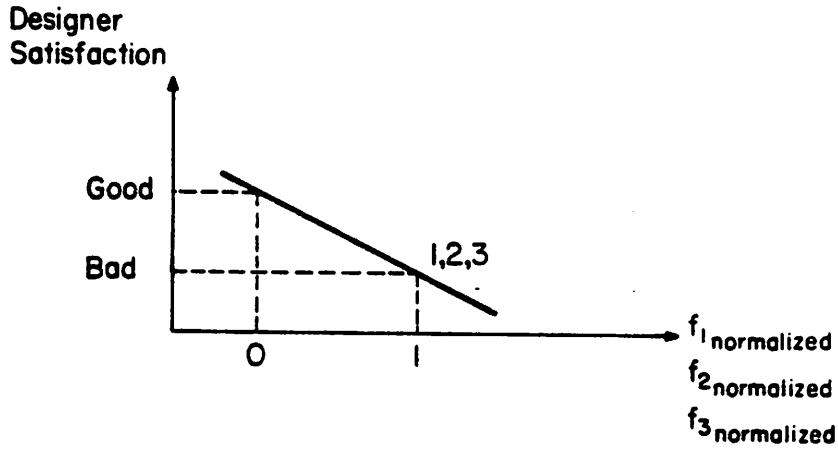


Figure 4.4. Designer Satisfaction Versus Good/Bad-Normalized Function Values.

in most cases if objectives are properly defined, e.g., by expressing them in db, etc.) Continual improvement is important for the following reason. In engineering design, it is usually too costly to carry an optimization to a truly optimum solution. What is desired is an effective first few optimization iterations rather than a completely convergent sequence of steps. Thus, the improvement achieved in the first few iterations must accurately reflect the designer's desires.

As mentioned earlier, in engineering design, many specifications must be met over the range of an independent parameter. These are represented as functional objectives and functional soft constraints for which good and bad *curves* are defined by the designer.

Intuitive Knowledge. One means of considering the third shortcoming and allowing designers to express their intuitive knowledge is to allow them to intelligently control parameter scaling. For example in electronic circuits, a

capacitance value might be 3.3pF. When scaled to units of picofarads (pF), the capacitance value becomes 3.3. This type of scaling is very important in achieving good performance of most optimization algorithms; speed of convergence can be enhanced dramatically with proper parameter scaling.

Parameter scaling can be handled either automatically or manually. Director, et al. [42] describes an automatic method of scaling in which one-dimensional searches along the coordinate axes in design parameter space are undertaken for each design parameter. Although many other automatic techniques are possible, for simplicity and to allow designer intuition, we choose to handle parameter scaling manually, though we don't preclude the possibility of using designer intuition along with some automatic technique.

Our parameter scaling is achieved by having the designer provide a *nominal variation* for each design parameter of his optimization problem. A designer estimates the nominal variation according to the following *uniform parameter influence rule*: a change in each parameter by its nominal variation should influence the objectives and constraints to roughly the same degree. For example, the frequency response of an amplifier might be influenced to as nearly the same degree by a 1pF change in a 5pF capacitor as by a 1k change in a 10k resistor. Technically, the nominal variations should be related to the size of the level sets of the Lagrangian of the constrained nonlinear programming problem—what we desire is that an optimization algorithm "see" as nearly an *isotropic* problem as possible [117]. Luenberger [95], in improving the performance of steepest descent by introducing scale factors on each variable, states this in another way by saying that one should strive to make the second derivatives (of the Lagrangian) with respect to each variable roughly the same. For an unconstrained optimization problem with a single quadratic cost function whose

level set major axes are in line with the coordinate axes, this scaling corresponds to dividing each design parameter by its major axis length, which transforms the level sets so that they are more "circular". In this context, these major axis lengths would be the nominal variations. Similarly, we use the estimated nominal variations to scale design parameters by dividing each by its corresponding variation. Note that this is a heuristic which tries to capture the intuition of the designer about his design problem. Even though the designer's intuition may fail to provide an effective scaling using this approach—especially for problems with many objectives—this scaling strategy is probably better than other strategies that use a rigid technique hidden from the designer.

For example, there is a popular notion that design parameters should be scaled to a value near unity (see, for example, page 279 of Gill et al. [51], page 23 of Hachtel et al. [60], page I.33 of Balling [18], page 797 of Agnew [12], or page 290 of Pierre et al. [119]). Since here each design parameter is to be scaled by dividing it by its nominal variation, this corresponds to the (possibly incorrect) idea that the nominal variation of each design parameter is simply the parameter value itself, i.e., that a change to either zero or double each initial parameter value would influence the objectives and constraints to the same degree. The nominal variation idea suggested here appears to be a more reasonable form of scaling.

Box Constraints. The ability to express easily box constraints on the design parameters is part of the problem description facilities described in section 4.4.2.2. The way these box constraints are handled is another extension of the Phase I - Phase II Method of Feasible Directions algorithm of Polak. They are handled directly, without any constrained-to-unconstrained transformations such as the change-of-variable trigonometric substitution suggested by Powell

[128] and many others, $x_1=\sin x_1'$, that eliminates the box constraint $-1 \le x_1 \le 1$. This substitution introduces nonlinear scaling and extra nonlinearities in all of the objective and constraint functions.

Multiobjective programming formulation. We are led to the following multiobjective programming formulation in which primed variables are normalized quantities that are defined just below:

Minimize over scaled design parameters in $\mathbf{x}'$ the objectives:

$$
\begin{aligned}
&M_1'(\mathbf{x}'),\\
&M_2'(\mathbf{x}'),\\
&\cdots\\
&M_{Nmulticost}'(\mathbf{x}'),
\end{aligned}
$$

$$
\begin{aligned}
&FM_1'(\mathbf{x}', W_1) && \forall\ W_1 \in FMInterval_1,\\
&FM_2'(\mathbf{x}', W_2) && \forall\ W_2 \in FMInterval_2,\\
&\cdots\\
&FM_{Nfmulticost}'(\mathbf{x}', W_{Nfmulticost}) && \forall\ W_{Nfmulticost} \in FMInterval_{Nfmulticost},
\end{aligned}
$$

Subject to the (soft or hard) ordinary inequality constraints:

$$
\begin{aligned}
&I_1'(\mathbf{x}') \le 0,\\
&I_2'(\mathbf{x}') \le 0,\\
&\cdots\\
&I_{Nineq}'(\mathbf{x}') \le 0,
\end{aligned}
$$

the (soft or hard) functional inequality constraints:

$$
\begin{aligned}
&FI_1'(\mathbf{x}', W_1) \le 0 && \forall\ W_1 \in FIInterval_1,\\
&FI_2'(\mathbf{x}', W_2) \le 0 && \forall\ W_2 \in FIInterval_2,\\
&\cdots\\
&FI_{Nfineq}'(\mathbf{x}', W_{Nfineq}) \le 0 && \forall\ W_{Nfineq} \in FIInterval_{Nfineq},
\end{aligned}
$$

and the (soft or hard) box constraints:

$$
\begin{aligned}
&x_1'_{\min} \le x_1' \le x_1'_{\max}\\
&x_2'_{\min} \le x_2' \le x_2'_{\max}\\
&\cdots\\
&x_{Nfineq}'_{\min} \le x_{Nfineq}' \le x_{Nfineq}'_{\max},
\end{aligned}
$$

where the following normalizations are used:

$$M_i'(\mathbf{x}') \triangleq \frac{M_i(\mathbf{x}') - MGood_i}{MBad_i - MGood_i},$$

$$FM_i'(\mathbf{x}', W_i) \triangleq \frac{FM_i(\mathbf{x}', W_i) - FMGood_i(W_i)}{FMBad_i(W_i) - FMGood_i(W_i)},$$

$$I_i'(\mathbf{x}') \triangleq \frac{I_i(\mathbf{x}') - IGood_i}{IBad_i - IGood_i},$$

$$FI_i'(\mathbf{x}', W_i) \triangleq \frac{FI_i(\mathbf{x}', W_i) - FIGood_i(W_i)}{FIBad_i(W_i) - FIGood_i(W_i)},$$

$$x_i' \triangleq x_i \,/\, NominalVariation_i\,,$$

$$x_{i_{min}}' \triangleq x_{i_{min}} \,/\, NominalVariation_i\,,$$

$$x_{i_{max}}' \triangleq x_{i_{max}} \,/\, NominalVariation_i\,.$$

In an actual optimization problem, any of the objectives could be maximized instead of minimized and any of the constraints could be "≥" instead of "≤". However, the formulation does not change since the ordering of the good and bad values for such cases causes the denominators in the above normalizations to change sign and automatically realize either "max f" to "min $-f$" or "$f \leq 0$" to "$-f \geq 0$" transformations. Note also that both hard and soft constraints are normalized with good and bad values or curves. However, hard constraints are treated differently by the optimization algorithm discussed later in section 4.5.2.2; they do not take part in any tradeoffs. For soft box constraints, the good values are used for the target constraint values $x_{i_{min}}$ and $x_{i_{max}}$. Also for these constraints, both the good and bad values are normalized by the nominal variations. How to specify the various quantities needed to formulate a design problem as the above mathematical programming problem is presented in section 4.4.2.2.

On the mathematical side, most DELIGHT optimization algorithms require that all objective and constraint functions, together with their first derivatives

with respect to the unknown vector $\mathbf{x}$ be continuous. This requirement is generally met if expressions used to define the objectives or constraints do not contain any *max*, *min*, *abs*, or references to other nondifferentiable functions provided by DELIGHT. Functional objectives or constraints must also be piecewise Lipschitz continuous in their second, "W", argument, with discontinuities only of the first kind [134]. Since this statement applies to the normalized functions that are "seen" by the algorithms, the corresponding functional good and bad curves, as a function of their "W" variable, need also be piecewise Lipshitz continuous. A benefit of this "mild" requirement is that piecewise constant "staircase" good and bad curves are allowed. These might be used, for example, as upper and lower bounding constraints on the step response of an engineering system.

It must be stressed here that interaction between the designer and the optimization process as well as human-engineered graphics to support this interaction are crucial for an efficient use of the mathematical formulation described here. Suitable information must be displayed to allow the designer to interactively modify good and bad values or curves. Such a man-machine interface has been implemented in the DELIGHT system and is described later in section 4.6.

4.4.2. DELIGHT Problem Description Facilities

4.4.2.1. Classical Problem Description

To show how to formulate a design problem as the mathematical programming problem of section 4.4.1.1, we first rewrite it here in an expanded format:

$$\begin{aligned}
\text{minimize:} \quad & cost(\mathbf{X}) \\
\mathbf{X} & \\
\text{subject to:} \quad & ineq_1(\mathbf{X}) \leq 0, \\
& ineq_2(\mathbf{X}) \leq 0, \\
& \cdots \\
& ineq_{Nineq}(\mathbf{X}) \leq 0, \\
& \\
& fineq_1(\mathbf{X},\omega) \leq 0 \quad \forall\, \omega \in [W_0, W_c], \\
& fineq_2(\mathbf{X},\omega) \leq 0 \quad \forall\, \omega \in [W_0, W_c], \\
& \cdots \\
& fineq_{Nfineq}(\mathbf{X},\omega) \leq 0 \quad \forall\, \omega \in [W_0, W_c].
\end{aligned}$$

In the above, all functional constraints are defined over the same interval, as required by the classical formulation of section 4.4.1.1, although this limitation has been corrected in the new multiobjective formulation. To use the above formulation, a user describes the problem in a set of files which he creates using a text editor. For example, assume a user has given the name *pbname* to his optimization problem. The table below shows the problem description filenames used in setting up the design problem. Also shown, for files which must contain RATTLE functions, are the RATTLE function names which have to be used for each of the functions in the above mathematical programming problem[20]:

Problem Description Files		
Filename	Function Defined	RATTLE Function-Name
pbnameS	-	-
pbnameP	-	-
pbnameC	cost	cost
pbnameI	**ineq**	ineq
pbnameFI	**fineq**	fineq

The filenames that must be used are intentionally designed with suffixes to

[20] Fixed problem function names are common in optimization systems. The LPNLP Fortran routines for solving constrained nonlinear programs via augmented Lagrangians is such an example and page 269 of [119] shows a table similar to the one given here.

show their type so that the files will all be adjacent in a sorted list of all of user files. Thus, the coexistence of several different design problem files in the same directory is made easier.

The "S", "P", and "C" files are essential; the "I" and "FI" files are optional. For example, if there are no inequality constraints then there need not be any "I" file.

Next we show exactly how to set up these files. At the end of this section, a complete set of input files for a simple example is given which illustrates the problem description facilities presented in this section.

"S" File. The "S" problem file is the setup file. It contains RATTLE assignment statements that set the values of certain global optimization variables as shown in the following table:

Assignment Statement Used	Variable is the Number of:
Nparam = value	Design parameters.
Neq = value	Equality constraints.
Nineq = value	Ordinary inequality constraints.
Nfineq = value	Functional inequality constraints.

Only the nonzero assignments shown above need be included in the "S" file, i.e., these variables all default to zero.

Optionally, design parameters may be given names in the "S" file. This allows the elements of the optimization unknown vector **X** to be used by name instead of by, e.g., *X(3)*, in any RATTLE expression, particularly those expressions used in describing the problem being set up. To give design parameters names, a

design_parameter statement must be in the "S" file for each design parameter. This statement has the format:

```
design_parameter PAR_NAME
```

where PAR_NAME is any unique name a user wishes to give to a design parameter[21].

"P" File. The "P" file is used to provide initial values (guesses) for the design parameters to be used in an optimization. It contains either assignment statements such as *X(3) = 5.6k* for each of the *Nparam* components of design parameter vector **X**, or, if the user has given names to his design parameters using *design_parameter* as explained above, *set* commands which set the initial values of the parameters by name, as in *set tension = 5.6k*. The *set* command is discussed further in section 4.7.3.

"C" File. The "C" file is used to define the cost function to be minimized. It defines the RATTLE function *cost* and has the following structural format:

```
function cost (x)  {
   array x(Nparam)
   return ( EXPRESSION_FOR_cost )
   }
```

The expression on the *return* statement is usually the actual cost function to minimize and is an arbitrary RATTLE expression involving the design parameters. They may appear in the expression either explicitly, or implicitly through their use by other RATTLE functions that do appear in the cost expression. Of course, other RATTLE statements of any kind that aid in the computation of the

[21] This name must obey the following rule: it must begin with a letter and be followed by letters, digits, or underscores. This is the same rule that applies to RATTLE variable names.

cost function may come before the *return* statement.

"I" File. The "I" file is used to define the optimization problem ordinary inequality constraints. It defines the RATTLE function *ineq* and may have the following structural format:

```
function ineq (j, x)  {
   array x(Nparam)
   if      (j=1)       return ( EXPRESSION_FOR_ineq1 )
   else if (j=2)       return ( EXPRESSION_FOR_ineq2 )
    . . .
   else if (j=Nineq)   return ( EXPRESSION_FOR_ineqNineq )
   }
```

The argument *j* selects one of the *Nineq* functions of **ineq**; the *if/else if* sequence of statements selects one of the inequality constraint expressions to return. These expressions are a function of the unknown vector **X**

"FI" File. The "FI" file is used to define the functional inequality constraints to be satisfied for every value of w in the interval $[W_0, W_c]$. After assigning values to the end points W_0 and W_c using, for example, *W0 = -55* and *Wc = 125*, it defines the RATTLE function *fineq* using the following structural format:

```
function fineq (j, x, w)  {
   array x(Nparam)
   if      (j=1)       return ( EXPRESSION_FOR_fineq1_VS_w )
   else if (j=2)       return ( EXPRESSION_FOR_fineq2_VS_w )
    . . .
   else if (j=Nfineq)  return ( EXPRESSION_FOR_fineqNfineq_VS_w )
   }
```

The argument *j* selects one of the *Nfineq* functions of **fineq**; the *if/else if* sequence of statements selects one of the functional inequality constraint expressions to return. These expressions are a function of the unknown vector **X**

and the independent variable for functional constraints, w.

Gradients. The files above that define problem description RATTLE functions may optionally contain RATTLE procedures for evaluating their gradients. Note, however, that if an optimization algorithm needs gradients and procedures for these are not provided, they will be automatically computed using finite differences. For this computation the following formulas are used (shown here for a scalar function f of a scalar variable x):

$$\delta = x \times Fd\_rel\_ + Fd\_abs\_$$

$$\partial f / \partial x = \frac{f(x+\delta) - f(x)}{\delta}$$

where *Fd_rel_* and *Fd_abs_* are user-settable variables which have default values of 10^{-4} and 10^{-7}, respectively.

For function *cost* in the "C" file, the following structural format is used to define procedure *gradcost* for computing the gradient vector of the cost:

```
procedure gradcost (x,g)  {
   array x(Nparam), g(Nparam)
   g(1) = EXPRESSION_FOR_∂cost/∂x1
   g(2) = EXPRESSION_FOR_∂cost/∂x2
    . . .
   g(Nparam) = EXPRESSION_FOR_∂cost/∂xNparam
   }
```

The structure of the other gradient procedures, *gradineq*, and *gradfineq*, is similar and can be seen in the example below. DELIGHT provides the command *testgrad* for determining whether the gradient procedure is correct by comparing what it calculates with a finite-difference calculation.

A Complete Example. We use a modified version of test problem number 24

from Himmelblau [64] as a simple example to illustrate the problem description files that need to be set up. One of Himmelblau's original ordinary inequality constraints has been written as a functional inequality constraint. The modified problem then is:

$$\underset{(x_1,x_2)^T}{\text{Minimize:}} \quad (x_1-2)^2 + (x_2-1)^2$$

$$\text{subject to:} \quad x_1^2 \leq x_2$$

$$\text{and:} \quad -\left(\omega - \frac{x_1}{x_1+x_2}\right)^2 + x_1 + x_2 \leq 2 \quad \forall \omega \in [0,1] .$$

Let us name this problem *H24*. Then, to describe this optimization problem, the following files are set up. For the following, recall that in RATTLE, anything following a "#" character up to the end of the line is considered a comment:

File H24S:

```
## H24S - Setup file for problem H24.

Nparam = 2
Nineq = 1
Nfineq = 1
```

File H24P:

```
## H24P - Initial guess for problem H24.

X(1) = 0.1
X(2) = 0.1
```

File H24C:

```
## H24C - Cost for problem H24.

function cost (x)  {
   array x(2)
   return ( (x(1)-2)**2 + (x(2)-1)**2 )
   }

procedure gradcost (x,g)  {
   array x(2), g(2)
   g(1) = 2 * (x(1)-2)
   g(2) = 2 * (x(2)-1)
   }
```

File H24I:

```
## H24I - Inequality constraints for problem H24.

function ineq(j,x)  {
   array x(2)
   if (j==1)  return ( x(1)**2 - x(2) )
   }

procedure gradineq(j,x,g)  {
   array g(2), x(2)
   if (j==1)  {
      g(1) = 2 * x(1)
      g(2) = -1
      }
   }
```

File H24FI:

```
## H24FI - Functional inequality constraints for problem H24.

Wc = 1
W0 = 0

function fineq (j,x,w)  {
   array x(2)
   if (j==1)
      return (-(w - (x(1)/(x(1)+x(2))))**2 + x(1) + x(2) - 2)
   }

procedure gradfineq (j,x,w,g)  {
   array x(2), g(2)
   if (j==1)  {
      g(1) = -2 * (w-x(1)/(x(1)+x(2))) * (-x(2))/(x(1)+x(2))**2 + 1
      g(2) = -2 * (w-x(1)/(x(1)+x(2))) *   x(1) /(x(1)+x(2))**2 + 1
      }
   }
```

Note that since *Nineq* and *Nfineq* are both one, the if-statements above are not really needed. How to have DELIGHT process these files is shown later when the *solve* command is introduced in section 4.5.4.1.

This problem description would be slightly simpler if the design parameters had been given names. To show this, the "S", "P", and "C" files are repeated below with this enhancement, assuming the names are *Length* and *Width*:

File H24S:

```
## H24S - Setup file for problem H24 with named parameters.

design_parameter Length
design_parameter Width
Nparam = 2
Nineq = 1
Nfineq = 1
W0 = 0
Wc = 1
```

File H24P:

```
## H24P - Initial guess for problem H24 with named parameters.

set Length = 0.1
set Width  = 0.1
```

File H24C:

```
## H24C - Cost for problem H24 with named parameters.

function cost (x)  {
   array x(2)
   return ( (Length-2)**2 + (Width-1)**2 )
   }

procedure gradcost (x,g)  {
   array x(2), g(2)
   g(1) = 2 * (Length-2)
   g(2) = 2 * (Width-1)
   }
```

4.4.2.2. Multiple Objective, Engineering-Oriented Problem Description

Before showing how to set up the required input files, let us review the steps necessary to formulate an engineering design problem using the new multiobjective programming formulation of section 4.4.1.2.

Starting with an initial set of goals or specifications, the designer first chooses an initial structure that has the potential of yielding acceptable performance. After preliminary selection of a set of design parameters, it is then necessary to translate the given goals into the multiobjective programming formulation. In general, this consists of three steps.

First, the designer must decide whether each performance goal is to be considered as an objective or a constraint. Both of these require the specification of desired values, e.g. good and bad values. However, objectives continue to improve throughout the optimization whereas constraints "stop being pushed" after they achieve their desired values (their good values).

The second step is to decide whether each constraint is to be considered *hard* or *soft*. As explained in section 4.4.1.2 and by Nye and Tits in [112, 117] ([117] is included as appendix F of this dissertation), hard constraints are ones that the designer wishes the algorithm to initially give the greatest emphasis and which, once satisfied, the designer wishes to remain satisfied and not take part in any subsequent design tradeoffs. Soft constraints, on the other hand, are those which the designer is interested in conveniently trading off against one another and against the performance objectives during an optimization run.

The last step in setting up the problem formulation is to indicate the relative importance of the objectives and constraints by providing a *good* and a *bad* value for each. These values are set according to the uniform satisfaction/dissatisfaction rule of section 4.4.1.2. The precise values specified need only be "ball-park" values since DELIGHT contains commands for modifying them during an optimization run.

As before, the design problem is described in a set of files. We shall see that the syntax shown in this section is more user-oriented by being less cumbersome than that of the previous section. Assuming the same design problem name, *pbname*, the table below shows the problem description filenames and RATTLE functions used in setting up the design problem:

Problem Description Files		
Filename	Function Defined	RATTLE Function-Name
pbnameS	-	-
pbnameP	-	-
pbnameM	**M**	multicost
pbnameFM	**FM**	fmulticost
pbnameI	**I**	ineq
pbnameFI	**FI**	fineq

The "S" and "P" files are the only problem description files which are essential; the other files listed above are optional.

Next we show exactly how to set up these files. As before, at the end of this section, an example set of input files is given. We emphasize that all numeric values requested in the following file formats are "raw" values, i.e., are not scaled in any way.

"S" File. The contents of the "S" file are identical to that of the formulation of the previous section except for an enhanced *design_parameter* statement and a few different global variables, shown in the following table:

Assignment Statement Used	Variable is the Number of:
Nparam = value	Design parameters.
Nmulticost = value	Ordinary performance objectives.
Nfmulticost = value	Functional performance objectives.
Nineq = value	Ordinary inequality constraints.
Nfineq = value	Functional inequality constraints.

In the new problem formulation, all design parameters must be declared,

each with a *design_parameter* statement. This is due to the additional information to be provided for each parameter as well as for enhanced readability. This statement has the format (curly brackets are used to indicate a required choice among different options listed vertically):

```
design_parameter PAR_NAME variation=V [ min={  MIN   } max={  MAX   } ]
                                        {(VB,VG)}       {(VG,VB)}
```

where each of the V's, VB's, VG's, and MIN, and MAX, are any numeric value (variations, of course, should be positive).

The optional *min* and *max* arguments create box constraints on the design parameter. Following either by a single value creates a hard box constraint while following by a parenthesized pair of values creates a soft box constraint. For a soft constraint, VB is the bad value while VG is the good value. Note that *Nineq*, the number of ordinary inequality constraints, does not in any way include design parameter box constraints.

"P" File. The contents of the "P" file are identical to what they were before except only *set* commands are used to set the initial guess to be used for optimization.

"M" File. The "M" file is used to define multiple performance objectives for the design problem. For each objective the user provides a name, whether to minimize or maximize, the actual expression to be optimized, the good and bad values, and any other RATTLE statements which are necessary in the computation of the objective expression. These optional RATTLE statements are introduced by the optional keyword *using*. If more than one statement is desired, then *using* must be followed by an opening curly bracket "{" and the group of statements must end in a closing curly bracket "}". For an example of such a

group of statements, see constraint 1 in the "FI" file example at the end of this section.

The structural format of the "M" file is:

```
prob_function multicost
   objective 1 ...
   objective 2 ...
   objective 3 ...
    . . .
end_multicost
```

where each *objective* line has the format:

```
objective n 'LABEL' {minimize} O_EXPR good=G bad=B [using
                    {maximize}
   RATTLE statement]
```

O_EXPR, the actual objective expression to optimize, is an arbitrary RATTLE expression of the design parameters. G and B are the good and bad values, respectively, for the performance objective. The objective is given a label that is used in certain optimization output procedures to ease identification of the objectives.

"FM" File. The "FM" contains *functional* performance objectives, that is, objectives which must be satisfied for every value of a "W" variable. The "FM" file is identical to the "M" file in structure and format except for two differences. First, a line which specifies the interval range over which the constraint is to be satisfied must be given. For readability, each objective is followed by a *for_every* line of the form:

```
for_every W from FROM_VAL to TO_VAL initially {by   } INC_VAL
                                              {times}
                                              {oct  }
                                              {dec  }
                                              {log  }
```

which is explained shortly. Second, the good and bad specifications are actually

curves which may be given by arbitrary expressions involving the "W" variable. In the example "FM" file given later, the good value for objective 1 has been specified as such an expression.

The *for_every* line tells the algorithm how many samples (and how they are spaced) to use initially in sampling the functional objective over the "W" variable to find its maximum (or minimum). The *by* keyword causes the usual additive increment of the "W" variable; this results in linearly-spaced samples. The *times* keyword causes the "W" variable to be increased by multiplying it by INC_VAL. The *oct* and *dec* keywords cause the "W" variable to be increased (or decreased) by multiplying it by a computed variable to give INC_VAL points per octave and per decade, respectively. (In these cases as well as the next, "INC_VAL" is improperly named.) For the *log* keyword the samples are logarithmically spaced (as for *times*, *oct*, and *dec*) and the value of INC_VAL is the total number of sample points. A functional objective of a system response that usually is plotted on a logarithmic x-axis, for example, would usually require logarithmic sample spacing. The purpose of the keyword *initially* is to stress the fact that the *for_every* line specifies only an initial set of sample points; as explained later, the optimization algorithm increases the number of samples automatically as an optimal point is approached. The various capitalized arguments, FROM_VAL, TO_VAL, and INC_VAL, may be arbitrary numeric values.

"I" File. The design problem inequality constraint specifications are given in the "I" file. For each constraint the designer provides a name, whether the constraint is "greater than" or "less than", the actual expression to be constrained, the good and bad values, the keyword *soft* if the constraint is desired to be soft, and any other RATTLE statements which are necessary in the computation of the constraint expression. Just as for "M" files, the optional RATTLE statements are

introduced by the optional keyword *using*. As seen below, the syntax here is more human-engineered since it eliminates the cumbersome *if/else if* statements of our earlier problem description.

The structural format of the "I" file is:

```
prob_function ineq
   constraint 1 ...
   constraint 2 ...
   constraint 3 ...
   ...
end_ineq
```

where each *constraint* line has the format:

```
constraint n 'LABEL' C_EXPR {<=} good=G bad=B [soft] [using
                            {>=}
   RATTLE statement]
```

C_EXPR, the actual expression to constrain, is an arbitrary RATTLE expression of the design parameters. One of the operators "<=" or ">=" must be chosen to indicate whether C_EXPR is to be less than or greater than the good value. G and B are the good and bad values, respectively, for the constraint specification. The optional keyword *soft* indicates that the designer considers this to be a soft constraint. Note that for scaling purposes, good and bad values should be specified for hard constraints as well as soft constraints even though the satisfaction of the former is considered essential to the design and they do not take part in any tradeoff exploration.

"FI" File. The "FI" file is for functional constraints and differs from the "I" file in the same way the "FM" file differed from the "M" file: each constraint is followed by a *for_every* clause which includes a specification of the initial number of samples for discretization.

A Complete Example. We recast the Himmelblau test problem of the previous

section into the new problem formulation to illustrate the problem description files that need to be set up. Again, the problem name is *H24*. To describe this optimization problem, the following files are set up. For simplicity, we show them without the gradient procedures. In the "P" file for this example, the choices for design parameter variations and soft box constraints are, for the most part, *ad hoc*, based on the known solution of this problem, $(1,1)^T$. Notice the backslash ("\") in file *H24M* below. In this dissertation a backslash escape character ("\") at the end of a statement or format line means that the line is continued onto the next; when the statement is actually used, both lines or all the arguments, respectively, must be typed on the same line.

File H24S:

```
## H24S - Setup file for problem H24.

design_parameter  Length  variation=0.2  min=(.1,.2)  max(3,4)
design_parameter  Width   variation=0.4  min=.1       max=5.5
Nparam = 2
Nineq = 1
Nfineq = 1
```

File H24P:

```
## H24P - Initial guess for problem H24.

set Length = 0.1
set Width  = 0.1
```

File H24M:

```
## H24M - Multiple objectives for problem H24.

prob_function multicost

   objective 1 'Label-M1' minimize (Length-2)**2+(Width-1)**2   \
                          good=1.5 bad=3.5

end_multicost
```

File H24I:

```
## H24I - Inequality constraints for problem H24.

prob_function ineq

   constraint 1 'Label-I1' Length**2-Width <= good=0 bad=.2 soft

end_ineq
```

File H24FI:

```
## H24FI - Functional inequality constraints for problem H24.

prob_function fineq

   constraint 1 'Lab-FI1' RetVal <= good=Force/1k bad=10 using {
      term1 = Length / (Length+Width)
      term2 = -(Force - term1)**2
      RetVal = term2 + Length + Width - 2
      }
   for_every Force from 2.5k to 5.5k initially by .5k

end_fineq
```

The following points should be noted:

(1) The objective and constraint syntax of this section has eliminated the *if/else if* statements of the last section. In general, the syntax given in this section is more human-engineered than that of the last. As another example, the *prob_function* declarations relieve the user of having to type the procedure argument lists and array declarations for x shown in the previous section.

(2) Each functional objective and constraint may not only have a different interval for its independent variable, but the interval sampling may be independently specified to be either linear or logarithmic with various spacing possibilities. In the previous section, all functional constraints had the same interval, $[W_0, W_c]$, with linear spacing.

(3) In the "S" file, the upper and lower box constraints for design parameter *Length* are both soft; for *Width* they are both hard. However, for a particular design parameter, soft and hard constraints may be mixed.

(4) The one inequality constraint is soft due to the keyword ***soft*** appearing on the *constraint* line, whereas the one functional constraint is hard due to the absence of the keyword *soft*.

One final note concerns gradient computation. Just as for the previous problem.description, if an optimization algorithm needs gradients and procedures for these are not provided, they will be automatically computed using finite differences. However, for the multiobjective problem formulation, the perturbation used is based on the nominal variation and is given by

$$\delta = \mathit{NominalVariation} \times \mathit{Fd\_rel\_}.$$

4.5. Optimization Algorithms and Problem/Algorithm Interfaces

We have seen two mathematical programming formulations and the associated problem description facilities of DELIGHT. We now turn to the algorithms side of the coin. We start in section 4.5.1 with an introductory description of a library of optimization algorithms, called the *RATTLE Optimization Algorithms Library*. This includes as an example in section 4.5.1.2, the structure of a simple algorithm "outer loop" and various extensions, most notably, enhancements for interaction such as the insertion of a test to catch soft interrupts (see section 4.2.6). Then in section 4.5.1.3 we use this basic algorithm structure to describe a class of algorithms known as *methods of feasible directions*. Section 4.5.2 extends the basic feasible directions algorithm by presenting a practical algorithm suitable for engineering design problems. This algorithm is based on the Phase I - Phase II Method of Feasible Directions algorithm of Polak et al. [123]. We first discuss generally why these methods are good for engineering design and point out some shortcomings before presenting the new algorithm. The new

algorithm is not only more advanced in terms of efficiency and the class of problems it can handle, but also in terms of *ease of use*. Next in section 4.5.3 we return to the details of the Algorithms Library. This includes a listing of the entries and how algorithm sub-blocks are substituted. In section 4.5.4 we arrive at how the optimization problem set up and the optimization algorithm chosen are coupled. This involves, in particular, the *solve* command. Two other features of the *problem interface* are also presented. Finally in section 4.5.5 we present commands for actually starting an optimization run in DELIGHT.

Before proceeding, let us step back and view the problem/algorithm interface in the context of four major interfaces provided by DELIGHT. These are shown in the following table along with the section of the dissertation where they are discussed:

DELIGHT Interfaces				
1	User	⇒	Problem formulation	4.4
2	Problem	⇔	Algorithm	4.5.4
3	Design performance Algorithm performance	⇒ ⇒	User User	4.6.1 4.6.2
4	Problem	⇔	Simulation program	4.7

The four interfaces are shown pictorially in figure 4.5. The numbered semi- and bi-directional arrows show what two entities are related by each interface. The first is the problem description facilities already discussed in sections 4.4.2.1 and 4.4.2.2. The second involves the coupling between a problem that has been set up and a chosen algorithm via the *solve* command. It also consists of a special software interface between algorithms and problem description

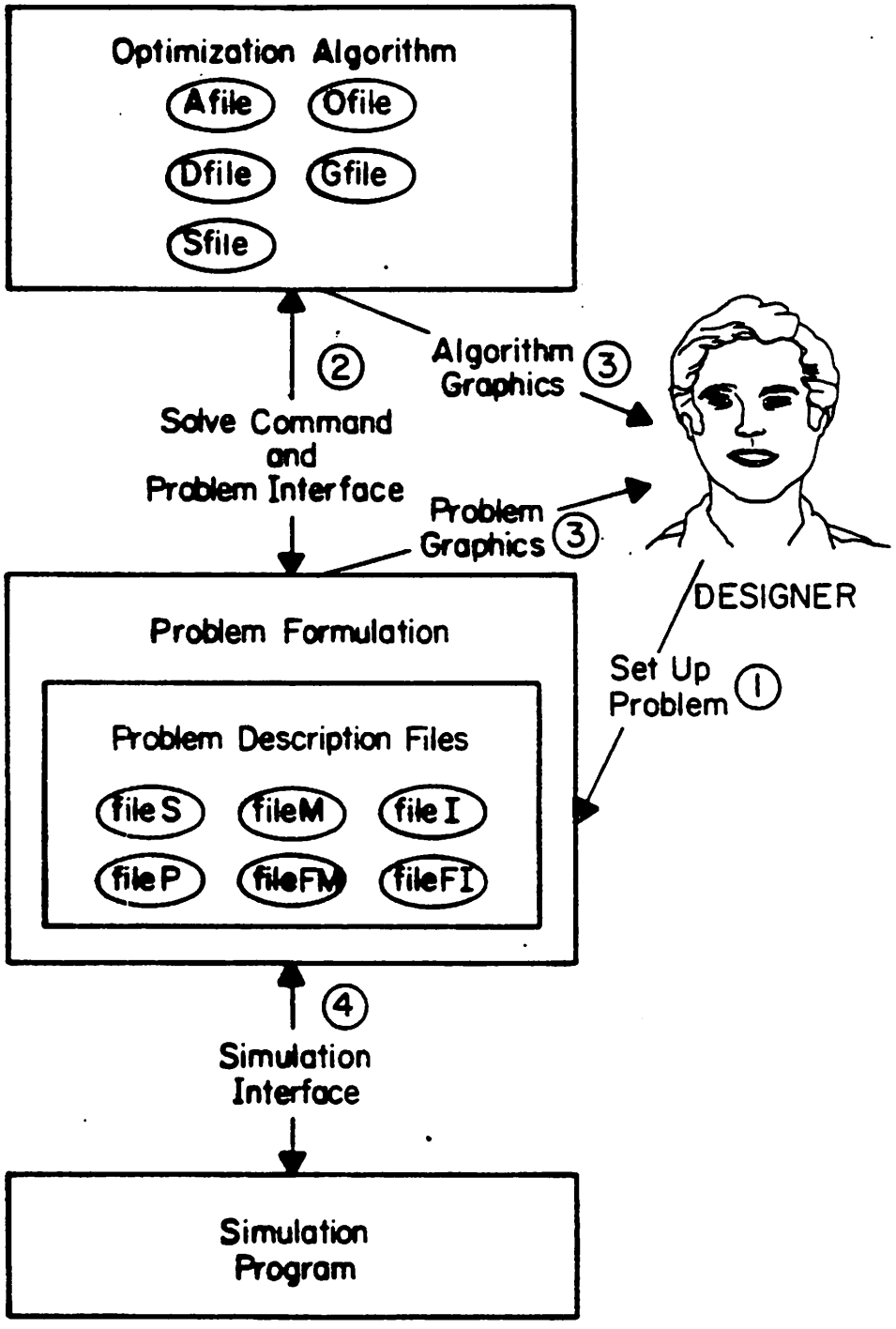


Figure 4.5. Four Interfaces Provided by DELIGHT.

functions whose threefold purpose is given in section 4.5.4.2. The third interface is related to the graphical and non-graphical means used by DELIGHT to keep the user informed about how his problem or the algorithm is performing. The final interface involves versions of DELIGHT that have been coupled to a particular simulation program. In a problem description one can easily set input parameters and retrieve output responses from a simulation program. Related to this are the DELIGHT features (for systems personnel) for actually creating the application-specific versions of the system. These features are discussed in appendix B.

4.5.1. Introduction to the RATTLE Optimization Algorithms Library

4.5.1.1. Purpose and Structure

Before exploring the RATTLE Optimization Algorithms Library in detail, a brief exposition of several issues involved in program libraries is presented[22]. A *program library* should not be simply a collection of programs from varied sources, written in isolation, that are grouped together in one place with uniform documentation, but rather, a set of routines that are conceived and written within a unified framework, to be available to a general community of users. Today there is wide recognition of the value of program libraries designed to solve useful problems since, in the particular case of DELIGHT, the latest developments in mathematical optimization techniques can be made available to many users.

The demands upon a library such as the RATTLE Optimization Algorithms Library vary from user to user according to their differing levels of interest and

[22] A more complete discussion of this topic is given by Gill et al. in [51] in discussing their Fortran program library for optimization; some of this material has been adapted from there.

expertise[23]. For example, a user who is familiar with the details of an algorithm may wish that its implementation contain sufficient flexibility so that it can be tuned to his particular problem, while another user may simply wish to use an algorithm without being required to learn any of its details. In the RATTLE Library, the *parameter* statements to be introduced in the next section for user-settable algorithm parameters partially meet this goal since, to enhance speed of convergence, the algorithm parameters declared can be modified by the user at any time, or simply left at their default values. The resemblance of many of the library entries to their mathematical descriptions also helps meet this goal since an algorithms expert can simply modify an existing sub-block and substitute it into his algorithm for the one from the library.

The structure of a library and the coding of its entries should reflect as much as possible the ideas of good design that are often called "structured programming" [37] —here, of course, the structured nature of RATTLE assists in the coding task— and "program development by stepwise refinement" [157]. Using these approaches, an abstract algorithm is decomposed into finer and finer steps until the algorithm consists of a highly structured combination of concise and well-defined sub-blocks. In our case, the RATTLE implementation is then designed to display the same structure. Additionally, the language extensibility features of DELIGHT discussed in sections 4.2.8 help meet the DELIGHT design goal of allowing algorithms to look as much like their mathematical descriptions as possible.

One of the benefits of these approaches is that it is possible to identify closely related or identical sub-blocks that occur in several algorithms and thus

[23] Related to this is the discussion in section 3.3 of the needs of the various classes of users supported by DELIGHT.

use the same procedure as sub-steps in different algorithms. An additional advantage is that it may reveal structural similarities in algorithms that were initially considered to be quite different because their inherent nature was hidden by notation or computational detail. Thus, the creator's as well as the expert user's understanding of the library entries is improved.

Getting back to the RATTLE Optimization Algorithms Library, we note that besides the above general ideals, the Library takes on the additional and important goal of allowing DELIGHT users to build up many variations of optimization algorithms by combining either existing blocks contained in the library or new blocks that they create. To meet this goal, the Library contains well-structured and tested RATTLE implementations of sub-blocks of many early and recent major optimization algorithms. (For an excellent introduction to many of the classical optimization techniques contained in the Library, see Luenberger [95] or Polak [121].) This integrated set of procedures may be combined to create algorithms for unconstrained and constrained, both ordinary and semi-infinite, optimization problems. (The Library does not classify problem objectives and constraints into categories—e.g., linear, sum of squares, quadratic, nonlinear, sparse, sparse linear—to the extent of Gill et al. in [51] but instead distinguishes unconstrained from constrained problems and equality, inequality, functional inequality, box, and singular value constraints.) Following the program library ideals presented above, the Library is organized to exploit to the utmost the natural modularity of most optimization algorithms which, in the simplest case, can be assembled from such blocks as (search-) *direction* and *stepsize* sub-blocks. Just how this assembly is accomplished is shown in the next section.

Carrying the decomposition to finer levels, search-direction sub-blocks can be constructed from sub-procedures which determine the gradients to be used

for direction construction and from linear or quadratic programming algorithms. Similarly, step-size sub-blocks can be built up from constrained and unconstrained step-size blocks. Other Library sub-blocks include outer approximations procedures, adaptive parameter adjustment procedures, and procedures for estimating Lagrange multipliers. Not mentioned yet are several "main" procedure blocks that implement the outer loop required to combine the first level sub-blocks into a complete algorithm. Many of these procedures use the basic outer loop structure shown next.

4.5.1.2. Structure of Optimization Algorithms

The general structure of a typical optimization algorithm "outer loop" in DELIGHT is as follows:

Algorithm Structure (4.3)

```
repeat  {
    compute h = direction (X)
    compute λ = stepsize (X,h)
    update  X = X + λ*h
    }
until ( convergence criterion is met )
```

Starting with an initial guess $\mathbf{X} = \mathbf{X}_0$, the *repeat* loop above is entered. First, a search direction $\mathbf{h} \in \mathbb{R}^{Nparam}$ is computed depending on the current value of $\mathbf{X}$. Then, a step-size to move $\mathbf{X}$ along that search direction is computed. Finally, $\mathbf{X}$ is updated with the step-size and search direction computed. All of this is repeated until some convergence criterion is met. Since DELIGHT is an interactive program, however, very often algorithms in DELIGHT do not have any convergence criterion; the loop above is repeated forever. The user simply interrupts RATTLE execution when he is satisfied with the performance of his

problem. Interrupts for this purpose are discussed below.

The algorithm structure shown above is only the skeleton of an actual DELIGHT algorithm. In order to meet several of the DELIGHT goals presented in chapter 3, we make four additions to the above structure.

Addition 1. First of all, there needs to be a point during each pass through the loop at which output is presented to the user. This output is to show both problem performance and algorithm performance. For the former, it usually prints the current values of the design parameters and then executes a special procedure into which a user can put special problem-dependent output. For the latter, important algorithm parameters are printed.

Addition 2. In order to allow a user to interrupt an optimization run at a "major stopping point", i.e., have the algorithm suspend execution after completing the current iteration, there needs to be a test for a soft interrupt. (Recall that a soft interrupt is generated when the user presses the special interrupt key on his terminal just once.) As explained and demonstrated previously in section 4.2.6, this is done by testing the keyword *interrupt* in an if-statement. This addition and the previous one have been combined into a single *multi-line define* called *interaction*. Thus, this define takes the following form[24]:

```
define interaction
   output
   if (interrupt)
      suspend
   end
```

This define shows that after a soft interrupt, execution suspends just *after* the normal algorithm output prints.

---

[24] The *interaction* statement actually has a few other functions, for example, to interrupt execution after the specified number of iterations has occurred following, e.g., the command *run 5*.

Another consideration regarding *interaction* is where to place it in algorithm (4.3) above. The effect of placing it just after the *repeat* or just after the *update* is the same except that the former allows the initial guess to be observed before the first iteration occurs. Another possibility is to place it just after the *direction* computation. If the search direction is printed in the algorithm *output*, this position has the advantage that the direction printed corresponds to the present value of **X**. Thus, after a soft interrupt, the user could observe a graphical indicator of the quality of the search direction and resume execution if he approved. A disadvantage is that it would require the additional time needed for the next direction computation before the output of the present iteration could be observed. The chosen position for many entries in the RATTLE Algorithms Library is just after the *repeat* line.

Addition 3. This addition to algorithm (4.3) involves the use of *array sequences* for **X**. Often part of the sequence of unknown vectors $\{\mathbf{X}_{Iter}\}$, $Iter = 0, 1, 2, \cdots$, where *Iter* is the current optimization iteration number, is needed for display or analysis by the user or even for use by the algorithm itself. For example, the DELIGHT user may wish to plot a particular component of **X** over the last few optimization iterations in order to determine if it is "bouncing" back and forth between its box constraints. Thus, the last few elements of the sequence $\{\mathbf{X}_{Iter}\}$ need to be saved. This is accomplished by declaring **X** as an *array sequence* instead of as an ordinary *array*.

An *array sequence* is a data structure which creates a sequence of arrays that may be accessed with a sequence index as well as normal array indices. For example, the declaration *array_sequence M[10](3,4)* creates 10 copies of the

3×4 array M[25]. An array sequence is accessed by enclosing the sequence index in square brackets and the ordinary subscripts in normal parenthesis. In this example, we could access M using, for example, $M[1](2,2)$ or $M[i+j/2](k+1,k+2)$. When accessing an array sequence the index in square brackets may take on any integer value; the actual array subscript is computed modulo the sequence size. Thus, in the above example, the last 10 values M_{i-9}, M_{i-8}, $\cdots$, M_i of the sequence of 3×4 arrays M are stored. Note that array sequences declared outside of procedure bodies are automatically global (known inside all procedure bodies without being imported; see section 4.2.5).

Addition 4. The last addition to the basic algorithm structure in (4.3) is to set up a standard means of introducing user-settable algorithm parameters. Such parameters do not *have* to be declared in the following way but if they are, they and the information about them are stored and can be requested by algorithm users. This addition is achieved by having each algorithm parameter declared with the *parameter* statement; it has the following syntax:

```
parameter PAR_NAME = DEFAULT_VALUE 'EXPLANATION STRING'
```

For each algorithm parameter, a *parameter* statement is placed above the algorithm procedure definition (in the same file that contains the procedure). For example, above the procedure *stepsize* that defines the Armijo step-size subblock [16] are the two declarations:

```
parameter Alpha = .5  'Slope of Armijo test line'
parameter Beta  = .5  'Trial step-size along h is Beta**k'
```

Algorithm parameters become ordinary variables in the *pool* of nonlocal variables (see section 4.2.4) and are given default values at the time the algorithm is

[25] The array actually declared in this example is *_M_(3,4,10)*; *M* becomes a macro which pushes back an access to the actual array *_M_*. Of course, all of this is transparent to the user.

compiled (see the *solve* command in section 4.5.4.1) since the *parameter* macro simply pushes back an assignment statement. Thus, these parameters need to be imported using *import* statements (see section 4.2.5). Other duties of the *parameter* macro are (1) to print the name, default value, and explanation string on the screen when the algorithm is compiled and (2) to store the explanation strings. A user can give the command *display_parameters* to cause the parameter names, present values, and explanation strings to be printed, allowing him to see at any time what algorithm parameters he has control over.

Before presenting the algorithm structure with the above four additions, let us note the following. By convention, all optimization algorithms in DELIGHT are simply different definitions of procedure *algo* (when a user runs an optimization, he is simply executing procedure *algo*). Of course, procedure *algo* may call other procedures or functions to perform its task. As we have seen, several common building block procedures that occur in many *algo* procedures include *direction*, *stepsize*, and *output*. These procedures call functions *cost*, *multicost*, *ineq*, *fineq*, etc., that define the problem being optimized. How to set up these functions was shown in section 4.4.2.

By combining the above additions, the new algorithm structure appears:

Algorithm Structure (4.4)

```
parameter AlgoParam1 = DefaultValue1  'Explanation1'
parameter AlgoParam2 = DefaultValue2  'Explanation2'
  . . .

procedure algo  {
   import AlgoParam1, AlgoParam2,  . . .
   Iter = 0
   repeat  {
      interaction
      compute h = direction (X[Iter])
      compute λ = stepsize (X[Iter],h)
      update  X[Iter+1] = X[Iter] + λ * h
      Iter = Iter + 1
      }
   forever
   }
```

where the following declarations occur elsewhere:

```
array_sequence X[20](Nparam)
global Iter
```

Not shown above are other RATTLE statements that use the algorithm parameters. These might occur, for example, in a sub-loop of the above procedure that calculates the search direction **h** so that it meets a particular criterion. Additionally, a recent feature added to the DELIGHT system by T. Wuu at Berkeley allows the above algorithm to be coded in such a way that for debugging or other purposes, each of the sub-blocks can be executed in a "single-step" manner by interactive requests.

As a first example of the construction of an algorithm by substituting library entries for the sub-blocks in algorithm (4.4), consider how an unconstrained optimization algorithm can be created. A user can combine a search-direction obtained through a quasi-Newton update formula, or through a conjugate gradient scheme, with a step-size rule based on cubic interpolation, or a pseudo-exact grid line search, all of which are sub-blocks from the RATTLE Optimization Algorithms Library. Each time a sub-block is substituted, the previous RATTLE

procedure body is superseded by the new body as demonstrated in the Newton-Raphson program development in section 4.2.7. DELIGHT features commands that make the use of this structured library extremely easy and effective, so that a large number of algorithms can be created from a relatively small number of procedures. Some of these commands to explore and substitute sub-blocks are discussed later in section 4.5.3.3.

A second example of a class of algorithms that utilizes the structure of algorithm (4.4) is shown in the next section.

4.5.1.3. Basic Feasible Directions Algorithms

The original feasible directions algorithms proposed by Zoutendijk [162], Polak [121], and others aim to solve the constrained nonlinear programming problem

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize:}}\ cost(\mathbf{x}) \\ &\text{subject to:}\ \mathbf{ineq}(\mathbf{x}) \le 0\,, \end{aligned} \tag{4.5}$$

with functions *cost* and **ineq** continuously differentiable, by taking a step along a search direction **h** that is a feasible direction. For any **x** in the *feasible set* $\{x \in \mathbb{R}^{Nparam} \mid ineq_j(\mathbf{x}) \le 0\,,\ j=1,2,\cdots,Nineq\}$, a *feasible direction* **d** is one which forms an angle of greater than 90° with the gradients of the cost and *active constraints*, i.e., the set $\{ineq_j(\mathbf{x}) \mid ineq_j(\mathbf{x})=0\}$. Small movements along such a direction result in a descent for both the cost and active constraints. In general, optimization algorithms try to construct a sequence whose accumulation points satisfy a set of necessary optimality conditions. Since it can be shown that the family of feasible directions algorithms produces sequences whose accumulation points satisfy a first order optimality condition called the *Fritz John condition* [121], we first state this condition.

A point $\hat{\mathbf{x}} \in \mathbb{R}^{Nparam}$ is a *Fritz John point* for problem (4.5) if $\hat{\mathbf{x}}$ is in the feasible set and there exist non-negative scalars $\mu_0, \mu_1, \cdots \mu_{Nineq}$, not all zero, such that

$$\mu_0 \nabla cost(\hat{\mathbf{x}}) + \sum_{j=1}^{Nineq} \mu_j \nabla ineq_j(\hat{\mathbf{x}}) = 0$$

and

$$\mu_j ineq_j(\hat{\mathbf{x}}) = 0, \quad j=1,2,\cdots,Nineq$$

where ∇ is the gradient operator. Then the Fritz John condition of optimality states that any solution to (4.5) is a Fritz John point [82].

A way of describing particular methods of feasible directions is simply to describe appropriate procedures for *direction* and *stepsize* of the algorithm outer loop given in algorithm structure (4.3) at the beginning of section 4.5.1.2. Letting the initial guess $\mathbf{x}_0$ be in the feasible set, these methods insure that the cost decreases while the iterates $\mathbf{x}_i$, $i=1,2,\cdots$ remain feasible by having the two procedures perform as follows. Procedure *direction* determines the search direction $\mathbf{h}$ such that a small change $\mathbf{x}_i + \lambda\mathbf{h}$ along $\mathbf{h}$, for sufficiently small λ, decreases the cost while remaining in the feasible set. One way of accomplishing this is to determine a direction which decreases the cost while moving into the interior of the feasible set away from the active constraint boundaries, i.e., to determine a *descent direction* for both the cost and the active constraints [121]. The search direction computation proposed by Pironneau and Polak in [120] results in a particularly well-conditioned procedure. The descent direction $\mathbf{h}$ is computed as the vector opposite to the nearest point to the origin in the convex hull of the set of active constraint gradients and the gradient of the cost. More precisely, $\mathbf{h}$ is given by

$$\mathbf{h} = -\mathrm{Nr\,co}\{\ \{\nabla ineq_j,\ j \text{ active}\} \cup \nabla cost\ \}$$

and is computed as the solution to the quadratic programming problem

$$\min_{\mathbf{h}} \{\ \|\mathbf{h}\|_2^2 \mid \mathbf{h} = \mu_0 \nabla cost + \sum_{j \text{ active}} \mu_j \nabla ineq_j\ ,\ \sum_{j \text{ active}} \mu_j = 1\ ,\ \mu_j \geq 0\ \} \quad (4.6)$$

using the *quadprog* macro as shown in the following "pseudo" RATTLE procedure:

Algorithm Segment (4.7)

```
procedure direction (x)  {

   Evaluate Vcost = gradcost(x)
   Place Vcost into a "Jacobian" matrix J whose rows
     are to contain the cost and active constraint gradients.

   for j = 1 to Nineq
      if ( ineq(j,x) == 0 )  {
         evaluate Vineq_j=gradineq(j,x)
         Consider this j as active by placing Vineq_j into J
         }

   Compute Q=JJ^T for use in the quadratic program:
     (using h=J^T mu, we have ||h||^2=h^Th=mu^TJJ^Tmu=mu^TQmu)

   quadprog mu_solution = argmin { mu'*Q*mu | ones'*mu=1, mu>=0}

   return ( -J'*mu_solution )
   }
```

Not shown above is the creation of the vector *ones* consisting of a column of ones of length equal to 1 plus the number of active constraints; "ones'*mu=1" represents the constraint $\sum_{j \text{ active}} \mu_j = 1$ from (4.6). Quadratic programming computation (4.6) above is used quite often in the RATTLE Optimization Algorithms Library.

Procedure *stepsize*, called directly after *direction* in the simple algorithm outer loop, chooses the scalar step size λ such that $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda \mathbf{h}$ remains feasible, i.e., $\mathbf{ineq}(\mathbf{x}_{i+1}) \leq 0$, and $cost(\mathbf{x}_{i+1})$ is sufficiently less than $cost(\mathbf{x}_i)$. Note that by using a search direction $\mathbf{h}$ that is a descent direction as suggested in the previous paragraph, such a λ can always be found. Particular step-size rules used

quite often in the RATTLE Algorithms Library are variations of the Armijo step-size rule [16]. Using one such rule, the step-size λ is computed such that $\mathbf{x}_i + \lambda\mathbf{h}$ remains feasible and such that

$$cost(\mathbf{x}_i + \lambda\mathbf{h}) - cost(\mathbf{x}_i) \leq -\alpha\lambda\|\mathbf{h}\|^2 \tag{4.8}$$

where $\alpha \in (0,1)$ is a user-settable algorithm parameter. One typical library step-size procedure computes trial λ values to satisfy this rule using $\lambda=\beta^k$ for $k=0,1,2,\cdots$ where $\beta \in (0,1)$ is another user-settable parameter. This approach is seen in the following "pseudo" RATTLE function, in which ψ_+ represents the constraint *violation*:

Algorithm Segment (4.9)

```
parameter Alpha = .5  'Slope of Armijo line.'
parameter Beta  = .5  'Trial stepsize along h is Beta**k'

function stepsize (x,h)  {
   k = 0
   repeat {
      Compute ψ = max ineq(j,x+Beta^k h)
                  j
      Compute ψ+ = max(ψ,0)
      if ( ψ+ == 0  &
           cost(x + Beta^k h)-cost(x)≤ - Alpha·Beta^k||h||^2 )
         return (Beta^k)
      k = k + 1
      }
   forever
   }
```

The above loop computes smaller and smaller trial step-size values $\lambda=Beta^k$ until $\mathbf{x}+\lambda\mathbf{h}$ remains feasible and (4.8) is satisfied.

Unfortunately, the description of procedure *direction* and algorithm segment (4.7) given above need to be modified in order to obtain a guaranteed convergent algorithm. Wolfe [159] has constructed examples that show that a "zig-zagging" phenomenon may take place and lead to a sequence of iterates having accumulation points which do not satisfy the Fritz John necessary conditions.

Even when the algorithm *is* convergent (which is most of the time), this same "zig-zagging" can cause its behavior to be very inefficient. To avoid these situations, constraints which are "almost" active must be considered when computing the search direction **h**. Thus, **h** must be any direction which is a descent direction for both the cost and the *ε-active* constraints $\{ineq_j(\mathbf{x}) \mid ineq_j(\mathbf{x}) \geq -\varepsilon\}$. Another necessary change to guarantee that the algorithm generate a sequence of points whose accumulation points satisfy the Fritz John necessary conditions is to dynamically vary ε in the outer loop. For more information, see [121].

In the form of the algorithms just presented, a feasible initial point must be provided. Since such a point is often not available, a feasible point may be generated using the Combined Phase I - Phase II Feasible Directions algorithm of Polak, et al. [123]. The simple algorithm outer loop given in algorithm structure (4.3) now appears:

Algorithm Structure (4.10)

```
repeat  {
   compute h = direction_Phase I (X)
   compute λ = stepsize_Phase I (X,h)
   update  X = X + λ*h
   }
until ( X is feasible )
repeat  {
   compute h = direction_Phase II (X)
   compute λ = stepsize_Phase II (X,h)
   update  X = X + λ*h
   }
until ( convergence criteria is met )
```

In phase I, algorithm (4.10) actually finds a feasible point by minimizing the maximum of all the inequality constraints $ineq_j(\mathbf{x})$[26]. In the next section, the basic feasible directions algorithm of this section is enhanced to make it more suitable for engineering design problems.

[26] In some formulations of the quadratic program used to determine the search direction, the cost is also considered in phase I. See [121].

4.5.2. Enhanced Feasible Directions Algorithms for Engineering Design

In section 4.4.1.2 a new multiobjective problem formulation, particularly well suited to engineering design, was introduced. This formulation was directed towards four requirements. Several of these were addressed in the same section and in section 4.4.2.2 where the corresponding problem description facilities were introduced. This section presents the enhanced optimization algorithm alluded to in section 4.4.1.2 that solves design problems formulated using the new problem formulation.

4.5.2.1. Why Feasible Directions Algorithms are Good for Engineering Design

Methods of feasible directions are very attractive algorithms for the types of optimization problems that occur in engineering design. In this section we list some of the reasons why this is so, also pointing out several shortcomings. The enhanced algorithm of the next section addresses these shortcomings.

Some of the reasons why feasible directions methods are particularly suited to engineering design are:

(1) As stated in the previous section, with the Phase I - Phase II versions, the algorithms do not require the initial point to be feasible, that is, to satisfy all of the design constraints. In most cases a feasible point is found after finitely many iterations, and once it is found, the cost function progressively improves while the sequence of points generated by the algorithm remains feasible. This is important since it gives a designer more freedom

to choose the point at which he considers the optimization process to be complete; stopping the optimization process at an early point still provides a feasible design, i.e., one that satisfies the constraints. If a feasible point does not exist or requires many iterations to reach, such algorithms are still useful since the design is improved at each iteration, i.e., in phase I, each iteration reduces the maximum constraint violation.

