NASA Technical Memorandum 100565

THE SIZING AND OPTIMIZATION LANGUAGE, SOL — A COMPUTER LANGUAGE FOR DESIGN PROBLEMS

STEPHEN H. LUCAS
STEPHEN J. SCOTTI

April 1988

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Abstract

This paper presents the Sizing and Optimization Language, SOL, a new high-level, special-purpose computer language developed to expedite application of numerical optimization methods to design problems and to make the process less error-prone. SOL provides a clear, concise syntax for describing an optimization problem, the OPTIMIZE description, which closely parallels the mathematical description of the problem. SOL offers language statements which can be used to model a design mathematically, with subroutines or code logic, and with existing FORTRAN routines In addition, SOL provides error-checking and clear output of the optimization results. Because of these language features, SOL is best-suited to model and optimize a design concept when the model consists of mathematical expressions written in SOL. For such cases, SOL's unique syntax and error checking can be fully utilized. SOL is currently available for DEC VAX/VMS systems. An evaluation copy of SOL is available from the Thermal Structures Branch, NASA Langley Research Center, which includes the SOL compiler, runtime library routines, and a SOL reference manual.

Introduction

The nonlinear mathematical programming method (formal optimization) has had many applications in engineering design (refs 1 and 2). Optimization software containing mathematical optimization algorithms is available in computer programs such as CONMIN, ADS, and NPSOL (refs 3, 4, and 5). Users must develop or obtain analysis software containing a mathematical model of the object being optimized, and then build an interface between the optimization software and the analysis software.

Three approaches to the problem implementation task have been identified in recent research. The traditional approach has been to write programs in a conventional language, such as FORTRAN. This approach is referred to as the *procedural* method, as optimization and analysis software are explicitly coupled with program statements, such as subroutine calls, logic statements, and loops. Second, the *query* method is exemplified by OPTDES.BYU, a system which provides for interactive selection of problem formulations, and also supports interfaces to standard analysis packages (ref. 6). In OPTDES.BYU design variables and constraints are interactively manipulated in a manner similar to the querying of a database. OPTDES.BYU also provides the capability for graphical output of design informa-

tion. The *constraint logic programming method* is a scheme for incorporating constraints as a programming tool within a declarative language such as PROLOG. Constraint logic programming allows goals to be defined in terms of satisfying a set of constraints (ref. 7). This technique has found some electrical engineering applications (ref. 8), but at present only linear constraints can be used.

Problem implementation can be tedious and error-prone. A lucid commentary on the difficulties of developing interfaces between optimization and analysis software appears in reference 9. Also, the majority of available implementations are only intended for specific types of problems. For example, the PASCO code, (ref. 10), only optimizes the design of stiffened plate structures. Often engineering users must create their own codes and can become overly involved in the details of computer implementation at the expense of focusing on proper formulation of the optimization problem. A need exists for tools which aid the user in developing analysis software and creating the required interface between optimization and analysis software.

This paper presents the Sizing and Optimization Language, SOL, a new high-level, special-purpose computer language to simplify implementation of optimization problems. SOL represents a *procedural* method, but differs from the traditional approach because SOL is a *special-purpose* language which provides statements with a clear, concise syntax for describing an optimization problem. A SOL compiler converts SOL statements into equivalent FORTRAN code. For example, SOL utilizes the ADS optimization software (ref. 4), so the compiler converts SOL's description of an optimization problem into equivalent FORTRAN subroutine calls to ADS. The task of building an interface between optimization software and modeling software is greatly simplifed using the language constructs avail-

able in SOL. Further, SOL provides error checking capabilities tailored to optimization problems.

The paper presents an overview of SOL, followed by a discussion on the description of an optimization problem in SOL. Details such as the choice of optimization algorithm, and the output of intermediate/final results are presented. The incorporation of existing FORTRAN codes into a SOL program and the error checking capabilities are also discussed. The paper concludes with general observations and assessments. Three examples illustrating the use of SOL appear in the appendices. The design of a two-bar, tubular truss appears in appendix A; the stress analysis of a cooling jacket is posed as an optimization problem in appendix B; and a complex SOL application to scramjet engine cooling jacket design appears in appendix C.

An Overview of SOL

Using SOL as a tool for engineering design involves writing a sequence of SOL statements that apply numerical optimization methods to a design problem. The process of solving a problem using SOL is shown in Figure 1. A program composed of SOL statements, which describes the design problem as an optimization problem subject to constraints, is passed as input to the SOL compiler. The SOL compiler translates the SOL program into an equivalent FORTRAN program, and does error-checking on the SOL program. The resulting FORTRAN program executes to solve the design problem. The SOL compiler is a program written in Pascal and was created with the MYSTRO compiler development system, a tool that simplifies compiler development (ref. 11).

SOL is a simple but powerful language. A brief overview of the language elements of SOL is offered here, and a detailed description appears in reference 12. A summary of

SOL statements appears in tables 1, 2, and 3. Table 1 shows the arithmetic, relational, and logical operators available in SOL; table 2 shows the built-in math functions; and table 3 offers a representative list of SOL statements with a description of their use. SOL offers many features found in "conventional" languages , e.g. FORTRAN or Pascal, such as variables; math operators; built-in math functions; loops; logic statements; subroutines; and some PRINT statements to allow the output of values. Presently SOL only allows INTEGER, REAL, and LOGICAL (true/false) types, and does not provide user-defined types or arrays.

SOL has unique language features as well, such as an OPTIMIZE description. Describing an optimization problem using SOL's OPTIMIZE description closely parallels the mathematical description of the problem. The next section discusses SOL's OPTIMIZE description in detail. SOL also includes a COMPONENT structure to facilitate the hierarchical modeling of a system, and a MACRO feature that allows the definition and use of abbreviations within a SOL program. The macro feature can be used to make SOL programs easier to write and more readable; macros are used in the example of appendix C. Additionally, FORTRAN code can be included directly inside a SOL program with a SOL FORTRAN BLOCK feature, as described subsequently.

SOL is quite simple aside from its unique features, but powerful when these elements are combined to address a design problem. SOL modeling and analysis of a design is accomplished by combining math-models, code logic, loops, external FORTRAN routines via FORTRAN BLOCKs, macros, and so forth, just as a FORTRAN or Pascal program would be written. SOL's OPTIMIZE description can then be used to describe the optimization of the design, and PRINT statements output the optimization results.

Optimization in SOL

SOL allows the description of an optimization problem with a high-level language feature, the OPTIMIZE description. Figure 2 shows a simple optimization problem posed mathematically, and in SOL. The mathematical formulation of the problem, shown in Figure 2(a), is to minimize some *objective* function of two bounded design variables x and y, subject to a single constraint relation. Mathematically, the problem is expressed as minimize $funct(x, y)$, where x and y are the design variables. The bounds on the design variables and the constraint relations are given under the heading, "Subject to." The single constraint relation is stated in the form of a general, nonlinear constraint function relation with the statement, "$constraint(x, y) = 5$." Although an equality constraint is depicted, SOL also allows inequality constraints. Finally the actual equations are given that define the objective and constraint functions in terms of the design variables.

SOL expresses the optimization problem in much the same way, as shown in Figure 2(b). The program begins with the word PROGRAM followed by the name of the program. The optimization problem is initiated by the word OPTIMIZE and terminated by the words END OPTIMIZE; the statements between are referred to as the "optimization description" or "OPTIMIZE description." A single variable given after the word OPTIMIZE states the objective function. Next, design variables and constraint relations appear between the words USE and END USE. The lower and upper bounds on the design variables appear enclosed by brackets. In addition, an initial value is required for each design variable to give the optimization software a "starting point." A single variable, constraint, in a constraint relation represents the constraint function. Any variable name can be used for design variables or the objective and constraint functions. Finally, between the words END USE

and END OPTIMIZE, equations are given which describe the objective and constraints as functions of the design variables. Although this example shows mathematical equations for the objective and constraint functions, SOL's optimization description is not limited to using single equations; a sequence of statements, such as subroutine calls and IF statements, could be used to define the objective and constraint functions. All that is needed for optimization is an algorithm (be it equation, subroutine, or other code) that will return the values for the objective and constraint functions given values for the design variables. The OPTIMIZE statement ends with the words END OPTIMIZE and then the SOL program ends with the word END followed by the program name.

SOL incorporates the methods of numerical optimization implemented in the ADS optimization routine for use within SOL programs. The ADS optimization routine contains a wide variety of algorithms. Although ADS supports SOL's optimization capabilty, the use of ADS is usually transparent to the author of SOL programs as shown by Figure 2(b). Also, only the objective and constraint function values are required in SOL, since all gradients are calculated using finite difference methods.

