SLANG
A PROBLEM SOLVING LANGUAGE
FOR
CONTINUOUS-MODEL SIMULATION
AND OPTIMIZATION

By Joe M. Thames, Jr.

TRW Systems Group
One Space Park
Redondo Beach, California

INTRODUCTION

SLANG is a mathematical problem modeling and solution language. It is one of several languages in the programming subsystem of the Computer User Executive (CUE) System developed by TRW Systems. SLANG is both a procedural and a command language designed primarily for the "casual" user. Consequently, much attention was paid to programming ease, "natural" syntax rules, readability, and debugging ease. On the other hand, SLANG is designed to permit the solution of very sophisticated mathematical problems, characterized by iterative solution methods. Its translated object code therefore contains complex numerical *solution logic* in addition to the object code of its procedural syntax.

The solution logic is generated by the presence of language commands such as "SOLVE" for the solution of simultaneous nonlinear algebraic equations; and "MINIMIZE" for finding the minimum of a function of several variables. For iterative methods, like both of the above, the compiled code will compute partial derivatives from the procedural formulas, of the objective function and/or constraints with respect to designated independent variables, as needed by the solution algorithm.

In addition to the above features, SLANG is user-extensible. The user may program macro operators using a SLANG macro facility; or he may program relocatable SLANG or Fortran subroutines, and may define calling statement syntax for either macros or subroutines using a *syntax macro processor*. Thus SLANG may be augmented and tailored to fit individual user needs.

Language Philosophy

A major indication of the simplicity and "naturalness" of a programming language is its readability. To be easily read, it must conform largely to the structural rules of ordinary written English. First of all, there is a single main thought stream of written text from which all digressions are temporary and which always resumes at a point immediately subsequent to the digression. Footnotes, and asides, have this character in English, as subroutines and macros do in programming languages. Secondly, in English there is no counterpart to the direct (non-return) transfer; consequently, its extensive use in programming language leads to reading difficulty.

SLANG syntax is designed so that direct transfers are largely unnecessary, although transfer statements are provided. In keeping with the pattern of digression with subsequent resumption, SLANG statements which "open" the main thought stream (e.g. conditional statements and cycling statements) have associated "closing" keywords, such as REJOIN, which resume it again.

SLANG statements are free field, allowing ample use of indentation; and SLANG operators are free of unnecessary delimiters; but blanks, as in English, may have significance as separators. The use of subordination through indentation and the use of statement keywords as group labels permits complex logical constructions to be easily coded or read. For example, the SLANG conditional statement may be used to describe a decision tree in a highly readable manner:

```
IF  <Conditional>
    THEN IF <Conditional>
            THEN <Statement>
                    "
                    "
                    "
            ELSE <Statement>
                    "
                    "
                    "
         REJØIN
    ELSE IF <Conditional>
            THEN <Statement>
                    "
                    "
                    "
            ELSE <Statement>
                    "
                    "
                    "
         REJØIN
REJØIN
```

Each branch of the decision tree is clearly labeled by the keywords THEN and ELSE, subordination is accomplished by indentation, and each conditional statement is "closed" by the keyword REJØIN. There is no need for unconditional transfers, and there is no ambiguity.

Problem Solving Features

Problem Model Structure

Mathematical models, in general, are composed of a set of *independent* variables, a set of *dependent* variables, and the functions, explicit or implicit, that relate them. Problems may be characterized as *direct* or *indirect*, pertaining to whether the unknowns of the problem are dependent variables or independent variables, respectively. The correspondence may be extended to solution methods, which involve direct computations or iterative (indirect) computations. Of course, compound models may be a mixture of direct and indirect problems, thus calling for a mixture of direct and iterative solution computations.

Procedural languages generally do not provide built-in methods for solving indirect problems, or even complex direct problems. They only provide procedural statements that can be used to construct algorithms for solving such problems.

Direct Methods - Procedural languages do provide built-in features for treating the simplest direct problems: *automatic parsing* for arithmetic replacement formulas; *function subprograms* for single-unknown multi-formula functions; and *procedure subprograms* for multi-unknown, multi-formula functions. All of these features serve to simplify the solution of direct problems because they implicitly handle the bothersome "mechanical" tasks which the user takes for granted, and they permit large problems to be treated as a group of individual smaller problems. However, for direct problems which involve *secondary* computations (e.g. numerical integration), procedural languages provide no built-in methods because such methods require intervening execution of selected parts of a problem model, and therefore must control the flow of the program. This has given rise to a number of simulation languages such as DSL/90, CSSL, and CSMP which are *structurally compatible* with the solution process of numerical integration in addition to having procedural capabilities.

Iterative Methods - Strictly procedural languages offer little capability for indirect problem solving because not only do such problems require secondary computations, i.e. partial derivatives, but to be foolproof such quantities should be computed from exact rather than approximate formulas. This requires that the language processor derive the secondary computation logic from the structure of the algebraic formulas in the model.

The simplest class of such problems, although far from simple, is the solution of a determined set of nonlinear algebraic equations:

$$g_1 \ (X_1, X_2 \ \ldots \ X_n) = 0$$

$$g_2 \ (X_1, X_2 \ \ldots \ X_n) = 0$$

$$\vdots$$

$$g_n \ (X_1, X_2 \ \ldots \ X_n) = 0$$

Such a system is said to have zero degrees of freedom because the number of unknowns X (independent variables) equals the number of equality constraints g (dependent variables). Several methods exist for solving such problems, the most general of which is probably the Newton-Raphson method. This method requires that the Jacobian matrix

$$\begin{bmatrix} \frac{\partial g_1}{\partial X_1} & \cdots & \frac{\partial g_1}{\partial X_n} \\ \vdots & & \vdots \\ \frac{\partial g_n}{\partial X_1} & \cdots & \frac{\partial g_n}{\partial X_n} \end{bmatrix}$$

be computed for each iteration.

Problems which have more unknowns than equality constraints, i.e. problems with one or more degrees of freedom are optimization problems, because no longer are there a finite number of solutions; and meaningful solutions can only be obtained by imposing criteria which fix the values of the independent variables as well as satisfy the constraints. The optimization criteria which accomplish this are

$$\frac{\partial f}{\partial X_1} = \frac{\partial f}{\partial X_2} = \ldots \frac{\partial f}{\partial X_n} = 0$$

where f is an arbitrary *objective* function of $X_1 \ldots X_n$, which is to be optimized.

A common requirement for a general capability for optimization and nonlinear equation solving is a mechanism for computing partial derivatives with respect to designated independent variables. Such a mechanism is a primary feature of SLANG. It permits complex indirect problems to be solved with equivalent ease of numerical integration.

Command Structures

SLANG's built-in problem solving capabilities are implemented by the user through statement groups called command structures. Statements which make up command structures are of three types: commands, command specifications and command declarations. A command structure is a specific sequence of such statements which specifies and executes the solution of a complete mathematical problem. Composite SLANG programs may therefore contain several command structures, one for each complex problem in the makeup of the overall program.

Command specifications are used to select and initialize numerical algorithms from the SLANG method library. Currently SLANG contains three specifications, INTEGRATION, OPTIMIZATION and CONTROLS.

Commands are SLANG statements that invoke the execution of algorithms which have been previously selected and initialized by the appropriate specification. The primary SLANG commands are: INTEGRATE, OPTIMIZE, MAXIMIZE, MINIMIZE, STEP OPTIMIZE, STEP MINIMIZE, STEP MAXIMIZE, PARTIALS, FIRST PARTIALS, NO PARTIALS, and SOLVE.

Command declarations are used to identify variables that have special significance in a given command structure (e.g. independent variables). The primary declarations are INDEPENDENT(S), CONSTRAINT(S), and LAMBDA(S).