(2) These methods usually behave satisfactorily for any initial guess and have guaranteed convergence properties that depend on mild conditions which are satisfied in general by engineering design problems. Also, they are robust in the sense that errors in function and gradient computations do not seriously affect their behavior. This is important since simulation programs often do not compute exact system responses—there may be cpu-time/accuracy tradeoffs that make extremely precise computations prohibitive.

(3) The search direction chosen by the most popular of these methods has the simple closest-point-to-the-convex-hull geometric interpretation given in the previous section. Such an easily understood interpretation is important since it allows an advanced user to understand how the algorithm behaves so he can interact with it. Moreover, the scheme used to obtain this search direction and the geometric interpretation itself allows the designer to ask that the algorithm emphasize the improvement of certain constraints over others.

(4) Feasible directions methods usually do not require the computation of the gradients of all the constraints at each iteration (only those that are ε-active). This is an important property in cases where gradients are computed using finite differences with costly simulations.

(5) There are algorithms [54] which solve problems involving both ordinary and functional constraints. In order to solve problems in this class, the functional constraint interval must be discretized, i.e., sampled at selected points. Furthermore, to insure that an optimal point is approached, the discretization must be refined to contain more and more sample points. The *semi-infinite* optimization algorithms from [54] have the advantage that they do not require the user to specify a priori a finite subset of sample points of interest in the interval; the discretization of the interval range is automatic and is refined by the algorithms dynamically as an optimal solution is approached. (However, in DELIGHT the user must supply an initial discretization with the *for_every* statement presented in section 4.4.2.2)

(6) These methods have been used successfully for several types of optimization problems. In his master's thesis, Tang [146] was probably the first to apply successfully recent feasible directions methods to an integrated circuit design problem that had been formulated as a constrained nonlinear programming problem. Although several problems were encountered while testing his (batch) optimization program, he did achieve a great success in improving the performance of an integrated wideband amplifier. Similarly, the INTEROPTDYN system, in spite of the disadvantages mentioned in sections 2.2 and 2.3, met with some success in applying the Phase I - Phase II method with functional inequalities to real engineering design problems.

Although the methods are very attractive, there are a few shortcomings which can cause an optimization problem to be hard to set up and the iteration progress to be painfully slow. One of these is the sensitivity of the algorithms to the scaling of the constraint functions. In order to determine a search direc-

tion, the methods usually consider ε-active constraints. In phase I, a constraint $ineq_j(\mathbf{x})$ is said to be ε-active if $ineq_j(\mathbf{x}) \geq \psi_+ - \varepsilon$ where ψ_+ is the maximum violation of all constraints as shown in algorithm segment (4.9). Clearly scaling any constraint could move it into or out of the ε-active index set. Ideally, whether or not a constraint is considered ε-active should depend only on the position of the point and the geometry of the feasible set and not on any such constraint scaling.

Another shortcoming is that once the ε-active constraints have been identified, the closest-point-to-the-convex-hull approach described in the previous section is again sensitive to scaling—this time to scaling of the gradients. A constraint with a "short" gradient is much more likely to affect the search direction than one with a "long" gradient. Ideally, if all the constraints are satisfied (but some are ε-active), the gradient of the cost function being minimized should play the major role in determining the search direction.

Feasible directions methods also suffer from slow convergence—in fact, linear convergence [82]. However, the rate of convergence refers to the tail of a completely convergent sequence of iterations while, as pointed out in regard to the APLSTAP system in section 2.2, usually a designer is constantly making tradeoff decisions in a design process which should emphasize getting the most design improvement from the first few optimization steps rather than on reaching the limit of a convergent sequence.

4.5.2.2. Enhanced Phase I-II-III Algorithm

Nye, Tits, and Sangiovanni-Vincentelli [113] discuss several enhancements to the basic method of feasible directions introduced in section 4.5.1.3 that address the shortcomings presented in the previous section. These

enhancements have been combined with other ideas by Nye and Tits [117, 112] into a new optimization algorithm intended for engineering design problems ([117] is included as appendix F of this dissertation). For reasons which will become clear below, we give our algorithm the name *The Phase I-II-III Method of Feasible Directions*. This algorithm is intended to solve multiobjective constrained optimization problems that have been set up according to the mathematical programming formulation of section 4.4.1.2 and the problem description facilities of section 4.4.2.2. Terms used below such as *hard*, *soft*, *good*, and *bad* have already been defined in those sections. We first present a general description of the algorithm, followed by a more detailed RATTLE procedure. A convergence result is also stated without proof. For more information on this algorithm, see [112].

The algorithm may be viewed as consisting of three distinct phases (hence the name). In phase I, the algorithm focuses all of its attention on satisfying the constraints to which the designer has given highest priority—the hard constraints. After all the hard constraints are satisfied, phase II is entered and the algorithm shifts its attention to improving all performance objectives and soft constraints simultaneously, while keeping all hard constraints satisfied. In phase II, the algorithm does not distinguish between objectives and soft constraints in seeking to improve the design. In order to compare the various objectives and soft constraints, each such function is normalized according to the formulas given in the multiobjective programming formulation at the end of section 4.4.1.2. Thus, the meaning of the good and bad values, as stated previously, applies consistently to both objectives and soft constraints: having any objective at its good value and any soft constraint at its good value should provide the same satisfaction to the designer, and analogously for the bad values.

Finally, phase III is entered if all the soft constraints and all the performance objectives take on values equal to or better than their corresponding target good values. In phase III, the algorithm concentrates only on improving the performance objectives while keeping both the hard and soft constraints satisfied. The following table summarizes the three phases:

Phase I-II-III Method of Feasible Directions		
Phase	Condition	Action
I	At least one hard constraint is not satisfied.	Try to satisfy all the hard constraints.
II	All hard are satisfied. At least one soft constraint is not satisfied or one performance objective is worse than its good value.	Try to improve both the performance objectives and the soft constraints simultaneously while keeping the hard constraints satisfied.
III	All hard and soft constraints are satisfied and all performance objectives are at their good values or better.	Try to further improve the performance objectives while keeping all constraints satisfied, in particular, soft constraints better than their good values.

We now describe the phase I-II-III algorithm in greater detail. The multiobjective problem formulation of section 4.4.1.2 (without functional objectives or constraints) can be stated as the mathematical programming problem

$$\min_{\mathbf{x}} \{ \max_{j \in [1,2,\cdots,N_f]} f_j(\mathbf{x}) \mid g_j(\mathbf{x}) \leq 0 \ \forall j \in [1,2,\cdots,N_g] \} \tag{4.11}$$

where the correspondences of functions f and g are given shortly. We say that $\mathbf{x}$ is a *feasible point* for problem (4.11) if $g_j(\mathbf{x}) \leq 0 \ \forall j \in [1,2,\cdots,N_g]$.

We define $\mathbf{x}^*$ to be a *locally Pareto* (or *locally noninferior*) *point* for problem (4.11) if $\mathbf{x}^*$ is a feasible point and there is no other feasible point nearby that produces a decrease in some f_j without increasing any others. To be more precise, the latter condition is that there exists a $\delta>0$ such that there is no point $x \in B(\mathbf{x}^*,\delta)$ having the following properties:

$$\begin{aligned} &g_j(\mathbf{x}) \le 0 \qquad \forall j \in [1,2,\cdots,N_g] \\ &f_j(\mathbf{x}) \le f_j(\mathbf{x}^*) \qquad \forall j \in [1,2,\cdots,N_f] \\ &f_{j_0}(\mathbf{x}) < f_{j_0}(\mathbf{x}^*) \quad \text{for some } j_0 \in [1,2,\cdots,N_f] \end{aligned}$$

Let us introduce the following notation. For any $\mathbf{x} \in \mathbb{R}^{Nparam}$, let

$$\begin{aligned} \psi_f(\mathbf{x}) &\triangleq \max_{j \in [1,2,\cdots,N_f]} f_j(\mathbf{x}) \quad \text{and} \\ \psi_g(\mathbf{x}) &\triangleq \max_{j \in [1,2,\cdots,N_g]} g_j(\mathbf{x}) . \end{aligned} \tag{4.12}$$

For any $\varepsilon \ge 0$, we define the *ε-active* sets of function indices

$$\begin{aligned} J_{f,\varepsilon}(\mathbf{x}) &\triangleq \{\, j \in [1,2,\cdots,N_f] \mid f_j(\mathbf{x}) \ge \psi_f(\mathbf{x}) - \varepsilon \,\} \quad \text{and} \\ J_{g,\varepsilon}(\mathbf{x}) &\triangleq \{\, j \in [1,2,\cdots,N_g] \mid g_j(\mathbf{x}) \ge -\varepsilon \,\} . \end{aligned} \tag{4.13}$$

We now define the analog of a Fritz John point. A feasible point $\hat{\mathbf{x}}$ is a *stationary point* for problem (4.11) if there exist nonnegative vectors μ_f and μ_g, not all zero, such that

$$\begin{aligned} &\sum_{j=1}^{N_f} \mu_{f_j} \nabla f_j(\hat{\mathbf{x}}) + \sum_{j=1}^{N_g} \mu_{g_j} \nabla g_j(\hat{\mathbf{x}}) = 0 , \\ &\mu_{g_j} g_j(\hat{\mathbf{x}}) = 0 \quad \forall j \in [1,2,\cdots,N_g] \quad \text{and} \\ &\mu_{f_j} (\psi_f(\hat{\mathbf{x}}) - f_j(\hat{\mathbf{x}})) = 0 \quad \forall j \in [1,2,\cdots,N_f] . \end{aligned} \tag{4.14}$$

The necessary conditions for optimality for problem (4.11) may now be stated.

Proposition. Suppose that $f_1, \cdots, f_{N_f}$, $g_1, \cdots, g_{N_g}$, are continuously differentiable and suppose that $\mathbf{x}^*$ is a locally noninferior point for problem (4.11). Then $\mathbf{x}^*$ is a stationary point for problem (4.11). (For a proof see [112].)

We are now ready to formalize the phase I-II-III algorithm for solving problem (4.11). For any feasible point $\mathbf{x}$ and any $\varepsilon \geq 0$, we define

$$\mathbf{h}_\varepsilon(\mathbf{x}) \triangleq \frac{\mathbf{h}}{||\mathbf{h}||} \quad \text{and} \quad \vartheta_\varepsilon(\mathbf{x}) \triangleq -||\mathbf{h}|| \; ; \tag{4.15}$$

with $\mathbf{h}$ computed as the vector opposite to the nearest point to the origin in the convex hull of the set of gradients of ε-active f and g functions. More precisely, $\mathbf{h}$ is given by

$$\mathbf{h} = -\text{Nr co}\{ \{\nabla f_j(\mathbf{x}),\, j \in J_{f,\varepsilon}(\mathbf{x})\} \cup \{\nabla g_j(\mathbf{x}),\, j \in J_{g,\varepsilon}(\mathbf{x})\} \} \tag{4.16}$$

and may be computed as the solution to a quadratic program similar to (4.6). Our algorithm, described in the following "pseudo" RATTLE procedure, requires an initial feasible point $\{\mathbf{x}_0 \mid g_j(\mathbf{x}_0) \leq 0 \;\; \forall j \in [1,2,\cdots,N_g]\}$. Note that the invocation of function $\vartheta_\varepsilon(\mathbf{x})$ produces a direction vector $\mathbf{h}_\varepsilon(\mathbf{x})$ (written "$\mathbf{h}_\varepsilon$" below for clarity) according to (4.16) and (4.15). Note also that for clarity this algorithm includes the step-size calculation; it would normally be part of procedure *step-size*.

Algorithm Structure (4.17)

```
parameter Eps0  = .1  'Initial and maximum value of epsilon'
parameter Delta = .1  'Used to control when epsilon gets cut'
parameter Alpha = .5  'Slope of Armijo test line'
parameter Beta  = .5  'Trial step-size along h is Beta**k'
```

```
procedure algo  {
   Iter = 0
   repeat  {
      ε = Eps0
      while ( ϑε(x_Iter) ≥ -Delta·ε )
         ε = ε / 2
      k = 0
      repeat  {
         if ( ψg(x_Iter + Beta^k·h_ε) ≤ 0  &
              ψf(x_Iter + Beta^k·h_ε) - ψf(x_Iter) ≤ Alpha·Beta^k·ϑε(x_Iter) )
            break
         }
      forever
      update  x_Iter+1 = x_Iter + Beta^k·h
      Iter = Iter + 1
      }
   forever
   }
```

The main convergence theorem may now be stated.

Main Convergence Theorem. Referring to the above algorithm, if indefinite looping occurs in the *while* loop, then the corresponding $\mathbf{x}_{Iter}$ is a stationary point for problem (4.11). Otherwise, the sequence $\{\mathbf{x}_{Iter}\}$ constructed by the algorithm is infinite and any accumulation point is a stationary point. (For a proof see [112].)

We continue by relating the phase I-II-III method described earlier in general terms with problem formulation (4.11) and algorithm (4.17). All three phases are in fact particular cases of problem (4.11). To make the correspondences we do not consider functional objectives or constraints. In phase I, all hard constraints are f_j functions while (4.11) has no g_j functions. In phase II, all multiple objectives and soft constraints are f_j functions and hard constraints become g_j functions. In phase III, all multiple objectives are f_j functions and all constraints, both soft and hard, are g_j functions. Missing from algorithm (4.17) is the mechanism shown in [117, 112] to switch to the next phase when the appropriate conditions are met.

A summary of the enhancements made by the phase I-II-III method and its

corresponding problem formulation includes the following:

(1) The ability to handle multiple objectives.

(2) Hard and soft constraints handled in three different phases.

(3) Good and bad values to facilitate scaling and tradeoff exploration.

(The following enhancements to computational efficiency are discussed more fully in [113].)

(4) An efficient way of handling both hard and soft box constraints on design parameters.

(5) A new definition of ε-active that is invariant to scaling of performance objectives or constraints, i.e., that gives more importance to the geometry of the feasible set rather than to its description (i.e., scaling by multiplicative factors). This definition involves gradients in first-order approximations to the objectives and constraints as hinted at in equation (4.15). Since gradients are often expensive to compute, there is a preselection scheme for choosing objectives and constraints likely to be ε-active (not shown in the definitions in (4.13)).

(6) An improved search direction using a gradient normalization that similarly does not depend on scaling. It also allows the designer to easily request that more emphasis be given to the improvement as a group of the objectives *and* soft constraints in phase II or of the objectives *only* in phase III. This is achieved by the designer adjusting certain *push factor*[27] algorithm parameters.

[27] This term was introduced and used extensively by Bhatti et al. [21] in the feasible directions algorithm used in INTEROPTDYN.

(7) An improved Armijo step-size procedure, whose initial trial step-size (or one of the subsequent early trials) is more likely to pass the Armijo step-size test.

As stated in the above convergence theorem, with all of the enhancements above the convergence properties of the basic feasible directions algorithms are preserved.

4.5.3. Details of the RATTLE Optimization Algorithms Library

4.5.3.1. Detailed Structure of Library Entries

There are various types of procedures in the RATTLE Optimization Algorithms Library. The library is implemented as a directory of files (usually protected from being written into by ordinary users) containing one procedure per file. The procedures are classified according to their use by the capital letter which begins their filename. The following table lists the various classes:

Filename Starting Letter	Class of Library Entry
A	Algorithm organizers
M	Main (outer loop) procedures
D	Direction procedures
S	Step-size procedures
O	Output procedures
G	Graphical output procedures
P	Other procedures

The A-files (Algorithm) serve to organize other entries in the Library into working algorithms consisting of default choices and other allowable substitutes for each of several named algorithm sub-blocks. As an example, let us consider

the A-file *Agradnt*[28] for the Armijo-Gradient algorithm (steepest descent/gradient search direction with Armijo step-size rule):

```
##=========
## Agradnt - Gradient search with Armijo step-size rule.
##=========

handles unconstrained

choices_for stepsize  stepsize(x,h)   are Sarmijo; Sexact, Scubic
choices_for direction direction(x,h)  are Dgradnt; Dpolrib,
                                          Dbfgs, Ddfp
choices_for output    "Type 'output'" are Ostate; Oshort

mainloop_is Mdirstep
```

This file shows the information contained in an A-file. The types of problems which can be solved by the algorithm are specified using *handles*. It can be followed by a comma-separated list containing any of the following:

```
unconstrained
multicost
functional_multicost
equalities
inequalities
functional_inequalities.
sv_inequalities
box_constraints
special_purpose
```

In the Armijo-Gradient algorithm above, only unconstrained problems can be handled. The possible filename choices for the various algorithm sub-blocks (such as *direction*, *stepsize*, or *output*, above), as well as the procedure name and procedure arguments, are specified using *choices_for*. (For sub-block *output*, the procedure name is not given. Instead, the define *output* calls procedure *output_* and thus *output* may be typed to request the algorithm output at any time.) The first filename choice in the comma-separated list is the default and is indeed the choice which fills the algorithm description given at the top of the A-

[28] The filenames in this section may seem unnecessarily short; they have intentionally been restricted to eight characters or less, with no special characters such as ".", ":", "#", etc., for DELIGHT portability reasons.

file. The default choice for sub-block *stepsize* above is *Sarmijo*. The "main" procedure file containing the algorithm outer loop is specified using *mainloop_is*.

Library main procedures are in M-files and in all cases the procedure name is *algo*. A typical M-file contains some descriptive comments and initialization followed by procedure *algo* containing the outer loop which calls the various sub-procedures for the algorithm first level sub-blocks. Each pass through this outer loop constructs the next iteration of the array sequence X[Iter] (see section 4.5.1.2) and then increments the iteration counter variable, *Iter*.

The various other file classes in the table above are self-explanatory with the following clarifications. P-files are for second level sub-procedures needed by other procedures. For example, a direction or main procedure might need a local iteration consisting of a Newton-Raphson sub-iteration. However, currently, the organization of the Algorithms Library lacks sufficient hierarchy to be able to maintain sub-block choices at different levels. More is said about this later when future research is discussed in chapter 6.

A Simple Optimization Algorithm Example. Let us continue with the example used above: the Armijo-Gradient algorithm for unconstrained minimization. The A-file has already been shown above. The contents of the M-file, file *Mdirstep* are almost identical to enhanced algorithm (4.4) at the end of section 4.5.1.2 and are shown below:

File Mdirstep:

```
##==========
## Mdirstep - Direction/Stepsize main procedure.
##==========

DESCRIPTION : Unconstrained minimization outer loop.
```

```
procedure algo  {
   array h(Nparam)
   repeat  {
      interaction
      evaluate h = direction(X[Iter])
      lambda = stepsize (X[Iter], h)
      update X[Iter+1] = X[Iter]  +  lambda * h
      Iter = Iter + 1
      }
   forever
   }
```

The Armijo-Gradient Method consists of the following (default) files containing procedures for the two sub-blocks, *direction* and *stepsize*:

File Dgradnt:

```
##==========
## Dgradnt -
##==========

DESCRIPTION : Search direction is minus the gradient.

procedure direction (x,h)  {
   array x(Nparam), h(Nparam)
   evaluate h = Gradcost (x)
   matop h = (-1) * h
   }
```

File Sarmijo:

```
##==========
## Sarmijo - Armijo step-size rule.
##==========

DESCRIPTION : Basic Armijo rule for unconstrained minimization.

parameter Alpha = .5  'Slope of Armijo line.'
parameter Beta  = .5  'Trial stepsize along h is Beta**k'

function stepsize (x,h)  {

   import Alpha, Beta
   array x(Nparam), h(Nparam), gcost(Nparam)
```

```
evaluate gcost = Gradcost (x)
k = 0
repeat {
   update xnew = x + Beta**k * h
   delta_cost = Cost(xnew) - Cost(x)
   if ( delta_cost <= Alpha * Beta**k * <<h,gcost>> )
      return (Beta**k)
   k = k + 1
   }
forever
}
```

The above code is mostly self-explanatory. However, the following features are worth pointing out. First, there are two user-settable algorithm parameters, *Alpha* and *Beta*, declared in file *Sarmijo* using the *parameter* statement and used in function *stepsize*. The user is allowed to modify them either before starting execution or while execution is suspended (such as after a soft interrupt); their value is known to function *stepsize* at execution time. Since different problems require different values of *Alpha* and *Beta* for efficient solution, it is important to be able to modify them. Second, both procedures *direction* and *stepsize* above call on the problem description function names *cost* and *gradcost* given in section 4.4.2.1 except the function names begin with a capital letter. This pertains to the *problem interface* of normalizations and stored values presented later in section 4.5.4.2. Briefly, function *stepsize* calls function *Cost*, which itself calls function *cost*.

4.5.3.2. Listing of Library Entries

In this section we list the various entries of the RATTLE Optimization Algorithms Library[29]. In the following tables of entries, the columns labeled "Handles" use the following legend:

[29] Many of the algorithms in the Library were written by Andre Tits while at Berkeley.

U - Unconstrained
E - Equality constraints
I - Inequality (ordinary) constraints
F - Functional inequality constraints
B - Box constraints on design parameters
S - Singular value constraints

RATTLE Optimization Algorithms Library

Number (used below)	Algorithm Filename	Handles	Description
1	Abfgs	U	Broyden-Fletcher-Goldfarb-Shanno quasi-Newton update formula with pseudo-exact line search.
2	Acnsnewt	I F	Recursive linear programming stabilized by the Method of Feasible Directions.
3	Adfp	U	Davidon-Fletcher-Powell update formula.
4	Afeasdir	I F B	Gonzaga-Polak-Trahan method of feasible directions with box constraints.
5	Agradnt	U	Armijo Gradient Method: steepest descent gradient direction with Armijo stepsize rule.
6	Amultpl	E	Augmented Lagrangian (multiplier method).
7	Apolrib	U	Polak-Ribiere conjugate gradient scheme with pseudo- exact line search.
8	Acsvf	I F S	Polak-Wardi method of feasible directions for singular value functional inequality constraints.

Using the numbers in the first column above as keys to the algorithms, the following tables shows the valid sub-block substitutions for each algorithm for sub-blocks *direction*, *stepsize*, *output*, and *graphics*. Under each numbered column, a star ("*") indicates the default sub-block choice while a dash ("-") indicates other allowable choices. The last table also shows the "main" outer loop procedure for each algorithm.

Valid *direction* Sub-block Substitutions

1	2	3	4	5	6	7	8		
*	-		-	-	-			Dbfgs	Broyden-Fletcher-Goldfarb-Shanno update formula
						*		Dcsvf	Polak-Wardi nearest-point algorithm
-	*		-	-	-			Ddfp	Davidon-Fletcher-Powell update formula
	*	*						Dfeasdir	Modified Gonzaga-Polak-Trahan Method of Feasible Direction for semi-infinite problems
-	-		*	*	-			Dgradnt	Straight negative gradient direction
-	-		-	-	*			Dpolrib	Polak-Ribiere conjugate gradient scheme

Valid *stepsize* Sub-block Substitutions

1	2	3	4	5	6	7	8		
-	*		*	*		-		Sarmijo	Armijo stepsize rule for unconstrained minimization
						*		Scsvf	Modified Armijo stepsize rule for singular values
								Scubic	Cubic interpolation formula
*			-			*		Sexact	Pseudo-exact line-search (improved grid search)

Valid *output* Sub-block Substitutions

1	2	3	4	5	6	7	8		
						*		Ocsvf	Output for singular value constraint algorithm
		*						Ofeasdir	Output for Gonzaga-Polak-Trahan algorithm
	*							Ofrle	Output for recursive linear program algorithm
					*			Omultpl	Output for multiplier method
-		-		-		-		Oshort	Just the cost and the norm of its gradient
*		*		*		*		Ostate	Current iterate, cost and gradient

Valid *graphics* Sub-block Substitutions		
1 2 3 4 5 6 7 8		
* *	* Garmijo	Armijo sub-loop graphics
*	* Gclock	Angles between each gradient and h, each on the *same* gradient clock (see section 4.6.3)
-	- Gclocks	Angles between each gradient and h, each on a *separate* gradient clock (see section 4.6.3)
-	- Ghisgpt	Histories of certain algorithm quantities for feasible directions

"Main" Procedures For the Various Algorithms		
1 2 3 4 5 6 7 8		
* * * *	Mdirstep	Unconstrained minimization direction/stepsize main loop
*	Mfeasdir	Feasible directions main loop with functional and box constraints
*	Mfrle	Pseudo-Newton main loop stabilized by the Method of Feasible Directions
*	Mmultpl	Main loop for multiplier method (currently supporting equality constraints only)
*	Mpwsvd	Main loop for minimizing largest singular value of a matrix

4.5.3.3. Exploring and Substituting Sub-Block Choices

The sub-block choices given on a *choices_for* line in an A-file may be interactively explored and substituted, even in the middle of an optimization run. First of all, a user may type *choices* alone to find out what algorithm sub-blocks exist that have choices; it prints the names of the sub-blocks, and additionally, the procedure (or function) name and argument list in case it is desired to call the procedure manually. (In some cases, as shown below, the command to type for executing the sub-block is printed.) Then, he may type *choices* followed by one

of the sub-block names to have the actual filename choices printed for that sub-block. Along with each filename, is the *DESCRIPTION* line from the file that describes the particular choice for the sub-block. (Note the *DESCRIPTION* lines in the listings of files *Mdirstep*, *Dgradnt*, and *Sarmijo* in section 4.5.3.1.) These commands are demonstrated below for Library algorithm *Apolrib* (recall that what is actually typed by a user is shown in **boldface** type):

```
1> choices
SUB-BLOCK CHOICE NAMES     PROCEDURE AND ARGUMENT USAGE
   stepsize                stepsize(x,h)
   direction               direction(x,h)
   output                  Type "output"
1> choices stepsize
stepsize CHOICES         DESCRIPTION
 Sexact         Pseudo-exact line-search (improved grid search)
 Sarmijo        Plain Armijo stepsize rule for unconstrained
                minimization
1> choices direction
direction CHOICES        DESCRIPTION
 Dpolrib        Polak-Ribiere conjugate gradient scheme
 Dbfgs          Broyden-Fletcher-Goldfarb-Shanno update formula
 Ddfp           Davidon-Fletcher-Powell update formula
 Dgradnt        Straight negative gradient direction
1> choices output
output CHOICES           DESCRIPTION
 Ostate         Prints current iterate, cost and gradient
 Oshort         Prints just the cost and the norm of its gradient
```

After exploring the possible filename choices for a particular sub-block, one may be substituted for the current one by using the *substitute* command. Its usage is shown in the following continuation of the above example:

```
1> substitute Oshort
Substituting Oshort for output ...
1> substitute Dgradnt
Substituting Dgradnt for direction ...
WARNING: 'output' procedure may have to be changed.
```

A warning is given when the *direction* sub-block is substituted because the *output* procedure may be printing variables that have to do with the old procedure but not the new. After performing substitutions, the *identify* command, discussed more fully along with the *solve* command in section 4.5.4.1, shows what problem is being solved, the algorithm originally used, and the current sub-

block choices. Its usage is shown in the following continuation of the above example:

```
1> identify
PROBLEM: (none)
ALGO: Apolrib
  main_loop:  Mdirstep
  stepsize:   Sexact
  direction:  Dgradnt
  output:     Oshort
```

4.5.4. Problem/Algorithm Interface

The interface mentioned in section 4.5 between a problem being solved and a desired optimization algorithm, either from the RATTLE Optimization Algorithms Library or from a user's own files, is introduced in this section. This interface has two distinct aspects. One concerns how a user indicates the problem to solve and which algorithm to use. The *solve* command accomplishes this and is introduced in section 4.5.4.1. The other aspect involves two features that simplify the task of creating algorithms. The first, affectionately known to this author and other algorithm experts as the *problem interface*, is presented in section 4.5.4.2. The second, the ability of the problem interface to construct *surrogate costs* needed by certain algorithms, is presented in section 4.5.4.3.

4.5.4.1. The *solve* Command

The *solve* command allows a user to specify a problem and/or an algorithm for optimization. As mentioned in the previous section, the algorithm may exist either in the RATTLE Algorithms Library or in several of the user's own files. This command has three possible formats:

solve PROB_NAME using A-FILE_NAME	- Select both an optimization problem to solve and an algorithm to do the solving. The algorithm is specified as the name of an "A-file" (see section 4.5.3.1).
solve PROB_NAME	- Select just an optimization problem to solve.
solve using A-FILE_NAME	- Select just an optimization algorithm.

Whether selecting a problem, an algorithm, or both, *solve* causes all the necessary files to be included which have to do with the particular problem or algorithm. For a problem, the files are included in the same order as the problem description files presented in the first table of section 4.4.2.1 for the classical problem description or the first table of section 4.4.2.2 for the multiobjective problem description. Note that the PROBLEM_NAME argument is not the name of any file; the filenames are the specified problem name appended with "S", "P", "FI", etc. For an algorithm, the A-file whose name is given is first read by DELIGHT. Then, for each sub-block and its corresponding *choices_for* line (see section 4.5.3.1), the first (default) filename choice is also included, i.e., its procedure contents are RATTLE compiled. The names, default values, and descriptions of any user-settable algorithm parameters are printed on the screen. After all sub-blocks have been included, the "main" procedure file specified on the *mainloop_is* line is included. The purpose of the last two command formats presented above—to select just a problem or algorithm—is for *changing* the problem or algorithm, e.g., when the one not being selected has already been chosen by a previous *solve* command. Such a change is possible at any time since as pointed out in section 4.2.5, when an existing RATTLE procedure is recompiled, the new procedure body completely supersedes the previous one.

As an example of the use of the *solve* command for a multiobjective problem,

suppose the problem name is *pbname*. Typing *solve pbname*, causes files *pbnameS* and *pbnameP* to first be processed. Then, any of the files *pbnameM*, *pbnameFM*, *pbnameI*, and *pbnameFI* that exist, are processed. The name of the file currently being included is printed on the screen so that any error messages are clearly associated with the file which gave rise to them. Most DELIGHT error messages state the line number of the erroneous line within the file.

The following shows a sample of what might appear on the terminal screen when using the *solve* command to select problem *pbname* and algorithm *Afdirmhs*[30]:

```
1> solve pbname using Afdirmhs
including pbnameS          (47sec)
including pbnameP          (51sec)
Unknowns are in X(3), 20 past values stored.
-----------------------------------------
including pbnameM          (55sec)
including pbnameFM         (58sec)
including pbnameI          (62sec)
including pbnameFI         (65sec)
including <Afdirmhs>       (68sec)
including <Dfdirmhs>       (70sec)
PARAMETER:  Pushcost_1 = 1 : Importance of cost in phase 1
PARAMETER:  Pushcost_23 = 10 : Pushes the cost in phase 2-3
PARAMETER:  Epsscale = 5 : Used in phase-2 eps-active scheme
including <Sfdirmhs>       (160sec)
PARAMETER:  Alpha = .3 : Slope of Armijo test line
PARAMETER:  Beta = .5 : Trial stepsize along h is Beta**k
PARAMETER:  Kmin = -5 : Smallest value of k tried is Beta**k
including  <Mfdirmhs>      (190sec)
PARAMETER:  Eps0 = 1 : Initial value of eps for eps-active scheme
PARAMETER:  Mu = .05 : Number of samples doubled if eps < Mu/q
PARAMETER:  Hcuteps = .1 : eps is cut if ||h|| < Hcuteps*eps
including  <Ophas123>      (205sec)
including  <GPScomb>       (215sec)
1>
```

The times shown in parenthesis are the total cpu times at the beginning of the corresponding file inclusion since the startup of DELIGHT; adjacent differences indicate how much time was spent in processing each file. The user-settable algorithm parameters printed above are discussed in [113].

[30] This algorithm is the enhanced phase I-II-III algorithm previously introduced in section 4.5.2.2. It was not included in the list of Library entries in section 4.5.3.2 because it is very recent and is currently undergoing other enhancements.

After including all of the necessary files, *solve* performs a compatibility check between problem and algorithm and also checks for certain problem description inconsistencies, reporting any errors to the screen. The first check is to insure that the problem and algorithm are compatible, e.g., a constrained problem cannot be solved using an unconstrained optimization algorithm. This is done by comparing the values of the global optimization variables *Neq*, *Nineq*, etc., with the problem types the algorithm can handle that were given on the *handles* line in the algorithm A-file. For example, if $Nfineq \geq 0$ but the algorithm cannot handle functional inequalities, we have a compatibility error and an error message is printed on the screen. Problem description inconsistency checks include a check of whether the global optimization variables have been set correctly. For example, if a problem "FI" file exists (and hence is included), but *Nfineq* is zero, we have an error. After fixing errors in any of the problem description files, it is safest to redo the entire *solve* command. Once the *solve* command has been completed successfully without any error messages, optimization is ready to begin. This is taken up in section 4.5.5.

The *identify* command, briefly introduced in section 4.5.3.3, shows the problem being solved, the number of each type of objective (for a multiobjective problem) or constraint, the algorithm being used, and the sub-block choices. Its usage is shown in the following continuation of the above example:

```
1> identify
PROBLEM: pbname
  5  Parameter(s)
  2  Multicost Objective(s)
  3  Inequality Constraint(s)
  2  Functional Inequality Constraint(s)
```

```
ALGO: Afdirmhs
  main_loop:   Mfdirmhs
  direction:   Dfdirmhs
  stepsize:    Sfdirmhs
  output:      Ophas123
  graphics:    GPScomb
```

4.5.4.2. Problem Interface: Normalizations and Stored Values

The *problem interface* is a term given to a sequence of RATTLE functions that come between an optimization algorithm and the problem description functions, *cost*, *gradcost*, *multicost*, *gradmulticost*, *ineq*, etc. Each problem description function has three other similarly-named functions associated with it. For example, function *multicost* has associated with it functions *Multicost*, *multicost_*[31], and *saved_multicost*. The calling order is shown in the following call graph, where "⇒" connecting two functions means the function on the left calls the one on the right:

Optimization procedure ⇒ Multicost ⇒ multicost_ ⇒ multicost

This call graph also applies to all the other problem description functions. The *saved_* functions such as *saved_multicost* are not called by any optimization procedures; they exist to allow a user to obtain the benefits of the problem interface discussed below.

These functions have a threefold purpose:

(1) They act as conversion "filters" between algorithms and problem description files that allow algorithms to "see" a normalized problem. For multiobjective problems, the normalizations are shown in the multiobjective programming formulation at the end of section 4.4.1.2. Thus, functions such as

[31] Recall that the underscore character "_" is considered a letter by DELIGHT.

Multicost that are called directly by an algorithm are passed a normalized design parameter vector $\mathbf{x}'$ and receive function values normalized by the good and bad values. For this purpose, it is actually, for example, function *multicost_* that performs these normalize and unnormalize tasks; function *Multicost* has to do with surrogate costs, discussed below.

(2) They simplify the task of writing algorithms by making sure that unnecessary duplication of function evaluations is avoided. For example, a problem description function such as *Cost* called twice in a segment of RATTLE code with exactly the same arguments results in the actual function *cost* being called only once; in many cases, this allows the RATTLE code to appear "cleaner", i.e., easier to understand. Thus, one can write

```
if ( Cost(xnew)-Cost(x) <= Cost(x)* ... )
```

instead of the following "dirtier" code that worries about the duplicate call of *Cost* with the same argument x:

```
tempvar = Cost(x)
if ( Cost(xnew)-tempvar <= tempvar* ... )
```

How this function of the problem interface is performed is discussed below.

(3) They automatically compute *surrogate* function values needed by optimization methods such as Augmented Lagrangian methods [119]. This task is performed by the first level functions *Cost*, *Multicost*, etc.[32] This function of the problem interface is discussed in the next section.

How we accomplish the second purpose of the problem interface, avoiding duplicate function evaluations, is now addressed. One possibility is to have each

[32] At the time this was written, the surrogate function of the interface had only been implement-

second level interface routine (e.g., *multicost_*) store, say, the two most recent $\mathbf{x}'$ vectors passed to it and their corresponding function values returned from the lowest level problem description function (e.g., *multicost*). Then, when called, a second level routine first compares the incoming $\mathbf{x}'$ vector with the stored vectors. If a match is found, the stored function value is simply returned. If no match is found, then the next lower level function is called.

For the reason discussed next, the second level interface routines need to store the *three* most recent $\mathbf{x}'$ vectors passed to them. In many Algorithm Library *stepsize* procedures that implement variations of the Armijo step-size rule, an attempt is made to find the largest step-size that passes the rule. First, an initial step-size is tried. If this step-size passes, successively larger step-sizes are tried. When a step-size is found that fails the rule, the previous (smaller) step-size is returned. "Pseudo" RATTLE code for such a step-size search is as follows:

```
Initialize λ, the first trial stepsize.
x_new' = x_Iter' + λh
delta_cost = Cost(x_new') - Cost(x_Iter')
(Assume this delta_cost passes the step-size rule.)
repeat  {
   λ_new = next larger λ
   x_new' = x_Iter' + λ_new h
   delta_cost = Cost(x_new') - Cost(x_Iter')
   if ( delta_cost fails step-size rule )
      return (λ)
   λ = λ_new
   }
forever
```

With the *Cost* function values stored for the two most recent $\mathbf{x}'$ vectors passed to function *Cost*, the above code would never have to duplicate the evaluation of function *cost* for argument $\mathbf{x}_{iter}'$. This is because the sequence of $\mathbf{x}'$ vectors passed to *Cost* is

---

ed for function *Cost*.

$$\mathbf{x}_{new}', \ \mathbf{x}_{iter}', \ \mathbf{x}_{new}', \ \mathbf{x}_{iter}', \ \cdots$$

and thus $\mathbf{x}_{iter}'$ is always one of the most recent two at any time. However, when the step-size test fails, the previous trial λ is returned. The $\mathbf{x}'$ vector associated with this λ is not one of the most recent two but *is* one of the most recent *different* three. This can be seen above if the first $\mathbf{x}_{new}'$ is considered to be associated with the λ being returned. Thus, for each of the problem description functions, the three most recent $\mathbf{x}'$ vectors are stored in the problem interface.

To achieve the problem-interface purpose of avoiding duplicate function evaluations by storing the three most recent function values and their corresponding $\mathbf{x}'$ vector pairs, a means of maintaining the three pairs is needed. An operating system *least-recently used* policy [137] can be imitated, using, as suggested by Knuth [83], a doubly linked list of indices to the three stored pairs. When a function call is made and the $\mathbf{x}'$ argument matches one of the stored vectors, the doubly linked list is (trivially) reorganized by having the matching index moved to the front of the list. This has the effect of sorting the indices in order of their most recent use: the front index is the most recent while the rear index corresponds to the stored vector to discard if no match is found. Since in our case there are only three indices, the linked list can be implemented without the links as shown in the following fragment of "pseudo" RATTLE code from problem interface routine *cost_*. Assume p_1, p_2, and p_3 are the three indices, and $\mathbf{x}_{store_1}$, $\mathbf{x}_{store_2}$, $\mathbf{x}_{store_3}$, and $cost_{store_1}$, $cost_{store_2}$, $cost_{store_3}$, are the corresponding stored pairs. For simplicity, we ignore any initialization or declarations:

```
function cost_(x)  {
   if ( x == x_store_p1 )            # Was first in the list.
      Continue
   else if  ( x == x_store_p2 )      # Was second in the list.
      Swap p1 and p2
   else if  ( x == x_store_p3 )  {   # Was third in the list.
      # Rotate pointers to bring p3 to first position:
      p_save = p1
      p1 = p3
      p3 = p2
      p2 = p_save
      }
   else  {                           # Was not in the list.
      # Rotate pointers to bring p3 to first position,
      # then store new pair into first storage position:
      p_save = p1
      p1 = p3
      p3 = p2
      p2 = p_save
      x_store_p1 = x
      cost_store_p1 = cost(x)
      }
   return ( cost_store_p1 )
   }
```

For reference, we include the following table that shows the problem interface functions, whether each function returns normalized or "raw" values, and the function arguments (in particular, whether the **x** argument is normalized or raw). Note that the interface functions for functional objectives or constraints are slightly different than the basic scheme presented earlier. This is due to the fact that the interface stores pairs only at the sampled values of the "W" variables.

Problem Interface Functions		
Function	Function Returns	Arguments
Cost	normalized	$(\mathbf{x}')$
cost_	normalized	$(\mathbf{x}')$
cost	raw	$(\mathbf{x}_{raw})$
saved_cost	raw	$(\mathbf{x}_{raw})$
Multicost	normalized	$(j,\mathbf{x}')$
multicost_	normalized	$(j,\mathbf{x}')$
multicost	raw	$(j,\mathbf{x}_{raw})$
saved_multicost	raw	$(j,\mathbf{x}_{raw})$
Dfmulticost	normalized	$(j,\mathbf{x}',l,q)$
dfmulticost_	normalized	$(j,\mathbf{x}',l,q)$
dfmulticost	raw	$(j,\mathbf{x}_{raw},l,nmesh)$
fmulticost	raw	$(j,\mathbf{x}_{raw},W)$
saved_dfmulticost	raw	$(j,\mathbf{x}_{raw},l,nmesh)$
Ineq	normalized	$(j,\mathbf{x}')$
ineq_	normalized	$(j,\mathbf{x}')$
ineq	raw	$(j,\mathbf{x}_{raw})$
saved_ineq	raw	$(j,\mathbf{x}_{raw})$
Dfineq	normalized	$(j,\mathbf{x}',l,q)$
dfineq_	normalized	$(j,\mathbf{x}',l,q)$
dfineq	raw	$(j,\mathbf{x}_{raw},l,nmesh)$
fineq	raw	$(j,\mathbf{x}_{raw},W)$
saved_dfineq	raw	$(j,\mathbf{x}_{raw},l,nmesh)$

For the above table, we use the following legend:

j	-	Objective or constraint index.
$\mathbf{x}_{raw}$	-	Raw design parameter vector.
$\mathbf{x}'$	-	Normalized design parameter vector.
l	-	Discretization index (0 to nmesh).
nmesh	-	Number of sampling meshes for j'th objective or constraint.
q	-	Multiplier from algorithm for number of mesh points: nmesh = q times the number of initial sampling meshes, which is one less than the number of samples of the "W" variable.

The purpose of the *saved* functions is to allow a user to get at the stored raw values from the problem interface, i.e., without duplicating function calls to *cost*, *multicost*, *fineq*, etc. Before using a *saved* function, one must generate a raw design parameter vector from the current normalized vector $\mathbf{x}_{Iter}'$ using:

```
makeXraw ARRAY_NAME
```

Then, for example, one could plot the sampled raw values of functional constraint number 2 at the present iterate $\mathbf{x}_{Iter}'$ using

```
makeXraw xraw
plot saved_dfineq(2,xraw,l,10) vs l from 0 to 10 by 1
```

assuming the *for_every* line in the "FT" file of the multiobjective problem description resulted in 10 initial sampling meshes, i.e., 11 initial sample points.

4.5.4.3. Problem Interface for Surrogate Cost

The task of automatically computing *surrogate costs* performed by the first level problem interface functions *Cost*, *Multicost*, etc., is presented in this section. To find out just where these surrogate cost functions originate, we review the brief history of multiplier methods presented in [119]. Then we present a segment of RATTLE code from function *Cost*.

Forerunners of multiplier methods are penalty function methods, in which a constrained minimization problem is transformed into a sequence of unconstrained minimization problems. One such approach is the method of Fiacco and McCormick [47], briefly mentioned earlier in section 2.1. Their transformation is accomplished by augmenting the original cost function with weighted terms of the problem constraint functions thereby defining a new *surrogate cost* function. By gradually removing the effects of the constraints in the new cost function by a controlled parameter adjustment, it is possible to generate a sequence of unconstrained problems whose solutions converge to the solution of the original constrained problem. However, the parameter adjusted may be required to increase without bound. Moreover, even if its adjustment remains bounded, the unconstrained problems can become ill-conditioned as the parameter is increased.

In the earlier fifties, Kuhn and Tucker [84] gave the relationship between the nonlinear programming problem

$$\underset{\mathbf{x}}{\text{minimize:}}\ cost(\mathbf{x})$$

$$\text{subject to:}\ \mathbf{eq}(\mathbf{x})=0 \quad \text{and} \quad \mathbf{ineq}(\mathbf{x})\leq 0$$

and the corresponding Lagrangian function

$$L(\mathbf{x},\mathbf{Meq},\mathbf{Mineq}) \triangleq cost(\mathbf{x}) + \sum_{j=1}^{Neq} Meq_j eq_j(\mathbf{x}) + \sum_{j=1}^{Nineq} Mineq_j ineq_j(\mathbf{x})$$

where the Meq_j's and the $Mineq_j$'s are real variables called *Lagrange multipliers*. Many early modified Lagrangian functions became the basis of what subsequently developed into *Augmented Lagrangian* algorithms, also called multiplier methods. These methods, independently introduced by Hestenes [62] and Powell [127] in 1968, proposed to add to the Lagrangian the sum of the squares

of the constraint functions. A series of unconstrained minimizations of this penalized Lagrangian is followed by an update of the multiplier vector according to a simple rule. Powell showed that if second order sufficient conditions for optimality are satisfied, the algorithm converges without requiring the penalty factors to grow without bound, an advantage not found in the usual penalty methods.

The augmented Lagrangian algorithm found in the RATTLE Optimization Algorithms Library presently deals only with equality constraints $\mathbf{eq}(\mathbf{x})=0$. It attempts to perform a sequence of unconstrained minimizations on Hestenes' and Powell's augmented Lagrangian by defining the surrogate cost

$$L(\mathbf{x},\mathbf{Meq},c) \triangleq cost(\mathbf{x}) + \sum_{j=1}^{Neq} Meq_j eq_j(\mathbf{x}) + \frac{c}{2}||\mathbf{eq}(\mathbf{x})||^2 \qquad (4.18)$$

and varying $\mathbf{Meq}$ and c in a prescribed manner. Since the library already contains several algorithms for unconstrained minimization, it is convenient and in accord with the goals of the Algorithms Library if these existing algorithms are used for this purpose. However, these algorithms can be used without modification only if the surrogate cost above is constructed outside of the algorithm, i.e., at a "lower level". This then is the task of the first level problem interface routines.

How this task is accomplished can be seen in the following slightly simplified version of procedure *Cost*:

```
function Cost(x)  {
   import CostFlag
   if ( CostFlag == COST )   # If computing cost only...
      return cost_(x)
   else  {
      import c
      array eqvec(Neq)
      for j=1 to Neq
         eqvec(j) = eq_(j,x)
      return ( cost_(x) + <<Meq,eqvec>> + (c/2)*||eqvec||**2 )
      }
   }
```

In the above, **Meq** is a global Lagrange multiplier vector. Imported variables *CostFlag* and *c* are set elsewhere, in particular in the augmented Lagrangian algorithm main procedure and one of its sub-procedures. If only the cost is being computed, function *Cost* does nothing more than pass the argument **x** through to the next lower problem interface level, function *cost_*. Note that the above function makes use of the "<<" and "||" matrix macros of section 4.3 in computing expression (4.18).

4.5.5. Running an Optimization

The *run* command starts an optimization algorithm in DELIGHT and has the format:

```
run  [ COUNT ]  [ OUTPUT_COMMAND ]
```

in which COUNT is the number of optimization iterations desired and OUTPUT_COMMAND is an optional command which will be executed each iteration directly after the standard algorithm output. Recall from section 4.5.1.2 that a call to an algorithm output procedure occurs within each "main" procedure outer loop. If *run* is typed alone, the algorithm will run indefinitely whereas *run 2* will carry out two iterations. To get output graphics such as the performance comb (explained later in section 4.6.2) after each iteration, one could type *run 3 Pcomb*, for example. As a final example, *run 3 ufunc()* would call a user-defined

procedure called *ufunc* directly after the standard algorithm output after each optimization iteration.

As explained previously in section 4.5.1.2, once an algorithm has started executing, the user can interrupt its execution by generating a soft interrupt. This causes the algorithm to suspend at a "major stopping point", i.e., to suspend execution after completing the current iteration. Of course a hard interrupt may also be generated to stop RATTLE execution immediately. After a soft interrupt, execution may be resumed simply by reissuing another *run* command. Before resuming the designer will probably have modified one or more design parameters, good or bad values, or other design problem or algorithm parameters. Section 4.6.2 shows how to adjust the good and bad values.

4.6. Graphics

We stressed in chapter 3 that the complex information needed to be conveyed to an optimization user to assist him in making decisions, particularly concerning optimization, required graphical feedback showing algorithm and problem performance, most effective in color. This section covers several aspects of DELIGHT graphics that address that need. We first present general graphical features of DELIGHT in section 4.6.1. These "low-level" features are used throughout all graphics applications in DELIGHT, including those for optimization in sections 4.6.2 and 4.6.3 where we direct ourselves towards several graphics tools that have been developed to allow a user to quickly see the performance of his problem being solved or the algorithm he is using. The overall goal of the use of these tools is to enhance the ability of DELIGHT to solve optimization problems quickly.

An important point to stress here is that if a user is unsatisfied with any of

the existing graphical output provided by DELIGHT, the low-level primitive graphics commands presented in section 4.6.1 allow him to create his own graphics procedures at any time. Whether written in RATTLE or Fortran, his graphics procedures can be created from the same low-level primitives; these primitives are available to both RATTLE procedures and to Fortran (or Ratfor) routines from a DELIGHT Ratfor graphics library.

4.6.1. General Graphics Features

To convey information effectively using graphics, DELIGHT features many high and low-level, terminal-independent graphics commands (or statements) for creating many types of graphical displays. A prerequisite to using these commands is an understanding of the notions of *viewport* and *world coordinate bounds*.

The concepts of *viewport* and *world coordinate system* are taken from *Principles of Interactive Computer Graphics* by Newman and Sproull [106]. A viewport is a rectangular region on the terminal screen where graphics output is sent. The *viewport* command specifies this region in a 0,0-1,1 coordinate system in which coordinate 0,0 is at the lower left corner of the screen and 1,1 is at the upper right corner. This *screen coordinate system* is identical for all graphics terminals, independent of the shape (aspect ratio) of the terminal screen. For example, *viewport 0 0 .1 .1* specifies a very small square (or possibly rectangular) region in the lower left-hand corner of the screen while *viewport .45 .45 .55 .55* specifies a small region in the center of the screen. By using the *viewport* command, graphical output may be directed to anywhere on the screen by simply setting the viewport coordinates that define the desired rectangular region. In other words, graphics procedures may be written without setting and without

any knowledge of the present viewport. The viewport may then be set for some particular graphical procedure and all its subsequent output will automatically be mapped into the viewport rectangle. This allows a user to place either one or several graphical displays on the screen simultaneously by simply setting the appropriate viewport right before invoking each procedure.

World coordinate bounds are the bounding x,y values between which all x,y coordinates are actually passed to the graphics commands. For example, after the command *world -100 -100 100 100*, which sets the lower left bound to -100,-100 and the upper right bound to 100,100, the command *cursor 0 0*, whose arguments are in world coordinates, would place the cursor at the very center of the present viewport ready for text output. Similarly, *vector -100 -100 100 100* would draw a diagonal completely across the present viewport. As a final example, a plot of an expression versus an independent variable could be drawn by setting the world coordinate bounds to the minimum and maximum expression values over the independent variable range and then passing the raw expression values, *without any mapping whatsoever*, to the *vector* command.

A particular set of viewport coordinates and a set of world coordinate bounds can be associated with a named object called a *window*. The purpose of this is to declare these coordinates for a window only once and then reference the coordinates by name. For example, suppose a graphics application needed to break the screen up into three horizontal sections. Windows *top*, *middle*, and *bottom* would first be declared at one place in the RATTLE file. The graphics procedure could then easily place output into any of the three windows by moving in and out of them by name using the *window* command, discussed below.

To define a window, the *define_viewport* and *define_world* commands are

used. These commands have the following syntax:

```
define_viewport WNAME SX1 SY1 SX2 SY2
define_world WNAME WX1 WY1 WX2 WY2
```

in which *WNAME* is the name of the window being defined, *SX1,SY1* and *SX2,SY2* are the lower left and upper right coordinates of the viewport rectangle in the 0,0-1,1 screen coordinate system, and *WX1, WY1* and *WX2, WY2* are the lower left and upper right world coordinate bounds. If both of the above commands define the same window name, then that name is associated with both quadruples of coordinates. If only one of the above commands is given, the values for the command not given default to *0 0 1 1*. After defining a window, the command

```
window WNAME
```

may be used to "enter" the window; it has the same effect as if both *viewport* and *world* commands had been given with the quadruples associated with the window. A related command *window_list* displays the viewport and world quadruples associated with all defined windows.

To give an example of the commands pertaining to windows, we use the example application given above that needs to break the screen up into three horizontal sections. This could be accomplished by the following terminal session:

```
1> define_viewport bottom 0  0  1 .33
1> define_viewport middle 0 .33 1 .67
1> define_viewport top    0 .67 1 1
1> window_list
 window name ---- viewport coor ----      ====== world coor ======
     screen  0.0   0.0     1.0   1.0      0.0   0.0     1.0   1.0
        top  0.0   .67     1.0   1.0      0.0   0.0     1.0   1.0
     middle  0.0   .33     1.0   .67      0.0   0.0     1.0   1.0
     bottom  0.0   0.0     1.0   .33      0.0   0.0     1.0   1.0
```

In the above list of windows, there is a built-in window called *screen* whose

viewport is the entire screen. After these declarations, graphical output could be sent to any of these windows by preceding the RATTLE procedure graphics statements with one of the following:

```
window screen
window top
window middle
window bottom
```

We are now ready for the syntax of the low-level graphics commands provided by DELIGHT. The following tables categorize the commands into those for setting the terminal type, setting the viewport and world coordinate bounds, drawing vectors, outputing text, erasing parts of the screen, miscellaneous low-level functions, and higher level graphics functions. In the command formats below, coordinates given in arguments that begin with the letter "S" are given in the 0,0-1,1 screen coordinate system whereas those that begin with the letter "W" are in world coordinates.

DELIGHT Graphics Commands

Command and Usage	Description
Setting the Terminal Type	
terminal	Print the present terminal type.
terminal choices	Print out a list of allowable terminal types for the arguments of the *terminal* command.
terminal NAME	Set the terminal type to the specified terminal name.
grinit [N]	Initialize the terminal for graphics. Optional argument *N* is the number of lines in the non-graphics part of some terminal screens (such as the Tektronix 4027).
Setting the Viewport and World Coordinate Bounds	

viewport SX1 SY1 SX2 SY2	Set the viewport rectangle bounds to lower left screen coordinate SX1,SY1 and upper right screen coordinate SX2,SY2.
world WX1 WY1 WX2 WY2	Set the world coordinate bounds to lower left coordinate WX1,WY1 and upper right coordinate WX2,WY2. These map x,y world coordinates passed to most graphics commands into the present viewport.

Drawing Vectors

move WX WY	Set the beginning of the next vector to draw at world coordinate WX,WY, ready for a *draw*.
draw WX WY	Draw a vector from the previous (moved to) position to world coordinate WX,WY.
vector WX1 WY1 WX2 WY2	Draw a vector from world coordinate WX1,WY1 to WX2,WY2.
clip_draw WX WY	Same as *draw* except the vector drawn is clipped to lie within the present viewport using the Cohen and Sutherland clip algorithm of Newman and Sproull (see page 66 of [106]).
clip_vector WX1 WY1 WX2 WY2	Same as *vector* except the vector drawn is clipped to lie within the present viewport.

Outputing Text

cursor WX WY	Position the cursor so that the next text output is at world coordinate WX,WY.
cursorel WX WY NX NY	Position the cursor NX character-width units to the right and NY character-height units above world coordinate WX,WY. See below.
text 'control_str' [arg1 ... arg6]	Output text characters at the present cursor position using a *printf*-like control string and optionally from zero to six arguments.
textsize N	Set the current output text size where N is between 1 and 10, 1 signifying the smallest and 10 the largest.

textdirection ANGLE	Set the current output text direction using the specified counter-clockwise rotation angle in degrees. For example if ANGLE is 90 degrees, then any output from the *text* command may be read vertically up the screen.

Erasing Parts of the Screen

erase	Erase the entire graphics screen.
erase NAME	Erase the window with window name NAME.
werase SX1 SY1 SX2 SY2	Erase the rectangular region defined by lower left screen coordinate SX1,SY1 and upper right coordinate SX2,SY2.

Miscellaneous Low Level Commands

color COLOR	Set the present color to COLOR where the permissible colors are *white*, *black*, *red*, *orange*, *yellow*, *green*, *blue*, *sky*, *light*, and *bright*.
bigdot WX WY [DIAMETER]	Output a "big dot" at world coordinate WX,WY. Optional argument DIAMETER (default=.5) is the diameter of the dot in units of the height of a character.

Higher Level Graphics Commands

box	Draw a rectangular box around the present viewport.
curve Y [TOPCOL BOTCOL THRESH]	Draw a connected curve of the contents of 1-dimensional array Y. Optionally, top color TOPCOL, bottom color BOTCOL, and threshold THRESH may be given so that the curve changes color when the plotted values go above or below the threshold value. Note, the user must set the world coordinates to bound the values in the array; this command simply sends the values to *move* and *draw*. See below.

barplot Y [TOPCOL BOTCOL shift=S THRESH]	Same as *curve* except vertical bars are drawn for each point. Optional shift S may be given to make all the bars shift to the right so several barplots can be output in the same viewport. S is given as the reciprocal of the number of plots that are to appear simultaneously, i.e., S=.05 will shift the bars to the right so that 20 bars will fit without overlap. See example below.
oval	Inscribe a maximum size oval in the present viewport.
plot ...	Discussed later in this section.
plot3d ...	Discussed later in this section.
scat ...	Discussed later in this section.

The *cursorel* command provides a means of making the position of graphics text independent of the viewport size or shape and is used for this purpose throughout most DELIGHT graphics applications. For example, suppose the present world coordinate bounds are lower left 0,0 and upper right 100,100 and we wish to output a five character label in the upper right corner of the present viewport. Instead of positioning the cursor in absolute world coordinates such as by using *cursor 90 98*, it should be positioned relative to some fixed corner or symbol. In our case we would use *cursorel 100 100 -5 -1* to drop down one and back five units of character size from the upper right corner of the present viewport. The five characters output at this position would appear in the upper right corner no matter what viewport or text character size were used. This use of *cursorel* is shown later in this section.

Interesting and useful applications of the *curve* and *barplot* commands are, respectively, for displaying a functional constraint versus its "W" variable and

for displaying several objective or constraint values versus their index, i.e., with each associated with a different vertical bar in the same viewport. For these applications, the top color, the bottom color, and the threshold arguments for these commands may be used to enhance the information in the display by following Myer's [104] rules for human perception of symbols. For example, making the top color red, the bottom color green, and the threshold equal to the good value can give a curve in which constraint violations are highlighted by being shown in red. For functional constraint 2 versus TIME, this could be accomplished, for example, with the following commands:

```
1> array fvals(101)
1> i = 0
1> loop TIME from 0 to 100ns by 1ns  {
1}     i = i + 1
1}     fvals(i) = fineq(2, X, TIME)
1}     }
1> curve fvals red green 5.5v
```

Here we assume that the good curve versus TIME is constant at 5.5v. The *loop* statement takes arguments identical to those on the *for_every* line of section 4.4.2.2.

Before outputing any graphics in DELIGHT, a user must be on one of the graphics terminals supported by DELIGHT graphics. To see what terminals are supported, *terminal choices* can be typed. Each line shown is for a different terminal or plotter. At the time this was written, the following graphics terminals and plotters were supported:

Terminals and Plotters Supported by DELIGHT		
Terminal Synonyms		Description
hp	hp2648a	HP2648a graphics terminal.
2623	hp2623a	HP2623a graphics terminal.
chp	hp2627a	HP2627a color graphics terminal.
7220	hp7220	HP7220 graphics plotter.
4027	tek4027	Textronix 4027 color terminal.
4025	tek4025	Tektronix 4025 black/white terminal.
4010	tek4010	Tektronix 4010 storage crt terminal.
4015	tek4015	Tektronix 4015 storage crt terminal.
4662	tek4662	Tektronix 4662 graphics plotter.
dumb		Any non-graphics terminal (80x22).
dumb132		Dumb targeted for a line printer (132x66).
ram	ramtek	Ramtek 6000 color terminal.
soltec		Soltec 281 Digital Plotter
none	off	Turn off all graphics output.