Aspects of using optimization software, such as the choice of available optimization algorithms, are specific to the actual optimization software in use. SOL contains an OPTIONS section as part of its OPTIMIZE description which gives user access to the optimization software dependent features, as shown in Figure 3. Since this discussion centers on the OPTIONS section, the actual design variable and constraint relation statements have been deleted for clarity. The OPTIONS section begins with the word OPTIONS. Between the word OPTIONS and the END USE, the SOL user can select any strategy, optimizer, or one-dimensional search available in ADS. Some recent research into tools for option selec-

tion exists. Specifically, an expert system tool for choosing among optimization methods available in ADS is described in reference 13. In this example, the statement "`optimizer = Modified Feasible Directions`" indicates that the method of modified feasible directions is desired. A "golden section" method of one-dimensional search is also specified. Additionally, the other options of ADS are all accessible through `OPTIONS` statements. The `PRINT` statement shown in the example of Figure 3 is discussed in the next section of this paper. The `OPTIONS` section ends with the words, `END USE`. At this point, code modeling the optimization problem would appear as in the example of Figure 2(b), but has been ommited here. Although SOL automatically supplies default option values for the new user, a knowledgable user can take full advantage of the more arcane options offered by the ADS software with the `OPTIONS` section.

The clarity of SOL's syntax, shown in Figure 2(b), becomes apparent when compared with the problem as originally coded in `FORTRAN`, shown in Figure 4. Variable declarations and initializations appear, along with subroutine calls invoking the ADS software. Upon further perusal of this code, the dim outline of the optimization problem can be seen. The `REAL*8 DESIGN`, `DESIGN_LOW`, and `DESIGN_UP` arrays represent the design variables, the design variables' lower bounds, and upper bounds respectively; the equations describing the objective and constraint functions appear after the call to the ADS software. In contrast, SOL offers a clear expressive way to describe an optimization problem which focuses attention on the optimization problem itself.

SOL Output of Optimization Results

SOL provides statements to request printing the values of the objective function, design variables, constraint functions, and termination criteria at user-selected points during the

optimization process. These output statements are placed in the OPTIONS section of an OPTIMIZE description. For example in Figure 3, the line PRINT design variables EVERY ITERATION requests output of design variable values at every iteration of the optimizer. The choice exists to print values at the initial design condition, at the one-dimensional search level, at the optimizer or strategy level, and at the end of the optimization. The user can request that only active or violated constraints be printed, PRINT everything or PRINT nothing.

Although an output capability already exists within ADS, the output produced by SOL PRINT statements is integrated with SOL's language statements. A sample of SOL's output for the final results of an optimization is given in Figure 5. A title header appears, followed by the final values for the objective function, design variables and constraint functions. The actual name of the design variable or variable representing the objective or constraint function from the SOL program is printed. In the case of design variables, the bounds of the variable are also printed. If a design variable is at a bound, an asterix appears at the far right, as in the case of inlet_pressure. In the case of constraints, the values, types and the final status of the constraint relations are printed.

Incorporating Existing Codes in SOL

Existing FORTRAN subroutines for analysis and computation can be called from a SOL program, with communication via subroutine parameters. Further, FORTRAN code can be included directly inside a SOL program via a SOL feature called a FORTRAN BLOCK. This makes incorporating existing FORTRAN subroutines and code fragments quite simple, when variables can be easily passed back and forth via subroutine parameters. For the cooling jackets designed with SOL as described in Appendix C, existing FORTRAN subroutines

were used to calculate temperatures, temperature dependent material stresses, and other constraint functions. Since the subroutines were already written in FORTRAN, rewriting them as SOL statements was not warranted.

The task of modifying a major existing analysis routine into a parameter-passing subroutine can be a difficult. SOL provides no support for this task. Furthermore, SOL does not provide the error-checking features outlined subsequently for the incorporated FORTRAN routines and/or FORTRAN BLOCK code fragments. SOL's error-checking is limited to SOL statements only. Because of these limitations, SOL is best suited for the development of models created entirely with SOL statements.

Error Checking by SOL

A key feature in the SOL compiler is the capability to check a SOL program for a variety of errors. For example, the SOL compiler gives an error message when an uninitialized variable is used in an equation , or when a LOGICAL variable is assigned an INTEGER value, or when the design variables of an optimization problem are altered by statements inside an OPTIMIZE description. The compiler does not check the correctness of the design problem formulation, but does catch a variety of errors automatically to expedite the successful implementation of an optimization problem. Additionally, the SOL compiler can produce a *listing* that is useful when debugging SOL programs. The listing includes the program indexed by reference line numbers, an optimization summary for each optimization which lists the objective, design variables, and constraints, a *variable cross-reference* that lists each variable and the lines on which the variable was used, and a listing of all the errors with the line numbers where the error occured.

An intentionally erroneous SOL program is given in Figure 6. The function of the

program is unimportant to this discussion, which focuses on the types of error-catching available with SOL. Line numbers appear to the left of the program statements, as they would appear in the listing. Figure 7 shows the error messages issued by the SOL compiler that accompany the program shown in Figure 6. The error messages list both the line number and the line where the error occured, a short message indicating the error, and a pointer to the approximate location of the error in the line.

The first error occurs on line 18, where the word `everything` is mispelled. The SOL compiler can usually correct the spelling of reserved words when the word is mispelled by only one character. The next error occurs on line 20 where the variable, `ar2`, is referenced before it has been initialized. Since `ar2` has never been assigned a value, the compiler issues an error message as seen in Figure 7. Yet another error occurs on line 21 where a design variable is illegally altered; the compiler gives an appropriate message. The final error message occurs inside subroutine `Get_Stresses`, called to return values for `stress1` and `stress2`. The subroutine never assigns a value to `stress1`, so the compiler issues an error message. This example, although not exhaustive, illustrates the general type of error caught by the compiler. In all, the SOL compiler has nearly ninety different possible error messages. By discovering these errors when the program is compiled, it is hoped that the time spent tediously tracing through a large program (for example to find which of twenty-four variables was left undefined) will be reduced.

Computer Implementation

Numerical precision or accuracy is a concern in computer applications of numerical optimization techniques (especially for finite difference derivatives). For this reason, all SOL `REAL` variables are double-precision (64 bit precision, `REAL*8`) in the `FORTRAN` code

produced by the SOL compiler. The FORTRAN code output by the compiler is VAX FORTRAN rather than standard FORTRAN. In addition, the SOL compiler is written in non-standard VAX Pascal. For these reasons, SOL is presently available only for DEC VAX/VMS systems. Converting the SOL compiler to emit standard FORTRAN would be a relatively minor change, but rewriting the compiler in standard Pascal is a major effort.

General Observations

There are several advantages to the use of a special-purpose language as a means to easing optimization problem implementation. First, the language itself can offer a clean, expressive method of describing an optimization problem. The language can also evolve to incorporate useful new ways to describe optimization problems, just as natural languages adapt to usage by incorporating new words. Second, the compiler allows the clear optimization descriptions to be translated automatically into a computer implementation. In this way, the compiler acts as intermediary between the user and the machine. The compiler offers advantages beyond its primary function of translation, because the compiler provides a vehicle for error-checking. Finally, the language description of an optimization problem is separated from the code that implements the problem. Thus, the compiler can be modified to utilize better optimization software and newer algorithms as they became available, without affecting existing programs (except for the optimizer specific options). Although altering the compiler is a non-trivial task, one can take advantage of better optimization methods by substantial modifications to a single piece of software (the compiler) rather than by modifications to every existing program.

Summary

A high-level, special-purpose programming language, SOL, has been developed to ex-

pedite implementation of optimization problems and to make the process less error-prone. SOL uses a procedural method to apply optimization. and integrates optimization and modeling within a single OPTIMIZE description. As a language, SOL provides language statements which can be used to model a design mathematically or to model a design with subroutines and other code. SOL also permits the user to incorporate existing analysis tools written in FORTRAN via subroutine calls and parameter-passing. SOL provides error checking geared to optimization and clear output of the optimization results. However, SOL's error-checking does not apply to external routines incorporated with the FORTRAN BLOCK feature. Because of its language features and error-checking, SOL is best-suited to model and optimize a design concept when the model consists of mathematical expressions written in SOL. Finally, SOL has the potential to evolve in several ways. First, the SOL compiler can be modified to utilize better optimization software when available, without affecting existing SOL programs (except for the optimizer specific options). Second, SOL itself can evolve, as useful features are recognized and incorporated within the language. Currently SOL is only available for DEC VAX/VMS systems. An evaluation copy of SOL is available from the Thermal Structures Branch, NASA Langley Research Center, which includes the SOL compiler, runtime library routines, and a SOL reference manual.