Associated with every command structure there is a group of formulas which comprise the mathematical problem to be solved. This group of formulas is called the command-model. It may be coded immediately within the command structure or may reside in SLANG subprograms which are executed within the command structure. The following paragraphs treat various command structures.

Numerical Integration - The command structure for numerical integration is composed of an INTEGRATION specification followed by subsequent occurrences of the INTEGRATE command. The INTEGRATION specification takes the following form:

```
INTEGRATION (<method>) <block>, <x>, <Δx>,
*           <y>,<dy1/dx>, ..., <yn>, <dyn/dx>
```

where <*method*> is the name of a SLANG library algorithm to be used for integration, and <*block*> is the name of the block (internal subprogram) containing the differential equations

$$\frac{dy_1}{dx} = f_1\ (y_1, \ldots, y_n, x)$$

$$\vdots$$

$$\frac{dy_n}{dx} = f_n\ (y_1, \ldots, y_n, x)$$

to be integrated, and <Δx> is the step size for integration variable <x>. INTEGRATION, when encountered in the execution stream, initializes the integration process at the current values of the quantities $x, y_1, \ldots, y_n$. If perturbations are made to these quantities during integration, the equations must be reinitialized by another instance of the INTEGRATION specification.

The INTEGRATE command is used to perform step by step integration of the differential equations specified. INTEGRATE takes the form

```
INTEGRATE <block>
```

where <*block*> is the name of a previously specified block.

Example:

```
INTEGRATIØN DERIVS, X, DX, Y1, DY1DX, Y2,
*           DY2DX, Y3, DY3DX
            :
DO UNTIL X GE XEND
INTEGRATE DERIVS
REPEAT
            :
BLØCK DERIVS
Y = 1 + EXP (X/Z)
DY1DX = Z * Y ** 1.5/ (A * Y1 + B * Y1 ** 2)
DY2DX = Y1 * Y2 + Y3 * Y2 ** 2
DY3DX = Y * Y3 + Y1 * Y2

END BLØCK
            :
```

In this example, no algorithm was specified in the INTEGRATION specification. This would cause a "nominal" algorithm to be used. The complete command structure for integration consists of the INTEGRATION specification, the block containing the model to be integrated, and the INTEGRATE command.

Nonlinear Algebraic Equations - The command structure for the solution of nonlinear equations is called a solve loop. It belongs to the iterative class of command structures known as *command loops* which begin with a command, e.g. SOLVE, and terminate with the END LOOP command. The solve loop has the following structure.

SØLVE $\langle g_1 \rangle, \langle g_2 \rangle, \ldots, \langle g_n \rangle$

VARY $\langle X_1 \rangle, \langle X_2 \rangle, \ldots, \langle X_n \rangle$

CØNTRØLS $\langle C_1 \rangle$ *Value*, ... $\langle C_n \rangle$ *Value*

Loop Model { SLANG Statements defining $\langle g_1 \rangle, \ldots, \langle g_n \rangle$ as functions of $\langle X_1 \rangle, \ldots, \langle X_n \rangle$

END LØØP

The $\langle g_1 \rangle, \ldots, \langle g_n \rangle$ are the names of the equality constraints which identify the nonlinear equations

$$g_1 \ (X_1, \ldots, X_n) = 0$$

$$\vdots$$

$$g_n \ (X_1, \ldots, X_n) = 0$$

to be solved.

The VARY (or INDEPENDENT) declaration declares $\langle X_1 \rangle, \ldots, \langle X_n \rangle$ as the independent variables to be determined in the process. These quantities must be initialized (guessed) prior to entry into the SOLVE loop. The VARY declaration sets up the automatic computation of partial derivatives. That is, all variables subsequently computed in the SOLVE model will "acquire" first partial derivatives with respect to $\langle X_1 \rangle, \ldots, \langle X_n \rangle$. These quantities are used by the numerical algorithm, a Newton-Raphson method, to iteratively solve the nonlinear equations.

The CONTROLS specification serves to define controls to be imposed on the solution algorithm, i.e. iteration limits, convergence tolerances, bounding constraints, etc. The $\langle c_1 \rangle$'s are control keywords pertaining to the particular algorithm. CONTROLS is optional, since each algorithm contains a nominal set of built-in controls.

Example:

Given: $A^2 + B^2 = 10 + C$

$A + B + C^2 = 16$

$A = B + 3$

Solve for: A, B, C

Equality constraints would be defined as:

EQ1 = $A^2 + B^2 - (10 + C)$ ⟶ 0

EQ2 = $A + B + C^2 - 16$ ⟶ 0

EQ3 = $A - B - 3$ ⟶ 0

The solve loop would look like:

initial guesses { A = 1, C = 1, B = A - 3

```
SØLVE EQ1, EQ2, EQ3
    VARY A, B, C
    DV1 = A ** 2 + B ** 2
    EQ1 = DV1 - C - 10
    DV2 = A + B - 16
    EQ2 = DV2 + C ** 2
    EQ3 = A - B - 3
END LØØP
```

As there are undoubtedly a number of solutions to such a set of equations (some imaginary), which solution is found depends upon the initial guess. It is, of course, up to the user to supply initial guesses which will converge to the solution he is looking for. For most physical problems, this presents little difficulty because the user generally has some idea about what the answers should be.

Nonlinear Optimization - SLANG provides several algorithms and macro-algorithms for nonlinear optimization. Each may be implemented through a command loop for optimization, i.e. an *optimization loop* which has the following structure:

$\langle$*optimization command*$\rangle$ $\langle$*obj*$\rangle$

INDEPENDENT $\langle X_1 \rangle, \ldots, \langle X_n \rangle$

$\langle$*command-declarations*$\rangle$ $\langle$*argument list*$\rangle$

"

"

$\langle$*statements*$\rangle$

"

"

$\langle$*obj*$\rangle$ = $\langle$*expression*$\rangle$

END LØØP

where $\langle$*obj*$\rangle$ is the name of the payoff function to be optimized and $\langle X_1 \rangle, \ldots, \langle X_n \rangle$ are the independent variables. In addition, a preceding OPTIMIZATION specification is necessary if the "nominal" algorithms associated with each command are to be replaced. The OPTIMIZATION specification has the form:

OPTIMIZATION (*<method>*)

CONTROLS *<c_1> value, ..., <c_n> value*

optional

The CONTROLS specification (as a separate statement) may optionally be included as one of the command declarations in the heading of the optimization loop. Its arguments are keywords and numerical values for the respective controls pertaining to a given algorithm.

Various "local" optimization methods are available in the SLANG library. Each one is specifically designed to utilize the partial derivatives that are invoked by the INDEPENDENT declaration. All computed quantities occurring within the optimization loop automatically acquire computed partial derivative values with respect to *<X_1>, ..., <X_n>*. If the optimization method is a second order method, then both first and second partials are computed.

Different "nominal" algorithms are associated with various optimization commands. The OPTIMIZE command is nominally a Newton-Raphson method which uses a minimum-norm, least-squares-pseudoinverse matrix inverter. It does not discriminate between minima, maxima, or saddle points, but simply finds the nearest critical point. The MINIMIZE, MAXIMIZE and CRITICALIZE commands are associated with a direction discriminating algorithm which uses the eigenvalues and eigenvectors of the second partials matrix to compute the "best" search direction.

The "STEP" commands, STEP OPTIMIZE, STEP MINIMIZE, etc., invoke macro algorithms (SLANG macros involving OPTIMIZE, MAXIMIZE, etc.) in which the search direction (once selected) is held fixed until a local extremum is found in that direction. Each fixed direction is therefore a composite "step", of the overall search, in which the payoff function is optimized with respect to the stepping interval in that direction. These algorithms demonstrate superior convergence capability for non-quadratic problems.