Typing *terminal* alone prints the present terminal type. To specify the Tektronix 4027, for example, one would type *terminal 4027*. After specifying a terminal, *grinit* should be typed to initialize the terminal for graphics. Presently it has some effect for only a few terminal types. Appendix B.3 gives the implementation details of how terminal-independence is achieved by DELIGHT graphics primitives.

The following example graphics commands produce the output shown in figure 4.6. (On some black and white terminals such as the HP2648a used for this demonstration, different colors are simulated using different dashed line types.) Here we assume that the windows *top*, *middle*, and *bottom* have already been set up as shown earlier in this section with the following exception: the upper right viewport x-coordinates of each are 0.5 instead of 1.0 so that the copy of the HP2648a display in figure 4.6 fits better on a page. Thus for example, we use *define_viewport top 0 .67 0.5 1* instead of *define_viewport top 0 .67 1 1*.

```
1> terminal hp                  # Set terminal type.
1> terminal                     # Check terminal type.
Terminal is hp

1> window top                   # Enter top window.
1> color white                  # Set color to white.
1> box                          # Draw box around viewport.

1> world -100 -100 100 100      # Set world coord. bounds.
1> color green
1> oval                         # Inscribe oval in window.
1> color sky
1> vector -80 -80 80 80         # Draw vector in world coord's.

1> procedure demo  {            # Procedure for simple graphics.
1|   color white
1|   cursor 0 0                 # Position cursor at window center.
1|   text 'PI=%r' PI            # Output text at present position.
1|   cursorel 100 100 -5 -1.5 # Position cursor near upper right.
1|   text 'x=10'                # Output label without power of 10.
1|   cursorel 100 100 -1 -1.0 # Position cursor for power of 10.
1|   text '2'                   # Output power of 10.
1|   }

1> window middle                # Enter middle window.
1> world -100 -100 100 100
1> box
1> demo()                       # Run the above procedure.
```

To demonstrate that through the use of the *cursorel* command the above procedure and in particular the position of text output are independent of the viewport, let us set a much smaller viewport and again call the procedure:

```
1> viewport .02 .4 .23 .6       # Set small viewport.
1> color yellow
1> box
1> demo()                       # Rerun the above procedure.
```

This viewport is on the left of and inside window *middle* in figure 4.6. Notice that the 10^2 is still in the upper right corner. We might add here that the development of graphics procedures such as the one above is aided greatly by the incremental program development features of RATTLE one-pass compilation (see section 4.2.7). This is due to the invariable fine tuning required to make a graphics display appear "just right".

We end our graphics examples by demonstrating the *barplot* command. For

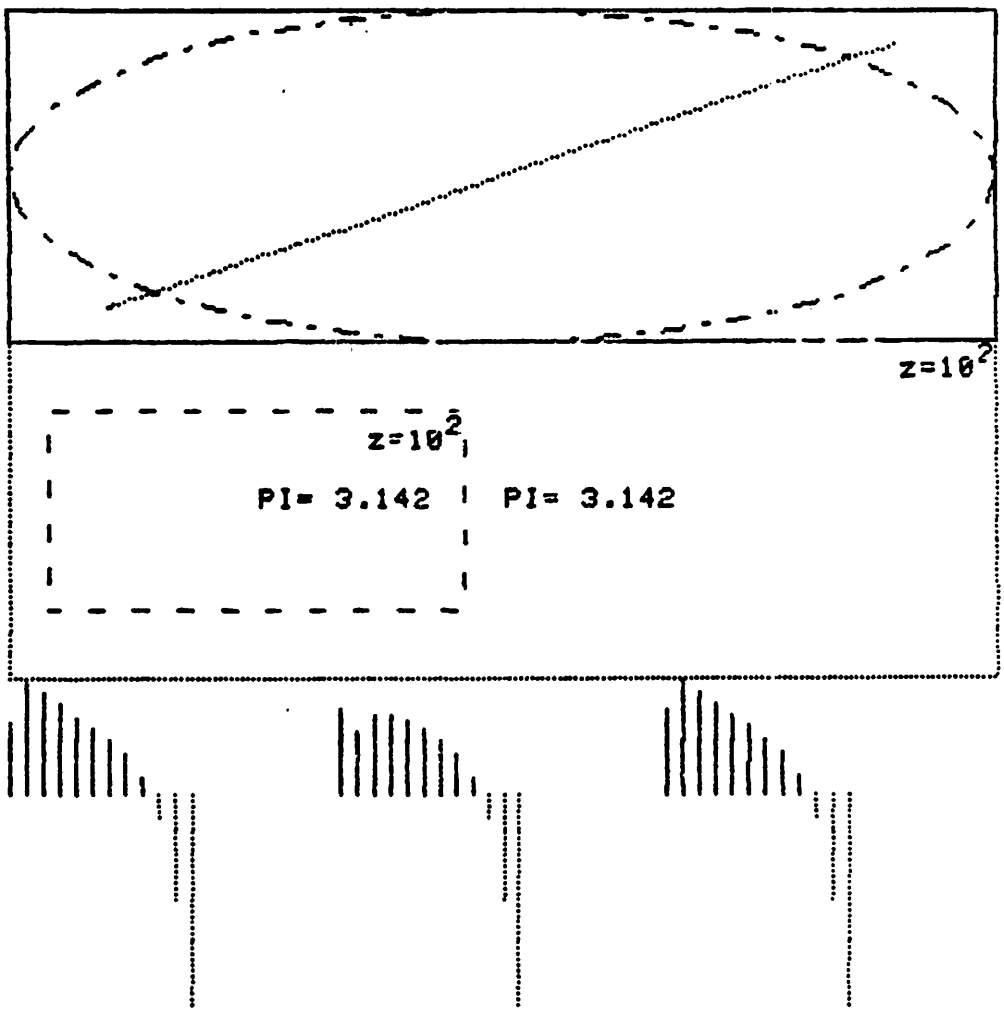


Figure 4.6. Graphics Output From a Sequence of Low-Level Graphics Commands.

this purpose, we extend the Newton-Raphson example procedure of section 4.2.7 to produce a 3-bar bar chart each iteration. The bar chart contains the log base 10 of each of the two function values f_1 and f_2, and the norm of the vector function $||f||$. We repeat this procedure below with the added lines shown in **bold** type:

```
procedure Newton (x)  {
   array x(2), f(2), J(2,2)
   array bars(3)
   shiftval = 0
   repeat  {
      Func (x,f)                    # Get function value.
      bars(1) = log10( abs(f(1)) )
      bars(2) = log10( abs(f(2)) )
      bars(3) = log10(||f||)
      barplot bars white sky shift=shiftval -3
      shiftval = shiftval + .05
      Jacobian (x,J)                # Get Jacobian.
      printf 'x = %-12.5r %-12.5r  '  x(1)  x(2)
      printf 'f = %-12.3r %-12.3r  ||f|| = %.3r/n'  f(1)  f(2)  ||f||
      matop Jinv   = inv (J)        # Get Jacobian inverse.
      matop deltax = Jinv * f       # Compute Newton step.
      matop x = x - deltax          # Update unknown vector.
      }
   until ( ||f|| <= 1.0e-14 )       # Repeat until small norm.
   }
```

Local variable *shiftval* above is incremented by .05 after each iteration and used as the *shift* argument on the *barplot* command to allow a maximum of 1/.05=20 different parallel bars. The *-3* threshold argument is the base of the bars drawn and causes the bars to be color *white* when they are greater than -3 (i.e., f_1, f_2, or $||f||$ greater than 10^{-3}), color *sky* when below -3. We place the bar charts in our bottom window and remember to set the world coordinate bounds. The initial guess used was determined experimentally to produce somewhat "wildly" varying function values:

```
1> window bottom        # Enter bottom window.
1> world 0 -14db 1 3db  # Set world coordinate bounds.
1> x(1) = .1            # Set initial guess.
1> x(2) = -5
1> Newton (x)
```

```
x =  .100000      -5.00000    f =  5.010       2.490e+1    ||f|| =  2.540e+1
x =  2.70813e+1    5.40625    f =  7.280e+2    2.146       ||f|| =  7.280e+2
x =  1.36351e+1    5.11837    f =  1.808e+2    1.256e+1    ||f|| =  1.812e+2
x =  6.97941       4.41363    f =  4.430e+1    1.250e+1    ||f|| =  4.603e+1
x =  3.72504       3.28495    f =  1.059e+1    7.066       ||f|| =  1.273e+1
x =  2.14772       2.12480    f =  2.488       2.367       ||f|| =  3.434
x =  1.39969       1.39959    f =  .5595       .5592       ||f|| =  .7911
x =  1.08878       1.08878    f =  9.667e-2    9.667e-2    ||f|| =  .1367
x =  1.00669       1.00669    f =  6.739e-3    6.739e-3    ||f|| =  9.530e-3
x =  1.00004       1.00004    f =  4.422e-5    4.422e-5    ||f|| =  6.253e-5
x =  1.00000       1.00000    f =  1.955e-9    1.955e-9    ||f|| =  2.765e-9
x =  1.00000       1.00000    f =  0.000       0.000       ||f|| =  0.000
```

As mentioned above, the bar charts are shown in the bottom window in figure 4.6. The left group of bars are for f_1, the middle for f_2, and the right for $||f||$.

For plotting graphs, DELIGHT provides the means of plotting arbitrary expressions versus one or two parameters (variables). With the *plot* command up to 9 "y-value" expressions versus a single parameter can be plotted on the same labeled axis. The axis is scaled to the minimum and maximum automatically and nice, "whole-number" axis labels are printed. The syntax of the plot command is:

```
plot Y1 [Y2 ... Y9] vs X from F to T [ {by    } I ]
                                     {times }
                                     {oct   }
                                     {dec   }
                                     {log   }
```

where everything after *to T* is optional. One of the clauses introduced by *by*, *times*, *oct*, *dec*, or *log* may be chosen and have the same meaning as they did for the *for_every* statement of section 4.4.2.2. If the optional increment keyword and *I* are not given, *by 1* is assumed. For *times*, *oct*, *dec* and *log* the x-axis of the graph is logarithmic.

Plots generated by the *plot* command may be targeted for black and white or color terminals. For black and white terminals, the curves drawn can have little triangle and square identifiers placed on them to help identify which curve goes

with which y-variable expression. This feature is obtained by typing *use <gpidents>*. (The triangular brackets surrounding the filename mean "look for this file in a standard place in the file system"; see [110]. Also, recall that *use* is defined to be *include*.) To have the curves drawn in color, *use <gpcolors>* is typed. (As mentioned before, on some black and white terminals such as the HP2648a, colors are simulated using various dashed line-types and intensities).

The following terminal session demonstrates the *plot* command:

```
1> viewport 0 0 1 1
1> use <gpidents>
```

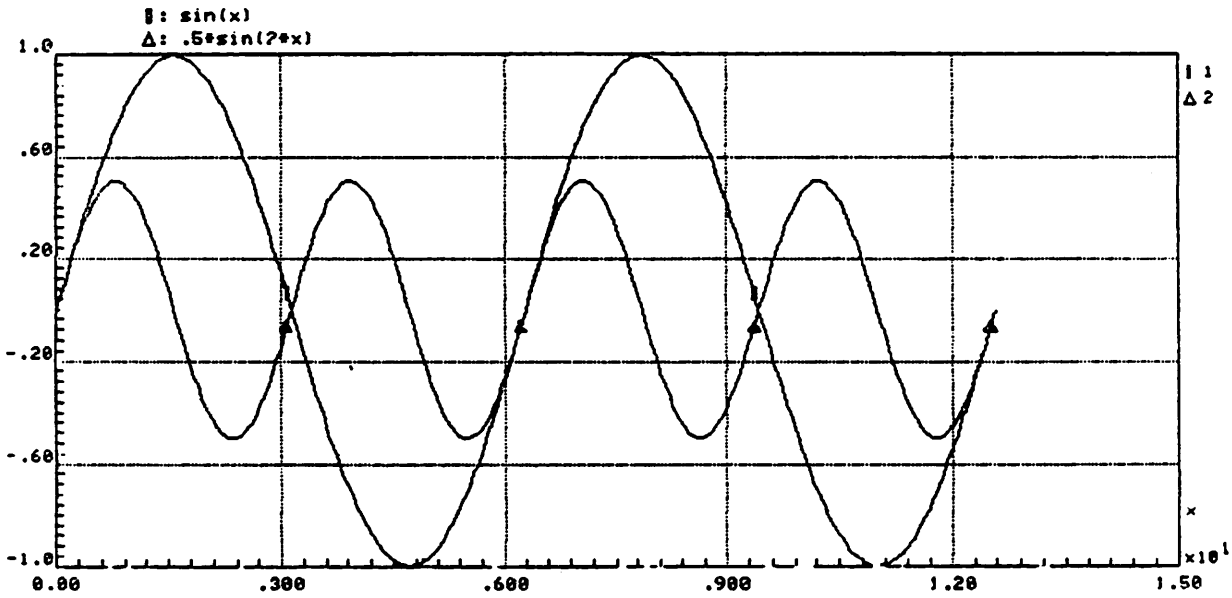


Figure 4.7. Graphics Output From First *plot* Command Example.

```
1> plot sin(x) .5*sin(2*x) vs x from 0 to TWOPI*2 by TWOPI/100
------ Compiling plot loop ------
1> plot (1/sqrt( freq**2 + 1 )) vs freq from .01 to 100 dec 20
------ Compiling plot loop ------
1>
```

The output plots for these commands are shown in figures 4.7 and 4.8 for the HP2648a black and white terminal (copies made on the HP2631G printer). Note that the x-axis of the plot in figure 4.8 is logarithmic due to the *dec* increment keyword above.

The *scat* has syntax identical to the *plot* command except that there must be at least two y-variable expressions. The last expression given provides the x-data that corresponds to the y-data provided by the other y-variable expressions. The *scat* produces a scatter plot of these sets of data-point pairs on a

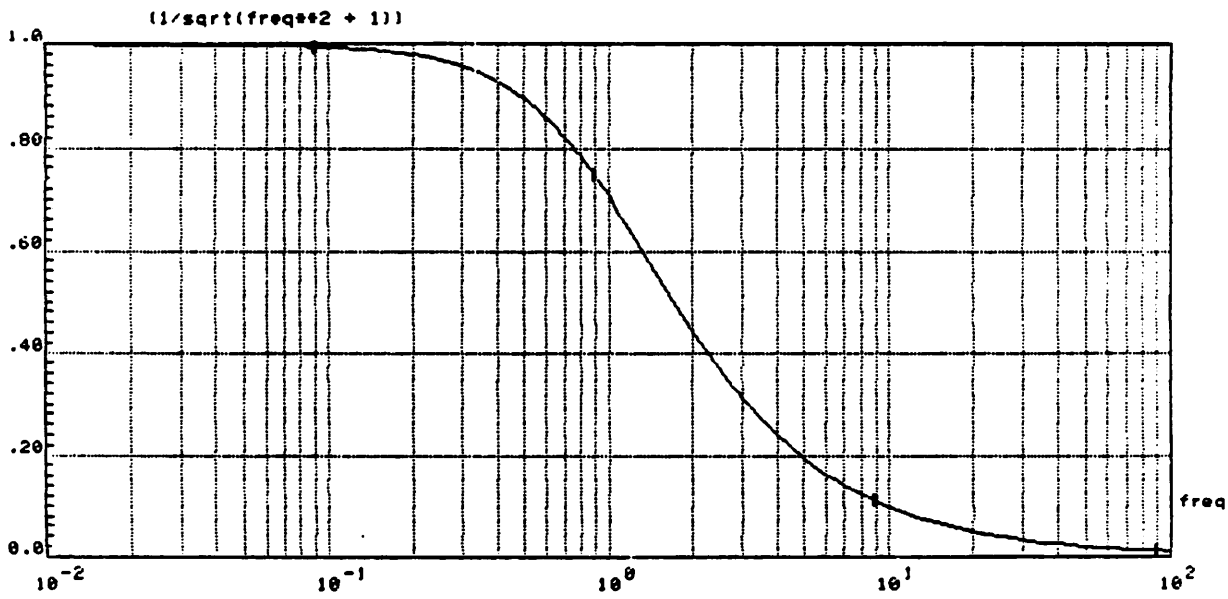


Figure 4.8. Graphics Output From Second *plot* Command Example.

labeled axis, i.e., the points are not connected as they are in *plot* output. Although a scatter plot is usually for random data such as from the two arrays in the command *scat ydata(k) xdata(k) vs k from 1 to 100*, we can demonstrate the *scat* command with the following:

```
1> scat sin(x) .5*sin(2*x) cos(x) vs x from 0 to TWOPI by TWOPI/50
------ Compiling scat loop ------
```

The resulting scatter plot is shown in figure 4.9.

For graphing a single expression vs two parameters on a 3-dimensional set of axes, the *plot3d* command is used[33]. Its syntax is:

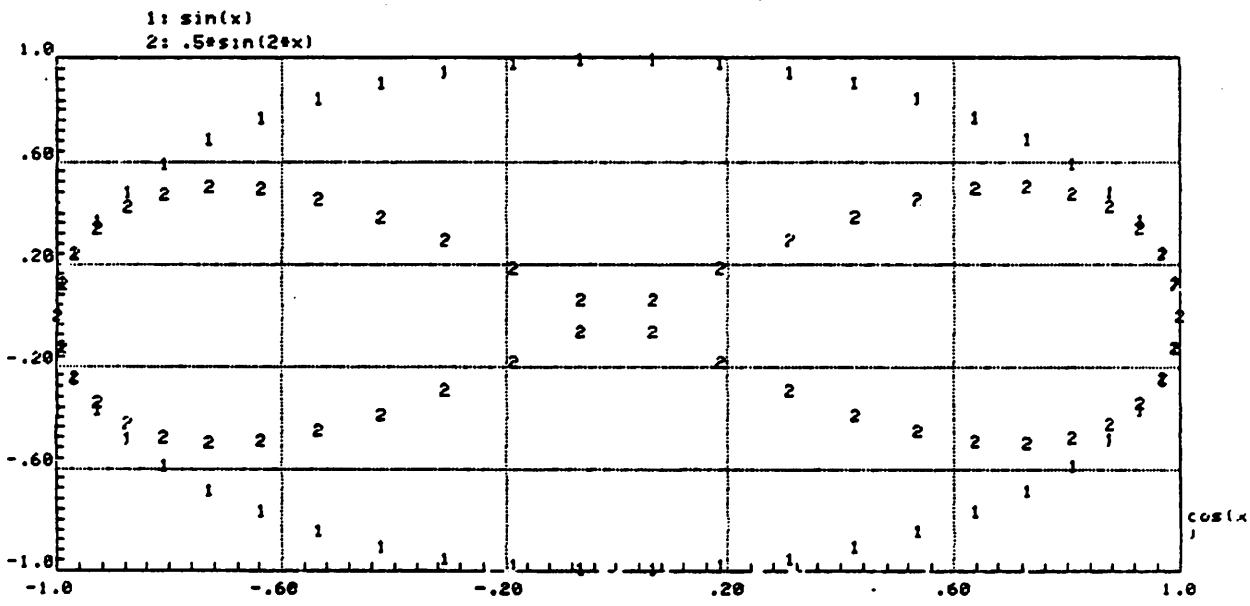


Figure 4.9. Graphics Output From *scat* Command Example.

[33] The original *plot3d* routine was written by L. Gelberg at Harris Semiconductor.

```
plot3d z_expr vs X from X_START to X_STOP by X_INCREMENT
              vs Y from Y_START to Y_STOP by Y_INCREMENT
```

where the command actually has to be typed on the same line unless an expression argument is continued onto the next line by having an open parenthesis character "(" end the line. (This is related to the statement continuation rules for RATTLE presented in section 4.2.5.) The following is a simple example of the *plot3d* command:

```
1> plot3d sin(x)*cos(y) vs x from 0 to 9 by .5 vs y from 0 to 6 by .5
Size:  21 x 13
```

The output plot for this command is shown in figure 4.10 for the HP2648a black and white terminal.

Both the *plot* and *plot3d* commands may be interrupted by pressing the special interrupt key on the terminal once, i.e., by generating a soft interrupt (see section 4.2.6).

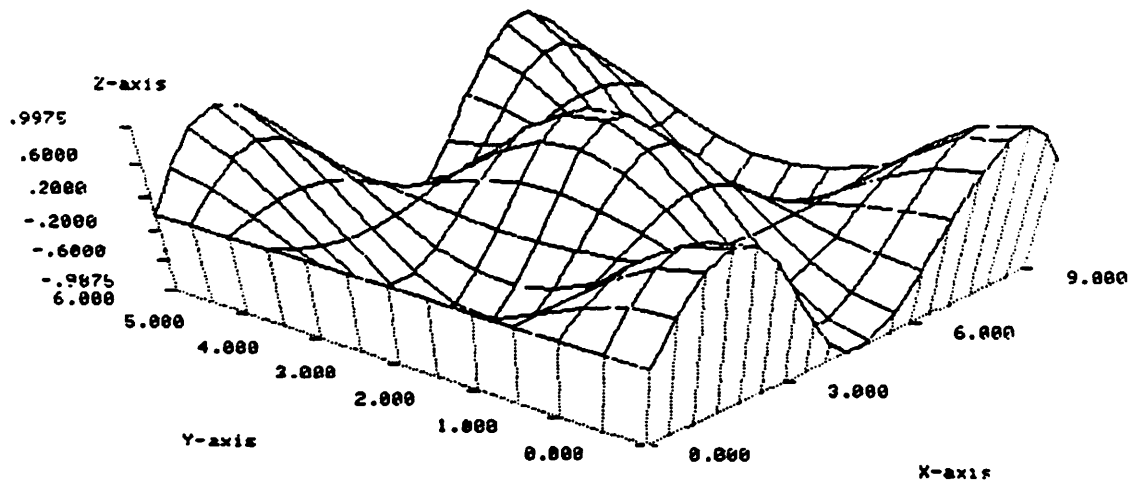


Figure 4.10. Graphics Output From *plot3d* Command Example.

4.6.2. Graphics for Observing Problem Performance

Commands such as *curve*, *barplot*, *plot* and *plot3d* presented in the previous section can be used by a designer to display graphically his problem performance at any time. These could be, for example, the response of his system versus time, frequency, loading, temperature, pressure, etc. In this section we introduce a new graphical display that allows a designer using the multiobjective problem formulation and the phase I-II-III optimization algorithm to grasp quickly the performance tradeoffs of his design. We also include a subsection on using this graphical display to perform tradeoffs.

During the progress of a multiple objective optimization computation, it is very desirable to have a display of objective and constraint values at each iteration which facilitates subjective evaluation of the design associated with that iteration. In DELIGHT this purpose is served by the *Pcomb* performance comb, a graphical display which shows the designer how close each of his multiple objectives and soft constraints are to their corresponding good and bad values. Since from our experience most designer interaction with DELIGHT is spent making tradeoffs in phase II of the phase I-II-III algorithm, hard constraints are not displayed. However, if any are violated in phase I, the message - *A Hard Constraint is Violated* - is printed at the top of the *Pcomb* display.

Referring to figure 4.11, the display consists of a vertical good line to the left and a vertical bad line to the right. On a color terminal, these are drawn in green and red, respectively. For all terminals, *G* appears above the good line while *B* appears above the bad line, as shown. Each objective or soft constraint is easily identified by its name as given in the problem description files and is displayed by two horizontal bars or *teeth* on the comb, one for the previous

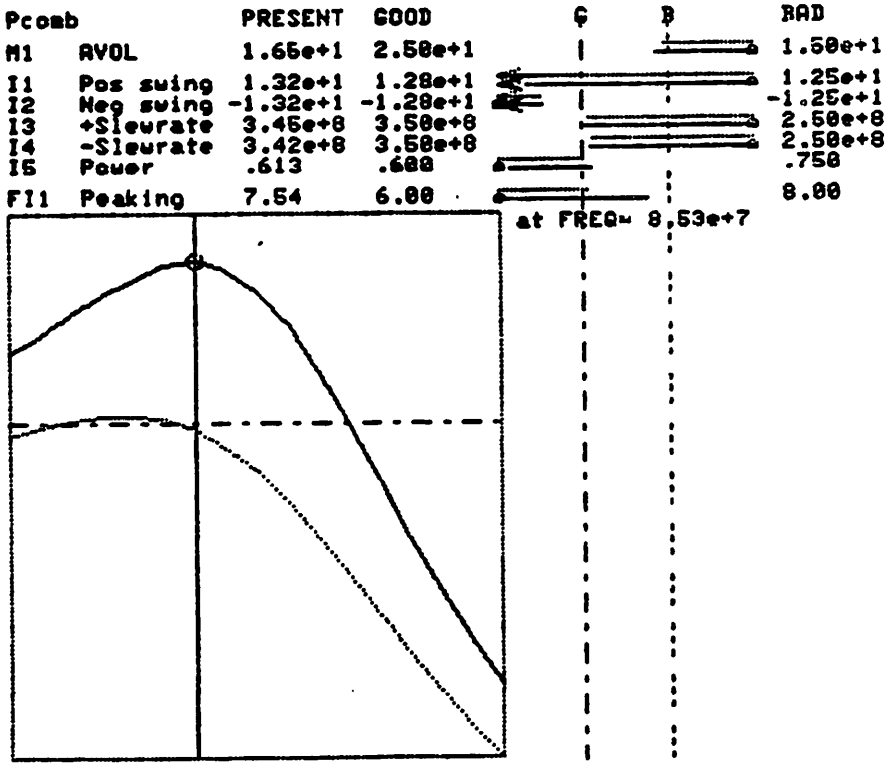


Figure 4.11. Example *Pcomb* Performance Comb.

comb drawn and one for the current comb. The previous comb teeth are in a lighter color or shade (and each slightly above the current teeth). The goal of the optimization algorithm is to move the tips of all the comb teeth to the left (in the direction of the good line). By minimizing the maximum of the normalized objectives and constraints, the optimization algorithm in effect insures that the rightmost tip after an optimization iteration is to the left of the rightmost tip before the iteration, even if some tips move slightly to the right, i.e., the performance of their objectives or constraints becomes worse.

The tip of each tooth is on the opposite end from a small diameter circular dot. The dot always is on the side of the smaller numeric value of the objective or constraint. Thus, if as for ordinary constraint I5 in figure 4.11, the dot is on the left, the good value is smaller than the bad value as for an objective being minimized or a constraint which must be less than its good value; a comb tooth which moves to the left toward the good line decreases its objective or constraint value. If as for ordinary objective M1 in figure 4.11, the dot is on the right, the good value is larger than the bad value as for an objective being maximized or a constraint which must be greater than its good value. In this case, a comb tooth which moves to the left toward the good line increases its objective or constraint value, as desired.

If an objective or constraint value is such that the tip of its comb tooth should be drawn off the *Pcomb* display, an arrow is drawn to show that the tooth is out of the comb range. The present values of inequality constraints I1 and I2 in figure 4.11 are both better than values for which their corresponding comb teeth can be plotted on the comb and thus they both have arrows. Note that I1 large is good since it is a ">=" constraint while I2 small (actually large negative) is good since it is a "<=" constraint.

Also shown on the *Pcomb* display are the actual numeric values of the objectives and constraints (the worst value is printed for functional) and, for each functional objective or constraint, a small plot of the actual objective or constraint, its good curve, and its bad curve. Each of these plots is versus the corresponding "W" variable as it varies from the FROM_VAL to the TO_VAL specified on the *for_every* line in the "FM" or "FI" file (see section 4.4.2.2). To the right of each plot is shown the value of the "W" variable at which the corresponding functional objective or constraint takes on its worst value. The position of this value of the "W" variable is shown by a big circular dot on the functional plot.

The performance comb may be output automatically during each optimization iteration (using e.g., *run 3 Pcomb*) or manually after, say, adjusting the good or bad values for a particular objective or soft constraint. Since the comb display shows the previous comb teeth as well as the present ones, the designer can easily see the results of such an adjustment of good or bad values as well as the improvement made by an optimization iteration.

It is important to realize that the comb shows the *previous* teeth from the *previous* comb drawn and not from the preceding optimization iteration: if two combs are drawn directly in succession, the corresponding previous and present teeth on the second comb will be *exactly* the same. This is important since if an entire comb on the graphics screen is accidentally erased, it cannot be redrawn with the correct previous teeth.

The performance comb may be requested at any time simply by typing *Pcomb* provided the user is working on one of the graphics terminals supported by DELIGHT graphics. Additional information about *Pcomb* features may be

found in the design examples of section 5.1.4.

Using the *Pcomb* to Perform Tradeoffs. Tradeoffs between competing objectives or constraints are explored by adjusting good and bad values after best (or near best) performance of the system being designed has been achieved following several iterations of optimization. Basically, after several optimization iterations have been carried out with a set of good and bad values, the designer displays a performance comb and decides whether he is happy with the present values of his objectives and constraints. (In a tightly constrained design problem most of the algorithm execution will probably be spent in phase II. Recall from section 4.5.2.2 that in phase II objectives and constraints are competing equally to be improved by the algorithm and the meaning of the good and bad values applies consistently to both.) If he is not happy with the present performance he adjusts good or bad values to reflect his feelings and resumes optimization. Commands to make these adjustments are detailed later in this section.

For example, suppose in an electronic circuit design problem that DC power is a performance objective and it has been given good and bad values of 30mw and 50mw, respectively. Suppose that at the current iteration of the optimization, the power is 40mw. For these values, the associated comb tooth would end exactly half way between the good and bad vertical lines. Suppose that the designer is unhappy with the way several objectives (and/or constraints) have traded off and he actually wants to reduce the power further, at the expense of other objectives. This means that he now considers the value 40mw to be worst than he did previously relative to other objectives. This means that the DC power objective bad value should be closer to 40mw than it is presently. Thus, setting the bad value to, say, 45mw or even 40mw is the proper action. He then redisplays the comb by typing *Pcomb* and runs a few more optimization

iterations by typing, for example, *run 5*. (An alternative approach would be to decide that 40mw is not as good as thought previously and therefore the good value of 30mw should be reduced to, say, 20mw or 25mw. Which approach to use is up to the designer[34].)

To modify good or bad values, the commands *setgood* or *setbad* are used and have the following format:

```
setgood TYPEn = VALUE
setbad  TYPEn = VALUE
setgood FTYPEn = EXPRESSION_VS_W
setbad  FTYPEn = EXPRESSION_VS_W
```

where TYPE is:

M	for	ordinary objectives,
I	for	ordinary inequality constraints,
Up	for	upper soft box constraints on design parameters,
Lo	for	lower soft box constraints on design parameters,

FTYPE is:

FM	for	functional objectives,
FI	for	functional inequality constraints,

n is the number, and EXPRESSION_VS_W is any expression containing the original "W" variable for the particular functional objective or constraint. The given expression becomes the new good or bad curve. For objectives or constraints, *n* is the same number given after *objective* or *constraint* in the problem

[34] That the two alternative approaches of either adjusting the bad value or the good value are not exactly equivalent can be seen in the objective/constraint normalization formula given in equation (4.1) of section 4.4.1.2. The approach to use should be based on the uniform satisfaction/dissatisfaction rule of that section.

description files. For design parameter soft box constraints, n is 1 for the first *design_parameter* statement in the "S" file, 2 for the second, etc. The correspondence between n and the design parameters may also be obtained from the *Pcomb* display or from the output of the *printdp* command, which prints each design parameter number, name, and present value. Examples of *setgood* and *setbad* commands are:

```
1> setgood MS=35e-3
1> setbad  Up8 = 80volts
1> setbad  FMS = (1 + FREQ**2)
```

More on these and other optimization-related commands may be found in the *DELIGHT.SPICE User's Guide* [114]. In particular, another important command is *setvariation*. Using the format

```
setvariation PARNAME = VALUE
```

a user can set the nominal variation of a particular design parameter to the value of an arbitrary expression. After an optimization has started, a designer can use the *setvariation* command to update his idea about what change in a parameter should influence the objectives and constraints to the same degree, thus possibly enhancing the speed of convergence of the run.

4.6.3. Graphics for Observing Algorithm Performance

In this section we introduce several graphical displays that allow a user running certain of the Library optimization algorithms to see quickly the current behavior of the algorithm. Just as for graphics to observe problem performance in the previous section, a user can easily construct his own (or modify existing) displays using any of the high or low-level graphics commands of section 4.6.1.

Graphical displays to show algorithm behavior fall naturally into classes

associated with each major sub-block of an algorithm. Since most algorithms contain sub-blocks for *direction* and *stepsize*, graphics procedures have been developed for them. For the former, the *gradient clock* display has proven to be quite effective in showing the quality of the computed search direction and in guiding the user to remedy a poor search direction.

The *gradient clock* is a circular display appearing like a clock that shows the angles between the computed search direction **h** and the other gradients that take part in the computation of **h**. For the Phase I - Phase II algorithms these gradients are of the cost and the active constraints; for the phase I-II-III algorithm these gradients are either of the hard constraints in phase I or of the active objectives and constraints in phases II and III. Active objectives were defined using equation (4.12) in section 4.5.2.2. Recall from the discussion of basic feasible descent algorithms in section 4.5.1.3 that small movement along direction **h** should decrease the cost while remaining in the feasible set. One way mentioned to guarantee this is to have **h** "point away" from these gradients by more than 90°. Usually, the greater the angles, the better the algorithm performs. Hence, these angles are an indicator of the quality of the search direction.

Referring to figure 4.12 we see that the search direction **h** is always drawn vertically "at twelve o-clock noon". Since the dimension of the vector space in which the gradients lie is *Nparam*, the number of design parameters, the actual locations of the gradients with respect to each other and **h** cannot be drawn correctly for greater than two or three dimensions. Instead, the gradient clock shows the correct angle individually between each gradient drawn and the vertically drawn search direction. Each gradient can be drawn on either the left or the right of the clock and still show the same angle (always $\leq$180°). To avoid

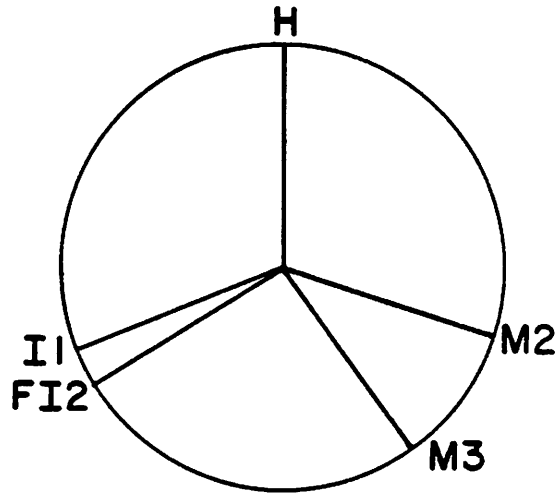


Figure 4.12. Example Gradient Clock Display.

congestion, gradients of the cost or objectives are arbitrarily drawn on one side, of the active constraints on the other. This is shown in figure 4.12.

How the gradient clock guides an advanced user in remedying a poor search direction is as follows. When the angles between the search direction **h** and the other gradients shown are badly unbalanced due to poor problem scaling, the user can restore a certain amount of balance by adjusting certain algorithm *push factors*. Push factors usually affect **h** by weighting certain gradients passed to the quadratic program for determining the closest point to the convex hull of the gradients involved (see section 4.5.1.3). For example, when discussing the phase I-II-III algorithm in section 4.5.2.2, we mentioned the existence of a push factor by which the user can emphasize as a group the objectives and the soft constraints in phase II. There is also a push factor parameter which provides similar emphasis during phase I.

For displaying the behavior of step-size procedures, which objectives or constraints are limiting the step-size along the search direction can be displayed with the *Scomb* "step-size comb" for the multiobjective phase I-II-III algorithm. For other Armijo-like step-size sub-blocks, a bar chart can be displayed that shows the impeding constraints and also the precise action of the Armijo step-size rule during the inner trial-step-size loop.

For designers working with the multiobjective phase I-II-III algorithm, the *Scomb* step-size comb can be displayed after each trial step-size in the inner loop of the *stepsize* block. It appears identical to the *Pcomb* discussed earlier except for the comb teeth. First, only one tooth is drawn for each objective or constraint; there is no notion of present and previous teeth. Second, objectives or constraints whose values are impeding the step-size along the search direction, i.e., that are failing a test that has to be satisfied such as the Armijo rule, have bright (or red) comb teeth. Light (or green) comb teeth indicate that the values of the corresponding objectives and constraints are acceptable, i.e., pass the test. All *Scomb* teeth must be light or the trial step is rejected. If a particular tooth remains bright until a very, very small trial step is tried, a user should be suspicious of inaccuracies in the gradient calculation of its associated objective or constraint since $\mathbf{h}$ is probably not a descent direction. For a demonstration of the *Scomb*, see the *DELIGHT.SPICE User's Guide* [114].

For any optimization algorithm that contains an Armijo-like step-size sub-block, there is a useful three part graphical display that can be output directly after each trial step-size in the inner Armijo loop. An example of this *Garmijo* display is shown in figure 4.13. The top of the screen shows a geometric interpretation of the Armijo test given in equation (4.8) and used in procedure *stepsize* for the simple unconstrained cost algorithm of section 4.5.3.1 The

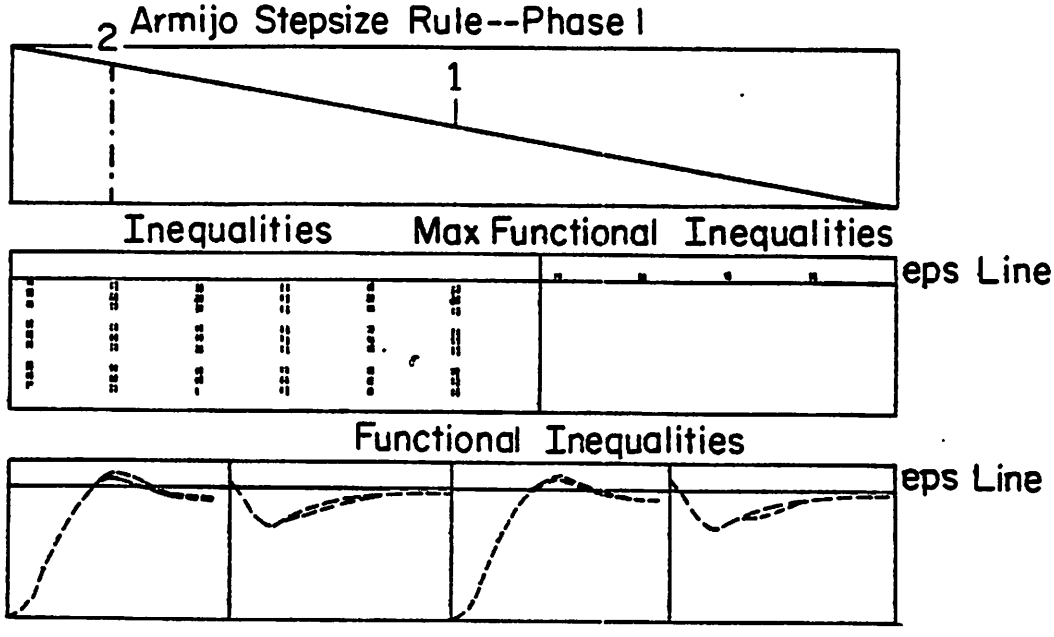


Figure 4.13. Example Armijo Step-size Rule Display.

horizontal axis shows the step-size along the present search direction while the vertical axis shows the scalar quantity involved in the test. Plotted on these axes are the Armijo test line (with slope given by sub-block parameter *Alpha*—see algorithm segment (4.9)) and a vertical bar for each trial step. Each bar is labeled with k in the step-size quantity $Beta^k$ .and reveals, by being below the test line, that the scalar quantity involved in the Armijo test indeed satisfies the test. This display is helpful since it can be used (by advanced users) to adjust parameters *Alpha* and *Beta* to avoid doing too much computation in procedure *stepsize*. For example, *Beta* (default 0.5) can be decreased to cause small steps to be tried more quickly, i.e., at smaller values of k.

The middle part of the *Garmijo* display shows two bar charts using the *barplot* command presented in section 4.6.1. Referring again to figure 4.13, each group of bars on the left is for one of the *Nineq* ordinary inequality constraints (in this case there are six). Within each group, the leftmost bar is for the first trial step, the next bar to the right for the second trial step, and so on. Similarly, the groups on the right are for each of the *Nfineq* functional inequality constraints; they correspond to the *Nfineq* small functional plots shown at the bottom of the screen and produced by the *curve* command presented in section 4.6.1. On the right bar chart, each bar shows the maximum over its functional constraint samples. The epsilon line, labeled *eps Line*, shows which constraints are considered $\varepsilon - active$. Both the *barplot* and the *curve* commands used here supply their top and bottom color and their threshold arguments so that the color of the bars and curves change to red when above the epsilon line. Thus, the user's attention is drawn to ε-active constraints. Whereas the top part of the *Garmijo* display shows if the cost is impeding the trial step, the middle and bottom parts show which constraint violations act as impediments (if any).

4.7. Simulation Interface

We have already covered three of the four major interfaces alluded to in section 4.5 and shown in figure 4.5: the problem description facilities, the coupling between an optimization problem and algorithm via the *solve* command and *problem interface*, and the graphical and non-graphical means used by DELIGHT to inform a user about about his problem or algorithm performance. This section addresses the final interface provided by DELIGHT—between a problem description and a necessary computer simulation—for versions of DELIGHT that are coupled to simulation programs.

The basic idea of this interface is to allow problem description expressions to depend on simulation results from an engineering system being designed instead of just depending directly on design parameters as was the case up to now. By viewing, as in section 4.4.1, a simulation program as a mapping from the design parameter vector $\mathbf{x}$ to the simulation response functions $\mathbf{v}(\mathbf{x})$, we thus allow the problem expressions to be indirectly a function of $\mathbf{x}$ by being a function of the system response functions $\mathbf{v}(\mathbf{x})$ as well as directly a function of the design parameter vector $\mathbf{x}$. The bold ambitions of the interface thus emerge: to allow an optimization process to easily send design parameter values to and retrieve output responses from a simulation program.

We begin in section 4.7.1 by presenting the functions and goals of the *simulation interface* and give a general theme that runs throughout all implementations. To meet the goals presented in section 4.7.1, part of the simulation interface implementation is independent of the particular simulator while part is (very) dependent on it. These two parts are the subjects of sections 4.7.2 and 4.7.3.

4.7.1. Functions and Goals

The overall function of the DELIGHT *simulation interface* is to allow the optimization of design problems whose objective and constraint values depend on the results of a computer simulation. For the most part, simulation programs usually have input parameters, outputs, simulation run controls such as the final time-domain simulation time, and options such as those for controlling integration method or accuracy, all of which can be chosen or set by the program user. Also, there is usually an initialization phase for describing the structure of the system being simulated. To allow (a) some of the simulator input

parameters to be design parameters, (b) others to be *settable parameters*, that is, input parameters that can be set to bring about different system test conditions but are not adjusted automatically by the optimization, and (c) some of the simulator outputs to be used in a problem description, requires that (1) optimization algorithms and problem description procedures be able to set the input parameters and (2) problem description expressions be able to retrieve the output values from the simulator. Of course these two requirements must be performed without user intervention (assuming the design parameters, settable parameters, and outputs have already been identified). The other three features of most simulators—run controls, options, and initialization—need to be interactively under control of the DELIGHT user.

All of the above requirements are handled by the simulation interface. Its function is broken into six broad categories having to do with:

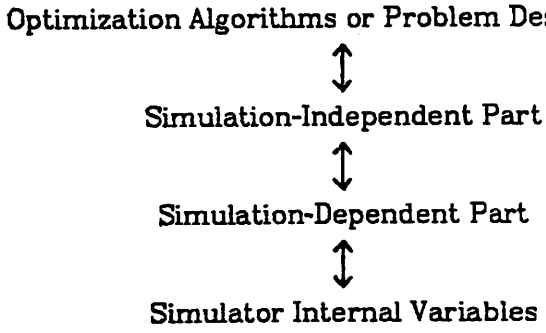
1. declaring design parameters and setting their values,
2. declaring settable parameters and setting their values,
3. declaring outputs (responses) and obtaining their values,
4. initializing the simulator,
5. setting simulator run controls and options, and
6. displaying user-requested simulator information.

In the next two sections, the various parts of the simulation interface are presented by category using the six categories above.

One of the original goals of DELIGHT is to have the simulation interface facilitate the use of DELIGHT in a variety of engineering design areas. To meet this goal, the interface must provide a mechanism that allows:

- DELIGHT optimization algorithms and other procedures to be completely independent of the particular simulator used, and
- different application-specific versions of DELIGHT to be easily created.

Each of these aims is handled as follows. The simulation interface is divided into a *simulation-independent* and a *simulation-dependent* part. Optimization algorithms and, to some extent, user problem descriptions deal only with the independent part and thus are themselves independent of the particular simulator. The independent part, consisting mainly of RATTLE defines, macros, and procedures, then interacts with the dependent part of the interface, a set of built-in Fortran routines, each one of which carries out a particular well-defined task written for a particular simulation program. Moreover, the interaction between the independent and dependent parts occurs only through arguments passed to- and function values returned from- these Fortran routines. Finally, the dependent routines interact directly with the simulator. For example, in the case of a Fortran simulator, they write into or read directly from variables in its common blocks. The information transfer through the different levels of the simulation interface can be depicted as follows:



The next section presents the dependent part of the interface while section 4.7.3 presents the independent part.

An overriding theme that pervades the implementation of the simulation

interface is *to avoid doing at execution time anything that can be done at compile time*. The foremost example of this idea occurs in the implementation of the first three of the six interface requirements listed above. All of these have to do with the declaration of simulator quantities by name, followed by the setting or getting of their values. Instead of passing the name of the quantity at run time—an approach that would require the dependent interface routine to perform a search for the name through a list each time the quantity was dealt with—we break up each of the declaration/value-related tasks as explained next.

Consider, for example, the tasks of declaring settable parameters and setting their values. We first declare a settable parameter *P1* in a manner whose details need not concern us here. Its value can then be updated interactively using, for example, *set P1 = 37*. The lengthy run-time search approach would be to have the *set* macro simply push back something like *SettableUpdate ('P1',37)*; the *SettableUpdate* interface routine would first search for *P1* in the list of declared parameters and then store the value *37*. Our approach is to do the search only once, at compile time, by providing an interface routine pair: the first routine takes a settable parameter name and returns an item that we call a *pointer*; the second routine takes a pointer and a value and makes the parameter update rapidly. Thus, the *set* macro would execute the statement

```
ptr = SettablePointer ('P1')
```

and, supposing that the value of *ptr* returned was 101, then push back

```
SettableUpdate (101,37)
```

which would perform the parameter update very rapidly. For this purpose, the precise nature of pointers is undefined. Their only requirement can be seen in

the following "conversation" between independent interface macros such as *set* and dependent routines such as *SettablePointer* and *SettableUpdate*:

> "Excuse me sir, I'm going to hand you a name and you kindly return to me something we'll call a pointer. Now I'm not affected in the least by what a pointer actually is as long as when I hand it back to you along with a value, you can profit by it to make use of that value as quickly as possible."

The dependent interface routines are typically set up so that a pointer points to either the variable location used to store the value or to another data structure which itself contains information that can be used to make the update rapidly.

4.7.2. Simulation-Dependent Part

The simulation-dependent part of the simulation interface consists of a set of routines, each element of which carries out a well-defined task required to interface DELIGHT with a particular simulation program. These routines are presented in the following tables, classified according to the six functions listed in the previous section. Names below such as *ClearParameters* are fictitious names that are used in this dissertation for clarity; under each such name, in parentheses, is the actual Ratfor name. (The Ratfor names are very short due to the Fortran six-character limit.) Arguments and other details are not given below; for more information, see [110].

First, routines that deal with design parameters:

Declaring Design Parameters and Setting Their Values	
DesignParameter (despar)	Declare the given simulator input parameter as a design parameter, returning a success or failure status flag.
ParameterPointer (parptr)	Return the *pointer* for the given design parameter (its name is passed).
ParameterUpdate (parupd)	Update the design parameter with given pointer to the value passed.
ParameterName (dpname)	Return the design parameter name associated with the given pointer.
ParameterValue (parval)	Return the present value of the design parameter associated with the given pointer.
ClearParameters (clrpar)	Clear all design and settable parameters, i.e, make it such that none is declared.

Routines *ParameterPointer* and *ParameterUpdate* act as an interface pair for run-time efficiency, similar to the example pair presented in the previous section.

For settable parameters, that is, simulator parameters that can be set to bring about different system test conditions but are not adjusted automatically by any optimization algorithms, we have the following routines:

Declaring Settable Parameters and Setting Their Values	
SettableParameter (stbpar)	Declare the given simulator input parameter as a settable parameter, returning a success or failure status flag.
SettablePointer (stbptr)	Return the pointer for the given settable parameter.
SettableUpdate (stbupd)	Update the settable parameter with given pointer to the value passed.

For simulation outputs, there are a few considerations. The purpose of declaring simulation outputs is to indicate to the simulator which data it needs to store when the actual simulation takes place. Such declarations are necessary since it is usually inefficient if not impossible due to memory storage considerations to store all the data that a simulator is capable of producing. Another consideration is how to name outputs. It is desired that designers refer to simulation output quantities in their problem description expressions in as much a problem-oriented manner as possible. This shields designers from having to get involved in the actual details of where the output quantities are stored or how they are transmitted.

As an example of a fairly difficult simulation interface to work with, consider Balling's [18] DELIGHT.STRUCT interface to the ANSR nonlinear structural response simulator. Although involving DELIGHT, this interface was implemented before the general simulation interface being discussed here was conceived. In Balling's interface, all response data versus time is stored in a large 1-dimensional array called *resp*. A user setting up an optimization problem has to be concerned with integer array subscripts that point to the start of where

various output responses are stored in *resp*. Moreover, since *resp* is 1-dimensional, these subscripts depend on how many time points are calculated in the time-domain simulation. Details such as these cause confusion and can be a source of error. Hence, the simulation interface should try to eliminate them from the matters a user must concern himself with. The simulation output aspects of DELIGHT have been designed with the above goals in mind.

Simulator output names accessible by any RATTLE expression are allowed to consist of a keyword part followed by an arbitrary list of arguments surrounded by parenthesis. For example, in the hypothetical output *Pressure(valve1)*, *Pressure* is the keyword while *valve1* is the one argument. Other possible output names are *vm(201)* for the magnitude of the node voltage at node 201 of an electronic circuit, *vm(3,4)* for the voltage across nodes 3 and 4, *displacement(level2,3)* for the displacement of node 3 on floor level 2, or *truss(AxialForce,16)* for the axial force on truss member 16. It is the task of the simulation-independent part of the interface to insure that all of the various output keywords for a particular simulator are set up as macros that gather their parenthesized arguments and perform other tasks discussed in the next section. In particular and most importantly, they must push back an appropriate function call to an interface routine that returns the value of the output.

One additional comment needs to be made about simulation outputs that are a function of some independent variable such as time or frequency. System personnel who actually interface DELIGHT to a particular simulator can choose one of two different routes. One is to include the independent variable such as *time* in the output arguments as in *voltage(101,time)* for the time-domain voltage at node 101. However, since this argument will *always* be the variable *time*, we can give RATTLE access to a Fortran variable (see appendix B.2) used in the

interface called, say, *TIME* and instead write *voltage(101)*; the value returned is understood to be at the present value of variable *TIME*. This is the approach taken in the DELIGHT interface to the SPICE circuit analysis program, addressed in section 5.1.

The interface routines that deal with simulation outputs are listed in the following table:

Declaring Outputs and Obtaining Their Values	
SimulationOutput (simout)	Declare the given simulator output so that it is accessible to any RATTLE expression, returning a success or failure status flag.
OutputPointer (outptr)	Return the *pointer* for the given output (its name is passed).
OutputValue (outval)	Return the present value of the output associated with the pointer given.
ClearOutputs (clrout)	Clear all outputs, i.e, make it such that none is declared.
OutputKeywords (okeywd)	Return the various output keywords for a particular simulator so they can be set up as macros (see section 4.7.3). The number of keywords is returned as the function value.

Routines *OutputPointer* and *OutputValue* also act as an interface pair for run-time efficiency except that a value is returned rather than a parameter value updated as for *ParameterUpdate*.

Routine *OutputValue* plays a special role in allowing RATTLE expressions to freely access simulation output values: if any design or settable parameters has

been updated since the last simulation, *OutputValue* must automatically run the required simulation before returning the output value. Since often a simulator can perform different kinds of simulations such as time-domain, frequency-domain, etc., the interface keeps a flag for each kind of simulation, indicating whether a new run is required or not. Each flag is set by *ParameterUpdate* or *SettableUpdate* if the parameter it is updating affects the corresponding kind of simulation. *OutputValue* then first checks these flags to determine if it must run another simulation.

The last three of the six function classes consist of the interface routines in the following tables:

Initializing the Simulator	
SimulationInit (sminit)	Run the initialization (setup) phase of the simulator on the given logical unit number. It has already been opened to the file that describes the structure of the system being simulated.

Setting Simulator Run Controls and Options	
SimulationRunControl (simrct)	For a given simulator run control, record the analysis information passed. This information can be either a scalar, or a vector as in the parameters *Tstart Tstop*, and *Tstep* that might be needed by the run control for *Time* in a time-domain simulation program.
SimulationOption (simopt)	Store the given value into the specified simulator option variable.

Displaying User-Requested Simulator Information	
SimulationDisplay (smdisp)	Display the user-requested simulator output from the command and optional arguments passed. For example, the independent part of the simulation interface converts the user command *display nodes 201 202* into the call *SimulationDisplay ('nodes', '201', '202').*

4.7.3. Simulation-Independent Part

The simulation-independent part of the simulation interface consists mainly of RATTLE defines, macros and procedures that are intended to simplify user control of and references to a simulator. As in the previous section, they are presented in the order of the six functions listed in section 4.7.1.

Declaring Design Parameters and Setting Their Values. The *design_parameter* declaration statement has already been shown for the classical and multiobjective problem formulations in sections 4.4.2.1 and 4.4.2.2. To declare design parameters that in particular are simulator input parameters, the *sim_design_parameter* statement is used and has exactly the same syntax. In an optimization problem involving a simulator, however, ordinary design parameters can still be declared using *design_parameter*. Thus, design problems can be formulated using both types of design parameters. Besides storing away the parameter variation, min and max bounds, etc., *sim_design_parameter* also calls simulation-dependent routine *DesignParameter* to declare the parameter and prints an error message if the status flag returned indicates an error.

The *set* command, already mentioned in regard to the "P" multiobjective problem description file in section 4.4.2.1, is used to set the value of simulation

design parameters, ordinary design parameters, and settable parameters and has the following format:

```
set NAME = VALUE
```

It takes the parameter name and passes it in succession to (1) a RATTLE routine for ordinary design parameters, (2) dependent routine *ParameterPointer*, and (3) dependent routine *SettablePointer*. If any of these returns a valid pointer, say *101*, then *set* pushes back the appropriate entry in the following list:

```
(1)   X(k) = VALUE
(2)   { X(k) = VALUE ; ParameterUpdate (101,VALUE) }
(3)   SettableUpdate (101,VALUE)
```

where *k* is the **X** index of the design parameter. Obviously for *set* to work properly, all the various parameter names must be unique.

From the discussion so far, it would seem that the only way to update simulation design parameters is using *set*. Hence, one may ask, "How are these parameters updated automatically during the course of an optimization run, considering that *set* statements need not be placed in any of the problem description functions?" The answer to this question concerns an additional task performed by the *prob_function* statement used throughout section 4.4.2.2[35]. Hidden from the user, *prob_function* loops over all simulation design parameters calling *ParameterUpdate* for each. In this way neither the user nor optimization algorithms need be concerned with keeping the simulator input parameters up to date.

Declaring Settable Parameters and Setting Their Values. Settable parame-

[35] To use the classical problem formulation with simulation design parameters requires the word *function* appearing throughout section 4.4.2.1 to be replaced by the word *sim_function* in order to carry out this task.

ters are declared by including the statement

```
settable_parameter PARNAME
```

for each in the "S" file and are set using the *set* command as explained above.

Declaring Outputs and Obtaining Their Values. Simulation outputs are declared by including for each the statement

```
simulation_output OUTPUT_NAME
```

in the "S" file. After such a declaration, *OUTPUT_NAME* can be used in any RATTLE expression, in particular in those that describe the problem to be optimized.

The simulation output mechanism works only if *OUTPUT_NAME* begins with one of the simulator output keywords, e.g., *Pressure* or *vm* from the examples in the last section. These keywords have been returned (during a setup process by systems personnel) by a one-time call to the dependent routine *OutputKeywords*. At that time, each of the keywords was defined to be a macro which gathers its following arguments and performs other tasks. For example, if *vm(3,4)* appeared in an expression, macro *vm* would do the following:

1. gather the arguments *3,4*,
2. pass 'vm(3,4)' to *OutputPointer* to get back the pointer, say 101, and
3. push back *OutputValue(101)*.

If the arguments *3,4* were invalid, i.e., not of a previously declared output, an error message would be printed on the screen by *vm*.

Initializing the Simulator. The simulation program is initialized using the statement

```
sim FILENAME
```

in the "S" file where *FILENAME* is the name of the file containing the structural description of the system to be simulated. *sim* opens the file to a Fortran logical unit number and passes that number to the dependent interface routine *SimulationInit*. If any errors occur during the initialization, *sim* prints an error message on the screen.

Setting Simulator Run Controls and Options. A simple way of controlling the run controls necessary for simulations that range over an independent variable is using the *sweep* command. It normally is included in the "S" file and has the following format:

```
sweep VARNAME from FROM_VAL to TO_VAL {by   } INC_VAL
                                      {times}
                                      {oct  }
                                      {dec  }
                                      {log  }
```

Usually, *VARNAME* is the name of an independent variable such as time, frequence, temperature, etc. The *sweep* command gathers the numeric arguments and the increment keyword and passes them all to dependent routine *SimulationRunControl*. As usual, any errors encountered are printed on the screen.

Simulator option variables may be interactively set at any time with the *option* command, having format:

```
option NAME = VALUE
```

This command simply passes the name and value to interface routine *SimulationOption*. No interface pair is needed in this case since option changes occur rarely.

Displaying User-Requested Simulator Information. Important output generated by a simulator (and perhaps normally sent to a line printer) can be displayed on the terminal screen at any time by using the *display* command. It has format:

```
display ITEM [ ARG1 ARG2 ... ARG6 ]
```

The item to display is followed by from one to six specific items. Each of the six arguments can be made to describe classes of items through the use of the *magic characters* "*" and "?". "*" represents a match of zero or more of any character, e.g., appearing alone it matches any entry. Thus, in electronic circuit design, the command *display element R2** might display all circuit resistors whose names started with *R2*. "?" matches any single character in a name. Thus, for example, *display model* ??*X* ??*Y* might display all system models whose names contained three characters and ended with either of the letters *X* or *Y*. Of course, arguments to *display* need not contain any magic characters.

Other dependent routines shown in the tables of the previous section that were not discussed here have to do with other less interesting functions of the interface. See the *DELIGHT Reference Manual* [110] for additional information.

4.8. Miscellaneous DELIGHT Features

This section covers various additional commands and features of DELIGHT. First of all, there are several features for meeting the DELIGHT goal of providing test and debug aids for the RATTLE interactive programming language. These include (1) the ability to display DELIGHT entities such as variables, arrays, defines, etc., and procedure execution times and call counts, (2) the ability to *enter* procedures and display or work with local entities, and (3) the ability to

turn on and off the echo of lines from a file being *included*.

The *display* command for displaying various DELIGHT entities has the following format[36]:

```
                                  { arrays     }
display [ local  ] { defines    } [ ITEM ]
        [ system ] { functions  }
                   { macros     }
                   { procedures }
                   { variables  }
```

The keywords *local* and *system* are optional. *local* entities are arrays, defines, or variables that were created inside a procedure body. *local* can only be given after the user has *entered* a function or procedure (see the discussion of *enter* below). Any type of DELIGHT entity is considered to be a *system* entity if its name ends in an underscore ("_"). This convention has been adopted by DELIGHT systems personnel in order to avoid name conflicts between user and system arrays, defines, macros, procedures, or variables. Conflicts will never exist if users follow the general rule of *never creating any name that ends in an underscore*. Just as for the display of simulation output in the previous section, the requested item above can use the "magic" characters "*" and "?" to request only items that match a certain pattern.