Nasa Langley Research Center
Hampton, Va. 23665-5225
February 1988

Table 1: SOL Operators

Arithmetic Operators	Description
+	addition
*	multiplication
/	division
-	subtraction
**	exponentiation

Relational Operators	Description
.eq.	equal to
.ne.	not equal to
.gt.	greater than
.ge.	greater than or equal to
.lt.	less than
.le.	less than or equal to

Logical Operators	Description
.and.	logical conjunction
.or.	logical disjunction
.not.	logical negation

Table 2: SOL Built-in Functions

Function	Function Description
ABS	Absolute value
ATAN	Arc Tangent
COS	Cosine
EXP	e raised to some value
LOG	The natural log of a value
INT	Truncates a value to integer
SIN	Sine
SQRT	Square-root
TAN	Tangent

Table 3: SOL Statements

Calculation Statements	Description
Assignment	Assigns a value to a variable
Expression	Combines variables, operators, and/or functions to give values
Control Statements	**Description**
Conditional DO loop	Repeats statements until a logical condition is true
IF/THEN/ELSE	Branches based on a logical condition
Iterative DO loop	Repeats statements a specified number of times
Subroutine call	Calls a SOL subroutine
Declaration Statements	**Description**
Subroutine declaration	Declares a subroutine and its parameters
Variable declaration	Declares a variable's type
Description Statements	**Description**
COMPONENT	Describes a hierarchical modeling construct
OPTIMIZE	Describes an optimization problem
Miscellaneous Statements	**Description**
FORTRAN block	Incorporates FORTRAN code into the program
?INCLUDE	Includes an existing file into the program
Macro call	Use a macro abbreviation
Macro definition	Defines a macro abbreviation
Output Statements	**Description**
PRINT	Output a value or optimization result

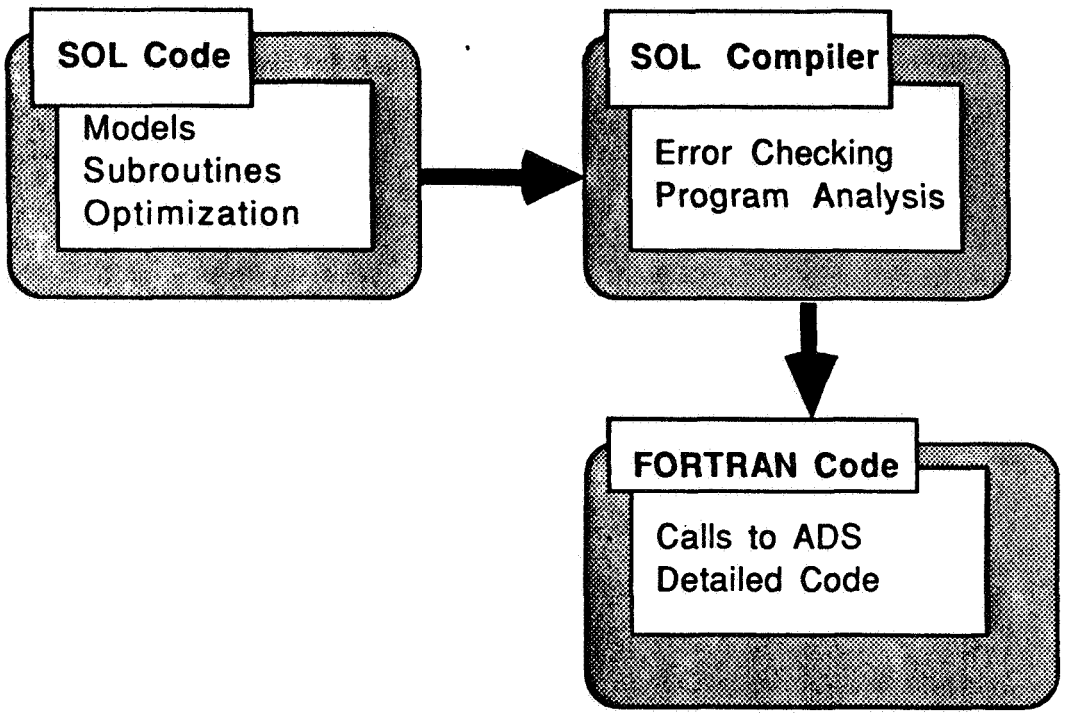

Figure 1: Schematic of SOL's use to solve an optimization problem

Minimize: funct(x,y)

Subject to:

$-20 \leq x \leq 50$
$0 \leq y \leq 10$
$constraint(x, y) = 5$

Where:

$funct(x, y) = 10 \times (y - x^2)^2 + (1 - x)^2$
$constraint(x, y) = x \times y$

2(a) Mathematical description:

```
PROGRAM easy
OPTIMIZE funct
USE

   x = -1.2 IN [-20, 50]
   y = 1 IN [0, 10]
   constraint .eq.  5

END USE

   funct = 10*(y - x**2)**2 + (1 - x)**2
   constraint = x * y
END OPTIMIZE
END easy
```

2(b) SOL program

FIGURE 2 – DESCRIPTION OF AN OPTIMIZATION PROBLEM

```
OPTIMIZE weight
USE
   :
   :
   :
OPTIONS
   !  Optimizer-specific settings
   !
  optimizer = MODIFIED FEASIBLE DIRECTIONS
  search    = GOLDEN SECTION
  PRINT design variables EVERY ITERATION
END USE
   :
   :
   :
END OPTIMIZE
```

FIGURE 3 – THE OPTIONS SECTION OF SOL'S OPTIMIZE DESCRIPTION

```
      PROGRAM EASY
C      Main Program declarations
      IMPLICIT REAL*8 (A-Z)
      INTEGER INFO, ISTRAT, IOPT, IONED, IPRINT, IGRAD,
     & NDV, NCON, NGT, NRA, NCOLA, NRWK, NRIWK, IDG(1),
     & IC(1), IWK(210)
      REAL*8 DESIGN(3), DESIGN_LOW(3), DESIGN_UP(3), DF(3),
     & A(3,3), CONSTR(1), WK(557)
C       Initializations for Optimization
      ISTRAT = 0
      IOPT = 5
      IONED = 7
      IPRINT = 1000
      IGRAD = 0
      INFO = -2
      NRIWK = 210
      NRA = 3
      NCOLA = 3
      NRWK = 557
      NDV = 2
      NCON = 1
C     Design variable, constraint declaration and initialization
      DESIGN_LOW(1) = -20
      DESIGN(1) = -1.2
      DESIGN_UP(1) = 50
      DESIGN_LOW(2) = 0
      DESIGN(2) = 1
      DESIGN_UP(2) = 10
      IDG(1) = -1
C      Calls to optimization software
1     CONTINUE
      CALL ADS(INFO, ISTRAT, IOPT, IONED, IPRINT, IGRAD,
     & NDV, NCON, DESIGN, DESIGN_LOW, DESIGN_UP, OBJ,
     & CONSTR, IDG, NGT, IC, DF, A, NRA, NCOLA, WK,
     & NRWK, IWK, NRIWK)
C     Modeling, analysis code
      X = DESIGN(1)
      Y = DESIGN(2)
      FUNCT = 10*(Y-X**2)**2+(1-X)**2
      CONSTR(1) = X*Y
      OBJ = FUNCT
      IF (INFO .NE. 0) GO TO 1
      END
```

FIGURE 4 – FORTRAN CODE EQUIVALENT TO THE SOL PROGRAM OF FIGURE 2(B).

```
*********************************************************
* OPTIMIZATION FINAL RESULTS                            *
*********************************************************

OBJ_FLOWRATE                =     2.68346


                    *** DESIGN VARIABLES OUTPUT ***
                    *******************************

DESIGN VARIABLE              CURRENT VALUE                    BOUNDS

PANEL_FLOWRATE          =    2.6835        IN  [ 2.500     ,  10.00     ]
INLET_PRESSURE          =    4500          IN  [ 1200.     ,  4500.     ] *
PIN_H_O_D_PANEL1        =    1.0972        IN  [0.2000     ,  6.000     ]

                    *** CONSTRAINTS OUTPUT ***
                    **************************

CONSTRAINT NAME               VALUE        TYPE      LIMIT       STATUS

PINSTRESS_PANEL1           0.11436E-01      <       1.0000      SATISFIED
MICROWIDTH_PANEL1          0.54095E-02      >     0.50000E-02   ACTIVE
GAS_P_OUT                 -0.97474E+06      >       600.00      VIOLATED
```