Constraints - In an optimization process, constraints of two kinds may be imposed: equality or inequality constraints. Equality constraints are generally used to characterize a property of the modeled process, whereas inequality constraints usually serve to limit the domain of variation of the independent variables.

Equality constraints take the form of implicit algebraic equations which are driven to zero during the optimization process. SLANG permits the constraint matching process to be computed serially through the use of a nested solve loop, or in parallel, using the method of Lagrange.

Optimization with serial constraint matching may be accomplished using the following optimization loop structure:

```
<optimization command>  <obj>
    INDEPENDENT <X1>, ..., <Xn>
    CØNTRØLS  <C1> value, ..., <Cn>  value
        :
    <y1>= <expression>
    :
    <ym>= <expression>
    SØLVE <g1>, ..., <gm>
        VARY  <y1>, ..., <ym>
          :
          <g1> =  <expression>
          :
          <gm> =  <expression>
          :
    END LØØP
    :
    <obj>  =  <expression>
    :
END LØØP
```

This method may often be used to some advantage over the method of Lagrange, because each constraint matched in this manner removes an independent variable (removes a degree of freedom) from the optimization process and requires no initial estimation of Lagrange multipliers. However, because iterations are nested, it will tend to be computationally less efficient.

In the method of Lagrange, the following equations are solved iteratively to locate a local extremum of the function F:

$$\frac{\partial F}{\partial X_1} + \lambda_1 \frac{\partial G_1}{\partial X_1} + \ldots + \lambda_m \frac{\partial G_m}{\partial X_1} = 0$$

$$\vdots$$

$$\frac{\partial F}{\partial X_n} + \lambda_1 \frac{\partial G}{\partial X_n} + \ldots + \lambda_m \frac{\partial G_m}{\partial X_n} = 0$$

$$G_1 (X_1, \ldots, X_n) = 0$$

$$\vdots$$

$$G_m (X_1, \ldots, X_n) = 0$$

where $\lambda_1, \ldots, \lambda_m$ are the Lagrange multipliers which correspond to each constraint. Lagrange multipliers are unknown constants that are solved in addition to the independent variables $X_1, \ldots, X_n$, during the iterative optimization process. Consequently, initial guesses of the Lagrange multipliers must be supplied along with initial values of the independent variables.

The optimization loop for Lagrange's method has the form:

```
<optimization command> <obj>
CØNSTRAINTS  <g1>, ..., <gm>
LAMBDAS      <λ1>, ..., <λm>
INDEPENDENT  <x1>, ..., <xn>
      :
      <g1> =  <expression>
      :
      <gm> =  <expression>
      :
      <obj>=  <expression>
      :
END LØØP
```

If the user has difficulty in estimating Lagrange multipliers, he may ignore them and SLANG will compute good estimates by making a single iteration of the optimization loop with extra generated constraints to remove all degrees of freedom.

This built-in estimation procedure takes advantage of the fact that when the number of independent variables equals the number of constraints, the optimization problem is reduced to one of solving the nonlinear constraint equations.

One may also take advantage of this fact to solve *nested* nonlinear equations, using a variation of the above structure combined with a nested solve loop:

```
CØNSTRAINTS <g1>, ..., <gn>
INDEPENDENT <x1>, ..., <xn>
   :
   SØLVE <h1>, ..., <hm>
   VARY  <y1>, ..., <ym>
         :
         <h1> =  <expression>
         :
         <hm> =  <expression>
   END LØØP
   :
   <g1> =  <expression>
   :
   <gn> =  <expression>
   :
END LØØP
```

Inequality constraints are also permitted in SLANG optimization loops. They are treated as single barrier constraints with zero as the nominal barrier; they are identified in a constraints declaration by the presence of a plus sign or minus sign following the constraint name to signify that the constraints may take only positive and zero values or negative and zero values.

Example:

```
CONSTRAINTS G1+, G2-, G3, G4
```

In this example G1 and G2 are inequality constraints, whereas G3 and G4 are taken to be equality constraints. Both equality and inequality constraints are represented in the same way in an optimization loop model.

Example:

Given the above constraint declaraction in the heading of a command-loop, suppose it is desired to limit the varation of some independent variable, X, between the limits 32 and 212, then two inequality constraints G1 (+) and G2 (-) would be used.

G1 = X - 32 (positive or zero values only)

G2 = X - 212 (negative or zero values only)

Partial Derivatives - SLANG permits the user to invoke the computation of partial derivatives of computed formulas with respect to any designated set of independent variables through the use of the PARTIALS, FIRST PARTIALS and NO PARTIALS commands. The command structure for computing partials has the following form:

```
PARTIALS <x1>, ..., <xn>
   :
   <statements>
   :
NØ PARTIALS
```

or,

```
PARTIALS
    INDEPENDENT <x1>, ..., <xn>
        :
        <statements>
        :
NØ PARTIALS
```

The statements lying between PARTIALS and NO PARTIALS are said to be within the *domain of independence* of the variables $x_1, \ldots, x_n$. Of course the same idea holds true within an optimization loop or a solve loop, because partial derivatives are computed with respect to the independent variables of the optimization or equation solving process. However, a command structure for partials may also occur within an optimization loop or a solve loop for the computation of additional partial derivatives that are required for another purpose, i.e.

```
SØLVE  <g1>, ..., <gn>
    VARY <x1>, ..., <xn>
    :
    PARTIALS <y1>, ..., <ym>
        :
        <statements>
        :
        <g1> =  <expression>
        :
        <gk> =  <expression>
        :
    NØ PARTIALS
        :
    <gk+1> =  <expression>
        :
    <gn>   =  <expression>
        :
END LØØP
```

In this structure, first partials of all computed variables in the solve loop will be computed with respect to the variables $x_1, \ldots, x_n$. In addition, first and second partials of all variables computed between PARTIALS and NO PARTIALS will be computed with respect to $y_1, \ldots, y_m$. In particular, the constraints $\langle g_1 \rangle, \ldots, \langle g_k \rangle$ would acquire first and second partials with respect to $\langle y_1 \rangle, \ldots, \langle y_m \rangle$ as well as first partials with respect to $\langle x_1 \rangle, \ldots, \langle x_n \rangle$; whereas the constraints $\langle g_{k+1} \rangle, \ldots, \langle g_m \rangle$ would only acquire first partials with respect to $\langle x_1 \rangle, \ldots, \langle x_n \rangle$.

The computed partials may be assigned to specific variables using the DERIV function,

LET $\langle \frac{\partial y}{\partial x} \rangle$ = DERIV (<y> / <x>)

or,

LET $\langle \frac{\partial^2 y}{\partial x \partial z} \rangle$ = DERIV (<y>/ <x>, <z>)

To assign the *vector* of partials of a variable with respect to all of the *active* independent variables, the subroutines PAR1 and PAR2 may be used:

```
CALL PAR1 (Y,Z)
```

will assign the first partials of Y to the vector Z. The order of partials in the Z vector will correspond to the sequential order in which the active independent variables were declared (through combinations of the INDEPENDENT, VARY, PARTIALS or FIRST PARTIALS statements). The PAR2 subroutine will assign second partials to a vector, but the order of partials will correspond to a linear array representation of the upper triangular second partials matrix.

Discussion

The SLANG command structures and their associated execution logic, were designed as building blocks for analysis program development. Each command structure provides a formulation and solution framework for a fundamental problem of mathematical analysis. They can be easily combined to solve more sophisticated problems which can be represented as composites of the respective fundamental problems. For example, implicit ordinary differential equations and multipoint boundary value problems are both composites of ordinary differential equations and implicit algebraic equations; thus they can be readily treated through the use of INTEGRATION structures and SOLVE loops.