The following terminal session demonstrates the *display* command. When starting DELIGHT, several global optimization-related variables already exist in the variable *pool*. Other variables, created at any time during the course of a session, are also listed:

[36] This is only part of the format. See [110] for additional options.

```
1> v1 = 1
1> v2 = 2
1> display variables *

9 variables:

   Iter              =   0.00000        GLOBAL
   Neq               =   0.00000        GLOBAL
   Nfineq            =   0.00000        GLOBAL
   Nfmulticost       =   0.00000        GLOBAL
   Nineq             =   0.00000        GLOBAL
   Nmulticost        =   0.00000        GLOBAL
   Nparam            =   0.00000        GLOBAL
   v1                =   1.00000
   v2                =   2.00000
```

Similarly, the other DELIGHT entities, arrays, defines, functions, macros, and procedures, can also be displayed. The *local* option is demonstrated below after introducing the *enter* command.

Another feature to aid program development is the *enter* command, already used in debugging the Newton-Raphson development example in section 4.7.2. It allows the local entities of a procedure—variables, arrays, and defines—to be displayed, used in RATTLE expressions e.g. for debugging purposes, or even assigned values. Arrays or variables that have been imported are also available while entered. These features are seen in the following simple example:

```
1} a = 1  ;  b = 2
1> procedure demo  {
1}     import a, b
1}     c = 3  ;  d = 4
1}     print a b c d
1}     }
1> demo()
 1.000  2.000  3.000  4.000
```

After executing this procedure as above, we may enter it and display its local variables. To leave the entered state, *leave* is typed:

```
1> enter demo
e> display local variables *

4 variables:

   a     =   1.00000       (IMPORTED)
   b     =   2.00000       (IMPORTED)
   c     =   3.00000
   d     =   4.00000

e> leave
1>
```

Note that after entering a procedure with the *enter* command, the prompt string changes to "e>", a reminder that any variables created or used are actually local to the entered procedure. As mentioned earlier, this is similar to other systems such as Pathcal [156] that rely on different prompt characters to indicate the state of the system.

Another useful command for debugging or analyzing run-time behavior is *display_time*. It displays a list of procedure names, total cpu time, direct cpu time, and number of times called for RATTLE procedures and built-in routines, sorted by total cpu time (largest first). Direct cpu time is the total amount of time actually spent in a procedure but *not* in any procedure called by it. The *clear_time* command resets all the call-counts and the cpu time values to zero, i.e., the *display_time* quantities are since the last *clear_time*. One of the uses of these commands is for pin-pointing major cpu time bottlenecks in a program so that these sections of code can be made more efficient. These commands are shown in the following continuation of the above terminal session:

```
1> clear_time
1> for i = 1 to 5
1>     demo()
 1.000   2.000   3.000   4.000
 1.000   2.000   3.000   4.000
 1.000   2.000   3.000   4.000
 1.000   2.000   3.000   4.000
 1.000   2.000   3.000   4.000
```

```
1> display_time
  TOTAL     DIRECT    NUMBER
SECONDS   SECONDS  OF CALLS  PROCEDURE/MACRO NAME
1.27      -------     ---    Cpu-time since last "clear_time"
.167      .100          5    demo
6.67e-2   6.67e-2        5    prinf6
```

The built-in DELIGHT routine *prinf6* listed above is called by the *printf* statement in procedure *demo*. Notice that the total time in *prinf6* plus the direct time in *demo* add up to the total time in *demo*. These computer times are on a VAX 11/780 running "the greatest operating system of all time," Berkeley UNIX!

For resuming an optimization or a software development at a later time the *store* and *restore* commands allow the entire state of DELIGHT to be written to and read from files called *memfiles*. A *memfile* is a large binary file which contains all user and DELIGHT program internal arrays, variables, etc., and all of the precompiled and user-written RATTLE procedures which a user has access to. This feature is similar to saving a workspace in an APL environment [52]. Both commands may be followed by the filename of the memfile to use; if no name is given, file *memfile* is used by default.

Other user-friendly DELIGHT features include the *history mechanism* of the UNIX *csh* [72] and a built-in editor. The simplest use of the *history* command is to just look at the *history list*, the most recent commands typed at the terminal:

```
1> history 13
 37 procedure demo  {
 38    import a, b
 39    c = 3  ;  d = 4
 40    print a b c d
 41    }
 42 demo()
 43 enter demo
 44 display local variables *
 45 leave
 46 clear_time
 47 for i = 1 to 5
 48    demo()
 49 display_time
```

The real purpose of the history list, however, is to be able to re-issue previous commands easily. One may re-issue a command by typing, for example, "!dis" if the command had started with the letters "dis" or "!49" if the command had been number 49 in the *history* command output. If "!dis" is typed, DELIGHT looks up the history list, starting from the bottom (most recently typed line), and reuses the first command line it finds which begins with the letters "dis". The command to be reused is first printed on the terminal. We continue with the above terminal session:

```
1> !de
demo()
 1.000  2.000  3.000  4.000
1> !42
demo()
 1.000  2.000  3.000  4.000
```

DELIGHT features a built-in, UNIX-like editor (without any *vi* screen mode)[37]. For a comprehensive document giving a complete list of all editor commands, see the *DELIGHT Reference Manual* [110].

[37] This editor is not needed on UNIX since there it is so easy to temporarily suspend an interactive program such as DELIGHT (via the ctrl-Z mechanism of *csh* [72]) and later reactivate it by bringing it back into the foreground. However, on systems where this is not possible, it is necessary to edit from within DELIGHT.

CHAPTER 5

DELIGHT Applications

In this chapter we present several applications of the DELIGHT system to various areas of engineering design. One of the foremost application areas to date, the design of electronic integrated circuits using DELIGHT.SPICE, is presented first in section 5.1. This includes an introduction to the types of design problems that occur in the electronic circuit area followed by examples of several problem formulations found to be useful for these types of design problems. After presenting a few specific features of DELIGHT.SPICE, we present the design of several circuits using the methodology introduced in chapter 4. The second part of the chapter, section 5.2, deals with several other engineering design applications. These are the design of digital filters, feedback control systems, and earthquake-resistant buildings. Other DELIGHT applications not discussed further include the MINLP system [85] which implements in RATTLE the MINMAX/MINBOX LP design approach of APLSTAP [60] (see section 2.2), and the recent DELIGHT.ARBSIM system [115] which allows DELIGHT to be easily used with an *arb*itrary *sim*ulator without requiring the normal load/linkage of the simulation program and its simulation interface routines with DELIGHT. Instead, the simulator runs as a completely separate program with all communication to and from DELIGHT through input and output files.

5.1. Electronic Circuit Applications Featuring DELIGHT.SPICE

DELIGHT.SPICE is the union of the DELIGHT system and the SPICE [105] circuit analysis program, for the purpose of applying optimization-based computer-aided design techniques to electronic circuit design. With DELIGHT.SPICE, circuit designers can take advantage of all the optimization and other features of DELIGHT presented in the previous chapter. They may automatically adjust circuit parameters such as resistor and capacitor values and device geometries such as bipolar transistor (BJT) areas and MOSFET lengths and widths. The problem description expressions can be a function of the node voltage across any two circuit nodes or the branch current through any voltage source as well as directly a function of the design parameters[1]. Moreover, these voltages and currents can be output results from any of the three SPICE analysis modes: DC (bias-point solution), AC (frequency-domain), or transient (time-domain).

As an example of a circuit design problem whose performance objectives and constraints are indirectly a function of the design parameters by being a function of circuit responses only, consider the following wide-band amplifier example. The design of the wide-band amplifier could have as a performance objective that the bandwidth of the amplifier be increased and as an ordinary inequality constraint specification that the DC power be less than some value. The design parameters could be a capacitor value and a BJT area. Neither the performance objective nor the constraint specification are explicit functions of the design parameters; this dependence is implicit through the solution of the

[1] Future enhancements mentioned in chapter 6 include other circuit response outputs such as noise or distortion and output currents through any element.

circuit equations by the simulator. In particular, the bandwidth can be evaluated by finding the -3dB point of the frequency response from an AC analysis, while the DC power can be computed as the product of the DC current through the power supply times the supply voltage. This DC current would be computed by a DC analysis. The DELIGHT.SPICE system computes these circuit responses using the simulation program SPICE.

Since the enhanced multiobjective problem formulation of sections 4.4.1.2 and 4.4.2.2 and the phase I-II-III method of feasible directions first introduced in section 4.5.2.2 were actually conceived by working in the electronic circuit design area, users of DELIGHT.SPICE will probably restrict themselves to those tools. However, they still retain the possibilities of selecting a different algorithm from the RATTLE Algorithms Library or of modifying or creating their own.

The remainder of this section is organized as follows. We first survey various aspects of the design of electronic circuits in section 5.1.1. Sections 5.1.2 and 5.1.3 then discuss several useful problem formulations and additional system features. Finally, section 5.1.4 presents the circuit design examples.

The simulation interface of DELIGHT to SPICE implements each of the interface routines covered in section 4.7.2. It needs no further discussion other than points discussed in section 5.1.3 such as the various SPICE options available with the *option* command. However, we take this opportunity to note that the interface of DELIGHT to SPICE would not have been possible without the help of E. Cohen's carefully created program reference manual for SPICE [30].

5.1.1. Nature of Electronic Circuit Design

In this section we present a brief survey of various aspects of the design of electronic circuits. We place particular emphasis on the design of monolithic *integrated* circuits (IC's). (Indeed, the last three letters in "SPICE" stand for "integrated circuit emphasis".) We first take a look at several classes of design problems and then review many of their properties. A list of typical performance objectives or constraints is then given for various kinds of IC's. The section closes with a discussion of performance tradeoffs and one of their particularly important roles in an IC industrial environment. We remark that many of these ideas originated while applying DELIGHT.SPICE and interacting with circuit designers in such an environment.

In the design of electronic circuits there are many classes of design problems that face designers. Probably the largest problem class is that of coming up with a circuit topology and meeting or optimizing a set of design specifications. During this process, circuit designers continually try to better their designs by improving a set of performance objectives subject to constraint specifications. Other classes of problems include comparing various circuit configurations to find the "best", and paying close attention to the models that they use in their simulations. The latter, referred to as *model parameter extraction*, involves determining the parameters of the models so that the mathematical equations of the models represent physical behavior of the electrical components which is significant to the way the components will be used. It also involves model simplification, that is, replacing complex models by simpler models for computational efficiency. Other very important concerns to most circuit designers not directly considered in this research stem from component statistical variation inherent in most manufacturing processes. (See, for

example, chapter 12 of Brayton and Spence [23] or [24].) These types of problems include design centering, tolerance assignment, yield estimation, and yield maximization. Also important are chip layout and packaging, reliability, and testing considerations.

In the following we review various properties typically found in modern integrated circuit design problems. To begin, the design parameters in these problems are usually those that can be controlled by mask creation or modification; circuit designers usually have little control over processing steps or parameters. Thus allowable design parameters include resistor and capacitor values and device geometries such as diode and BJT areas and JFET and MOSFET lengths and widths. For precise applications, BJT geometries, for example, can be addressed by many additional design parameters, e.g., the lengths, widths, and spacings of all the various mask stripes and apertures. Similarly, in some cases geometric properties of actual resistors can be design parameters. This is the case, for example, when the resistance does not follow the idealized L/W formula such as for odd shaped geometries or under high-frequency excitation where distributed and parasitic effects start to dominate.

An important property of integrated circuits is the fact that the values of many circuit parameters can be made to match to a fairly high degree. The accuracy with which two identical transistors or resistors can be matched has a first-order effect on the attainable performance in monolithic operational amplifiers as well as other types of analog integrated circuits such as voltage regulators, analog multipliers, analog-digital converters, voltage comparators, and others [55]. Thus a designer need only consider one out of such a matched set of parameters as a design parameter; the others in the set simply track the one. There are also parameters whose values track one another by being the

same function of e.g., a processing parameter. The foremost example is the tracking of resistor values that all depend on the sheet resistivity. For these reasons, as explained in section 5.1.3, such parameter matching and tracking has been considered in the DELIGHT.SPICE problem formulation.

Integrated circuit design parameters typically have box constraints. These can be due to minimum IC mask tolerances, circuit realizability or stability constraints, or the inability of the circuit simulator to function properly (or at all) under the circumstance in which a parameter lies outside of its box. For example, many element values such as resistor and capacitor values are required to be nonnegative for many of the reasons just listed. The goal to minimize chip area also leads to maximum box constraints on parameters such as device areas—though these are usually soft constraints that can be traded off against other performance factors.

Another property of IC design problems concerns the performance objectives and constraints that circuit designers face. Not only are they of varying importance to the designer, but often they are defined over a continuous interval of an independent variable such as time, frequency, temperature, sheet resistivity, power supply voltage, radiation level, manufacturing tolerance, etc. Several of these *functional* specifications can be seen in the examples of the next paragraph.

We now give typical objectives or constraints for various types of IC design problems. We emphasize (and it can be seen below) that these factors involve DC, AC, and transient circuit performance. Total DC power in a circuit is usually a concern of all circuit design problems. For integrated circuits, individual component areas are another concern because they affect total IC chip area and

hence cost. In analog IC's such as operational amplifiers, additional performance factors include bandwidth and open or closed-loop gain, gain and phase margins, slew rate or the maximum rate of change of the output voltage, maximum output voltage swing, various voltage drifts over temperature, linearity, input and output impedances, and various *rejection ratios*. These ratios represent the ability of the circuit to reject certain undesirable signals. For example, common mode input signals on a differential input amplifier are represented by the common mode rejection ratio (CMMR) while spurious noise on the power supply line is represented by the power supply rejection ratio (PSRR). Other factors include DC input bias and offset currents, input offset voltage, and how all of these drift over temperature, sometimes referred to as their *temperature coefficients*. Designers of high-frequency amplifiers are usually concerned with the above as well as noise figure, total output noise, equivalent input noise, and the presence of distortion components in the output. For digital circuits and many analog circuits with switching functions, there are many other performance factors. These include transient properties of waveforms such as switching time, rise time, settling time, percent overshoot, delay or access time, and input capacitance, which affects these other properties of previous stage waveforms. Static DC factors that affect digital circuit performance include switching threshold, fanout and the related maximum output drive current, logic output levels, and their resulting noise margins. Finally, in some cases there are such esoteric considerations as immunity from external radiation such as gamma radiation or high energy electromagnetic pulses.

Performance factors such as the ones just presented are very often in close competition. Thus, circuit designers are faced with many tradeoff decisions. To show the potential of the tradeoff methodology and features presented in the

previous chapter, we now consider a scenario of a typical IC design process in an industrial environment. A typical design process consists of the following overlapping steps:

1. circuit specification
2. process selection
3. topology synthesis
4. parameter selection
5. simulation

After marketing has derived a set of specifications and a production process has been selected, the circuit designer chooses an initial configuration to realize the desired function. Once this has been done, circuit parameter values (resistor, capacitor values, device geometries, etc.) are repeatedly varied until performance, evaluated by circuit simulation, meet the specifications. There are many feedback paths since the specifications often cannot be met and an alternative topology, process, or set of specifications itself must be considered. Experience has shown that circuit designers are very good at making the complex decisions involved in the first three steps above. However, an optimization-based system such as DELIGHT can greatly assists the designer in the time-consuming and repetitive parameter adjustment step of the design process. Also, the ability to perform tradeoffs by giving emphasis to particular objectives and constraints through adjustment of the good and bad values has an important advantage. It can result in a variety of attractive solutions, all of which can be realized with only minor changes in a few IC masks. Thus the designer can offer the marketing department several product options, each of which represents the circuits best performance for the particular emphasis given.

In conclusion, many circuit design problems are well suited to the multiobjective problem formulation of the previous chapter. This conclusion is further

warranted by the fact that circuit designers are often interested in performance *improvement* rather than finding a true optimum, due to the computational costs of circuit simulation. Hence they should be (and have been) interested in the approaches and algorithms presented in the previous chapter.

5.1.2. Circuit Design Problem Formulations

In this section we present several examples of problem descriptions that have been successfully used for some of the circuit design objectives and constraints given in the previous section. These pertain to the multiobjective problem formulation of sections 4.4.1.2 and 4.4.2.2. We proceed from the simplest DC objectives to more complex formulations such as one that uses an objective and a constraint in combination to maximize bandwidth or minimize switching delay. We stress that formulations other than the ones presented here may also be used in setting up circuit specifications.

An example of the simplest possible objective is to minimize the total DC power in a circuit. Suppose a small series resistor has been placed in series with the power supply for sensing its current. If this resistor has value .5 ohms and its two nodes are 101 and 102 then we might have the following in the indicated problem description files:

In "S" File:

```
simulation_output vdc(101,102)
```

In "M" File:

```
objective 3 'DC Power' minimize VDD*(vdc(101,102)/.5ohms) good=1.5w bad=1.8w
```

where *VDD* is the voltage of the DC power supply voltage source.

Another simple objective involves a simulation output versus a fixed value of

an independent variable. Consider maximizing the gain-bandwidth product of an operational amplifier which has been set up with a unity AC input source so that its voltage gain is identical to the magnitude of the voltage at node 101. After plotting the voltage gain versus frequency (using, for example, *plot db(vm(101)) vs FREQ from 100Hz to 10megHz dec 10*), we find a frequency in the middle of the dominant-pole, 20db/decade range, say, 100kHz, and we simply maximize the voltage magnitude at this fixed value of variable *FREQ*. (In the next section we show how variable *FREQ* is used for frequency-domain outputs, similar to the way Fortran variable *TIME* was used for time-domain simulation outputs in section 4.7.2.) This would require:

In "S" File:

```
simulation_output vm(101)
```

In "FM" File:

```
objective 5 'GB Product' maximize vm(101)*100kHz good=10megHz bad=8megHz using
   FREQ = 100kHz
```

In the above "S" file, the *simulation_output* statement declares the magnitude of the voltage at node 101 as an output; *vm* is a DELIGHT.SPICE output keyword (see section 4.7.2).

A slightly more complex performance objective is to have a particular node voltage as close to a given voltage level as possible. Suppose we want the voltage on node 101 close to 5 volts. One way to achieve this is to minimize the squared deviation from the desired, requiring the following in the indicated files:

In "S" File:

```
simulation_output vdc(101)
```

In "M" File:

```
objective 3 'Node 101' minimize (vdc(101)-5)**2 good=0 bad=(.5)**2
```

The good and bad values of this formulation are somewhat awkward in that they do not clearly represent how far the node voltage is from 5 volts. Another approach is to make two soft constraints: one to stay below, say, 5.2 volts and the other to stay above, say, 4.8 volts. In this case we would require:

In "I" File:

```
constraint 4 'Upper 101' vdc(101) <= good=5.2v bad=5.4v soft
constraint 5 'Lower 101' vdc(101) >= good=4.8v bad=4.6v soft
```

However, when both good values are nearly equal, this approach faces the danger of the two constraints having equal and opposite gradient vectors. This may cause some optimization algorithms to believe prematurely that necessary optimality conditions have been satisfied.

The two-constraint case just presented can also be applied to keep a node voltage between two curves over the range of an independent variable, i.e., for *functional* constraints. The simplest situation is when the two bounding curves are constant. This would be the case if the output voltage of a voltage reference was constrained to lie within a band over temperature. To keep node voltage 101 lying within a .05 volt band around 1.2 volts we might have:

In "FI" File:

```
constraint 4 'Upper Band' vdc(101) <= good=1.25 bad=1.3 soft
for_every TEMPDC from -55 to 125 initially by 10

constraint 5 'Lower Band' vdc(101) >= good=1.15 bad=1.1 soft
for_every TEMPDC from -55 to 125 initially by 10
```

where *TEMPDC* is the SPICE simulation temperatures in degrees centigrade[2].

The general situation is when the two bounding curves are functions of the functional "W" variable. Suppose we want to place upper and lower bounds on a transfer function versus frequency. Such constraints often occur in filter specifications. This may be viewed as placing upper and lower bounding curves on a Bode plot of the transfer function versus frequency. First we need the expressions versus the variable *FREQ* that describe the bounding curves. One approach is to give the expressions explicitly. For a lowpass transfer function, these might be

$$db\left(\frac{1}{\sqrt{1 + (FREQ/100megHz)^2}}\right) + 3db$$

for the upper bound and

$$db\left(\frac{1}{\sqrt{1 + (FREQ/100megHz)^2}}\right) - 3db$$

for the lower bound, where the function *db* gives its argument in units of decibels, i.e, it is just $20\log_{10}(\cdot)$. However, to demonstrate a more general approach that allows an arbitrary piecewise linear specification of the curves, we use the *interpolated_array* statement (explained more fully in [110]). The first three lines below declare a piecewise linear entity called *Up101*; when *Up101* is used in any expression, it represents the interpolated value of the piecewise linear function obtained by connecting the 6 x,y points *1megHz, 1.0, 2megHz, .9*, etc., from the second and third lines, versus the present value of independent variable *FREQ*. The "FI" file lines then use the two interpolated arrays to achieve the desired bounds:

[2] The use of *TEMPDC* here is not entirely correct; see the next section.

In "S" File:

```
interpolated_array Up101(6) over FREQ
1megHz 2megHz 5megHz 10megHz 20megHz 100megHz
1.0    .9     .9     .7      .3      .05

simulation_output vm(101)

interpolated_array Lo101(6) over FREQ
1megHz 2megHz 5megHz 10megHz 20megHz 100megHz
0.7    0.6    .6     .4      .2      .01
```

In "FT" File:

```
constraint 4 'Upper 101' vm(101) <= good=Up101 bad=1.2*Up101 soft
for_every FREQ from 1meghz to 100meghz initially dec 10

constraint 5 'Lower 101' vm(101) >= good=Lo101 bad=0.8*Lo101 soft
for_every FREQ from 1meghz to 100meghz initially dec 10
```

As before it is assumed that the magnitude of the voltage at node 101 is identical to the transfer function value. The above two functional inequality constraints force this magnitude to lie between two piecewise linear curves on a Bode plot.

Another important use for frequency-domain functional constraints is to place bounds on the closed-loop peaking of a feedback amplifier in order to meet stability or settling time specifications. The use of such "equivalent" frequency-domain criteria for time-domain specifications usually results in a great computational savings.

Let us now address constraints on the phase margin (*PM*) of an amplifier such as $PM \geq 45°$. Phase margin is defined as $180° - \varphi_1$ where φ_1 is the amplifier phase at the frequency where the open loop gain crosses unity. By substitution we can convert the *PM* constraint to the phase constraint $\varphi_1 \leq 135°$. (This constraint simply keeps the phase at unity gain away from 180° where closed-loop instability occurs.) The problem is to find the value of *FREQ* where the gain is unity. Once found we simply obtain φ_1 from, eg., *vp(101)* (phase at node 101) assuming the amplifier output is at node 101. To find where the open loop gain is unity we can use the *findroot* statement (explained more fully in [110]) to find

the root (zero) of an arbitrary expression, given lower and upper values which bracket the root. Thus we first obtain suitable values which bracket the root versus variable *FREQ*, say 500kHz and 10megHz, and then use

```
findroot vm(101)-1 vs FREQ from 500kHz to 10megHz
```

After this statement executes, variable *FREQ* equals the frequency value where expression *vm(101)-1* equals zero, i.e., the unity gain frequency. We then use the phase value *vp(101)* as our constraint as shown below:

In "S" File:

```
simulation_output vm(101)
simulation_output vp(101)
```

In "I" File:

```
constraint 4 'Phase' vp(101) <= good=135 bad=155 soft using
   findroot vm(101)-1 vs FREQ from 500kHz to 10megHz
```

In regard to frequency-domain transfer functions, one much tougher objective to formulate is to maximize the 3db bandwidth of an amplifier. Suppose a circuit has been set up with a unity AC input source so that the magnitude of the voltage at node 101 is equal to the amplifier voltage gain. A simple approach is to use the *findroot* statement as above to find the frequency where the gain falls to 3db (.707) of its low-frequency value, and then return this frequency as the objective value as in:

In "M" File:

```
objective 4 'Bandwidth' maximize FREQ good=4megHz bad=3megHz using
   findroot vm(101)-.707*GAINDC vs FREQ from 2megHz to 6megHz
```

where *GAINDC* is the known low-frequency gain of the amplifier. A problem with this approach is that instead of the amplifier gain staying constant out to the 3db point as desired, a large peak might develop that gives an artificially high

3db rolloff frequency. To prevent such a peak, functional constraint bounds as shown earlier in this section can be used to constrain the transfer function to lie within a band. However to maximize bandwidth, in some sense we want the upper bound of these functional constraints to increase instead of remaining fixed. What we desire is to solve the following mathematical problem:

$$\underset{\begin{bmatrix}\mathbf{x}\\ B\end{bmatrix}}{\text{maximize}} \; \{\, B \mid \mathit{Fineq}(\mathbf{x},\mu B) \le \mathit{Uval} \;\text{ and }\; \mathit{Fineq}(\mathbf{x},\mu B) \ge \mathit{Lval} \;\; \forall \mu \in [0,1] \,\}$$

where *Uval* and *Lval* are the values of the upper and lower transfer function bands and B is the approximate bandwidth, i.e., the value at the upper end of the frequency range for both band constraints. Note that the "W" value actually passed as the second argument to *Fineq* ranges from 0 to the present value of the bandwidth being maximized, B. This formulation, reminiscent of the idea of "designable desired" in [58], can be achieved in the multiobjective problem formulation using the *objective/constraint combination technique* of introducing a non-simulation design parameter B to use in a *maximize* objective and also in the calculation of the two banding functional constraints. For a low-frequency gain of unity (0db), the following might be in the indicated files:

In "S" File:

```
simulation_output vm(101)
design_parameter B
```

In "M" File:

```
objective 1 'Bandwidth' maximize B good=10megHz bad=5megHz
```

In "FI" File:

```
constraint 1 'Upper Band' db(vm(101)) <= good=3db bad=4db using
   FREQ = mu * B
for_every mu from .1 to 1 initially dec 15

constraint 2 'Lower Band' db(vm(101)) >= good=-3db bad=-4db using
   FREQ = mu * B
for_every mu from .1 to 1 initially dec 15
```

Note that the two functional constraints are *hard* (due to the absence of the keyword *soft*) and do not take part in any design tradeoffs with the bandwidth objective. The *for_every* lines above use the *dec* keyword to cause the variation of *mu* to be logarithmic over its interval range.

The above objective/constraint combination technique can also be applied to time-domain objectives such as minimizing the settling or delay times of an output pulse. Without further discussion we show what might be required in the various files to minimize the time required for an output pulse to settle to within .01 percent of 5 volts.

In "S" File:

```
simulation_output vtr(101)
design_parameter T
```

In "M" File:

```
objective 1 'Settling Time' minimize T good=2us bad=3us
```

In "FT" File:

```
constraint 1 'Upper Band' vtr(101) <= good=1.01*5v bad=1.015*5v using
   TIME = (1-mu)*T  +  mu*Tstop
for_every mu from 0 to 1 initially by 40

constraint 2 'Lower Band' vtr(101) >= good=.99*5v bad=.985*5v using
   TIME = (1-mu)*T  +  mu*Tstop
for_every mu from 0 to 1 initially by 40
```

In the above, the variation of *mu* from 0 to 1 causes *TIME* to vary from *T* to *Tstop*, the final time of the time-domain simulation by SPICE.

5.1.3. Additional DELIGHT.SPICE-Specific Features

This section introduces several features required for using DELIGHT optimization in a circuit design environment; most of these features are specific to DELIGHT.SPICE. These include the allowable design parameters and simulation outputs, sweep commands for controlling SPICE simulation, settable

parameters, matched parameters and parameter tracking, the options that can be controlled in SPICE, various display commands for viewing circuit data and properties, and other important features. But first, we discuss the circuit description file.

Circuit description file. As introduced in section 4.7.3, the simulation program is initialized using the *sim FILENAME* statement in the "S" file where *FILENAME* is the name of the file containing the structural description of the system to be simulated. For DELIGHT.SPICE and circuit design, this file is the normal SPICE input circuit file that describes the circuit interconnections, element values, and other input requirements. DELIGHT.SPICE does not make any requirement on the name of the circuit description file (since this name is only used with the *sim* command). A reasonable possibility (which may be dependent on the particular computer being used) is to use the optimization problem name followed by the characters *.CKT*, e.g., *pbname.CKT* for problem *pbname* of the previous chapter. The circuit description file is set up in almost exactly the same way as for the SPICE batch program. This input format will not be discussed here; see a SPICE user's guide such as [150] for more information. There are, however, a few important differences between a batch SPICE circuit file and a DELIGHT.SPICE circuit file. One technical difference is that a *.TRAN* line *must* be in the circuit file if any performance objectives or constraints depend on SPICE transient (time-domain) outputs. See the *DELIGHT.SPICE User's Guide* [114] (included as appendix I of this dissertation) for more details.

Another difference involves not technical rules as above but the actual nature of the circuit being simulated. The circuit described in both types of circuit files consist of elements belonging to the circuit being optimized, and external sources, loads, and/or feedback elements that are needed to test the circuit. In

SPICE, the user changes the test conditions *manually* (by editing the circuit file) and reruns SPICE for each of the design specifications. In DELIGHT.SPICE, the optimization must be able to change the test conditions *automatically* during execution, and it can only do so by changing element values. Thus, the circuit file must describe a single test circuit such that all the configurations required for the various test conditions may be obtained by setting element values only. These elements are examples of settable parameters that must each be declared using the *settable_parameter* statement of section 4.7.3. Settable parameters are discussed further below.

Design parameters. As indicated in the previous chapter, all SPICE design parameters must be declared by using a *sim_design_parameter* statement in the "S" file for each. For resistors, capacitors, and inductors, the parameter name of the design parameter for this declaration is simply the element name as in *R101*, *CBYPASS* or *LSHUNT*. For bipolar transistor areas the name is the element name followed by *_A* as in *Q9_A*. For MOSFET lengths or widths the name is the element name followed by *_L* or *_W* as in *MX37_L* or *MFEED_W*. The following table summarizes the allowable design parameters, with *XXX* representing the actual circuit file element name (excluding the first character):

DELIGHT.SPICE Allowable Design Parameters	
Parameter	Parameter Name
Resistor Value	RXXX
Capacitor Value	CXXX
Inductor Value	LXXX
BJT Area	QXXX_A
MOSFET Length	MXXX_L
MOSFET Width	MXXX_W

Other SPICE parameters that have not yet been implemented as design parameters include mutual inductors, transmission lines, controlled source gains, diode areas, and JFET parameters.

Simulation outputs. The output keywords returned from DELIGHT.SPICE simulation interface routine *OutputKeywords* (see sections 4.7.2 and 4.7.3) for accessing the output results of the various SPICE analysis types are for DC, AC, and transient node voltages[3] and are shown in the following table:

[3] At the time of this writing, output branch currents had not yet been implemented; they must currently be obtained as the voltage across a small series resistor.

DELIGHT.SPICE Allowable Simulation Outputs		
Output Keyword	Syntax	Description
vdc	vdc (NODE1 [, NODE2])	DC node voltage.
vm	vm (NODE1 [, NODE2])	Magnitude of AC node voltage at the present value of variable *FREQ*.
vp	vp (NODE1 [, NODE2])	Phase of AC node voltage in radians at the present value of variable *FREQ*.
vtr	vtr (NODE1 [, NODE2])	Time-domain node voltage at the present value of variable *TIME*.

Anything shown in square brackets in this table is optional: giving one node means the output voltage is with respect to the ground node (node 0) while giving two means that the output voltage is across the two.

A small digression on the use of these SPICE outputs is in order. *vdc* can be used in any RATTLE expression and returns the DC node voltage from the last SPICE DC analysis. As explained in section 4.7.2, if any SPICE circuit parameters have changed since the last analysis, *vdc*, through simulation interface routine *Output Value*, automatically performs a new DC analysis; this rule applies to the AC and transient SPICE simulation modes as well. *vm* and *vp* return the AC magnitude and phase of the specified node at the value of global variable *FREQ*. Thus, to print the magnitude at a particular frequency one could type:

```
FREQ = 15megHz
print vm(101)
```

while to plot the complete frequency response of several nodes one could type:

```
plot vm(101) vm(305) vs FREQ from 1megHz to 70megHz dec 20
```

Similarly, *vtr* returns the transient output voltage of the specified node at the value of global variable *TIME*. Thus, as for AC magnitudes and phases, one could print the transient output waveform at a particular time using:

```
TIME = 75ns
print vtr(101)
```

while a plot of the transient response versus time could be obtained using:

```
plot vtr(101) vs TIME from 0 to 100ns by 1ns
```

The *plot* limits and increment need not be identical to those on the *sweep* command which is discussed next.

Sweep commands. If there are one or more AC simulation outputs (*vm* or *vp*) declared, the "S" file must contain a *sweep FREQ* command to set the values of the SPICE AC run controls. Similarly, if there are one or more transient simulation outputs (*vtr*) declared, the "S" file must contain a *sweep TIME* command to set the values of the SPICE transient run controls. The general forms for these sweep commands are:

```
sweep FREQ from FROM_VAL to TO_VAL {by   } INC_VAL
                                   {times}
                                   {oct  }
                                   {dec  }
                                   {log  }

sweep TIME from FROM_VAL to TO_VAL by INC_VAL
```

In these commands, the arguments *FROM_VAL*, *TO_VAL*, and *INC_VAL* are arbitrary numeric values except that the *sweep TIME* limits must be identical to the ones on the *.TRAN* line in the SPICE circuit file mentioned earlier in this section.

The keywords *by*, *times*, etc., have exactly the same meaning as for the *for_every* statement of section 4.4.2.2 and the *plot* statement of section 4.6.1.

Settable parameters. As explained in section 4.7.1, a SPICE circuit parameter may be declared as a *settable parameter* so that it may be set, but without it being an optimization design parameter. One reason for declaring such a parameter is for simply exercising interactive control over a circuit element or device geometry without having to modify the circuit description file and without having DELIGHT.SPICE automatically adjust it in an optimization. Two other uses are for setting up test conditions and for parameter tracking.

Test conditions. The second use of settable parameters is for establishing the different test conditions for a circuit. For example, two different constraints might each involve a particular output voltage but under different load resistances. In this case, the load resistor would be a settable parameter. As another example, a calculation such as operational amplifier common mode rejection ratio (CMRR) usually involves the ratio of common mode gain to differential mode gain. One could use three voltage sources, one that would supply a one volt common mode signal, and two others that would each supply a one-half volt signal to the differential inputs. These three voltage sources would be declared as settable parameters and all three values would be set in the CMRR calculation. In the above two examples, the test configuration remains fixed. A change in the test configuration (circuit topology) may be approximated by declaring resistors as settable parameters and setting them to negligibly small values to achieve a short or very large values to achieve an open circuit[4]. The use of such "switch" resistors is shown in the first optimization exam-

[4] To avoid numerical problems, these two resistor values should not differ by more than *about* eight to ten orders of magnitude.

ple of the next section.

The circuit parameters that may be declared using the ***settable_parameter*** statement of section 4.7.3 include any of the allowable design parameters listed in the design parameter table above in addition to those shown in the following table:

DELIGHT.SPICE Allowable Settable Parameters	
Parameter	Parameter Name
(Any Designable Parameter)	(Same)
Voltage Source DC Value	VXXX
Voltage Source Magnitude	VXXX_M
Voltage Source Phase	VXXX_P
Mosfet Drain Area	MXXX_AD
Mosfet Source Area	MXXX_AS
Mosfet Drain Perimeter	MXXX_PD
Mosfet Source Perimeter	MXXX_PS
Temperature in Degrees Centigrade	TEMPDC

Note in particular that the analysis temperature in SPICE may be set by first declaring settable parameter *TEMPDC* and then setting it with the *set* command. For more on this parameter, see [114].

When declared, settable parameters take their initial values from the values in the circuit description file. These values remain until changed by a *set* command. Care should be taken by the user to insure that all settable parameters

have the appropriate value for each test condition. This can be done by either setting *all* related settable parameters *before* each circuit analysis as in the pseudo-code:

```
set SP1 = 1
set SP2 = 0
set SP3 = 0
Perform test which needs only SP1 set.

set SP1 = 0
set SP2 = 1
set SP3 = 0
Perform test which needs only SP2 set.
```

or by reseting *each* settable parameter that was set *after* the circuit analysis as in:

```
set SP1 = 1
Perform test which needs only SP1 set.
set SP1 = 0

set SP2 = 1
Perform test which needs only SP2 set.
set SP2 = 0
```

The first approach is probably clearer and safer if there are not too many settable parameters.

Parameter tracking. The third use of settable parameters is for handling matched parameters and parameter tracking. Recall from the discussion in section 5.1.1 that in many circuits, two or more parameters must match, i.e., naturally take on the same values, or track, i.e., be proportional to another single parameter. For example, BJT areas in a differential pair are usually matched. To allow such parameters to be considered as a single design parameter, certain procedures must be followed in DELIGHT.SPICE. First, just one of the parameters is declared as a *sim_design_parameter*. Second, all of the other parameters which must match or track the first are declared as *settable_parameters*. Third, the DELIGHT.SPICE keyword *track_param_update* is

followed by *set* commands which set each of the settable parameters to the one declared as a design parameter, optionally multiplied by the tracking proportionality constant. These *set* commands are then followed by the word *end*. For example, if a circuit had one differential pair whose BJT areas were to be designable, the following might be used in the "S" file:

```
sim_design_parameter Q1_A
settable_parameter   Q2_A

track_param_update
   set Q2_A = Q1_A
   end
```

If there were a differential pair which had to be matched and several current mirror bias transistors whose areas had to proportionally track the area of a particular transistor, the "S" file might contain the following:

```
sim_design_parameter Q1_A
settable_parameter   Q2_A
sim_design_parameter QBIAS0_A
settable_parameter   QBIAS1_A
settable_parameter   QBIAS2_A
settable_parameter   QBIAS3_A

track_param_update
   set Q2_A =  Q1_A
   set QBIAS1_A = 4*QBIAS0_A
   set QBIAS2_A = 2*QBIAS0_A
   set QBIAS3_A = 6.4*QBIAS0_A
   end
```

Simulator options. Various SPICE options can be set with the *option* command of section 4.7.3. These along with their default values are shown in the following table:

SPICE Settable Options for the *option* Command	
Option Name	Default Value
abstol	1.0e-8
gmin	1.0e-12
itl1	100
itl2	20
itl3	2
itl4	10
itl5	5000
reltol	1.0e-3
trtol	7
vntol	1.0e-5

See a SPICE user's guide such as [150] for the meanings of the options listed above.

DELIGHT.SPICE provides a few but growing number of commands which are directly related to SPICE and the circuit being simulated. As shown in the examples of the next section, these commands are indispensable for verifying a circuits performance and structure. The commands all call the interface routine *SimulationDisplay* (see section 4.7.2) for displaying user-requested simulator output. The following table shows the command formats and their descriptions:

DELIGHT.SPICE Circuit-Related Commands			
Command Format			Description
probe	{nv}	[P_ARG]	Print the requested SPICE DC node voltages.
	{op}	[P_ARG]	Print the requested SPICE DC element operating points.
	{acnv}	[P_ARG]	Print the requested SPICE AC node voltage magnitudes and phases.
display	{device} {element} {model} {node} {stat}	[ARG] [ARG] [ARG] [ARG]	Display the specified property of the circuit being simulated.
reset_stat			Reset the *display stat* output (counts and cpu times).

The *probe nv* command prints the node voltages of the *most recent* DC or transient analysis time-point. (Internally, it simply prints the values in the SPICE table *LVNIM1*; this table contains the most recent Newton iteration circuit solution.) Optional argument *P_ARG* may be a list of from one to six specific items desired; if not given then all available are printed. The *probe op* command prints the element operating points from the most recent SPICE DC analysis. The *probe acnv* command prints the node voltages from the most recent SPICE AC analysis, done at the present value of variable *FREQ*. (Internally, it simply prints the magnitude and phase of the values in the SPICE table *LCVN*; this table contains the complex node voltages of the most recent AC simulation.)

For the *display* command, optional argument ARG also is from one to six specific items. The use of the *magic characters* "*" and "?" in these arguments

has already been discussed in section 4.7.3. *display stat* is used to print the SPICE statistics, i.e., the cpu times and Newton iteration counts, where applicable, for the various SPICE execution phases. The result is similar to the output produced by a *.OPTION ACCT* line in batch SPICE. To reset all the cpu times and iteration counts, *reset_stat* is typed.

Most of the DELIGHT.SPICE features introduced in this section are shown in the example files and terminal dialogue of the next section.

5.1.4. Circuit Design Examples

In this section, the application of DELIGHT.SPICE to the design of two particular circuits is described. The first is a bipolar operational amplifier ("opamp") and the second is a CMOS chain of inverters for driving a large capacitive load.

Design of a Bipolar Operational Amplifier

In the first example, an operational amplifier is designed using bipolar technology with a *given* set of fabrication process parameters (SPICE *.MODEL* parameters). The amplifier is to operate with 15 volt power supplies, drive a load capacitance of 5pF, and meet the following constraints:

1. The phase margin with unity feedback must be greater than 45 degrees.
2. The gain-bandwidth product must be greater than 3 megHz.
3. The slew rate must be greater than 5 volts/us.
4. The settling time to within .1% of final value, using the test circuit of figure 5.2-c and a 5 volt step input, must be less than 1.5μs.

The following objectives are also of interest:

5. The quiescent DC power with zero nominal output voltage should be minimized.
6. The input offset voltage due to device mismatches should be minimized.

The DC and some of the other specifications above, while important, are not of primary concern and can be relaxed to achieve better performance of others. For example, if a small increase in power or decrease in slew rate is needed to achieve the gain-bandwidth produce specification, the results are more desirable.

In "real-world" circuit design, only a set of specifications such as the above would be given. The designer would begin the design by searching for a circuit topology having the potential to satisfy the constraints and yield acceptable values of the objectives. For the purposes of this example, since DELIGHT.SPICE does not aid in the synthesis of topologies, a particular configuration is assumed to be given, namely that inside the dotted lines of figure 5.1. (Note that the lack of a power output stage in this simple amplifier is consistent with the absence of specifications on available output power above.)

The setup of the problem description files for this particular example is shown in the first subsection that follows. The next subsection shows how to check the files and, in particular, the validity of the measurement techniques used in setting them up. An investigation of the initial circuit performance leads to a modification of several of the problem description files. Finally, the last subsection presents the actual optimization session.

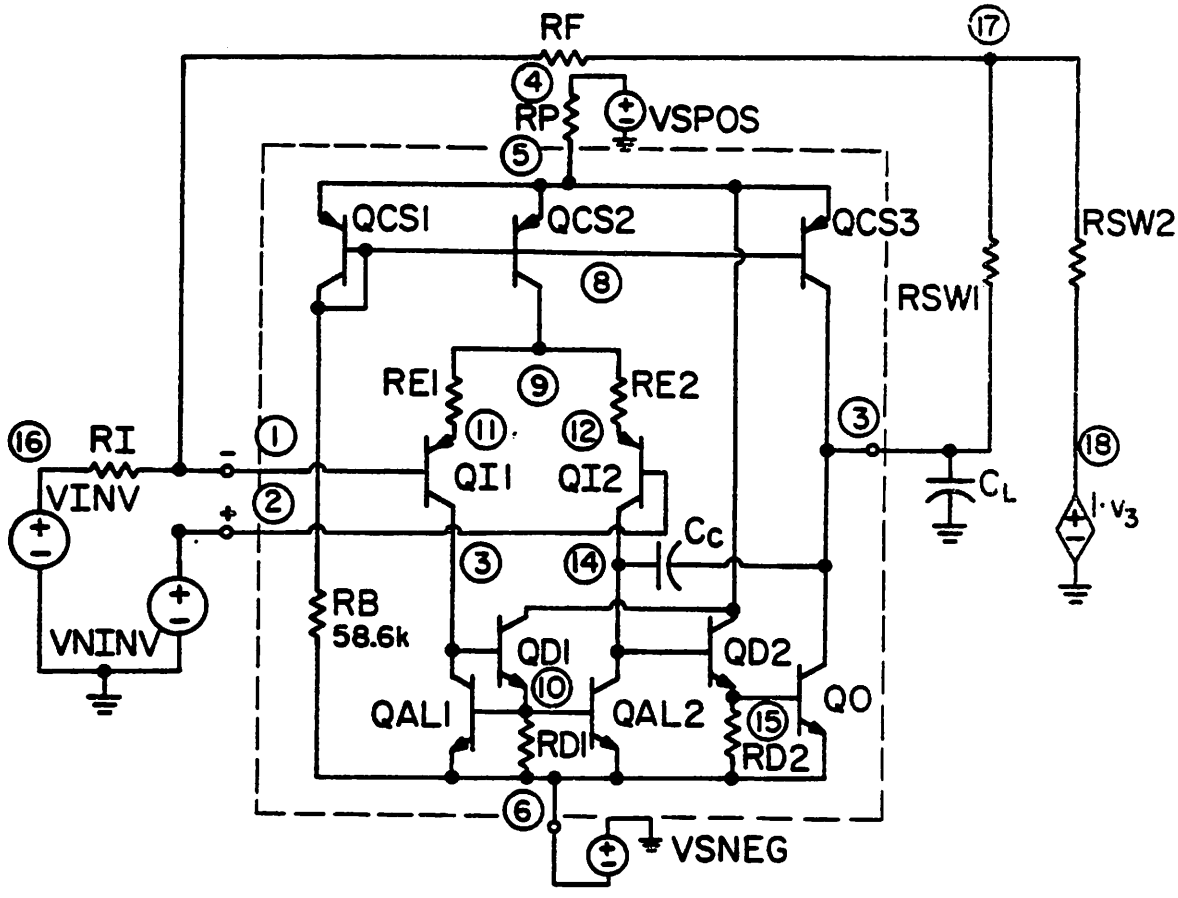


Figure 5.1. Bipolar Opamp Circuit Configuration For First Design Example.

Setting Up the Problem Description Files

Test configurations. The first file to set up is the SPICE circuit description file. Before it can be created, however, several preliminary tasks are done. From the problem specifications, the designer determines what test configurations are needed. For this example, the set of four separate test configurations shown in figure 5.2 suffice to obtain all of the circuit performance data needed to compute all of the objectives and constraints. *Extensive use is made of feedback*, even when measuring characteristics that are, strictly speaking, open-loop characteristics. RP in configuration (a) is a low-value current-sensing resistor. The unity-gain VCVS in configuration (b) insures that the

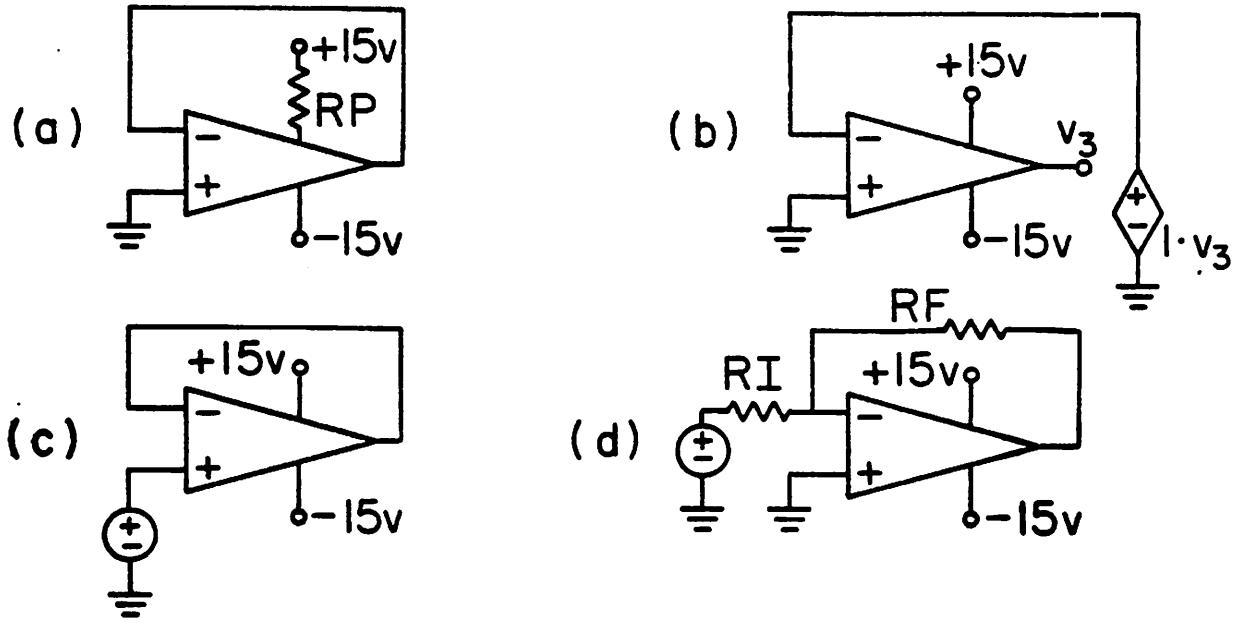


Figure 5.2. Test Configurations For the Bipolar Opamp.

voltage offset measured between the input terminals of the amplifier is not contaminated by a contribution which is the product of the input offset current and the amplifier output resistance. Suitable values of RI and RF in configuration (d) are determined at this time. The rationale for this determination and the basis for its validity will be discussed in the next subsection.

By appropriately changing the values of some of the test circuit elements shown outside the dotted lines in figure 5.1, an approximation to each of the configurations of figure 5.2 can be realized. One of the "switch" resistors RSW1 and RSW2 will have the value 1 ohm, and the other 1 megohm. The designer can have DELIGHT.SPICE interchange the two values as needed during execution, thereby realizing an approximation to a single-pole double-throw switch. RSW1, RSW2, RI, RF, VNINV and the magnitude of VINV must be declared as ***settable parameters***. In order to reduce the chance that confusion about the values of these parameters causes their values to be incorrectly set and invalid measurements to be made, a set of base values for these parameters is established. The base values are those in the circuit file listed below. (They could also be set with the *set* command, as discussed earlier.)

Selection of design parameters. The designer must select which circuit parameters are to be design parameters. Ideally, to explore the potential of a design, all (16, here) circuit parameters which the designer has control of should be included. However, since the cpu time spent by the system is roughly proportional to the number of design parameters[5], this strategy would be impractical. Thus, only circuit parameters that have considerable effect on the design specifications should be considered as design parameters. Also, the number of possible design parameters is reduced by using *track_param_update*

[5] Presently gradients are computed using finite differences.

to force the values of normally matched parameters to be equal. In this example, these are the areas of the QI, QAL, and QD transistor pairs and the values of the RE and RD resistor pairs. This should not affect the optimization since the optimum design of the circuit is probably one in which these matchings hold anyway. For simplicity, and because there are excess degrees of freedom in determining the bias currents, the number of possible design parameters is further reduced by letting the values of RB and the area of QCS1 remain fixed, with a QCS1 bias current of .5mA. Finally, preliminary optimization iterations not shown here with the remaining possible design parameters reveals that several others have little or no effect on the specifications. Thus, there are five remaining design parameters:

1. area of QCS2
2. capacitance of CC
3. resistance of RE1/RE2 matched pair
4. areas of QI1/QI2 matched pair
5. areas of QAL1/QAL2 matched pair

Having determined values for all circuit parameters which are not in fact design parameters, a favorable initial guess of the values for all the design parameters is made. Making the values in the circuit file be the desired initial values probably reduces the opportunities for errors; the values in the circuit file listed below are the desired initial values.

Setup for transient analysis. Since constraints 3 and 4 above require SPICE transient analyses, it is necessary to estimate a suitable value for the *Tstop* (final time) parameter on the *.TRAN* line [150] in the circuit file. If *Tstop* is too small, the validity of any time-domain measurements can be destroyed. If it is too large, cpu and user time is wasted. But it is not necessary that its value remain fixed throughout the optimization. The considerations which govern its

adjustment are explained in the next subsection. On the basis of the design specifications for this example, a value for *Tstop* of 2μs is initially used in the circuit file.

The *.MODEL* lines in the circuit file are set up, using the given device process-dependent parameters, in exactly the same way as are the *.MODEL* cards in batch SPICE.

Circuit description file. The steps presented above lead to the circuit file for this example, file *xamp.CKT* (the problem is named *xamp*), that is shown below:

```
BIPOLAR OPAMP EXAMPLE

* FIRST, THE AMPLIFIER ITSELF:

RB 6 8 58.6K
QCS1 8 8  5 6 MODPNP
QCS2 9 8  5 6 MODPNP
QCS3 3 8  5 6 MODPNP
QI1 13 1 11 6 MODPNP
QI2 14 2 12 6 MODPNP
.MODEL MODPNP PNP BF=50   IS=2E-15A CJE=.3PF VJE=.55V MJE=.5
+                 VBF=50V JLC=2E-16A CJC= 1PF VJC=.55V MJC=.5
+                  RB=300  TF=30NS   CJS= 3PF VJS=.52V MJS=.5
RE1 11 9 1K
RE2 12 9 1K
RD1 10 6 50K
RD2 15 6 50K
QAL1 13 10 6  6 MODNPN
QAL2 14 10 6  6 MODNPN
QD1   5 13 10 6 MODNPN
QD2   5 14 15 6 MODNPN
QO    3 15  6 6 MODNPN
.MODEL MODNPN NPN BF=200   IS=6E-15A CJE= 1PF VJE=.70V
+                 VBF=120V JLC=5E-16  CJC=.3PF VJC=.55V MJC=.5
+                  RB=200   TF=.35NS  CJS= 3PF VJS=.52V MJS=.5
CC 3 14 25PF

*  THE TEST CIRCUITRY:

VSPOS 4 0 DC 15V
VSNEG 6 0 DC -15V
VINV 16 0 AC 1
VNINV 2 0 PULSE 0V1 5V2 0DELAY .02USRISE
CL 3  0  5PF
RP 4 5 1
RI 1 16 1MEG
RF 1 17 1
```

```
RSW1 17  3 1
RSW2 17 18 1MEG
EO 18 0 3 0 1
.TRAN .06US 2US
.END
```

Problem "S" File. Listed below is the "S" file for this example, file *xampS*. It runs the SPICE setup on the circuit file, declares the design and settable parameters, takes care of matched parameters, declares simulation outputs, sets up the SPICE analysis run controls, and sets the DELIGHT global optimization variables.

```
sim xamp.CKT
                                                          # INITIAL
                                                          # GUESSES:
sim_design_parameter QCS2_A variation=.5  min=.05  # 1
sim_design_parameter CC     variation=20p min=1p   # 25pF

sim_design_parameter RE1    variation=100 min=1    # 1k
settable_parameter   RE2
sim_design_parameter QI1_A  variation=.2  min=.05  # 1
settable_parameter   QI2_A
sim_design_parameter QAL1_A variation=.2  min=.05  # 1
settable_parameter   QAL2_A

track_param_update
   set RE2    = RE1
   set QI2_A  = QI1_A
   set QAL2_A = QAL1_A
   end

settable_parameter VINV_M
settable_parameter VNINV
settable_parameter RI
settable_parameter RF
settable_parameter RSW1
settable_parameter RSW2

simulation_output vdc(4,5)  # Power supply current-sensing
simulation_output vdc(2,1)  # Input pin voltage - for offset
simulation_output vm(3)     # Magnitude of output voltage
simulation_output vp(3)     # Phase     of output voltage
simulation_output vtr(3)    # Transient    output voltage

Tstop  = 2us; global Tstop
sweep TIME from 0 to Tstop by .05us
sweep FREQ from 100 to 100megHz dec 10

Nparam = 5
Nmulticost = 2
Nineq  = 3
Nfineq = 2
```

Note that a small but positive lower bound (a one-sided hard box constraint) is

placed on all the design parameters in their declaration statements. This is to keep DELIGHT.SPICE from trying zero or negative values of them in the course of the optimization, a circumstance which can produce numerical difficulties or invalid results in SPICE.

No "P" file is needed in this example because the initial guesses for all the design parameters are the corresponding values in the circuit file.

Problem "M" file. In file *xampM* we formulate the objectives to minimize the total DC power and the input offset voltage due to device mismatches. The latter is estimated using techniques given by Gray and Meyer [55]. These techniques appear below as a rather lengthy procedure that adds up the worst case contributions from a one standard deviation mismatch of each of the three input differential amplifier matched parameter pairs listed under *track_param_update* in the "S" file[6]. The direction in which to make each perturbation below is determined from a consideration of the circuit.

```
prob_function multicost
   objective 1 'power' minimize 30v*vdc(4,5) good=100mw bad=150mw
   objective 2 'o.s. volt.' minimize vos good=5mv bad=7mv using {
      set RSW1 = 1meg          # Set switch for test configuration.
      set RSW2 = 1
      vos0 = vdc(2,1)

      RE1save = RE1            # Offset with unbalanced emitter
      RE2save = RE2            # resistors.
      set RE1 =  .99 * RE1
      set RE2 = 1.01 * RE1
      vos = vdc(2,1)
      set RE1 = RE1save
      set RE2 = RE2save
```

[6] Another more sophisticated technique is to calculate the offset voltage variance using the propagation of variance technique of Spoto [142].

```
        QI1_Asave = QI1_A          # Offset with unbalanced input
        QI2_Asave = QI2_A          # transistors.
        set QI1_A = 1.04 * QI1_A
        set QI2_A =  .96 * QI2_A
        vos = vos + vdc(2,1)
        set QI1_A = QI1_Asave
        set QI2_A = QI2_Asave

        QAL1_Asave = QAL1_A        # Offset with unbalanced current
        QAL2_Asave = QAL2_A        # source transistors.
        set QAL1_A = 1.04 * QAL1_A
        set QAL2_A =  .96 * QAL2_A
        vos = vos+vdc(2,1)
        set QAL1_A = QAL1_Asave
        set QAL2_A = QAL2_Asave

        vos = vos - 3.0*vos0       # Subtract 3 systematic contributions.

        set RSW1 = 1               # Return switch to starting position.
        set RSW2 = 1meg
        }

end_multicost
```

In each case above, as with the other specifications in other problem description files, the good and bad values are chosen according to the uniform satisfaction/dissatisfaction rule of section 4.4.1.2. Note for example, in objective 2 above that when any design parameter is changed from its present value to a value required for the test setup for a particular "measurement", it is changed back to its old value once the simulation result is obtained. This is the purpose of the temporary variables that end in *save* above. These variables are local RATTLE variables and are thus assigned to without using the *set* command.

Problem "I" file. File *xampI* contains the formulations of the phase margin, gain-bandwidth product, and slew rate constraints. The phase margin constraint has been (trivially) converted into a phase constraint as shown in section 5.1.2. All the constraints are made soft using the keyword *soft* so that all of them along with objectives take part in design tradeoffs.

```
function straighten (mu)  {    # This makes the expression to findroot
   FREQ = 10**mu               # as straight as possible, i.e., vm(3)
   return ( db(vm(3)) )        # vs FREQ is straight on log-log paper
   }                           # so straighten(mu) vs mu is straight
                               # on linear paper.

prob_function ineq

   constraint 1 'phase' ph >= good=-135 bad=-160 soft using {
      set RI = 10k
      set RF = 2meg
      findroot straighten(mu) vs mu from log10(1megHz) to log10(20me)
      ph = vp(3) - 180
      set RI = 10meg
      set RF = 1
      }

   constraint 2 'g-b prod.' gbp >= good=3megHz bad=1megHz soft using {
      set RI = 10k
      set RF = 2meg
      FREQ = 10kHz
      gbp = FREQ * vm(3)
      set RI = 10meg
      set RF = 1
      }

   constraint 3 'slew rate' slewrate >= good=5 bad=3 soft using {
      TIME = .5us
      slewrate = vtr(3)/.5
      }

end_ineq
```

The *findroot* command in the *using* statement block of constraint 1, along with the function *straighten* at the top of the file, serve to search for the unity-gain (0db) frequency of the circuit, so that the phase margin can be evaluated as the phase at this frequency. This technique was shown in section 5.1.2.

The file contains several numerical values whose determination will be explained in the next subsection: the two end point frequencies (1megHz and 20megHz) of the *findroot* call, the 10kHz value of *FREQ* in constraint 2, and the .5μs *TIME* value in constraint 3.

