a project report on

REL

Computer Languages for

Numerical Engineering Problems

Richard Henry Bigelow

REL Project Report No. 5

Co-principal investigators :
Bozena Henisz Dostert
Frederick B. Thompson

California Institute of Technology
Pasadena, California, 91109

Computer Languages for Numerical Engineering Problems

Richard Henry Bigelow

REL Project Report No. 5

California Institute of Technology
Pasadena, California 91109

The research reported here was
supported in part by the National Institutes
of Health, grant GM01335

COMPUTER LANGUAGES FOR NUMERICAL ENGINEERING PROBLEMS

Thesis by

Richard Henry Bigelow

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1973

(Submitted December 7, 1972)

ACKNOWLEDGEMENTS

I wish to thank Dr. Frederick B. Thompson for his patience and help during my graduate study.

This work was supported in part by the National Science Foundation and the National Institutes of Health.

ABSTRACT

Recent and anticipated advances in computer hardware capabilities have made hardware limitations insignificant for many numerical engineering problems. The difficulties of programming computers now constitute the greatest block to their effective utilization by engineers and scientists. Consequently, new languages that are specialized to numerical engineering problems are needed.

Relmath is such a language. It is designed to solve ordinary differential equations and to manipulate the resulting functions. Systems of equations can be stated in a normal mathematical form and solved by a simple statement. Printed and plotted output can be readily obtained.

Relmath also allows the definition of procedures for solving differential equations. Its procedural language is quite different from general programming languages. It is restricted to a certain class of algorithms, and the calculations that are common to all these methods are made implicit in the language. The language is highly supportive for procedures in this class. The user need only state the important mathematical steps, such as the formulas defining a Runge-Kutta scheme or the method of estimating the error, if error control is desired.

Some considerations for the design and implementation of numerical engineering languages are discussed using Relmath as an example. The decisions involved in the design of Relmath are detailed. The behavior of a representative numerical algorithm in a paging environment is analyzed, which shows the importance of properly designing algorithms for such environments. Relmath's method of compiling its procedures is discussed. The compiled code is as fast as standard library subroutines. Finally, a plan for further research to develop a more supportive environment for the implementation of similar languages is outlined.

TABLE OF CONTENTS

I. INTRODUCTION

Throughout their history computers have been used to solve numerical problems arising in science and engineering. Originally, the restricted capabilities of computers and their expense were the primary limitations on this usage. However, over the years, and especially recently, computer hardware has become enormously more powerful while the cost of achieving a given performance level has dropped markedly. Consequently, hardware limitations are no longer so important, and the difficulties of programming now constitute the primary limitation on the effective use of computers. Algorithmic languages like FORTRAN were a first step toward easing these difficulties, but they do not go far enough. Engineers and scientists who are not professional programmers need languages that are still more supportive of their work and further reduce the programming effort required to solve their problems.

This thesis is concerned with languages that meet this need. These are high-level, application-oriented languages which are specialized to numerical engineering problems. They allow users to state their problems in a normal and familiar mathematical form and have them solved easily. Hence, they significantly reduce the programming effort needed to solve these problems. We will show the usefulness of these languages and discuss their design and implementation.

As a concrete and explicit example, Relmath will be presented. This language was designed and implemented by the author. It is primarily intended to be used to solve ordinary differential equations, but it also has facilities for evaluating and manipulating functions defined by other means. We will use it to illustrate the benefits of specialized languages which enhance the users' ability to solve their particular problems.

Based upon our experience with Relmath and other languages, we will then examine some aspects of the design and implementation of high-level, specialized languages and recommend some goals for further research. This examination will be divided into five parts, which will be covered in separate chapters. These parts are:

1. language design. It is important for a language to be natural and highly supportive within its application area. To illustrate how these goals can be attained, the design of Relmath is discussed in detail, with emphasis upon the considerations that led to the present form of the language. The special considerations involved in designing the procedural part of Relmath, which is very natural for writing certain classes of algorithms to solve differential equations, are presented.
2. numerical methods. High-level languages for numerical engineering problems solve some problems by invoking built-in numerical algorithms. Relmath, for example, has a built-in algorithm for solving ordinary differential equations. The numerical

considerations involved in designing these algorithms are discussed.

3. paging. Since virtual memory environments will be increasingly used, their implications for these specialized languages must be considered. An analysis of the paging characteristics of a representative algorithm shows that specialized algorithms are needed for efficient operation.

4. compiling. It is desirable to compile efficient code for computations that will be executed repeatedly. This objective imposes certain restrictions on the language so that effective optimizing compilers can be applied. Relmath's compiling technique is presented.

5. support for language implementation. The earlier chapters discuss languages that help engineers and scientists to more effectively utilize computers by reducing the programming difficulties they face. The implementation of such languages is another major task that currently receives little support. This large implementation effort now blocks the ready development of these languages. A program is presented to develop a more supportive environment for their implementation. This environment also makes it easier to modify and interface these languages.

II. THE RELMATH LANGUAGE

We will use Relmath as an example of a language for numerical engineering problems. It is a prototype language oriented especially toward the solution of ordinary differential equations. It has been implemented under the REL system on an IBM 370/155. This chapter presents the language; later chapters will discuss certain aspects of its design.

We will first present a formal specification of Relmath's syntax and semantics. This will be followed by some examples. The formal presentation is not intended to be a user's manual. Indeed, a natural language, such as Relmath, is better learned from some examples and actual usage than from a manual.

Formal Specification of Relmath

The metalanguage used to describe Relmath's syntax is similar to other syntax languages. Metalanguage names are in lower case and may include the symbol _; upper case letters and special characters denote themselves as text. The string : : = means "is defined as." Brackets [ ] enclose optional strings. Alternatives are delimited by the underlined word or. Alternatives are also indicated by giving more than one definition of a metalanguage name. The name can then mean any of the alternatives in any of its definitions. List of followed by a plural metalanguage name denotes a list of at least one of those metalanguage entities separated by commas.

For example,

array_list : : = list of arrays

means the same as

array_list : : = array or array_list, array

String of is the same as list of, but the commas are omitted.

For example,

number : : = string of digits

means the same as

number : : = digit or number digit

A letter is any of the usual alphabetic characters A, B, C, . . . , Z. A digit is a numeric digit 0, 1, . . . , 9.

All input is on cards in columns 1 through 71. A non-blank character in column 72 indicates that the statement is continued on the next card in column 1. Except within character strings, a sequence of blanks is equivalent to one blank, which acts as a name separator.

The metalanguage name eos denotes the end of the statement.

The syntax is presented in the following order: names and declarations, expressions, including arrays and functions, assignment statements, differential equations and the SOLVE statement, output statements, FOR and IF clauses, and procedures.

Names and declarations

namechar : : = letter or digit or $ or # or @or _ or ? or %

name : : = letter string of namechars

There are other restrictions on names which do not appear in this syntax. A name cannot

1. be the same as any of the function names in Table 1.
2. be SUM, PROD, MAX, MIN, AND, ALL, OR, or ANY.

A name must be separated from any namechar by some other type of character, possibly one or more blanks. Violations of these restrictions may cause ambiguities.

declaration : : = list of names : = data_type [S]

data_type : : = SCALAR or ARRAY sublist or FUNCTION

or ARRAY sublist FUNCTION

or ARRAY FUNCTION [S] sublist

or CONDITION FUNCTION

or SYSTEM or EQUATION

or INDEPENDENT VARIABLE

or PROCEDURE or DEPENDENT VARIABLE

or PRECISION

or ARRAY < * > or STEP SIZE

or STEP NUMBER or DERIVATIVE FUNCTION

The optional S after the data type has no effect.

Names declared with a particular data type are represented by a corresponding metalanguage name, as in the following table:

Metalanguage name	Data type
scalar	SCALAR
array_name	ARRAY < sublist>
function_name	FUNCTION
condition_function_name	CONDITION FUNCTION
array_function_name	ARRAY FUNCTION <sublist>, ARRAY <sublist> FUNCTION
system_name	SYSTEM, EQUATION
t	INDEPENDENT VARIABLE
procedure_name	PROCEDURE
procedural_array	ARRAY <*>
h	STEP SIZE
eps	PRECISION
n	STEP NUMBER
df	DERIVATIVE FUNCTION
z	DEPENDENT VARIABLE

A declaration defines the name, allocates storage for it, and initializes the storage, if necessary. A scalar is initialized to zero, and all elements of an array are initially zero.

sublist : : = <list of n_exprs>

An n_expr is a numeric expression, which will be formally defined later. Each expression is evaluated and rounded to an

integer. The integer must be greater than zero. Each integer is a dimension for the array. The elements of the array are stored contiguously. The total array size, which is the product of the dimensions, cannot exceed 127.

Array functions also have dimensions with the same meaning as for arrays. An array function can be thought of as a function whose value is an array or as an array whose elements are functions.

Functions are numeric functions. Condition functions are functions whose values are true or false. System names refer to systems of ordinary differential equations. The remaining data types are associated with procedures and will be discussed later.

Names for scalars, functions, condition functions, systems, and procedures need not be declared. A name may only have one meaning at a time, but it may be used for different entities at different times. Only the most recent use is valid. An assignment statement sets the type of a name as well as its value.

Expressions

digit_string : : = string of digits

numberl : : = digit_string or digit_string.

or . digit_string or digit_string . digit_string

number : : = numberl or numberl E digit_string

or numberl E + digit_string

or numberl E - digit_string

The digit string after an E in a number is a power of ten; the value of xE± n is $x \cdot 10^{\pm n}$. The value of a number is always a double precision floating number on the IBM System/370. It has about 16 decimal digits of precision, and its magnitude can be 0 or approximately 10^{-78} to $7 \cdot 10^{75}$. All numeric quantities in Relmath are represented in this form.

primitive : : = number or scalar or array_element
or function_value
or generator_value or |n_expr|
or (n_expr)
or LOG BASE n_expr (n_expr)

factor : : = primitive or + factor or - factor
or primitive ** factor

term : : = factor or term * factor or term/factor

sum : : = term or sum + term or sum - term

modulus : : = sum or modulus MOD sum

n_expr : : = modulus
or IF condition THEN n_expr, [ELSE] n_expr
or modulus IF condition, [ELSE] n_expr

These rules give the syntax of a numeric expression; the various forms of a primitive will be defined later. The semantic transformations of these rules are the usual arithmetic operators of FORTRAN, with some additions. Thus, +, -, *, /, and ** denote the binary operations of addition, subtraction, multiplication,

division, and exponent iation, respectively; + and - also denote the unary operations of unary plus and negation. |x| is the absolute value of x. LOG BASE x(y) is log y/log x. x MOD y = x- |y| ⌊x/ |y|⌋ , where ⌊z⌋ is the greatest integer less than or equal to z. Hence, $0 \leq x \text{ MOD } y < |y|$ for all x, y. In conditional expressions of the forms x IF u, ELSE y or IF u THEN x, ELSE y, if u is true x is evaluated, and y is not; otherwise y is evaluated and x is not. Hence, y need not be meaningful when u is false.

We now define Boolean expressions.

relation_op : : = = or < or > or <= or >= or ¬= or ¬ < or ¬>

simple_relation : : = n_expr relation_op n_expr

relation : : = simple_relation

or relation relation_op n_expr

condition_primitive : : = relation or condition_function_value

or condition_generator_value

or (condition)

negation : : = condition_primitive

or NOT negation

conjuntion : : = negation

or conjunction AND negation

disjunction : : = conjunction or disjunction OR conjunction

condition : : = disjunction

These rules define conditions, or Boolean expressions. In a relation of the form relation relation_op n_expr, the operator is

applied to the last number of the inner relation and the n_expr, and the result is anded with the truth value of the inner relation. Hence, the value of a relation is a pair, a truth value and the value of the last n_expr in it. This interpretation means that $1 < x < 2$ has its usual mathematical meaning.

The relation_ops denote the following relations:

= equal	¬= not equal
< less than	<=, ¬> less than or equal
> greater than	>=, ¬< greater than or equal

The other rules have their usual interpretations.

Functions

The names listed in Table 2-1 are function names. The definitions of the functions are also given.

Function name	Function definition
LOG, LN, LOG10	$\log_e x$, $\log_e x$, $\log_{10} x$
EXP	e^x
SQRT	$\sqrt{x}$
SIN, COS, TAN	Trigonometric functions, arguments in radians
SIND, COSD, TAND	Same, arguments in degrees
ASIN, ACOS, ATAN	$\sin^{-1} x$, $\cos^{-1} x$, $\tan^{-1} x$ ATAN (x, y) = ATAN (y/x), value in proper quadrant
ASIND, ACOSD, ATAND	Same, values in degrees
SINH, COSH, TANH	Hyperbolic trigonometric functions
ERF, ERFC	$\frac{2}{\pi}\int_0^x e^{-u^2}\,du$, 1-ERF(x)
GAMMA, LGAMMA	$\int_0^\infty u^{x-1} e^{-u}\,du$, $\log_e$ GAMMA(x)
ABS	$\lvert x\rvert$
FLOOR	$\lfloor x \rfloor$ = greatest integer $\leq x$
CEIL	$\lceil x \rceil$ = least integer $\geq x$
SIGN	1 if x > 0, 0 if x= 0, -1 if x < 0
ROUND	integer nearest x = SIGN (x) · FLOOR (|x| + .5)
TRUNC	integer part of x=SIGN (x) · FLOOR (|x|)
MOD	MOD(x, y) = x MOD y = x- |y| · FLOOR (x/ |y|)

Table 2-1. Built-in Relmath functions

function_value : : = total_function arglist

or generator arglist

total_function : : = function_name [string of 's]

or total_array_function sublist

or array_function_name sublist string of 's

total_array_function : : = array_function_name [string of 's]

arglist : : = (list of arguments)

argument : : = n_expr or condition or array

or total_function

or total_array_function

or condition_function_name

or temporary_number_function

or temporary_condition_function

generator : : = SUM or PROD or MAX or MIN

This syntax shows how functions are used. The syntax for temporary functions depends on some forms used in defining the syntax of a function assignment statement. That definition will be given later, and then the notation for temporary functions will be defined.

A ' following a function name denotes differentiation; the order of the derivative equals the number of quote symbols. Array functions can also be differentiated; the quotes may be either before or after the sublist.

Numeric functions can be defined by a formula in a function assignment statement or by a table of numeric values. Such a table is constructed by solving a system of differential equations for the function. Array functions can only be defined by such tables. Functions defined by numeric tables can be differentiated as many times as their maximum orders in the systems used to compute their values. Those defined by formulas cannot be differentiated.

When a function defined by a table is invoked in a function_value, it must have only one argument. This argument's value must be within the range covered by the table. The domain value closest to the argument is found in the table, and the corresponding range value is returned. The same thing is done for array functions; the value is the array associated with the nearest domain point.

The generators can have any number of arguments. They return the sum, product, maximum, or minimum, respectively, of the arguments.

Arrays

array : : = array_name or total_array_function arglist

or array_function_name arglist string of 's

array_element : : = array sublist

The number of n_exprs in the sublist must equal the number of dimensions. Each n_expr x_i is evaluated and rounded to an

integer K_i. Each K_i must satisfy $1 \leq K_i \leq d_i$, where d_i is the corresponding dimension. The value of this element of the array is used.

Generator functions

generator_value : : = generator FOR var=range (n_expr)

or FIRST var = range SUCH THAT (condition)

or FIRST var = range ST (condition)

var : : = name

primitive : : = var

range : : = initrng TO endrng BY incrng

or initrng BY incrng TO endrng

or initrng TO endrng

or initrng, nextrng, . . . , endrng

or initrng, . . . , endrng

initrng : : = n_expr

incrng : : = n_expr

endrng : : = n_expr

nextrng : : = n_expr

A range defines a sequence of values. In the third form, incrng is implicitly one, except in a FOR clause on SOLVE and PLOT statements. Then incrng = (endrng-initrng)/100. The fourth form is equivalent to the first with incrng=nextrng-initrng. Form 5 is equivalent to form 3. Finally, the sequence is initrng,

initrng + incrng, initrng + 2 incrng, ..., initrug + n· incrng, where n= FLOOR ((endrng-initrng)/incrng). If n<0, the range is null.

A var is just a place-holder for the current value of the range sequence. These values do not affect the values of the same name elsewhere. The var may be used in the n_expr or condition as a numeric primitive, as the syntax shows.

The value of the first form of a generator_value is the sum, product, maximum, or minimum of the values that the n_expr takes on for the values of var in the range. The two forms involving FIRST are equivalent. The value is the first value of var in the range such that the condition is true. If the condition is not true for any value in the range, the output is K, the maximum number representable in Relmath. K is approximately $7 \cdot 10^{75}$.

If the range is null, the generator functions have the following values:

operator	value if range is null
SUM	0
PROD	1
MAX	-K
MIN	K
FIRST	K

Condition functions and generators

condition_function_value : : = condition_function_name arglist

or condition_fn arglist

condition_fn : : = AND or ALL or OR or ANY

The condition functions are defined by function assignment statements, which will be discussed later. The arguments of the built-in functions AND, ALL, OR, and ANY can be any number of conditions. The first two compute the conjunction of the arguments, and the last two compute their disjunction.

condition_generator_value : : = FOR condition_generator

var=range (condition)

condition_generator : : = ALL or SOME or ANY

SOME and ANY are equivalent. They compute the disjunction of the condition values obtained by stepping var through the range, and ALL computes the conjunction. If the range is null, ALL returns true, and SOME and ANY return false.

Assignment statements

assignment : : = scalar = n_expr

or array_name sublist = n_expr

In the second form the sublist must conform to the same restrictions as in the case of an array_element. The n_expr is evaluated, and its value is assigned to the scalar or array element.

Function assignment statements

function_assignment : : = function_name parmlist=n_expr

or condition_function_name parmlist=condition

parmlist : : = (list of parameters)

parameter : : = number_var or array_var

or function_var or array_function_var

or condition_var

or condition_function_var

primitive : : = number_var

array : : = array_var

function_name : : = function_var

array_function_name : : = array_function_var

condition_primitive : : = condition_var

condition_function_name : : = condition_function_var

number_var : : = name

array_var : : = name

function_var : : = name

array_function_var : : = name

condition_var : : = name

condition_function_var : : = name

The names which appear in the parmlist are formal parameters. They can be used as numbers, arrays, conditions, etc. as shown in the syntax. They serve only as place-holders for the

actual arguments. Neither this statement nor the use of the function affects any values associated with these names in any way.

This statement defines the function or condition function by a formula. When the function is used in a function_value or condition_function_value, the arguments are substituted for the formal parameters. The value of the function is then obtained by evaluating the formula. To do this, the arguments must agree with the parameters. An n_expr can be substituted for a number_var, an array for an array_var, and a condition for a condition_var. Total_array_functions can be substituted for array_function_vars. Total_functions and temporary_number_functions can be substituted for function_vars, and condition_function_names and temporary_condition_functions for condition_function_vars.