Example - The following is a relatively simple example of the use of SLANG to formulate a program for the nonlinear two point boundary value problem

$$\ddot{y} = -(1 + e^{y})$$

$$y(o) = o, \; y(1) = 1$$

This example was taken from Reference 1, in which it was formulated as an example of CSSL to compare with a previous formulation of the same problem in MIDAS III (Reference 2).

The object of the problem is to determine the unknown initial condition $\dot{y}(o)$ such that the terminal condition is met. The iterative method

formulated in CSSL is a Newton-Raphson procedure which uses the partial derivative $u(t) = \partial y(t)/\partial \dot{y}(o)$ computed by integrating the variational equation:

$$\ddot{u} = -(e^{y})\,u$$

$$u(o) = 1,\ \dot{u}(o) = 1$$

in parallel with the equation for $\ddot{y}$.

This problem is much easier to formulate in SLANG, because the SOLVE algorithm is a Newton-Raphson method, and partial derivatives are automatically computed. Thus there is no need to formulate the numerical algorithm or the variational differential equation. The SLANG formulation is as follows:

```
      DYDTØ = 1
      SØLVE GY1
          VARY DYDTØ
          T = 0
          Y = 0
          DYDT = DYDTØ
          INTEGRATIØN DERIVS, T, .05, DYDT, DY2DT2,
*             Y, DYDT
          DØ UNTIL T GE 1
              INTEGRATE DERIVS
          REPEAT
          GY1 = Y-1
      END LØØP
PRINT VARIABLES
BLØCK DERIVS
      DY2DT2 = -(1 + EXP(Y))
END BLØCK
STØP
END
```

The INTEGRATION statement defines the two derivatives DY2DT2 and DYDT, which must be integrated to solve the second order differential equation for DYDT and Y respectively. The solution to determine DYDTØ is found using a SOLVE loop, with DYDTØ as independent variable, in which the differential equation is integrated from T = 0 to T = 1. Convergence is satisfied by the constraint GY1 which satisfies Y(1) = 1.

One could just as easily have formulated a system of differential equations with two point conditions by adding more differential equations in the DERIVS block and the INTEGRATION specification; and adding a corresponding set of independent variables and algebraic constraints in the SOLVE loop. Moreover, the entire construction could be imbedded within an optimization loop to solve a parameter optimization or optimal control problem.

Numerical Experimentation -One major use of SLANG as a program development tool is in the formulation and trial of numerical algorithms. The PAR1 and PAR2 subroutines provide access to analytic partials that may be computed from any set of formulas. Fortran subroutines, SLANG subroutines, and SLANG macros may be constructed and called from SLANG programs. The fundamental SLANG algorithms may be augmented by any and all of these facilities for the generation of special methods to fit any problem. Several composite algorithms for optimization (including the "STEP" macro-algorithms) have been developed in SLANG by such an evolutionary process of experimentation. The total amount of effort involved was considerably less than would have been expended using a procedural language alone.

Procedural Features

SLANG procedural syntax is a compatible mixture of the "best" features of several extant languages; including FORTRAN, MAD, Algol 60, and SIMSCRIPT. SLANG assignment statements and subroutine calls are identical to Fortran. The SLANG conditional statements are a mix of Algol syntax and MAD structural rules. And the cycling statements have SIMSCRIPT origin. On the other hand, some additional features such as the indexed GØ TØ statement, may be original.

General Statement Features

Labels - Statement labels may be either integers, as in Fortran, or alphanumeric names appended to dollar signs, e.g. $BEGIN, $STATEMENT100, $12N, etc. A particular statement label, $NØDEn, where n is any integer, has particular significance with regard to the indexed GØ TØ statement, treated below. The end of a statement label is delimited by one or more trailing blanks.

Continuation and Comments - Statement continuation is signified by an asterisk (*) appearing in the first column of a line. A line may be treated as a comment by the presence of a slash (/) in the first column. If another slash is placed in the second column, then the compiler will skip to the next page before printing. Thus portions of code may be spaced arbitrarily in the compiler printout.

Identifiers - SLANG identifiers may be of any length, but identifiers of greater than six characters will be truncated. The internal representation of SLANG identifiers may be no more than six characters.

Debugging Statements

The user may code SLANG source statements for debugging which may be deactivated in later processing by the presence of single commands.

Debug Label - Any SLANG statement may be identified as a debugging statement by preceding it with the word DEBUG. Once program checkout has been completed, all such statements may be deleted by insertion of the command DELETE DEBUG at the beginning of the source program.

Trace Statements - The user may trace the computation process in a segment of code by placing the statements TRACE and NO TRACE before and after the segment. This will cause computed quantities, referenced by SLANG names and source line numbers, to be immediately printed. TRACE statements may also have DEBUG labels, thus they may be deleted using the DELETE DEBUG statement when no longer needed.

Control Statements

Conditional Branching Statements - As previously illustrated, the SLANG IF statement has the form

IF <*conditional expression*>
THEN
{*one or more statements*}
optional { ELSE {*one or more statements*} }
REJØIN

If the conditional is satisfied, the THEN branch is taken, followed by control transfer to the statement following REJOIN. Otherwise, the ELSE branch is executed. Any legal SLANG statements may be used in either branch. However, if a command-loop heading statement appears, then the rest of the heading statements must appear also. If both the THEN branch and the ELSE branch contain only one statement, then the IF statement may be a single statement without REJOIN, i.e.

IF <*conditional expression*>
* THEN <*statement*>
* ELSE <*statement*>

Conditional Cycling Statements - Conditional cycling may be accomplished with constructions of the form:

DØ [WHILE / UNTIL] <*conditional expression*>
⋮
optional { EXIT <*statement label*> } (*optional*)
AFTER <*expression*> LØØPS
⋮
REPEAT

DO WHILE causes transfer to the statement following REPEAT if the conditional expression is not satisfied, whereas DO UNTIL causes transfer if the conditional is satisfied.

Conditional Expressions - Conditional expressions are made up of relational expressions connected by the logical operators ØR and AND. Relational expressions are of the form:

<*arith. expr.*><*relational operator*><*arith. expr.*>

or,

(<*arith.expr.*>) <*relational operator*> (<*arith. expr.*>)

In the first form, one or more blanks must separate arithmetic expressions from relational or logical operators. The relational operators are:

EQ, NE, GT, LT, GE, LE *for* =, ≠, >, <, ≥, ≤

The Unconditional Cycling Statement - The SLANG equivalent of the Fortran DØ statement is:

DØ FØR <*ident.*> = <*arith. expr.*> TØ
* <*arith. expr.*> STEP <*arith. expr.*> (*optional*)
⋮
REPEAT

Direct Transfer Statements - SLANG direct transfer statements include the GØ TØ statement,

GØ TØ <*statement label*>

and the *indexed* GØ TØ statement,

GØ TØ NØDE (<*arith. expr.*>)

The statement label must not include the dollar sign if it is an alphanumeric name. In the indexed GØ TØ, the arithmetic expression will be evaluated and truncated to the nearest integer i, resulting in control being transferred to the statement labeled $NODEi.

Return Transfer Statements - The SLANG statement

JUMP TØ <*statement label*>

causes a direct transfer to the labeled statement with pushdown storage of the transfer point. The next subsequent occurrence of the statement

JUMP BACK

causes direct return to the transfer point plus one statement.

Simplified Input/Output

The input/output capabilities of SLANG are simplified extensions of Fortran input/output features. User-written Fortran subroutines may be called from SLANG to utilize full Fortran input/output.

List Directed Input - The SLANG input statement READ DATA causes a free field data file to be read from the standard input unit. The data file contains input data statements of the form:

name = value,

for an undimensioned variable and

name = value$_1$, value$_2$, ..., value$_n$,

or,

name(index) = value$_1$, value$_2$, ..., value$_n$,

for dimensioned variables. The end of a data file is designated by a dollar sign.