Problem "FI" file. File *xampFI* contains the formulation of the settling time constraint; it contains upper and lower bounds that keep the transient output waveform within .1% of its final value of 5 volts over the range from the required settling time to the final simulation time:

```
prob_function fineq

   constraint 1 'setlng top' vtr(3)-5 <= good=.005v bad=.01v soft
   for_every TIME from 1.5us to Tstop initially by .02us

   constraint 2 'setlng botm' vtr(3)-5 >= good=-.005v bad=-.01v soft
   for_every TIME from 1.5us to Tstop initially by .02us

end_fineq
```

One consideration above is the choice of $2\mu s$ for the value of *Tstop* (see the "S" file); this consideration is explained in the next subsection. Indeed, the final transient analysis time was made a variable just so it would be easy to modify without having to edit any problem description files. Note that we do not use the objective/constraint combination technique from section 5.1.2 since we are not trying to minimize the settling time but only constrain it to be less than $1.5\mu s$.

Starting DELIGHT.SPICE. With the problem description files complete, DELIGHT.SPICE may be started:

```
DELIGHT.SPICE '<memspice>'
DELIGHT: Restoring from <memspice> ...
Identifier: SPICE Basic Memfile with Precompiled Algorithm "Afdmlfd"

                ***** Welcome to DELIGHT *****
   A General Purpose Interactive Computing System with Graphics
                            for
 Optimization-Based Computer-Aided-Design of Engineering Systems.
                      Developed by the
         Optimization-Based Computer-Aided-Design Group
                    University of California
                      Berkeley, Ca.  94720.
```

The *solve* command may be executed immediately:

```
1> solve xamp
including xampS       (3sec)
Unknowns are in X(5), 20 past values stored.
-------------------------------------------
including xampM       (52sec)
including xampI       (74sec)
including xampFI      (94sec)
```

This display represents a successful execution of *solve*. If instead this command produced one or more error messages, the designer would have to resolve all of

them by analyzing them and making appropriate corrections in the problem description files. The *solve* command would then have to be repeated until it completed without errors.

Checking the Problem Description Files

Parameters. Before starting the optimization, several checks of the problem setup are done. One easy check is to verify that the design parameters and settable parameters have the correct initial values:

```
1> printdp
No  Name        Value         Variation
1   QCS2_A      1.00000       .5000
2   CC          2.50000e-11   2.000e-11
3   RE1         1.00000e+3    1.000e+2
4   QI1_A       1.00000       .2000
5   QAL1_A      1.00000       .2000
1> printsp
Name        Value
RE2         1.00000e+3
QI2_A       1.00000
QAL2_A      1.00000
VINV_M      1.00000
VNINV       0.00000
RI          1.00000e+6
RF          1.00000
RSW1        1.00000
RSW2        1.00000e+6
```

The *identify* command of section 4.5.4.1 is now demonstrated. It shows the problem name being solved, the number of each type of constraint, and the algorithm and its sub-procedure names: this algorithm is the preselected enhanced feasible directions algorithm which has already been RATTLE compiled for DELIGHT.SPICE and stored in the starting memfile, "<memspice>" (see section 4.8 for more on memfiles.)

```
1> identify
PROBLEM: xamp
  5  Parameter(s)
  2  Multicost Objective(s)
  3  Inequality Constraint(s)
  2  Functional Inequality Constraint(s)
ALGO: Afdmlfd
  main_loop:  Mfdirmhs
  direction:  Dfdmlfd
  stepsize:   Sfdirmhs
  output:     Ophas123
  graphics:   GPScomb
```

Circuit functionality. Clearly, in the preliminary design of the circuit or in the preparation of the circuit file, errors can be made which result in gross malfunctioning of the circuit. In general, DELIGHT.SPICE cannot be expected to correct such conditions. It behooves the designer to check that with the initial design parameter values, the circuit is functioning properly. This is done by executing commands that request appropriate specific circuit analyses, and evaluating the results. But this is not the only reason to "manually" request circuit analyses. Another reason is that there are almost always assumptions made concerning the behavior of the specified circuit that are essential to the validity of the "measurement techniques" implicit in the problem description files. These assumptions must be checked by the designer, not only at the outset, but in general, throughout the optimization. The "measurement" assumptions of this example and their verification are discussed shortly, after showing how the DC node voltages are checked.

An examination of the DC bias solution node voltages is the first step in checking if the circuit is functioning properly. This can be done by first printing any DC *simulation_output* which forces a SPICE DC solution:[7]

[7] If no DC *simulation_output* has been declared, the variable *FREQ* may be set and an AC *simulation_output* may be printed since an AC analysis in DELIGHT.SPICE first triggers a DC bias solution.

```
1> print vdc(2,1)
Dc-5.379e-5
```

The characters "Dc" that are seen on the screen indicate that SPICE is presently performing a DC analysis. Similarly, "A" is printed during an AC analysis at a particular frequency (the present value of variable *FREQ*), while "Tr" is printed during a transient analysis. A display of the resultant DC node voltages is requested using the *probe* command from section 5.1.3:

```
1> probe nv
 Node Voltage      Node Voltage      Node Voltage      Node Voltage
   0) 0.00000        1) 5.37907e-5     2) 0.00000        3) 4.45617e-5
   4) 1.50000e+1     5) 1.49983e+1     6)-1.50000e+1     8) 1.43181e+1
   9) .959258       10)-1.43584e+1    11) .660234       12) .660191
  13)-1.37984e+1    14)-1.37818e+1    15)-1.43421e+1    16) 0.00000
  17) 4.91762e-5    18) 4.45617e-5
```

These values appear to represent normal quiescent behavior of the operational amplifier. Note that it was not necessary to precede the *print vdc(2,1)* command above by any commands to set test circuit settable parameters because their base values (from the circuit file) were appropriate for the DC measurement that was desired.

Validity of the slew rate measurement technique. The "measurement technique" assumptions of this example relate principally to the "I" file. First note that for the slew rate constraint (constraint I3), it is assumed that the initial part of the output response consists of an ideal ramp, and that its slope can be calculated by sampling the waveform at *TIME*=.2μs. To check these assumptions, a plot of the transient response of the output is requested. No settable parameters have to be modified; what is required is simply:

```
1> plot vtr(3) vs TIME from 0 to 2us by .05us
------ Compiling plot loop ------
```

This produces the plot shown in figure 5.3. The waveform obtained is clearly not

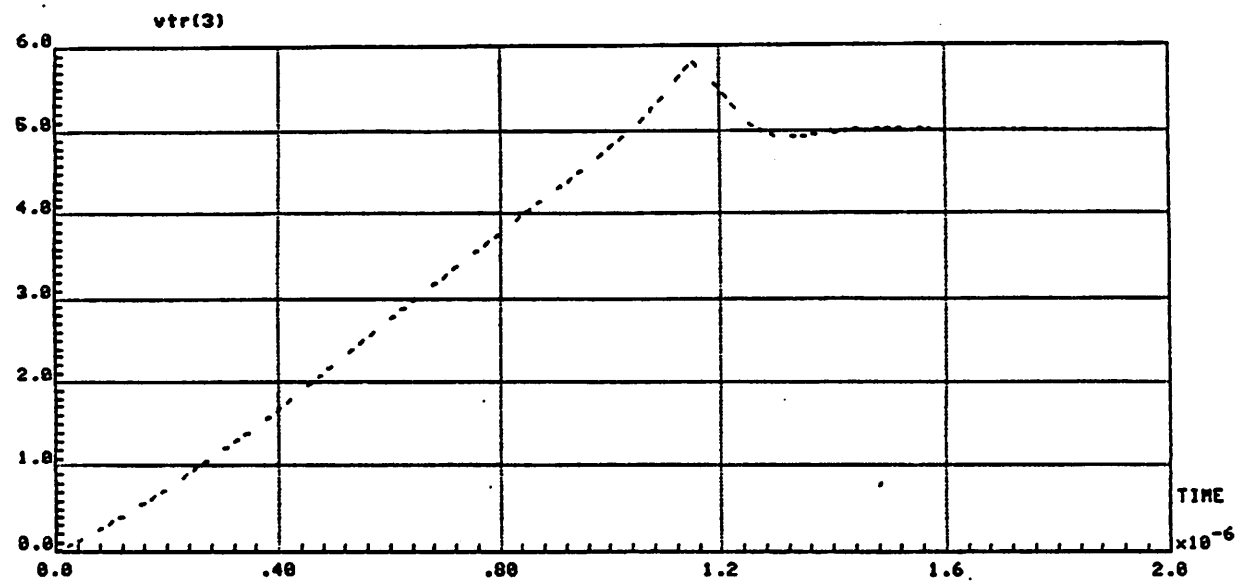


Figure 5.3. Response to 5 Volt Step Input, Measured Using Config. of Figure 5.2-d.

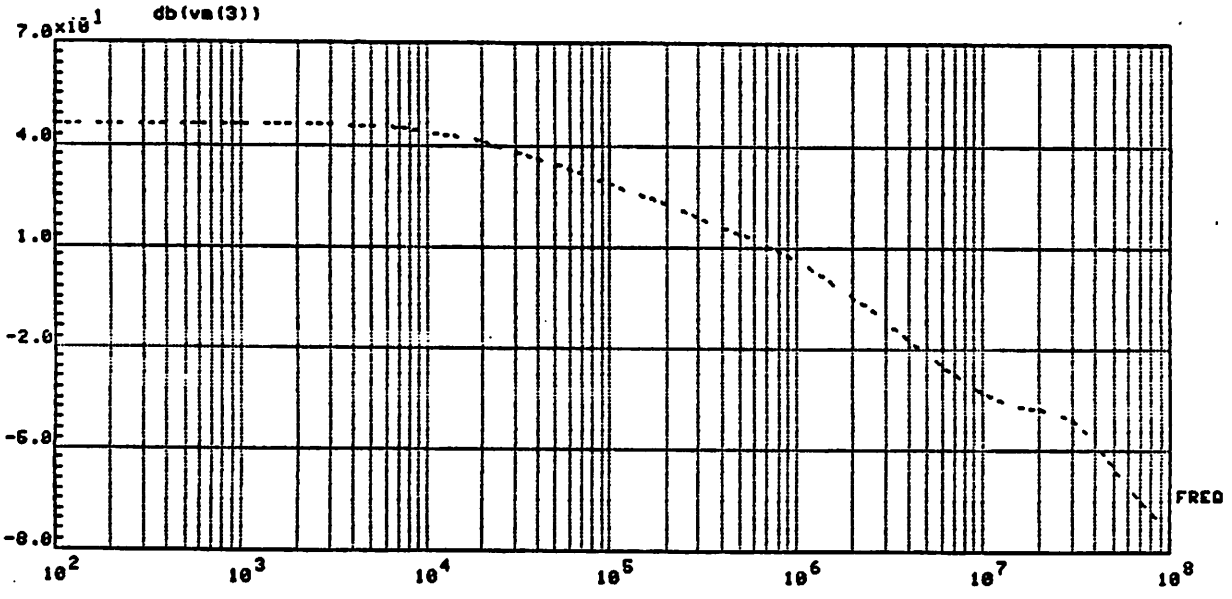


Figure 5.4. Bode Plot of Voltage Gain, Measured Using Config. of Figure 5.2-d.

a perfect ramp, but this is not cause to doubt the circuit operation. Instead it is recognized that the ideal ramp assumption is implicit in the definition of slew rate, and in this case the definition seems to be unrealistic, at least for the initial design parameter values. The underestimate of slew rate due to the sagging ramp can be tolerated, but what is essential is that the sample time be within the slewing portion of the waveform. This is the case here for the .5μs sample time in the "I" file, but as the design parameters are automatically adjusted, the transient response should improve and this assumption may become violated. The assumption can always be checked after each optimization iteration, but just to be conservative, the .5μs sample time is reduced to .2μs. Thus, the "I" file lines,

```
constraint 3 'slew rate' slewrate >= good=5 bad=3 soft using {
   TIME = .5us
   slewrate = vtr(3)/.5
   }
```

are replaced by

```
constraint 3 'slew rate' slewrate >= good=5 bad=3 soft using {
   TIME = .2us
   slewrate = vtr(3)/.2
   }
```

It is also assumed that the variable *Tstop*, set in the "S" file and used in the "FI" file, is sufficiently large that the maximum excursion of the waveform from its final value, after 1.5μs, occurs before the time *Tstop*. With a *Tstop* of 2μs, this appears to be satisfied, and will probably remain satisfied as the optimization progresses.

Validity of the AC measurement techniques. There are two assumptions about the AC constraints in file *xampI*. In the computation of gain-bandwidth product (constraint 2), one assumption is that the frequency at which the gain is

calculated is above the first corner frequency of the closed-loop configuration, so that the closed-loop and open-loop gains are very nearly the same, but below the first non-dominant pole frequency. The ratio RF/RI (=200) was chosen to give a large frequency range in which both these conditions are satisfied, and the absolute values of their resistance were chosen to be large values to prevent significant output loading.

In the phase margin constraint (constraint 1), since the *from* and *to* values of the *findroot* command must bracket the desired root, a second assumption is that the unity-gain frequency lies between 1megHz and 20megHz.

In order to check these two assumptions the AC response is plotted:

```
1> set RI=10k
1> set RF=2meg
1> plot db(vm(3)) vs FREQ from 100 to 100megHz dec 8
------ Compiling plot loop ------
```

From figure 5.4, the first assumption, that the frequency at which the gain is calculated for the gain-bandwidth product be above the first corner frequency, is not satisfied. Thus, the "I" file line,

FREQ=10kHz

is replaced by,

FREQ=100kHz

For the second assumption, the unity-gain frequency is about 1.5 megHz, which is inside the 1 megHz to 20 megHz bracket of the *findroot* command. However, in case DELIGHT chooses at some point to improve other performance functions and in so doing temporarily degrades the frequency response, the lower bracket value is decreased as follows. The "I" file line,

```
findroot straighten(mu) vs mu from log10(1megHz) to log10(20megHz)
```

is replaced by,

```
findroot straighten(mu) vs mu from log10(.5megHz) to log10(20megHz)
```

Settable parameters RI and RF must be now restored to their base values. But first, we demonstrate how to display all (or selected) AC node voltages. These are from the most recent SPICE AC analysis and value of *FREQ* and in this case would be *FREQ*=100megHz from the *plot* command above:

```
1> probe acnv
 Node Magnitude     Phase          Node Magnitude     Phase
   0) 1.00000e-20  0.00000          1) .113553       -8.06453
   2) 1.00000e-20  0.00000          3) 2.04024e-4     7.40084e+1
   4) 1.00000e-20  0.00000          5) 1.09870e-6    -1.78920e+2
   6) 1.00000e-20  0.00000          8) 1.93069e-7     9.90045e+1
   9) 2.81721e-3  -9.72725e+1      10) 5.15565e-4     4.62364
  11) 8.73046e-2  -1.05606e+1      12) 6.45852e-4    -9.80959e+1
  13) 2.75535e-3   7.87632        14) 3.56210e-4     6.56757e+1
  15) 8.46624e-5   7.47483e+1     16) 1.00000        0.00000
  17) 2.04031e-4   7.39926e+1     18) 2.04024e-4     7.40084e+1
1> set RI=1meg
1> set RF=1
```

Note that the four (near) zero AC magnitudes above are all of nodes connected to voltage sources. After performing another *solve* command (so that DELIGHT.SPICE considers all the changes made to the problem description files), optimization is ready to begin.

Running the Optimization

We first give the command *Pcomb* to display the initial performance comb, shown in figure 5.5-a. This shows that the offset voltage objective is very "bad" since its comb tooth ends with an arrow and is thus out of the comb range. Also, the gain-bandwidth constraint has an arrow pointing in the opposite direction since it is well satisfied. On a color terminal, the good and bad vertical lines would be green and red respectively. However, this output was produced on the

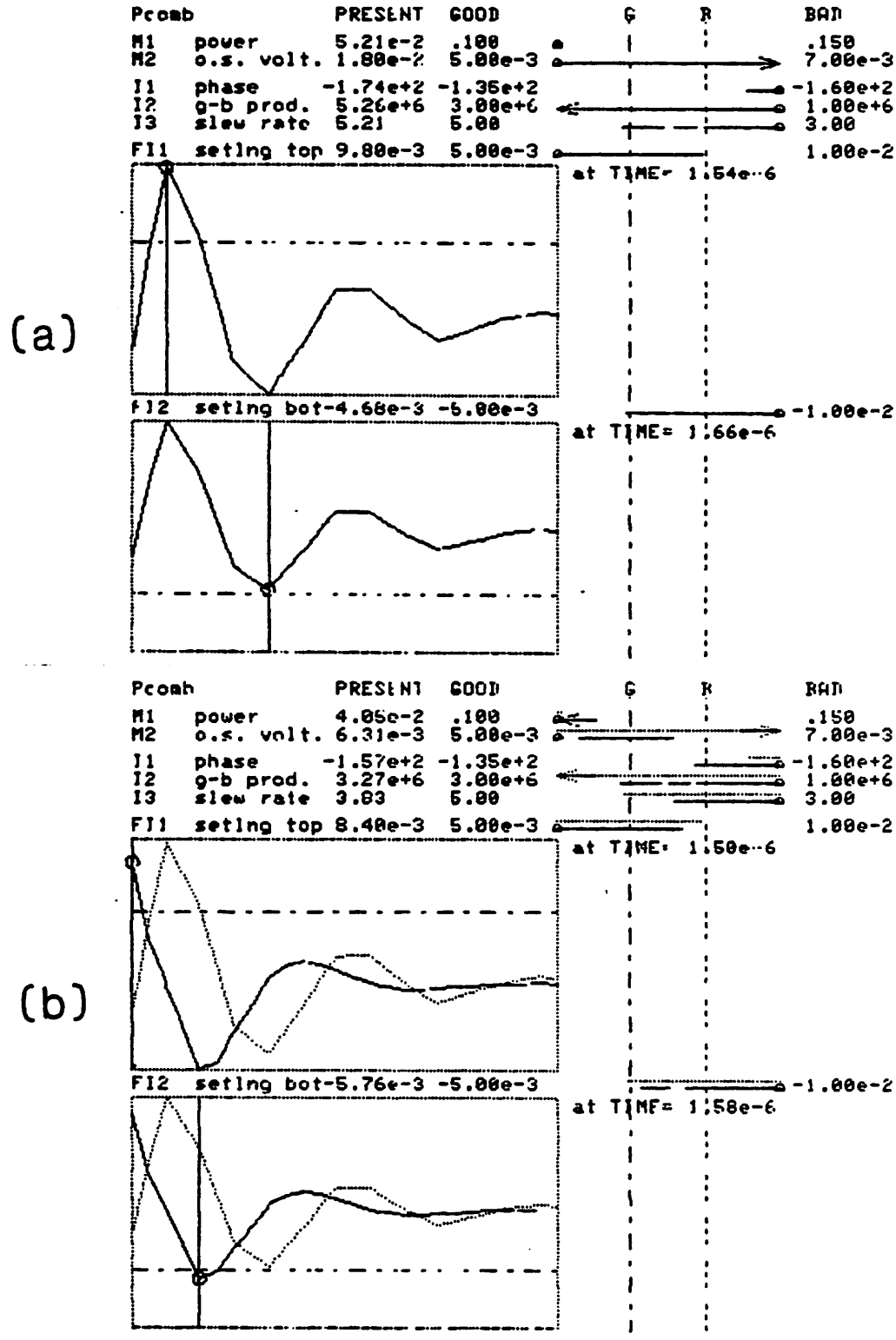


Figure 5.5. Graphical Output for the Bipolar Opamp Optimization.

HP2648a black and white graphics terminal in which colors are simulated using various dashed line types. Nevertheless, the line types for the good and bad curves on the functional plots correspond to those of the vertical good and bad lines. This distinguishes the good curve from the bad curve for each functional objective or constraint[8].

Optimization is started by typing the command *run 5* to request five optimization iterations. The following standard algorithm output appears on the screen. To save space, we only show here the first two design parameters for the last four iterations since the others undergo little or no change:

```
Iter=0 Phase2 MxM+SC= 6.476
Parameter    Value     %wrt 0 Prev
 1 QCS2_A    1.000        0%   0%
 2 CC        2.500e-11    0%   0%
 3 RE1       1.000e+3     0%   0%
 4 QI1_A     1.000        0%   0%
 5 QAL1_A    1.000        0%   0%
Iter=1 Phase2 MxM+SC= 2.040 k=0
Parameter    Value     %wrt 0 Prev
 1 QCS2_A    .5000      -50% -50%
 2 CC        2.500e-11    0%   0%
Iter=2 Phase2 MxM+SC= .9562 k=1
Parameter    Value     %wrt 0 Prev
 1 QCS2_A    .3749      -63% -25%
 2 CC        3.368e-11   35%  35%
Iter=3 Phase2 MxM+SC= .9208 k=3
Parameter    Value     %wrt 0 Prev
 1 QCS2_A    .3172      -68% -15%
 2 CC        3.270e-11   31%  -3%
Iter=4 Phase2 MxM+SC= .8900 k=2
Parameter    Value     %wrt 0 Prev
 1 QCS2_A    .3479      -65%  10%
 2 CC        3.444e-11   38%   5%
Iter=5 Phase2 MxM+SC= .8678 k=3
Parameter    Value     %wrt 0 Prev
 1 QCS2_A    .3367      -66%  -3%
 2 CC        3.487e-11   39%   1%
```

For all iterations, phase II is indicated and the initial maximum of all objectives and soft constraints, normalized as shown in section 4.4.1.2, is 6.476 (shown by "MxM+SC" above). Since the normalization causes 0 to correspond to good

[8] The bad curves in figure 5.5-a are difficult to see since they coincide with the top or bottom edges of the box drawn.

values and 1 to bad, the value of 6.476 is *very* bad. During the five requested iterations, the normalized maximum continually decreases (as guaranteed by the algorithm) reaching a value of .8678 as seen above. This implies that the rightmost comb tooth be just to the left of the vertical bad line, which is seen by the dark comb teeth on the requested performance comb in figure 5.5-b. By comparing adjacent light and dark (previous and present) teeth, this comb also reveals that the offset voltage has improved by coming considerably down and the phase margin improved by coming up but at the expense predominately of a decrease in gain-bandwidth product and slew rate. (In fact, without even looking at the numeric values, we know the slew rate, for example, has gotten worse since its dark tooth is further to the right than its light tooth. Also, we know its value has decreased since the dark tooth is closer to the small circular dot.) These changes are in accord [55] with the changes in the design parameters shown above: the area of bias transistor QCS2 and hence the current to the input differential pair has been more than cut in half while compensation capacitor has almost doubled. Also shown for each iteration in the standard algorithm output is the trial k in the step-size computation $Beta^k$.

After requesting five more optimization iterations, we notice right after the first that the algorithm has taken a very small step. This is seen in the right column of the algorithm output which shows the percentage change in the design parameter values with respect to the previous iteration. Thus we immediately press the special interrupt key once to generate a soft interrupt (see section 4.2.6), thereby allowing the algorithm to complete the current iteration and stop at the "major stopping point". We now issue the commands *setgood M2 = 7mv* and *setbad M2 = 10mv* to relax the offset voltage objective since, as mentioned earlier, the DC specifications are not of primary concern.

The performance comb is then redisplayed, as shown in figure 5.5-c. After requesting one iteration, we get the following algorithm output:

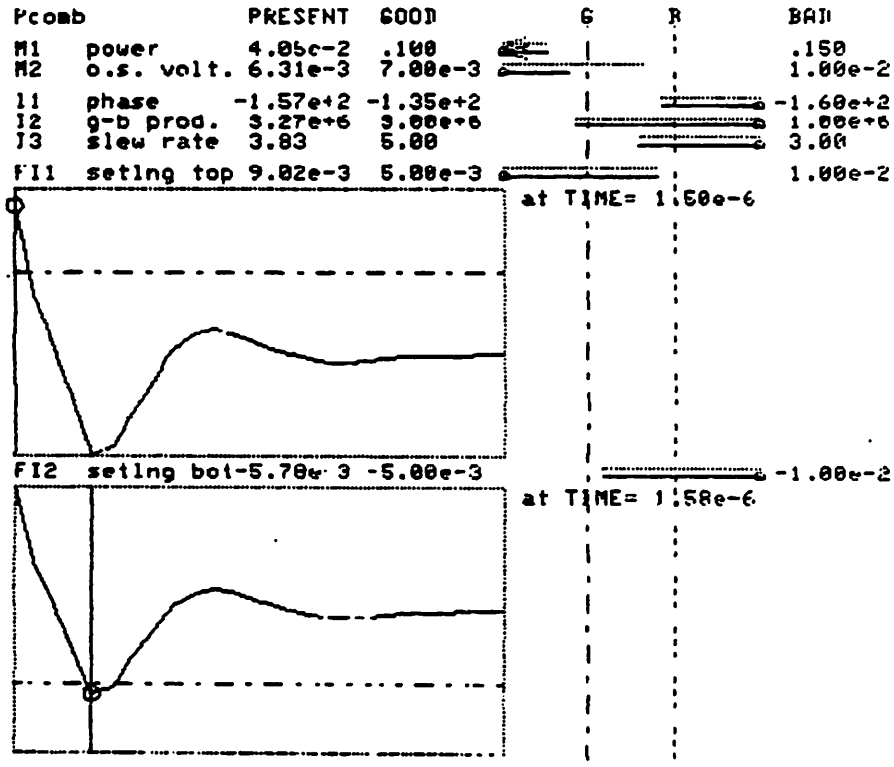
```
Iter=7 Phase2 MxM+SC= .7202 k=1
Parameter    Value      %wrt 0 Prev
 1 QCS2_A    .4480        -55%  33%
 2 CC        4.333e-11     73%  24%
 3 RE1       1.101e+3      10%   1%
 4 QI1_A     .9910         -1%   0%
 5 QAL1_A    .9908         -1%   0%
```

The relaxation of the offset voltage has caused the algorithm to emphasize the constraints of the two rightmost dark comb teeth in figure 5.5-c, the phase margin and upper settling time constraints. Coincidentally, both of these circuit properties are improved by the increase in the input pair current (through the increase in the area of bias transistor QCS2) and the increase of the compensation capacitor CC shown above. After running two more iterations and observing negligible change in the design parameters, our optimization session is temporarily suspended due to ..... lunch[9]!

After restarting DELIGHT, reissuing the *solve* command, and restoring all previous design parameter and good and bad values, we decide that the last two parameters shown above should have more effect than they do on the circuit performance. To recall what their variations were, we issue the *printdp* command. We then increase their variations as shown:

[9] Before leaving DELIGHT, the command *savedp into savefile* is issued to save all present design parameter and good and bad values into a file that can simply be *included* after restarting DELIGHT to restore the state of the optimization.

(c)



(d)

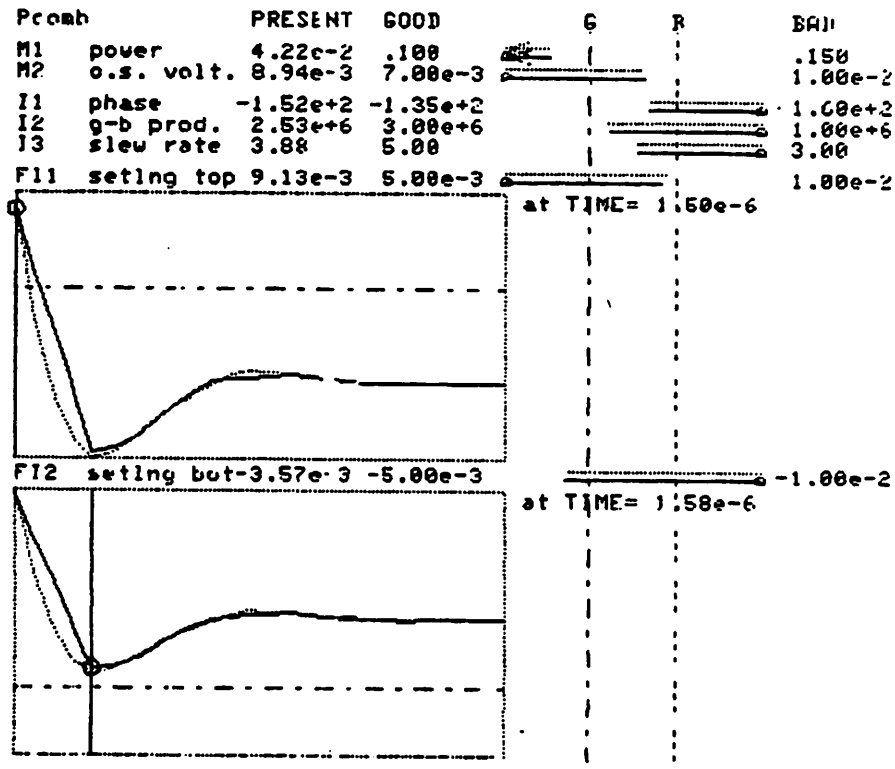


Figure 5.5. (continued)

```
2> printdp
No  Name        Value         Variation
1   QCS2_A      .436859       .5000
2   CC          4.38018e-11   2.000e-11
3   RE1         1.10137e+3    1.000e+2
4   QI1_A       .990922       .2000
5   QAL1_A      .991777       .2000
2> setvariation QI1_A= .6
2> setvariation QAL1_A = .6
```

After a few more iterations, the design parameters do not change and we display the performance comb shown in figure 5.5-d It appears that the offset voltage objective is again "holding up progress" so we again relax it via *setgood M2 = 10mv* and *setbad M2 = 15mv*[10]. A request for another iteration results in good progress. The normalized maximum has fallen to .5357, resulting in the rightmost comb tooth in figure 5.5-e being almost halfway between good and bad lines. The fourth and fifth design parameters have also started changing as seen below. This shows the success of our previous adjustment of their variations:

```
Iter=11 Phase2 MxM+SC= .5357 k=0
Parameter   Value     %wrt 0 Prev
 1 QCS2_A   .6509      -45%  26%
 2 CC       5.201e-11  108%  19%
 3 RE1      1.196e+3    20%   1%
 4 QI1_A    .8692      -13% -11%
 5 QAL1_A   .9569       -4%  -3%
```

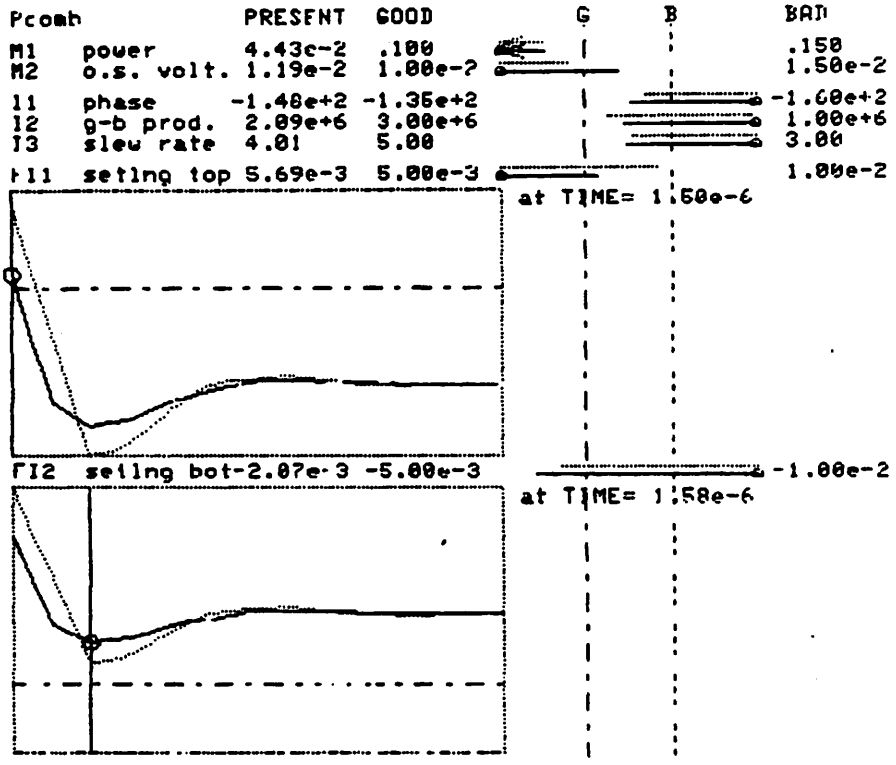
After several more iterations of optimization, it continues to appear that the simultaneous achievement of all the specifications is impossible. Thus we make a major emphasis on improving the gain-bandwidth product by doubling its constraint good value and also increasing its bad value—but without any further relaxation of the offset voltage objective:

```
2> setgood I2 = 6megHz
2> setbad I2 = 15megHz
```

Two optimization iterations later, the gain-bandwidth product has improved

[10] If 10mv seems large for an operational amplifier, recall that this is the worst case offset voltage; the rms value of the offset will be significantly less.

(e)



(f)

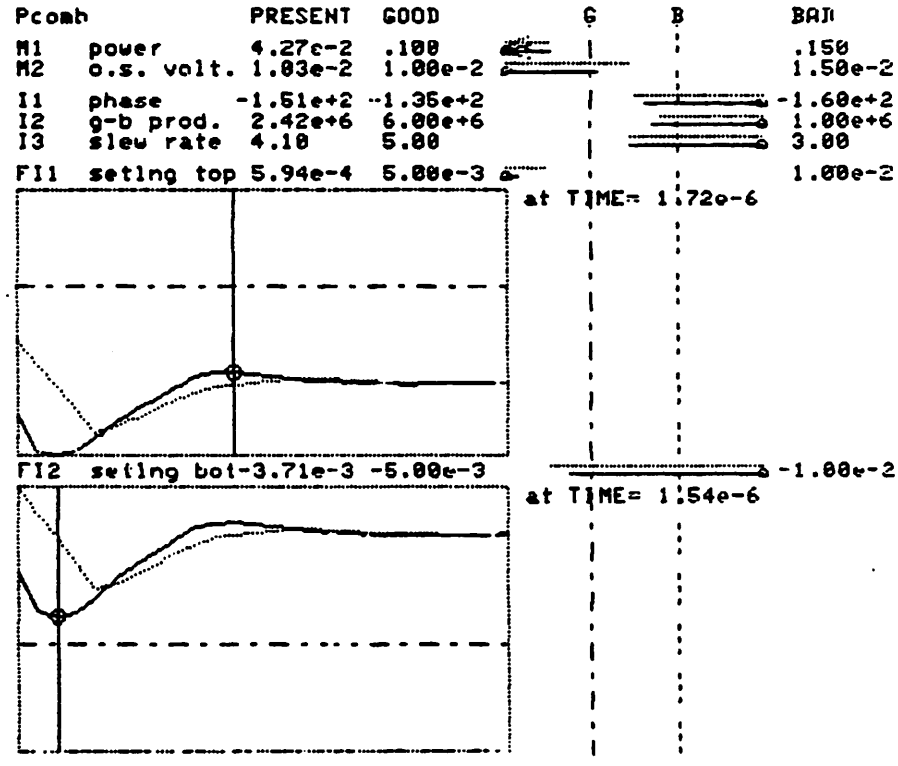


Figure 5.5. (continued)

slightly as shown in the performance comb in figure 5.5-f. We also have the following output:

```
Iter=16 Phase2 MxM+SC= .7160 k=1
Parameter    Value     %wrt 0 Prev
 1 QCS2_A    .4647       -54% -19%
 2 CC        4.306e-11    72%  -6%
 3 RE1       1.226e+3     23%   0%
 4 QI1_A     .6852        -31%  0%
 5 QAL1_A    .9676         -3%  0%
```

Our opamp session ends with the conclusion that although the circuit performance shown above does not satisfy the specifications, we *have* demonstrated both performance tradeoffs by adjusting good and bad values and the successful effect of parameter variations, and we have obtained significant *performance improvement.*

Design of a CMOS Inverter Chain

In our second design example we optimize the performance of a chain of CMOS inverters used to drive a relatively large capacitive load. For this we consider the following three objectives:

1. Minimize the overall delay,
2. Minimize the integral of the supply current during switching (which effectively minimizes the average power used by the circuit), and
3. Minimize the total area of the gates of all the MOS transistors.

By adjusting good and bad values, we emphasize each of these objectives individually. The geometries of the first inverter are held fixed while all other gate geometries are design parameters. (If the first inverter gates were also designable, the delay could be reduced to an arbitrarily small value by simply increasing all geometries without bound.) The purpose of this first optimization run is to

verify the theoretical results for delay minimization explained in the next paragraph. The purpose of emphasizing the other two objectives is to see how the results differ from those of the first optimization. This optimization problem and the various tradeoffs involved were pointed out by R. Newton and D. Hodges at the University of California, Berkeley.

For a chain of MOS inverters, an analysis of the number of stages that minimizes the overall delay for a given ratio C_{ratio} of load capacitance to total gate capacitance on the first inverter is given by Mead and Conway in [100]. By assuming that the size of each successive MOS gate pair is a factor of f larger than the previous, they show that the minimum delay is achieved when $f=e$, the base of natural logarithms, and that the optimum number of stages is $N=\ln(C_{ratio})$. For our problem we first choose $N=3$ inverters and then calculate the ratio that would require this N as $C_{ratio} = e^3 \simeq 20$. Using parameter values from our SPICE *.MODEL* lines shown below and techniques from [65], we derive a gate capacitance value of $4 \cdot 10^{-4} pF/\mu^2$. Using this value and and the dimensions of the first CMOS pair $L_n=4\mu$, $W_n=8\mu$, $L_p=4\mu$, and $W_p=16\mu$, the load capacitance corresponding to this C_{ratio} is calculated as $C_{load} = C_{ratio} \cdot (Area_n + Area_p) \cdot 4 \cdot 10^{-4} = .768pF$. For CMOS inverters, our SPICE *.MODEL* lines below indicate the well known property that the mobility of P-type material is about half that of N-type material. Thus, for the same channel length, the channel width of the P-type devices should be twice that of the N-type devices to give equal rise and fall delays. In our problem description "S" file, N-type device widths are design parameters while P-type widths are settable parameters that track twice their corresponding N-type width using the *track_param_update* feature of section 5.1.3.

The circuit description file set up for SPICE is:

```
* CMOS inverter chain example.
Vdd     8 0 5volts
Rseries 8 9 1ohm
Gbuffer  0 10 8 9  1p
Cinteg  10  0      1p
Rpath   10  0      1e9
Cload 4 0  .768pF
M1p 2 1 9 9 p  l=04u  w=16u
M1n 2 1 0 0 n  l=04u  w=08u
M2p 3 2 9 9 p  l=08u  w=32u
M2n 3 2 0 0 n  l=08u  w=16u
M3p 4 3 9 9 p  l=16u  w=64u
M3n 4 3 0 0 n  l=16u  w=32u
.model n nmos vto=.9 nsub=2E15 tox=.1u uo=800
+             level=1 vmax=1e5 cgso=245p cgdo=245p
.model p pmos vto=-.9 nsub=2E15 tox=.1u uo=400
+             level=1 vmax=1e5 cgso=245p cgdo=245p
vin 1 0 pulse 0 5 3nsDelay
.tran .4ns 20ns
.end
```

The corresponding circuit diagram for the three cascaded inverters is shown in figure 5.6. The elements at the top of figure 5.6 are for the integral of the supply current as explained shortly. The initial values of the device geometries are chosen so that each succeeding stage doubles the gate areas of the previous stage, i.e., initially $f=2$ instead of the optimal $f=e$. The values are shown in the following table:

MOS Device	Length	Width
M1n	4μ	8μ
M1p	4μ	16μ
M2n	8μ	16μ
M2p	8μ	32μ
M3n	16μ	32μ
M3p	16μ	64μ

The three objectives are formulated as follows. Delay minimization is accomplished using the objective/constraint combination technique of section 5.1.2: we declare a non-simulation design parameter called *Tdelay* and minimize it subject to the functional constraint that the (falling) final inverter output response to a step input remain below 2.5 volts (half of *Vdd*) for every time from *Tdelay* to the final analysis time. Thus we are defining delay using the popular

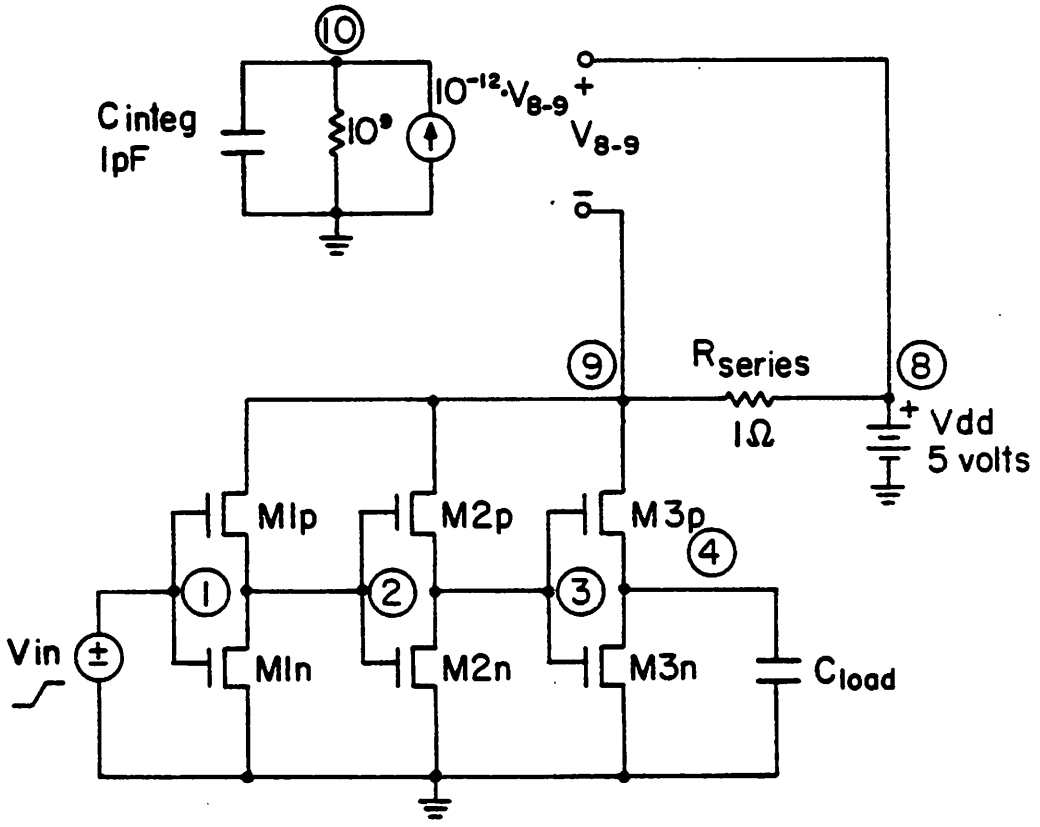


Figure 5.6. CMOS Inverter Chain Circuit Diagram for the Second Design Example.

"50-percent point" notion. The integral of the supply current is calculated during the circuit simulation by using the auxiliary controlled source and capacitor C_{INTEG} shown in figure 5.6. The integral is equal to the voltage at the top node of the capacitor (node 10) at the final transient analysis time and is in units of coulombs, e.g., we would write "2pC" for $2 \cdot 10^{-12}$ coulombs. (The purpose of the 10^9 ohm resistor in parallel with C_{INTEG} is to avoid the SPICE circuit error message "NO DC PATH TO GROUND"; it does not affect the integration since its time constant of one millisecond is much greater than the final integration time of 20ns.) The total gate area simply sums the product of the length and widths of all MOS transistors. The following lists our "M" file for this problem description:

```
prob_function multicost

   objective 1 'Delay' minimize Tdelay good=3ns+3ns bad=10ns

   objective 2 'Int Current' minimize vtr(10)-vdc(10) good=10pC bad=15pC using
      TIME = 20ns

   objective 3 'Area' minimize Area good=10k*(1u**2) bad=15k*(1u**2) using  {
      Area = 0000   +   4u * M1P_W   +   4u * M1N_W
      Area = Area   +   8u * M2P_W   +   8u * M2N_W
      Area = Area   +  16u * M3P_W   +  16u * M3N_W
      }

end_multicost
```

The good and bad values would normally be obtained using the uniform satisfaction/dissatisfaction rule of section 4.4.1.2. For some objectives, as in the bipolar amplifier example, this would be done after investigating the initial performance of the circuit. However, the good and bad values for the second and third objectives above are simply set to values much larger than the corresponding initial circuit performance. This is to allow the delay in objective number one to be given the greatest emphasis during the first optimization run.

After setting up the problem description files and verifying that the circuit is operational, we first plot the input and all inverter output waveforms

using the command *plot vtr(1) vtr(2) vtr(3) vtr(4) vs TIME from 0 to Tstop by .4ns*. The result is shown in figure 5.7-a in which the waveforms are identified using the circuit node numbers from figure 5.6. This plot shows that the initial 50-percent time delay to the chain output on node 4 is approximately 10.2ns[11]. We next give the command *Pcomb* to display the initial performance comb, shown in figure 5.7-b. This shows that the initial guess for design parameter *Tdelay* of 5ns has caused a hard constraint, namely, the functional constraint used for time delay, to be violated. Objectives *M2* and *M3* are also clearly seen to be well "satisfied", as desired. The initial value of the integral of power supply current is .391pC while the initial total area is 2020μ^2. Optimization is now ready to begin.

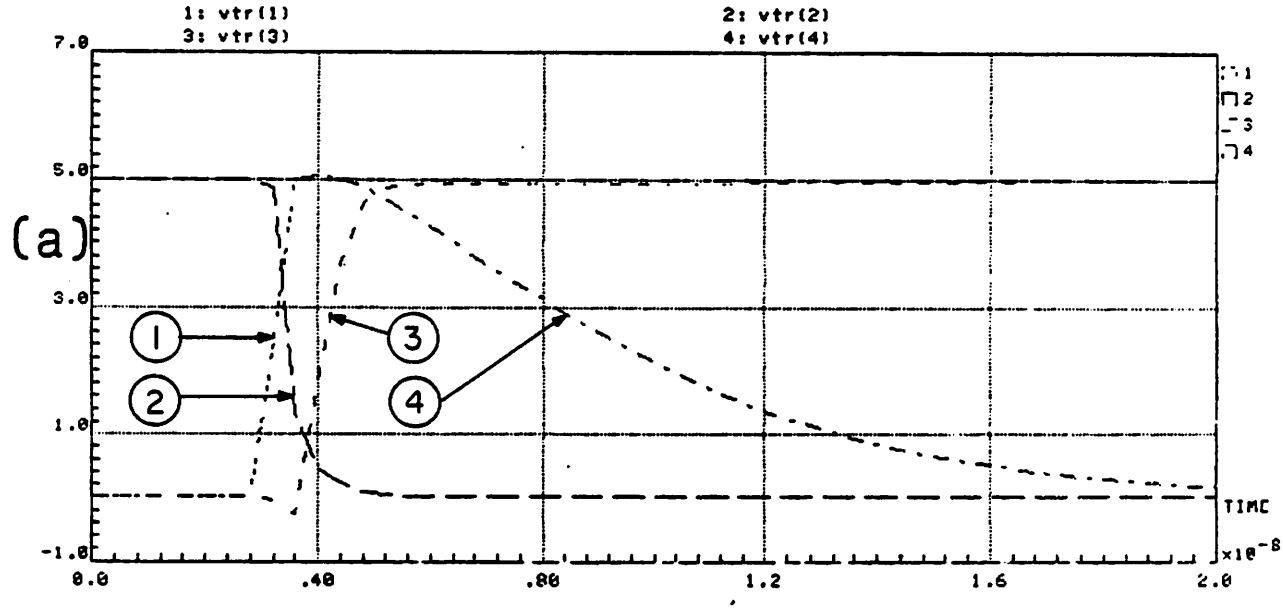
The command *run 1* is given to run one optimization iteration. The following standard algorithm output appears on the screen:

```
1> run 1
TrDc
Iter=0 Phase1 MxHard= 4.616
Parameter    Value     %wrt 0 Prev
 1 M2N_W     1.600e-5      0%   0%
 2 M3N_W     3.200e-5      0%   0%
 3 Tdelay    5.000e-9      0%   0%
LFD<TrTrTr>(k=0)Tr(k=-1)Tr(k=-2)TrTrDc
Iter=1 Phase2 MxM+SC= 1.745 k=-1
Parameter    Value     %wrt 0 Prev
 1 M2N_W     1.747e-5      9%   9%
 2 M3N_W     3.184e-5     -1%  -1%
 3 Tdelay    1.298e-8    160% 160%

Interrupt...
2>
```

For iteration 0, phase I is indicated and the maximum hard constraint value, normalized as shown in section 4.4.1.2, is 4.618. Since this normalization causes 0 to correspond to good values and 1 to bad, the value of 4.618 is *very* bad. After one iteration, phase II is entered since all hard constraints become satisfied.

[11] For simplicity, in this section we do not subtract off the 50-percent time of the input waveform, which from the figure is approximately 3.02ns.



(b)

Pcomb		PRESENT	GOOD	G	B	BAD
- A Hard Constraint is Violated -						
M1	Delay	5.00e-9	6.00e-9			1.00e-8
M2	Int Curren	3.91e-13	1.00e-11			1.50e-11
M3	Area	2.02e-9	1.00e-8			1.50e-8

(c)

Pcomb		PRESENT	GOOD	G	B	BAD
M1	Delay	1.30e-8	6.00e-9			1.00e-8
M2	Int Curren	4.02e-13	1.00e-11			1.50e-11
M3	Area	2.04e-9	1.00e-8			1.50e-8

(d)

Pcomb		PRESENT	GOOD	G	B	BAD
M1	Delay	6.87e-9	6.00e-9			1.00e-8
M2	Int Curren	1.19e-12	1.00e-11			1.50e-11
M3	Area	5.92e-9	1.00e-8			1.50e-8

Figure 5.7. Graphical Output for the CMOS Inverter Chain Optimization.

ehe trial k in the step-size computation $Beta^k$ is shown in parenthesis, intermixed with the "Tr" and "Dc" characters that indicate when SPICE is performing transient and DC analyses. The value of k that satisfied the step-size test is shown to be -1. The obvious reason why the one hard functional constraint has become satisfied in just one iteration is that it is over the range from *Tdelay* to *Tstop* and *Tdelay* as seen above has greatly increased. The new maximum of all normalized objectives and soft constraints ("MxM+SC" above) is 1.745. This is also seen by the position of the first comb tooth on the corresponding performance comb in figure 5.7-c.

After giving the command *run 5*, observing significant progress, and hence giving three more *run 1* commands, the maximum of the normalized objectives and soft constraints falls to .3684, as seen in the following output:

```
Iter=9 Phase2 MxM+SC= .3684 k=2
Parameter    Value     %wrt 0 Prev
 1 M2N_W    2.554e-5    60%   0%
 2 M3N_W    6.999e-5   119%   0%
 3 Tdelay   7.473e-9    49%  -3%
```

After a few more iterations and performance combs, we reach a point at which our maximum normalized objective has fallen to .2165 and the design parameters are changing very slowly. The latter fact is seen in the right column below that shows the percentage change with respect to the previous iteration:

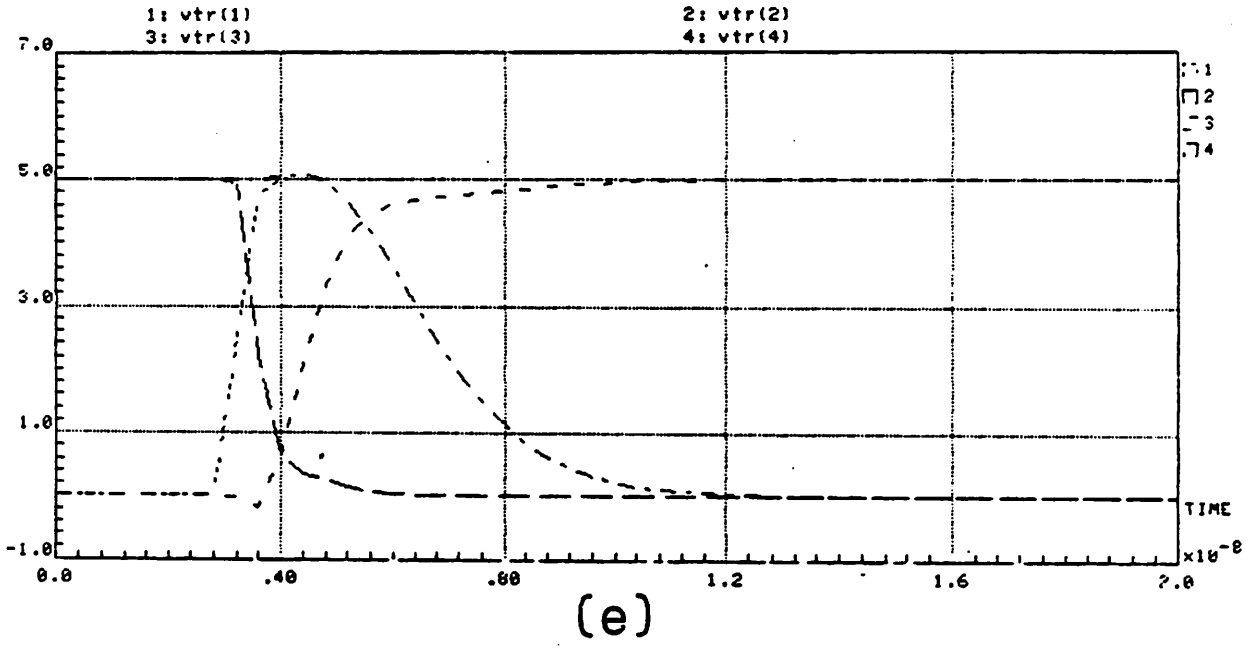
```
Iter=15 Phase2 MxM+SC= .2165 k=3
Parameter    Value     %wrt 0 Prev
 1 M2N_W    3.981e-5   149%   0%
 2 M3N_W    1.014e-4   217%   0%
 3 Tdelay   6.866e-9    37%  -1%
```

Therefore, we end this part of the optimization run. The performance comb in figure 5.7-d shows that the delay 6.87ns is fairly close to the good value of 6.0ns. This compares with 10.2ns initially. Also shown are the present values for the

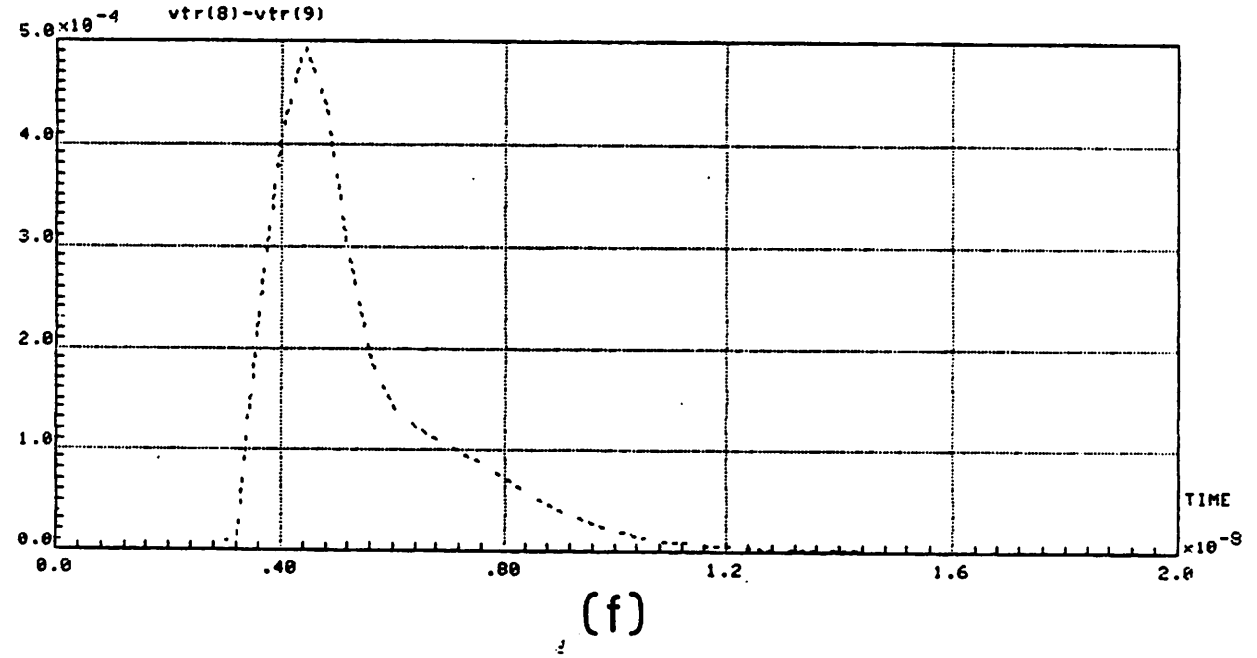
other objectives. The integral of current is 1.19pC (compared to .391pC initially) while the total area is $5920\mu^2$ (compared to $2020\mu^2$ initially). Of course, these tradeoffs are to be expected: to reduce delay, one has to "beef up" the transistors, which increases the transient power supply current spike. Figure 5.7-e shows a plot of the input and all inverter output waveforms.

Let us now compare the resulting device geometries with the theoretical optimal ones given earlier. Using the widths given above and the formula $Area = L_n \cdot W_n + L_p \cdot W_p$, we calculate the areas of each stage as $Area_1 = 4\mu \cdot 8\mu + 4\mu \cdot 16\mu = 96\mu^2$, $Area_2 = 8\mu \cdot 39.81\mu + 8\mu \cdot 79.62\mu = 955\mu^2$, and $Area_3 = 16\mu \cdot 101.4\mu + 16\mu \cdot 202.8\mu = 4867\mu^2$. The ratios of succeeding stage areas are then $Area_2 / Area_1 = 955/96 = 9.95$ and $Area_3 / Area_2 = 4867/955 = 5.1$, which are both considerable larger than the theoretical optimum ratio of $e = 2.71$. The difference can partly be attributed to the effect of source and drain capacitances and other MOSFET model parasitics not considered in the idealized analysis of Mead and Conway. More importantly, however, their analysis assumes that the gate capacitance is constant whereas in reality, gate capacitance is a strong function of the amount of charge under the gate and hence of the gate-to-source voltage.

We now turn our attention to emphasizing the integral of the supply current during switching. We first plot the supply current spike and its integral; these are shown in figures 5.7-f and 5.7-g. The present integral value of 1.19pC from the comb in figure 5.7-d is seen at the (TIME=20ns) right side of the latter plot. To achieve our desired emphasis, we set the good and bad values for objective 2 to a fraction of the present value using the commands *setgood M2 = .4pC* and *setbad M2 = .8pC*. The good and bad values for the delay objective are not changed so that these two objectives may compete.