A function can be redefined, perhaps with different parameters. It can also be changed from one defined by a table to one defined by a formula or vice versa. Whenever the function is invoked, its most recent definition is used.

Functions may be recursive.

Temporary function arguments

We now give the syntax for temporary functions.

temporary_number_function : : = FUNCTION parmlist: n_expr

temporary_condition_function : : =

FUNCTION parmlist : condition

This notation defines a function which is the argument of another function. The parameters of the defined function are given in the parmlist. The n_expr or condition is a formula that defines a function of these parameters. For example,

FUNCTION (X, Y): X ** 2 + Y ** 2

defines a function argument f where $f(x, y)=x^2 + y^2$.

Systems and the SOLVE statement

primitive : : = total_function

equation : : = n_expr = n_expr

or equation FOR var=range

initial_condition : : = total_function = n_expr

or initial_condition FOR var = range

eql_item : : = equation or [WITH] [INITIAL]

initial_condition or system_name

equation_list : : = list of eql_items

sysasgn1 : : = system_name: equation_list

eql_statement : : = equation_list eos

sysasgn2 : : = BEGIN system_name eos

string of eql_statements

END system_name

system_assignment : : = sysasgn1 or sysasgn2

In an equation, the unknown functions are represented by derivatives without argument lists. An initial condition specifies the initial value of some function. The FOR clauses on equations

and initial conditions allow them to be iterated for various values of some array subscript.

The first type of system assignment is used for small systems. The system consists only of the eql_items in this statement. The second type is really a compound statement, since it spans several input statements. The system consists of all the eql_items between the BEGIN and END. Actually, the syntax rules are incomplete. Other types of statements beside eql_statements can appear between the BEGIN and END. Even other system_assignments can appear. However, these statements are not part of the system definition and could just as well appear outside it.

System names are allowable eql_items. Hence, systems can have subsystems, but not recursively. When the system is used, the current definitions of its subsystems are expanded until no subsystems remain. All the resulting equations are in the system. So are all the initial conditions, but their order is important. Conditions override conditions on their left or in earlier equation_statements and also in subsystems to the left or above.

solve_statement : : = SOLVE equation_list

FOR scalar=range [with PRECISION=n_expr]

with : : = , or WITH

or, WITH

This statement solves the differential equation system defined by the equation list and its subsystems and saves the values of the unknown functions and their derivatives for values of the scalar in the range. If no increment is specified in the range, it is taken to be 1/100 times the end value minus the initial value. The end value can not be less than the initial value, and the increment must be positive. This is the only statement that defines a function by a table.

The scalar in the FOR clause is the independent variable of the system. It may also appear in the equations. Its value is undefined after the SOLVE statement is executed. The dependent variables of the system are the functions and array functions which appear without argument lists. Any other function is an auxiliary function which must be known. An unknown function name is used as a numeric primitive; its dependence on the independent variable is implicit.

Certain restrictions exist on the equations. They must be linear in the maximal order derivatives of the unknowns; they may be non-linear in the lower-order derivatives. The system must be triangular in the maximum order derivatives. If an unknown is an array function, two maximal derivatives with different subscripts can not appear in the same equation. The standard method for solving the system will not solve stiff systems.

The system will be solved by a built-in, fourth-order method that combines a Runge-Kutta scheme and an Adams-Moulton predictor-corrector method. The step size is automatically controlled to keep the estimated local relative truncation error below the requested precision. However, only the computed values for the values of the independent variable specified in the range are saved. The derivatives are saved through the maximal orders. Only these derivative values can be requested in later statements; higher-order derivatives cannot be used. The method will terminate with an error message if the step size must be reduced to less than $2^{-10}=1/1024$ times the range increment.

The precision can be set for just one SOLVE statement by adding the PRECISION clause to the statement. It can also be set globally by a PRECISION statement.

precision_statement : : = PRECISION= n_expr

This statement sets the global precision, which retains this value until another PRECISION statement is executed. The global precision is used to control the step size for any SOLVE statement which does not have a PRECISION clause. The default global precision of 10^{-4} is in effect until the first PRECISION statement.

PRINT statements

plot_element : : = n_expr or array

out_element : : = plot_element or condition

out_list : : = list of out_elements

print_statement1 : : = PRINT out_list

or PRINT out_list IN FORM form

print_statement : : = print_statement1 [FOR ALL var]

form : : = string

string : : = charstring

or charstring IF condition, [ELSE] string

or IF condition THEN string, [ELSE] string

A charstring is any sequence of characters enclosed in double quotes ("). A double quote within the sequence is represented by two double quotes. A charstring is a literal value; blanks are not deleted within it.

A string can be conditional, as shown. The value is the first string if the condition is true; otherwise, it is the second.

A print_statement has several output forms, depending on whether it has a form or a FOR ALL clause. When neither is present, the out_elements' values are printed using a default format. Numbers are printed in an exponential form if they are very large or small. Arrays are expanded to a list of numbers; the order is determined by varying the last subscript fastest. Hence, a two-dimensional array is shown row-by-row. A condition is displayed as TRUE or FALSE. Several out_elements are printed to a line until the list is exhausted.

If a form is given, it controls the format of the output lines. All characters except periods (.) and underscores (_)

represent text. These text fields are moved to the output line unchanged. A sequence of underscores with an optional period represents a fixed number. A number from the out_list placed in this field is shown with its decimal point where the period is and digits and a sign where the underscores are. Leading zeroes and plus signs are not shown. If there is no period, the number is printed as an integer. A sequence of at least six periods represents a floating field. A number placed in such a field has a minus sign, if it is negative, one digit, a decimal point, more digits, and a four-character exponent Esmn, where s is a blank or a minus sign. Numbers are always rounded at the right end. If a field is too short, it is filled with asterisks.

A condition may be placed in a field which is just underscores. It will print as TRUE or FALSE, or as T or F if the field has less than five underscores, followed by blanks.

The form defines a single line. The form is processed from left to right, with each number or condition in the out_list going in one field. When the end of the form is reached, the line is printed, and the form is re-used from the left.

A FOR ALL clause indicates that the var's range is the domain of a function in the out_list. There must be at least one function defined by a numeric table in the out_list; if there is more than one, they must all have the same domain. This domain is used as the range of the var. The PRINT statement is repeated for each value in the range.

print_digits_st : : = PRINT n_expr DIGITS

This statement sets the number of digits to print when no form is given. If no such statement has been executed, the number is $\lceil -\log_{10} P + 1 \rceil$, where P is the current global precision. A few more digits than the number requested may be printed, depending on the magnitude of the printed number.

eject_st : : = EJECT

skip_st : : = SKIP [1 LINE] or SKIP n_expr [LINES]

These statements control the spacing of the output. EJECT starts the next output line at the top of the next page. A SKIP statement skips 1 or n blank lines, where n is the value of the n_expr truncated to an integer.

print_string_st : : = PRINT string

This statement prints the string's value without the enclosing quotes.

PLOT statements

Plotted output is produced by these statements.

plot_statement1 : : = PLOT plot_list

or PLOT plot_list VS [.] n_expr

plot_statement : : = plot_statement1 [FOR ALL var]

plot_list : : = list of plot_elements

The expressions in the plot_list and n_expr should be functions of a var. In the first form of a plot_statement1, the

plot_elements are plotted against the var. In the second form they are plotted against the values of the n_expr. If a plot_element is an array, its elements are all plotted. All the plot_elements are plotted to the same scale on one sheet. The plotting area is 15 inches horizontally by 10 inches vertically.

The range of the var can be explicitly shown by a FOR clause, as in the next section. If the increment in the range is not given, it is assumed to be .01 times the end value minus the initial value. The range can also be implicitly derived from the domains of the tabular functions being plotted, just as in a PRINT statement, by a FOR ALL clause.

Initially, no labels are drawn on the axes, and default values are used for the minimum and maximum values on each axis. These default values will include the full data range and be rounded to reasonable values. Labels and non-default endpoints for the axes can be specified.

axis : : = HORIZONTAL or VERTICAL

label_st : : = axis LABEL IS string

range_st : : = axis RANGE IS n_expr TO n_expr

or axis RANGE IS STANDARD

The string in a label_st will be used to label the appropriate axis. In a range_st, the first n_expr is the minimum for the axis and the second is the maximum. The given label or range remains

in effect for all plots until changed by another such statement. The default range is set by the second form of a range_st, which sets the range to STANDARD.

FOR and IF clauses

for_ok_st : : = label_st or range_st or skip_st
or eject_st or print_string_st or assignment
or precision_statement or function_assignment
or print_statement or plot_statement

for_st1 : : = for_ok_st or for_st1 FOR var=range
or for_st1 IF condition

for_st2 : : = for_ok_st or FOR var=range for_st2
or IF condition THEN [for_st2]
[eos ELSE [for_st2]]

if_st1 : : = solve_statement [IF condition]

if_st2 : : = IF condition THEN [if_st2]
[eos ELSE [st2]] or solve_statement
or IF condition THEN for_st2 eos
ELSE if_st2

st2 : : = for_st2 or if_st2

statement1 : : = for_st1 or if_st1 or st2
or system_assignment or declaration

statement : : = statement 1 eos

program : : = string of statements

A FOR or IF clause can be added to any statement except a system_assignment, a declaration, or a solve_statement, which can only take an IF clause. The IF and FOR clauses must all be on the right end of the statement or all on the left. An ELSE is paired with the last THEN. An IF clause on the right makes the statement to its left conditional; it is executed only if the condition is true. An IF... THEN... ELSE executes the statement after the THEN if the condition is true and the statement after the ELSE if it is false. A FOR clause repeats the statement to its left or right, once for each value of the var in the range. A FOR clause is especially useful for assigning values to arrays.

Procedures

Procedures may be defined in Relmath to solve systems of ordinary differential equations. The Relmath procedural language is oriented toward stating procedures to do this task, and other types of procedures cannot be written.

Within procedures, names declared with the data types INDEPENDENT VARIABLE, DEPENDENT VARIABLE, DERIVATIVE FUNCTION, STEP SIZE, STEP NUMBER, PRECISION, and ARRAY <*> have significance. Elsewhere, they are meaningless. The metalanguage names for these Relmath names are t, z, df, h, n, eps, and procedural_array, respectively. These declarations identify the significant mathematical entities in the procedure.

Procedures solve systems of the form z'=df(t, z). They compute $z_n=z(t_n)$ from previous values of z and z'. They may adjust the step size in order to maintain control of the truncation error. They may call other subprocedures.

z is a vector, since the procedures work on systems of equations. When a particular system is to be solved using a procedure, Relmath transforms the internal representation of the system to the standard form z'=df(t, z).

primitive : : = t index or z index or z ' index
or procedural_array or h or n or eps
or df (n_expr, n_expr)
or MAX (n_expr)

proc_assignment : : = procedural_array = n_expr
or z ['] index = n_expr
or h= n_expr or scalar = n_expr

index : : = <n> or <n+digit_string>
or <n-digit_string>

These rules extend the definition of an n_expr primitive and add a new assignment statement. An index is used to refer to a discrete value of t or z. t<n> is t_n, the current t. t<n+i> = $t_{n+i}=t_n+i\cdot h$, t<n-i>=$t_{n-i} = t_n-i\cdot h$, where i is a digit string. z and z' followed by an index n±i refer to z and z' at $t=t_{n\pm i}$.

Values of df, z, z', and procedural_arrays are actually vectors whose length is that of z, i. e. the number of first-order

variables in the system being solved. When these entities appear in an n_expr, all operations are applied to their components in parallel. The only exceptions are applications of df, which operate on the second argument as a vector, and the use of MAX with one argument. MAX will then return the maximum component of its argument. Note that df must have a number as its first argument.

The value of h is the current step size. The step number is n, and eps is the precision set by a PRECISION clause or statement.

A procedural assignment sets the value of z, z', a procedural array, a scalar, or h. In the first three cases the n_expr may be a vector, whose components are assigned in parallel, or a number, which is assigned to all the components of the left-side vector. Changing the value of h by a proc_assignment has side effects which will be discussed later.

with_option : : = h=n_expr or n=n_expr
    or fixed h [=n_expr]
    or scalar = n_expr
    or procedural_array = n_expr

with_clause : : = WITH list of with_options

repetition_count : : = digit_string TIMES

apply_statement : : = APPLY procedure_name
    [repetition_count] [with_clause]

An apply_statement calls a subprocedure. The action of the with_clause and the repetition_count will be explained later.

block : : = string of basic_proc_statements

do_statement : : = DO [n_expr TIMES] [UNTIL condition]
eos block END

proc_if_statement : : = IF condition THEN
[basic_proc_st1] [eos ELSE [basic_proc_st1]]
or basic_proc_st2 IF condition

basic_proc_st2 : : = proc_assignment or apply_statement
or print_statement1

basic_proc_st1 : : = basic_proc_st2
or do_statement or proc_if_statement

basic_proc_statement : : = basic_proc_st1 eos

These rules define the syntax for grouping procedural statements and conditional statements. The block of a do_statement may be executed a set number of times or until a condition is satisfied. The proc_if_statement is the standard PL/I-type conditional or the alternate Relmath format. An ELSE is associated with the last IF... THEN.

segment : : = block or SINGLE-STEP: eos block
or MULTI-STEP: eos block

proc_definition : : = BEGIN procedure_name eos
string of segments END procedure_name

statement1 : : = proc_definition

A procedure definition can have three segments. The single-step segment is headed by the line SINGLE-STEP: , and a multi-step segment by MULTI-STEP:. There can also be an initial segment, which has no header line. It immediately follows the BEGIN statement. The initial segment is applied only once at the beginning of the solution computation. The single-step segment is applied after the initial segment and after any change in h. When enough values of z have been computed, the system will use the multi-step segment. If a multi-step segment is given but no single-step, the standard single-step segment is used; it computes z_{n+1}, z_{n+2}, z_{n+3}, z'_{n+1}, z'_{n+2} from z_n and z'_n by a fourth-order Runge-Kutta-Gill formula. A segment header with no following statements also invokes the corresponding standard segment. The standard multi-step segment computes z_{n+1} from z_n, z'_n, z'_{n-1}, z'_{n-2}, and z'_{n-3} by a fourth-order Adams-Moulton predictor-corrector formula. A procedure with no header statements is considered to have only a single-step segment.

The initial segment is used to initialize variables and perhaps to set the step size. The single- and multi-step segments compute $z_{n+1}, \ldots, z_{n+K}$ and $z'_{n+1}, \ldots, z'_{n+K'}$ from z_n, z'_n, and possibly some previous values of z and z'. The single-step segment may also change z_n, $z_{n-1}, \ldots, z_{n-\ell}$ and z'_n, z'_{n-1}, $\ldots$, $z'_{n-\ell'}$. $K' \leq K$ and $\ell' \leq \ell$ are required; K or ℓ may be zero. If $K' < K$ or $\ell' < \ell$, the system will compute the missing derivatives.

K, K', ℓ, and ℓ' need not be the same each time the segment is executed. The system will analyze the multi-step segment to determine the previous values of z or z' that it needs and will not call it until these values have been computed by repeated use of the single-step segment. The single-step segment may also refer to previous values of z; these will be the last values computed before applying the single-step segment.

A procedure may call another procedure by the APPLY statement, but such calls cannot be recursive. Only a main procedure can have segments; subprocedures can not. The action of a subprocedure can be modified by a with_clause or repetition_count. A with_option with a scalar or procedural array on the left sets that scalar or array just as in an assignment before calling the procedure. If h or n is set, the value of h or n is changed for the subprocedure but is reset after the APPLY statement is finished. Changing h or n changes references in the subprocedure to z and z' so they refer to other values of z and z'. If h is changed to h·𝕞 and n to n+p, then an index n+i in the subprocedure is effectively changed to n+p+i·𝕞. h may only be changed by multiplying or dividing it by a constant power of 2, and n may be changed only by adding a constant integer to it. A repetition_count of K repeats the subprocedure K times. Each time n is advanced by the number of z's computed the last time. The system will assure that needed values of z' are computed.

If a with_clause is also given, it is used to initialize h and n, and only n is changed by the repetitions.

When h is changed by a proc_assignment, the procedure stops executing. If this happens in a subprocedure, all higher-level procedures also stop and control returns directly to the system. If h is increased, the values of z and z' which have been computed are assumed to be valid and are retained. Otherwise, these values are discarded and the single-step segment is applied with the new h and the same values of t_n, z_n as before. If h is reduced to less than $h_0 \cdot 2^{-10} = h_0/1024$, where h_0 is the original step size given in the SOLVE statement, the system terminates the solution of the system and gives an error message. Whenever h is changed, n is set to 0, so it counts from the last change. h can only be changed by a factor which is a power of 2. If FIXED is used in a with_option that sets h, proc_assignments that set h in the subprocedure are bypassed.

A procedure defines only part of the computations necessary to solve a differential system. The Relmath processor transforms the system to the form z'=df(t, z). It then repeatedly interprets the procedure, each time calculating the next few values of z and z' from their current and previous values. Each time it advances by the number of values of z computed last time. It allocates the needed storage and pushes the current and calculated values down to the previous ones, saving only those that are needed.

During the computation h may be changed, but only those function values when $t=t_0+\mathbb{m}h_0$, where $\mathbb{m}$ is an integer and t_0 and h_0 are the original t and step size given in the SOLVE statement, are saved.

A procedure is used by a USE statement.

use_statement : : = USE procedure_name

[WITH FIXED STEP SIZE]

procedure_name : : = STANDARD or STANDARD

SINGLE-STEP or STANDARD MULTI-STEP

statement1 : : = use_statement

Any SOLVE statement after the USE statement will use this procedure. The standard procedure and its segments are referred to by the names shown. The optional FIXED clause causes all proc_assignments that set h to be bypassed.

Compilation

A procedure can be compiled, which greatly reduces its execution time.

compile_statement : : = COMPILE list of procedure_names

statement1 : : = compile_statement

The procedures are translated to FORTRAN code on an intermediate data set. Standard control cards are used to call the FORTRAN compiler and link-edit the program with the rest of Relmath. Subsequent runs of Relmath will have the compiled form of the procedure and will use it until the procedure is redefined. Compiling a procedure also compiles all of its subprocedures.

A compiled procedure acts as a new primitive of the language. That is, all of its subprocedures and defined functions are frozen in the code as they are at the time of compilation. Future changes in these entities will not affect the compiled code until the procedure is recompiled.

Miscellaneous

All data in Relmath are permanent except for the values of procedural_arrays. They are preserved from one run to the next.

The REL environment in which Relmath runs provides three types of storage: a paged virtual memory, a list area, and a section of main memory. The virtual memory is used for all permanent data. The list area holds the sentence parses while they are being interpreted and temporary lists. The main memory section holds the procedural arrays and other data used by a procedure. It is also used by the PLOT statement processor to hold the points being plotted.

All statements, function definitions, differential equations, and procedures are represented as list structures. These structures are copied to the virtual memory for storage and copied back to the list area when needed. They are interpreted by the REL semantic driver and Relmath semantic routines. Compiled procedures also have associated machine code which is executed and has the same effect as interpreting the list-structure form of the procedure.

