DELIGHT: AN OPTIMIZATION-BASED COMPUTER-AIDED DESIGN SYSTEM

by

W. Nye, E. Polak, A. Sangiovanni-Vincentelli and A. Tits

Memorandum No. UCB/ERL M81/19

15 April 1981

(cover)

DELIGHT: AN OPTIMIZATION-BASED COMPUTER-AIDED DESIGN SYSTEM

by

W. Nye, E. Pólak, A. Sangiovanni-Vincentelli and A. Tits

Memorandum No. UCB/ERL M81/19

15 April 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
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 sub-procedures for the ones he finds unsatisfactory.

c. An optimization algorithm expert who requires his computer programs 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 z = argmin { c'*x | x>=0,x<=d,A*x<=b }
```

where the array z 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 }
12-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 recompiling 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. Another define, "steploop", allows similar action in a subloop ,with "breakpt" playing the role of "interaction".

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 3 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 3

Iter=0  cost=3.000        ||gradcost||=4.472
Iter=1  cost=2.410        ||gradcost||=3.971
Iter=2  cost=1.945        ||gradcost||=3.531
Iter=3  cost=1.577        ||gradcost||=3.146

Interrupt...
2> Alpha = .6
2> run

Iter=4  cost=.6063        ||gradcost||=1.823
Iter=5  cost=9.548e-2     ||gradcost||=.6180
Iter=6  cost=2.387e-2     ||gradcost||=.3090
Iter=7  cost=5.967e-3     ||gradcost||=.1545
Iter=8  cost=1.492e-3     ||gradcost||=7.725e-2
Iter=9  cost=3.729e-4     ||gradcost||=3.862e-2
Iter=10 cost=9.324e-5     ||gradcost||=1.931e-2

    (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. Fairly sophisticated graphics, based on the macros and built-in functions of RATTLE, have been devised, some for the display of complex information required to monitor the progress of optimization algorithms, and some for use in digital filter design (e.g., for repeatedly displaying the frequency response of the filter and the constraint violations corresponding to the current value of the design variables. 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-76-04264 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.

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, pp.1-