FIGURE 5 – SAMPLE SOL OUTPUT OF OPTIMIZATION RESULTS

```
1    : PROGRAM ThreeBarTruss
2    : !
3    : !  Classical Three-Bar Truss problem; formulated by Vanderplaats in
4    : !  Nasa Contractor Report 177985
5    : !
6    : DECLARE
7    :   SUBROUTINE (stress1, stress2) = Get_Stresses(area1_3, area2)
8    : END DECLARE
9    :
10   : OPTIMIZE truss
11   : USE
12   :   area1_3 = .5 IN [.01, 20000]
13   :   area2   = .5 IN [.01, 20000]
14   :   stress1 .lt. 0
15   :   stress2 .lt. 0
16   : OPTIONS
17   :   optimizer = feasible directions
18   :   print everthing every iteration
19   : END USE
20   :   truss = 2*SQRT(2.0)*area1_3 + ar2
21   :   area1_3 = 2*area1_3
22   :   (stress1, stress2) = Get_stresses(area1_3, area2)
23   : END OPTIMIZE
24   : END ThreeBarTruss
25   :
26   : SUBROUTINE (stress1, stress2) = Get_stresses(a1, a2)
27   :
28   :   st1 = (2*a1 + sqrt(2.0)*a2)/(2*a1*(a1 + sqrt(2.0)*a2)) -1
29   :   stress2 = 1/(a1 + sqrt(2.0)*a2) - 1
30   : END Get_Stresses
```

FIGURE 6 – AN INTENTIONALLY ERRONEOUS SOL PROGRAM

```
   18   :   print everthing every iteration
*** ERROR                ^ MISSPELLED "EVERYTHING" CORRECTED.
   20   :   truss = 2*SQRT(2.0)*area1_3 + ar2
*** ERROR                                    ^ UNINITIALIZED IDENTIFIER.
   21   :   area1_3 = 2*area1_3
*** ERROR          ^ ILLEGAL USE OF AN OPTIMIZATION DESIGN VARIABLE
   26   : SUBROUTINE (stress1, stress2) = Get_stresses(a1, a2)
*** ERROR                   ^ PARAMETER HAS NOT BEEN DEFINED IN SUBROUTINE
```

FIGURE 7 – THE ERROR MESSAGES ISSUED FOR THE PROGRAM IN FIGURE 6.

Appendix A: Symmetric Two-Bar Tubular Truss Example

The symmetric two-bar tubular truss is a classic enginnering design application of numerical optimization methods (refs. 1 and 14). A truss schematic appears in Figure A1. The truss consists of two members of tubular steel, where the truss is to support the load, $2P$. The height of the truss is denoted by h, and the span of the truss is fixed at $2B$. A cross- sectional view of a tubular member is also shown, where d represents the tube diameter. The design problem is to select d and h such that the weight of the truss is minimized, and the members neither yield compressively nor buckle. The load (P) is assumed as 33 kips; the half-span (B) as 30 in.; and the tube wall thickness (t, not shown) as 0.1 in. In addition, the compressive strength of the material ($\sigma^{\max}$) is taken as 100,000 psi., the modulus (E) as 30×10^6 psi., and the density (ρ) as 0.3 lb/in^3.

The truss can be modeled with mathematical equations. When L is the length of a member, $L = \sqrt{B^2 + h^2}$, the equations (A1), (A2) and (A3) model the weight of the truss, the member stress, and the Euler buckling stress respectively:

$$weight = 2\rho\pi dtL \tag{A1}$$

$$\sigma = \frac{P}{\pi t}\frac{L}{hd} \tag{A2}$$

$$\sigma^e = \frac{\pi^2 E}{8}\frac{\left(d^2 + t^2\right)}{L^2} \tag{A3}$$

The design problem can be posed as an optimization problem, in which truss weight is the objective function to be minimized with d and h as design variables. Compressive stress and Euler buckling constraints insure that the truss neither yields nor buckles.

A mathematical formulation of the problem is shown in Figure A2. The design variables and constraint relations appear under the heading "**Subject to:**," followed by the equations for the objective and constraint functions under the heading "**Where:**." The design variables are bounded, and the compressive stress and Euler stress constraints are expressed as inequality constraint function relations.

A complete SOL program for the two-bar truss appears in Figure A3, with line numbers displayed to facilitate discussion. Lines 2 through 8 initialize the constants required for calculations, and the optimization description appears on lines 10 through 22. Design variables and constraint function relations appear between the words `USE` and `END USE`, with the analysis equations appearing between the words `END USE` and `END OPTIMIZE`. The optimization description in SOL parallels the mathematical description of Figure A2, except that a single SOL variable represents the Euler buckling constraint, and member length is explicitly calculated on line 17. This program was passed as input to the SOL compiler to produce an equivalent `FORTRAN` program which was executed. The optimization results produced by SOL appear in Figure A4, showing the minimum truss weight was 12.8 lb., with both constraints active.

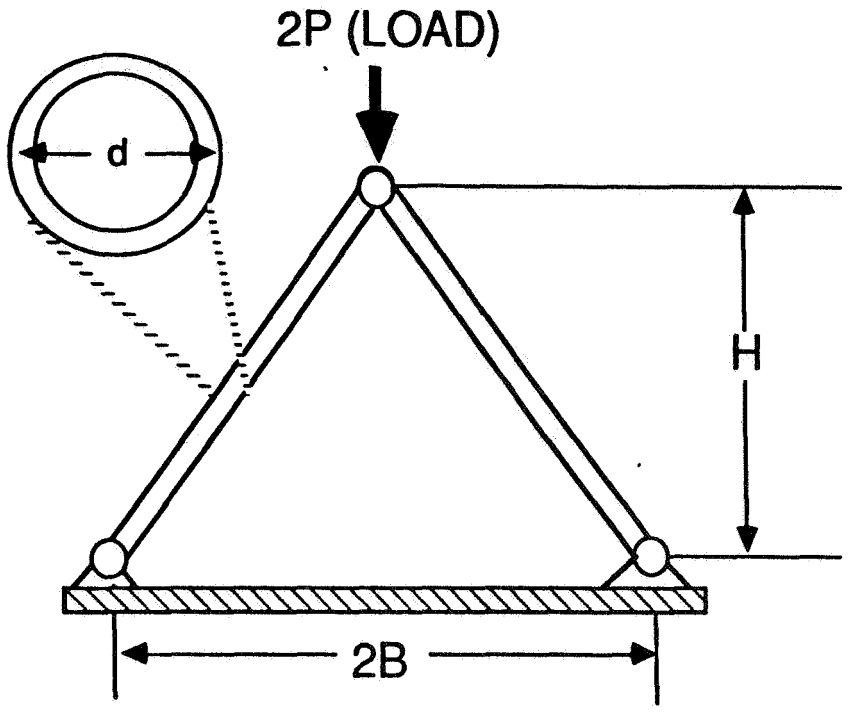

FIGURE A1: Two-Bar, Tubular Truss Schematic

Minimize: weight(d,h)
Subject to:

$1 \leq d \leq 3$
$10 \leq h \leq 30$
$\sigma(d,h) < \sigma^{max}$
$\sigma(d,h) - \sigma^{e} < 0$

Where:

$weight = 2\rho\pi dtL$
$\sigma = \frac{P}{\pi t}\frac{L}{hd}$
$\sigma^{e} = \frac{\pi^2 E}{8}\frac{(d^2+t^2)}{L^2}$

FIGURE A2 – MATHEMATICAL FORMULATION OF TWO-BAR TRUSS OPTIMIZATION

```
1    : PROGRAM TwoBar
2    : t         = 0.1          ! Wall thickness
3    : P         = 33000        ! Load
4    : B         = 30           ! Half-span of truss
5    : E         = 30000000     ! Modulus
6    : pi        = 3.141592554
7    : rho       = 0.3          ! Material Density
8    : MaxStress = 100000       ! Maximum allowable stress
9    :
10   : OPTIMIZE weight
11   : USE
12   :   d = 1  IN [1, 3]
13   :   h = 15 IN [10,30]
14   :   stress .lt. MaxStress
15   :   buckle .lt. 0
16   : END USE
17   :   L        = SQRT(B**2 + h**2)
18   :   weight   = 2*rho*pi*d*t*L
19   :   stress   = (P/(pi*t))*L/(h*d)
20   :   e_stress = ((pi**2*E)/8)*((d**2 + t**2)/L**2)
21   :   buckle   = stress - e_stress
22   : END OPTIMIZE
23   : END Twobar
```

FIGURE A3 – SOL PROGRAM FOR TWO-BAR TRUSS OPTIMIZATION

```
*********************************************************
* OPTIMIZATION FINAL RESULTS                            *
*********************************************************

WEIGHT                        =      12.8120


                        *** DESIGN VARIABLES OUTPUT ***
                        *******************************

DESIGN VARIABLE                 CURRENT VALUE                      BOUNDS

D                               =     1.8784          IN  [ 1.000      ,  3.00      ]
H                               =     20.233          IN  [ 10.00      ,  30.00     ]

                        *** CONSTRAINTS OUTPUT ***
                        **************************

CONSTRAINT NAME                   VALUE          TYPE       LIMIT         STATUS

STRESS                         0.10001E+06        <     0.10000E+6       ACTIVE
BUCKLE                          -2.0921           <     0.0000E-00       ACTIVE
```

FIGURE A4 – SOL OUTPUT OF TWO-BAR OPTIMIZATION RESULTS

Appendix B: A Simple Cooling Jacket Example

SOL has been applied to the design of cooling jackets. Cooling jackets are used in high temperature environments to maintain structures at survivable temperatures. A cross section of a typical cooling jacket installed on a structure bearing the primary structural loads (the primary structure) is shown in Figure B1. The jacket is a sandwich structure which consists of an outer wall which is heated by the high environmental heat flux, an inner wall which is attached to the primary structure, and a system of webs which connects the outer and inner walls. A coolant, flowing between the outer and inner walls inside channels formed by the webs, removes the heat incident on the outer wall of the cooling jacket. There are two loads that the cooling jacket must withstand, the load from the coolant pressure, p, and the thermal load due to the temperature difference between the cooling jacket outer and inner walls. The pressure load will cause the outer wall to be loaded in bending. A high thermal load will typically cause the outer wall of the cooling jacket to yield and deform plastically in compression. The question which arose is "how can the outer wall withstand the pressure load when it is already yielded due to the thermal load?" Ordinarily, such questions would be answered with a non-linear, finite element analysis. However, lacking such tools, the question was posed as an optimization problem to minimize strain energy, and a simple SOL program was written to do the optimization.