Examples of Relmath

The first section of this chapter gave a formal specification of Relmath. This section will present some examples of its use.

The first three examples are solutions to differential equations. Figure 2-2 shows the Relmath solution of a chemical reaction problem. The output is in Figure 2-3.

```
BEGIN CR
NO' = -ALPHA*NO2*NO*OL
NO2' = NO2*OL*(ALPHA*NO-LAMBDA*NO2)
OL' = -NO2*OL*(THETA+MU/NO)
END CR
ALPHA = 0.1
LAMBDA = .02
MU = 2.45E-4
THETA = 1.83E-3
BETA = .001
O3(T) = BETA*NO2(T)/NO(T)
SOLVE CR WITH INITIAL  NO=1.0,   NO2=0.2,   OL=2.0
      FOR T=0,2,...,180
EJECT
PRINT   "T      NO      NO2      OL      O3
         NO'    NO2'    OL' "
SKIP
PRINT  T,  NO(T),  NO2(T),  OL(T),  O3(T),  NO'(T),  NO2'(T),
       OL'(T)  IN FORM"          ---       --.------
--.------      --.------       .........
.........      .........       .........." FOR ALL T
VERTICAL RANGE IS 0 TO 2
HORIZONTAL RANGE IS STANDARD
HORIZONTAL LABEL IS "                    TIME(MIN)"
VERTICAL LABEL IS "                      CONCENTRATION (PPM)"
PLOT NO(T), NO2(T), OL(T), O3(T) FOR ALL T
```

Fig. 2-2 -- Chemical reaction example

T	NO	NO2	OL	O3	NO'	NO2'	OL'
0	1.00000	0.20000	2.00000	2.000E-04	-4.000E-02	3.840E-02	-8.300E-04
2	0.90778	0.28748	1.99798	3.167E-04	-5.214E-02	4.884E-02	-1.206E-03
4	0.79281	0.39319	1.99511	4.959E-04	-6.219E-02	5.602E-02	-1.678E-03
6	0.66260	0.50717	1.99122	7.654E-04	-6.692E-02	5.667E-02	-2.222E-03
8	0.52984	0.61473	1.98620	1.160E-03	-6.469E-02	4.968E-02	-2.799E-03
10	0.40771	0.70219	1.98002	1.722E-03	-5.669E-02	3.716E-02	-3.380E-03
12	0.30496	0.76226	1.97269	2.499E-03	-4.586E-02	2.293E-02	-3.960E-03
14	0.22427	0.79495	1.96417	3.545E-03	-3.502E-02	1.019E-02	-4.563E-03
16	0.16381	0.80504	1.95439	4.914E-03	-2.577E-02	4.410E-04	-5.232E-03
18	0.11979	0.79879	1.94316	6.668E-03	-1.859E-02	-6.204E-03	-6.015E-03
20	0.08817	0.78192	1.93022	8.868E-03	-1.331E-02	-1.030E-02	-6.956E-03
22	0.06555	0.75884	1.91521	1.158E-02	-9.527E-03	-1.253E-02	-8.091E-03
24	0.04933	0.73263	1.89770	1.485E-02	-6.858E-03	-1.351E-02	-9.450E-03
26	0.03760	0.70531	1.87724	1.876E-02	-4.978E-03	-1.370E-02	-1.105E-02
28	0.02905	0.67816	1.85333	2.335E-02	-3.651E-03	-1.340E-02	-1.290E-02
30	0.02274	0.65192	1.82547	2.867E-02	-2.706E-03	-1.281E-02	-1.500E-02
32	0.01804	0.62702	1.79318	3.475E-02	-2.029E-03	-1.207E-02	-1.732E-02
34	0.01450	0.60368	1.75604	4.162E-02	-1.537E-03	-1.126E-02	-1.985E-02
36	0.01181	0.58199	1.71369	4.929E-02	-1.178E-03	-1.043E-02	-2.252E-02
38	0.00973	0.56195	1.66590	5.775E-02	-9.110E-04	-9.610E-03	-2.528E-02
40	0.00812	0.54353	1.61255	6.695E-02	-7.115E-04	-8.816E-03	-2.806E-02
42	0.00685	0.52666	1.55372	7.685E-02	-5.607E-04	-8.058E-03	-3.075E-02
44	0.00585	0.51127	1.48965	8.737E-02	-4.456E-04	-7.342E-03	-3.328E-02
46	0.00505	0.49726	1.42078	9.842E-02	-3.569E-04	-6.669E-03	-3.555E-02
48	0.00441	0.48456	1.34770	1.099E-01	-2.880E-04	-6.041E-03	-3.748E-02
50	0.00389	0.47307	1.27118	1.216E-01	-2.339E-04	-5.456E-03	-3.897E-02
52	0.00347	0.46271	1.19214	1.335E-01	-1.912E-04	-4.913E-03	-4.000E-02
54	0.00312	0.45339	1.11156	1.454E-01	-1.572E-04	-4.413E-03	-4.051E-02
56	0.00283	0.44503	1.03047	1.571E-01	-1.299E-04	-3.952E-03	-4.050E-02
58	0.00260	0.43756	0.94991	1.685E-01	-1.079E-04	-3.529E-03	-3.999E-02
60	0.00240	0.43089	0.87084	1.796E-01	-9.001E-05	-3.144E-03	-3.901E-02
62	0.00223	0.42496	0.79413	1.902E-01	-7.538E-05	-2.793E-03	-3.763E-02
64	0.00210	0.41970	0.72054	2.003E-01	-6.337E-05	-2.475E-03	-3.591E-02
66	0.00198	0.41504	0.65066	2.097E-01	-5.344E-05	-2.188E-03	-3.393E-02
68	0.00188	0.41092	0.58495	2.185E-01	-4.520E-05	-1.930E-03	-3.176E-02
70	0.00180	0.40730	0.52372	2.266E-01	-3.833E-05	-1.699E-03	-2.947E-02
72	0.00173	0.40411	0.46711	2.341E-01	-3.259E-05	-1.493E-03	-2.713E-02
162	0.00130	0.38241	0.00084	2.935E-01	-4.204E-08	-2.425E-06	-6.125E-05
164	0.00130	0.38240	0.00073	2.935E-01	-3.635E-08	-2.098E-06	-5.298E-05
166	0.00130	0.38240	0.00063	2.935E-01	-3.144E-08	-1.814E-06	-4.582E-05
168	0.00130	0.38240	0.00055	2.935E-01	-2.719E-08	-1.569E-06	-3.962E-05
170	0.00130	0.38239	0.00047	2.935E-01	-2.351E-08	-1.357E-06	-3.427E-05
172	0.00130	0.38239	0.00041	2.935E-01	-2.033E-08	-1.173E-06	-2.964E-05
174	0.00130	0.38239	0.00035	2.935E-01	-1.758E-08	-1.015E-06	-2.563E-05
176	0.00130	0.38239	0.00031	2.935E-01	-1.520E-08	-8.775E-07	-2.217E-05
178	0.00130	0.38239	0.00026	2.936E-01	-1.315E-08	-7.588E-07	-1.917E-05
180	0.00130	0.38238	0.00023	2.936E-01	-1.137E-08	-6.562E-07	-1.658E-05

Fig. 2-3 -- Output from chemical reaction example

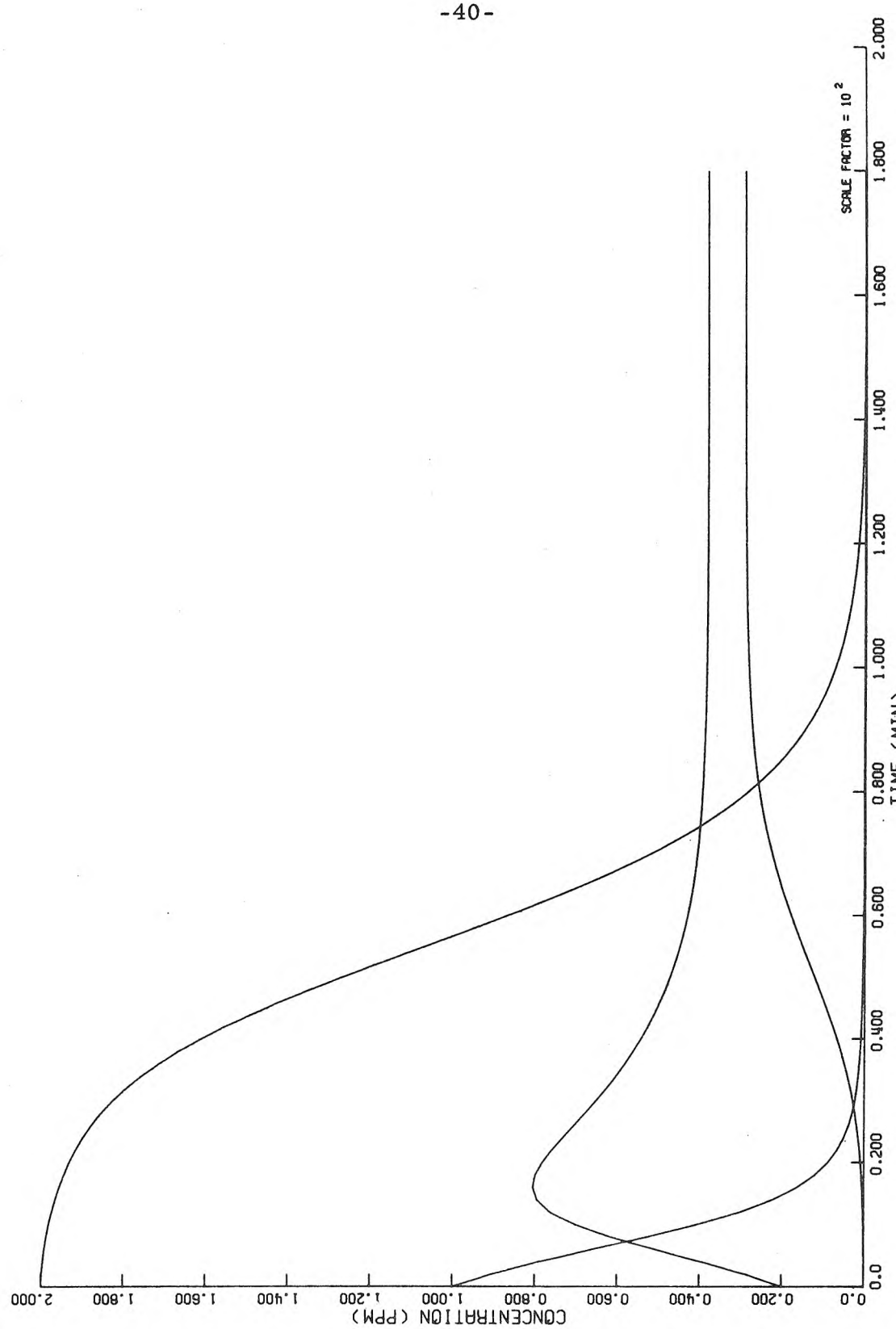

Fig. 2-3 -- continued

This example solves the differential equations

$$\frac{d[\,NO_2\,]}{dt} = [\,NO_2\,]\,[\,OL\,]\,(\alpha\,[\,NO\,] - \lambda\,[\,NO_2\,])$$

$$\frac{d[\,NO\,]}{dt} = -\,\alpha\,[\,NO_2\,]\,[\,NO\,]\,[\,OL\,]$$

$$\frac{d[\,OL\,]}{dt} = -[\,NO_2\,]\,[\,OL\,]\,(\theta + \mu/[\,NO\,])$$

which describe the interactions between NO, NO_2, and OL (olefins) in one model for the chemical reactions involved in the production of smog [Sein69, 7-13; Frie69, 1176-1177]. The parameter values were chosen by Friedlander and Seinfeld to simulate experimental observations.

Figure 2-4 shows the solution of a system that describes the dynamics of a gas absorber. The equations are taken from [Lap71, 84]. This example illustrates the use of an array function. The output is in Figure 2-5.

```
M: = ARRAY <6>
Y : = ARRAY FUNCTION <6>
Z(I, T) = M<I>+0. 16 * Y<I> (T) + 75
L = 40. 8
G = 66. 7
M <1> = . 73576500
M <2> = . 74875687
M <3> = . 75929635
M <4> = . 76774008
M <5> = . 77443837
M <6> = . 77971110
BEGIN GA
Y <1>' = (-(L+G*(M<1> + . 08 * Y<1>)) * Y<1>
        + G * (M<2> + . 08 * Y<2>)* Y<2>)/Z(1, T)
Y <I>' = (L * Y <I-1> - (L + G * (M<I> + . 08 * Y<I>)) * Y<I>
        + G * (M<I+1> + . 08 * Y<I + 1>) * Y<I+1>)/Z(I, T)
        FOR I=2, . . . , 5
Y<6>' = (L * Y<5> - (L+G * (M<6> + . 08 * Y<6>)) * Y<6>)*/Z(6, T)
INITIAL Y<1> = -. 03424992
INITIAL Y<2> = -. 06192031, Y<3> = -. 08368619, Y<4> = -. 10042889,
        Y<5> = -. 11306320, Y<6> = -. 12243691
END GA
SOLVE GA FOR T = 0 TO 40 BY . 5
EJECT
PRINT "       GAS ABSORBER"
SKIP
PRINT   T,   Y(T)   FOR ALL T
HORIZONTAL LABEL IS "          GAS ABSORBER"
VERTICAL LABEL IS " "
VERTICAL RANGE IS -. 2 TO 0
PLOT 0, Y(T)   FOR ALL T
```

Fig. 2-4 -- Gas absorber example

GAS ABSORBER

0	-0.03425	-0.06192	-0.08369	-0.1004	-0.1131	-0.1224
0.5000	-0.03425	-0.06192	-0.08364	-0.09990	-0.1080	-0.08898
1	-0.03424	-0.06186	-0.08324	-0.09755	-0.09855	-0.06911
1.500	-0.03421	-0.06165	-0.08219	-0.09365	-0.08875	-0.05631
2	-0.03410	-0.06115	-0.08042	-0.08886	-0.07984	-0.04745
2.500	-0.03387	-0.06032	-0.07806	-0.08372	-0.07205	-0.04096
3	-0.03350	-0.05916	-0.07526	-0.07856	-0.06532	-0.03598
3.500	-0.03296	-0.05768	-0.07218	-0.07356	-0.05950	-0.03203
4	-0.03227	-0.05596	-0.06894	-0.06882	-0.05444	-0.02879
4.500	-0.03143	-0.05403	-0.06564	-0.06436	-0.05000	-0.02608
5	-0.03047	-0.05197	-0.06235	-0.06021	-0.04608	-0.02378
5.500	-0.02942	-0.04981	-0.05910	-0.05634	-0.04260	-0.02179
6	-0.02830	-0.04760	-0.05594	-0.05275	-0.03947	-0.02004
6.500	-0.02713	-0.04538	-0.05289	-0.04940	-0.03666	-0.01850
7	-0.02594	-0.04317	-0.04996	-0.04630	-0.03410	-0.01713
7.500	-0.02474	-0.04100	-0.04716	-0.04340	-0.03178	-0.01590
8	-0.02355	-0.03888	-0.04448	-0.04071	-0.02965	-0.01478
8.500	-0.02237	-0.03682	-0.04194	-0.03819	-0.02770	-0.01377
9	-0.02122	-0.03483	-0.03952	-0.03584	-0.02590	-0.01284
9.500	-0.02010	-0.03292	-0.03724	-0.03365	-0.02424	-0.01199
10	-0.01903	-0.03110	-0.03507	-0.03159	-0.02270	-0.01121
10.50	-0.01799	-0.02935	-0.03302	-0.02967	-0.02127	-0.01049
11	-0.01700	-0.02769	-0.03109	-0.02787	-0.01994	-9.82313E-03
11.50	-0.01605	-0.02611	-0.02926	-0.02619	-0.01871	-9.20353E-03
12	-0.01514	-0.02461	-0.02754	-0.02460	-0.01755	-8.62760E-03
12.50	-0.01428	-0.02318	-0.02591	-0.02312	-0.01647	-8.09136E-03
13	-0.01346	-0.02184	-0.02438	-0.02173	-0.01547	-7.59136E-03
13.50	-0.01268	-0.02057	-0.02294	-0.02042	-0.01452	-7.12456E-03
14	-0.01195	-0.01936	-0.02158	-0.01920	-0.01364	-6.68832E-03
14.50	-0.01125	-0.01823	-0.02030	-0.01804	-0.01281	-6.28026E-03
15	-0.01060	-0.01716	-0.01909	-0.01696	-0.01204	-5.89828E-03
15.50	-9.97743E-03	-0.01614	-0.01796	-0.01594	-0.01131	-5.54045E-03
16	-9.39175E-03	-0.01519	-0.01689	-0.01499	-0.01063	-5.20509E-03
16.50	-8.83922E-03	-0.01429	-0.01589	-0.01409	-9.99031E-03	-4.89062E-03
17	-8.31819E-03	-0.01345	-0.01494	-0.01325	-9.38950E-03	-4.59562E-03
17.50	-7.82706E-03	-0.01265	-0.01405	-0.01246	-8.82537E-03	-4.31880E-03
18	-7.36428E-03	-0.01190	-0.01321	-0.01171	-8.29555E-03	-4.05895E-03
18.50	-6.92831E-03	-0.01119	-0.01243	-0.01101	-7.79788E-03	-3.81498E-03
19	-6.51772E-03	-0.01053	-0.01169	-0.01035	-7.33034E-03	-3.58587E-03
19.50	-6.13110E-03	-9.90238E-03	-0.01099	-9.73280E-03	-6.89104E-03	-3.37067E-03
20	-5.76713E-03	-9.31333E-03	-0.01033	-9.15093E-03	-6.47825E-03	-3.16851E-03
20.50	-5.42452E-03	-8.75906E-03	-9.71763E-03	-8.60391E-03	-6.09031E-03	-2.97857E-03
21	-5.10207E-03	-8.23755E-03	-9.13790E-03	-8.08964E-03	-5.72572E-03	-2.80010E-03
21.50	-4.79862E-03	-7.74690E-03	-8.59267E-03	-7.60613E-03	-5.38304E-03	-2.63238E-03
38.50	-5.92365E-04	-9.55406E-04	-1.05857E-03	-9.36110E-04	-6.62029E-04	-3.23617E-04
39	-5.56976E-04	-8.98323E-04	-9.95314E-04	-8.80169E-04	-6.22465E-04	-3.04277E-04
39.50	-5.23701E-04	-8.44650E-04	-9.35840E-04	-8.27571E-04	-5.85265E-04	-2.86092E-04
40	-4.92413E-04	-7.94183E-04	-8.79919E-04	-7.78116E-04	-5.50288E-04	-2.68994E-04