Simplified Output Commands - SLANG provides the output commands PRINT VARIABLES, PRINT PARTIALS, and PRINT FIRST PARTIALS, for printing values according to built-in format. Each of these commands may have an associated argument list that specifies the variables to be printed and their respective order. The PRINT PARTIALS and PRINT FIRST PARTIALS statements cause printing of all of the active partial derivatives for each variable in the order that the independent variables were declared.

If no argument list is present, then printing of the associated data for all variables will be printed in alphabetical order.

Text Printing - Output messages and labeled output may be printed using the PRINTØUT statement:

PRINTØUT <*name$_1$*>, <*name$_2$*>, ..., <*name$_n$*>

PRINTØUT $<*text*>$

PRINTØUT $<*text$_1$*>$ <*name$_1$*>, ..., $<*text$_n$*>$ <*name$_n$*>

The first character of the field will specify carraige control, therefore PRINTØUT 1 will cause page ejection on a line printer whereas PRINTØUT 0 will cause a line to be skipped.

Subprograms

The SLANG user may construct and invoke three types of subprograms: blocks (internal), subroutines (external) and macros (substitutive). In each case the communication of variables is carried out according to different rules.

Global and Local Variables - All SLANG variables are treated as *global* to all subprograms unless otherwise specified by the LOCAL declaration,

LØCAL <*variable list*>

If the LØCAL statement appears without a variable list, then all variables in the program or subprogram will be treated as local. In this case, specific variables may be declared global using the GLØBAL specification. In addition, both LØCAL and GLØBAL may be used to specify dimensioned variables.

The GLØBAL and LØCAL declarations are also used for *dynamic* dimensioning. The following are legal statements:

```
GLØBAL  X(10,20),  Y(N,M),  Z(2*N+1)
LØCAL  MUCH(JUNK)
```

Storage allocated using GLØBAL applies for all subprograms, whereas LØCAL storage allocation must be specified in each individual subprogram.

Internal Subprograms - SLANG internal subprograms are called blocks. They constitute protected segments of coding (within a program or subroutine) having the following structure:

```
BLØCK (<dummy parameter list>)
                optional
   :
optional { EXIT
   :
END BLØCK
```

Except for their special use in conjunction with the INTEGRATE command, blocks are invoked by the EXECUTE statement:

EXECUTE <*block*>(<*actual parameter list*>)
optional

Actual parameters are transmitted by *value;* thus entire arrays may not be transmitted.

The EXECUTE statement may also be used to execute blocks that reside in separate subprograms. This is enabled by the EXTERNAL BLØCK specification:

EXTERNAL BLØCK <*block 1*>, <*block 2*>, ...

External Subprograms - SLANG subroutines like Fortran subroutines, are individually compiled programs. They have the following form:

```
SUBRØUTINE <name> (<dummy parameter list>)
   :                        optional
   ENTRY <name> (<dummy parameter list>)
   :                        optional
   RETURN
   :
END
```

Each ENTRY may have its own distinct parameter list, as if it were a separate subroutine. Subroutines are invoked using the CALL statement:

CALL <*name*> (<*actual parameter list*>)

SLANG subroutines have a facility for non-standard returns to designated points in the calling program. This is accomplished by placing statement labels in the actual parameter list of the CALL statement (with leading dollar signs on numbers as well as names) and marking the corresponding position in the SUBROUTINE or ENTRY parameter list with dollar signs. Non-standard return is accomplished using the statement:

RETURN i

where i is an integer corresponding to the ith dollar sign in the SUBROUTINE or ENTRY parameter list.

Example:

Calling Program	Subroutine
:	SUBROUTINE (A,B,$,$)
Call SUB(A,B,$10,$L6)	:
:	RETURN 1
10 *<statement>*	:
:	RETURN 2
$L6 *<statement>*	:
:	RETURN
END	:
	END

The RETURN 1 and RETURN 2 statements causes return transfer to statements 10 and L6 respectively, whereas RETURN causes return transfer to the statement following CALL SUB.

The actual parameters for SLANG subroutines are transmitted by *name* as in Fortran. Thus entire arrays may be transmitted.

SLANG/Fortran Communication - Fortran subroutines may be called from SLANG programs using the CALL statement also. Variables may be transferred through the actual parameter list as if the subprogram were a SLANG subroutine. However, no other communication medium is available, i.e. SLANG global variables do not have meaning in the Fortran subprogram.

Fortran programs may also call SLANG routines using the statement:

CALL SLANG (*6H <name>*)

where *<name>* is the name of the SLANG subroutine. The transfer of arguments is considerably more involved, and is beyond the scope of this paper.

Macros - A SLANG macro may be defined as follows:

```
MACRO  <name>
       replacement text
END MACRO
```

The macro is invoked (substituted) at compile time by the presence of the macro name, optionally followed by an argument list:

<name> (*<argument list>*)

The macro call must occupy one line of text unless it is set off by asterisks, i.e. a macro SUM might be used in either of the following ways:

SUM (X,Y)

or,

Z = *SUM (X,Y)*/Z

The replacement text may be any SLANG statements, but if the call is imbedded within other statements (as in the second example) care must be exercised to generate the appropriate replacement. In addition to SLANG statements, macro time statements (characterized by having "MC" as the first two letters) may be used in the replacement text. Some examples are:

a. Macro time assignment statement

MCLET *<macro variable>* = *<macro expression>*.

b. Macro time conditional GØ TØ

MCGØTØ *<macro label>* IF *<macro conditional>*.

c. Macro time loop

MCFØR *<macro variable>*=*<macro variable>*
STEP *<macro variable>*TØ*<macro variable>*
INSERT*<replacement text>* REPEAT

d. Output of the value of a macro time variable

MCVAL (*<macro variable>*)

Output statement labels may be produced using the macro LAB(n) where n is a positive integer. Unique labels will be produced on every macro call. To generate a macro time label, which is merely a scan transfer point for MCGØTØ statements, the macro MCL(n) may be used, where n is any positive integer or macro variable.

The position of arguments in replacement text is specified by the macro ARG(n) where n is a positive integer or macro variable designating the position of the argument in the calling sequence. An argument may be referenced *by name* by using ARGNAME(n). This is useful if it is possible that the argument might be a macro, since a reference by means of ARG(n) will call that macro immediately, whereas reference by ARGNAME(n) will not.

Macro variables are designated by MCVn where n is a positive integer from 1 to 9. A number, MCLIST, may be referenced which has a value equivalent to the *number* of arguments in the current macro call.

The following is an example of a SLANG macro. Macro time statements are written in italics,

hereas SLANG replacement text is written in block ype.

```
MACRØ PAYØFF
    CØNTINUE
    MCLET MCV1 = 1.
MCL(1) IF INDEX LE MCVAL(MCV1)
        THEN ØPTIMIZE PAYØFF
            CØNTRAINT C1
            LAMBDA L1
            INDEPENDENT ARG(1)
            MCFØR MCV2 = 2 TØ MCV1
            INSERT, ARG(MCV2) REPEAT
            MCLET MCV1 = MCV1 + 1.
            ARG (MCV1) = ARG (MCLIST)
        ELSE MCGØTØ MCL(1) IF MCLIST GR MCV1+1.
            MCFØR MCV2 = 2 TØ MCV1
            INSERT REJØIN
            REPEAT
END MACRØ
```

or this macro, the macro call

```
PAYØFF (A1, A2, A3, A4, AAA)
```

ll generate the replacement text:

```
IF INDEX LE 1
    THEN ØPTIMIZE PAYØFF
        CONSTRAINT C1
        LAMBDA L1
        INDEPENDENT A1
        A2=AAA
    ELSE IF INDEX LE 2
        THEN ØPTIMIZE PAYØFF
            CØNSTRAINT C1
            LAMBDA L1
            INDEPENDENT A1, A2
            A3=AAA
        ELSE IF INDEX LE 3
            THEN ØPTIMIZE PAYØFF
                CØNSTRAINT C1
                LAMBDA L1
                INDEPENDENT A1, A2, A3
                A4=AAA
            ELSE
            REJØIN
        REJØIN
    REJØIN
```

Given more arguments in the macro call, the generated decision tree would have more branches, each THEN branch containing an OPTIMIZE loop heading with successively more independent variables.