The elastic-perfectly plastic stress-strain curve in Figure B2 was assumed for the outer wall material. The stress-strain state of the outer wall with only thermal loading is shown in the Figure. Also it was assumed that the stress through the thickness of the cooling jacket outer wall will have the form in Figure B3 with the pressure induced bending moment locally relieving the compressive thermal stress over a portion of the thickness. With

this model, there are two unknowns, the surface stress (σ_{surface}) and the elastic thickness (t_{outer}). To solve for these quantities, the jacket's outer wall was assumed to behave like a beam composed of two materials, an outer material with a finite elastic modulus ($E > 0$) and an inner material with a zero modulus ($E = 0$) modeling the yielded part of the outer cooling jacket wall. With the applied pressure inducing the moment M_p, the beam equations to be satisfied are:

$$\sigma_{\text{ave}} = \frac{\sigma_{\text{yield}} + \sigma_{\text{surface}}}{2} \tag{B1}$$

$$M_p = (\sigma_{\text{surface}} - \sigma_{\text{ave}}) \frac{(t_{\text{outer}})^2}{6} \tag{B2}$$

Equation (B2) is actually the beam stress equation transposed to solve for applied moment, (M_p). These equations do not completely solve the problem because the additional unknown σ_{ave} has been introduced. An additional criteria was assumed. The solution would minimize the elastic strain energy of the outer (unyielded) part of the beam. The strain energy (U) of the outer part of the beam is found from the relation:

$$U \propto \int_{t_{\text{outer}}} \sigma^2 dt = \left(\frac{t_{\text{outer}}}{\sigma_{\text{surface}} - \sigma_{\text{yield}}} \right) \times \left(\frac{\sigma^3_{\text{surface}}}{3} - \frac{\sigma^3_{\text{yield}}}{3} \right). \tag{B3}$$

The problem is posed as an optimization problem; minimize the elastic strain energy of the outer part of the beam subject to the design variables, t_{outer} and σ_{surface}. Further, the pressure induced moment expressed by equation (B2) can be used as an equality constraint to force the governing beam equations to hold. Since the actual moment is a known quantity, the moment calculated by equation (B2) must equal the actual moment.

A SOL program for the problem is shown in Figure B4. The yield stress (σ_{yield}), total thickness of the jacket's outer wall, and pressure induced moment (M_p) are known quantities initialized on lines 5–7. The minimization of the strain energy begins on line 9 with

an OPTIMIZE description, with the design variables and constraint relation stated between lines 10 and 14. The beam equations are stated on lines 18 and 19, followed by the objective function of minimizing strain energy on line 20. There were two subtleties involved in the implementation. First, the lower bound on the design variable surface_stress is slightly larger than yield stress to insure that division by zero does not occur in equation 3 on line 20. Second, the normalization option chosen on lines 15–16 scales the design variables to the same order of magnitude. The OPTIMIZE description ends on line 22, followed by a calculation and a PRINT statement to output how far through the thickness the elastic portion of the beam starts.

A feasible solution was found with the outer, elastic portion of the beam starting approximately half-way through the thickness of the outer wall. The thermally yielded outer wall can support a pressure load and in fact does so elastically.

This example shows the SOL's capability to solve an optimization problem with a minimum of effort. SOL does not formulate the problem, however, the speed with which a problem can be modeled using SOL can help to get a quick solution. SOL's capability for quick modeling and optimization encouraged the use of an optimization method for the jacket design.