Fig. 2-5 -- Output from gas absorber example

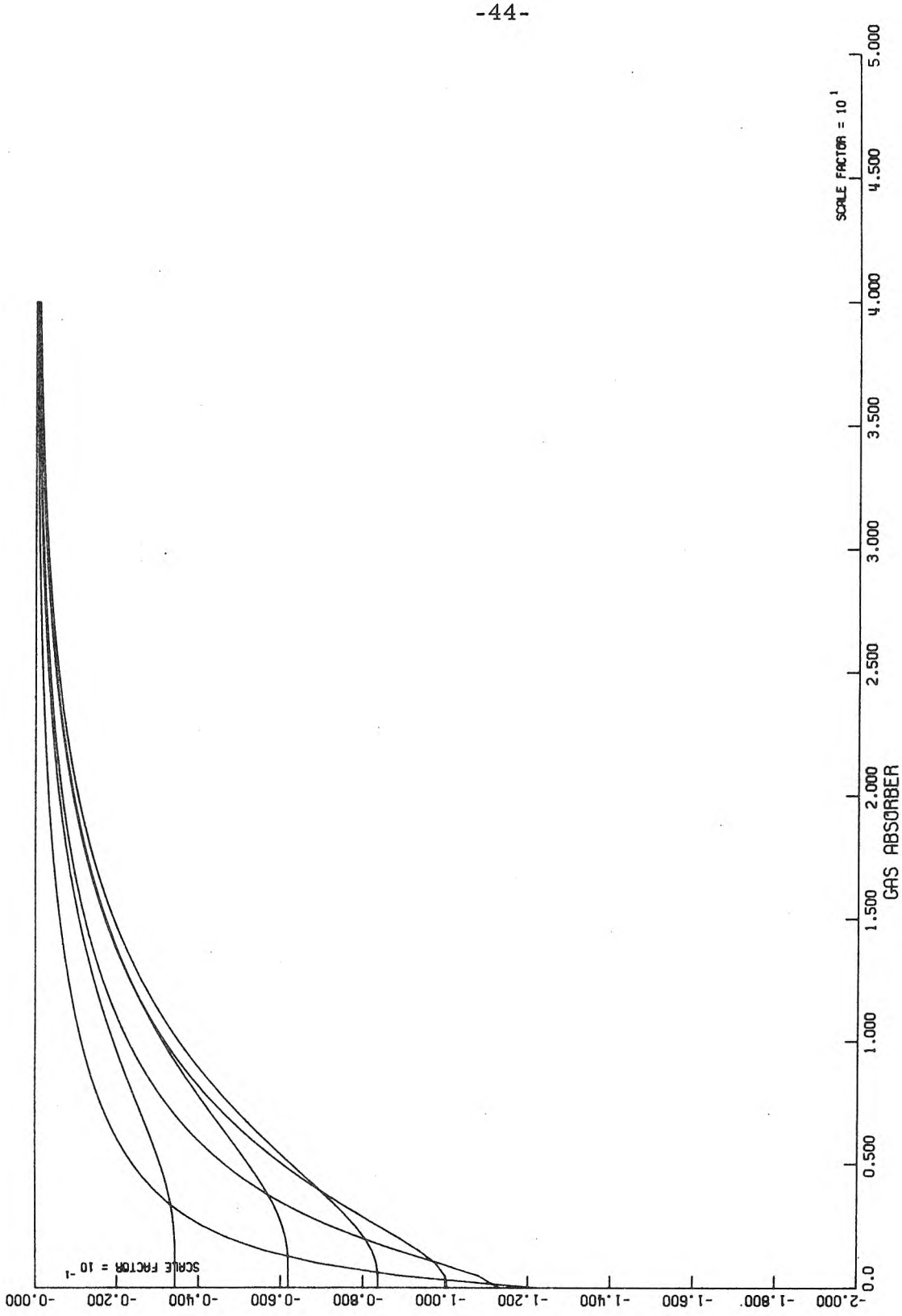

Fig. 2-5 -- continued

The final example of this group, Figure 2-5, solves for the orbit of a small body in the gravitational field of two much larger bodies that are at a constant distance from each other. The rotating coordinate system has the center of mass of the two large bodies at the origin and the two bodies on the x axis. They are a unit distance apart, and MU is the ratio of the mass of the large body to that of the whole system. The problem is due to J.M. Varah [Var71]. It has the feature that conditions are changing very rapidly right at the start of the solution. Figure 2-7 is the output.

```
BEGIN ORBIT
X' ' = X + 2*Y' - (1-MU) * (X+MU)/D1(T) - MU*(X-1+MU)/D2(T)
Y' ' = Y-2*X' - (1-MU) * Y/D1(T) - MU*Y/D2(T)
D1(T) = ((X(T) + MU)**2+Y(T)**2)**1.5
D2(T) = ((X(T)-1+MU)**2+Y(T)**2)**1.5
INITIAL X = .994, Y = 0, X' = 0
END ORBIT
MU = 0.012277471
PRECISION = 1E-5
SOLVE ORBIT WITH INITIAL Y' = -2.031732629557
      FOR T = 0 TO 11.124
HORIZONTAL LABEL IS "  ORBIT 1  "
VERTICAL RANGE IS -2 TO 2
HORIZONTAL RANGE IS -3 TO 3
PLOT Y(T) VS X(T)  FOR ALL T
SOLVE ORBIT WITH INITIAL Y' = 2.113898796645
      FOR T = 0 TO 5.437
HORIZONTAL LABEL IS "  ORBIT 2  "
PLOT  Y(T)  VS X(T)  FOR ALL T
```

Fig. 2-6 -- Orbit example

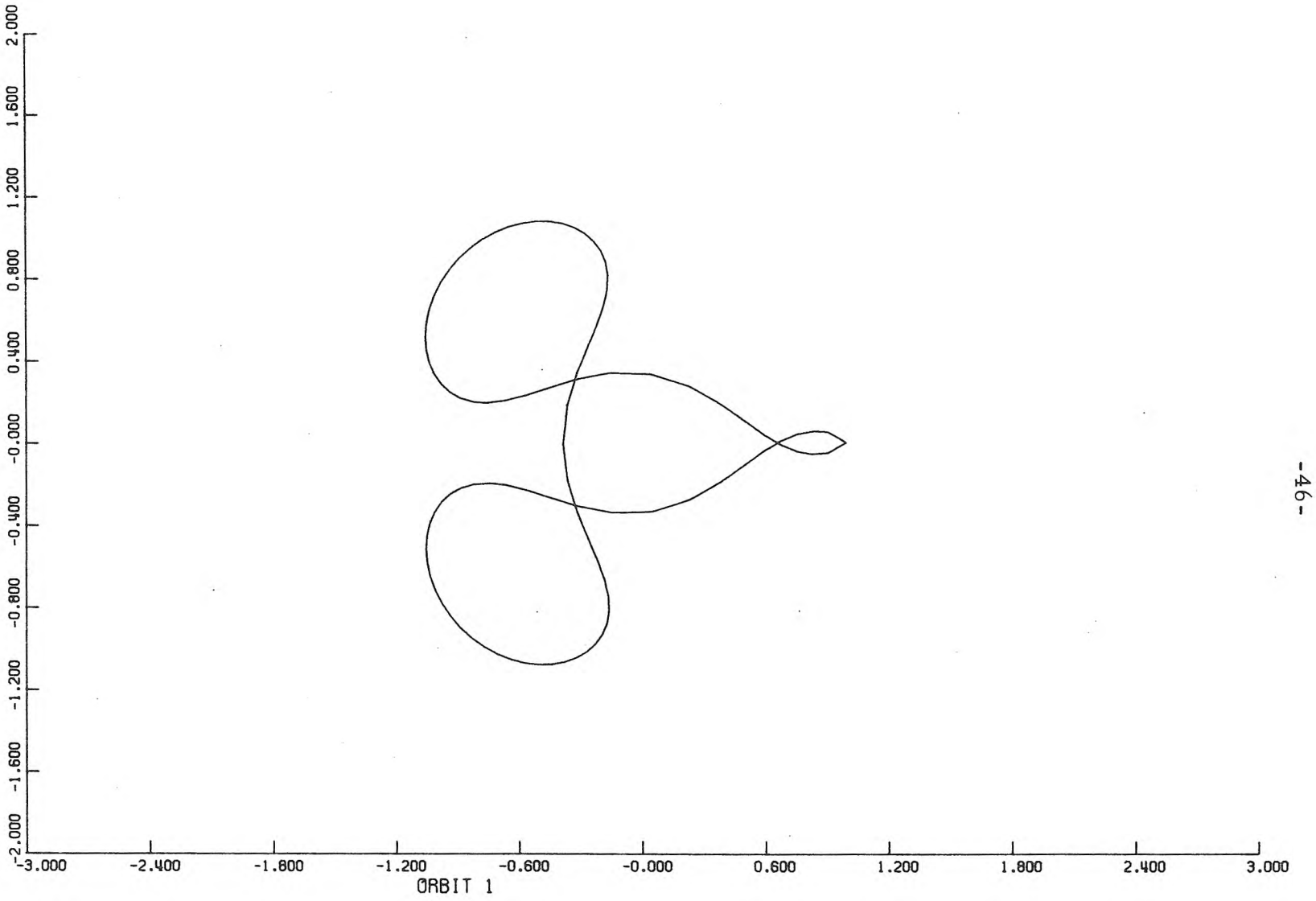

Fig. 2-7 -- Output from orbit example

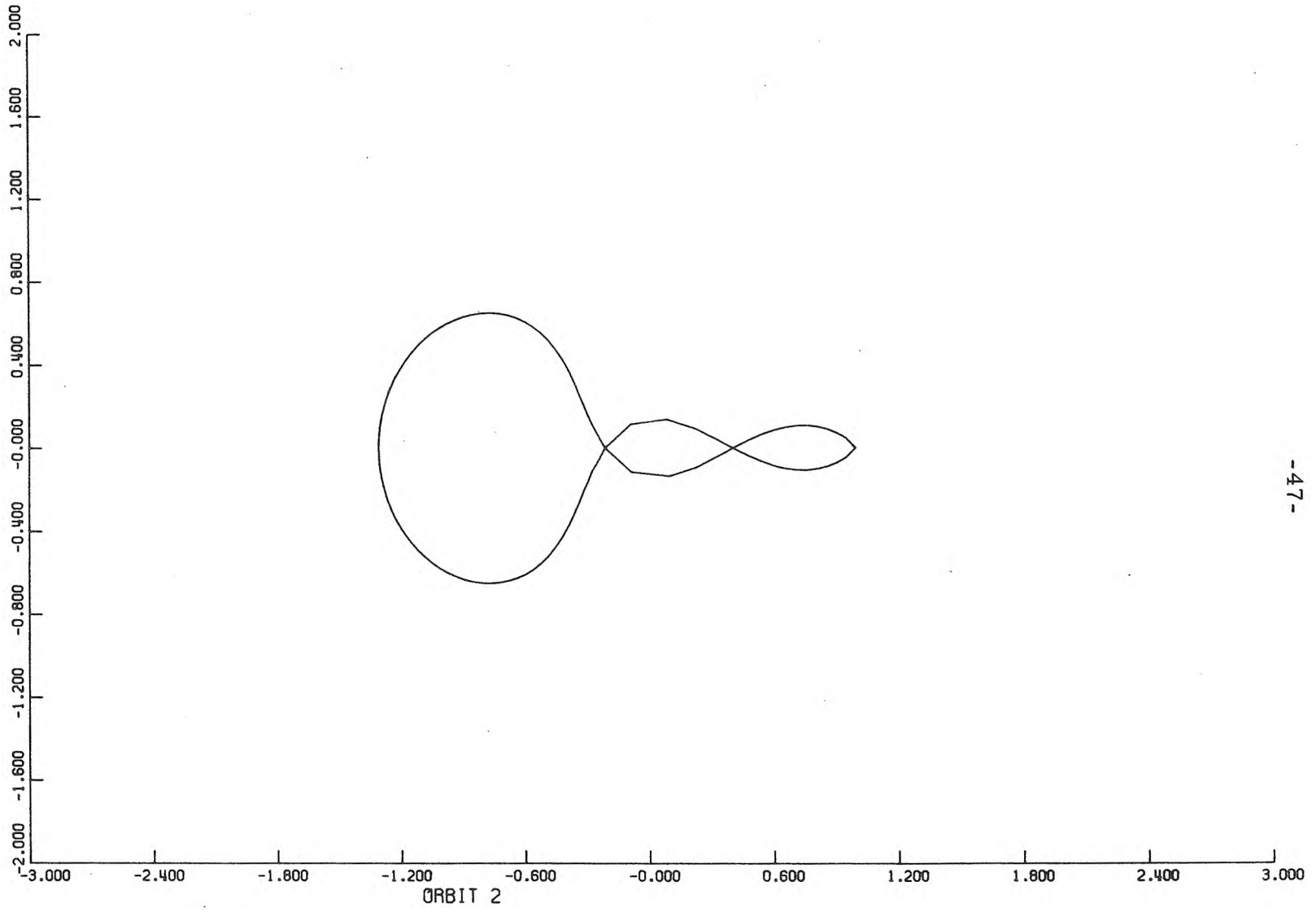

Fig. 2-7 -- continued

In these examples the differential equations were stated directly as equations in a normal mathematical form. They are solved by a simple, direct statement, and the output is also obtained easily. These features are characteristic of high-level languages for solving numerical engineering problems.

Figure 2-8 gives a few examples of the other facilities available in Relmath, not including procedures. The output is shown in Figure 2-9.

```
INT1(F, A, B, N, H) = H/3*(F(A)+F(B)
      +4*SUM FOR T=1, 3, . . . , N-1 (F(A+T*H))
      +2*SUM FOR T=2, 4, . . . , N-2 (F(A+T*H)))
INT(F, A, B, N) = INT1(F, A, B, N, (B-A)/N)  IF N¬=0,  ELSE 0
X(T) = EXP(-T)*SIN(T)
Y(T) = X(T)  IF -1E-6<=T<=1+1E-6,  ELSE 0
PRINT     " T          X          Y "
PRINT  T, X(T),  Y(T)  FOR T=-1 BY .1 TO 2
PRINT 0, 1, INT(X, 0, 1, 10) IN FORM
      "INTEGRAL OF X FROM __ TO __ = ___.______"
M : = ARRAY <10, 10>
M<I, J> = I-J FOR J=1 TO 10 FOR I=1 TO 10
PRINT   "ARRAY M"
PRINT  M IN FORM "__ __ __ __ __ __ __ __ __ __"
PRINT "SYMMETRIC" IF FOR ALL I=1 TO 10
      (FOR ALL J=I+1 TO 10 (M<I, J>=M<J, I>)),
      ELSE "ANTI-SYMMETRIC" IF FOR ALL I=1 TO 10
      (FOR ALL J=I TO 10 (M<I, J>=-M<J, I>)),  ELSE "NEITHER"
*  TABLE OF PRIMES
V : = ARRAY <10>
V<1> = 2
V(I> = FIRST J=V<I-1> TO 1000 SUCH THAT
      (FOR ALL N=1 TO I-1 (J MOD V<N>¬=0)) FOR I=2 TO 10
SKIP
PRINT  "TABLE OF PRIMES, LARGEST FIRST"
PRINT V<I> FOR I=10, 9, . . . , 1
```

Fig. 2-8 -- Examples other than differential equations

T	X	Y
-1	-2.2874	0
-0.90000	-1.9267	0
-0.80000	-1.5965	0
-0.70000	-1.2973	0
-0.60000	-1.0288	0
-0.50000	-0.79044	0
-0.40000	-0.58094	0
-0.30000	-0.39891	0
-0.20000	-0.24266	0
-0.10000	-0.11033	0
-8.326673E-17	-8.326673E-17	-8.326673E-17
0.10000	0.090333	0.090333
0.20000	0.16266	0.16266
0.30000	0.21893	0.21893
0.40000	0.26103	0.26103
0.50000	0.29079	0.29079
0.60000	0.30988	0.30988
0.70000	0.31991	0.31991
0.80000	0.32233	0.32233
0.90000	0.31848	0.31848
1.0000	0.30956	0.30956
1.1000	0.29666	0
1.2000	0.28072	0
1.3000	0.26260	0
1.4000	0.24301	0
1.5000	0.22257	0
1.6000	0.20181	0
1.7000	0.18116	0
1.8000	0.16098	0
1.9000	0.14154	0
2.0000	0.12306	0

INTEGRAL OF X FROM 0 TO 1 = 0.245836

Fig. 2-9 -- Output from examples of fig. 2-8

```
ARRAY M
 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
 1  0 -1 -2 -3 -4 -5 -6 -7 -8
 2  1  0 -1 -2 -3 -4 -5 -6 -7
 3  2  1  0 -1 -2 -3 -4 -5 -6
 4  3  2  1  0 -1 -2 -3 -4 -5
 5  4  3  2  1  0 -1 -2 -3 -4
 6  5  4  3  2  1  0 -1 -2 -3
 7  6  5  4  3  2  1  0 -1 -2
 8  7  6  5  4  3  2  1  0 -1
 9  8  7  6  5  4  3  2  1  0
ANTI-SYMMETRIC

TABLE OF PRIMES, LARGEST FIRST
   29
   23
   19
   17
   13
   11
    7
    5
    3
    2
```

Fig. 2-9 -- continued

These examples show some of the Relmath notations for sums, other repeated operators, and other operators, such as MOD, which are commonly used in engineering mathematics. These notations are close to those commonly used.

The remaining examples are of procedures. The first one, in Figure 2-10, is the fourth-order Runge-Kutta-Gill procedure [Mat70, 2].

```
BEGIN RKG
K, Q: = ARRAYS <*>
H : = STEP SIZE
N : = STEP NUMBER
Z : = DEPENDENT VARIABLE
DF : = DERIVATIVE FUNCTION
T : = INDEPENDENT VARIABLE
K = H*Z'<N>
Z<N+1> = Z<N>+. 5*K
Q = K
K = H*DF(T<N>+. 5*H, Z<N+1>)
Z<N+1> = Z<N+1>+B12*(K-Q)
Q = B1*K+C1*Q
K = H*DF(T<N>+. 5*H, Z<N+1>)
Z<N+1> = Z<N+1>+B22*(K-Q)
Q = B2*K+C2*Q
Z<N+1>=Z<N+1>+H/6*DF(T<N+1>, Z<N+1>)-Q/3
END RKG

B1 = 2-SQRT(2)
B2 = 2+SQRT(2)
C1 = -2+1. 5*SQRT(2)
C2 = -2-1. 5*SQRT(2)
B12 = B1/2
B22 = B2/2
```

Fig. 2-10 -- Runge-Kutta-Gill procedure

This is a simple procedure. It may be used as a main procedure by the statement,

USE RKG

In that case, it would be repeatedly applied to generate the solution values. The step size would be fixed, since the procedure does not modify it.