The LET Operator - Macros as defined above may also be called using the LET operator. This alters the calling syntax slightly so that it resembles a function call:

LET *<arg1>* = *<macro>* (*<arg2>*, *<arg3>*, ...)

This has no impact on the macro definition.

Extension Facilities

The SLANG macro facility is a restricted version of an open syntax translator which makes up the first pass of the SLANG compiler. This translator is a Fortran implementation of the ML/1 syntax macro processor developed by P. J. Brown at University Mathematical Laboratory, Cambridge, England (Reference 3). ML/1 is a very powerful and well designed syntax macro translator that is highly machine independent.

SLANG syntax translation is accomplished by a system-defined set of macros which translate SLANG into a Fortran-like intermediate language called MODTRAN. To augment SLANG syntax, the user may also code translation macros using the ML/1 language. As ML/1 is a recursive language, the user need not know MODTRAN, but may translate new syntax into legal SLANG syntax, which then will be translated into MODTRAN by the "compiler" macro pack.

REFERENCES

1. J. C. STRAUSS, D. C. AUGUSTIN, M. S. FINEBERG, B. B. JOHNSON, R. N. LINEBARGER, F. J. SANSOM
   The SCI Continuous System Simulation Language (CSSL)
   Simulation, December 1967

2. G. N. BURGIN
   MIDAS III - A Compiler Version of MIDAS
   Simulation March 1966

3. P. J. BROWN
   The ML/1 Macro Processor
   Communications of the ACM October 1967

4. R. E. BELLMAN and R. E. KALABA
   Quasilinearization & Nonlinear Boundary Value Problems
   American Elsevier Publishing Co. New York 1965

BIBLIOGRAPHY

1. P. NAUR, J. W. BACKUS, F. L. BAUER, J. GREEN, C. KATZ, J. McCARTHY, A. J. PERLIS, H. RUTISHAUSER, K. SAMELSON, B. VAUQUOIS, J. H. WEGSTEIN, A. VAN WIJNGAARDEN, M. WOODGER

   Revised Report on the Algorithmic Language - ALGOL 60

   Communications of the ACM January 1963

2. R. W. FLOYD

   The Syntax of Programming Languages - A Survey

   IEEE Transactions on Electronic Computer - August 1964

3. E. I. ORGANICK

   Algorithmic Languages and Compilers

   Lecture Notes, University of Houston, 1965

4. B. ARDEN, B. GALLER and R. GRAHAM

   The Mad Manual

   University of Michigan Press, Ann Arbor 1965

5. A. J. PERLIS, R. ITTURIAGA, T. A. STANDISH

   A Preliminary Sketch of Formula Algol

   Carnegie Inst. of Technology, Pittsburgh, Pa., April 1965

6. H. MARKOWITZ, B. HAUSNER and H. KARR

   SIMSCRIPT: A Simulation Programming Language

   Prentice Hall, Englewood Cliffs, N.J. 1962

7. D. E. KNUTH and J. L. McNELEY

   SOL - A Symbolic Language for General Purpose Systems Simulation

   IEEE Transactions on Electronic Computers August 1964

8. M. D. McILROY

   Macroinstruction Extensions of Compiler Languages

   Communications of the ACM April 1960

9. S. SCHLESLINGER and L. SASHKIN

   POSE: A Language for Posing Problems to a Computer

   Communications of the ACM May 1967

10. B. M. LEAVENWORTH

    Syntax Macros and Extended Translation

    Communications of the ACM November 1966

11. M. I. HALPERN

    Toward a General Processor for Programming Languages

    Communications of the ACM January 1968

12. S. ROSEN, ED.

    Programming Systems and Languages

    McGraw-Hill, New York 1967

13. P. WEGNER

    Programming Languages, Information Structures and Machine Organization

    McGraw-Hill, New York 1968

APPENDIX

The following are examples of iterative problems solved in SLANG. The SLANG problem is given, followed by the output data from the last iteration in which convergence was achieved. Each of these problems was solved with "nominal" controls, thus convergence could probably have been more rapid if special controls had been imposed for each one.

MIDAS III/CSSL TWO POINT BOUNDARY VALUE PROBLEM

The following are the execution results of the problem treated on pp. 17, 18. The problem converged in 5 iterations of SOLVE (Implicit Equation Solver) yielding $\dot{y}(o) = 2.53711$.

SLANG Program

```
/ ..... SLANG VERSION OF MIDAS III/CSSL TWO POINT B.V. EXAMPLE PROBLEM .....
      DYDTO=1
      SOLVE GY1
          VARY DYDTO
          T=0
          Y=0
          DYDT=DYDTO
          INTEGRATION DERIVS,T,.05,DYDT,DY2DT2,Y,DYDT
          DO UNTIL T GE 1
              INTEGRATE DERIVS
          REPEAT
          GY1=Y-1
      END LOOP
      PRINT VARIABLES
      BLOCK DERIVS
          DY2DT2 = -(1+EXP(Y))
      END BLOCK
      STOP
      END
```

```
Final Iteration Results
IMPLICIT EQUATION SOLVER

ITERATION NUMBER  5

          IMPLICIT INDEPENDENT VARIABLES
DVDT0   2.53711E+00

          IMPLICIT CONSTRAINTS          NORM=  1.13687E-13
GV1    -1.13687E-13

          DG/DV
GV1     0.28763E-01

          DG/DV AFTER SCALING
GV1     1.00000E+00

          RANK  1. ABSOLUTE VALUE OF DETERMINANT  1.00000E+00

          DG/DV INVERSE
GV1     1.59043E+00

          ROW SCALE FACTORS
    1.59043E+00

          INITIAL COLUMN NORMS SQUARED
    1.00000E+00

          FINAL COLUMN NORMS SQUARED
    1.00000E+00

          COLUMN PERMUTATIONS
          1

          RESIDUALS
    0.

          DELTA V
    1.80810E-13



          DELTA V / V
    7.12663E-14

               VARIABLE VALUES

DVDT0   2.53711E+00   DVDT   -7.64216E-01   DV2DT2 -3.71828E+00   GV1    -1.13687E-13   T       1.00000E+00
Y       1.00000E+00
```

OPTIMAL DESIGN AND CONTROL EXAMPLE

This example, taken from Reference 4, minimizes the functional

$$J(y, a) = \frac{1}{2} \int_0^1 (x^2+y^2)\, dt + \frac{a^2}{2}$$

over the function y and the parameter a, where

$$\frac{dx}{dt} = ax+y, \quad x(0) = c, \quad \frac{da}{dt} = 0$$

Reference 4 formulates the Euler equations and boundary conditions

$$\dot{x} = ax+y, \quad x(0) = C$$

$$\dot{y} = x+ya, \quad y(1) = 0$$

$$\dot{a} = 0 \qquad \mu(0) = a(0)$$

$$\dot{\mu} = yx$$

and solves the problem by the method of quasilinearization. A Fortran program is given which contains approximately 100 statements.

In the following SLANG formulation, the equations are identical to Reference 4's, but the method is a simple parameter optimization with an imbedded two point boundary value problem involving four differential equations.