(e)



(f)

Figure 5.7. (continued)

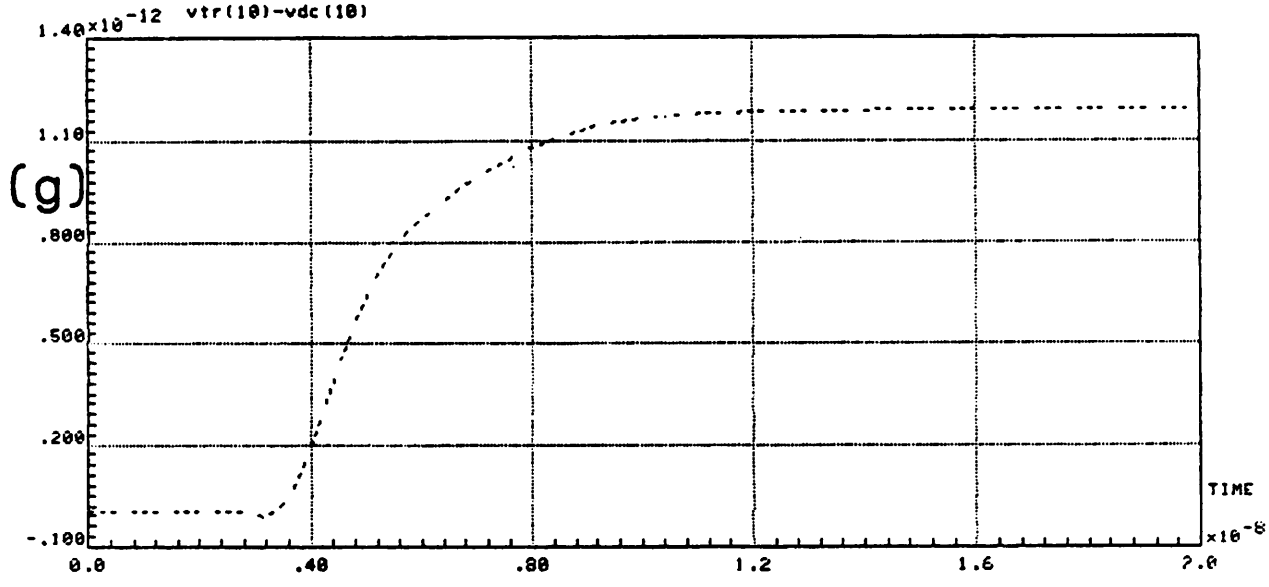
We first display the performance comb shown in figure 5.7-h to see the initial objective competition[12]. After running three iterations, we observe the expected decrease in both device widths and the increase in delay, as shown in the following output:

```
Iter=18 Phase2 MxM+SC= .7371 k=-2
Parameter    Value     %wrt 0 Prev
 1 M2N_W    1.025e-5    -36% -43%
 2 M3N_W    7.472e-5    134%  -8%
 3 Tdelay   8.139e-9     63%   8%
```

The corresponding performance comb is shown in figure 5.7-i. Another *run 3* command ends with an iteration in which a very small step is taken. The performance comb in figure 5.7-j reveals that the two objectives are in close competition. In general, at the optimal solution one would expect to either be in a local (or global) minimum of one of the objectives or have two or more normalized objective values equal, meaning one cannot improve without another becoming worse (see the discussion of *noninferior points* in section 4.4.1.2). The comb teeth in figure 5.7-j do not have precisely the same length but are very close—certainly both are ε-active and thus take part in the search direction computation. Thus, we end this second part of the optimization. In reducing the integral of the current from 1.19pC to .675pC, the delay has increased from 6.87ns to 7.92ns. Figures 5.7-k 5.7-l show the present supply current spike and its integral. Notice that the peak spike current has decreased from .5mA in figure (f) to .2mA in figure (k).

For our third optimization run, we completely remove the emphasis from the integral objective M2 as well as give emphasis to the area objective M3, so that delay and area alone may compete. After suitably adjusting the good and bad

[12] On this comb, the name of objective M2 as mysteriously changed from "Int Curren" to "Integral C" !



(g)

(h)

Pcomb		PRESENT	GOOD	G	R	BAD
M1	Delay	6.87e-9	6.00e-9			1.00e-8
M2	Integral C	1.19e-12	4.00e-13			8.00e-13
M3	Area	5.92e-9	1.00e-8			1.50e-8

(i)

Pcomb		PRESENT	GOOD	G	R	BAD
M1	Delay	8.14e-9	6.00e-9			1.00e-8
M2	Integral C	6.95e-13	4.00e-13			8.00e-13
M3	Area	3.93e-9	1.00e-8			1.50e-8

(j)

Pcomb		PRESENT	GOOD	G	R	BAD
M1	Delay	7.92e-9	6.00e-9			1.00e-8
M2	Integral C	6.75e-13	4.00e-13			8.00e-13
M3	Area	3.74e-9	1.00e-8			1.50e-8

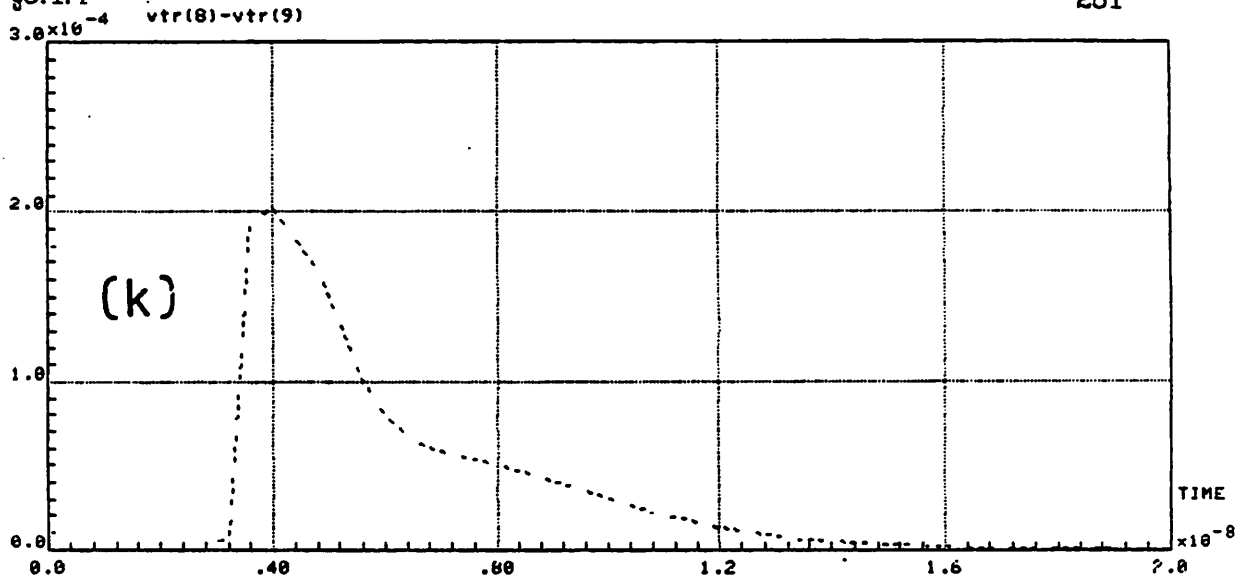
Figure 5.7. (continued)

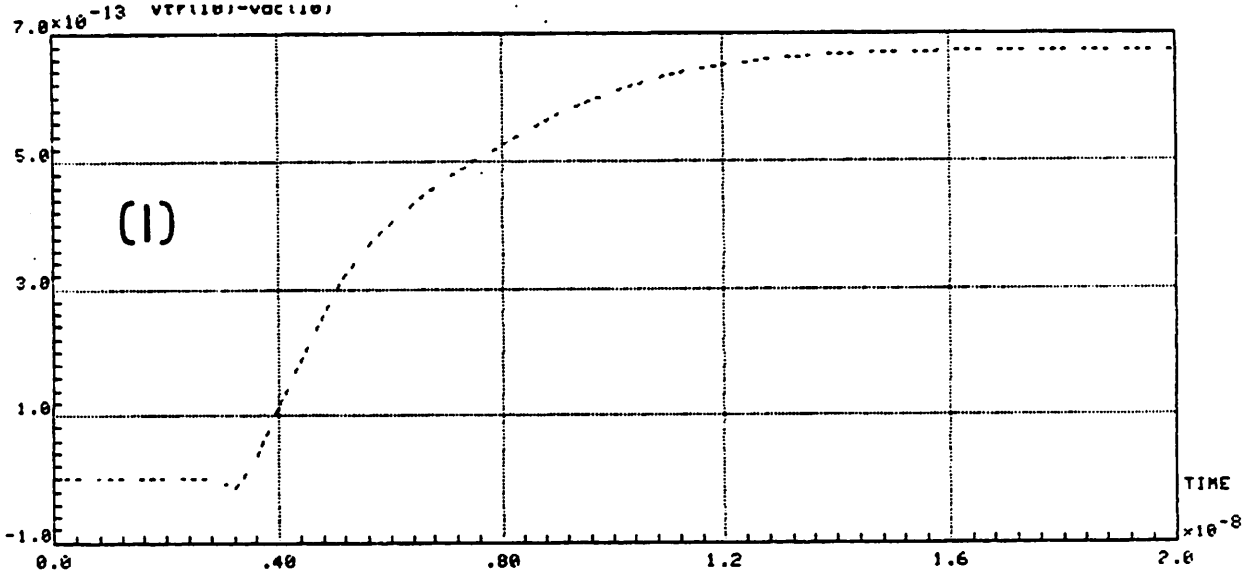
values, we obtain the initial comb shown in figure 5.7-m. We then proceed with a sequence of optimization steps, including several adjustments of good and bad values. The optimization is ended when a step is encountered that produces a very small change in the design parameters:

```
Iter=27 Phase2 MxM+SC= 1.079 k=3
Parameter    Value     %wrt 0 Prev
 1 M2N_W    9.226e-6    -42%   0%
 2 M3N_W    2.612e-5    -18%   1%
 3 Tdelay   1.032e-8    106%   0%
```

The corresponding performance comb in figure 5.7-n shows that the delay and area objectives are in precise competition (their comb teeth have the same length). Comparing these results to those of minimizing the delay alone shows that the delay has increased from 6.87ns to 10.3ns while the area has been reduced from 5920μ^2 to 1570μ^2. These tradeoffs are also to be expected.

The following table summarizes the results of the three optimization runs:





(m)

Pcomb		PRESENT	GOOD	G	R	BAD
M1	Delay	7.92e-9	6.00e-9			1.00e-8
M2	Integral C	6.75e-13	1.00e-11			1.50e-11
M3	Area	3.74e-9	1.00e-9			3.00e-9

(n)

Pcomb		PRESENT	GOOD	G	R	BAD
M1	Delay	1.03e-8	6.00e-9			1.00e-8
M2	Integral C	2.75e-13	1.00e-11			1.50e-11
M3	Area	1.57e-9	5.00e-10			1.50e-9

Figure 5.7. (continued)

CMOS Inverter Chain Optimization Results				
	Initial Performance	Run 1: Delay alone emphasized	Run 2: Delay and Integral Current emphasized	Run 3: Delay and Area emphasized
Delay	10.2ns	6.87ns	7.92ns	10.3ns
Integral Current	.391pC	1.19pC	.675pC	.275pC
Area	$2020\mu^2$	$5920\mu^2$	$3740\mu^2$	$1570\mu^2$
M2N_W	16μ	39.8μ	11.9μ	9.23μ
M3N_W	32μ	101.4μ	70.04μ	26.1μ

In conclusion, although this example contains only a few design parameters and concerns a relatively small circuit, it does demonstrate the design methodology introduced in chapter 4 in action. By adjusting the good and bad values as shown in the above optimization runs, any point on the *tradeoff curve* for these three competing objectives could be obtained. Another comment worth mentioning is that it appears to be quite reasonable to not show hard constraints on the Pcomb display. Instead, as shown in figure 5.7-b, an appropriate message is simply printed.

Other Circuit Design Examples

Other recent examples of circuit optimization are reported by Nye, et al. [116] (included as appendix G of this dissertation), for both product development and redesign of several industrial analog and digital circuits that resulted in substantial improvement in their performance. The results of these

optimizations are presently being incorporated into several of the products they discuss. Since the values of the design parameters with which the optimizations started were from final designs considered almost optimal by experienced designers, the improvements in performance reported are quite significant.

The first circuit they optimized was a high-speed operational amplifier, implemented with a complementary bipolar, dielectrically-isolated gigahertz process. This amplifier was to offer maximum bandwidth, the single objective, subject to constraints on stability, minimum settling time, output voltage swing, slew rate, DC power dissipation, and DC offset voltage. The DC limits, while important, were not of primary concern and could be relaxed to achieve better AC performance since the relative marketability of these specifications was very subjective. For example, if significant bandwidth could be obtained at the cost of a small increase in power, the amplifier would be more desirable.

The authors of [116] reported an efficient technique for handling the stability constraint. Using a functional constraint, they limited the peaking in the amplifier closed-loop frequency response over all frequencies in a certain interval, chosen so that it would surely contain the peak. This technique is in contrast to the much more computationally expensive approach of measuring stability using the settling time or overshoot of a transient simulation output waveform. They mention that in general, care must be used in choosing how to compute the quantities involved in the problem specifications.

The optimization results for the operational amplifier reported in [116] are as follows. After five optimization iterations using the enhanced phase I-II-III algorithm, the bandwidth was substantially improved at the cost of stability (greater peak in the closed-loop frequency response). After the bad value for the stabil-

ity constraint was changed, a few more iterations resulted in a high bandwidth amplifier with good stability but at the cost of a little more DC power. Next, the emphasis was shifted to obtaining an amplifier with low power consumption. The good and bad values for the DC power constraint were lowered and after a few more optimization iterations, a low power version of the amplifier was obtained. This example demonstrates the important role discussed in section 5.1.1 that performance tradeoffs play in an industrial environment: the various amplifier versions obtained by trading off particular objectives and constraints (through adjustment of their good and bad values) can be realized with only minor changes in a few IC masks and can be offered by the marketing department as several different product options.

Other applications of DELIGHT.SPICE mentioned in [116] include the design of a digital to analog converter, an A/D comparator, a digital bus precharge circuit, and a switched capacitor 10'th-order modem filter where tight phase linearity specifications were met without using costly equalization circuitry. The performance of each of these circuits was significantly improved, as shown in table 2 of [116].

5.2. Other Engineering Applications

In this section we present several other engineering design applications. These include the design of digital filters in section 5.2.1, feedback control systems in section 5.2.2, and earthquake-resistant buildings in section 5.2.3.

5.2.1. Digital Filter Design

The application of DELIGHT to the design of digital filters was a recent undertaken at Berkeley by a small team of researchers. The results have been pub-

lished in [91, 90] ([90] is included as appendix E of this dissertation). The aspect of the design of digital filters considered was the problem of determining a set of coefficients for a rational transfer function so as to best approximate some desired response (typically magnitude and phase frequency responses) while meeting other specifications such as special stability requirements.

The optimization approach of DELIGHT offered much greater flexibility than the rigid design specifications allowed by classical techniques, which construct digital filter transfer functions by conversion from classical analog filter transfer functions such as Butterworth, Chebyshev, elliptic, or Bessel. Such classical techniques have been automated in the filter design program FILSYN [145]. The DELIGHT approach is to transcribe the design problem directly into the classical mathematical programming formulation of section 4.4.2.1. This allows the specification of practical constraints, for example, on the polynomial degree, on the magnitude of the coefficients, or even for special stability requirements. After formulating the design problem in this way, it was solved using the Phase I - Phase II feasible directions algorithm with functional constraints, described earlier in section 4.5.1.3. This work benefited greatly from the already mentioned feature of this algorithm that the discretization of the functional constraint frequency range is automatic and refined dynamically as an optimal solution is approached.

An example of the special stability constraints that were included in the problem formulation is to keep the roots of the denominator quadratics (see below) inside a circle of radius ρ less than one. After having selected the degree of the filter, both the numerator and denominator of the transfer function were expressed as products of quadratics of the form

$$z^2 + bz + c \ .$$

This formulation, besides leading to a filter structure with low quantization noise [144], simplified the expression of the required conditions for stability, which were written as the following constraints on the coefficients of the denominator quadratics:

$$1 + \frac{b}{\rho} + \frac{c}{\rho^2} \geq 0, \qquad 1 - \frac{b}{\rho} + \frac{c}{\rho^2} \geq 0, \qquad 1 - \frac{c}{\rho^2} \geq 0$$

These constraints were trivially implemented as ordinary inequality constraints in procedure *ineq*.

The results reported by Lee et al. in [91] for the design of a lowpass digital filter show that the problem formulation and optimization algorithm used were quite successful in achieving good filter designs. Moreover, designs from FILSYN [145], which *does not* allow phase constraints for the lowpass filter considered, were taken and used as initial guesses to the optimization *with* phase constraints. Significant improvements in phase responses were obtained along with stronger stability properties.

Throughout the work, the interactive nature of DELIGHT optimization was indispensable. Several times observation of the optimization progress led to slight reformulation of the problem and to modification of some critical algorithm parameters. For example, an algorithm parameter was modified when it was noticed that the **X** iterates were bouncing on and off certain constraint boundaries. These observations were facilitated by several graphical displays that were output after each optimization iteration. For example, a RATTLE procedure easily constructed from the basic graphics commands of DELIGHT was used to draw plots of the locations of the complex filter poles in the s-plane.

5.2.2. SISO and MIMO Control Systems

An application of DELIGHT in the area of control system design is to multiple-input/multiple-output (MIMO) feedback control systems. The DELIGHT-MIMO package is the result of interfacing the basic DELIGHT system to various multivariable simulation and design aids. It is currently under development at the University of California, Berkeley, Imperial College, London, Kingston Polytechnic, London, and Lawrence Livermore National Laboratory, Livermore, California.

DELIGHT-MIMO is intended both as a practical design tool and as a test bed for concepts to be used in interactive control system design. The first version of the system was constructed by incorporating a number of subroutines from the Imperial College Multivariable Design System (MDS) [138] into DELIGHT using the application package development features of appendix B. These subroutines allow designers to define system components and interconnections, to evaluate time and frequency responses, and to use classical design techniques to produce an initial design for the optimization. Optimization is carried out by a RATTLE implementation of the Polak-Wardi algorithm [125] which permits constraints on both time responses and on singular values over a frequency range. Presently, DELIGHT-MIMO is being substantially revised by (1) the introduction a graphical, menu-driven, system component interconnection capability, (2) the replacement of the MDS routines by numerically more robust ones from the SLICE library [34], and (3) the addition of a number of utility programs. These utilities implement various modern design techniques which designers will be allowed to use either independently or as a means for obtaining an initial design for DELIGHT optimization.

Typical MIMO control system design problems were briefly mentioned in chapter 1. We now review several common requirements of these problems. Typical ordinary inequality constraints are those for stability that restrict the eigenvalues of the closed loop system to a specified region in the s-plane. Thus, if $\lambda_j(\mathbf{x})$, $j=1,2,\cdots,N_\lambda$ are the complex eigenvalues of the closed loop system, it may be required that they all lie in a parabolic region in the complex λ-plane defined by $\mathrm{Re}[\lambda]+a\cdot\mathrm{Im}[\lambda]^2\le b$ for a given $a>0$ and $b\ge 0$. This results in the N_λ inequality constraints

$$ineq(j,\mathbf{x}) = \mathrm{Re}[\lambda_j(\mathbf{x})] + a\cdot\mathrm{Im}[\lambda_j(\mathbf{x})]^2 - b \quad (\le 0) \quad \text{for } j=1,\cdots,N_\lambda$$

in which, given $\mathbf{x}$, the set of $\lambda_j(\mathbf{x})$'s is calculated using built-in DELIGHT-MIMO utility routines.

Typical functional inequality constraints arise from converting rise time, overshoot, and settling time constraints on the time-domain step responses of the closed loop system $y_j(\mathbf{x},t)$, $j=1,2,\cdots,N_y$ to those confining these step responses to lie within the specified envelopes:

$$Lower_j(t) \le y_j(\mathbf{x},t) \le Upper_j(t) \quad \forall t\in[0,Tstop], \quad j=1,2,\cdots,N_y\ .$$

These then result in the $2N_y$ functional constraints

$$fineq(j,\mathbf{x},t) = y_j(\mathbf{x},t) - Upper_j(t) \quad (\le 0) \quad j=1,2,\cdots,N_y$$

and

$$fineq(j,\mathbf{x},t) = Lower_j(t) - y_j(\mathbf{x},t) \quad (\le 0) \quad j=N_y+1,\ N_y+2,\cdots,\ 2N_y$$

in which, given $\mathbf{x}$, the set of $y_j(\mathbf{x},t)$'s is again calculated using built-in routines. As mentioned in section 4.4.1.2, the functions $Lower_j(t)$ and $Upper_j(t)$ must be piecewise Lipschitz continuous in their t argument, with discontinuities only of the first kind.

Another important example of functional constraints is for establishing bounds on the smallest singular value $\sigma(\mathbf{x},\omega)$ of the return difference matrix $F(\mathbf{x},\omega)$, the purpose of which is to reduce the sensitivity to parameter variations. These have the form

$$\sigma(\mathbf{x},\omega) \geq \mathit{Bound}(\omega) \quad \forall\omega \in [\omega',\omega'']$$

which results in the functional constraint

$$\mathit{fineq}(j,\mathbf{x},\omega) = \mathit{Bound}(\omega) - \sigma(\mathbf{x},\omega) \quad (\leq 0) .$$

Again the function $\mathit{Bound}(\omega)$ must be piecewise Lipschitz continuous in ω. Other similar constraints place bounds on the *largest* singular value of another transfer function matrix to insure that the system is insensitive to output disturbances. These kinds of constraints can be handled by, for example, the Polak-Wardi algorithm for nondifferentiable optimization problems.

One possible role for the single cost function—single because this work was performed prior to the existence of the multiple objective problem formulation—is to express the desire to minimize the energy being input into the plant during a specified operation. The cost would then take the form of the following integral:

$$\mathit{cost}(\mathbf{x}) = \int_0^{\mathit{Tstop}} u(\mathbf{x},t)^2 \, dt$$

where $u(\mathbf{x},t)$ is the input to the plant. Another possibility is to minimize the integral squared feedback error of the system giving

$$\mathit{cost}(\mathbf{x}) = \int_0^{\mathit{Tstop}} \varepsilon(\mathbf{x},t)^2 \, dt$$

The integrations above could be performed using either standard numerical analysis techniques or during the time-domain solution of differential equations.

Finally, there are often design parameter box constraints such as positivity and bounds on the compensator design parameters.

Another related use of DELIGHT for optimization of control systems that did not, however, involve DELIGHT-MIMO is the work of Meena Karandikar [75] at Berkeley on the design of SISO control systems. Using the RATTLE language and the Phase I - Phase II optimization algorithm from the RATTLE Algorithms Library, she implemented the algebraic design methodology of Chen et al. [29] for SISO control systems. The closed loop disturbance transmission was minimized. In order to get low disturbance transmission, the closed loop feedback compensator gain should increase. Since this increases the magnitude of the signal at the input to the plant, a functional constraint on the compensator gain was introduced to avoid the practical problem of plant saturation. Similar to the MIMO case above, the complex zeros of the closed loop characteristic polynomial were constrained to lie in a certain region of the s-plane. In her conclusions, Meena notes that these design objectives could not have been handled easily by classical, non-optimization techniques. She also identifies a tradeoff situation between the bandwidth over which the disturbance transmission is reduced and the step response.

5.2.3. Earthquake-Resistant Structures

An important design problem in civil engineering is found in the area of structural design of multistory, steel-framed buildings. Current design philosophy requires that buildings must withstand small earthquakes with no damage and large ones with repairable damage, avoiding collapse. In a typical braced frame design, the design parameter vector $\mathbf{x}$ can include the section moment of inertia or cross sectional area of the frame or bracing members. The different

horizontal floors are assumed to be rigid and to concentrate the mass of the structure. Horizontal displacements of the floors and roof form the components of a displacement vector. This lumped parameter model then obeys a set of differential equations in the displacement vector which are driven by a forcing function that represents the ground motion. It is common to consider a whole family of excitations, both large and small, in carrying out a design. A common objective is to reduce initial cost by reducing the weight of the structure. This is done by minimizing the volume of the frame members subject to several groups of constraints. The designer may formulate one group of constraints corresponding to a static model subjected to gravity loads and another corresponding to the dynamic model given by the set of differential equations. The use of these constraints is to limit displacements and internal forces in the structure over the entire duration of a family of earthquake excitations.

The DELIGHT.STRUCT [18] version of DELIGHT interfaces to the ANSR [101] nonlinear structural response simulator. The associated library of software for different classes of structural design problems currently contains software for the seismic-resistant design of steel frames discussed above. However, Balling indicates in [18] that the system could easily be extended to handle optimization problems involving the design of steel bridge decks, concrete dams, storage tanks, offshore platforms, energy-absorbing restrainers for nuclear power plants, spatial piping systems, etc.

In his doctoral work, Balling handled the following specific steel frame design problem. Earthquake records used as a forcing function to drive the simulation were actual ground motion accelerograms selected and scaled to levels representing moderate and severe ground motions. The design variables

included column moments of inertia, brace cross-sectional areas, and a dummy story drift variable to handle functional costs. The cost to minimize was a linear combination of the following six terms:

1. the volume of the design elements,
2. the sum of the squares of the maximum story drifts during a moderate earthquake,
3. the severe earthquake input energy,
4. the severe earthquake inelastically dissipated energy,
5. this energy dissipated by shear link and dissipator elements, and
6. this energy dissipated by the columns.

The shear link and dissipator elements in number 4 act as "fuses" which can dissipate large amounts of energy without causing significant damage to the structure. Since this work was carried out before the multiobjective problem formulation had been conceived, the weighted-sum approach was necessary. Ordinary inequalities included constraints on the axial gravity force, gravity end moments, and severe earthquake energy dissipation for column elements of the frame. The latter two were also constraints for brace elements. Functional inequality constraints were categorized into those for moderate and those for severe earthquake excitations. For the former the constraints were over time and on the column end moments, the brace axial forces, the story drifts, and the floor accelerations. The latter class included a constraint on the structure sway, i.e., the horizontal displacement at the top of the frame divided by the total frame height, based on the observation that the collapse of a frame may be detected by large displacements at the top of the frame. In total there were 141 ordinary inequality constraints and 69 functional constraints in the four story example presented.

The optimization runs reported by Balling indicated that six iterations were required to satisfy all 200 of the constraints using the Phase I - Phase II algorithm. He also noted that the volume of the structure remained nearly constant for all iterations and thus the feasible design was not constructed by simply increasing the strengths of all the frame members, but rather by re-distributing the strength of the structure among the members. Without optimization techniques, he adds, often an engineer faced with an infeasible design is tempted to resolve the problem by simply increasing the sizes of the relevant frame members. After obtaining the initial feasible design, the weights in the cost functions were adjusted to obtain final designs in which several of the terms in the cost function were emphasized individually.

CHAPTER 6

Conclusions and Future Research

We have considered the benefits of applying optimization to engineering design, identified shortcomings of other attempts to achieve similar application, and designed and implemented the DELIGHT system both to meet a resulting set of design criteria and as part of a broad project dedicated to optimization-based computer-aided design. DELIGHT, for "DEsign Laboratory with Interaction and Graphics for a Happier Tomorrow", contains many features that help meet these criteria. Features were explored that, in particular, both provide engineering designers with a powerful new design tool with which to optimize the performance of their designs as well as facilitate the development of optimization algorithms. For the "non-optimization" designer DELIGHT provides simple command and algorithm execution. For the advanced designer it provides features which make it easy to rescale or modify either the design problem being solved or the optimization algorithm being used, without losing any previous optimization progress. A new engineering-oriented multiple objective problem formulation was introduced that allows arbitrary problem formulation through a general expression capability, and a means of classifying and conveying the relative importance of design specifications. Powerful, highly flexible color graphics can be used to display both properties and performance of the system being designed as well as information which facilitates optimization algorithm tuning. In particular, a methodology and commands for performing design tradeoffs were introduced that use a new graphical display called the *Pcomb* performance comb.

The optimization algorithm expert is given a high-level language which permits him to easily access common mathematical numerical analysis software in writing compact procedures that bear a close resemblance to the mathematical description of the algorithms being implemented. This extensible nature of the language and the associated test and debug aids cause most of the usual coding errors to be eliminated and the programming time to be shortened greatly. A library of both classical and recent optimization subprocedures gives designers the ability to access complete and proven algorithms or create a large number of different algorithms from the relatively few basic building blocks in the library. Finally, since the DELIGHT system is intended to be used in many different design areas, there is a simulation interface methodology and other necessary features that facilitate the coupling of DELIGHT to existing simulation programs. We then took a look at several application areas and demonstrated the usefulness of the system by showing the optimization of several electronic integrated circuits and reporting successful results in the other areas examined.

There are a number of enhancements or areas of future research that can help the DELIGHT system better meet its goals. One is simply to add to the library of optimization algorithms. In particular, more emphasis should be placed on the superlinearly convergent Lagrangian-based algorithms for constrained optimization since recently several of their shortcomings have been eliminated (see, for example, Tits [148]). A related enhancement is that algorithm choices and sub-block substitution need to be multi-level and hierarchical instead of just single-level as at present. In other words, there needs to be *sub*-sub-block choices under various sub-blocks. Another important area deserving additional attention concerns how to effectively convey information that assists

a user in adjusting algorithm parameters. The gradient clock and the Armijo graphics display presented in this dissertation consider just a small subset of the existing parameters. Future work in this area should also emphasize graphical displays that convey information easily.

A useful enhancement to DELIGHT graphics is an automatic viewport manager. This would automatically set up and place viewports on the terminal screen so users could easily add or remove graphical displays without having to shrink or expand viewports already on the screen. Similar results have recently appeared in regard to data base screen generators (see, for example, the work of Shoens [133]). A related area with great potential concerns a menu-driven user interface for DELIGHT. One approach is to take advantage of the uniform user/program environment available through the Berkeley *hawk* graphics editor [76] and Squid data base system [77]. Hawk provides a friendly user "front end" for several VLSI design tools that facilitates their usage by beginners as well as by experienced designers. A menu-driven input to DELIGHT could reduce the apparent complexity of the many possible actions from which a designer can choose, at different states in an optimization process.

There are several specific enhancements that can improve the effectiveness of DELIGHT.SPICE as a CAD tool. One is to allow additional output types (keywords) such as ones for noise or distortion output at circuit nodes. Another is for output currents through any element. This might require much effort since presently SPICE computes directly the currents through only certain elements, eg., voltage sources. However, SPICE3 [129] may alleviate this problem in the near future. A very important addition to DELIGHT.SPICE that will save a large amount of cpu time and make circuit optimization much more practical is to have SPICE compute directly the sensitivities necessary for optimization

gradients, instead of the present finite-difference approach. For output vector **v**, SPICE would compute $\partial \mathbf{v} / \partial \mathbf{x}$ in the chain rule formula

$$\frac{df}{d\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}} + \frac{\partial f}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial \mathbf{x}}$$

where f is *multicost*, *ineq*, etc. The other partials on the right-hand side above would either be user supplied (ok), computed using finite differences (better), or computed using symbolic differentiation of the RATTLE problem description expressions (best). Preliminary work on the latter has been started [109]. Also, work is already underway to allow SPICE to compute both DC, AC, and transient sensitivities [111]. For the chain rule expression above, design parameter tracking and matching need to be carefully considered.

Other CAD tools that could be implemented in RATTLE and thus take advantage of the existing SPICE simulation interface include design centering, tolerancing, tuning, and yield improvement. In fact, multiobjective optimization involving yield as a tradeoff has been recently studied by Lightner and Director [93].

The conceptually simple simulation interface methodology provided by DELIGHT, makes it easy to interface to simulation programs in various engineering areas. These might be other circuit simulators at Berkeley such as SPICE3 [129], SPLICE2 [107, 135], or RELAX [92], or simulation programs in other engineering areas, as suggested by the recent interest in using DELIGHT optimization in optics.

In conclusion, at the simplest conceptual level, the "kernel" program of the DELIGHT system does little more than provide a simple interactive programming language along with powerful extensibility and the ability to add existing (For-

tran) routines. The greatest contributions of the DELIGHT system are not so much in creating these three features but in what has evolved using them. To quote an author of the Interlisp programming environment, "Interlisp was not designed, it evolved—but this was the right approach" [147]. Similarly, many important features of DELIGHT were not planned at the outset but instead originated during an evolutionary design process.

Our feelings about the need for such a design system can best be stated by paraphrasing the introduction of Strunk and White [74]:

> We felt that the designer trying to use optimization as a tool was in serious trouble most of the time, a man floundering in a swamp, and that it was the duty of anyone attempting to help him in this task, to drain this swamp quickly and get his man up on dry ground, or at least throw him a rope. In creating DELIGHT, we have tried to hold steadily in mind this belief and concern for the troubled state of optimization use in engineering design.

It is hoped that DELIGHT and this dissertation will provide a workable system and methodology for using optimization in engineering design. In this way, perhaps the barriers may be breeched that have thus far prevented the widespread use of such a powerful computer-aided design technique.

References

[1] "AEDCAP User's Guide," Report A-CIR-000-4, SofTech, Inc., Waltham, Mass. (1973).

[2] "ASTAP Advanced Statistical Analysis Program," IBM Program Product Document SH20-1118-0, IBM Data Processing Division, White Plains, N.Y. (1973).

[3] *OPNODE User's Manual*, Automatic Measurement Division, Hewlett-Packard Company, Sunnyvale, California (1975).

[4] "VP/CSS Executive Language (EXEC)," NCSS Technical Note Form 109-4, National CSS, Inc., Norwalk, Connecticut (March 1975).

[5] "ISPICE Part 1: Circuit Description and Modeling," *IEEE Circuit and Systems Magazine* vol. 11, no. 6, pp. 3-7 (December 1977).

[6] *COMPACT Version 4.8, United Computing Systems*, Compact Engineering, Inc., Los Altos, California (October 1978).

[7] "ISPICE Part 2: Simulation and Analysis of Results," *IEEE Circuit and Systems Magazine* vol. 12, no. 1, pp. 2-8 (February 1978).

[8] *Reference Manual for the Programming Language Ada—Proposed Standard Document*, US Department of Defense (July 1980).

[9] *SLICE User's Guide*, Harris Semiconductor, Melbourne, Florida (1982).

[10] "Harwell Subroutine Library," Library Reference Manual, Harwell, England (1982).

[11] M. R. Aaron, "The Use of Least Squares in System Design," *IRE Transactions on Circuit Theory* vol. CT-3, pp. 224-231 (December 1956).

[12] D. Agnew, "Improved Minimax Optimization for Circuit Design," *IEEE Transactions on Circuits and Systems* vol. CAS-28, no. 8, pp. 791-803 (August 1981).

[13] A. V. Aho and J. D. Ullman, *Theory of Parsing, Translation, and Compiling* Volume 1, Prentice-Hall, Englewood Cliffs, New Jersey (1972).

[14] A. V. Aho and J. D. Ullman, *Theory of Parsing, Translation, and Compiling* Volume 2, Prentice-Hall, Englewood Cliffs, New Jersey (1973).

[15] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

[16] L. Armijo, "Minimization of Functions Having Continuous Partial Derivatives," *Pacific Journal of Mathematics* vol. 16, pp. 1-3 (1966).

[17] M. Bales, "Csubst," Cadman Manual Page, Electronics Research Laboratory, University of California, Berkeley, California (November 1980).

[18] R. J. Balling, K. S. Pister, and E. Polak, "DELIGHT.STRUCT: A Computer-Aided Design Environment for Structural Engineering," Memo No. UCB/EERC-81/19, Earthquake Engineering Research Center, University of California, Berkeley, California (December 1981).

[19] D. W. Barron, *An Introduction to the Study of Programming Languages*, Cambridge University Press, Cambridge, London, U. K. (1977).

[20] M. A. Bhatti, K. S. Pister, and E. Polak, "OPTDYN - A General Purpose Optimization Program for Problems With or Without Dynamic Constraints," Report No. UCB/EERC-79/16, Earthquake Engineering Research Center, University of California, Berkeley, California (July 1979).

[21] M. A. Bhatti, T. Essebo, W. T. Nye, K. S. Pister, E. Polak, A. Sangiovanni-Vincentelli, and A. L. Tits, "A Software System for Optimization-Based Interactive Computer-Aided Design," Memo No. UCB/ERL M80/14, Electronics Research Laboratory, University of California, Berkeley, California (April 1980).

[22] G. Billingsly, K. Keller, and M. Bales, *MFB Reference Manual*, University of California, Berkeley, California (July 1982).

[23] R. K. Brayton and R. Spence, *Sensitivity and Optimization*, Elsevier Scientific Publishing Company, Amsterdam, Netherlands (1980).

[24] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "A Survey of Optimization Techniques for Integrated Circuit Design," *Proc. IEEE* vol. 69, no. 10, pp. 1334-1362 (1981).

[25] P. J. Brown, *Macroprocessors and Techniques for Portable Software*, John Wiley & Sons, Inc., New York, N.Y. (1974).

[26] D. A. Calahan, "Computer Design of Linear Frequency Selective Networks," *Proc. IEEE*, no. 53, pp. 1701-1706 (1965).

[27] D. A. Calahan, *Computer-Aided Network Design*, McGraw-Hill Book Company, New York (1972).

[28] W. C. Cave, "An Automated Design Formulation for Integrated Circuits," pp. 241-279 in *Computer-Aided Integrated Circuit Design*, ed. G. J. Herskowitz, McGraw-Hill Book Company, New York (1968).

[29] M. J. Chen, C. A. Desoer, and G. F. Franklin, "Algorithmic Design for Single-Input Single-Output Systems With a Two-Input One-Output Controller," Memo No. UCB/ERL M81/12, Electronics Research Laboratory, University of California, Berkeley, California (1981).

[30] E. Cohen, "Program Reference for SPICE2," Memo No. ERL-M592, Electronics Research Laboratory, University of California, Berkeley, California (June 1976).

[31] IBM Corporation, "IBM System/360 Operating System: PL/I Reference Manual," Form C28-8202, IBM Data Processing Division, White Plains, New York (1967).

[32] J. Cullum, "An Algorithm for Minimizing a Differentiable Function That Uses Only Function Values," pp. 117-127 in *Techniques of Optimization*, ed. A. V. Balakrishnan, Academic Press, New York (1972).

[33] K. C. Daly and P. Katzberg, "COSDIC Documentation: The Classical Design Suite," Publication No. 73/18, Department of Computing and Control, Imperial College, London SW7 2BT, U.K. (June 1973).

[34] M. J. Denham and C. J. Benson, "SLICE: A Subroutine Library for Control System Design," Internal Report 01/82, School of Electronic Engineering and Computer Science, Kingston Polytechnic, Kingston upon Thames KT1 2EE, U.K. (1982).

[35] C. A. Desoer and S. K. Mitra, "Design of Lossy Ladder Filters by Digital

Computer," *IRE Transactions on Circuit Theory* vol. CT-8, pp. 192-201 (September 1961).

[36] E. W. Dijkstra, *A Primer of Algol 60 Programming*, Academic Press, New York, N.Y. (1962).

[37] E. W. Dijkstra, "Notes on Structured Programming," pp. 1-81 in *Structured Programming*, ed. C. A. R. Hoare, Academic Press, New York, N.Y. (1972).

[38] G. Dilley, private communication, Earthquake Engineering Research Center, University of California, Berkeley, California (September 1982).

[39] S. W. Director and R. A. Rohrer, "Automated Network Design—The Frequency-Domain Case," *IEEE Transactions Circuit Theory* vol. CT-16, pp. 330-337 (1969).

[40] S. W. Director and R. A. Rohrer, "The Generalized Adjoint Network and Network Sensitivities," *IEEE Transactions Circuit Theory* vol. CT-16, pp. 318-323 (1969).

[41] S. W. Director, *A Survey of Decomposition Techniques for Analysis and Design of Electrical Networks*, presented at the University of Florida (1974).

[42] S. W. Director and G. D. Hachtel, "The Simplicial Approximation Approach to Design Centering," *IEEE Transactions on Circuits and Systems* vol. CAS-24 , (July 1977).

[43] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, Pa. (1979).

[44] R. I. Dowell, *Automated Biasing of Integrated Circuits*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (April 1972).

[45] C. Eastman and R. Thornton, *A Report on the GLIDE 2 Language Definition*, Computer Aided Design Group, Institute for Physical Planning, Carnegie-Mellon University, Pittsburgh, Pennsylvania (preliminary draft, March 1979).

[46] S. I. Feldman, "The Programming Language EFL," Comp. Sci. Tech. Rep. No. 78, Bell Laboratories, Murray Hill, New Jersey (June 1979).

[47] A. V. Fiacco and G. P. McCormick, "The Sequential Unconstrained Minimization Technique for Nonlinear Programming. Algorithm II, Optimum Gradients by Fibonacci Search," Technical Paper RAC-TP-123, Research Analysis Corporation, McLean, Virginia (June 1964).

[48] P. E. Fleischer, "Optimization Techniques," pp. 175-217 in *System Analysis by Digital Computer*, ed. J. F. Kaiser, John Wiley & Sons, Inc., New York (1966).

[49] R. Fletcher and M. J. D. Powell, "A Rapidly Convergent Descent Method for Minimization," *The Computer Journal* vol. 6, pp. 163-168 (1963).

[50] R. Fletcher and C. M. Reeves, "Function Minimization by Conjugate Gradients," *Computer Journal* vol. 6, pp. 149-154 (1964).

[51] P. E. Gill, W. Murray, S. M. Picken, and M. H. Wright, "The Design and Structure of a Fortran Program Library for Optimization," *ACM Transactions on Mathematical Software* vol. 5, no. 3, pp. 259-283 (September 1979).

[52] L. Gilman and A. J. Rose, *APL - An Interactive Approach*, John Wiley & Sons, Inc., New York, N.Y. (1970).

[53] J. J. Golembeski, "Computer-Optimized Model Determination," pp. 76-112 in *Computer-Aided Integrated Circuit Design*, ed. G. J. Herskowitz, McGraw-Hill Book Company, New York (1968).

[54] C. Gonzaga, E. Polak, and R. Trahan, "An Improved Algorithm for Optimization Problems with Functional Inequality Constraints," *IEEE Transactions on Automatic Control* vol. AC-25, no. 1, pp. 49-54 (1980).

[55] P. R. Gray and R. G. Meyer, *Analysis and Design of Analog Integrated Circuits*, John Wiley & Sons, Inc., New York, N.Y. (1977).

[56] D. Gries, *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, N.Y. (1971).

[57] G. D. Hachtel and R. A. Rohrer, "Techniques for the Optimal Design and Synthesis of Switching Circuits," *Proc. IEEE*, no. 55, pp. 1864-1877 (1967).

[58] G. D. Hachtel, M. R. Lightner, and H. J. Kelly, "Application of the Optimization Program AOP to the Design of Memory Circuits," *IEEE Transactions on Circuits and Systems* vol. CAS-22, no. 6, pp. 496-503 (June 1975).

[59] G. D. Hachtel, T. R. Scott, and R. P. Zug, "An Interactive Linear Programming Approach to Model Parameter Fitting and Worst Case Circuit Design," *IEEE Transactions on Circuits and Systems* vol. CAS-27, no. 10, pp. 871-881 (October 1980).

[60] G. D. Hachtel and P. Zug, *APLSTAP - Circuit Design and Optimization System - User's Guide*, IBM Yorktown Research Facility, Yorktown, N.Y. (1981).

[61] D. E. Hall, D. K. Scherrer, and J. S. Sventek, "A Virtual Operating System," *Communications of the ACM* vol. 23, no. 9, pp. 495-502 (September 1980).

[62] M. R. Hestenes, "Multiplier and Gradient Methods," *Journal of Optimization Theory and Applications* vol. 4, pp. 303-320 (November 1969).

[63] R. Hettich, "Semi-Infinite Programming," Lecture Notes in Control and Information Sciences, Springer Verlag (1979).

[64] D. M. Himmelblau, *Applied Nonlinear Programming*, McGraw-Hill, New York, N.Y. (1972).

[65] D. A. Hodges and H. Jackson, *Analysis and Design of Digital Integrated Circuits*, John Wiley & Sons, Inc., New York, N.Y. (1982).

[66] F. R. A. Hopgood, *Compiling Techniques*, Elsevier Scientific Publishing Company, Amsterdam, Netherlands (1969).

[67] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Woodland Hills, California (1976).

[68] T. E. Idelman, F. S. Jenkins, W. J. McCalla, and D. O. Pederson, "SLIC - A Simulator for Linear Integrated Circuits," *IEEE Journal of Solid State Circuits* vol. SC-6, pp. 188-203 (August 1971).

[69] K. Jensen and N. Wirth, "Pascal User Manual and Report," Lecture Notes in Computer Science (18), Springer Verlag (1977).

[70] S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

[71] W. Joy, The vi Editor, seminar presented at the Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (1979).

[72] W. Joy, "Csh(1)," in *UNIX Programmer's Manual*, Seventh Edition, Virtual

VAX-11 Version, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (June 1981).

[73] W. Joy, "Vi(1)," in *UNIX Programmer's Manual*, Seventh Edition, Virtual VAX-11 Version, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (June 1981).

[74] W. Strunk Jr. and E. B. White, *The Elements of Style*, third edition, MacMillan Publishing Co., Inc., New York, N.Y. (1979).

[75] M. Karandikar, *Implementation of Chen et al.'s Design Methodology for SISO Control Systems* (titled by Bill Nye), Master's Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (December 1982).

[76] K. Keller, "A Symbolic Layout Design System," *Proceedings of the 1982 IEEE International Symposium on Circuits and Systems*, (1982).

[77] K. Keller, "A Symbolic Design System for Integrated Circuits," *Proceedings of the 1982 Design Automation Conference*, (1982).

[78] B. W. Kernighan, "RATFOR—A Preprocessor for a Rational Fortran," *Software—Practice and Experience*, (October 1975).

[79] B. W. Kernighan and P. Plauger, *Software Tools*,, Addison-Wesley, Reading, Mass. (1976).

[80] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

[81] J. Kleckner, private communication, Electronics Research Laboratory, University of California, Berkeley, California (November 1982).

[82] R. Klessig and E. Polak (edited by R. Trahan), *Notes for Optimization and Optimal Control*, Electronics Research Laboratory, University of California, Berkeley, California (Spring 1980).

[83] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass. (1968).

[84] H. W. Kuhn and A. W. Tucker, "Nonlinear Programming," *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pp. 481-493 (1951).

[85] P. Labuhn, *MINLP System Report*, Master's Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (December 1982).

[86] L. S. Lasdon and A. D. Waren, "Optimal Design of Filters with Bounded, Lossy Elements," *IEEE Transactions Circuit Theory* vol. CT-13, pp. 175-187 (1966).

[87] B. M. Leavenworth, "Syntax Macros and Extended Translations," *Communications of the ACM* vol. 9, no. 11, pp. 790-793 (November 1966).

[88] R. J. LeBlanc and J. J. Goda, "Ada and Software Development Support: A New Concept in Language Design," *IEEE Computer* vol. 15, no. 5, pp. 75-82 (May 1982).

[89] H. Ledgard, J. A. Whiteside, and A. Singer, "The Natural Language of Interactive Systems," *Communications of the ACM* vol. 23, no. 10, pp. 556-563 (October 1980).

[90] T. P. Lee, W. T. Nye, and A. L. Tits, "The Design of Digital Filters Using Interactive Optimization," *Proceedings of the The 20'th IEEE Conference on*

Decision and Control, (December 1981).

[91] T. P. Lee, W. T. Nye, and A. L. Tits, "The Design of Digital Filters Using Interactive Optimization," *IEEE Transactions on Circuits and Systems* vol. CAS-30, no. 11, (1983).

[92] E. Lelarasmee, *The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits: Theory and Applications*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (May 1982).

[93] M. R. Lightner and S. W. Director, "Multiple Criterion Optimization with Yield Maximization," *IEEE Transactions on Circuits and Systems* vol. CAS-28, no. 8, pp. 781-791 (August 1981).

[94] M. R. Lightner and S. W. Director, "Multiple Criterion Optimization for the Design of Electronic Circuits," *IEEE Transactions on Circuits and Systems* vol. CAS-28, no. 3, pp. 169-179 (March 1981).

[95] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley, Reading, Mass. (1973).

[96] D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *Journal of the Society of Industrial and Applied Mathematics* vol. 11, no. 2, pp. 431-441 (June 1963).

[97] D. G. Mayne, W. T. Nye, E. Polak, P. Siegel, and T. Wuu, "DELIGHT-MIMO: An Interactive, Optimization-Based Multivariable Control System Design Package," Memo No. UCB/ERL M82/53, Electronics Research Laboratory, University of California, Berkeley, California (July 1982).

[98] D.Q. Mayne, E. Polak, and A. Sangiovanni-Vincentelli, "Computer-Aided Design via Optimization: A Review," *Automatica*, vol. 18, no. 2, pp. 147-154 (1982).

[99] W. J. McCalla, *Computer-Aided Design of Integrated Bandpass Amplifiers*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (June 1972).

[100] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (1980).

[101] D. P. Mondkar and G. H. Powell, "ANSR-I General Purpose Program for Analysis of Nonlinear Structural Response," Report No. EERC 75-37, Earthquake Engineering Research Center, University of California, Berkeley, California (December 1975).

[102] M. A. Murray-Lasso, "Analysis of Linear Integrated Circuits by Digital Computer Using Black-Box Techniques," pp. 113-159 in *Computer-Aided Integrated Circuit Design*, ed. G. J. Herskowitz, McGraw-Hill Book Company, New York (1968).

[103] M. A. Murray-Lasso, "An Interactive Optimization Program for Use on a Time-Shared Computer," Internal Memorandum, Bell Telephone Laboratories (April 1968).

[104] W. Myers, "Computer Graphics: The Need for Graphics Design," *IEEE Computer* vol. 14, no. 6, pp. 86-92 (June 1981).

[105] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Memo No. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, California (May 1975).

[106] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer*

Graphics, second edition, McGraw-Hill, New York, N.Y. (1979).

[107] A. R. Newton, "The Simulation of Large Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems* vol. CAS-26, pp. 741-749 (September 1979).

[108] W. T. Nye, "Dynamic Arrays Via a Modified Ratfor Preprocessor," Memo No. UCB/ERL M80/23, Electronics Research Laboratory, University of California, Berkeley, California (June 1980).

[109] W. T. Nye, unpublished research, Electronics Research Laboratory, University of California, Berkeley, California (1981).

[110] W. T. Nye, *DELIGHT Reference Manual*, Electronics Research Laboratory, University of California, Berkeley, California (September 1981).

[111] W. T. Nye and D. C. Riley, "Transient Sensitivity in SPICE," EECS 290H course project report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (June 1982).

[112] W. T. Nye and A. L. Tits, "An Enhanced Methodology for Interactive Optimal Design—The Phase I-II-III Method of Feasible Directions," in preparation (1983).

[113] W. T. Nye, A. L. Tits, and A. Sangiovanni-Vincentelli, "Enhanced Methods of Feasible Directions for Engineering Design Problems," in preparation (1983).

[114] W. T. Nye, D. C. Riley, and A. Sangiovanni-Vincentelli, *DELIGHT.SPICE User's Guide*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (February 1983).

[115] W. T. Nye, *DELIGHT.ARBSIM User's Guide*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (1983).

[116] W. T. Nye, A. Sangiovanni-Vincentelli, J. P. Spoto, and A. L. Tits, "DELIGHT.SPICE: An Optimization-Based System for the Design of Integrated Circuits," *Proceedings of the 1983 Custom Integrated Circuits Conference*, (May, 1983).

[117] W. T. Nye and A. L. Tits, "An Enhanced Methodology for Interactive Optimal Design," *Proceedings of the 1983 IEEE International Symposium on Circuits and Systems*, (May 1983).

[118] R. G. Oliver, *GENGRAF: A General Purpose Subroutine Package for Computer Graphics*, Division of Structural Engineering and Structural Mechanics, University of California, Berkeley, California (1983).

[119] D. A. Pierre and M. J. Lowe, *Mathematical Programming Via Augmented Lagrangians*, Addison-Wesley, Reading, Mass. (1975).

[120] O. Pironneau and E. Polak, "Rate of Convergence of a Class of Methods of Feasible Directions," *SIAM Journal of Numerical Analysis* vol. 10, pp. 161-274 (1968).

[121] E. Polak, *Computational Methods in Optimization*, Academic Press, New York, N.Y. (1971).

[122] E. Polak, "Algorithms for a Class of Computer-Aided Design Problems: A Review," *Automatica*, vol. 15, pp. 795-813 (September 1979).

[123] E. Polak, R. Trahan, and D. Q. Mayne, "Combined Phase I - Phase II Methods of Feasible Directions," *Mathematical Programming* vol. 17, no. 1,

pp. 32-61 (1979).

[124] E. Polak, K. J. Astrom, and D. G. Mayne, "INTEROPTDYN-SISO: A Tutorial," Memo No. UCB/ERL M81/99, Electronics Research Laboratory, University of California, Berkeley, California (December 1981).

[125] E. Polak and Y. Wardi, "A Nondifferentiable Optimization Algorithm for the Design of Control Systems Subject to Singular Value Inequalities Over a Frequency Range," *Automatica*, vol. 18, no. 3, pp. 267-283 (1982).

[126] E. Polak, "Semi-Infinite Optimization in Engineering Design," unpublished memo, Electronics Research Laboratory, University of California, Berkeley, California (1982).

[127] M. J. D. Powell, "A Method for Nonlinear Constraints in Minimization Problems," in *Optimization*, ed. R. Fletcher, Academic Press, New York (1969).

[128] M. J. D. Powell, "Problems Related to Unconstrained Optimization," pp. 29-55 in *Numerical Methods for Unconstrained Optimization*, ed. W. Murray, Academic Press, New York, N.Y. (1972).

[129] T. Quarles, *SPICE3 Preliminary Report*, in progress at the Electronics Research Laboratory, University of California, Berkeley, California (1982).

[130] J. Raskin, "A Plea for Experiments in Language Design," *Communications of the ACM* vol. 24, no. 1, p. 41 (January 1981).

[131] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal* vol. 57, no. 6, pp. 1905-1929 (1978).

[132] R. A. Rohrer, "Fully Automated Network Design by Digital Computer: Preliminary Considerations," *Proc. IEEE*, no. 55, pp. 1929-1939 (1967).

[133] L. A. Rowe and K. A. Shoens, "Programming Language Constructs for Screen Definition," *IEEE Transactions on Software Engineering* vol. SE-9, no. 1, pp. 31-39 (January 1983).

[134] W. Rudin, *Principles of Mathematical Analysis*, third edition, McGraw-Hill, New York, N.Y. (1976).

[135] R. A. Saleh, A. R. Newton, and J. E. Kleckner, *SPLICE Version 1.4 User's Guide*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (1983).

[136] P. O. Scheibe and E. A. Huber, "The Application of Carroll's Optimization Technique to Network Synthesis," *Proceedings of the Third Annual Allerton Conference on Circuits and Systems*, pp. 182-191 (1965).

[137] A. C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, New Jersey (1974).

[138] B. R. Shearer and A. D. Field, "Multivariable Design System (MDS): An Interactive Package for the Design of Multivariable Control Systems," Publication No. 75/29, Department of Computing and Control, Imperial College, London SW7 2BT, U.K. (1975).

[139] P. Siegel, *A Graphical Front End for DELIGHT-MIMO*, Master's Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (August 1982).

[140] A. Singer, H. Ledgard, and Jon F. Hueras, "The Annotated Assistant: A Step Towards Human Engineering," *IEEE Transactions on Software Engineering* vol. SE-7, no. 4, pp. 353-374 (July 1981).

[141] N. Solntseff and A. A. Yezerski, "A Survey of Extensible Programming

Languages," *Annual Review in Automatic Programming* vol. 7, no. 5, pp. 267-307 (1974).

[142] J. P. Spoto, *Computer-Aided Evaluation of Operational Amplifier Performance*, Master's Thesis, Department of Electrical Engineering, University of Florida, Gainesville, Florida (October, 1974).

[143] J. M. Stewart, "Portable Software with Emphasis on Crystallography," *Proceedings of the Conference on Software Standards in Chemistry*, University of Utah, (July 1979).

[144] J. Szczupak and S. K. Mitra, "Recursive Digital Filters with Low Roundoff Noise," *Circuit Theory and Applications* vol. 5, pp. 275-286 (1977).

[145] G. Szentirmai, "FILSYN - A General Purpose Filter Synthesis Program," *Proc. IEEE* vol. 65, pp. 1443-1458 (October 1977).

[146] Y. Tang, *Optimization of Wideband Amplifier Using Feasible Directions Method*, Master's Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (June 1979).

[147] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *IEEE Computer* vol. 14, no. 4, pp. 25-32 (April 1981).

[148] A. L. Tits, *Lagrangian Based Superlinearly Convergent Algorithms for Ordinary and Semi-Infinite Optimization Problems*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (December 1980).

[149] M. E. Van Valkenburg, *Introduction to Modern Network Synthesis*, McGraw-Hill Book Company, New York, N.Y. (1962).

[150] A. Vladimirescu, A. R. Newton, and D. O. Pederson, *SPICE Version 2F.0 User's Guide*, Electronics Research Laboratory, University of California, Berkeley, California (1980).

[151] W. J. Walsh, *Computer Design of Temperature Desensitized Integrated Selective Amplifiers*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (June 1968).

[152] P. B. Weil, "Mom Told Me To Be A Dentist—Laments of a CAD Researcher," *IEEE Circuits and Systems Magazine* vol. 7, no. 7, pp. 3-5 (June 1975).

[153] L. Weinberg, *Network Analysis and Synthesis*, John Wiley & Sons, Inc., New York, N.Y. (1960).

[154] C. Weissman, *LISP 1.5 Primer*, Dickenson Publishing Company, Belmont, California (1967).

[155] J. Wieslander and H. Elmquist, "INTRAC: A Communication Module for Interactive Programs: Language Manual," CODEN LUTFD2/(TFRT-3149)/1-060/(1978), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden (August 1978).

[156] J. Wilander, "An Interactive Programming System for Pascal," *BIT*, no. 20:2, pp. 163-174 (1980).

[157] N. Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM* vol. 14, no. 4, pp. 221-227 (April 1971).

[158] N. Wirth, "Modula—A Language for Modular Multiprogramming," *Software - Practice and Experience* vol. 7, (1977).

[159] P. Wolfe, "On the Convergence of Gradient Methods Under Constraints," IBM Research Report RC 1752, IBM Yorktown Research Facility, Yorktown Heights, N.Y. (1967).

[160] P. Woodward and S. G. Bond, *Algol 68-R Users Handbook*, HMSO, London (1974).

[161] B. A. Wooley, "The Design Optimization of Integrated Broadband Amplifiers," Memo No. ERL M284, Electronics Research Laboratory, University of California, Berkeley, California (September 1970).

[162] G. Zoutendijk, *Methods of Feasible Directions*, Elsevier Scientific Publishing Company, Amsterdam, Netherlands (1960).

APPENDIX A

DELIGHT Implementation

In this appendix we consider the implementations of various aspects of the DELIGHT system. In section A.1 we discuss the implementation of the RATTLE language. This includes a discussion of the compiler-compiler approach used to generate parse tables for the RATTLE parser and how we achieved arrays whose dimensions may vary at run time. Section A.3 takes a look at the RATTLE parser and how complete statements execute when typed. Finally, section A.4 discusses how terminal-independent graphics is achieved.

A.1. RATTLE Language

A.1.1. RACC Compiler-Compiler

RACC, for "RAtfor Compiler-Compiler", is a general tool for recognizing the syntax structure of programming languages. It is called a *compiler-compiler* since it compiles statements used to describe a language compiler. RACC was written in late 1979 by this author, based on the discussion in Aho and Ullman [15] of the implementation of the UNIX compiler-compiler Yacc [70] and has primarily been used for the RATTLE language of DELIGHT[1]. The class of specifications accepted is a very general one, called *LALR(1) grammars*. The theory behind RACC is not covered here since it has been described elsewhere [15, 13, 14].

[1] This discussion is adapted in part from [70].

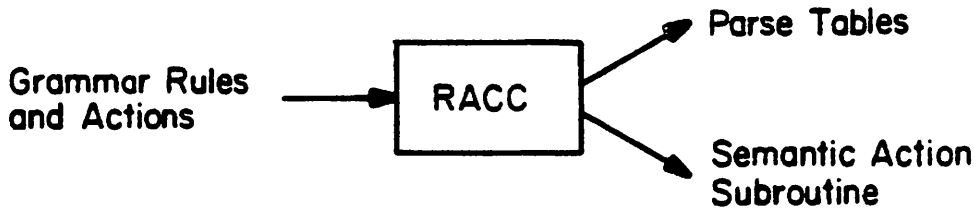
The RACC user prepares a specification of the input language. This includes rules which describe the syntax, Ratfor code to be executed when the syntax structures are recognized, and a low-level routine to do the basic input. RACC then produces a file containing tables that are used by a subroutine called a *parser driver routine*. A driver routine together with its parse table is called a *parser* and the driver is usually the same for all parsers. To parse (recognize) the input language, the driver routine calls the low-level input routine, called the *lexical analyzer*, to pick up basic items called *tokens* from the input. These tokens are organized by the parser according to the given syntax structure rules, called *grammar rules*. When one of these rules has been recognized, the Ratfor code supplied with this rule, called the *semantic action*, is executed. Semantic actions have the ability to return values and make use of the values of other actions.