Figure 2-11 shows the standard procedure as it would be written in Relmath. The standard procedure is actually coded in FORTRAN.

```
BEGIN STAN
H = H*.25
SINGLE-STEP:
APPLY RKG WITH H = 2*H
P : = ARRAY <*>
EPS : = PRECISION
P = Z<N+2>
APPLY RKG 2 TIMES
E = MAX(|P-Z<N+2>|/MAX(.001, |Z<N+2>| ))/10
H = .5*H IF E>=EPS
APPLY RKG WITH N=N+2
MULTI-STEP:
P = Z<N>+H/24*(55*Z'<N> -59*Z'<N-1>+37*Z'<N-2>-9*Z'<N-3>)
Z<N+1> = Z<N>+H/24*(9* DF(T<N+1>, P)+10*Z'<N>
       -5*Z'<N-1>+Z'<N-2>)
E = MAX( |P-Z<N+1>|/MAX(.001, |Z<N+1>| ))*(19/270)
H = .5*H IF E>=EPS
H = 2*H IF E<.02*EPS
END STAN
```

Fig. 2-11 -- Standard procedure

This procedure has three segments. The initial one simply divides the step size by four. The single-step segment first computes Z_{n+2} using RKG with a temporarily doubled step size. It then uses RKG twice to compute Z_{n+1} again. The two

values of Z_{n+2} are compared to compute the error estimate E. If this is too large, the procedure is repeated with the step size halved. If not, RKG is used once more to compute Z_{n+3}. The multi-step segment computes Z_{n+1} from Z_n, Z'_n, Z'_{n-1}, and Z'_{n-3} using the fourth-order Adams-Moulton predictor-corrector formulas [Lap71, 180-181]. The relative truncation error is estimated by comparing the predicted and corrected values, and the step size is adjusted if necessary. Since the multi-step segment requires three previous values of Z' and the single-step segment computes three values of Z, the single-step segment will be used only once before starting the multi-step segment.

Figure 2-12 shows another procedure based on the same Runge-Kutta and predictor-corrector formulas as the standard procedure. However, STANINT uses RKG only at the start of the solution. Thereafter, it uses other means to compute the previous values needed by the multi-step segment.

```
BEGIN STANINT
SW = 0
H = H*.25
SINGLE-STEP:
IF SW=0 THEN DO
* FIRST TIME ONLY
        APPLY RKG WITH H=2*H
        P = Z<N+2>
        APPLY RKG 2 TIMES
        E = MAX ( |P-Z<N+2>| /MAX(.001, |Z<N+2>|))/10
        H = .5*H IF E>=EPS
        APPLY RKG WITH N=N+2
        HO= H
        SW= 1
        END
```

Fig. 2-12-- Standard procedure with interpolation

```
ELSE IF H>HO THEN DO
*       H INCREASED
        Z<N-1> = Z<N-2>
        Z<N-2> = Z<N-4>
        Z<N-3> = Z<N-6>
        Z'<N-1> = Z'<N-2>
        Z'<N-2> = Z'<N-4>
        Z'<N-3> = Z'<N-6>
        HO = H
        END
ELSE  DO
*       H  DECREASED
        P = 1/256*(80*Z<N>+135*Z<N-1>+40*Z<N-2>
            +Z<N-3>)+15/128*H*(6*Z'<N-1>-Z'<N>+Z'<N-2>)
        Z<N-3> = 1/256*(12*Z<N>+135*Z<N-1>+108*Z<N-2>
            +Z<N-3>)+3/128*H*(9*Z'<N-2>-Z'<N>-18*Z'<N-1>)
        Z<N-2> = Z<N-1>
        Z<N-1> = P
        Z'<N-2> = Z'<N-1>
        Z'<N-3> = DF(T<N-3>, Z<N-3>)
        Z'<N-1> = DF(T<N-1>, Z<N-1>)
        HO = H
        END
MULTI-STEP:
APPLY STANMS
END STANINT
BEGIN STANMS
P = Z<N>+H/24*(55*Z'<N>-59*Z'<N-1>+37*Z'<N-2>-9*Z'<N-3>)
Z<N+1> = Z<N>+H/24*(9*DF(T<N+1>, P)
          +19*Z'<N>-5*Z'<N-1>+Z'<N-2>)
E = MAX( |P-Z<N+1>| /MAX(.001, |Z<N+1>|))*(19/270)
H = .5*H IF E>= EPS
H = 2*H IF E<.02*EPS AND N>=5
END STANMS
```

Fig. 2-12 -- continued

When the step size has been increased, the single-step segment sets Z_{n-1}, Z_{n-2}, and Z_{n-3} equal to the old Z_{n-2}, Z_{n-4}, and Z_{n-6}, respectively, and similarly for Z'. When the step size has been halved, the needed values are computed by the following sixth-order interpolation formulas [IBM68, 338].

$$Z_{n-\frac{1}{2}} = \frac{1}{256}\,(80Z_n+135Z_{n-1}+40Z_{n-2}+Z_{n-3}) + \frac{h}{2}\cdot\frac{15}{128}\,(-Z'_n+6Z'_{n-1}+Z'_{n-2})$$

$$Z_{n-3/2} = \frac{1}{256}\,(12Z_n+135Z_{n-1}+108Z_{n-2}+Z_{n-3}) + \frac{h}{2}\cdot\frac{3}{128}\,(-Z'_n-18Z'_{n-1}+9Z'_{n-2})$$

where h is the old step size before halving. This procedure has been found to be faster than STAN in some cases where the step size was changed often.

In Figure 2-13 we have a procedure based on different formulas.

```
BEGIN HYBRID
SINGLE-STEP:
K1, K2, K3, K4, K5, K6: = ARRAYS <*>
K1 = H*Z'<N>
K2 = H*DF(T<N>+.5*H,  Z<N>+.5*K1)
K3 = H*DF(T<N>+.5*H,  Z<N>+.25*K1+.25*K2
K4 = H*DF(T<N>+H, Z<N>-K2+2*K3)
P = Z<N>+(K1+4*K3+K4)/6
K5 = H*DF(T<N>+2/3*H, Z<N>+(7*K1+10*K2+K4)/27)
K6 = H*DF(T<N>+.2*H, Z<N>+(28*K1-125*K2
        +546*K3+54*K4-378*K5)/625)
Z<N+1> = Z<N>+(14*K1+35*K4+162*K5+125*K6)/336
E = MAX( |P-Z<N+1>|/MAX(.001, |Z<N+1>|))
H = H/2 IF E>=EPS
K5 = H*DF(T<N>+1.5*H, Z<N>+.375*K1+1.125*K4)
K6 = H*DF(T<N>+2*H, Z<N>+(-6*K1+4*K2
             +24*K3-24*K4+16*K5)/7)
Z<N+2> = Z<N>+(7*K1+32*K3+12*K4+32*K5+7*K6)/45
MULTI-STEP:
P = DF(T<N>+.7*H, -5.7937275*Z<N>
        +3.5721*Z<N-1>+3.2216275*Z<N-2>
        +H*(3.6869175*Z'<N>+6.07257*Z'<N-1>+.9558675*Z'<N-2>))
Q = .533624116608*Z<N>+.449425795053*Z<N-1>
       +.0169500883392*Z<N-2>+H*(.702480859835*Z'<N>
        +.194238030901*Z'<N-1>+.0866070809949*P)
P = Z<N>+H*(.309523809524*Z'<N>
        -.00980392156863*Z'<N-1>+.700280112045*P)
Z<N+1> = Z<N>+H/6*(Z'<N>+DF(T<N+1>, P)
        +4*DF(T<N>+.5*H, Q))
E = MAX( |P-Z<N+1>|/MAX(.001, |Z<N+1>|))*(.0347/.209)
H = H/2 IF E>=EPS
H = 2*H IF E<.02*EPS
END HYBRID
```

Fig. 2-13 -- Procedure with Sarafyan and Kohfield-Thompson formulas

The single-step segment uses a self-embedding Runge-Kutta scheme of Sarafyan [Lap71, 72-73]. This scheme computes Z_{n+1} to fourth order in P and then to fifth order. The two values are compared to estimate the error. If it is below the bound, Z_{n+2} is also computed. This method needs only 9 evaluations of

the derivative function to compute Z_{n+1}, Z_{n+2}, and their derivatives, while the method used in STAN required 11.

The multi-step segment uses a hybrid method of Kohfield and Thompson [Koh67]. A value of $Z_{n+0.7}$ is predicted, and P is set to its derivative. Then a predicted Q = $Z_{n+0.5}$ is computed using P. P is changed to a predicted Z_{n+1}. Finally, a corrector formula due to Gragg and Stetter [Gra64] is applied to obtain Z_{n+1}. The corrector is fourth-order, but it is approximately 76 times as accurate as the Adams-Moulton fourth-order corrector. This accuracy essentially compensates for the increased computation, since it allows a step size which is about twice that useable with the Adams-Moulton formula for the same accuracy. Kohfield and Thompson give a family of formulas up to order 14; they have found that their sixth-order formulas require about two-thirds the compute time of the corresponding Adams-Moulton formulas.

These procedures are written in a simple, direct style. The mathematical operations are directly expressed as they would be in a text describing the methods. Hence, this is a high-level language for writing algorithms of this kind.

The REL System

In this section we will describe the aspects of REL which are relevant to the design of Relmath. REL is also discussed in [Dos71; Thomp69].

The REL system used to implement Relmath is a batch system which runs under a standard IBM OS/360 on a 370/155. It consists of a group of programs, including a parser, a semantic interpreter, a list processor, and a paging controller. This system supports the development of high-level languages.

A language implemented under REL has a dictionary which describes its syntax. The syntax rules are written in a special language that is processed by the REL dictionary builder program. An example is

```
SYN:  'NU=> 'NU +NU'
CHK:  ,-ASF
SET:  0+ASF
SEM:  ADD
```

This rule specifies the syntax for addition. NU is the part-of-speech name for a number. ASF is the add-subtract feature; it must be off in the second input NU phrase and is set on in the output NU phrase. Hence, sums are collected from left-to-right. ADD is a semantic routine, a program written in assembler code that performs the addition. Every rule has a semantic routine which performs the semantic transformation indicated by the syntax.

The parser uses the dictionary to parse the input statements. It operates in a bottom-up manner. When the parser is done, it outputs the parse tree of the sentence. Each node, which corresponds to a phrase, contains a pointer to the semantic routine of the rule that was used to construct that node. Thus, if the summation rule was applied to construct an NU phrase, it will contain a pointer to ADD and pointers to its constituents.

During the semantic phase, the REL interpreter, SEM, is called to evaluate the sentence phrase. It recursively calls itself to evaluate the constituents of any phrase, and then calls the semantic routine. This program replaces the phrase's pointers to itself and the constituents with the output value of the phrase. For instance, ADD obtains the values of the constituent phrases, adds them, and makes the sum the value of the output phrase. The semantic routines may also produce printed or plotted output.

A rule can also have a syntax routine. This routine is called by the parser when the rule is applied. It can apply tests to determine whether the rule is really valid, insert names in the dictionary, create and bind variable phrases, and otherwise modify the parsing graph. A condition routine creates a variable by replacing the semantic routine pointer in the output phrase by an identifier of the variable and marking the phrase. The parser keeps a list in each phrase of the variables that are free in it.

A condition routine binds a variable by deleting it from the variable list of the output phrase. It also calls an REL utility to obtain the refresher stack, a list of the phrases where the variable appears and those phrases that depend on these. This stack is saved in the output phrase.

A semantic routine can be marked as a generator by adding "(G)" after its name in the SEM: statement of a rule. A flag is placed in the output phrase's routine pointer, and SEM will not evaluate the phrase's constituents before calling the routine. The routine can then modify the unevaluated constituent phrase trees and call SEM to evaluate them.

Generators are usually used in conjunction with variables. For instance the rule

```
SYN:  'NU'=>' S  U  M  F  O  RNU  =FR (NU  )'
CND:  FORBIND
SEM:  SUM(G)
```

recognizes the sum notation in Relmath. FR is a FOR range, and the left NU must be a variable phrase. The syntax routine FORBIND binds this variable and computes its refresher stack in the right NU. SUM uses SEM to evaluate the FR. It then uses the refresher stack to repeatedly set the variable phrase in the right NU to successive values in the range and calls SEM to evaluate the NU. The refresher stack is also used to reset the semantic routine and constituent pointers of the phrases that depend on the variable before each call to SEM. Other phrases

are not reset or re-evaluated. The values of the NU phrase are added and the sum is the value of the output phrase.

In Relmath all names become variable phrases. Those not bound by specific syntax routines are bound by a general routine associated with the rule to recognize a sentence. This rule's semantic routine is a generator that sets these variable phrases to the values of their names and then calls SEM to evaluate the sentence.

The REL system has three types of memory: a list area, a paged virtual memory, and an unstructured block of main memory. The list area is divided into three-word list elements. Any word can be used as a list link or as data; the first byte of the element indicates how the words are used. A macro obtains the next element from the free list. If the list is empty, it calls a garbage collection routine that finds all the unused elements and puts them on the list. The virtual memory consists of 1024-byte pages which may be either in core or on the disk. A REL subroutine will load a page given its virtual memory address and will return its core address. When the core area allocated to pages is filled, the page referenced the longest time ago is replaced by the new page. Thus, the most recently referenced pages remain in core. REL subroutines can be used to copy list structures from the list area to pages and back.

REL is a batch program that runs in two job steps. The first step does the syntactic processing of each input sentence. It saves the parses on an intermediate data set. The second step does the semantic processing. It reads the parses back into core and calls SEM to evaluate each one. In each step the programs, including both the REL routines and the syntax or semantic routines, are link edited into a load module. Thus, the paging is used only for the data, not the programs.

III. LANGUAGE DESIGN

In this chapter we will discuss the design of a language for solving numerical engineering problems.

The User and the Application Area

It is first necessary to characterize the intended users of the language and its application areas. In this thesis we are generally concerned with users who are scientists and engineers who are not highly skilled in the techniques of numerical analysis or computer programming. Thus, they are not expected to develop new numerical methods or to do extensive programming. Relmath was designed for these people.

An application area must be chosen for the language. For Relmath, ordinary differential equation initial value problems were selected as the primary application area. These problems are of interest to the users, and their numerical treatment is understood. It is not easy for an unsophisticated user to solve systems of such equations using a conventional programming language.

This problem area has two parts. We wanted to facilitate the statement of the equations, the initial values, and the domain of the solution. We also wanted to facilitate the statement of the numerical method for solving such equations. For the latter part, we wanted to develop a higher-level language than FORTRAN

for a class of methods that included at least those of interest to students and other unsophisticated users.

The definition, evaluation, and printing or plotting of functions was selected as a secondary application area for Relmath. Almost all of these capabilities were required to support the solution of differential equations, since we wanted to be able to define auxiliary functions and to manipulate and display the solutions.

Other languages, of course, will have other problem areas. One language with a broad application area is NAPSS [Sym68; Sym69]. It includes operators to solve ordinary differential equations, systems of linear equations, and single nonlinear equations, as well as operators to do integration, differentiation, and interpolation. POSE [Schl67] and AMTRAN [Seitz68; Rein70] have operators for a variety of problems similar to those handled by NAPSS. AMTRAN and NAPSS also include the ability to define procedures in a language similar to that of PL/I. Examples of languages with more restricted application areas are SALEM [Mor68] and PDEL [Car70] for certain classes of partial differential equations, Gear's language for ordinary differential equations [Gear66], and the Lincoln Reckoner [Stow66] for matrix manipulations.

The language's application area should be of interest to the user group. The interests of this group determine the types

of numerical problems that should be supported and the level of support that can be provided. Generally, the narrower and more specific the users' interests, the more the language can be tailored to their problems.

This leads to a general principle of language design. Those computations which are common to all problems in an application area should be made implicit in the language. The user should not have to specify these. Indeed, the computations which are implicit essentially define the application area of the language, since problems that violate these implicit assumptions can not be stated. The explicit statements that the user writes serve to specify the problem within the class of allowable problems. The more implicit knowledge about the application area that is included in the language, the higher the level of support for this application area.

Another general principle is that the language should be natural. That is, its syntax and semantics should correspond directly to the users' normal notation and its meaning within the application area. For instance, differential equations should be expressible directly as equations, not in terms of other language constructions, and they should be solvable with a simple statement. However, the language need not support all variations of normal mathematical notation nor a significant portion of normal English.

We will give examples of the application of these principles in the design of Relmath.

Standard Problems

The standard problems of a language are those that can be solved by built-in operators of the language. The syntax for stating these problems should be natural.

In Relmath, initial value problems for systems of differential equations are obviously standard. The differential equations can be stated directly as equations in a natural syntax and solved by a simple statement. We shall discuss the design of these aspects of the language more thoroughly in order to show the considerations involved.

Of course, it was necessary to have equations as a data type. Some syntax for derivatives had to be selected. The prime notation (X', X'', etc.) was selected because it is commonly used and simple. It was also decided that the unknown functions in the equations should not have arguments. This is another common usage. For instance, the equations in the ORBIT system of Figure 2-6 were originally written as

$$x'' = x+2y'-(1-\mu)\frac{x+\mu}{((x+\mu)^2+y^2)^{3/2}}-\mu\frac{x-1+\mu}{((x-1+\mu)^2+y^2)^{3/2}}$$

$$y'' = y-2x'-(1-\mu)\frac{y}{((x+\mu)^2+y^2)^{3/2}}-\mu\frac{y}{((x-1+\mu)^2+y^2)^{3/2}}$$

[Var71]. This convention also allows the processor to determine which functions are unknown from the syntax alone. NAPSS always requires an argument after a derivative. This is more uniform -- all functions have arguments -- but NAPSS also requires that the unknown functions be named in its SOLVE statement. For example,

SOLVE Y''(X)+Y'(X)+EXP(X)Y(X)=SIN(2X), FOR Y(X),
ON 0<X<2, WITH Y'(X ← 0) ← 1, Y(X ← 0) ← 0;

[Sym69]. For systems with several unknowns, naming them in each SOLVE statement would be cumbersome.

Arrays and array functions were included because they are a natural way to express problems with many homogenous elements. The gas absorber system of Figure 2-4 is an example. However, the size and usage of array functions were restricted to avoid implementation problems, as we shall see.

The ability to store equations and initial conditions in named systems is important. Often, the user would like to enter a system and then solve it repeatedly while varying some parameter. Subsystems were allowed so that hierarchical systems could be stated. In such systems, equations and initial conditions that belong together can be placed in a subsystem. Hence, part of the system can be replaced without changing the rest.

Two forms of system definitions are allowed. The BEGIN. . . END form is more general and works well for large

systems. However, it seemed cumbersome for small systems, so the short form illustrated by

EQ1: Y'=3*X, X' = -2*Y

was also included.

NAPSS allows single equations, but not initial values, to be named. It has no subsystems.

Differential equations often utilize auxiliary functions. Examples are the ORBIT system of Figure 2-6 and the GA system of Figure 2-4. Consequently, function definitions were included in Relmath. Their form is simple and matches common usage.