SLANG Program

```
/ ..... OPTIMAL DESIGN AND CONTROL EXAMPLE ..... BELLMAN AND KALABA
/          X, STATE VARIABLE
/          Y, CONTROL VARIABLE
/          U(=MU), LAGRANGE MULTIPLIER
/          A     , DESIGN PARAMETER
/
 READ DATA
     MINIMIZE J
          INDEPENDENT A
              SOLVE GY
                   VARY YO
                   Y = YO
                   X=1
                   U=A
                   Z=1
                   T=0
                   INTEGRATION DERIVS,T,.1,Z,ZDOT,X,XDOT,Y,YDOT,U,UDOT
                   DO UNTIL T GE 1
                        INTEGRATE DERIVS
                   REPEAT
                   GY=Y
              END LOOP
          J = Z/2 + A**2/2
          PRINTOUT T,X,Y,U,A,J
     END LOOP
     BLOCK DERIVS
          ZDOT = X**2 + Y**2
          XDOT = -A*X + Y
          YDOT = X + A*Y
          UDOT = Y*X
     END BLOCK
     PRINTOUT J,A
     STOP
     END
```

Final Iteration Results
IMPLICIT EQUATION SOLVER

ITERATION NUMBER 2

```
        IMPLICIT INDEPENDENT VARIABLES
YO    -6.40259E-01

        IMPLICIT CONSTRAINTS          NORM=  1.86916E-16
GY     1.86916E-16

        DG/DY
GY     1.85113E+00

        DG/DY AFTER SCALING
GY     1.00000E+00

        RANK  1, ABSOLUTE VALUE OF DETERMINANT  1.00000E+00

        DG/DY INVERSE
GY     5.40210E-01

        ROW SCALE FACTORS
   5.40210E-01

        INITIAL COLUMN NORMS SQUARED
  1.00000E+00

        FINAL COLUMN NORMS SQUARED
  1.00000E+00

        COLUMN PERMUTATIONS
        1

        RESIDUALS
  1.57772E-30

        DELTA Y
 -1.00974E-16


        DELTA Y / Y
  1.57708E-16

T      1.00000E+00   X      5.40210E-01   Y      1.86916E-16   U     -3.92095E-17   A      2.32900E-01
J      8.47253E-01
```

```
                                QUASI-LINEAR TRANSFORMATION

                                  ITERATION NUMBER   8

NEWTON-RAPHSON MATRIX

  ROW NUMBER    VALUE

       1        1.2264190E+00

FIRST PARTIALS OF U
                -5.9858785E-11

NORM SQUARED OF FIRST PARTIALS OF U 5.9858785E-11, LAST ITERATION 2.9600692E-05

EIGENVALUES AND EIGENVECTORS

        EIGENVALUE      EIGENVECTOR
 1.     4.7672616E-03   1.0000000E+00

DF/DX (SCALED AND ROTATED)
                -3.7411740E-12

DELTA X (SCALED AND ROTATED)
                7.8476374E-10

BOUNDS ON DELTA X
                6.2500000E-02

CHANGES IN INDEPENDENT VARIABLES          NORM= 4.9047734E-11
                4.9047734E-11

ABSOLUTE VALUE OF DELTA X / X
                2.1058676E-10

J       8.47253E-01    A       2.32908E-01
```

PERIODOGRAM ANALYSIS PROBLEM

This example, taken from Reference 4, determines the parameters α_i and w_i to match the periodic function

$$f(t) = \sum_{i=1}^{R} \alpha_i \cos w_i t$$

to a set of observations b_i, i=1, ..., M, where $M \geq 2R$. Reference 4's approach is to convert the problem into a variational problem involving multipoint boundary values, and use the method of quasilinearization to minimize the quantity.

$$\sum_{i=1}^{M} (f(t_i)-b_i)^2$$

The SLANG formulation is a simple parameter optimization with the desired parameters α_i, w_i as independent variables, and $\sum_{i=1}^{M} (f(t_i)-b_i)^2$ as payoff function.

SLANG Program

```
/ ..... PERIODOGRAM ANALYSIS PROBLEM ..... BELLMAN AND KALABA
/    A = ALPHA
/    W = OMEGA
/
     VARIABLE A(3),W(3),B(8),T(8),R(8)
$IN  READ DATA
     MINIMIZE SUMSQ
          INDEPENDENT A(1),A(2),A(3),W(1),W(2),W(3)
          SUMSQ = 0
          DO FOR I = 1 TO M
               F = 0
               DO FOR J = 1 TO 3
                    F = F + A(J)*COS(W(J)*T(I))
               REPEAT
               R(I)=F-B(I)
               SUMSQ=SUMSQ+R(I)**2
          REPEAT
          PRINT VARIABLES
     END LOOP
     PRINT VARIABLES
     GO TO IN
     END
```

Final Iteration Results
QUASI-LINEAR TRANSFORMATION

ITERATION NUMBER 1*

NEWTON-RAPHSON MATRIX

ROW NUMBER	VALUE					
1	7.1945417E+00	-7.8623151E-01	6.7400098E-01	3.0667900E+00	1.2507971E+00	5.5509134E-01
2	-7.8623151E-01	7.5142736E+00	1.2214430E+00	-4.0303656E+00	-1.3547234E+00	-2.0097558E-02
3	6.7400098E-01	1.2214430E+00	6.4362585E+00	-1.1696258E+00	-1.1347122E+00	6.0139467E-01
4	3.0667900E+00	-4.0303656E+00	-1.1696258E+00	7.0677832E+01	-7.9924643E+00	1.0160436E+00
5	1.2507971E+00	-1.3547234E+00	-1.1347122E+00	-7.9924643E+00	1.5645081E+01	2.1933065E+00
6	5.5509134E-01	-2.0097558E-02	6.0139467E-01	1.0160436E+00	2.1933065E+00	1.1989551E+01

FIRST PARTIALS OF U

-4.5036811E-12 -1.7186874E-11 7.5547237E-11 2.2254022E-11 -3.4422434E-11 -6.6589048E-11

NORM SQUARED OF FIRST PARTIALS OF U 1.1016943E-10, LAST ITERATION 3.4463038E-06

EIGENVALUES AND EIGENVECTORS

	EIGENVALUE	EIGENVECTOR				
1.	4.1151238E-01	8.7864959E-01	-9.6425861E-02	-7.2045382E-03	-8.6344629E-04	7.6944399E-02
		-4.6119494E-01				
2.	1.1036507E-01	5.4742281E-02	9.9179405E-01	1.3450886E-02	3.9773464E-02	5.3346973E-02
		-9.4433111E-02				
3.	2.1704581E-03	-7.4282860E-03	-1.6192508E-02	9.9947889E-01	4.1711420E-03	9.6618583E-03
		-2.4775314E-02				
4.	6.1589221E+00	1.7045151E-02	-1.2974603E-02	-3.0618096E-05	8.5968219E-01	-5.0780872E-01
		-5.1169911E-02				
5.	3.8053507E+00	8.1938255E-02	-3.9499469E-02	-2.5280624E-03	4.9463232E-01	8.1102649E-01
		2.9882174E-01				
6.	5.1224224E-01	4.6681750E-01	7.2349023E-02	2.8332302E-02	-1.2118303E-01	-2.7477915E-01
		8.2816388E-01				

DF/DX (SCALED AND ROTATED)

2.3850715E-11 2.4161349E-12 2.8270692E-12 1.7211184E-11 -2.8146225E-11 -4.3250021E-11

DELTA X (SCALED AND ROTATED)