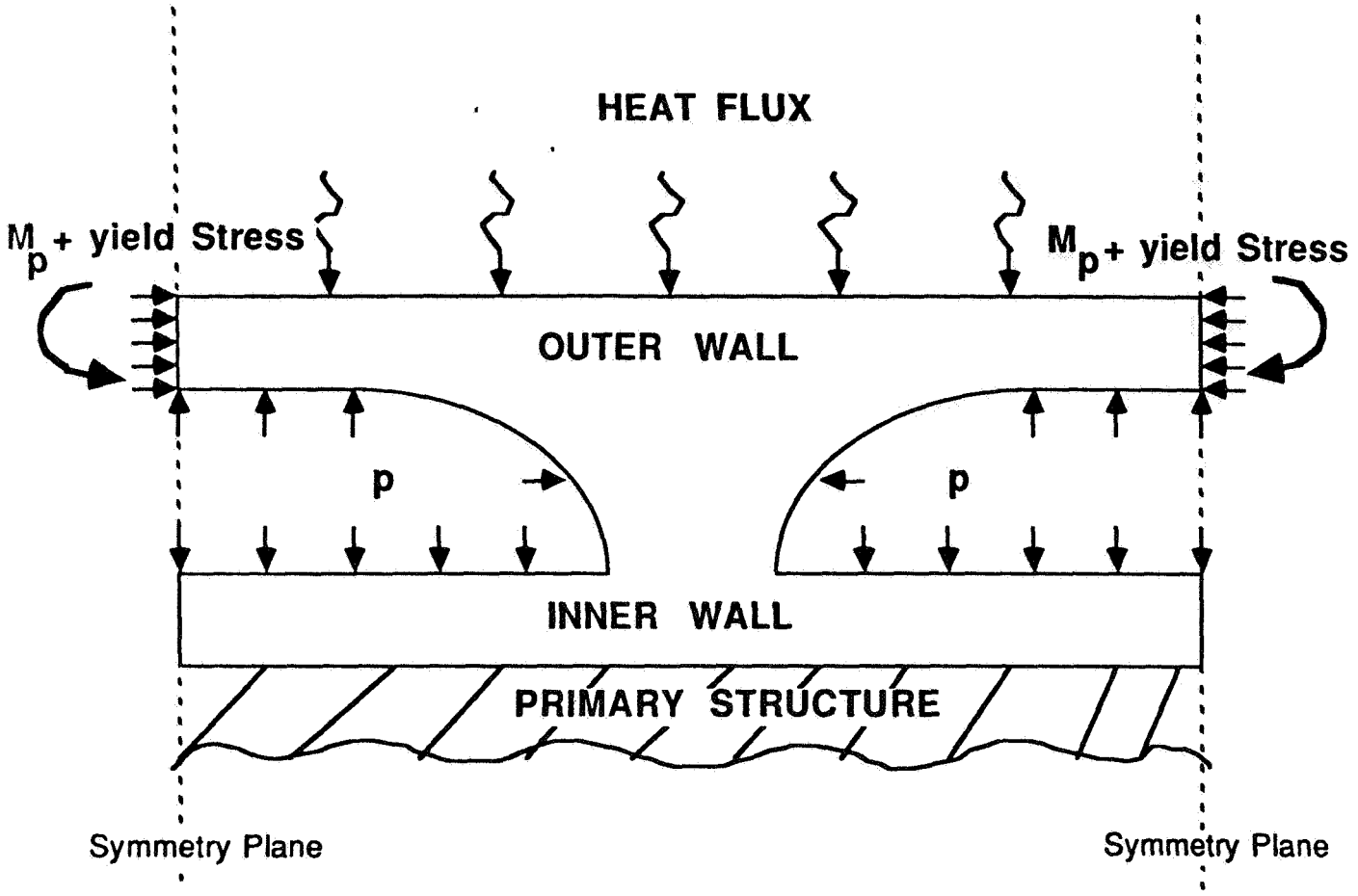

Figure B1: Schematic of Cooling Jacket Model

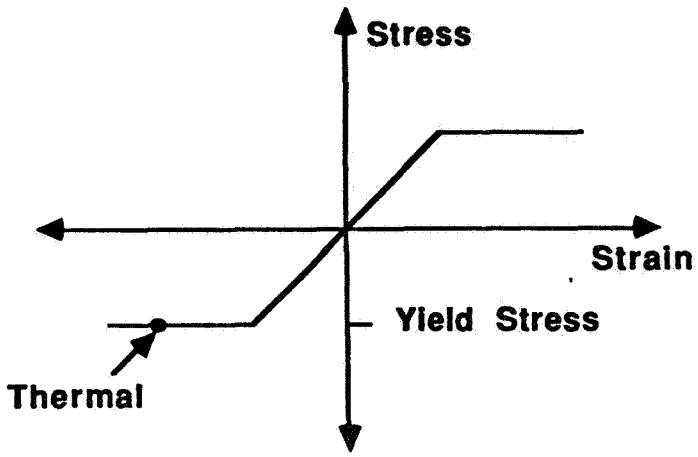

Figure B2: Assumed Stress-Strain Curve

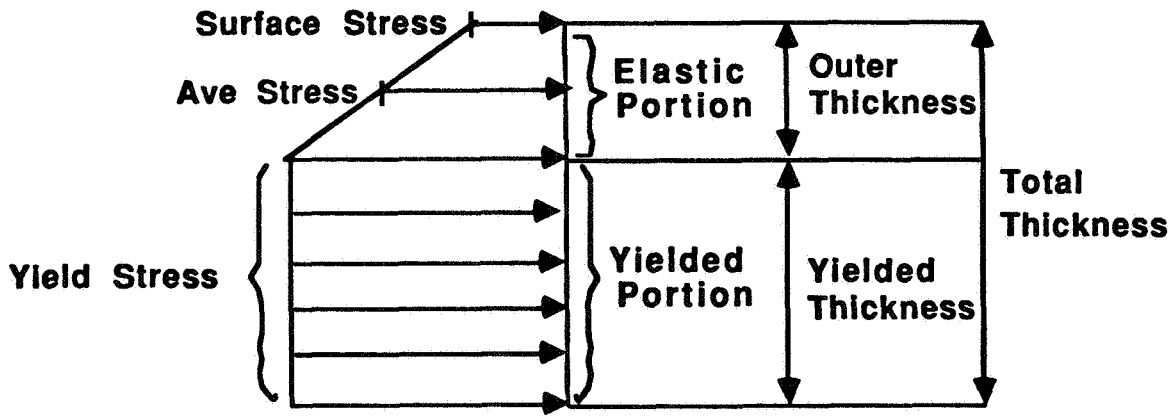

Figure B3: Moment unloads Compressive Thermal Stress

```
1  : PROGRAM cool_jacket
2  : ! test to find if cooling jacket wall under mechanical (moment) load
3  : ! and thermal (into yield) load is determinate
4  : !
5  :   yield_stress    = -30000 ! psi
6  :   Total_thickness = .020   ! inches
7  :   m_input         = .4     ! in-lb/in
8  :
9  :   OPTIMIZE strain_energy
10 :    USE
11 :    surface_stress = 0.0 IN [yield_stress+0.0001, abs(yield_stress)]
12 :    t_outer = 0.005 IN [0.0, total_thickness]
13 :    m_calculated .eq. m_input
14 :
15 :   OPTIONS
16 :     \normalize
17 :   END USE
18 :     ave_stress        = (surface_stress + yield_stress)/2    ! equation 1
19 :     m_calculated      = (surface_stress - ave_stress)*((t_outer**2)/6)
20 :    strain_energy     = (t_outer/(surface_stress - yield_stress))*
21 : &                       (surface_stress**3 - yield_stress**3)/3.
22 :   END OPTIMIZE
23 :  y = t_outer/total_thickness
24 :  PRINT 'Yielded part starts: ', y
25 : END cool_jacket
```

FIGURE B4 – SOL PROGRAM FOR COOLING JACKET EXAMPLE

Appendix C: Scramjet Engine Cooling Jacket Design Example

The SOL program for the design of the cooling jackets on an actively cooled structural panel is described in this appendix. This example is characterized by the use of SOL's *macro* feature to abbreviate sections of program text. Cooling jackets are required for environments so severe that passive insulation cannot keep a structure at a survivable temperature. As seen in Figure C1, a cooling jacket is a sandwich formed by two metal walls separated by webs which act as both spacers and cooling fins. One side of the sandwich is attached to the structural panel, and the heat flux is incident on the opposite side. A coolant is circulated in the channels between the fins to remove the incident heat flux. In the design of cooling jackets, the important constraints on the design are maximum metal temperature, maximum stress due to coolant pressure, and thermal fatigue of the jackets. Additionally, the coolant must satisfy pressure drop and Mach number constraints. The methods of analysis used to model these operating constraints of cooling jackets are described in reference 15. In this appendix, only the SOL program for the design of a single cooling jacket panel is described.

The SOL program which describes this cooling jacket design problem is given in Figure C2. The program has been annotated with numbers in parenthesis to facilitate the discussion. In overview, subroutine, macro and variable declarations are made in sections (3) – (5), key variables are initialized in (8), and the panel optimization occurs during steps (9) – (21).

The program begins with the word `PROGRAM` followed by the program name `coolchan`, as seen in (1). The `DECLARE` section at begins (2), ends at (6), and consists of subroutine, macro, and variable type declarations; a discussion of each declaration follows. The cool-

ing jacket analysis subroutine is declared in (3). After the word SUBROUTINE, the output parameters of the subroutine (it's "dependent variables") are listed, followed by an equals sign, followed by the subroutine's name, followed by the input parameters (it's "independent variables"). A property of SOL subroutines is that the subroutine cannot modify it's input parameters and must initialize its output parameters. This property is insured by SOL's error checking on the use of the subroutine. The variables defined as output variables represent: temperature of the hydrogen coolant exiting the panel, pressure of the hydrogen coolant exiting the panel, Mach number of the coolant exiting the panel, temperature of the outer wall of the cooling jacket at the panel exit, temperature of the inner wall of the cooling jacket at the panel exit, cooling jacket unit weight (not including the inner wall weight), worst value for normalized stress of the cooling jacket webs in the panel (stress greater than 1 is failed), worst value for normalized stress of cooling jacket outer face, worst value for thermal fatigue cycle life due to plastic strain, and worst value for thermal fatigue cycle life due to plastic strain including estimate of creep damage.

The subroutine input parameters which follow the subroutine name `channel_fin_sub` are: the number of segments into which the panel will be divided for analysis, the height of the channel webs, the spacing between webs, the thickness of the webs, the sandwich inner wall thickness, the sandwich outer wall thickness, the axial (i.e. in the flow direction) distance from an arbitrary fixed point at which the analysis begins, the corresponding heat flux, the axial distance at which the analysis ends, the corresponding heat flux (heat fluxes between the beginning and ending locations are linearly interpolated), the axial distance which corresponds to the inlet manifold location, the hydrogen coolant temperature corresponding to the axial position where the analysis begins, the corresponding coolant

pressure, and the coolant flowrate per unit panel width (i.e. in the cross flow direction).

After the subroutine declaration, a macro definition is given. Macros are text replacement procedures that execute at compile time. With a macro, a short word can stand for a longer sequence of text which is called, its *replacement text.* The replacement text can have multiple lines which could include other macro calls. When a macro appears in the text of the program, the SOL compiler replaces the macro name with its defined replacement text. This step is called macro *expansion.* Thus, macros can be thought of as abbreviations which are replaced with their definition at compile time. In their most complex form, macros can be defined to have *replaceable parameters*, which are analogous to subroutine parameters. Macros can be defined and expanded at any place in the program. For this example, the macro definition is placed in the declaration section for clarity and to facilitate discussion.

The macro, `?parallel_channels`, is used to abbreviate the analysis required for cooling jacket design. It has 5 parameters and additional *parameter text* to separate the parameters and make the meaning of the parameters clear (4). The parameter #1 names the panel section being designed, #2 and #4 identify the axial locations where the panel analysis begins and ends, and #3 and #5 are the corresponding heat flux values. The replacement text for the macro defines input variables for the cooling jacket analysis subroutine call, and calls this analysis routine. The output variables returned by the analysis subroutine have a suffix (given by parameter #1) appended to them in the macro replacement text. The input and output variable naming convention allows for analysis of arbitrary combinations of panels with linear heat flux variations by using a series of `?parallel_channels` calls. However, in this example, only a single macro call for analyz-

ing one panel with constant heat flux is used.

The declaration section of the SOL program continues with the declaration of variable types. Since REAL (64 bit precision) variables are the default type, only integer or logical variable types need be declared. The one type declaration here is an INTEGER type declaration for the number of sub-segments each panel is divided in the analysis subroutine (5). The declaration section ends with an END DECLARE statement (6).