NAPSS provides an example of a language with several types of standard problems. Integrals and derivatives can be evaluated by notations such as $\int$X * SIN(X), (X ← 0 TO 1) and f'(3.7). Several types of equations, including ordinary differential equations, single nonlinear equations, and linear systems, can be solved by the SOLVE statement. The processor can determine the type of equation, but it is faster if the user specifies it. The user can also specify other information, such as the number of solutions wanted and the neighborhood where a solution is to be found, which may be helpful to the processor or may be needed to ensure that the desired solutions are obtained. All these problems can be expressed in a natural syntax.

In order to solve an equation it is usually necessary to write it in a particular form. This form isolates certain functions

and other entities that can be used by the numerical method used to solve the equation. For instance, a differential equation must be in the form $Z' = f(t, Z)$ in order to apply most numerical methods to solve it. This form isolates the independent variable t, the dependent variable Z and its derivative, and the derivative function f. The numerical method is stated in terms of these entities. The original equation, however, may have quite a different form. Language processors with facilities to solve equations must be able to make these transformations between the equation and the form required by the numerical method.

In Relmath, a system of differential equations is transformed by the computer to the form $Z' = f(t, Z)$ in order to solve it. Several steps are involved. The individual unknown functions and array functions must be collected. If a function appears with a derivative higher than first order, intermediate variables must be introduced to make the system first order. This is a standard reduction; if X has maximal order n, then U'_n replaces $X^{(n)}$, U_K replaces $X^{(K-1)}$ for $K=1, \ldots, n$, and the equations $U'_K = U_{K+1}$ are added for $K=1, \ldots, n-1$. The functions and intermediate variables are collected to form Z, the dependent variable vector, and space for each function is allocated in Z. Finally, we must solve the equations in order to compute the maximal derivative of each function.

The last step is the most difficult in its full generality. In Relmath, it was decided to restrict the system of equations to be quasi-linear and triangular in the maximal derivatives. This greatly simplified the program to solve for the derivatives. The time gained was then used to develop the procedural part of the language. Due to the prototype nature of the current Relmath language and because the problems of solving more general systems seemed to be better understood than those of higher-level procedural languages, it was felt that effort should be concentrated on the latter.

We shall present the current method and then later discuss the problem of changing it to handle other types of systems. During the syntax phase, names and derivatives which appear without an argument list are parsed as numeric variables. These are bound when the equation is recognized. When the SOLVE statement is executed, the processor scans the lists of variables in the equation and builds a global list of all the names that are differentiated. These names are the unknown functions; all other names refer to their current meanings. The storage for the function names is initialized by setting some flag bits and saving the domain of the function. The domain is obtained from the FOR clause of the SOLVE statement. For each unknown function F, the maximal order M_F of any of its derivatives in the system is found. For each F an equation is found that can be solved

for $F^{(M_F)}$. This is done by scanning the list of equations for one that has a unique F such that $F^{(M_F)}$ occurs in the equation and F does not already have an equation. This equation is associated with F in the global function list, and the list is reordered in the order in which equations are found for the functions. The process is repeated until all functions and equations are accounted for.

The method is slightly different for an array function. It is parsed as an array variable in the syntax phase. During the process of associating functions and equations, an array function can have more than one equation. This is necessary since different equations may apply to different array elements. The subscripts of the maximal derivatives are ignored; this requires that all such derivatives in one equation have the same subscript.

After equations are assigned to the functions, the global function list is ordered so that no maximal order derivative depends on a maximal derivative of a function later in the list. From the declared sizes of the array functions, the number of ordinary functions in the list, and the maximal derivative orders, the total amount of storage needed for the unknown function vector is computed. Each function is assigned a displacement in this vector. The vector holds $F^{(0)}, \ldots, F^{(M_F-1)}$ for each function F. Storage for the vector, its derivative, and that needed by the procedure to be used is allocated in the unstructured storage area of REL.

The unknown function vector is passed to the procedure as the dependent variable vector Z. When Z' must be computed, a subroutine of the SOLVE statement processor is called. This routine copies $F^{(1)}, \ldots, F^{(M_F-1)}$ from Z to Z' for each function F. It then uses the global function list and the refresher stacks of the equations to insert the values of the functions in Z into the phrase structure of the equations. It replaces each number variable with a two-element structure containing a value v and a coefficient c=0. For any number phrase in the equation, its v and c elements represent cx+v, where x is the function being solved for in this equation. A structure with v=0 and c=1 and the address of x replaces a numeric variable for the maximal order derivative x. The arithmetic semantic routines combine these structures, calculating the new c and v elements as if the operation were performed on the two linear terms. For example, the multiplication routine combines (c_1, v_1) and (c_2, v_2) to obtain $(c_1v_2+c_2v_1, v_1v_2)$, it requires $c_1c_2=0$. The routine also passes the x address on. The semantic routine that recognizes an equation solves for x by $x=(v_2-v_1)/(c_1-c_2)$ and puts this value at the address of x in Z'.

For an array function, the array variable for its maximal order derivative is replaced by a structure that has the address of the derivative array in Z'; the other derivative variables are replaced by structures pointing to the arrays in Z. Bits are set

in each structure to show that the arrays are in core and whether they are for the maximal order derivative, and a pointer to the array function's dimensions is also placed in the structures. The semantic routine that combines an array and a subscript list to get a number will produce structures similar to those substituted for numeric variables. That is, the numbers for the known derivatives will have c=0; those for the maximal derivative will have c=1, v=0, and the address of x pointing to the appropriate array element. The remaining processing is the same as for numeric functions.

By these means Relmath computes the value of the derivative vector Z'. To solve the problem using a language which did not directly support differential equations, such as FORTRAN, the user would construct Z in much the same way and assign the functions to displacements in Z. He would then explicitly solve the equations himself and write a program that computed the derivative vector. In this program he would refer to the functions by their displacements rather than their names.

We mentioned that the subscripts of the maximal derivatives of array functions are ignored during the process of assigning equations to functions. To handle array functions with full generality would require a program to compare subscripts to determine when they are equal, taking into account the fact that the subscripts may contain indices whose values are generated by FOR clauses with different ranges. Hence, it must determine

for which values of these indices, if any, the subscripts are equal. To do this for arbitrary expressions would require deductive capabilities. Deductive processes are poorly understood now, in that they require very large amounts of space and time. Hence, it is not yet possible to effectively utilize them to solve this problem.

The problem may be resolvable by restricting the form of subscripts to, say, constants or $I \pm c$, where I is the index of a FOR clause and c is a constant. But even then the programming task would be great. Again, it was decided to accept this restriction and work instead on the procedural language.

We emphasize that the transformation of equations into a standard form should be done implicitly, as it is in Relmath. We are discussing languages for solving numerical problems, and the user should not have to do any explicit symbol manipulation.

The restriction on the form of the differential equations could be removed in Relmath. The processor could be extended to handle nonlinear and also stiff systems. However, these extensions would have entailed a large amount of programming. It would be necessary to recognize and process special cases, such as linear systems or those in which the maximal derivatives are explicitly solved for. The general case would require symbolic differentiation programs to develop the formulas for the Jacobians needed for nonlinear and stiff systems. These

programs could be written following the techniques used in MATHLAB [Eng69; Eng71] or MACSYMA [Mar71b]. These are two good languages for explicit symbolic manipulation of expressions. However, this would be a major extension of the symbolic processing now done in Relmath. The problem of recognizing the equality of array subscripts would also be more severe than at present. It was decided to concentrate on other parts of the language in this prototype system.

Procedural Languages

In this section we are concerned with linguistic facilities for writing procedures. In writing a procedure, the user is describing the method to be used to solve a problem. When standard facilities such as those discussed in the last section are used, he need only state the problem and ask for its solution; the method is built-in.

Previous languages for describing numerical procedures have been similar to FORTRAN, PL/I, and ALGOL in the following sense. They have been designed to express any numerical algorithm, and hence they have provided a low level of support for a very wide class of algorithms. Furthermore, they can manipulate scalars, arrays, and functions, but not equations. While all numerical problems and algorithms can be stated without explicitly using equations, doing so is less

natural than using equations. Even NAPSS, which does allow equations for stating standard problems solved by its built-in procedures, does not allow a user-written procedure to refer to equations. Therefore, if a user writes a procedure for solving differential equations, say, he can not write the equation itself as input to the procedure. Instead, he must transform it to some standard form and pass only the required functions and scalars to the procedure.

One purpose of Relmath was to investigate an alternate type of language. The Relmath procedural language was designed to provide a high level of support for a class of algorithms for solving differential equations. It was also designed to allow the user-written procedures to solve equations written in the same form as for the built-in procedure. To accomplish this, the common parts of the algorithms in the desired class were identified. These computations were made implicit in the language, in accordance with the first general design principle stated in the first section of this chapter. The language was designed to allow the rest of the algorithms to be expressed in a natural style. Finally, the procedures can refer to the derivative function and dependent vector which result from the transformation of the equations to the form $Z'=f(t, Z)$. Hence, the algorithms are written in terms of t, f, Z, and Z', which are

commonly used to describe such algorithms, but they can still solve equations written in a natural way.

The algorithms to be supported are those based on Runge-Kutta, predictor-corrector, and hybrid formulas. These include the methods commonly covered in introductory texts in numerical analysis and in subroutine libraries such as Caltech's MODDEQ [Mat70] and IBM's HPCG [IBM68] programs. They are of interest to students, to users who want to compare these methods, and to those who prefer one of these methods. For instance, Hull and Creemer have found high-order Adams-Moulton formulas to be desirable [Hull63]. This class of methods is too restricted for the sophisticated user or numerical analyst, but Relmath was not intended to be for their use.

As mentioned earlier, the Relmath procedural language was designed as an alternative to general programming languages. Although it is limited to a certain class of methods, it is intended to provide a high level of support for these methods by incorporating their common parts implicitly in the language processor. It should show the level that can be attained and that should be a goal for future languages that have a broader scope. Examples of the basic formulas for the Relmath methods are:

Runge-Kutta, fourth order [Lap71, 49]

$$K_1 = hf(t_n, z_n)$$

$$K_2 = hf(t_n + \tfrac{1}{2}h,\ z_n + \tfrac{1}{2}K_1)$$

$$K_3 = hf(t_n+\tfrac{1}{2}h,\ z_n+\tfrac{1}{2}K_2)$$

$$K_4 = hf(t_n+h,\ z_n+K_3)$$

$$Z_{n+1} = Z_n+\frac{1}{6}(K_1+2K_2+2K_3+K_4)$$

Predictor-corrector, Adams-Moulton fourth order [Lap71, 180-181]

$$\bar{Z}_{n+1} = Z_n+(h/24)\ [\ 55Z'_n-59Z'_{n-1}+37Z'_{n-2}-9Z'_{n-3}]$$

$$Z_{n+1} = Z_n+(h/24)\ [\ 9\bar{Z}'_{n+1}+19Z'_n-5Z'_{n-1}+Z'_{n-2}]$$

$$T = (19/270)\ \left|Z_{n+1}-\bar{Z}_{n+1}\right|$$

Hybrid, Kohfield-Thompson fourth-order [Koh67]

$$\bar{\bar{Z}}_{n+.7} = -5.7937275Z_n+3.5721Z_{n-1}+3.2216275Z_{n-2}$$
$$+h[\ 3.6869175Z'_n+6.07257Z'_{n-1}+.9558675Z'_{n-2}]$$

$$\bar{\bar{Z}}_{n+.5} = .533624116608Z_n+.449425795053Z_{n-1}$$
$$+.0169500883392Z_{n-2}+h[\ .702480859835Z'_n$$
$$+.194238030901Z'_{n-1}+.0866070809949\bar{\bar{Z}}'_{n+.7}]$$

$$\bar{Z}_{n+1} = Z_n+h[\ .309523809524Z'_n-.00980392156863Z'_{n-1}$$
$$+.7002801120045\bar{\bar{Z}}'_{n+.7}]$$

$$Z_{n+1} = Z_n+(h/6)[\ \bar{Z}'_{n+1}+4\bar{\bar{Z}}'_{n+.5}+Z'_n]$$

$$T = (.0347/.209)\ \left|Z_{n+1}-\bar{Z}_{n+1}\right|$$

T is the absolute local truncation error. $\bar{Z}'_{n+1}$ denotes $f(t_{n+1}, \bar{Z}_{n+1})$; primes applied to $\bar{\bar{Z}}_{n+.7}$ and $\bar{\bar{Z}}_{n+.5}$ denote similar applications of f. f(t,Z), of course, computes Z'.

Several similarities among these formulas were noted and used in designing Relmath. First, they are written in terms of the independent variable t, the dependent variable Z, the step size h, and the function f that computes Z'. Second, they compute Z_{n+1} from Z_n and possibly previous values of Z; the repetition of the formula to generate the total solution is implied. Third, they are written for single equations, but they are extended to systems by applying the formulas in parallel to each component of Z. Fifth, they use temporary variables, which become vectors in the case of a system. Sixth, any multi-step formula must be accompanied by some single-step formula to get started.

Other formulas exhibit features not shown in these. Sarafyan's method uses special Runge-Kutta formulas to compute Z_{n+1} and Z_{n+2} in one step [Lap71, 72-73]. Some methods iterate a corrector until convergence is achieved or for some maximum number of times. IBM's HPCG algorithm uses a Runge-Kutta scheme to start and then refines the values of Z_1, Z_2, and Z_3 by interpolation formulas before applying a multi-step method [IBM68, 337-343].

The hardest part of programming these algorithms is controlling the step size. Practical methods do not simply repeat the formulas with a fixed step size. They estimate the truncation error and adjust the step size so that the error is kept below some prescribed limit. Predictor-corrector and some Runge-Kutta schemes,

such as Sarafyan's, compute the truncation error estimate directly. Most Runge-Kutta schemes, however, do not. It is necessary to apply the scheme to compute Z_{n+1} and then again to get Z_{n+2} from Z_{n+1}. The scheme is applied a third time with a doubled step size to compute Z_{n+2} again from Z_n. The difference between the two values of Z_{n+2} gives an error estimate.

When the step size is changed in a multi-step scheme, the required previous values of Z and Z' must be made available at the values of t implied by the new h. One way of doing this is to start over with the Runge-Kutta scheme, just as at the beginning. Another is to make use of the known previous values and compute the others by interpolation. For example, suppose Z_{n-1}, Z_{n-2}, and Z_{n-3} are needed. If h is doubled and Z_{n-4}, Z_{n-5}, and Z_{n-6} have also been saved, then the three values needed are just Z_{n-2}, Z_{n-4}, and Z_{n-6}. These become Z_{n-1}, Z_{n-2}, and Z_{n-3} for the new h; h can not be doubled again for three steps. If h is halved, then Z_{n-1} becomes Z_{n-2}. The new values of Z_{n-1} and Z_{n-3} are computed by interpolation. This method requires that h only be halved or doubled, but that is commonly done in any case.

The step size can not be changed arbitrarily. It should not be allowed to get too small. It also can not be increased at any time. The SOLVE processor requires that the solution values be obtained at intervals of the original step size, which is the

increment of the FOR range. Hence, the step size can not be increased when that action would cause one of the required solution values to be skipped. Again, this is easier to control if the step size is only halved or doubled.

The language was designed with the common characteristics and the differences of these algorithms in mind. The user can declare names for the independent and dependent variables, the derivative function, the step size and step number, and the requested precision. Hence, he can write the procedure in terms that are natural to the specification of these algorithms. Only the step from Z_n to Z_{n+1}, or possibly further, need be specified; the system will automatically repeat the procedure. Array arithmetic with component-by-component operators was implemented so that the formulas for the basic methods could be written normally. Segments were introduced to accommodate procedures with multi-step and single-step parts; the system analyzes the segments to determine the values of Z and Z' that need to be saved and how many values must be computed before the multi-step part can be used.

The APPLY statement has two important functions. First, it permits a hierarchical structure in a procedure. Subprocedures can be used to isolate parts of the methods and to share them among procedures. For instance, the basic formulas that define a Runge-Kutta scheme can be put in a subprocedure and then used

by several other procedures. Second, it allows a subprocedure to be called with varying conditions. The step size and step number can be changed so that a procedure written to compute Z_{n+1} from Z_n can be used to compute Z_{n+2} from Z_n or from Z_{n+1}, for example. This is required to estimate the error in some methods.

Other facilities were also included to compute error estimates and change the step size. The MAX operator with one argument can obtain the maximum error in a vector. An assignment to the step size halts the procedure and causes it to restart with the single-step segment. The system does all the checking needed to insure that the solution is obtained at the required values of the independent variable, since these checks are common to all these algorithms. The IF... THEN... ELSE and DO... END constructions permit complex decisions in the segments; these were needed to support methods that restart by interpolation, such as STANINT.

The UNTIL clause on a DO statement was included to permit iteration until a convergence criterion was satisfied. The USE and APPLY statements allow a fixed step size so that the effect of executing a procedure without adjusting the step size for error control can be easily studied.

The Relmath procedural language shows the level of support that can be attained for a class of numerical methods. While this class is limited, the language provides a goal for other

languages. We would like to develop similarly high-level languages with broader scopes or for other types of problems, such as partial differential equations.

Declarations and Names

As a convenience to the user, it is desirable not to require declarations of the names. At the same time, it is necessary to be able to determine the attributes of a name, and in some cases declarations may be needed to do this. Thus, some balance is needed here.

In some languages, a lack of declarations can require considerably more processing time. For instance, consider the expression "A+B". In NAPSS, A and B may each be either a scalar or an array, real or complex, with long or short precision. The NAPSS interpretive routine for addition must distinguish all these cases and perform the proper operation. In general, if a language does not have declarations and an operator can apply to operands with many different attributes, then the exact operation to be performed can not be determined at the time an expression is parsed. In some cases, a global analysis of the program can determine the attributes that the operands will have, and the exact operation can then be determined. But in other cases this is not possible, and the operation can be determined only by examining the operands' attributes when the operation is executed. Usually, it is easier to implement a language by always deferring this

determination until execution, since then a global analysis is not needed.

In Relmath this problem arises only in differential equations, where a number may be either a single value or a coefficient-value pair. The arithmetic operators must determine whether a coefficient is present and act accordingly. This single test is easy and less complicated than the tests needed in a language like NAPSS with a variety of types of numbers. Furthermore, the non-procedural part of the language is used by less experienced people and should be convenient. Not requiring declarations, except for arrays and array functions, simplifies this part of the language. Arrays must be declared so that their dimensions will be known.

In the procedural part of Relmath, the names for the independent and dependent variables, the derivative function, the step size and step number, and the precision must be declared. This allows the system to associate these names with the corresponding entities. The declarations of the step size and precision are necessary to distinguish them from ordinary scalars; the derivative function must be distinguished from other functions. The use of the other names might be determinable from a global analysis of the procedure, but that did not seem worthwhile. These six names are natural ones to declare, they would normally be described in a textbook or in comments in a program, and they can be used in several procedures without redeclaration.