-5.7958682E-11 -2.1892207E-11 -1.3025178E-09 -2.7945124E-12 7.3964864E-12 8.4432751E-11

BOUNDS ON DELTA X

2.5203081E-01 1.2743192E-01 2.0272198E-02 2.7871735E-01 5.1394551E-01 8.4709835E-01

CHANGES IN INDEPENDENT VARIABLES NORM= 1.1738221E-10

-6.2383146E-13 1.3790533E-12 -2.6340525E-11 -4.2446811E-12 -1.7472054E-11 1.1295654E-10

ABSOLUTE VALUE OF DELTA X / X

6.2384630E-13 2.7587381E-12 2.6321264E-10 3.8236304E-12 8.6071433E-12 3.3038870E-11

VARIABLE VALUES

A	9.99976E-01		4.99886E-01		1.00073E-01	R	1.60000E+00		4.52413E-01	
	-6.79691E-01		-8.20313E-01		-3.67904E-01		-3.81874E-01		3.5108?2E-01	
	-0.	F	3.51064E-01	I	8.00000E+00	J	4.000005+00	M	7.000005+00	
R	-6.50702E-05		-1.08731E-04		-9.92344E-05		-7.92199E-05		-4.87865E-05	
	-3.50602E-05		-1.82409E-05		-0.	SUMSQ	3.49670E-08	T	0.	
	8.30000E-01		1.67000E+00		2.50000E+00		3.33000E+00		4.17000E+00	
	5.00000E+00		-0.	W	1.11012E+00		2.02995E+00		3.41921E+00	

ORBIT DETERMINATION PROBLEM

This example, taken from Reference 4, is an orbit determination problem formulated as a multipoint boundary value problem. The four differential equations

$$\dot{x} = u$$

$$\dot{u} = -\frac{x}{(x^2+y^2)^{1.5}}$$

$$\dot{y} = v$$

$$\dot{v} = -\frac{y}{(x^2+y^2)^{1.5}}$$

are to be solved subject to the multipoint boundary conditions $y(t_i) = (x(t_i)-1) \tan \theta_i$, i=1,2,3,4

This problem was solved in Reference 4 by Quasilinearization. A FORTRAN program is given which requires approximately 120 statements, not counting utility subprograms.

The SLANG formulation utilizes a SOLVE loop which varies the initial conditions x(0), y(0), u(0) and v(0) to satisfy the multipoint constraints $g_i = y(t_i) - (x(t_i)-1) \tan \theta_i$, i=1,2,3,4

for $t_1 = .5$, $t_2 = 1.0$, $t_3 = 1.5$, $t_4 = 2.0$. Integration is carried through to t = 2.5 where final results are printed.

SLANG Program

```
/ ..... ORBIT DETERMINATION AS MULTIPOINT B.V. PROBLEM ..... REFERENCE 4
/         SHOOTING METHOD
/
        XO=1.0
        YO=0
        UO=0
        VO=2*3.14159/3.664
        TAN1=SIN(.251297)/COS(.251297)
        TAN2=SIN(.51024 )/COS(.51024 )
        TAN3=SIN(.78369 )/COS(.78369 )
        TAN4=SIN(1.07654)/COS(1.07654)
    SOLVE G1,G2,G3,G4
        VARY XO,YO,UO,VO
        T = 0
        X=XO
        Y=YO
        U=UO
        V=VO
        INTEGRATION DERIVS,T,.05,X,XDOT,Y,YDOT,U,UDOT,V,VDOT
        DO UNTIL T GE 2.5
            INTEGRATE DERIVS
            IF T EQ .5 THEN G1 = Y - (X-1)*TAN1
*               ELSE IF T EQ 1 THEN G2 = Y - (X-1)*TAN2
*               ELSE IF T EQ 1.5 THEN G3 = Y - (X-1)*TAN3
*               ELSE IF T EQ 2.0 THEN G4 = Y - (X-1)*TAN4
        REPEAT
        PRINT VARIABLES
    END LOOP
   BLOCK DERIVS
        DEN = (X**2 + Y**2)**1.5
        UDOT =-X/DEN
        VDOT =-Y/DEN
        XDOT = U
        YDOT = V
    END BLOCK
    STOP
    END
```

Final Iteration Results
IMPLICIT EQUATION SOLVER

ITERATION NUMBER 7

VARIABLE VALUES

DEN	9.[illegible]E+0	G1	5.325 7E-15	G2	1.598872E-14	G3	3.9[illegible]E-14	G4	1.4921[illegible]
TAN1	2.9[illegible]E-1	TAN2	5.[illegible]E-01	TAN3	9.[illegible]E-01	TAN4	1.[illegible]E+0	T	2.[illegible]
UDOT	-2.[illegible]E-1	U	4.47[illegible]E-5	U	-6.84267E-01	VDOT	-2.[illegible]E-1	V	4.[illegible]E-1
V	2.474[illegible]E-1	XDOT	-6.6[illegible]E-1	X	1.999981E+00	X	1.[illegible]E+0	YDOT	2.474[illegible]
Y	[illegible]	Y	[illegible]						

IMPLICIT INDEPENDENT VARIABLES

X	1.999981E+	V	-5.59892E-[illegible]	UD	4.47519E-[illegible]	VC	4.[illegible]E-1

IMPLICIT CONSTRAINTS NORM= 1.951658E-13

G1	5.325 7[illegible]-12	G2	1.598772E-14	G3	3.9[illegible]799E-14	G4	1.492145E-13

DG/DV

G1	-2.628[illegible]E- 1	9.6[illegible]73E-1	-1.[illegible]921 E-1	4.972926E-1
G2	-6.146[illegible]E- 1	9.[illegible]322E- 1	-5.[illegible]133E-01	9.754355E- 1
G3	-1.23[illegible]72E+00	8.141[illegible]E- 1	-1.[illegible]76E+0	1.394[illegible]E+00
G4	-2.7[illegible]E+	[illegible]73E- 1	-4.2 [illegible]	1.[illegible]E+0

DG/DV AFTER SCALING

G1	-2.[illegible]388E-01	9.[illegible]1E-1	-1.133[illegible]E-1	4.359[illegible]E-01
G2	-3.807[illegible]E-01	5.854[illegible]E- 1	-5.[illegible]E-01	6.137[illegible]E-01
G3	-4.76[illegible]E-01	3.[illegible]E- 1	-6.17883E-01	5.399[illegible]E-01
G4	-5.1254[illegible]E- 1	1.[illegible]E- 1	-7.96701E- 1	3.[illegible]E-01

RANK = 4, ABSOLUTE VALUE OF DETERMINANT 1.695588E-13

DG/DV INVERSE

G1	2.1251[illegible]E+01	-4.[illegible]E+ 1	3.491[illegible]E+01	-7.2[illegible]E+00
G2	3.[illegible]E+	-4.[illegible]E+	[illegible]E+	-4.[illegible]E-01
G3	-1.[illegible]E+	2.7[illegible]E+ 1	-1.[illegible]E+01	3.57221E+
G4	2.[illegible]E+	-8.[illegible]E+	[illegible]E+	-2.[illegible]E+

ROW SCALE FACTORS

8.7[illegible]E-1 8.29253E-01 3.87325E-1 1.89298E-01

INITIAL COLUMN NORMS SQUARED

1.19647E+ 1.10[illegible]E+00 9.[illegible]E-1 6.92586E-01

FINAL COLUMN NORMS SQUARED

1.19647E+ [illegible] 8.74254E-01 4.451[illegible]E- 2 6.17495E-05

COLUMN PERMUTATIONS

4 3 2 1

RESIDUALS

8.[illegible]E-28 8.[illegible]E-28 -4.84676E-27 -4.84676E-27

DELTA V

3.[illegible]E-13 2.13365E-14 -1.65855E-13 1.342J7E-13

DELTA V / V

1.[illegible]E-13 3.81[illegible]E-09 3.7[illegible]E-09 2.[illegible]E-13