A FORTRAN block (7) follows in which variables required by the external FORTRAN subroutines are given types and placed in a FORTRAN common block. Since SOL does not offer the common blocks or 32 bit precision REAL variables required by the subroutines, a FORTRAN block was necessary.

At (8), several variables used in the cooling jacket panel analysis are initialized: the crossflow dimension of the cooling jacket panel (panel_width), the thickness and density of the inner wall (wallthickin and innerdens), the coolant Mach number limit (max_coolant_mach), and the heat flux incident on the panel (heatrate).

The OPTIMIZE description begins with total_panel_flowrate declared as the objective function in (9). The design variables of flow rate and inlet pressure are given with initial values and the limiting values at (10). The design variables for the channel fin geometry of panel1 follow (11). The constraints in the optimization (12) are in the following order: peak outer wall temperature, peak normalized web stress, peak normalized outer wall stress, maximum coolant Mach number, minimum creep influenced fatigue life, and minimum coolant pressure drop of panel1. The OPTIONS section for the optimization follows with optimization print intervals declared (13), ADS strategy and one dimensional search options reset (14), the default ADS optimizer iteration limits increased (15), and

normalization of the design variables by the side constraints specified (16). The declaration of the design variables, constraints, and options ends with the `END USE` statement.

Analysis begins with setting the gas temperature and pressure entering the panel, and initializing the `total_weight` to zero (17). The panel location of the inlet manifold is set in `panelstart` (18). The analysis of the cooling jacket named "panel1" which is 75 inches long having a constant incident heat flux is performed by the macro call at (19). Most of the constraint functions declared in the above `USE` section are automatically initialized in this macro and the objective function and the pressure drop constraint are initialized at (20). This completes the analysis of the cooling jackets. The optimization statement ends at (21).

In the last lines of the SOL program, a `print` statement outputs the cooling jacket weight, the line `END coolchan` ends the main program, and the SOL subroutine declared at (3) (`channel_fin_sub`, which contains the interface to the `FORTRAN` analysis routines) is added to the source with the `?include` macro (22).

This example shows that macros enable a concise and understandable description of the cooling jacket design problem. Once the macro is defined in step (4), its usage in steps (17) through (20) is fairly clear; and is easily generalized to handle multiple panels. Note that in this appendix, only the SOL code which models the design problem is given; a detailed description of the models used to determine the constraint functions is given in reference 15.

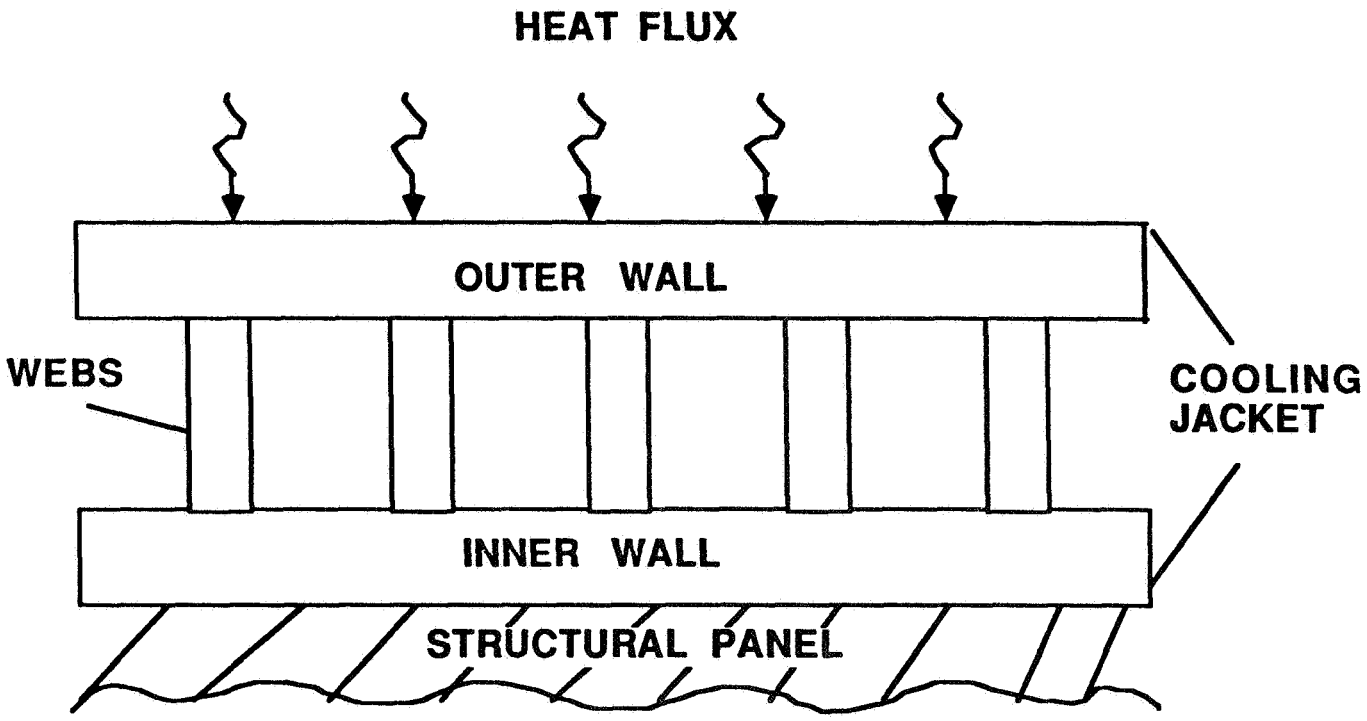

Figure C1: Cross-Sectional view of Cooling Jacket Sandwich

```
(1)  PROGRAM coolchan
     !   SOL program that designs a channel fin cooling jacket panel
     !
(2)  DECLARE
     !  ***** subroutines declaration begins
     !
(3)  subroutine (gas_t_out,gas_p_out,gas_mach_out,outerstr_t,innerstr_t,
     &           weight_per_sqft,normwebstress,nwallstress_at_web,
     &           pp_fatigue_life,pc_fatigue_life)
     & = channel_fin_sub (number_steps:integer,webheight,webspacing,webwidth,
     &           wallthickin,wallthickout,comp_begin,qbegin,comp_end,qend,origin,
     &           gas_t_in,gas_p_in,gas_flowrate)
     ! calculates channel fin heating over comp_begin -> comp_end
     !
     !  ***** subroutines declaration ends
     !
     !  ***** macro declaration begins
     !
     ! macro which analyzes one channel fin cooling jacket with coolant flow
     !  parallel to heating variation.
     !  x in in., q in btu/sqft-s., linear interpolation of q between begin and end
(4)  ?def?parallel_channels #1    begin x=#2 q=#3  end x=#4 q=#5  {
        qbegin       = #3          ! btu/s-sqft
        qend         = #5          ! btu/s-sqft
        webwidth     = webwidth_#1
        webspacing   = chanwidth_#1
        webheight    = chan_h_o_d_#1 * webspacing
        webspacing   = webspacing+webwidth
        wallthickout = wallthout_#1
        gas_flowrate = panel_flowrate/panel_width ! lb/in-s
        if(gas_p_in.lt.pmin) then ! do smooth,non-neg,additive map of press
           gas_p_in =-(pmin-pmapmin)**2/(gas_p_in-2.*pmin+pmapmin) + pmapmin
        endif
        flow_length = abs(#2-#4) ! inches
        origin      = panelstart ! inches
        comp_begin  = #2         ! inches
        comp_end    = #4         ! inches
     !  analysis subroutine channel_fin_sub
      (gas_t_out, gas_p_out, gas_mach_out, outerstr_t, innerstr_t,
     & weight_per_sqft, normwebstress, nwallstress_at_web,
     & pp_fatigue_life, pc_fatigue_life)
     & = channel_fin_sub(numbersteps, webheight, webspacing, webwidth,
     & wallthickin, wallthickout, comp_begin ,qbegin, comp_end, qend,
     & origin, gas_t_in, gas_p_in, gas_flowrate)
```

FIGURE C2 — SOL PROGRAM FOR A COOLING JACKET PANEL

FIGURE C2 *(continued)*

```
(4) (con't)
    weight_per_sqft        = weight_per_sqft + wallthickin * innerdens*144.
    weight_#1              = weight_per_sqft * flow_length * panel_width/144. ! flowlength -inches
    total_weight           = total_weight + weight_#1
    gas_t_out_#1           = gas_t_out
    gas_p_out_#1           = gas_p_out
    gas_mach_out_#1        = gas_mach_out
    outerstr_t_#1          = outerstr_t
    innerstr_t_#1          = innerstr_t
    weight_per_sqft_#1     = weight_per_sqft
    normwebstress_#1       = normwebstress
    nwallstress_at_web_#1  = nwallstress_at_web
    pp_fatigue_life_#1     = pp_fatigue_life
    pc_fatigue_life_#1     = pc_fatigue_life
    gas_p_in               = gas_p_out
    gas_t_in               = gas_t_out
  }   !   ***** macro declaration ends *****
  !
  !***** type declaration
  !
(5) integer numbersteps
(6) END DECLARE
(7) ! FORTRAN BLOCK for common and single precision real variables
    !  for pressure mapping
    /*
          real*4 pmin,pmapmin
          common/realpress/pmin,pmapmin
    \*
    !
    !** constants and fixed parameters
    !
(8) pmin              = 400.    ! psi, for pressure mapping
    pmapmin           = 300.    ! psi, for pressure mapping
    numbersteps       = 11      ! number of segments in a panel analysis
    panel_width       = 42.     ! inches
    wallthickin       =.010     ! default inner wall thickness
    innerdens         = .3      ! density inner wall (for weight calculation)
    max_coolant_mach  = .25     ! constraint: coolant maximum Mach number
    heatrate          = 100.    ! parameter
```