Arrays with a dimension of "<*>" are used in procedures as temporary vectors. These are declared so that they are distinguishable from scalars. At some future time the language processor might be modified so that this distinction is made by a global analysis, but not now. Again, these names can be used in several procedures.

We will briefly describe the method of handling undeclared names in Relmath since similar mechanisms could be used in other languages. The names are found in a prescan of the sentence before parsing it. Initially, they are given all possible parts of speech, but those not compatible with their contexts will drop out of the parse. Semantically, all data types have a common header. This header contains flag bits to tell what type the name actually is. There is also a value field, whose contents depend on the actual data type. For example, it contains the value of a scalar or a pointer to an array.

Notation for Operators

The notation for the individual operators of the language should be as close as possible to normal practice. This will make the language easier to use. In designing this notation, we must take note of some idiosyncracies in mathematical usage.

For example, in Relmath the expression "1<X<2" is true if X is between 1 and 2. In PL/I, however, the expression is always

true. PL/I was designed as a general programming language. Its syntax for expressions gives a consistent means of manipulating a variety of data types, but it does not give sufficient attention to the normal meanings of its symbols. Relmath deals with a smaller number of data types in a more natural manner. Other examples are the vertical bar for absolute value and the MOD operator for modulus.

The notation for the operators like SUM with a FOR clause is not consistent. For instance, we can write

SUM FOR X=1, . . . , 10 (X*X)

but we also write

FOR ANY X=1, . . . , 10 (F(X)<1)

instead of ANY FOR X=1, . . . , 10 (F(X)<1). In other words, the SUM, ANY, FIRST, and similar operators do not all fit one form with only the operator named changed. This was done because the small changes in the forms made the different operators read better.

The temporary function notation in Relmath, however, was a compromise between several considerations. To discuss it, we will need an example. Consider a function I defined by

I(F) = 1 IF FOR ALL X=0, .1, . . . , 1 (F(X)>0), ELSE 0

I expects a unary function as its argument. We can use I as in

G(T) = T*T

PRINT I(G)

but we would also like to be able to write

PRINT I(T*T)

In this case it is clear that "T*T" should be interpreted as a function of T.

In several mathematical notations expressions are interpreted as functions by taking one or more of their variables as parameters of the function. In some cases, the parameter variables are identified in the notation; in others, they are identified by a convention on the use of names. For example, in "$\int at^2 dt$," the "dt" indicates that "at^2" is a function of t, not a. It has the same use in "$\frac{d}{dt}(bt^2+\sin(t))$." In writing integral transforms, there is often a convention that some letter, such as s, is the parameter variable in the expression to which the transform is applied.

In Relmath the parameter variables are always shown explicitly. In the example above, we would write

PRINT I(FUNCTION(T): T*T)

In this case, the phrase "FUNCTION(T): " seems unnecessary, but that is only because T is the only name in the argument expression. If other names were present, we would either have to designate T as the parameter name or have some convention as to which names could be parameters.

The method of allowing the user to establish a convention was rejected because it would be cumbersome if temporary functions

were nested, or if a name might sometimes be a parameter variable and sometimes not. In some cases, the user might not have a clear convention to follow. The present notation is unambiguous, although a bit lengthy. It is also practically identical to that used in the lambda calculus and semantically the same as the "dt" notation in integrals and derivatives.

In ALGOL 60 another method was used, the "call by name" [Naur63, 12]. There we could define I by

```
real procedure I(f, t); real f, t;
    begin I=1;
    for t: = 0 step . 1 until 1 do
        if f ≤ 0 then I : = 0
    end I
```

This may be called by I(sin(x), x). The argument x replaces t, and sin(x) replaces f. Hence, the for clause causes the value of f to change as t is stepped. This hidden dependence of one parameter on another is not desirable in a language for relatively unsophisticated users, although it is a powerful device for the sophisticated programmer.

Input and Output Formats

Relmath receives its input as a linear character string. This in itself imposes some modifications of normal mathematical notation, which is two-dimensional. Thus, exponents and subscripts must be written as part of the line with special operators to distinguish them.

Some work has been done by others to accept hand-drawn two-dimensional input [Bern69; Black 69; Mar71a; Will72]. However, this work has concentrated on the problems of recognizing the input and providing similar, two-dimensional output. These languages do not have powerful statements to solve standard problems. It would be desirable to combine this type of input with a more powerful language. If suitable hardware were available, we would attempt this with Relmath.

In designing the output statements for Relmath, we wanted simple statements that people who were not experienced programmers would find easy to use. Thus, we provided an unformatted print statement and a restricted type of formatted statement. The format specification is by means of a picture of the output line rather than a series of format items describing each field. This type of specification is more restrictive than FORTRAN's, but also easier. It is based on that used in JOSS [Bry67].

The plot statements are also simple. Plots can be obtained with no specifications from the user, or simple statements can be used to specify the titles and ranges of the axes. The amount of control given to the user is much less than that available in a normal FORTRAN plotting package. However, we wanted a simple system with sufficient control for most users rather than a complicated system that would force the user to make many choices.

Generally, for numerical engineering languages of the type we are considering it is better to have simple output statements with a minimum of choices. These are sufficient for most purposes, and they keep the user from making elaborate output formats.

Remaining Problems

Based on our experience with Relmath, we can identify four remaining problem areas for languages for differential equations. Indeed, these comments apply to languages for other types of numerical engineering languages as well, although there may be some additional problems in some specific fields. The problem areas are:

1. The integration of two-dimensional, hand-drawn input with powerful languages for solving standard problems.
2. Greater use of implicit symbolic manipulation to derive expressions that are needed in numerical problems, as in the computation of a Jacobian to solve nonlinear systems or stiff differential equations.
3. The difficulties of processing array subscripts in a system of equations.
4. The development of procedural languages that are high-level, as Relmath is, but with a broader scope or for a different type of problem.

Another type of change for Relmath would be to extend it to solve other kinds of standard problems, such as linear systems or nonlinear equations. Linguistically, this would not present great problems since a common mathematical notation can be used. Semantically, it may be necessary to modify the basic representation of arrays and tabular functions. Some such modifications are discussed in the chapter on paging. However, only the routines that deal with these basic representations need be changed; the rest of the Relmath programs would still work. These extensions could be made, but they are not necessary. The language is useful as it now is.

IV. NUMERICAL METHODS

In this chapter we will discuss very briefly some considerations involved in selecting numerical methods to solve the standard problems supported by a numerical engineering language. Most of these points have been discussed further by Rice [Rice68], and this chapter is largely based on his statements.

It is of primary importance that the methods be reliable. Of course, they should be properly implemented, without programming errors. But they should also be capable of solving the problem, even in numerically complicated cases. In many cases this requires augmenting classical methods of numerical analysis with logic to handle the complications. For example, several numerical techniques exist to converge to a root of a nonlinear equation given an initial guess. However, a reliable program to solve a nonlinear equation should also have methods of finding good initial guesses, detecting discontinuities and asymptotes, handling multiple roots, and other numerical difficulties. These problems are compounded when a system of nonlinear equations is to be solved.

Numerical methods should also be efficient. To some extent, this conflicts with the criterion of reliability. A reliable and broadly applicable program may have to test for many special cases. These tests could be eliminated if the program were

restricted to simpler problems, and hence it would be faster. However, for numerically unsophisticated users reliability is a more important goal. Such users may not realize that a problem presents special difficulties, or they may not know how to overcome them.

It would be desirable for the program to be able to quickly recognize simple cases so that simple and fast methods can be applied to them. However, this is not always easy. For instance, a function may appear to be smooth at a large and small step size, but not at an intermediate one. For differential equations, the program should distinguish stiff systems from others. To help the program the user should be able to give options specifying the type of equations involved, the range in which roots should be found, the number of solutions wanted, and other information which would be helpful to the program. However, this information should not be required.

A third consideration is the difficulty of implementing a method. When special cases exist, they can frequently be refined almost indefinitely. Some refinements will yield useful special cases, but at some point the additional effort required to implement further special cases will not be sufficiently rewarding. Other cases may simply be too difficult to process at all without greatly enlarging the scope of the system. Relmath should process stiff equations, but the effort required to do so did not seem

justified in a prototype system. Still, for non-stiff systems the standard procedure of Relmath is superior to the MODDEQ subroutine on Caltech's FORTRAN library [Mat70] and the HPCG subroutine in IBM's Scientific Subroutine Package [IBM 68, 337-343], both also restricted to non-stiff systems. MODDEQ does not control the truncation error while using a Runge-Kutta scheme to start the computation, and so it does not solve systems such as the ORBIT example of Figure 2-6 which require a small step size at the start. HPCG contains a programming error, and it does not return the solution values at the proper values of the independent variable. Its numerical method is also inferior, since it did not solve the ORBIT problem as accurately as Relmath did.

New numerical methods are needed in some fields which are poorly understood now. Partial differential equations are especially difficult to treat with any great generality. Even in better-known fields, we need methods that can decide which traditional numerical technique is best for a particular problem. This requires blending numerical analysis with a limited form of artificial intelligence.

V. PAGING

Many computer systems today use paging. Some, like REL, have software paging. Others utilize machines that have hardware virtual memory. In either case the paged data are referenced with virtual memory addresses. Some mechanism must recognize whether a virtual address is in core, load its page if not, and translate the address to a real address. In a software system this is done by calling some subroutine. In a hardware system, the entire program and data are in virtual memory. The hardware translates every address to a real address and invokes a subroutine in the operating system if the page must be loaded.

A hardware system has the advantage that the translation from a virtual to a real address is fast when the page is in core. In a software system this translation is done by a subroutine, which takes considerably more time. Consequently, this translation should not be done more than necessary. A program must also keep track of both real and virtual addresses and their correspondence. These considerations complicate programs using software paging. Finally, in some cases it is desirable to have contiguous virtual pages addressable as a contiguous area in core. This is easy in a hardware system, but in a software system it is difficult since other pages may have to be moved.

Hardware systems can run any program without need of modification, whereas programs for a software paging system must be especially written for it. In some ways this is an advantage, but it does not mean that any program should be run on a hardware paging system without modification. We will see that programs written without regard to the paging environment can be very inefficient. Since hardware systems impose paging on all programs the language designer should write his programs to take account of the paging.

These considerations will become increasingly important as more machines with virtual memory hardware are installed. Such machines are especially good for interactive time-shared systems. Numerical engineering languages are best when implemented in an interactive environment, and so attention to the paging properties of the language processor's routines is important and will become more so.

The use of paging in Relmath

Relmath uses REL's virtual memory to store all its permanent data. Scalars are stored simply as numbers. Procedures and functions defined by formulas are stored by copying their list structure representations to pages using a REL utility. When needed, they are copied back to the list area by another utility.

An array is stored on a page. The elements are ordered by varying the last subscript most rapidly, so the rows of a matrix are contiguous. The array's dimensions are stored on the same page, if possible, or on another. Restricting an array to a single page avoids the problems of storing larger arrays, which will be discussed later. This restriction is not too serious in a prototype intended mainly for use with differential equations. However, it should be removed in an operational system, especially if array operators are added.

Functions defined by a table and array functions are similarly structured. Both have a header followed by the values in the range of the function. For an array function, these values are arrays; each array must be on a single page. The header includes the initial value of the domain variable, its increment, and the number of the final value, counting from zero. It also gives the order of the highest derivative defined in the table. For each value of the domain variable, the derivatives are stored contiguously from order zero to the highest. The pages on which the values are stored need not be contiguous in the virtual memory. Each page is linked to the previous and next ones. The header points to the last page and also to the last page referenced. These pointers include the domain value and derivative order of the first value on the pages. Consequently, a search for a particular function value can proceed forward or backward from the beginning or end of the page list or from the last page referenced.

Tabular functions can only be defined now by solving a differential system. The temporary data needed by the solve procedure are stored in the unstructured portion of REL's memory, which is not paged. The function values are copied to their pages as they are computed. Hence, the pages are accessed sequentially. When a function is printed or plotted, its values are usually accessed sequentially also.

Two changes could be made in this scheme. One would be to allocate the pages for a function contiguously. Then the exact virtual address of any function value could be computed from the header information. No page links or pointers would be needed, and any function value could be found in two page accesses. This method could be implemented, but it would require a manager of the unused pages, essentially a storage allocator for the virtual memory.

Another change would be to group all the functions in a differential system into a family. When a system is solved, the values of all functions in the family would be stored together for each domain value. The function header would point to the family where the function is currently defined and give its offset in the family. This modification may be better while the system was being solved. Let F be the number of functions in the system, and assume they all have the same maximal order and storage size for each value. Then they each take N pages with n domain

values per page, a total of M=nN domain values. If F pages will fit in core at once, there is no problem. However, if this is not true, then one page from each function must be loaded from the disk for each domain value. This is because the function values are moved to the functions in the same order each time, and the later functions will displace the earlier ones. Since each page is written to the disk before being replaced, 2FnN disk operations are required. The new scheme requires only about 2FN operations, since each page will be needed only once. (The 2 is needed since the page will be read in for initialization and later written back.) Since n is now about 60 for a function with one derivative, this is quite a difference.

Unfortunately, this modification is worse than the current method when a function is printed or plotted. The current method requires N page loads to display one function, while the modified method would require FN. However, if several functions, say s, in one family are displayed in one statement, the modified method would still need only FN page loads. The current method will need sN, and would need snN if $s > K$, where K is the number of pages that can be in core at once. For the batch system, K could be quite large, say about 100. Hence, this modification would not be desirable. For most systems there would be no improvement while solving the system and a degradation while

displaying a function. But if K were smaller, say about 20, the new method would be better in many cases.

The storage representation for functions could be extended to allow functions whose domains were not evenly spaced. A flag in the header would designate such a function, and the domain value would be stored with each function value. A search would be needed to find a function value. This change would only require changing the subroutines that set and retrieve function values. Of course, some means of defining such functions would have to be added to the language.

Need to control paging

The REL system has a large unstructured area that is used for temporary storage by procedures. It also has a list area. Neither of these is paged. A software paging system can be set up by its implementer so that not all of memory is paged, and frequently used data and list structures can be kept in unpaged memory. A hardware system pages everything. Unless some means is provided to lock pages in core, the number of page faults could be very high, particularly while processing list structures.

This raises an important point. In any paging environment, the programmer should design his algorithms to take account of the paging. The system should allow his programs to find out the number of pages available and to control which page is replaced when that is necessary to load another page. Unfortunately,

manufacturers of hardware paging machines frequently ignore these needs. They claim that programs can be run without special algorithms, and their operating systems often do not provide the information and control needed to optimally control the paging. These claims are true, but programs may be very inefficient without special algorithms.

These considerations are well illustrated in the processing of matrices. McKellar and Coffman [McKel69] have analyzed conventional algorithms for multiplying and inverting matrices and also algorithms written for a paging environment. They have determined the number of page faults for different algorithms, that is, the number of times that a page that is not in core is referenced. Each page fault requires a disk read to load the page and possibly a write to move a replaced page to the disk. Since disk operations are several thousand times longer than core access times, a large number of page faults can greatly increase the elapsed time of a program. It also increases the load on the disk. Both factors cause greater interference with other programs in a multiprogramming system.

To examine this effect, we consider one of McKellar and Coffman's algorithms for matrix multiplication. It stores successive rows on a page, with each on just one page. This row storage scheme requires $P = \lceil n / \lfloor p/n \rfloor \rceil$ pages, where the matrix is

n x n and each page contains p numbers. ($\lfloor x \rfloor$ is the greatest integer $\leq x$; $\lceil x \rceil$ is the smallest integer $\geq x$.) Let $m=\lfloor p/n \rfloor$, the number of rows per page. Then the algorithm to form C=AB is

```
begin
    for I: = 1 step 1 until P do
        for i : = m x (I-1)+1 step 1 until m x I do
        for j : = 1 step 1 until n do C[ i, j] : = 0;
        for K : = 1 step 1 until P do
        for J : = 1 step 1 until P do
            for i : = mx(I-1)+1 step 1 until mxI do
            for j : = mx(J-1)+1 step 1 until mxJ do
            for k : = mx(K-1)+1 step 1 until mxK do
                C[ i, j] : = C[ i, j] + A[ i, k] x B[ k, j]
end
```

In this algorithm one page of C is loaded and zeroed, and the corresponding page of A is loaded. The pages of B are loaded. Each is processed with the page of A to form the matrix product in the page of C. When this is done for all pages of B, the next pages of A and C are loaded, and the process repeated.

This algorithm requires $N_3 = P(P+2)$ page faults if K, the number of available page frames, is 3. This assumes that the page replacement algorithm is optimal; in particular, when a new page of B is loaded, it should replace the previous page of B.

When K>3, the algorithm can be improved. Let $K=2I+1<P+2$. Then the improved algorithm would partition A and C into blocks of I pages each. A block of A and of C is loaded,

and then each page of B is sequentially processed with the A block to generate the C block. This modification requires $N_K = P\lceil P/I \rceil + 2P$ page faults if each page of B replaces the previous page of B and not one of A or C.

A conventional algorithm would multiply a row of A by a column of B for each element of C. Hence, if K=3, every page of B is loaded for each element of C. Thus, there are $Q_3 = (n^2+2)P$ page faults. If $3 \leq K < P+2$, we can lock K-3 pages of B in core. This gives $Q_K = n^2(P-K+3)+K-3+2P$ faults. If K=P+2, then all pages of B can be locked, and $Q_{P+2} = 3P$, the minimum possible.

The improvement from using the modified algorithm can be quite spectacular. If K=3, n=64, and p=1024, then P=4 and $Q_3 = 16,392$, whereas $N_3 = 24!$ Since 12 faults are required just to reference each page of each matrix once, this shows that careful attention to the paging environment can result in great economies.

Any paging scheme must have a method of replacing some page with a needed one when no more empty page slots are available in core. McKellar and Coffman assumed that this page replacement algorithm is optimal. It should replace each page of B with the next one. When the algorithm is not optimal, the number of page faults is increased.

For example, a common replacement algorithm, called a LRU ("least recently used") algorithm, retains the K-1 pages

that have been most recently used and replaces the oldest page. Let K=2I+1. Denote the pages of A by $A_1, \ldots, A_P$, and similarly for B and C. After the first I pages of A and C have been processed with B_1, the list of pages ordered from oldest to most recently used is A_1, C_1, A_2, C_2, ..., A_{I-1}, C_{I-1}, A_I, B_1, C_I. The process then attempts to repeat with B_2 replacing B_1. But the LRU algorithm will replace C_1 by B_2 instead. (We assume that A[i, k] is referenced first in forming A[i, k]B[k, j].) C_1 will then be reloaded, replacing A_2. The program will continue, replacing each A and C page just before it is needed until A_I replaces B_1. C_I will not be reloaded. Hence, 2(I-1)(P-1)+2I+P page faults occur for each block of A and C pages, compared with 2I+P faults when each B_i replaces B_{i-1}. The total number of faults is $L_K=N_K+2(P-1)(P-\lceil P/I \rceil)=N_3$ $+(P-2)(P-\lceil P/I \rceil)$. Hence, $L_K > N_K$ if P>1 and I>1. Furthermore, $L_K>N_3$ if P>2 and I>1, and L_K increases as K (and hence I) increases!