Note that RACC simply reads a file containing the grammar rules and their associated actions and outputs a file containing the parse table and a file with a subroutine containing all the gathered up actions. This subroutine is then compiled and load/linked with the driver routine, the lexical analyzer, and any other routines to form the language recognizer or *compiler*. At that point, RACC no longer plays any role. This process is shown in figure A.1.

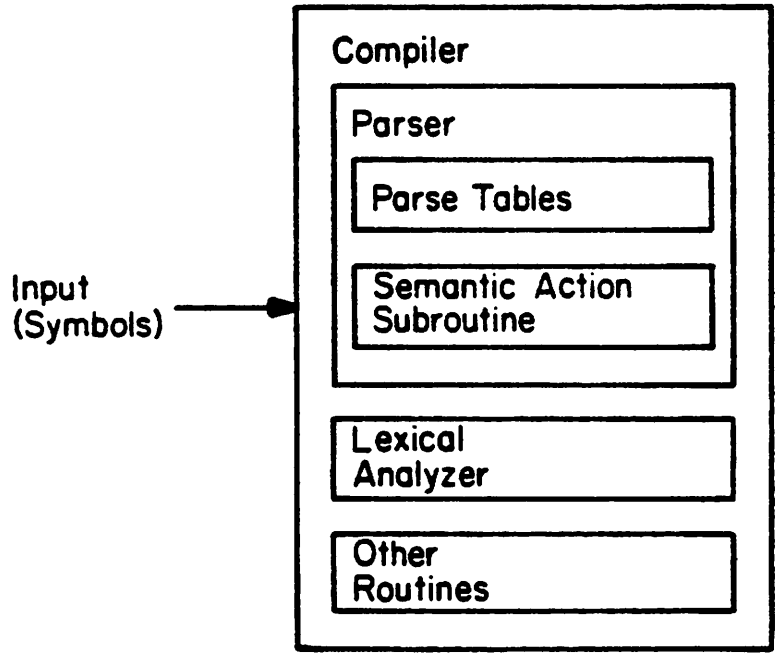
A grammar rule describes an allowable syntax structure and gives it a name. For example, the rule associated with a (simplied) RATTLE *for* statement might be

```
<for-stmt>  =  for <name> = <number> to <number> <statement>
```

Here, *<for-stmt>* is the name of the rule. The first equal sign has no other significance other than serving as punctuation. All of the items to the right of



(a) Generating the Parser



(b) Structure of the Compiler

Figure A.1. Generating a Language Compiler Using RACC.

the first equal sign represent the structure of interest—in this case, the *for* statement. Items that are NOT surrounded by the triangular brackets "<" and ">" are called *terminal symbols* [15] and must appear literally in the input to a RACC parser. These are *for*, the equal sign, and *to* in the above example. Terminal symbols are usually items which are more conveniently or efficiently recognized directly by the lexical analyzer. For example, it is easy to write a program that can recognize the next input word such as *for* and return it as one item to the parser instead of returning the three letters *f*, *o*, and *r* separately.

The items surrounded by triangular brackets above, *<name>*, *<number>*, and *<statement>*, are called *nonterminal symbols*. This name arises because they do not "terminate" or act as leaves of a *parse tree* diagraming the structure. Instead, they act as inner nodes of the tree which must have "son" branches coming from them that ultimately end in terminal symbols, i.e., nonterminals must themselves be the names of other grammar rules. Before giving these rules for our example, we need to consider the following.

The standard lexical analyzer used with the parser associated with RACC has built in several terminal symbols that it recognizes automatically. The built-in symbols are shown in the following table:

Built-in Lexical Analyzer Terminal Symbols	
Symbol	Description
xname	Any sequence of letters, digits, or underscores, beginning with a letter or underscore.
xrealnum	Any real floating-point number using the same rules for exponents and scale factors presented in section 4.2.5 for RATTLE numbers.
xinteger	Any integer. A number is an integer if it consists of a sequence of digits and nothing else, i.e., no decimal point, scale factors, or exponent.
xnewline	The (fictitious) NEWLINE character at the end of every input line.
xexpr	A complete RATTLE expression, using the balanced parenthesis rule: an expression ends at the first blank or tab following balanced parenthesis.
xassmt	A complete RATTLE assignment statement.
xstring	Any quoted string, quoted by either single or double quotes.

Naturally these symbols can appear on the right-hand side of any grammar rule.

We can now give the complete grammar for the (simplified) *for* statement. For simplicity, we allow the body of the *for* loop to only be a RATTLE assignment statement:

```
<for-stmt>  =  for <name> = <number> to <number> <statement>
<name>      =  xname
<number>    =  xinteger
<number>    =  xrealnum
<statement> =  xassmt
```

In the grammar rules input to RACC there are often rules with the same name

(to the left of the equal sign) such as the third and fourth rules above. These give alternate syntax structures to a nonterminal such as *<number>* above and may be written more compactly using the vertical bar ("|") to mean "or":

```
<number>  =  xinteger | xrealnum
```

In general, the grammar rules input to RACC can be in any order. However, the name of the *first* grammar rule has special importance; it is taken to be the controlling nonterminal symbol for the entire programming language. Using the definition in [15], it is called the *start symbol*. In effect, the parser is designed to recognize the start symbol. Thus, this symbol generally represents the largest, most general structure described by the grammar rules. In the grammar for RATTLE, the start symbol is called *<executable-stmts>*, as shown in the first two rules for this grammar:

```
<executable-stmts>   =   <statement-list>
<statement-list>     =   <statement>  |
                         <statement-list>  ;  <statement>
```

The second grammar rule above says that a statement list is either a single statement or a list of statements separated by semicolons. The rule is *recursive* since the name of the rule, *<statement-list>*, also appears on the right-hand side. This is perfectly legal since a single statement would be recognized first, possibly followed by a semicolon and another statement.

The end of the input to the parser is signaled by a special token called the *end-of-input symbol*. If all tokens read up to, but not including, the end-of-input symbol form a structure which matches the start symbol (i.e., matches its right-hand side), the parser subroutine returns to its caller if the end-of-input symbol is seen next. We say that the parser *accepts* the input. Thus, the end-

of-input symbol implicitly appears in the start symbol grammar rule to the far right, as shown in the following rewrite of the first rule above:

```
<executable-stmts>  =  <statement-list> end-of-input
```

It is the job of the lexical analyzer to return the end-of-input symbol when appropriate. In the RATTLE lexical analyzer, it is returned directly after every input line, i.e., directly after returning terminal *xnewline*. In this way if the symbol is expected, the right-hand side of the above rule is recognized and the parser returns, ready to execute what has just been typed. If the end-of-input symbol is not expected, i.e., it is grammatically incorrect according to the grammar rules, the parser detects an error, which in this particular case is quietly ignored.

If all the parser did was to recognize correct language syntax, it would serve little purpose. What it must do is create some intermediate list containing instructions codes that can be used to rapidly execute the statement(s) just typed. The lines of code that create this list and do other things are called *semantic actions*. To each grammar rule a semantic action may be associated that is performed each time the rule is used to recognize parser input. This is called *syntax-directed translation* [15, chapter 7] and allows the parser designer to express the generation of instruction codes directly in terms of the syntactic structure of the source language. In RACC input, a semantic action is one or more Ratfor statements enclosed in curly braces "{" and "}" directly following the right-hand side of a grammar rule. For example, the two rules

```
<number>  =  xinteger  { flag = INTEGER }
<number>  =  xrealnum  { flag = REALNUMBER }
```

each have associated a semantic action consisting of a single Ratfor assignment

to a flag variable that might be used by the *for* loop semantic action to determine whether an integer or real was recognized for the *<number>* nonterminal. Here, we assume that *REALNUMBER* and *INTEGER* are Ratfor defines.

But the *for* loop grammar rule

```
<for-stmt>  =  for <name> = <number> to <number> <statement>
```

contains two occurrences of *<number>*. If one were integer and the other real, the single variable *flag* would retain the value based on the second number. For this reason, there is a *dollar sign convention* similar to the "$$" convention in Yacc that allows a semantic action to return several values (that may be accessed in other actions) by setting one of the pseudo-variables *$i*, *$j*, or *$k* to any integer value. For example, an action which does nothing but set *$i* to one is

```
{ $i = 1 }
```

To obtain the values set by previous actions, a semantic action may use the integer pseudo-variables *$i1*, *$i2*, *$i3*, ..., *$j1*, *$j2*, *$j3*, ..., and *$k1*, *$k2*, *$k3*, ... where the numbers refer to distinct items on the right-hand side of a rule, reading from left to right. For the *for* loop rule above the two *<number>* nonterminals are the 4'th and 6'th items on the right-hand side. Thus, *$i4* has the value *$i* was set to by the action for the first *<number>* while *$i6* has the value *$i* was set to by the action for the second *<number>*. The problem above with the single flag variable can now be handled with the following grammar rules:

```
<for-stmt>  =  for <name> = <number> to <number> <statement>
      {
        numflag1 = $i4
        numflag2 = $i6
      }
<number>    =  xinteger  { $i = INTEGER }
<number>    =  xrealnum  { $i = REALNUMBER }
```

If the statement *for k = 2 to 5.0 y = ...* was parsed by this grammar, after recognizing (and stacking) the *for*, *k*, and =, the *2* would next be recognized causing execution of *$i=INTEGER* and then the *5.0* causing execution of *$i=REALNUMBER*. Thus, after the whole statement was parsed, *numflag1* would be *INTEGER* while *numflag2* would be *REALNUMBER*.

Two other dollar sign convention pseudo-variables are *$r* and *$s*. *$r1*, *$r2*, etc. always contain the real value of any number terminals read. *$s1*, *$s2*, etc. always contain the character string (in the sense used in [79]) of the actual token read for any type of terminal symbol. Thus, the semantic action for the *for* loop grammar rule above could obtain the real values of both *<number>* items by including the lines

```
value1 = $r4
value2 = $r6
```

in the action.

A special terminal symbol recognized by RACC is *xempty* which signifies an empty input token. See [70] for more on the use of grammar rules whose right-hand sides are empty.

Not unfrequently, the input statements being read do not conform to the syntax structure of the language due to errors. The parsers produced by RACC have the very desirable property that they can detect these input errors at the *earliest* place at which this can be done with a left-to-right scan [15]. Thus, input

errors can be detected and reported quickly and efficiently.

All of the features discussed in this section (plus a few more) have been used to create a set of grammar rules plus semantic actions for the RATTLE language. Intermediate list instructions codes are generated for RATTLE statements using the *back-patching* technique explained in [15]. In the DELIGHT source directory, the grammar rules and actions reside in a file called *grlang* (for "GRammar for LANGuage"). After running RACC on this file with the command *racc grlang*, Ratfor file *rruser.r* contains all of the actions gathered from file *grlang*. It is compiled and load/linked with the other DELIGHT routines. RACC also outputs the parse tables in file *rrdata*. This file is automatically read by DELIGHT during a *-force* option startup for creating generally used memfiles (see section B.2).

Incidentally, RACC has been successfully used to implement a grammar for SPICE input that allows an arbitrary expression anywhere in the circuit description file that a numeric value would normally be placed. This input parser gives DELIGHT.SPICE the same expressions capabilities of SLICE [9] and will soon be coupled to DELIGHT optimization in a way that will allow *any* SPICE input parameter to be an optimization design parameter. Presently, only those circuit parameters shown in the first table in section 5.1.3 may be design parameters.

A.1.2. Dynamic Arrays

This section presents a brief discussion of how the dimensions of RATTLE arrays may be expressions that vary at run time. This is accomplished by allocating the storage space for RATTLE arrays in one large array, using a *dynamic memory manager* that allows the allocated *dynamic arrays* to expand or shrink at any time. The implementation of such a memory manager may be tackled in many ways.

One approach is to start with one large program array and then allocate new dynamic arrays on the "top" of (at one end of) the large array—a stack allocation technique. This permits only the array on the top to expand or shrink dynamically. If expansion of the allocated array below the top is desired, then the top array must first be eliminated; the arrays must be eliminated in the reverse of the order in which they were allocated.

In order to handle the more general case of several dynamic arrays being created or expanded at the same time and in an arbitrary order, a more sophisticated memory allocation strategy is needed. The *Boundary Tag Method* [67, 83] is a heap allocation technique which starts out with the one large array being considered as one big free block. When a dynamic array is requested, it becomes an allocated block in this large free block. When arrays are cleared, *free* blocks are left behind. These free blocks are added to a doubly linked list of such blocks, used to facilitate the location of a free block for subsequent dynamic array allocations. Following [67], a *first-fit strategy* is used when searching down the list of free blocks for one whose size is greater than or equal to the requested size. In fact, the internal routines of the memory manager are closely patterned after procedures *allocate* and *free* in [67]. When no free block can be found large enough to meet the size of the requested dynamic array, internal subroutine *crunch* is called to compress all the allocated blocks together leaving one large free block. If this block is still not large enough, the program using the memory manager has run out of memory.

To expand the size of existing dynamic arrays, the slightly more complicated procedure used is as follows:

(1) Try to allocate a new dynamic array of size equal to the new total size, i.e., the old size plus the expansion size. If successful, copy the old array into

the new, remove the old, and return.

(2) If the new array cannot be allocated, no existing free block is large enough. Therefore, call subroutine *crunch* to compress all the allocated blocks together leaving one large free block the size of remaining memory. Then, try to allocate an array with the new total size again.

(3) If not successful, perform a "double-swap-flop" [71] to move the dynamic array being expanded right below the one free block left after a crunch. Then, it may be expanded into this free block even though there is not enough room for the new total size. It this space is not enough, again the program using the memory manager has run out of memory.

Under any conditions, the least amount of work to expand an array is to copy the entire array one time. This implies that expanding a large array can be costly and thus the number of expansions should be minimized. This can be achieved by expanding by an amount greater than the new amount actually needed. To give an example of this technique in RATTLE, suppose we are reading into an array an unknown number of numbers. The inefficient approach uses something like

```
k = 0
repeat  {
   k = k + 1
   array data(k)
   read data(k)
   if ( end_of_data )
      break
   }
forever
```

while the more efficient technique might use

```
array data(0)
k = 0
repeat  {
   k = k + 1
   if ( k > arydim(data) )
      array data(k+50)
   read data(k)
   if ( end_of_data )
      break
   }
forever
```

In the above, *arydim* is a DELIGHT built-in function that returns the array dimension of its argument. Also note that although the second approach requires array *data* to be expanded in chunks of 50, for a large number of data points it requires 50 times fewer expansions of the array. The second approach thus exhibits much greater run-time efficiency at the small cost of possibly 49 wasted (unused) array elements.

One final note is that dynamic arrays using this same memory manager are also used throughout most of the actual DELIGHT Ratfor source routines. How Ratfor (Fortran) is given such "unheard-of" power is by using the modified Ratfor preprocessor explained fully by this author in [108].

A.2. Parser and Parse/Execute Loop

The RATTLE language is compiled using an efficient bottom-up parser called an *LALR(1) parser* [15], named because it scans the input using LookAhead from Left to right and constructs a Rightmost derivation in reverse. The LALR method will work on most programming-language grammars. Although possible, the LALR parser is *not* used to parse RATTLE expressions or assignments but only for "higher level" RATTLE syntax such as *for* loops, *while* statements, etc. This was seen in the table of section A.1.1 where terminal symbols *xexpr* for expressions and *xassmt* for assignments were considered built-in by the standard lexical analyzer. The lexical analyzer parses expressions and assignments using an

operator precedence technique [15, 56] in which the *precedence table* has been compacted into two *precedence functions* [66] which represent the in-stack versus the incoming precedences of the various arithmetic operators. For more on implementing such an parser, see Gries [56]; the expression parser is discussed here only in the context of how and when it is called by the LALR RATTLE parser. For the following, we assume the reader has already read appendix section A.1.

An LALR parser has an input, a stack, and a parsing table. The input terminal symbols, $a_1, a_2, \cdots, a_n$ are read from left to right, one symbol at a time. For example, for the RATTLE input

```
while ( k <= 5 )  {
   y(k) = k**2
   k = k + 1
   }
```

the parser would read a_i input symbols with the following correspondences:

a_1	while	
a_2	xexpr	(expression)
a_3	{	
a_4	xnewline	
a_5	xassmt	(y(k) assignment)
a_6	xnewline	
a_7	xassmt	(k assignment)
a_8	xnewline	
a_9	}	
a_{10}	xnewline	

where *xexpr*, *xassmt*, and *xnewline* are built-in lexical analyzer terminal symbols defined in section A.1.1. The stack contains a string of the form $s_0X_1s_1X_2s_2 \cdots X_ms_m$, where s_m is on top of the stack. Each X_i is a grammar symbol, either terminal such as *for* or nonterminal such as *<for-stmt>*, and each s_i is a symbol called a *state*. Each state symbol summarizes the informa-

tion contained in the stack below it and plays a central role in the action of the parser. The parsing table consists of two parts, a parsing action function ACTION and a goto function GOTO.

The action of the parser driver routine is as follows. It first determines s_m, the state currently on top of the stack, and picks up a_i, the current input symbol. It then consults ACTION(s_m,a_i), the parsing action table entry for state s_m and input a_i, to determine the next move of the parser. The entry ACTION(s_m,a_i) can have one of four values given below. Also needed is the function GOTO, which takes a state and a grammar symbol as arguments and returns a state. It is essentially the transition table of the finite-state machine implemented by the driver routine. The four possible values of entry ACTION(s_m,a_i) then are:

1. **shift** s, indicating that the parser is to execute a shift move, pushing current input symbol a_i and next state s onto the stack, where s=GOTO(s_m,a_i).
2. **reduce** on grammar rule <*A*> = *right-hand-side*, indicating that the parser is to execute a reduce move by popping the top grammar symbols (and their accompanying state symbols) off of the stack that match the *right-hand-side* symbols of the grammar rule. The parser then pushes <*A*>, the name of the grammar rule, and s onto the stack, where s=GOTO(s_{newtop},<*A*>). The current input symbol is not changed in a reduce move. Below we see that this requires a *repeat* loop to reuse the same input symbol.
3. **accept**, meaning that parsing is completed and the parser driver routine returns.
4. **error**, indicating that the parser has discovered a syntax error in the input and either calls an error recovery routine or prints an error message.

The LALR parsing algorithm is very simple. Initially the parser has just the initial state s_0 on the stack and input $a_1 a_2 \cdots a_n$ waiting to be parsed. Then, in the *major parse loop* (see below), the parser executes shift or reduce moves according to the ACTION and GOTO tables until an accept or error action is encountered. In DELIGHT, the parser driver routine is subroutine *RattleParser*[2]. Before presenting it we discuss how this parser is used to execute RATTLE statements as they are typed.

In the discussion of incremental program development in 4.2.7 we saw that in DELIGHT *complete* RATTLE statements execute when typed. This is accomplished by setting up the RATTLE grammar rules input to RACC so that the parser encounters an accept action and returns when a complete RATTLE statement is typed. (For this purpose, all of the statements of a procedure and its body are considered as one complete statement; however, procedures are handled in a special way as shown below.) A conceptual version of the DELIGHT main program thus appears:

```
call DelightInitialization
repeat  {    # PARSE/EXECUTE LOOP.
   call RattleParser   to parse one complete statement.
   if ( no ERROR and statement is not a procedure )
      call RattleExecute
   }
forever
```

Subroutine *RattleParser*, by executing the semantic actions associated with the RATTLE language grammar rules (from file *grlang*—see section A.1.1), creates an intermediate list of codes that are used by subroutine *RattleExecute* to execute the RATTLE statements parsed.

---

[2] Throughout this section, long, self-explanatory subroutine names are used. A table at the end of the section gives the corresponding actual Ratfor subroutine names.

We now show conceptually what subroutine *DelightInitialization*, called first in the main program above, contains:

```
subroutine DelightInitialization

call SetupMemory       to set up dynamic memory manager
call SetupInterrupts   to set up to catch hard interrupts
call SetupOverflows    to catch overflows and other floating
                       point exceptions

if ( a memfile name is given in the "DELIGHT" command arguments )  {
   call RestoreStore   to restore from the memfile
   return
   }

if ( "-force" option is given in the command arguments )  {

   # Begin forced startup:

   call ExpressionParserInit     to read expression operators and
                                 precedences from file <exops> and
                                 to read built-in sin, cos, etc.
                                 functions from file <exfuns>.

   call ParserInit      to allocate dynamic array for the parse stack
                        and to read the parse tables from file <rrdata>.

   call ReadFunctions   to read built-in routine names and number
                        of arguments from file <rrfuns>. Also, if
                        file builtnam exists, read name/arg-count
                        pairs for built-in routines in subroutine abuilt.

   Declare many Fortran (Ratfor) variables for RATTLE access using the
      deci, decial, decr, etc. routines from appendix section B.2.

   Push back "include <standefs>" to include all the standard defines
      and macros.
   }

return      to the main program to the parse/execute loop.
end
```

Before presenting the LALR parser subroutine, there is a consideration of how it interfaces to the expression parser. At the top of the major parse loop the parser driver subroutine calls lexical analyzer subroutine *LexicalToken* to get the next input symbol and it is this subroutine which directly calls expression parser subroutine *ExpressionParser*. The latter call is made when three conditions are met:

(1) the incoming input symbol can possibly *start* a RATTLE expression. This excludes NEWLINE, quoted strings, and RATTLE keyword terminal symbols such as *while*, *array*, etc. and is determined in *LexicalToken*.

(2) the expression symbol *xexpr* is a valid input symbol in the present parse state (the state on top of the stack). This is tested by insuring that, of the four possible values of an ACTION table entry, the entry ACTION(s_{top},*xexpr*) is not an *error* entry. (See section A.1.1 concerning symbol *xexpr*.) This test is performed in *RattleParser* at the top of the major parse loop, where flag *gexpr* is set to indicate that an expression is expected. *gexpr* is communicated to *LexicalToken* through a Fortran common block.

(3) an expression is expected as a *shift* entry, i.e., ACTION(s_{top},*xexpr*) is a shift entry. This is required so that the intermediate list from the expression parser is generated at the right place in the overall list. The test is in *RattleParser* right under the one above and sets flag *getnow* to tell *LexicalToken* to "get it now", i.e., parse the expression immediately. If *getnow* is not set, *LexicalToken* will still return the *xexpr* symbol but will not call *ExpressionParser* to actually parse the expression. This goes on until *RattleParser* performs enough reduce moves to get into a state in which an expression *is* expected as a shift entry.

Another incidental function performed by subroutine *LexicalToken* is to make history substitutions on input lines that begin with "!" (see section 4.8)

We are now ready to show conceptually what subroutine *RattleParser* contains:

```
subroutine RattleParser

repeat  {   # MAJOR PARSE LOOP.

   if ( hard interrupt generated )  {
      Set flag that what was just parsed contains an error so this
         code will not be executed by RattleExecute.
      call AllFlush    to close all input files being included and flush
                       all input buffers and pushed back characters.
      return
      }

   if ( expression is a valid input symbol )  {        # Condition (2)
      gexpr = YES
      if ( expression is expected as a shift entry )  # Condition (3)
         getnow = YES
      }

   call LexicalToken[3]    to get the next input grammar symbol into
                           variable InputSymbol.
   repeat  {         # Loop for reusing the same input symbol.

      Get TopState from top of parse stack

      action = ACTION(TopState, InputSymbol)

      if ( action is accept )
         return

      if ( action is shift )  {
         Push InputSymbol onto parse stack.
         TopState = action     # action positive indicates a shift and
                               # action is coded with the value of
                               # GOTO(TopState, InputSymbol))
         next 2      # Go to top of MAJOR PARSE LOOP.
         }
```

[3] *LexicalToken* converts "raw" input tokens such as names, newlines, or special characters into grammar symbols by looking up all names in the DELIGHT symbol table to see if they are a RATTLE keyword or keysymbol such as *if*, *while*, {, =, etc. *LexicalToken* may return any of the following grammar symbols:

- xinteger
- xrealnum
- xname
- xnewline
- end-of-input
- a grammar terminal symbol that is a RATTLE keyword or keysymbol
- xstring
- xexpr
- xassmt

```
      if ( action is reduce ) {
         Reduce via grammar rule number -action  # action negative indicates a
                                                 # reduce and -action is coded
                                                 # with the number of the
                                                 # grammar rule to reduce by.
         call ExecuteAction    to execute the semantic action code
                               associated with grammar rule number -action.
         next        # Repeat loop with the same input symbol.
         }

      if ( action is error )
         Perform error recovery.

      }
   forever

   }
forever        # End of MAJOR PARSE LOOP.

return
end
```

The above subroutine *RattleParser* implements the LALR parsing algorithm; after called it executes until an accept or error action is encountered, as shown above.

One final subroutine to consider is *RattleExecute*. It is called in the DELIGHT main program parse/execute loop. A *very brief* conceptual view of this routine follows:

```
subroutine RattleExecute
repeat  {
   Get next instruction opcode from intermediate list.
   # Branch on the opcode with a large Fortran computed goto:
   go to ( if, print, assignment, etc., end-of-list ) , opcode
   if:            ...
                  Go to ExpressionEvaluation
   while:         ...
                  Go to ExpressionEvaluation
   print:         ...
                  Go to ExpressionEvaluation
   ...      # Other statement types.
```

```
end-of-list:  if ( stack not empty ) {
                 Pop execution state from stack.
                 next     # Execute next statement.
                 }
              else
                 return   # Finished with execution.

ExpressionEvaluation:

              call ExpressionEvaluate

              if ( return code is end-of-expression )
                 next     # Execute next statement.

              if ( return code is function-call ) {

                 if ( function is ordinary built-in )
                    call builtn

                 if ( function is application built-in )
                    call abuilt   # (See section B.1)

                 if ( function is RATTLE ) {
                    Stack execution state.
                    Set intermediate list pointer to
                       point to function's list.
                    next
                    }
                 }
   }
forever
```

Subroutine *abuilt*, called above, contains application-specific built-in routines that were added to DELIGHT using the procedures explained in section B.1.

The following list gives the Ratfor subroutine names in the right column for the various routines of this section.

Name Used in This Section		Ratfor Name
ParserInit	-	rrinit
RattleParser	-	rrpars
ExecuteAction	-	rruser
LexicalToken	-	rrtok
ExpressionParserInit	-	exinit
ExpressionParser	-	expars
RattleExecute	-	excute
ExpressionEvaluate	-	exevas

DelightInitialization	-	dlinit
SetupMemory	-	setmem
SetupInterrupts	-	setint
SetupOverflows	-	setovf
RestoreStore	-	rrstor
ReadFunctions	-	rrrfun
AllFlush	-	aflush

A.3. Device-Independent Graphics

This section first gives an overview of why the graphics features in DELIGHT should be independent of the particular graphics display device[4] used. It then explains just what happens when the graphics terminal type is set using the *terminal* command. In doing so, it explains exactly *how* the graphics features are independent of the particular device used.

There is a strong case for writing application programs that output graphics by using a set of low-level graphics primitives that are independent of the particular graphics display device used. As pointed out by Oliver in [118], such a set of routines would be useful for a number of reasons:

(1) To have a standard (known) set of graphics routines which any application can use.

(2) To separate the graphics generation and control in an application program from the rest of the code.

(3) To allow the graphical displays produced by a program to be output on a number of different display devices, i.e., to achieve a high degree of *device-independence* both for existing terminals and future additions.

[4] Throughout this section, *display device* and *terminal* are used interchangeably and have the same meaning.

(4) To have the ability to produce *hardcopy* (printed graphical output) without any additional programming.

Such a library of low-level, device-independent graphics primitives has been written and included as built-in routines in DELIGHT. A list of these routines will not be given here; such a list can be found in [110]. Instead, we give a general description of the library and then concentrate on subroutine *grterm*, called by the DELIGHT *terminal* command to set the graphics terminal type.

The graphics library allows graphical displays to be output on an ever expanding number of graphics terminals and devices. Library routines perform the task of sending the different terminal-dependent, "funny" character strings required to control the various modes of each graphics device. Thus, such character strings need never appear in any graphics application code. This is a radical departure from the past when many graphics programs explicitly contained the control strings for such popular terminals as the Tektronix 4010 directly in their output statements. Thus, these programs were intimately tied to one particular graphics terminal.

The terminal-dependent control strings for the various display devices handled by the graphics library are all contained in a file called *<grtdefs>*, for "GRaphics Terminal DEFinitionS"[5]. This file also contains descriptions of the capabilities and special features of each individual device. If a new display device is acquired, it can be used with the library routines (and hence with all graphics applications based on them) after the appropriate device descriptions have been added to file *<grtdefs>*. This means that no new routines have to be written to deal with a particular display device—unless of course an operation

[5] From inside DELIGHT, this file can be printed on the screen by typing *list <grtdefs>*. (The triangular brackets surrounding the filename mean the file resides in a standard place (directory) instead of with the current user's files.)

must be performed which is not supported by the library routines.

The idea for such a file came from the TERMCAP feature of the UNIX *vi* screen editor [73]. Also, another device-independent graphics package that appeared after our library and that uses a terminal capabilities file is the MFB (Model Frame Buffer) package of Keller et al. [22].

When the *terminal* command is given in DELIGHT, built-in subroutine *grterm* is passed the specified terminal name argument. *grterm* opens file *<grtdefs>*, locates the part containing the description for the specified terminal, and reads the control strings and capabilities for that terminal. To locate the desired description, *grterm* looks for a match on lines that contain terminal names; these lines DO NOT begin with a blank. All lines that DO begin with a blank contain the actual terminal descriptions. Terminal capabilities are recognized by the occurrence of certain keywords in this description while terminal control strings follow certain other keywords. A portion of file *<grtdefs>* might appear:

```
hp
hp2648
hp2648a
   binary vechead '\e*pa' vectail 'Z' abshead 'i' relhead 'j'
   hidev 32 mask 31 hiyor 32 loyor 32 hixor 32 loxor 32 relmax 15 relmin -16
   delxor 32 delyor 32 reladd 32 texthead '\e*l' texttail '\n' chardelx 7
   chardely 10 symax 359 sxmax 719 sysqr 359 sxsqr 359 colorhead '\e*m2a'
   colortail 'B' erase '\e*dA' colors 10 white 1 black '1a1' red 9 orange 5
   yellow 6 green 4 blue 8 sky 7 light 7 bright 1 werasehead '\e*m1a1b'
   werasetail 'E\e*m2aE' werasediagc poscurhead '\e*dk\e*d' poscurtail 'O'
   tsizehead '\e*m' tdirtail N textsizes 1 2 3 4 5 6 7 8 8 8 tsizetail M
   tdirhead '\e*m' tdircalc 90 1 init '\e*m1m\e*m1N' werasedelays 1.0 1.7 1.7

4027
tek4027
   ascii coorsep ' ' abshead '!vec' relhead '!rve0 0 ' nbeforerel 2
   textsub 1 chardelx 8 chardely 10 erase '!era g' werasehead '!pol' end '\n'
   symax 349 sxmax 639 sysqr 349 sxsqr 349 colorhead '!col c' colors 10 white 0
   black 7 red 1 orange 6 yellow 4 green 2 blue 3 sky 5 light 4 bright 0
   init '!wor30!gra1,29!mix c6 100,60,0!mix c5 0,100,100\n\'
   '!lea f1 31/wor0/13\nPRESS f1 FOR TOP OF SCREEN.\n' werasedelays 1 4 4
   textsizes 1 2 3 4 5 6 7 8 9 10 texthead '!str/' tsizehead '' texttail '/'
```

The above portion describes the HP2648a and Tektronix 4027 terminals. Termi-

nal names such as *hp*, *hp2648*, and *hp2648a* above that are on adjacent lines are *synonyms* for the same terminal. Capability keywords such as *binary* and *ascii* stand alone whereas most other keywords above are followed by control strings.

There is a simple rule which tells how many of the characters following such a keyword are considered the associated control string. After a keyword is read by *grterm*, a special routine is called which returns exactly one *token*—an integer, real number, name, quoted string, or a single non-letter, non-digit character. Thus, the rule is that the characters following a keyword that make up exactly one token are considered the control string or *value* of that keyword.

The backslash character ("\") in a token is an *escape character* that changes the meaning of the succeeding character according to the following correspondences:

Escape	Meaning	ASCII Value (decimal)
\e	ASCII escape	27
\n	NEWLINE	10
\r	carriage return	13
\\	the character \ itself	92
\nnn	character whose ASCII value is nnn	nnn
\C	C itself for any other character	-

For example, to erase the screen on an HP2648a, the control string following the *erase* keyword is sent to the terminal. From the above this is "\e*dA", which indicates the ASCII escape character followed by the three characters "*dA". One additional feature of the backslash character is to continue a quoted string onto the next line. Thus, if

```
init '!wor 0 \'
'!mix c8 100,50,30'
```

appeared in file *<grtdefs>*, it would be the same as if

```
init '!wor 0 !mix c8 100,50,30'
```

had appeared there. This feature of the backslash character is used above in the control string following the *init* keyword for terminal *tek4027*.

A list of all keywords recognized by the graphics library along with a description and default value for each is given in file *<keywords>*.

APPENDIX B

DELIGHT Application Package Development Features

This appendix shows the details of several DELIGHT features for applications package development and also useful for other purposes. Recall that to be able to easily develop such packages, it must be easy to interface to existing simulation programs. This requires at the least the following two features. Section B.1 shows how to easily add existing routines to a set of DELIGHT built-in routines which are callable from RATTLE. Section B.2 shows how Fortran variables are accessed from built-in routines so that they can be manipulated in the same manner as ordinary RATTLE variables and arrays.

B.1. Adding Built-in Routines

This section describes how to add existing Fortran routines to DELIGHT so that they are callable from RATTLE procedures with exactly the same syntax as the RATTLE procedures themselves. These routines might be simulation interface routines for a particular simulation program, utility routines, library routines, or routines containing any computation which needs the greater run-time efficiency of Fortran (equivalently Ratfor) over RATTLE. Note that another option is to translate the Fortran routines into RATTLE. This translation could be computationally more costly since programs written in RATTLE usually run slower than their Fortran equivalents. In addition, translating a subroutine into RATTLE could be costly in terms of programmer time since Fortran routines are often structureless and may be next to impossible to translate into the struc-

tured, "goto-less" RATTLE language. Thus, such translation should be avoided.

The addition of a new built-in routine to DELIGHT requires three operations:

1. make DELIGHT aware of the routine,
2. allow DELIGHT to call the routine, and
3. load/link the routine with DELIGHT.

To make DELIGHT aware of a new routine, a one line entry is added to file *builtnam*, which should reside in the directory where the general user *memfile* (see section 4.8) is to be created. (See [110] for more details on making memfiles and the *-force* option for starting DELIGHT.) This entry associates a RATTLE name with the routine and consists of the name by which the new routine is going to be known to RATTLE and the number of arguments to the routine. The RATTLE name need not be the same as the actual Fortran name. However in general, a good idea is to use either the Fortran name (perhaps ending in an underscore, to make it a "system entity" to avoid name clashes with user names—see section 4.8) or a more explanatory name. To allow DELIGHT to call a new routine, a call to it is added to Ratfor subroutine *abuilt* ("Application BUILT-in's"). All the calls in *abuilt* must be in one-to-one correspondence with the entries in file *builtnam*. Finally, the procedure for load/linking a new routine with DELIGHT is highly system-dependent and will not be covered here. For the UNIX system, however, see [139].

As an example of the first two operations, suppose we wish to build in to DELIGHT the two Fortran subroutines *clrnum* and *clrden*, each having no arguments. The names by which these are known to RATTLE can be arbitrary but in our case, we let them be known by the self-explanatory names *ClearNumerator* and *ClearDenominator*. Thus, in file *builtnam* we would have:

```
ClearNumerator   0
ClearDenominator 0
```

while Ratfor subroutine *abuilt* simply requires a computed goto entry (based on argument *funcno*, the entry number) and a call statement for each. A *conceptual* version of this subroutine would appear:

```
      subroutine abuilt (funcno)
      go to (1,2), funcno
1     call clrnum
      return
2     call clrden
      return
      end
```

After this subroutine had been load/linked with DELIGHT, a memfile created, and DELIGHT started from this memfile, a user could type *ClearNumerator()* to have subroutine *clrnum* execute and *ClearDenominator()* to have *clrden* execute.

In reality, subroutine *abuilt* would be a bit more complex than this. Other considerations it must handle include passing arguments to the built-in routines, returning a function value from a built-in routine that is to act like a function in RATTLE expressions, and special considerations for passing and receiving back *integer* arguments. Integer arguments are a consideration because all variables in RATTLE are presently double-precision floating-point numbers (see section 4.2.4). Thus to pass integer arguments, the RATTLE double-precisions arguments must either be copied into temporary integers, copied back to double from temporary integers, or both. For these purposes, there is a large work array called *iwork* (see below) that can be used for this temporary copying. Subroutine *rcopyi (D, I, N)* can be used to copy *N* items from double-precision array *D* to integer array *I*. Similarly, subroutine *icopyr (I, D, N)* can be used to copy integers back into double-precision arrays. When assigning to scalar

integer temporaries from double-precision arguments, DELIGHT function *iround* should be used to round the doubles and avoid roundoff errors. These techniques are shown in the example below.

Inside subroutine *abuilt*, RATTLE arguments are received via the Fortran double-precision array *rarray*, with the first argument in *rarray(e1)*, the second in *rarray(e2)*, etc. For double-precision arguments of a built-in routine, *rarray* can be used to simply "pass the RATTLE arguments through", as shown in the example below. For integer arguments, as mentioned in the previous paragraph, *rarray* entries must be copied into or out of integer temporaries. To have a built-in routine return a function value, *rarray(retp)* is assigned the value to be returned.

To give a brief example of the other features of subroutine *abuilt* and the above argument techniques, we now consider an example with again two built-in routines. The first is to be known as *FuncExamp* to RATTLE, have Fortran name *funcex*, and return a double-precision function value with one double-precision argument. The second is to be known as *ProcExamp* to RATTLE, have Fortran name *procex*, and have the twelve arguments shown below. These arguments consider all the various combinations of argument types: input only (read from but never written onto), output only (only written onto), and input/output (both read from and written onto), as well as scalars and arrays, both integer and double-precision:

```
 1  -  double-precision scalar input
 2  -  double-precision scalar input/output
 3  -  double-precision scalar output
 4  -  double-precision array  input         (size N4)
 5  -  double-precision array  input/output  (size N5)
 6  -  double-precision array  output        (size N6)
 7  -  integer          scalar input
 8  -  integer          scalar input/output
 9  -  integer          scalar output
10  -  integer          array  input         (size N10)
11  -  integer          array  input/output  (size N11)
12  -  integer          array  output        (size N12)
```

For this example, file *builtnam* would contain

```
FuncExamp 1
ProcExamp 12
```

while, with " · · · " indicating other Ratfor code not shown here for clarity, subroutine *abuilt* would contain

```
  ...
  subroutine abuilt ( funcno, ..., retp, ..., iwork, ... )
  ...
  go to (1,2), funcno

1 rarray(retp) = funcex ( rarray(e1) )
  return

2 i7 = iround (rarray(e7))  # Copy inputs.
  i8 = iround (rarray(e8))  # (i7, i8, i9, and iwork are temporaries.)
  call rcopyi (rarray(e10), iwork(1),    N10)
  call rcopyi (rarray(e11), iwork(1+N10), N11)

  call procex ( rarray(e1), rarray(e2),  rarray(e3),          #  1  2  3
                rarray(e4), rarray(e5),  rarray(e6),          #  4  5  6
                i7,         i8,          i9,                  #  7  8  9
                iwork(1),   iwork(1+N10), iwork(1+N10+N11) )  # 10 11 12

  rarray(e8)  = i8          # Copy outputs.
  rarray(e9)  = i9
  call icopyr (iwork(1+N10),     rarray(e11), N11)
  call icopyr (iwork(1+N10+N11), rarray(e12), N12)
  return

  end
```

Because arguments to built-in subroutines and functions can only be double-precision or integer, modifications to the built-in routines themselves may have

to be made. Any Fortran arguments that are *real* must be converted to double-precision, to conform to the double-precision arguments which are passed from *abuilt*. This is easily done in some cases by putting an *implicit double precision (a-h,o-z)* statement at the beginning of each built-in Fortran routine, which will change the implicit typing for all real variables to double-precision. Any explicit real declarations such as *real v(10)* (as opposed to *dimension v(10)*) must be changed to double-precision as *double precision v(10)*.

For additional and more complete details about all matters in this section and the next, see appendix B of Polly Siegel's Master's Report [139].

B.2. Accessing Fortran Variables

When using existing subroutines which have been incorporated into DELIGHT, it may be necessary to access some of the variables of the routines. For example, many Fortran programs use common blocks as a means of passing or receiving information. To avoid having to make extensive modifications to these routines when they are made built-in in DELIGHT (in order to set or get the value of these common block variables), one needs to be able to directly access the variables in RATTLE statements. This can be done by creating a special built-in Fortran subroutine which contains calls to DELIGHT *variable-declaration* routines which associate each Fortran variable with a RATTLE variable name. For example, variables *FREQ* and *TIME* from DELIGHT.SPICE are declared in this way (see section 5.1.3). The RATTLE and Fortran names need not be the same. However, as for built-in routine names in the previous section, it is a good idea to use either the Fortran name (perhaps ending in an underscore, to make it a system entity) or a more explanatory name. Also, the RATTLE names could end in "_F", for example, to act as a reminder that they are Fortran declared variables.

The declaration subroutines are described in the following table. For each case, the name in quotes, which must end in a dollar sign ("$") string terminator, is the RATTLE variable name. Scalar variables and arrays declared with these routines become members of the *pool* of nonlocal RATTLE variables (see section 4.2.4).

DELIGHT Subroutines for Fortran Variable Declaration	
Subroutine Call	Action
call deci ('NAME$',ivar)	Declares Fortran integer variable *ivar*.
call decia1 ('NAME$',iary,N1)	Declares Fortran integer array *iary*, having the one dimension *N1*.
call decia2 ('NAME$',iary,N1,N2)	Declares Fortran integer array *iary*, having the two dimensions *N1* and *N2*.
call decr ('NAME$',rvar)	Declares Fortran real (double-precision) variable *rvar*.
call decra1 ('NAME$',rary,N1)	Declares Fortran real (double-precision) array *rary*, having the one dimension *N1*.
call decra2 ('NAME$',rary,N1,N2)	Declares Fortran real (double-precision) array *rary*, having the two dimensions *N1* and *N2*.

In the array declarations above, the dimensions should be identical to those of the actual Fortran array. Not shown above are subroutines *decia3* and *decra3* for declaring three-dimensional Fortran arrays.

The following example of the special built-in Fortran subroutine needed to make calls to the above declaration routines contains examples of those routines:

```
subroutine Vinit
common /cname/ ivar, iarray(200), xvar, xarray(10,20)
double precision xvar, xarray
call deci   ('ivar_F_$',   ivar)
call decia1 ('iarray_F_$', iarray, 200)
call decr   ('xvar_F_$',   xvar)
call decia2 ('xarray_F_$', xarray, 10, 20)
return
end
```

Since declared Fortran variables exist in the *pool*, they are accessed in RATTLE procedures by importing them (see section 4.2.5). For example, the following RATTLE procedure uses the variables declared above:

```
procedure SetFortranVars {
   import ivar_F_, iarray_F_, xvar_F_, xarray_F_
   ivar_F_ = ...
   for k = 1 to 200
      iarray_F_(k) = ...
   ...
}
```

If it is desired to make a declared variable global (see section 4.2.5) so that it does not need to be imported, the Fortran subroutine such as *Vinit* above can have a *call decglo ('NAME$')* statement *after* the normal declaration call. In the above example, to make RATTLE variable *ivar_F_*global, we would have:

```
call deci   ('ivar_F_$', ivar)
call decglo ('ivar_F_$')
```

See [110] for an important restriction on how declared Fortran variables can be used in RATTLE.

APPENDIX C

DELIGHT Machine-Dependent Primitives

In this appendix section we simply list the Ratfor names and descriptions of the various machine-dependent primitives used to achieve portability of the DELIGHT system. By implementing each of these primitives on a particular computer, DELIGHT should operate correctly without having to modify any other of its source routines; at the time of this writing, DELIGHT has already been ported successfully using this strategy to the following computer systems:

VAX 11/780 running Berkeley UNIX

VAX 11/780 running DEC VMS

Harris H800 running Vulcan

The primitives are broken up into the following categories:

1. Machine-dependent defines
2. Character input and output
3. Character conversions
4. Moving character bytes
5. Bit operations on integers
6. Assigning and reading variables by address
7. File opening, closing, and manipulation
8. Random access file input and output
9. Standard system calls
10. Catching hardware traps
11. Numerical analysis related
12. Miscellaneous primitives

The Ratfor routines listed have been carefully named to avoid certain machine-dependent name "conflicts". For example we would never be so foolish as to

name a routine any of the common names *close, open, read, rewind, date, time, and, delay,* etc. In the following descriptions, the meaning of each acronym routine name is given in square brackets.

Machine-Dependent Defines

File *chrdefs* — [CHaRacter DEFineS] Ratfor character defines for the computer character set (e.g., ASCII) and other symbols. For example for ASCII, it contains *define(BLANK,32), define(COLON,58), define(DIG0,48), define(DIG1,49), define(LETA,97), define(LETB,98), define(NEWLINE,10),* etc.

File *iodefs* — [I/O DEFineS] Ratfor defines related to input/output such as ones for the standard Fortran logical unit numbers, an efficient record size for direct access I/O, and the maximum number of characters in a filename.

File *machdep* — [MACHine DEPendencies] Ratfor defines for hardware related machine-dependencies. This file includes, in particular, the following (shown here for the VAX 11/780 computer):

define (NUMBYTPERI,4)	-	Number of character bytes per Fortran integer variable.
define (NUMBITPERI,32)	-	Number of bits per Fortran integer variable.
define (NUMBITPERR,32)	-	Number of bits per Fortran real variable.
define (NUMBITPERD,64)	-	Number of bits per Fortran double-precision variable.
define (NUMBITPERC,32)	-	Number of bits per Fortran complex variable.
define (NUMIPERI,1)	-	Number of integers per Fortran integer variable.
define (NUMIPERR,1)	-	Number of integers per Fortran real variable.
define (NUMIPERD,2)	-	Number of integers per Fortran double-precision variable.
define (NUMIPERC,2)	-	Number of integers per Fortran complex variable.
define (MAXREAL,1.7e38)	-	Maximum representable real number (approximate).
define (MINREAL,1.0e-37)	-	Minimum representable real number (approximate).

define (MAXINTEGER,2147483647)	-	Maximum representable integer.
define (MachinePrecision,6.002e-8)	-	Smallest single-precision number than when added to 1.0 creates a different number (approximate).

Character Input and Output

Function *getchr* [GET CHaRacter] Get character from present input, returning it simultaneously in the argument and as the function value. This function calls primitive routine *gtilin* to place the next input line into a character array.

Subroutine *grflus* [GRaphics FLUSh] Flush any characters in the graphics library buffer out in *raw mode*, i.e., without any NEWLINES.

Function *gtilin* [GeT Input LINe] Return in the given character string array the next input line from the present logical unit number, ending it with a NEWLINE character. Return as function value the position of the NEWLINE or else one of the following "irregular" codes: EOF (end-of-file); INTERRUPT (user interrupt during read-in); READERROR (error during read-in). This function may make a machine-dependent system call to avoid (time-consuming) Fortran I/O.

Subroutine *oflush* [Output FLUSH] Flush any characters in the output buffer by writing them out in raw mode without a terminating NEWLINE. This is used for example, to output the DELIGHT terminal prompt "1> ".

Subroutine *outchr* [OUTput a CHaRacter] Output a character to the present output logical unit number (set and reset with Ratfor routines *sochan* and *rochan*).

Subroutine *outraw* [OUTput RAW] Output characters from a given character array to the terminal in *raw mode*, i.e., right away without there being a NEWLINE sent later to flush any output buffers which may exist.

Character Conversions

Function *a1toc* [A1 TO Character] Convert a character in Fortran "A1" format to a character value in the Software Tools [79] sense (usually between 1 and 128).

Function *chrtyp* [CHaRacter TYPe] Determine the type of the character argument, returning either LETTER, DIGIT, SPECCHAR, or ILLEGAL.

Function *ctoa1* [Character TO A1] Convert a character value in the Software Tools [79] sense to Fortran "A1" format.

Function *dtoi* [Digit TO Integer] Return the integer value of the given digit character.

Function *itod* [Integer TO Digit] Return the character digit of the given (single digit) integer.

Function *lotoup* [LOwer TO UPper] Perform lower case to upper case conversion.

Function *uptolo* [UPper TO LOwer] Perform upper case to lower case conversion.

Moving Character Bytes

Function *byttoc* [BYTe TO Character] Return as a character variable the specified byte of the argument array. This routine is used to convert in-coming quoted character string arguments in subroutines to character variables. The number of bytes per integer word does not affect this routine in any way! In the Ratfor program:

```
call ss ('abc$')
end
subroutine ss (str)
integer byttoc
integer c
c = byttoc( str, 1 )
return
end
```

c should have the integer value for the character *a*, i.e., 97 for ASCII.

Function *ctobyt* [Character TO BYTe] Store the given character into the specified byte position of an array. This is the inverse of subroutine *byttoc*.

Bit Operations on Integers

Function *bwand* [Bit-Wise AND] Return the bit-wise *and* of two integers.

Function *bwnot* [Bit-Wise NOT] Return the bit-wise *not* (complement) of an integer.

Function *bwor* [Bit-Wise OR] Return the bit-wise *or* of two integers.

Function *gtbita* [GeT BIT from Array] Get value of specified bit of a given *bit-array*, returning 1 if bit is set, 0 if bit is not set.

Subroutine *stbita* [SeT BIT in Array] Set value of specified bit in a given *bit-array* to zero if argument is 0, one if argument is not 0.

Assigning and Reading Variables by Address

Function *adadri* [ADd ADdRess Integer] Add an integer to a machine address and return the new address. This primitive is required on certain computers where addresses are not "purely" increasing integers, i.e., they may have certain high order bits turned on which might even make the address a negative integer and incapable of being dealt with using Fortran integer arithmetic.

Function *getadr* [GET ADdRess] Return the machine address of the given argument with the stipulation that adjacent integers, i.e., in an array, must be one apart in address value. Thus on the VAX 11/780, for example, which has byte addressing, the byte address is divided by 4 to make it an integer boundary address that meets this stipulation.

Subroutine *setadr* [SET ADdRess] Set the integer at the given address to the value given.

Function *getvad* [GET Value at ADdress] Return the value of the integer at the given address.

Function *subadr* [SUBtract ADdResses] Subtract two given addresses returning the integer difference. See the discussion of *adadri* above for why this is needed.

File Opening, Closing, and Manipulation

Subroutine *cloze* Close the given logical unit number and return it to the stack of available unit numbers, originally set in a *data* statement in subroutine *openp*.

Function *filprm* [FILe Packed string ReMove] Remove the file with the given packed string filename. As the function value return OK if the file was removed ok, NOTTHERE if the file is not there or cannot be accessed, or ERROR if the file is there but cannot be removed.

File *openhdtl* — [OPEN HeaD TaiL] File which contains the *head* and *tail* strings which are appended before and after a filename that is surrounded by triangular brackets such as "*<grtdefs>*".

Function *openp* — [OPEN Packed] Open the file in the given packed string for reading or writing, returning a logical unit number or ERROR if it couldn't be opened. If the filename is surrounded by triangular brackets such as "*<grtdefs>*" then the brackets are removed and the *head* and *tail* strings from file *openhdtl* are put successively before and after the filename until the file is found. This opens the file to reside in "standard" places in the file system. Moreover, the standard places can be defined by a user in his own *openhdtl* file. For more information, see [110].

Function *opuniq* — [OPen UNIQue] Open a temporary file whose filename is unique from any others already opened with this function and that starts with the given character string, returning both the logical unit number as the function value and the actual filename used in another packed string argument.

Subroutine *stooff* — [STandard Output OFF] Turn off output from standard output logical unit number STANDOUT (defined in file *iodefs*), possibly by assigning it to a dummy file. This is used to stop normal output from a simulation program such as SPICE from coming to the terminal screen.

Subroutine *stoon* — [STandard Output ON] This is the inverse of *stooff*: turn back on output from STANDOUT by re-assigning it to the terminal.

Random Access File Input and Output

Subroutine *rdrec* — [ReaD RECord] Read from the given logical unit number, opened in direct access mode by *openp*, the specified record into an array.

Subroutine *wrrec* — [Write RECord] Write to the given logical unit number, opened in direct access mode by *openp*, the specified record number from the record in a given array. If the record number is given as APPENDTOEND, write the new record after the last record written, and return its record number in the same argument.

Standard System Calls

Function *exqtim* — [EXecution TIMe] Return as function value and in the real variable argument the current cpu time in seconds. The time returned can be either for the current executing program only or for the current user since he logged in (signed on) to the computer—it does not matter: only differences in the returned time are used.

Subroutine *gdatep* — [Get DATE Packed] Return in the argument a packed string containing the present date.

Function *getarg* — [GET ARGument] Get the next command line argument and return it in the character string argument. The function value returns the argument length or EOF when there are no more arguments.

Subroutine *gtimep* — [Get TIME Packed] Return in the argument a packed string containing the present time of day.

Subroutine *sdelay* — [Seconds DELay] Cause program execution to delay or pause for the specified number of seconds.

Subroutine *syscmd* — [SYStem CoMmanD] Execute an arbitrary operating system command contained in the packed string argument. This is called from a Fortran program and must return to the program right after the call.

Subroutine *syswap* — [SYstem SWAP] Swap to the operating system from an executing Fortran program, with the ability to return to Fortran execution right after some sort of operating system "exit" command is given.

Catching Hardware Traps

Subroutine *setint* — [SETup INTerrupt] Do whatever is necessary so that a certain common variable gets set to YES when a terminal user generates an interrupt by pressing the special interrupt key.

Subroutine *setovf* — [SETup OVerFlows] Do whatever is necessary so that a certain common variable gets set to YES when a floating point overflow or other numerical exception occurs.

Numerical Analysis Related

Function *randm* — Return a random number between 0.0 and 1.0 given the previous random number as argument.

Miscellaneous Primitives

Subroutine *aflush* [All FLUSH] Flush the DELIGHT internal arrays or stacks indicated by the coded argument variable. This routine most-likely does not contain any machine-dependencies.

Subroutine *pabort* [Print and ABORT] Print the given fatal error message and query the user as to the type of program abort desired. Then, abort or return to program execution.

Subroutine *qtstop* [QuieT STOP] Stop execution of a program without printing anything. This is necessary because sometimes a Fortran *stop* statement causes the word "stop" to print on the terminal and this is not wanted here.

APPENDIX D

DELIGHT: An Optimization-Based Computer-Aided Design System

Presented at the 1981 IEEE International Symposium on Circuits and Systems, Chicago, Illinois, April 1981.

DELIGHT: AN OPTIMIZATION-BASED COMPUTER-AIDED DESIGN SYSTEM

W. Nye, E. Polak, A. Sangiovanni-Vincentelli and A. Tits

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

This paper describes the design criteria and the main features of DELIGHT, a new interactive, optimization-based, computer-aided design system.

1. INTRODUCTION

Optimization pervades engineering system design: it is carried out over candidate configurations and over parameter values. Given a particular system configuration, optimization is used to determine parameter values which satisfy a set of specifications or optimize a performance function. Commonly, designers resort to heuristic, cut-and-try methods based on repeated system simulation. Unfortunately, such methods are woefully inadequate when the number of design parameters is large, the design specifications are complex, or the system is nonlinear. It then becomes necessary to utilize proper optimization algorithms for parameter computation and thus allow the designer to concentrate on the conceptual aspects of a design.

Despite considerable research activity on computer optimization of electronic circuits (see [1,2] for a review), optimization algorithms have not been used in design as widely as might be expected, mainly because (i) the best known simple algorithms were inadequate and designers lacked mathematical sophistication to use the more complex ones, and (ii) coupling optimization packages to simulation programs was difficult. For example, commonly used penalty-functions, with the variable metric method as a subroutine, were too primitive to solve design problems involving yield maximization or complex performance specifications, expressed as inequalities, which must be satisfied for all values of a parameter, such as frequency, temperature, or time ranging continuously over an interval. Similarly, since existing circuit simulators do not compute time domain sensitivities, they cannot be used efficiently in conjunction with most optimization algorithms.

New semi-infinite and nondifferentiable optimization algorithms solve problems involving yield maximization or a continuum of inequalities [3]. However, these algorithms are sensitive to the choice of internal parameters, to initial values of the design parameters and to the conditioning of the mathematical programming problem into which the design problem was transcribed.

Recently, a new design methodology based on interactive computing has emerged. It permits one to observe, interrupt, diagnose, modify and restart a computation as it progresses, resulting in very substantial savings not only in computing time, but also in the overall time needed to carry out a design. For example, the fact that an initial design cannot be adjusted to meet specifications can be identified in an interactive CAD system by observing the output. The designer may therefore stop the computation and either modify the structure of his design or experiment with relaxation of the specifications. Next, making use of the heuristic information displayed on the screen, he could reduce ill-conditioning by changing the description of the design problem into a different mathematical programming problem. Finally, he would be in an ideal situation to perform trade-offs. The circuit design system APLSTAP [4] developed at IBM and the optimization-based computer-aided design system INTEROPTDYN [5] developed at Berkeley, are two examples of such software systems. APLSTAP is intended mainly for circuit designers with little or no background in optimization, INTEROPTDYN is for sophisticated designers with an optimization background.

The DELIGHT system represents a considerable advance over these earlier systems in terms of flexibility and ease of use in a variety of contexts. Thus, it serves both the sophisticated and the unsophisticated designer. In addition, it provides a "guru" with an ideal environment for developing and testing new optimization algorithms and for extending the capabilities of the system software.

2. THE DELIGHT SYSTEM

The DELIGHT system was conceived for multi-disciplinary as well as multipurpose use and aims to provide a congenial, efficient and portable environment for the following potential users:

a. An unsophisticated designer requiring only command and algorithm execution.
b. An advanced designer who wishes to adjust his optimization algorithms, for example, by modifying algorithm parameters or by substituting new stepsize or direction-finding subprocedures for the ones he finds unsatisfactory.
c. An optimization algorithm expert who requires his computer programs

Reprinted from 1981 IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, April 1981

CH1635-2/81/0000-0851 $00.75 © 1981 IEEE

to resemble as much as possible the mathematical description of the algorithm he is implementing or creating so as to minimize the effort involved in testing alternatives.

d. A systems expert who needs to add new built-in functions, utilities, or other system features.

Since it is impossible to foresee all the features a CAD system will eventually need, we made sure that extensions and modifications of the system will be easy to carry out. We shall now describe the DELIGHT system with some detail.

2.1 RATTLE The Interactive Language.

Our system uses the interactive programming language RATTLE (an acronym for RATfor Terminal Language Environment) which we have evolved from the structured language RATFOR. The similarity to RATFOR allows DELIGHT system users to learn RATTLE easily. RATTLE encourages good programming practice by providing structured constructs such as "while", "repeat-until", "if-then-else", etc., and hence results in highly readable programs. RATTLE executes rapidly because it is compiled into an intermediate form, with only one pass over the source code, it is not interpreted.

RATTLE has powerful extension capabilities, that is, the ability to create new language constructs or new commands from existing ones. This facilitates its use in many different design environments. In particular, RATTLE has defines, similar to those in RATFOR but with several extensions, as well as a new, powerful feature, the macro. Macros are written as ordinary RATTLE procedures. However, they are not executed at run time, but rather when their name is encountered during the compilation of other procedures. They can act as filters in the stream of input characters being received by the RATTLE compiler. For example, one can write a macro to scan the next few tokens, which need not be valid RATTLE code (they are never parsed by the compiler), make decisions based on what is found, and then send valid RATTLE code to the compiler. This is accomplished using the push-back stack mechanism of [7]. We have used macros to enable us to carry out very complex computations by means of very simple commands. For example, using the macro 'lp', we can solve a linear program with the following RATTLE code

```
lp x = argmin { c*x | x>=0, x<=d, A*x<=b }
```

where the array x is assigned the minimizing value of x. The macro lp scans ahead, determines what is being requested, and pushes back onto the push-back stack a normal RATTLE procedure call. In addition, the macro creates all the necessary work arrays and inputs for the call to a built-in Harwell Library linear programming FORTRAN routine. Thus, macros relieve the programmer of such burdens as the creation of work arrays for library routines and the use of complicated language syntax, as well as provide enhanced readability. Presently, there is an arsenal of numerical analysis software available to the user through macros. The following is a sample of these macros:

Computation	Macro Syntax
eigenvalues	matop lambda = eigen(A)
inverse of matrix	matop Ainv = inv(A)
solve linear eqns.	lineq A*x = b
quadratic program	qp z = argmin { x'*Q*x + b*x | x>=d, A*x<=c }
l2-norm of vector	||v|| (in any
inner product	<<x,y>> expression)

Most of the above macros use LINPACK routines [11].