FIGURE C2 *(continued)*

```
      ! ***** panel optimization begins
(9)   OPTIMIZE total_panel_flowrate ! optimization will design to minimize flowrate
      USE     !***** optimization parameter declaration begins
      ! design variables
(10)  ┌Panel_Flowrate    =   3.00   IN  [ 1.000 , 4.0   ]
      │Inlet_Pressure    =   1000.0 IN  [ 1000. , 1500. ]
      └Chan_H_O_D_Panel1 =   0.5641 IN  [0.4    , 0.8   ]
(11)  ┌Chanwidth_Panel1  =   0.09   IN  [0.06   , 0.12  ]
      │Wallthout_Panel1  =   0.016  IN  [0.01   , 0.018 ]
      └Webwidth_Panel1   =   0.02   IN  [0.02   , 0.025 ]

      ! behavior constraints
(12)  ┌outerstr_t_panel1          .lt. 2000.
      │normwebstress_panel1       .lt. 1.
      │nwallstress_at_web_panel1  .lt. 1.
      │gas_mach_out_panel1        .lt. max_coolant_mach
      │pc_fatigue_life_panel1     .gt. 600.
      └pdrop                      .lt. 100.
      OPTIONS ! for optimization
      !** print options
(13)  ┌PRINT Design varaibles Initially, Every Iteration, At Termination
      │PRINT Objective Initially, Every Iteration, At Termination
      │PRINT Constraints Initially, Every Iteration, At Termination
      └PRINT Termination Criteria
      !** Optimizer choice
(14)  Strategy = SEQUENTIAL QUADRATIC
      Search   = GOLDEN SECTION
      !** ADS options
(15)  \itmax=80
      \jtmax=40
      !** design variable normalization
(16)    \normalize
      END USE !***** optimization parameter declaration ends
      !
      !***** analysis begins
      !
      ! initialize before series of panels
(17)    gas_p_in     = inlet_pressure ! initial gas pres. (psi)
        gas_t_in     = 1000           ! initial gas temp. (deg r)
        total_weight = 0.0            ! initialize total weight
        !
        ! panel 1
        !
(18)    panelstart = 0.        ! x distance for start of coolant film
```

FIGURE C2 *(continued)*

```
(19)  ?parallel_channels panel1
                         begin x=0.   q= heatrate
                         end   x=75.  q= heatrate
(20)    total_panel_flowrate = panel_flowrate
        pdrop                = inlet_pressure - gas_p_out
      !
      !***** analysis ends
      !
(21)  END OPTIMIZE
      !
      !***** panel optimization ends
      !
      print 'total weight', total_weight

      END coolchan

      ! analysis subroutine
(22)  ?include sdmchannel.sub
```

References

[1]Schmit, L. A.: Structural Synthesis — Its Genesis and Development. *AIAA J.*, vol. 19, no. 10, Oct. 1981, pp. 1249-1263.

[2]Ashley, H.: On Making Things Best — Aeronautical Uses of Optimization. *J. Aircr.*, vol. 19, no. 1, Jan. 1982, pp. 5-28.

[3]Vanderplaats, G.N.: *CONMIN - A FORTRAN Program for Constrained Function Minimization—User's Manual.* NASA TM X-62282, 1973.

[4]Vanderplaats, G.N.: *ADS - A FORTRAN Program for Automated Design Synthesis — Version 1.10.* NASA Contractor Report 177985, Grant NAG1-567, 1985.

[5]Gill, P.E.; Murray, W.; Saunders, M; and Wright, M.: User's Guide for NPSOL (Version 4.0): a FORTRAN Package for Nonlinear Programming. *To be used with:* User's Guide for LSSOL (Version 1.0): A FORTRAN Package for Constrained Linear Least-Squares and Convex Quadratic Programming. Systems Optimization Laboratory, Stanford University. January 1986. *Available from the Stanford Office of Technology Licensing, 950 Cambridge Avenue, Suite 250, Palo Alto, California 94306, USA.*

[6]Balling, R.J.; Parkinson, A.R.; and Free, J.C.: Experience with OPTDES.BYU; An Interactive Package with Optimization and 2D/3D Graphics Modules. Proc. NASA Symposium on Recent Experiences in Multidisciplinary Analysis and Optimization, NASA Langley Research Center, April 24-26, 1984.

[7]Lassez, C.: Constraint Logic Programming – A New General Framework for Developing Languages More Powerful than Traditional Logic Programming Languages. *BYTE*, vol.12, no. 9, August, 1987, pp. 171–176.

[8]Heintze, N.C.; et. al.: CLP(R) and Some Electrical Engineering Problems. Proceedings of the 4^{th} International Conference on Logic Programming, Melbourne, 1987; Monash University Technical Report no. 73, June 1986.

[9]Balling, R.J.; Parkinson, A.R.; and Free, J.C.: Methods for Interfacing Analysis Software to Optimization Software. *Computers & Structures*, vol 22, no. 1, 1986, pp. 87–98.

[10]Anderson, M.S.; and Stroud, W.J.: A General Panel Sizing Computer Code and Its Applications to Composite Structural Panels. *AIAA J.*, vol. 17, no. 8, August 1979, pp. 892–897.

[11]Collins, W.R.; and Noonan, R.E.: *The MYSTRO System: A Comprehensive Translator Toolkit.* Final Report, NASA Langley Research Center, Grant NSG-1435, 1985.

[12]Lucas, S. H.; and Scotti, S.J. *The SOL Reference Manual.* NASA Technical Memorandum 100566, 1988.

[13]Rogers, J.L.; and Barthelemy, J.M. *An Expert System for Choosing the Best Combination of Options in a General-Purpose Program for Automated Design Synthesis.* NASA Technical Memorandum 86329, March 1985.

[14]Fox, R.L.; *Optimization Methods for Engineering Design.* Addison-Wesley Publishing Co., 1971.

[15]Scotti, S.J.; Martin, C.J.; and Lucas, S.H.: *Active Cooling Design for Scramjet Engines Using Optimization Methods.* NASA Technical Memorandum 100581, March 1988; *also appears in:* Proceedings of the 29^{th} Structures, Structural Dynamics and Materials Conference, Williamsburg, Va., AIAA Paper No. 88-2265, April 18-20 1988.

NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-100565		

4. Title and Subtitle	5. Report Date
The Sizing and Optimization Language, SOL -- Computer Language for Design Problems	April 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Stephen H. Lucas and Stephen J. Scotti	
	10. Work Unit No. 505-63-31-03

9. Performing Organization Name and Address	11. Contract or Grant No.
NASA Langley Research Center Hampton, VA 23665-5225	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, DC 20546-0001	14. Sponsoring Agency Code

15. Supplementary Notes

Stephen H. Lucas, Vigyan Research Assoc., Inc. Stephen J. Scotti, Thermal Structures Branch, Loads and Aeroelasticity Division, NASA Langley Research Center, Hampton, VA. Presented at 29th Structures, Structural Dynamics and Materials Conference, Williamsburg, VA. 18-20 April 1988.

16. Abstract

The Sizing and Optimization Language, SOL, a new high-level, special-purpose computer language has been developed to expedite application of numerical optimization to design problems and to make the process less error-prone. SOL utilizes the ADS optimization software(NASA CR 177985) and provides a clear, concise syntax for describing an optimization problem, the OPTIMIZE description, which closely parallels the mathematical description of the problem. SOL offers language statements which can be used to model a design mathematically, with subroutines or code logic, and with existing FORTRAN routines. In addition, SOL provides error-checking and clear output of the optimization results. Because of these language features, SOL is best-suited to model and optimize a design concept when the model consists of mathematical expressions written in SOL. For such cases, SOL's unique syntax and error-checking can be full utilized. SOL is presently available for DEC VAX/VMS systems. A SOL package is available which includes the SOL compiler, runtime library routines, and a SOL reference manual.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Optimization, nonlinear mathematical programming, computer languages, design tools.	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	43	A03