One important use of defines is in the creation of user oriented commands for invoking RATTLE procedures for complex graphics. These procedure use high and low level, terminal independent graphical routines which are incorporated in the system. For example, one can define a "window" by name, so that the command "window name" is a substitute for specifying the particular set of world coordinates [8], and corresponding viewport coordinates (in the (0,0)-(1,1) coordinate system of the terminal screen), which are associated with the window.

The RATTLE language supports incremental program development [6], that is, the ability to test, by just typing it in, a single statement, procedure, or section of an algorithm, without having to write and load/link a whole program. The following is a complete RATTLE statement which would execute when typed in:

```
while ( f(x) > eps ) {
   x = x - f(x) / df(x)
   print x
   }
```

In this example, the while-loop body consists of two statements; the closing '}' is needed before starting execution.

An important RATTLE feature, from the programmer's or algorithm developer's point of view, is the fact that execution can be interrupted by the user or by the program and later resumed after modifying variable values, or even re-compiling an entire sub-procedure.

2.2. FORTRAN Functions and Routines and the RATTLE Algorithmic Library.

The built-in FORTRAN functions and routines fall into the following categories:

a. Standard FORTRAN functions such as

sin, cos, exp, log, etc.

b. General purpose numerical analysis software such as that found in LINPACK [11] or the Harwell Subroutine Library [12].

The algorithmic library consists of an integrated set of RATTLE routines implementing algorithms for unconstrained and constrained, both ordinary and semi-infinite, optimization problems. This library is organized to exploit to the utmost the natural modularity of modern optimization algorithms which, in the simplest case, can be assembled from such blocks as search-direction, step-size and update subalgorithms. In turn, search-direction subalgorithms can be constructed from subprocedures which determine the gradients to be used for direction construction and from linear or quadratic programs. Similarly, step-size subalgorithms can be built up from constrained and unconstrained step-size blocks. More complex blocks include outer approximations subprocedures and adaptive parameter adjustment subprocedures. For example, to construct an unconstrained optimization algorithm, one may combine a search-direction obtained through a quasi-Newton update formula, or through a conjugate gradient scheme, with a step-size rule based on cubic interpolation, or the golden section rule. The features of RATTLE make the use of such a modular library extremely easy and effective, so that a large number of algorithms can be generated from a relatively small number of procedures. It is easiest to explain how this modularity is used by means of a simple example.

A large class of unconstrained minimization algorithms have a structure which is incorporated in the RATTLE procedure ucmin below:

```
#===============
# ucmin
#===============
procedure ucmin  {
   repeat  {
      interaction
      evaluate h = dir(X[Iter])
      lambda = step (X[Iter], h)
      update X[Iter+1] = X[Iter] + lambda * h
      Iter = Iter + 1
      }
   forever
   }
```

This procedure calls two subprocedures: dir (search-direction computation) and step (step-size computation). In order to construct a particular unconstrained minimization algorithm, one creates a file containing instructions combining a file containing ucmin with files containing the appropriate search-direction and step-size subprocedures (or functions), as in the following program for the armijo gradient method [14]:

```
#==========
# armgrad
#==========
include step.arm
include dir.gradient
include ucmin
```

The files step.arm and dir.gradient contain the following specific subprocedures step and dir:

```
#===========
# step.arm
#===========
parameter Alpha = .5
parameter Beta = .5
function step (x, h)  {
   import Alpha, Beta
   array x(), h()
   evaluate gcost = gradcost (x)
   k = 0
   repeat {
      update xnew = x + Beta**k * h
      breakpt
      del = cost(xnew) - cost(x)
      if (del <= Alpha*Beta**k * <<h, gcost>>)
         return Beta**k
      k = k + 1
      }
   forever
   }
```

```
#===============
# dir.gradient
#===============
procedure dir (x, h)  {
   array x(), h()
   evaluate h = gradcost (x)
   matop h = (-1) * h
   }
```

The above code is mostly self-explanatory However, the following features are worth pointing out First, we note that in procedure ucmin, the vector X[k] is an element of a sequence which has been declared using "array_sequence" (for an example see Section 3). Since part of this sequence is frequently needed for display or analysis purposes, the user can save its last n elements. Second, the concept of imported variable, as exemplified by Alpha and Beta in the function step, is borrowed from [13]. Alpha and Beta, declared outside the procedure, are given a default value at compile time. This allows the user to modify them before starting execution; their value is known to the function step at run time. Since different problems require different values of Alpha and Beta for efficient solution, it is crucial to be able to modify them (or other algorithm parameters). Third, RATTLE permits interruption of a process and resumption of execution after checks or modifications have been carried out. The user relies on information displayed at each iteration, preferably in graphical

form, in deciding when an optimization computation should be interrupted. The define "interaction" enables a user supplied procedure "output", to be executed at every passage through "interaction". When the "break" key is depressed, execution is suspended right after the procedure "output" has been executed (depressing the "break" key generates an interrupt).

Besides using the "break" key, the user can control execution of an optimization process through the define "run". Typing "run 5", causes 5 iterations of the process to be carried out; computation stops at the "interaction" point. Typing simply "run" causes the process to execute until the "break" key is depressed.

2.3 Interfaces to Problems and Simulation Routines.

The DELIGHT system includes an algorithm-problem interface which simplifies the coupling of RATTLE algorithmic library routines with design problems. An important feature of this interface is the way in which design problems are formulated. In a design, the cost and many constraint functions are frequently composites, consisting of simple functions which are evaluated on results of simulation, for example, the overshoot constraint $y(t,x) < 1.1$ for all $t > 0$, where x is the design parameter and $y(t,x)$ is the corresponding step response. Consequently, in DELIGHT, the formulation of a design problem "prob" consists of a set of files whose names are . "prob.descr" (simulation structure and design parameters of the problem), "prob.data" (initial values of design parameters), "prob.cost" (cost function), "prob.gradcost" (gradient of the cost), and, possibly, "prob.hesscost" (Hessian of the cost) together with corresponding files for the various types of constraints (equality, inequality and functional inequality). When an algorithm needs a "surrogate" cost (as in the case of augmented Lagrangian methods or exact penalty function methods), the interface constructs it automatically. In addition, the interface makes sure that unnecessary duplication of evaluations is avoided. As we have seen in the preceeding section, in the DELIGHT system, a program for an optimization algorithm is a file containing a list of all the algorithmic procedures to be used and of the information needed (gradients, Hessians). The coupling of a problem with an algorithm is carried out by means of the define "solve" which checks to make sure that the problem and algorithm are compatible (e.g., a constrained problem cannot be solved using an unconstrained optimization algorithm). For example, "solve prob using armgrad" couples the problem "prob" with the algorithm "armgrad".

The second interface facilitates the use of a variety of simulation programs for engineering design (e.g., of electronic circuits, control systems, structures). Usually, a simulation program has input parameters, options, outputs, as well as simulation run controls, all of which can be chosen by the designer. An optimization algorithm program calls RATTLE function and gradient evaluation procedures which, in turn, may have to call a simulation program and hence must be able to set input parameters and retrieve output values. To allow the required choices to be made interactively by the designer and automatically by optimization algorithms, it is necessary to have an interface to the simulation program. Our interface is readily usable by any RATTLE procedure and consists of two parts: one part which is written in RATTLE and is used for all simulation programs and a second one which is written for each simulation program. For example, the simulation dependent interface routine "output_keywords" allows a RATTLE translation macro to be independent of any special syntax for specifying simulation outputs. In circuit design, suppose that SPICE [10] is to be used as the simulation program. The system first calls the routine "output_keywords" which returns the special node voltage keywords of SPICE, "vm", "vp", "vr", etc.. It then creates a define for each output keyword, which invokes the translation macro to gather the keyword and its arguments, considered to be the source tokens following the key word which are enclosed in a set of balanced parentheses. This string, for example, "vm(3,55)", is then passed to another interface routine so that it can parse the arguments in any way needed.

3. EXAMPLE

We shall now present an elementary illustration of the use of the DELIGHT system in solving a very simple unconstrained optimization problem. The specification of this problem, pb1, is contained in the following files, which are in the format discussed in Section 2.3. Note that the file pb1.descr contains only the specification that 25, 2-component past design parameter values be stored.

```
#============
# pb1.descr
#============
array_sequence X[25](2)
#==========
# pb1.data
#==========
X[0](1) = 1
X[0](2) = 1
#==========
# pb1.cost
#==========
function cost (x)  {
   array x(2)
   return  (x(1)**2 + 2*x(2)**2)
   }
#==============
```

```
# pb1.gradcost.
#==============
procedure gradcost (x,g) {
   array x(2), g(2)
   g(1) = 2 * x(1)
   g(2) = 4 * x(2)
   }
```

We shall show what appears on the screen when a designer solves this problem by means of the Armijo gradient method discussed in Section 2.2. We have underlined the user input to distinguish it from computer output. The basic RATTLE prompt is "1>", while "2>" indicates that the process has been interrupted once. In this example, the user first specifies the design problem and the algorithm to be used for its solution by means of the "solve" define. Then (i) he assigns to Alpha the value .9; (ii) he includes the procedure "output" contained in the file "printstate", and (iii) requests that the process run for 2 iterations. Unhappy with the way the computation is progressing, he changes the parameter Alpha to .6 and resumes execution. When he is satisfied with the values displayed, he stops the process by depressing the "break" key.

```
          1> solve pb1 using armgrad
array_sequence X[25](2)
data: X[0](1) =  1.000
data: X[0](2) =  1.000
parameter:  Alpha = .5
parameter:  Beta = .5
please specify an output action
type run to execute
          1> Alpha = .9
          1> include printstate
          1> run 2
Iter=0  cost=3.000      ||gradcost||=4.472
Iter=1  cost=2.410      ||gradcost||=3.971
Iter=2  cost=1.945      ||gradcost||=3.531
          Interrupt...
          2> Alpha = .6
          2> run
Iter=3  cost=.6063      ||gradcost||=1.823
Iter=4  cost=9.548e-2   ||gradcost||=.6180
Iter=5  cost=2.387e-2   ||gradcost||=.3090
    (here, the user depresses "break")
          Interrupt...
          2> reset
          1>
```

4. CONCLUSION

We have briefly described the design criteria, the structure and the main features of the optimization-based computer-aided design system DELIGHT. Two important features were not discussed in in the paper, but should nevertheless be mentioned. The first is that the system incorporates an editor (a subset of the UNIX editor). The second feature is a "store-restore" command permitting to store a computation in its full state, from which it can be restarted at later time. To evaluate DELIGHT we have tested it in the solution of a few complex design problems such as the design of a digital filter and the design of control systems subject to constraints on singular values over a range of frequencies. At present, simulation programs for structural design of braced frames under seismic loading are being interfaced. Integrated circuit design will be attempted as soon as the time-domain sensitivity computation is implemented in SPICE and the simulation interfaces discussed are completed.

ACKNOWLEDGEMENTS

This research was supported by the National Science Foundation Grants ENV-7[illegible]-04264 and ECS-79-13148, by AFOSR Contract No. F49620-79-C-0178, and by a grant from Harris Semiconductors.

REFERENCES

[1] R.K. Brayton and R. Spence, Sensitivity and Optimization, Elsevier, 1980

[2] J.W. Bandler and M.R. Rizk, "Optimization of Electrical Circuits", Mathematical Programming Study, vol. 11, pp 1-64, 1979.

[3] E. Polak, "Algorithms for a Class of Computer-Aided Design Problems: A Review", Automatica, vol. 15, pp. 795-813, Sept. 1979.

[4] G.D. Hachtel, T.R. Scott and R.P. Zug, "An Interactive Linear Programming Approach to Model Parameter Fitting and Worst-Case Circuit Design", IEEE Trans. on Circuits and Systems, vol. 27, pp. 871-882, Oct. 1980.

[5] M.A. Bhatti, T. Essebo, W. Nye, K.S. Pister, E. Polak, A.L. Sangiovanni-Vincentelli and A.L. Tits, "A Software System for Optimization-Based Interactive Computer-Aided Design", Memorandum N. UCB/ERL M80/14, University of California, Berkeley, April 1980.

[6] J. Wilander, "An Interactive Programming System For Pascal", BIT vol. 20, n. 2, pp. 163-174, 1980.

[7] B.W. Kernighan, P. Plauger, Software Tools, Addison-Wesley, Mass., 1976.

[8] W.M. Newman, R.F. Sproul, Principles of Interactive Computer Graphics, 2'nd Edition, Mcgraw-Hill, N.Y., 1979.

[9] W.T. Nye, RATTLE/DELIGHT Programming Manual, University of California, Berkeley, 1980.

[10] L.W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits", ERL memo no. ERL-M520, University of California, Berkeley, May, 1975.

[11] J.J. Dongarra, et.al., LINPACK Users' Guide, SIAM, Philadelphia, Pa., 1979.

[12] Harwell Subroutine Library, Harwell, England.

[13] N. Wirth, "Modula -- A Language for Modular Multiprogramming", Software - Practice and Experience, vol. 7, 1977.

[14] L. Armijo, "Minimization of Functions Having Continuous Partial Derivatives", Pacific J. Math., vol. 16,

APPENDIX E

The Design of Digital Filters Using Interactive Optimization

Presented at The 20'th IEEE Conference on Decision and Control, San Diego, California, December 1981.

THE DESIGN OF DIGITAL FILTERS USING INTERACTIVE OPTIMIZATION

T.P. Lee, W.T. Nye and A.L. Tits

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

Summary

It is shown how modern optimization techniques can be used to design digital filters. The high flexibility of this approach makes tractable a large class of specifications, such as special stability requirements. Efficient use of optimization requires a highly interactive environment, such as the Berkeley DELIGHT System. As an example, the design of a low-pass filter is considered.

Introduction

The design of digital filters consists of the approximation of desired magnitude and phase requirements with a suitable ratio of polynomials corresponding to a stable transfer function, and the realization of this transfer function with a filter structure which may be implemented in hardware or software. Traditionally, designers use classical design techniques to approximate their desired filter responses. These are usually indirect, in the sense that to design a general infinite impulse response digital filter one designs an analog filter transfer function using classical techniques and then converts it into a digital transfer function. This conversion can be done by any one of the most commonly used methods such as impulse response invariant, matched, or bilinear z-transformation [1]. These design techniques have been automated in the filter design program FILSYN [2].

Here, we take the direct approach of transcribing the design problem into a mathematical programming problem. This allows one to specify practical constraints, for example, on the polynomial degree, on the magnitude of the coefficients, or even for special stability requirements. These features may not be possible with the classical design approach. The mathematical programming problem can be solved using recent semi-infinite optimization algorithms [3]. Since these algorithms are sometimes sensitive to the choice of internal parameters, to initial values of the design parameters, or to the conditioning of the mathematical programming problem, our design methodology requires a highly interactive computing environment. Such an environment is available to us in the computer-aided design system DELIGHT [4]. This system permits the designer to interrupt, observe, diagnose, modify, and restart a computation as it progresses, resulting in not only savings in computer time, but also in the overall time needed to carry out the filter design.

In the DELIGHT system, all algorithms and problem formulations are coded in the interactive structured programming language RATTLE, described in [4,5]. An important RATTLE feature is that execution can be interrupted and resumed even after modifying the mathematical problem formulation by recompiling any of the cost or constraint procedures. There is an algorithmic library consisting of an integrated set of RATTLE procedures implementing algorithms for unconstrained and constrained, both ordinary and semi-infinite, optimization problems. The semi-infinite constrained algorithm proposed by Gonzaga, Polak and Trahan [6] is particularly suited to our design problem.

Example: design of a low-pass filter

Our mathematical programming formulation includes the following functional (or semi-infinite) constraints ($\omega \in [0,\pi]$ is the frequency):

amplitude constraints (fig. 1):

$$1-\epsilon_p d \le \text{amplitude}(\omega) \le 1+\epsilon_p d \qquad \forall \omega \in [0,\omega_p]$$
$$0 \le \text{amplitude}(\omega) \le 1+\epsilon_p d \qquad \forall \omega \in (\omega_p,\omega_a)$$
$$0 \le \text{amplitude}(\omega) \le \epsilon_a d \qquad \forall \omega \in [\omega_a,1.]$$

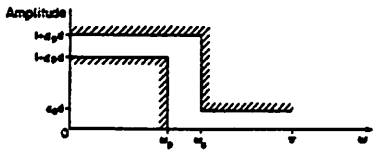


Figure 1
Amplitude Constraints

* This research was supported by National Science Foundation grants No. ECS-79-13148 and PFR-79-08261 (RANN) and the Air Force Office of Scientific Research (AFSC) United States Air Force under Contract No. F44620-76-C-0100, and by a grant from the Semiconductor Products Division of the Harris Corporation.

Reprinted from THE 20th IEEE CONFERENCE ON DECISION AND CONTROL, December 1981

0191-2216/81/0000-0819 $00.75 © 1981 IEEE

phase constraints (fig. 2):

$$(m-\varepsilon_\phi d)*\omega \leq \text{phase}(\omega) \leq (m+\varepsilon_\phi d)*\omega \qquad \forall\omega \in [0,\omega_p]$$

Figure 2
Phase Constraints (unspecified average slope)

where m, the slope of the phase function, is unspecified and d, a measure of the allowable deviation in both magnitude and phase from ideal responses, is to be chosen as small as possible. After having selected the degree of the filter, we express both numerator and denominator of the transfer function as products of quadratics (with a linear factor if the degree is odd) of the form

$$z^2 + bz + c$$

This formulation, besides leading to a filter structure with low quantization noise [7], simplifies the expression of conditions for stability, which can be written as the following constraints on the coefficients of the denominator quadratics (see, e.g., Arapostathis and Jury [8]):

$$1 + b + c > 0,\ 1 - b + c > 0,\ 1 - c > 0$$

Stronger stability requirements (roots inside circle of radius $\rho < 1$) can be expressed as

$$1 + \frac{b}{\rho} + \frac{c}{\rho^2} \geq 0,\ 1 - \frac{b}{\rho} + \frac{c}{\rho^2} \geq 0,\ 1 - \frac{c}{\rho^2} \geq 0$$

These are ordinary inequality constraints.

As implicitly suggested, the design parameter vector X has as components

1. the coefficients of the numerator and denominator quadratics and an overall multiplicative factor,
2. m, the unspecified slope of the desired phase response, and
3. d, the measure of the allowable deviations of magnitude and phase from the ideal response, which is desired to be as small as possible.

Hence, the cost function to minimize is simply $f(X) = d$.

Interacting with the optimization process

During each optimization iteration, various parameters and graphical output are displayed on the terminal screen. Plots include filter amplitude and phase response together with the current constraint bounds (which depend on X through d) as well as the position of the poles of the denominator polynomial. With this information, d and m, for example, can be adapted interactively to speed up the computation. Information is also displayed concerning the internal behavior of the algorithm. If one notices poor computational progress, one can interrupt execution, use the information displayed to modify algorithm parameters or choose a different algorithm subprocedure, and then resume execution without any loss of the optimization progress already achieved.

Conclusion

Although a shortage of space prevents us from showing numerical results for the low-pass filter example, the mathematical programming formulation and optimization algorithm mentioned above were quite successful in achieving good filter designs. We even took designs from FILSYN[2], which does not allow phase constraints for our particular type of filter, and used then as initial guesses to our optimization with phase constraints. We obtained significant improvements in the phase responses and we could also achieve stronger stability properties.

Throughout this work, the interactive nature of the optimization was indispensable. Several times, observation of the progress led to reformulation of the problem and to modification of some critical algorithm parameters. For example, we modified an algorithm parameter when we noticed that the iterates we bouncing on and off certain constraint boundaries.

Acknowledgements

The authors thank P. Dodd and Y. Wardi for their invaluable help and Professors E.I. Jury and J.C. Walrand and Dr. B. Gopinath for fruitful discussions.

References

[1] L.R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice Hall, N.J., 1975.
[2] G. Szentirmai, "FILSYN - A General Purpose Filter Synthesis Program," Proc. IEEE, vol. 65, pp. 1443-1458, Oct. 1977.
[3] E. Polak, "Algorithms for a Class of Computer-Aided Design Problems: A Review," Automatica, vol. 15, pp. 795-813, Sept. 1979.
[4] W.T. Nye, E. Polak, A.L. Sangiovanni-Vincentelli and A.L. Tits, "DELIGHT: An Optimization-Based Computer-Aided Design System," Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Chicago, Illinois, 1981, pp. 851-855.
[5] W.T. Nye. RATTLE/DELIGHT User's Manual, University of California, Berkeley, 1980.
[6] C. Gonzaga, E. Polak and R. Trahan, "An improved Algorithm for Optimization Problems with Functional Inequality Constraints," IEEE Transactions on Automatic Control, vol. 25, pp. 49-54, 1980.
[7] J. Szczupak and S.K. Mitra, "Recursive Digital Filters with Low Roundoff Noise," Circuit Theory and Applications, vol. 5, pp. 275-286, 1977.
[8] A. Arapostathis and E.I. Jury, "Remarks on Redundancies in Stability Criteria and a Counterexample to Fuller's Conjecture," Int. J. on Control, vol. 29, no. 6, pp. 1027-1034, 1979.

APPENDIX F

An Enhanced Methodology for Interactive Optimal Design

Presented at the 1983 IEEE International Symposium on Circuits and Systems, Newport Beach, California, May 1983.

An Enhanced Methodology for Interactive Optimal Design

W.T. Mye
University of California, Berkeley

A.L. Tits
University of Maryland, College Park

Introduction.

Optimization techniques have been applied successfully to numerous design problems in various branches of engineering. (For an excellent survey in the area of integrated circuit design, see [1].) However, in many cases, the mathematical problem solved by the optimization algorithm may be remote from the real world problem the designer is facing. This is due to the rigidity of the classical nonlinear programming problem which can be stated as

$$\min\{ f(x) \mid g(x) \leq 0 \} \qquad (*)$$

where $f(x)$ is a cost or objective function to be minimized and $g(x)$ represents several inequality constraints and where x is the vector of design parameters. While this formulation does encompass the general idea of optimizing some design objective while meeting various design specifications, it fails to take into account several important characteristics of a large class of design problems.

First, it is rarely the case that a single objective has to be optimized. In most applications, various objectives compete against each other and a compromise has to be reached. Amalgaming several objectives into a single cost function has the disadvantage, particularly acute in an interactive environment, of hiding the physical significance of these objectives.

Second, the above mathematical formulation (*) does not accept any violations of the constraints g. In design applications, constraints specifications are often relatively flexible and thus better put in words than in numbers. Hence, moderate violation of a constraint should be acceptable by the optimization algorithm: often, this will permit it to achieve a better value of the objective function(s). Notice that the formulation

$$g(x) \leq 0$$

is particularly inadequate since it gives no way of estimating the importance of a given constraint violation.

Third and more generally, formulation (*) expresses only partially the knowledge a designer has about his problem. On the one hand, some of this knowledge, built on experience and physical intuition, is often impossible to express numerically. Also, corresponding specifications may appear to be necessary only after (*) has been solved and has yielded an inadequate design. On the other hand, (*) does not allow the designer to express some of his intuitive knowledge such as the degree of confidence he has in the initial guess provided for each design parameter.

Obviously, there is no unique way to get around some of the difficulties mentioned above. The methodology described in this paper makes use of some ideas and concepts which seem particularly well suited to the designer's intuition. It has been implemented on the DELIGHT [3] interactive system and applied successfully to the design of "real world" integrated circuits by "real life" designers.

A new methodology.

The various design specifications are first put into one of the following categories:

1. An objective is a quantity that the designer would like to see as small (large) as possible; example: the gain of an operational amplifier.
2. A soft constraint is a quantity that the designer would like to see smaller (larger) than some threshold, or, if this cannot be achieved, as close as possible to this threshold; example: the stability margin of a control system
3. A hard constraint is a quantity that the designer requires to be below (above) some threshold, any violation being unacceptable; example: a resistance value must be nonnegative.

Obviously the various specifications or, at least, the objectives and soft constraints are competing against each other. It is then necessary to be able to meaningfully compare the values of various specifications for a given x-vector value, i.e., to define the normalized value of a specification. This is done through the introduction of the concepts of *good* and *bad* values of the various specifications. These values must be chosen according to the following rule: having all the objective and soft constraints achieve their respective good values should provide the same level of satisfaction to the designer for each, while achieving the bad values should provide the same level of dissatisfaction. Furthermore, the good value of a soft constraint must be the threshold value aimed at. Technically, these good and bad values are then used to produce a normalized specification with value between 0 and 1, where 0 and 1 correspond respectively to the designer's good and bad values. A minimax related optimization, as described in the next section, is then performed on those normalized values. The use of two values to perform our multiobjective normalization may be viewed as an extension of Lightner and Director's [2] suggested use of the reciprocal of user supplied desired objective values in a weighted max norm solution.

CH1845-7/83/0000-1050 $1.00 © 1983 IEEE

In engineering design, many specifications are best represented as *functional specifications*, i.e., specifications on a complete response curve (e.g., a frequency or time response). These specifications are represented as functional objectives and functional soft constraints for which *good* and *bad* curves must be defined by the designer.

It must be stressed here that interaction between the designer and the optimization process as well as human engineered graphics to support this interaction are crucial for an efficient use of the methodology described here. Suitable information must be displayed to allow the designer to interactively modify good and bad values or curves. Such a man-machine interface has been implemented in the DELIGHT system. An important part of the graphics interface is the performance comb.

The *performance comb* is a graphical display which shows how close each objective and soft constraint is to its good and bad values. The display consists of a vertical *good* line on the left and a vertical *bad* line on the right. Each objective or soft constraint is displayed as a horizontal bar or *tooth* of the comb. The goal of the optimization algorithm is to move the tips of all the comb teeth to the left. This performance comb, displayed whenever needed, allows the designer to grasp quickly the current performance of his design.

An optimization algorithm.

We propose an algorithm from the feasible directions family [4] for performing the design outlined in the previous section (it also efficiently handles soft and hard *box* constraints, i.e., bounds on the design parameters). This algorithm consists of three consecutive phases.

In phase 1, the worst hard constraint violation is progressively decreased until all the hard constraints are satisfied. In phase 2, objective values and soft constraint values are simultaneously improved (through a minimax optimization of the normalized specifications), while hard constraints are forced to remain satisfied. If phase 2 succeeds, in the sense that all the objectives and soft constraints reach their target *good* values, phase 3 is entered. In phase 3, the objectives are further improved while keeping both hard and soft constraints satisfied.

The proposed algorithm is convergent in the sense that any accumulation point of the sequence of x-vectors it constructs satisfies an optimality condition. Its speed of convergence is improved by properly scaling the design parameters, using an additional piece of information provided by the designer. Based on his intuitive knowledge of the design problem, he provides the *nominal variation* of each design parameter, which can be thought of as the distance to the next parameter value he would try if optimizing the parameter value manually.

References

[1] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "A Survey of Optimization Techniques for Integrated Circuit Design," *Proc. IEEE* 69(10) pp. 1334-1362 (1981).

[2] M. R. Lightner and S. W. Director, "Multiple Criterion Optimization for the Design of Electronic Circuits," *IEEE Trans. on Circuits and Systems* CAS-28(3) pp. 169-179 (March 1981).

[3] W. T. Nye, E. Polak, A. Sangiovanni-Vincentelli, and A. L. Tits, "DELIGHT: An Optimization-Based Computer-Aided Design System," *Proc. of IEEE International Symposium on Circuits and Systems*, (April 1981).

[4] E. Polak, "Algorithms for a Class of Computer-Aided Design Problems: A Review," *Automatica*, 15 pp. 795-813 (September 1979).

Acknowledgements

This work was supported by the National Science Foundation under grants ECS-82-04462, CEE-81-06790, and ECS-79-13148, by the Air Force Office of Scientific Research (AFOSR) United States Air Force Contract No. F49620-79-C-0178, and by a grant from the Semiconductor Products Division of the Harris Corporation.

APPENDIX G

DELIGHT.SPICE: An Optimization-Based System for Design of Integrated Circuits

Presented at the 1983 Custom Integrated Circuits Conference, Rochester, N.Y., May, 1983.

DELIGHT.SPICE: AN OPTIMIZATION-BASED SYSTEM FOR THE DESIGN OF INTEGRATED CIRCUITS

Bill Nye
Alberto Sangiovanni-Vincentelli
Department of EECS, University of California Berkeley, Ca.

James Spoto
Harris Semiconductors, Melbourne, Fla.

Andre Tits
Electrical Engineering Department, University of Maryland, College Park, Ma.

Abstract

DELIGHT.SPICE is the union of the DELIGHT interactive optimization-based computer-aided-design system and the SPICE circuit analysis program, both developed at the University of California, Berkeley. With the DELIGHT.SPICE tool, circuit designers can take advantage of recent powerful optimization algorithms to automatically adjust parameters of electronic circuits in order to improve their performance. They may optimize arbitrary performance criteria as well as study complex tradeoffs between multiple competing objectives, while simultaneously satisfying multiple constraint specifications. Industrial analog and digital circuits have been redesigned using this tool yielding substantial improvement in circuit performance.

1. INTRODUCTION

For our purposes, circuit design can be considered as a two-phase iterative process. The designer first selects an initial circuit configuration and then determines values of circuit parameters (e.g. resistor and capacitor values, and device geometries such as bipolar transistor areas and MOSFET lengths and widths) that satisfy a set of specifications and optimize a set of possibly competing design objectives. This process is repeated until a satisfactory design has been achieved. The most creative part of the design process is, in general, the selection of the circuit topology. For large nonlinear circuits, the selection of values of the design parameters is often time consuming, and is usually stopped short of even a local optimum of the design objectives. This is due to the fact that the specifications and objectives may depend on AC, DC, and transient responses, which in turn depend on many design parameters. Thus, it is usually difficult for designers to predict the effect of parameter changes on circuit performance without numerous circuit simulations.

DELIGHT.SPICE is the result of the union of the DELIGHT interactive optimization-based computer-aided design system [1] and the SPICE circuit analysis program [2] both developed at the University of California, Berkeley. Optimization performed by DELIGHT.SPICE, intends to free the designer from the difficult task of selecting values of the design parameters thus making it possible for him to concentrate on more creative aspects of the design process.

The idea of using optimization to design circuits dates back to the late sixties (e.g. [3]). However, there has been little use of parametric optimization in the circuit designer community. This has probably been due to the often primitive optimization algorithms used, to the lack of adequate interaction in the software for circuit optimization, and to inadequate computing facilities [4]. DELIGHT.SPICE uses new powerful optimization algorithms and is heavily based on interaction. It is hoped that it will find a wide user community.

As pointed out previously, circuit performance and specifications are, in general, functions of the design parameters through circuit responses. For example, the design of a wide-band amplifier could have as a *performance objective* (i.e., improve it as much as possible) the bandwidth of the amplifier and as a *constraint specification* (i.e., just meet the spec) that the DC power be less than some value. The design parameters could be a capacitor value and a BJT area. Neither the performance objective nor the constraint specification are explicit functions of the design parameters; this dependence is implicit through analyses of the circuit equations. In particular, the bandwidth can be evaluated by finding the -3dB point of the frequency response from an AC analysis, while the DC power can be computed as the product of the DC current through the power supply times the supply voltage. This DC current would be computed by a DC analysis. Thus the performance and specification evaluations required to perform optimization of a circuit often involve expensive circuit simulations. The DELIGHT.SPICE system computes these circuit responses using the simulation program SPICE.

The organization of this paper is as follows. Section 2 provides a brief overview of the problem formulation and of the basic optimization algorithm of DELIGHT.SPICE. Section 3 presents a detailed example of the design of a wide-band operational amplifier carried out with the help of DELIGHT.SPICE. The concluding remarks are given in Section 4.

2. PROBLEM FORMULATION AND THE BASIC ALGORITHM

2.1 Standard Formulation

The use of DELIGHT.SPICE for optimization requires the formulation of the design problem as a certain standard mathematical programming problem. Fortunately, formulating circuit design problems in this way is almost always possible.

The simplest nonlinear mathematical programming problem is the *unconstrained* nonlinear programming problem:

$$\underset{X}{\text{minimize:}}\ M1(X)$$

in which the minimum (or maximum) value of some scalar function of the design parameters viewed as elements of a vector X is sought. In engineering design, however, it is much more likely to have additional inequality constraints that must be met. An example is that the power dissipated in a circuit be less than 1.5 watts. We then have the *constrained* nonlinear programming problem:

$$\underset{X}{\text{minimize:}}\ M1(X)$$

such that:

$$\begin{aligned} I1(X) &\le \text{SpecValue1} \\ I2(X) &\le \text{SpecValue2} \\ &\cdots \\ In(X) &\le \text{SpecValuen} \end{aligned}$$

where I1, I2, $\cdots$, In are, in general, nonlinear vector-valued functions of the design parameter vector X, and SpecValue1, $\cdots$, SpecValuen are scalars representing the limits on the specifications that the circuit must satisfy.

2.2 DELIGHT.SPICE Design Problem Formulation

Unfortunately, some meaningful integrated circuit design problems cannot be formulated as the standard mathematical programming problem given above. For example, many commonly occurring constraints require that some specification be met *over a range of* an *independent parameter*, such as time, temperature, frequency, or even the voltage of an independent voltage source. These constraints are called *functional* inequality constraints and must be handled in a special way by optimization algorithms [7,8]. An example of a functional constraint is "Maintain the common

mode rejection ratio within prespecified limits for every frequency in the interval 100kHz to 10megHz". DELIGHT.SPICE allows designers to specify these functional constraints in addition to the ordinary constraints and objective function introduced above. By adding functional inequality constraints to the problem formulation we arrive at the following *semi-infinite*[1] nonlinear programming problem:

$$\underset{X}{\text{minimize:}}\ M1(X)$$

such that:

$$\begin{aligned} &I1(X) && \leq \text{SpecValue1} \\ &\cdots \\ &In(X) && \leq \text{SpecValuen} \end{aligned}$$

and

$$\begin{aligned} &FI1(X,W1) && \leq \text{SpecCurve}(W1) && \forall W1 \in [W1_0, W1_d] \\ &\cdots \\ &FIp(X,Wp) && \leq \text{SpecCurve}(Wp) && \forall Wp \in [Wp_0, Wp_d]. \end{aligned}$$

where FI1, $\cdots$, FIp are functional inequality constraints and the SpecCurve's are the specifications for the functional constraints that are in general given in the form of a function (possibly a constant) of the independent parameters W1, $\cdots$, Wp.

When we applied DELIGHT.SPICE in an industrial environment, we observed that for circuit design problems, formulating a design problem as a mathematical programming problem and capturing the intent of circuit designers are not easy tasks. Sometimes these intents cannot be completely specified at the beginning of the design process. Thus, the system was modified to provide ways of specifying or changing design requirements while performing an optimization. Moreover, entering these requirements had to be made easy and intuitive for a circuit designer.

The way DELIGHT.SPICE deals with these aspects is the result of this close interaction with circuit designers at Harris Semiconductor. We observed first that designers mostly want to choose values of the design parameters so that a *set of objectives* rather than a single objective are "optimized" subject to a set of ordinary and functional constraints. Consequently the most general problem formulation allowed in DELIGHT.SPICE replaces the previous single objective formulation with

$$\underset{X}{\text{minimize}}\ \{M1(X), \cdots, Mr(X)\}.$$

where the minimization is interpreted in the minimax sense, i.e., the maximum of the r objective function values is minimized.

In addition, we observed that not all constraints are perceived by designers in the same way. Some constraints are treated as *hard* and some as *soft*. Hard constraints are constraints the designer considers most essential to have satisfied and which, once satisfied, the designer wishes to remain satisfied and not take part in any subsequent design tradeoffs. Obviously, any constraint whose satisfaction is necessary for physical realizability, such as a resistor value being positive, are treated as a hard constraint. Soft constraints, on the other hand, are those which the designer is interested in trading off against one another and against the performance objectives during intermediate iterations of an optimization run. DELIGHT.SPICE allows the user to specify whether a constraint entered is hard or soft, the default being hard.

Since objectives and soft constraints may be traded off by the designer, it is important to specify their relative importance to the optimization algorithm of DELIGHT.SPICE. For example, a constraint on power dissipation in a circuit such as *power≤400mw* might be very important to prevent chip overheating whereas the constraint *Zin≥10megohm* for high input impedance may be less important since often a considerably lower input impedance is acceptable.

A natural way of indicating the relative importance of objectives and soft constraints is by having the designer specify two values for each: a *good* value and a *bad* value. The meaning of these values is limited to the following understanding: having all of the various objectives and soft constraints achieve their corresponding good values should provide the same level of "satisfaction" to the designer for each, while achieving the bad values should provide the same level of "dissatisfaction". This provides a very simple way to do tradeoff analyses: if a designer is unhappy with the performance level achieved by a particular objective or constraint, he simply changes what he considers to be satisfactory or unsatisfactory by adjusting the good and bad values with interactive commands, and then resumes execution of the optimization.

The hard and soft constraint features mentioned above apply as well to design parameter box constraints. These are minimum and maximum bounds on each design parameter.

2.3 The Algorithm

DELIGHT has a large library of optimization algorithms [6]. However, this paper concentrates on one particular algorithm that has been developed for circuit design. The user does have the possibility of selecting a different algorithm but in that case the problem formulation and interactive features of the system would be different from what is explained in this paper.

The optimization algorithm in DELIGHT.SPICE is an enhanced version of the Phase I - Phase II Method of Feasible Directions with Functional Constraints [7,8]. The enhancements include the capability of handling multiple objectives (instead of just a single cost function), the notion of good and bad values to facilitate scaling and tradeoff exploration by designers, and an efficient way of handling both hard and soft box constraints on design parameters. *Feasible Directions Methods* were chosen for the following reasons:

1. Feasible Directions Methods, unlike other methods, have some guaranteed convergence properties;

2. Feasible Directions algorithms have been developed to solve optimization problems in which constraints are specified over intervals of an independent parameter such as time, temperature or frequency. These algorithms have been tested on a variety of engineering design problems;

3. If a set of parameter values that satisfies the constraints is found during the optimization process, the constraints remain satisfied during the remainder of the optimization run. This is important since it gives the designer more freedom to choose the point at which he considers the optimization process to be complete; stopping the optimization process at an early point still provides a feasible design, i.e., one that satisfies the constraints.

The algorithm described in full detail in [5] may be viewed as consisting of three distinct phases (and in fact may be called "The Phase I-II-III Method of Feasible Directions"). In phase I, the algorithm focuses all of its attention on satisfying the constraints to which the designer has given highest priority – the hard constraints. After all the hard constraints are satisfied, phase II is entered and the algorithm shifts its attention to improving all performance objectives and soft constraints simultaneously, while keeping all hard constraints satisfied. In the normal operation of phase II, the algorithm does not distinguish between objectives and soft constraints in seeking to improve the design[8]. In order to compare the various objectives and soft constraints, each such function is scaled (normalized) by the difference between its good and bad values. Thus, the meaning of the good and bad values, as stated previously, applies consistently to both objectives and soft constraints: having any objective at its good value and any soft constraint at its good value should provide the same satisfaction to the designer, and analogously for the bad values. Finally, phase III is entered when all the soft constraints and all the performance objectives take on values equal to or better than their corresponding good values. In phase III, the algorithm concentrates only on improving the performance objectives while keeping both the hard and soft constraints satisfied.

2.4 The *Pcomb* Graphical Display

In the progress of a multiple objective optimization computation, it is very desirable to have a display of objective and constraint values at each iteration which facilitates subjective evaluation of the design associated with that iteration. In DELIGHT.SPICE this purpose is served by the *Pcomb* performance comb, a graphical display which shows the designer how close each of his multiple

[1] This name stems from the fact that the second argument to FI in the formulation may be viewed as leading to an infinite number of constraints.

[8] There is, however, an optional *push factor* algorithm parameter, *Pushcost_F3*, by which the user can emphasize as a group either the objectives or the soft constraints in phase II.

objectives and soft constraints are to their corresponding good and bad values. Since most designer interaction with DELIGHT.SPICE is spent in phase II of the algorithm, hard constraints are not displayed. However, if any are violated in phase I, the message "- A Hard Constraint is Violated -" is printed at the top of the *Pcomb* display.

Referring to figure 1, the display consists of a vertical good line to the left and a vertical bad line to the right. On a color terminal, these are drawn in green and red, respectively. For all terminals, *G* appears above the good line while *B* appears above the bad line, as shown. Each objective or soft constraint is displayed by two horizontal bars or *teeth* on the comb, one for the previous comb drawn and one for the current comb. The previous comb teeth are in a lighter color or shade (and each slightly above the current one). The goal of the optimization algorithm is to move the tips of all the comb teeth to the left (in the direction of the good line). By minimizing the maximum of the objectives or constraints, the algorithm in effect tries to move the rightmost tip to the left, even if other tips move slightly to the right, i.e., their performance becomes worse.

The tip of each tooth is on the opposite end from a small diameter circular dot. The dot always is on the side of the smaller numeric value of the objective or constraint. Thus, if as for ordinary constraint I5 in figure 1, the dot is on the left, the good value is smaller than the bad value as for an objective being minimized or a constraint which must be less than its good value; a comb tooth which moves to the left toward the good line decreases its objective or constraint value. If as for ordinary objective M1 in figure 1, the dot is on the right, the good value is larger than the bad value as for an objective being maximized or a constraint which must be greater than its good value. In this case, a comb tooth which moves to the left toward the good line increases its objective or constraint value, as desired.

If an objective or constraint value is such that the tip of its comb tooth should be drawn off the *Pcomb* display, an arrow is drawn to show that the tooth is out of the comb range. The present values of inequality constraints I1 and I2 in figure 1 are both better than values for which their corresponding comb teeth can be plotted on the comb and thus they both have arrows. Note that I1 large is good since it is a *greater than* constraint while I2 small (actually large negative) is good since it is a *less than* constraint.

Also shown on the *Pcomb* display are the actual numeric values of the objectives and constraints and, for each functional objective or constraint, a small plot of the actual objective or constraint, its good curve, and its bad curve. Each of these plots is versus the corresponding "W" variable as it varies in the range specified by the user. To the right of each plot is shown the value of the "W" variable at which the corresponding functional objective or constraint takes on its worst value. The position of this value of the "W" variable is shown by a big circular dot on the functional plot.

The performance comb may be output automatically during each optimization iteration or manually after, say, adjusting the good or bad values for a particular objective or soft constraint. Since the comb display shows the previous comb teeth as well as the present one, the designer can easily see the results of such an adjustment of good or bad values as well as the improvement made by an optimization iteration.

2.5 Performing Tradeoffs

Tradeoffs between competing objectives or constraints are explored by adjusting good and bad values after several iterations of optimization. Basically, after several optimization iterations have been carried out with a set of good and bad values, the designer displays a performance comb and decides whether he/she is happy with the present values of his/her objectives and constraints. (In a tightly constrained design problem most of the algorithm execution will probably be spent in phase II. Recall that in phase II objectives and constraints are competing equally to be improved by the algorithm and the meaning of the good and bad values applies consistently to both.) If he is not happy with the present performance he adjusts good or bad values to reflect his feelings and resumes the optimization.

For example, suppose DC power is a performance objective in an optimization design problem, and it has been given good and bad values of 30mw and 50mw, respectively. Suppose that at the current iteration of the optimization, the power is 40mw. For these values, the associated comb tooth would end exactly half way between the good and bad vertical lines. Suppose that the designer is unhappy with the way several objectives (and/or constraints) have traded off and he actually wants to reduce the power further, at the expense of other objectives. This means that he now considers the value 40mw to be worst than he did previously relative to other objectives. This means that the DC power objective bad value should be closer to 40mw than it is presently. Thus, setting the bad value to, say, 45mw or even 40mw is the proper action. He then re-displays the comb and runs a few more optimization iterations.

3 DESIGN EXAMPLES

All of the examples introduced in this paper are derived from actual product development or redesign activities at Harris Semiconductor; the results of our efforts are presently being incorporated into several of the products discussed. The values of the design parameters with which the optimization sessions started were the end-products of manual design procedures by experienced designers. Thus, the improvements in performance that we report are even more significant.

3.1 High-Speed Operational Amplifier Specifications.

The circuit shown in figure 2 is an operational amplifier being produced at Harris Semiconductors. This circuit is implemented with a complementary bipolar, dielectrically-isolated gigahertz process. This technology offers diffused resistors, MOS capacitors and vertical NPN and PNP transistors with ft's of 1.5GHZ and 1.0GHZ respectively. This amplifier was to offer maximum bandwidth and stability at a reasonable closed-loop gain (5), along with a minimum settling time. The following were the initial design goals:

Gain-Bandwidth Product	≥ 300megHz
Output Voltage Swing	≥ 12.5v
± Slew Rate	≥ 300v/us
DC Power Dissipation	≤ 700m.w
Offset Voltage	≤ 5mv
"Stable" at a Closed Loop Gain of 5	

The DC limits, while important, were not of primary concern and could be relaxed to achieve better AC performance since the relative marketability of these specifications was very subjective. For example, if significant gain-bandwidth product could be maintained and stability increased at the cost of a small increase in power, the product might be more desirable.

3.2 Opamp Problem Formulation.

The first step in using DELIGHT.SPICE is the choice of design parameters and their initial values. Ideally, to explore the potential of a design, one should include all the parameters of a circuit. However, since the cpu time spent by the system is proportional to the number of design parameters[3], this strategy would be impractical. Thus, only circuit parameters that have considerable effect on the design objectives and specifications should be considered. For this example, the parameters chosen are the ones labeled in figure 2.

After the selection of a set of design parameters, it was necessary to translate the goals listed above into the mathematical programming formulation of DELIGHT.SPICE. In general, this consists of four steps. The first step is to express the objectives and constraints as explicit functions of the design parameters or of quantities that can be computed by SPICE. For example, consider the unity gain bandwidth product *GBW*. GBW can be determined by measuring the gain at a relatively low frequency (on the dominant pole portion of the Bode plot) and extrapolating. Thus

$$GBW = FREQ \cdot \left| \frac{Vout}{Vin} \right|$$

where FREQ is the frequency at which the open loop gain is measured.

[3] Presently gradients are computed by finite differences.

Second, the designer must decide whether each performance goal is to be considered as an objective or a constraint. Both of these require the specification of desired values. However, objectives continue to improve throughout the optimization while constraints "stop being pushed" after they achieve their desired values. For our opamp example we decided to consider the gain-bandwidth product as the only objective; all other goals were considered as constraints.

The third step is to decide whether each constraint is to be considered *hard* or *soft*. For the present example we decided that we wanted all of the constraints to take part in tradeoffs and hence we indicated (with the keyword *soft* as shown in Table 1) that all were soft.

The last step in setting up the problem formulation is to indicate the relative importance of the objectives and constraints by providing a *good* and a *bad* value for each. The precise values specified need only be "ball-park" values since DELIGHT.SPICE contains commands for modifying them during the optimization session. For our opamp example, the single objective gain-bandwidth product was calculated as the product of the gain at frequency 50megHz times 50megHz; this frequency was in the middle of the low frequency single pole rolloff range. The goal to be greater than 300megHz reflects our objective to improve the gain-bandwidth product as much as possible but at least have it as large as that of the initial design. This goal translates into having the gain greater than 300megHz/50megHz = 6 at this frequency. Thus the bad value for the corresponding objective was set to approximately 6 in decibels or 15db. The good value was initially set at 25db, corresponding to a gain-bandwidth product of about 900megHz. Good and bad values were determined for the various soft constraints in a similar manner, keeping in mind the uniform satisfaction/dissatisfaction rule.

3.3 Opamp Problem-Description Files.

A taste of the problem-oriented input language in DELIGHT.SPICE can be obtained by examining portions of our problem description files in Table 1. File *opamp1* shows that the output voltage swing was formulated as a separate inequality constraint for positive and negative swing. Similarly, the slew rate specification also resulted in two constraints. The slew rate was computed in an approximate way to avoid expensive transient simulations. In particular, DC charging currents and the compensation (plus parasitic) capacitance *CH* were used to approximate the slew rate by the expressions:

$$+\text{SLEWRATE} \simeq +\text{ISLEW}/\text{CH}$$

$$-\text{SLEWRATE} \simeq -\text{ISLEW}/\text{CH}$$

where

$$+\text{ISLEW} = (I_4 + I_5)\Big|_{\text{VIN}=+1}$$

$$-\text{ISLEW} = (I_4 + I_5)\Big|_{\text{VIN}=-1}$$

where I_4 and I_5 are defined in figure 2. The offset voltage specification does not appear as a constraint. V_{os} is made up of a preferential term and a statistical term. Since V_{os} was not of vital importance to this product, we avoided Monte Carlo analysis by considering only the preferential term. To insure that V_{os} was centered at zero, we simply forced certain nominally matched opamp parameters to track one another. To quantify the stability goal at a gain of five, we constrained the closed-loop frequency response to be less than 6.5 for every frequency in an interval. The interval range end-points were frequency values that we decided would surely contain the peak. Thus, stability was handled as a *functional* inequality constraint over frequency. Another approach that is much more computationally expensive would be to measure stability using the settling time or overshoot of a transient simulation output waveform. In general, care must be used in choosing how to compute the quantities involved in the specifications. Note that a similar problem arises when a designer simply wants to *examine* the performance of his/her design using circuit simulation, e.g., with batch SPICE.

The problem description files are shown in Table 1.

3.4 Opamp Optimization Sessions.

After completing the problem description files, entering the DELIGHT.SPICE environment, and issuing the command *solve opamp* to process those files, optimization was ready to begin. The command *run 5 Ptomb* requested that five iterations be performed, each followed by output of the *Ptomb* performance comb [6]. After the five iterations, the gain-bandwidth product was substantially improved at the cost of stability (greater peak in the closed-loop gain). We then changed our idea about what peaking was "bad" by reducing the corresponding bad value from 6 to 7.5. A few more iterations resulted in a high gain-bandwidth product and good stability but at the cost of a little higher DC power level. A view of the before and after results are shown in figure 3 with the latter representing a significant improvement over the initial circuit performance.

We next decided to change our major emphasis for the product to one of low power consumption. The good and bad values for the DC power constraint were lowered and after a few more optimization iterations, a low power version of the opamp was obtained with the following characteristics:

Gain-Bandwidth Product	= 460megHz
Peaking	= 19%
DC Power	= 450mw
Slewrate	= 250v/us

As demonstrated above, the ability to give different emphasis to objectives and constraints can result in a variety of attractive solutions, all of which can be realized with only minor changes in a few IC masks. Thus the designer can offer the marketing department several product options, each of which represents the circuits best performance for the particular emphasis given.

Other applications of DELIGHT.SPICE include the design of a LAC and an A/D comparator that show the variety of constraint specifications that can be handled in the system. Then, we optimized the design of a digital bus precharge circuit. In our last application we optimized a switched capacitor 10'th-order modem filter where tight phase linearity specifications were met without using costly equalization circuitry.

Lack of space in this paper prevents a similar discussion about the other design examples. The performance of each of them was significantly improved, as summarized in Table 2.

4 CONCLUDING REMARKS

DELIGHT.SPICE is a powerful tool for the design of analog circuits and of digital cells. Industrial ICs have been considerably improved with the use of the system. Presently, the system can be used by experts in optimization algorithms for developing new algorithms and by circuit designers with some background in optimization methods. We are working to make the system usable by designers with very little or no background in optimization techniques and to make the problem description easier to enter. Both of these goals can be achieved using a menu-driven user interface for DELIGHT. To build such an interface, we can take advantage of the uniform user/program environment available through the *hawk* color graphics editor, under development at the University of California, Berkeley, by Ken Keller. A menu-driven input to DELIGHT could reduce the apparent complexity of the many possible actions from which a designer can choose, at different states in an optimization process. In addition, work is in progress to make the interface between DELIGHT and SPICE simpler and more efficient, by coupling DELIGHT with SPICE3, also being developed at the University of California at Berkeley by Tom Quarles.

ACKNOWLEDGEMENTS

We thank Harris Corporation and in particular Jon Cornell, Vice-President, for providing an ideal environment in which to perform our experiments. We are grateful to Paul Hernandez, John Lazar, Gerry Cotreau, Graham Flower, Alex de la Plaza, Tom Guy, Jim Sutton and Bob Webb, all experienced designers, for working closely with us. We also thank Paul Gray and Bob Meyer of the University of California, Berkeley, for discussing with us the use of DELIGHT.SPICE in integrated circuit design. Special thanks to the brave students who took EECS 241 in the fall of 1982 at the University of California, Berkeley, and provided us with their feedback on the DELIGHT.SPICE system.

This work was supported by the Air Force Office of Scientific Research (AFOSR) United States Air Force Contract No. F49620-79-C-0178, by a grant from the Semiconductor Products Division of the Harris Corporation, and by a grant from MICRO.

REFERENCES

[1] W.T. Nye, E. Polak, A. Sangiovanni-Vincentelli, and A.L. Tits, "DELIGHT: An Optimization-Based Computer-Aided Design System," *Proceedings of the 1981 IEEE International Symposium on Circuits and Systems*, April 1981.

[2] L.W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," ERL Memo No. ERL-M520, University of California, Berkeley, May, 1975.

[3] R. A. Rohrer, "Fully Automated Network Design by Digital Computer: Preliminary Considerations," *Proc. IEEE*, 55, pp. 1929-1939, 1967.

[4] R. Brayton, G. Hachtel and A. Sangiovanni-Vincentelli, "A Survey of Optimization Techniques for Integrated Circuit Design," *Proc. IEEE*, vol. 69, no. 10, pp. 1338-1361, October 1981.

[5] W.T. Nye, A.L. Tits, "An Enhanced Methodology for Interactive Optimal Design," *Proceedings of the 1983 IEEE International Symposium on Circuits and Systems*, May 1983.

[6] W.T. Nye, *DELIGHT Reference Manual*, University of California, Berkeley, September, 1981.

[7] E. Polak, "Algorithms for a Class of Computer-Aided Design Problems: A Review," *Automatica*, vol. 15, pp. 795-813, Sept. 1979.

[8] D.Q. Mayne, E. Polak, and A. Sangiovanni-Vincentelli, "Computer-Aided Design via Optimization: a Review," *Automatica*, vol. 18, no. 2, pp. 147-154, 1982.

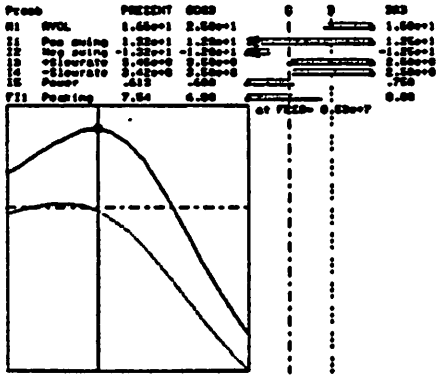


Figure 1. Example of the *Pcomb* graphical display.

File opampII

```
prob_function multicost
   objective 1 "AVOL" maximize (dbgain) good=25db bad=15db using {
      FREQ = 50meghz
      dbgain = db(vm(24)/vm(888))
      }
end_multicost
```

File opampI

```
prob_function ineq
   constraint 1 'Pos Swing' vdc(13) >= good=12.8v bad=12.5v soft
   constraint 2 'Neg Swing' vdc(16) <= good=-12.8v bad=-12.5v soft
   constraint 3 '+Slewrate'  ...
   constraint 4 '-Slewrate'  ...
   constraint 5 'Power' power <= good=600mw bad=750mw soft using
      power = ( vdc(222,2)*15v + vdc(111,1)*(-15v) )
end_ineq
```

File opampFT

```
prob_function fineq
   constraint 1 'Peaking' vm(24) <= good=8 bad=8 soft
   for_every FREQ from 70meghz to 150meghz initially dec 70
end_fineq
```

Table 1. Opamp Problem Description Files.

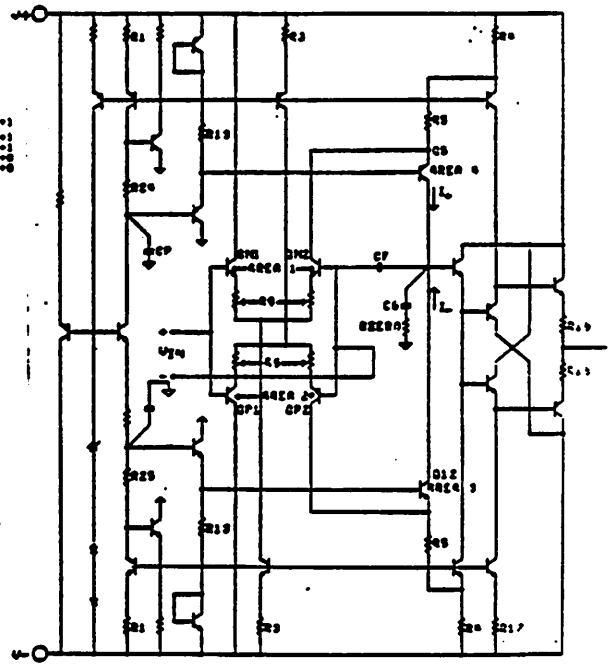


Figure 2. High Speed Opamp circuit diagram. All labeled resistor, capacitor, and transistor areas are design parameters.

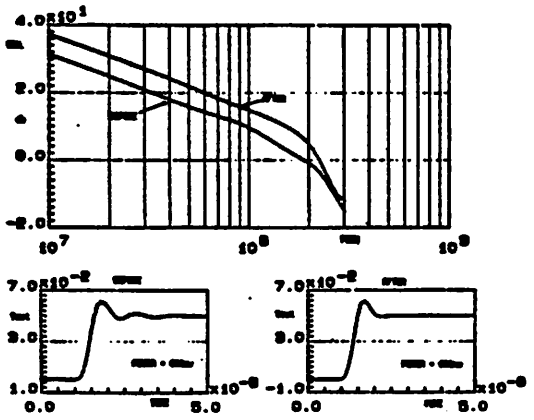


Figure 3. Results of DELIGHT.SPICE optimization of the High Speed Opamp. Gain-bandwidth product more than doubled (6db) and transient unit-step settling time is noticeably better.

PRODUCT	PERFORMANCE FACTOR	BEFORE	AFTER	% IMPROVEMENT	# ITERATIONS	CPU
HIGH SPEED OP AMP	Gain-bandwidth Stability (Peaking) Power Slew Rate	290 MHZ 21% 600mw 350v/us	650 MHZ 22% 690mw 340v/us	124% -4% -15% -3%	35	1 hr.
LOW-POWER HIGH-SPEED OP AMP	Gain-bandwidth Stability (Peaking) Power Slew Rate	290 MHZ 21% 600mw 350v/us	460 MHZ 19% 450mw 250v/us	58% 10% 33% -46%	7	11 min.
DAC-CONTROL AMPLIFIER	Settling Time Power	545ns 100mw	295ns 95mw	38% 5%	12	10 min.
A/D COMPARATOR	Delay Time Power Gain	375ns 18mw 1000	165ns 20mw 915	127% -5% -9%	6	7 min.
15TH ORDER MODEM FILTER	Group Delay Pass Band Stop Band	.15 °/HZ ±.5db -57db	.025 °/HZ .65db -48db	500% -30% -16%	55	1 hr.
BUS PRECHARGE CIRCUIT	Run1 Delay Run2 Delay Area	38ns 50ns 180	36ns 38ns 168	5% 31% -8%	6	30 min.

Table 2. DELIGHT.SPICE optimization results obtained on industrial designs at Harris Semiconductor.

APPENDIX H

DELIGHT for Beginners

Memo Number UCB/ERL M82/55, Electronics Research Laboratory, University of California, Berkeley, California, July 1982.

APPENDIX I

DELIGHT.SPICE User's Guide

Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, February 1983.