This example shows that if the page replacement algorithm is not optimal, an attempt to improve the multiplication algorithm by better utilizing the available page slots may actually increase the number of faults. Consequently, the program should be able to control the replacement of pages. However, if such control is not available, the programmer can change the algorithm. If the same method is used with K=2I+2, instead of K=2I+1,

then the extra page slot will receive B_2 when it is first loaded. The A and C pages will not be replaced. B_3 will replace B_1, B_4 will replace B_2, and so on. Hence, the number of page faults will equal N_{K-1}. However, an optimal algorithm could use the extra slot by locking B_1 in core. This reduces the number of page faults by $\lceil P/I \rceil - 1$.

G. Ingargiola has pointed out that even with an LRU scheme the programmer can assure that the page replacement is optimal. He can do this by referencing each page that he wants to keep in core just before referencing some element on a new page. These extra references to the pages in core are simple, just reading and storing back a number, and they order the pages so that page that should be replaced will be the oldest. The LRU scheme will then replace it. For example, in the multiplication algorithm the program should reference elements on A_1, C_1, A_2, $C_2, \ldots, A_I$, and C_I just before referencing B_2. B_2 will then replace B_1, as it should. Similar action is taken before each reference to a new page of B.

By this method the program can be modified to effectively give it control of the paging even though the system does not explicitly do so. The method relies on the programmer's exact knowledge of the system's page replacement algorithm.

We have assumed that the program can determine K. If not, then it must assume K=3, which may result in a larger

number of faults than is necessary. The determination of K must be accurate; we have just seen that, in a LRU scheme, if a program assumes K=2 I+2 when actually K=2I+1, then the number of faults is considerably increased. Hence, it is not sufficient to have high priority slots whose pages will probably be in core but may be replaced.

We have seen that the number of page faults generated by a program can be much higher than necessary if the program is not written with regard to the paging environment. Hence, the language implementer should write his standard algorithms with attention to the paging. The system should give him the capability to control the paging, and he should exercise this capability. But even if the system does not do this, the implementer can still make significant improvements by taking account of the characteristics of the system's paging method, although the number of faults may be higher than is strictly necessary.

We emphasize that this discussion applies to the language implementer. The engineer or scientist who uses the language should not be able to control the paging directly. Relmath, for instance, does not allow the user to directly refer to pages as entities at all. Languages with procedural languages more like FORTRAN would allow the user to access array elements. He may then want to write his procedures so as to take account of the page boundaries but he should not be able to control the

paging. This is to protect the system and the user himself, who is not expected to be a skilled systems programmer, from errors and deadlocks. In general, we would not expect the user to even consider the page boundaries in writing procedures, even though this will cause inefficiencies.

VI. COMPILING

Any language processor does two things with an input sentence: it parses the sentence and executes it. The parse extracts all the information conveyed by the syntax alone. It tells what semantic transformations are to be applied and how their inputs and outputs are connected. During the execution phase, these transformations must be carried out.

One method of doing this is to code the parse into some structure which is then interpreted. For example, in REL the parse tree itself is interpreted. In other systems, a linear string is produced from the parse and is then interpreted. In any case, the coded structure must tell what transformations are to be done and what their inputs are. For each transformation there is a interpreter routine that performs it. This routine must handle all the operand types that all allowed by the syntax for this transformation.

Another method is to compile machine code and then execute it. Generally, this code will be executed faster than the corresponding interpretive code, but it also will take longer to produce the compiled code from the parse. However, the actual times depend strongly on the complexity of the complier and the characteristics of the language.

As an example, consider a language consisting only of arithmetic expressions and statements that assign values to

scalars. All numbers are of one type, say single-precision floating point. It is easy to compile code for this language. The temporary storage for the parts of an expression could be put in a stack whose current top is addressed by a register. All operands and results are in the stack. The machine code for any transformation is then a fixed bit string. This code could be produced about as quickly as the code for a post-fix Polish interpreter. However, the machine code would execute only slightly faster than the interpreter, and it would take considerably more space. If the code string were being stored for execution later, it would take much longer to move the compiled string than an interpretive string.

A more sophisticated compiler could produce better code which would be significantly faster and shorter. Temporary results could be kept in registers when possible and moved to fixed storage locations when not. Operations could be re-ordered or changed to produce equivalent but faster code. On a global basis, common subexpressions could be recognized and computed only once. If the language were augmented to include loops, registers could be assigned to the most frequently used variables and results, and constant expressions could be moved out of loops. If arrays were included, subscript calculations could be optimized in loops. Of course, these optimizations are done in

some commercial compilers for general programming languages, such as IBM's FORTRAN IV H-level compiler [IBM70].

Two types of optimization are done. First, unnecessary calculations may be eliminated, as when constant expressions are moved out of loops. Second, the mechanics of passing control to the transformations and passing intermediate results along are optimized. The simple transformations, such as addition, are not called but are built into the code. The results are passed in registers or fixed storage locations. In the case of array subscripts within loops, the entire process of calculating the subscript and then the element's address may be collapsed into a single instruction to bump a pointer to the element.

A complex optimizing compiler can produce a program that will take much less time to execute than a program produced by a simple compiler. The program will also be shorter. However, the complex compiler will take longer to produce its program than the simple compiler would. Hence, full optimization should be confined to calculations that will be done many times. For example, a Relmath procedure will be executed ten to a hundred times or more while solving a system of differential equations, and each time each assignment statement is executed once for each component in the dependent vector. Consequently, full optimization is justified for these procedures.

The amount of optimization that can be obtained with a given level of analysis depends on the language. In NAPSS, variable names can be scalars or arrays, real or complex, and of long or short precision. Names with any combination of attributes can be combined by the arithmetic operators. Consequently, the parse of an arithmetic expression cannot determine the attributes of the names from the syntax alone. Declarations could resolve the attributes, but declarations are not required. Hence, the semantic transformations for these operators must determine the attributes of their operands dynamically. Since these transformations are complex, a simple compiler would generate calls to fixed subroutines.

For the language considered earlier with one data type, the code could be improved by local optimization. However, for NAPSS very little improvement can result from such local analysis since most of the execution time will still be spent in the subroutines. A global flow analysis is needed to determine, as much as possible, the data attributes that a name will have at each point in a procedure. This information can then be used to eliminate the subroutine calls when the operands are simple, such as two real numbers.

Thus, for a language with complex semantic transformations, local optimization is not very effective. Global analysis may be very effective and may permit more local optimizing to be done, but it is also more expensive in the compile stage.

Compilation, particularly if sophisticated optimization is used, can greatly improve the speed of a calculation. However, sophisticated compilers are very difficult to produce. This production expense prohibits their development for many special languages.

In Relmath we can avoid this development expense. Since the procedures use only one type of number and arrays are declared, it is not necessary to do any global analysis to determine the attributes of the names. We can translate the statements of the Relmath procedure to FORTRAN and then have this FORTRAN program compiled. Normal FORTRAN arithmetic operators can be used, so the compiler can optimize the calculations. In this way we can obtain the advantages of a sophisticated compiler without incurring a great development cost.

Briefly, the semantic routines in Relmath include code generators as well as the interpretive routines. During compilation, the code generators are executed instead of the interpretive routines. Each routine returns the name of the FORTRAN variable that holds its value as its output. Hence, another routine can generate code that uses the output of this routine. This method works well and produces code that executes as fast as a hand-coded FORTRAN program. The following table shows the times in seconds to solve the three examples given in Chapter 2

using the hand-coded standard program and the compiled version of the STAN procedure.

Table 6-1-- Comparison of hand-coded and compiled procedures

	Standard	STAN
CR (Chemical Reaction)	1.4	1.3
GA (Gas Absorber)	10.6	10.8
ORBIT	21.0	21.2

The compiled procedures can also be run in a normal FORTRAN environment. Comparisons on the same differential systems between STAN and subroutines in the Caltech library and in IBM's Scientific Subroutine Package show that STAN runs as fast as the others on the CR and GA systems. The library routines were not able to solve the ORBIT system.

We have seen that compilation is desirable for calculations that will be done repeatedly, and that more optimization should be done for calculations that will be performed very frequently. We have also seen that languages with complex semantic transformations gain little from compiling these transformations unless a global analysis is done. However, optimizers that do such global analysis are expensive to write. Relmath solves these problems by having only simple transformations to be compiled and by using the FORTRAN optimizing compiler to do the actual compiling to

machine code. The Relmath language allows this translation to FORTRAN to be done without extensive analysis of the procedures, and hence it permits us to use a sophisticated optimizing compiler that has already been developed.

This example shows the value of designing the language to make effective compiling easier. But we also need to develop better compiling techniques so that global analysis and optimization will be easier to do.

VII. SUPPORT FOR LANGUAGE IMPLEMENTATION

Relmath successfully meets the goals of a numerical engineering language within its application area. It is a natural and high-level language for stating ordinary differential equations to be solved and also for stating certain algorithms to compute the solutions. It is effective in reducing the effort needed to solve such equations and thereby helps remove a major block to the greater utilization of computers by scientists and engineers.

However, a great amount of effort was needed to achieve this result. Other numerical engineering languages have also required considerable effort to implement. Reviewing this experience with Relmath has led to the recommendation of a program to develop a more supportive environment for implementing such languages. This environment would significantly reduce the effort and expense of producing these languages. It includes a broad procedural language which can be specialized and extended to specific classes of algorithms. This chapter presents this program.

One of the goals of Thompson and his colleagues in producing REL has been to provide support for a wide class of applications languages, including some that are not numerical as well as some that are. The facilities of REL - a parser, a paging system, a list manager, and an interpreter - can be adapted to a specific language by putting in the proper syntax and semantic routines. These facilities help in the development of languages.

However, there is one major difficulty in using REL currently. The semantic routines must be written in the assembler language, since no current higher-level programming language provides appropriate interfaces to REL's subroutines and data structures. FORTRAN programs can be used to do purely numerical calculations, but these are a small part of a language. Most of Relmath's code does symbolic manipulations and processing of data structures that are not efficiently represented in FORTRAN.

Peter Szolovits is currently designing and implementing a language writer's language for REL. This language will be used to write other REL languages. It allows the implementer to write the syntax of his language. He can also write the code for the semantic routines in a programming language roughly similar to PL/I. However, this language includes data types such as list elements and pages that are peculiar to REL. It also generates code with REL's calling conventions. Thus, it has implicit knowledge of the REL environment and specifically supports writing REL languages.

REL currently is available to only a few people. We need a system that is generally available with the same function as REL - to provide support to implementers of applications languages. This system would include a programming language that specifically supports writing the application language routines to run in the system.

Using this system, we can then implement a basic numerical language. This language will not support any specific applications or have any operators to solve standard problems. It will allow wri ting subroutines in a procedural language similar to FORTRAN or PL/I. It will have the usual data types, such as scalars, functions, and arrays, and also array functions. Tabular functions will be manipulable as functions rather than as arrays, and it will be possible to set a function's value at a point. Iteration is a common technique that needs more support than that provided by a DO statement; for instance, there should be a notation for the previous and next values of a variable.

Ths most important difference from current languages is that equations will be directly representable and will be rewritable in some standard forms, just as Relmath rewrites differential equations in the form $z'=f(t, z)$. A numerical algorithm for solving equations generally assumes that they have been stated in some form so that the significant entities are isolated. The algorithm is stated in terms of these significant scalars, functions, or whatever. A procedure to implement this algorithm should be able to have the input equations rewritten in this standard form.

Some parts of the rewriting operator in Relmath ar e needed in other types of problems. For instance, systems of equations often have many names for the unknowns; these must be collected into a single unknown vector to fit the form of numerical algorithms.

It is often necessary to have the equations in a form like z'=f(t , z) which is solved for some variable, or in a form like $a(x, y)\frac{\partial^2\phi}{\partial x^2} + b(x, y)\frac{\partial^2\phi}{\partial y^2} = K$ where the coefficients of certain variables are isolated. These rearrangements are common to many problems. Others are more specialized. In a differential equation, higher-order derivatives are replaced by new variables to reduce the equation to a first-order system. This operation is particular to rewriting differential equations.

Thus, we expect the primitive rewriting operator to have several parts in its implementation. It will be able to collect several names into a vector when processing a system, to rearrange equations in order to match certain forms, and to establish a correspondence between certain names in the forms and expressions in the equations. It may also be able to perform complex calculations itself, such as solving a nonlinear system to match a linear form, but such calculations must be controllable. Finally, it will be able to do some specialized transformations for particular types of equations, such as reducing differential equations to first order.

The operator will be invoked in a procedure by some syntax such as "REWRITE EQ IN FORM Z' = F(T, Z)." Additional clauses could control optional transformations that are not always applied. If the operator could not fit the equations to the form, this

procedure would not be used to solve the equations and another would be tried. Hence, the type of the equations could be used to select an appropriate procedure.

Another important feature of the base language is that it will be extensible. This will permit the construction of good syntax to call the standard numerical procedures and of higher-level procedural languages for various classes of algorithms. We want the power to define a language similar to the Relmath procedural language by this means, although it is not necessary to duplicate Relmath. The translation to the base language would be done by the extension mechanisms and definitions, whereas Relmath has a fixed program to translate its procedures to FORTRAN.

We do not expect the engineer or scientist who would use such a system to make great use of the extension capabilities. These capabilities permit the language to be changed. The user does not have the training or interest to make extensive changes in his language. He is concerned with solving particular problems, not with the construction of linguistic tools. The language implementer, however, is concerned with the language and has the skill to change it to fit the users' requirements. We therefore expect that the extension capabilities will be used primarily by implementers to construct higher-level languages.

Basing these languages on a single language has some other advantages. A single compiler for the base language can support the others as well, and so some effort to develop a good compiler with considerable optimization is justified. The different application languages can interface much more readily than if each one were developed independently, since they will have common data structures for scalars, arrays, functions, and the other entities supported by the base language. Independent languages, on the other hand, are likely to have data structures that are not compatible.

The support given by the basic numerical language and the underlying system and its language development language will also simplify modification of the application languages. Many modifications could be made by the extension facility alone. Others might require adding new routines to the base language. For example, one such modification would be to include chemical compounds and equations in a language like Relmath. Then chemical equations could be converted to differential equations in some cases. We expect such modifications to be made to produce specialized languages for particular users.

In this chapter we have presented an outline of a language development effort that would support specialized applications languages for numerical engineering. Two parts of this effort are the least understood and require further research. One is

the operator to rewrite equations. This should be as broad as possible to accommodate many types of equations and standard forms, and it will be difficult to achieve this breadth. The second is the extension capability. This must also be broad, and it must be possible to compile the resulting programs.

VIII. ASSESSMENT AND RECOMMENDATIONS

With our experience with Relmath and other languages, we can now assess the status of specialized languages for numerical engineering problems relative to the topics discussed in this thesis and present recommendations for their further development. Basically, these languages already greatly enhance their users' ability to solve some standard problems, such as ordinary differential equations. These are some needed improvements, which will be presented soon, but the examples of Relmath and other languages show their usefulness. However, new concepts are required to increase the support given to the implementation of these languages and to increase their capabilities to express procedures naturally.

We will discuss this assessment further, beginning with the areas which do not require extensive new concepts for language design or implementation. In terms of the topics presented earlier, these areas are language design as it relates to standard problems, numerical methods, and paging.

Language features to facilitate solving standard problems for which well-known and effective numerical techniques exist are well-developed. Relmath permits ordinary differential equations to be stated in a natural style using normal mathematical notation and to be solved by simple and direct statements. It shows

that languages designed in accordance with the general principles given in chapter 3 - that computations common to all problems should be made implicit in the language and that the syntax and semantics should be natural - do enhance the users' problem solving capabilities. Other languages are applicable in a similar way to other problem classes.

There are still some linguistic improvements that could be made. These include the processing of hand-drawn two-dimensional input and greater implicit symbolic manipulation when it is needed to derive expressions required by better numerical techniques. These problems are treated by isolated programs now, but these programs have not been integrated with languages for solving numerical problems. A general analysis of subscript expressions in systems of equations requires deductive techniques, which are not currently economical. Acceptance of reasonable limits on these subscripts, improvements in pattern recognition and symbolic manipulation programs, and their integration with powerful languages will yield more useful, though restricted, numerical engineering languages. Removal of all restrictions will await great improvements in pattern recognition and deduction algorithms.

To implement a language that provides a built-in procedure for solving a class of standard problems, we obviously need a numerical method that reliably solves problems in this class. Such methods exist for some problem classes, such as ordinary

differential equations and linear systems, but for other classes new and more reliable methods are needed. These methods may require combining several more restricted techniques with appropriate logic for deciding which would be best to use in particular circumstances.

Our analysis of paging shows that languages run in a paging environment should be properly implemented so that page faults are minimized. Relmath was written with careful consideration to this goal. If the operating system makes relevant parameters of the environment, such as the number of available page frames and the page replacement algorithm, accessible to the language processor, then the paging can be optimized in critical sections of the computations. Even in systems which do not allow access to these parameters, the implementer can still use what information he has about the paging to avoid disastrous operational behavior. This action requires some attention to the detailed characteristics of the computations, but no new concepts are needed.

We now turn to those areas where new concepts are needed: support for language implementation and procedures. We have seen that specialized numerical engineering languages greatly reduce the programming effort needed to solve problems and thereby increase the ability of engineers and scientists to utilize computers effectively. However, little has been done to lessen the programming effort needed to implement these languages. The expense of this

effort has blocked their further development. We need an environment that is widely available to support such implementation efforts. Relmath was written within the REL system, which was designed for this purpose. However, more easily used systems that give more support to numerical languages are needed.

With respect to procedures, Relmath has a procedural language that is very supportive of a restricted class of algorithms. Procedures written in this language can be compiled by a highly effective optimizing compiler. The central problem is to extend these benefits to other types of algorithms.

In the last chapter we presented an approach to both of these problems. A system similar to REL but with a higher-level language for writing routines is to be developed. Then a basic numerical language will be implemented. It will be extensible and will have a primitive operator to rewrite equations into standard forms. The language implementer will use these capabilities to construct more specialized languages for various types of problems and various users. These languages will include highly supportive procedural sublanguages, like Relmath's, but for other classes of algorithms. A single optimizing compiler could compile procedures written in these languages effectively, since they will all be extensions of the same basic numerical language. This common base will also facilitate interfacing the more specialized languages.

This approach will be difficult, since the equation rewriting operator, the extensibility, and the compiler require new computer science concepts. However, the success of this plan would provide a base for the relatively easy development of languages specialized to numerical engineering problems. The availability of more languages of this type will further increase the numerical problem solving capabilities of scientists and engineers.

LIST OF REFERENCES

[Bern69] BERNSTEIN, M. I. and WILLIAMS, T. G. A two-dimensional programming system. Information Processing 68: Proc. of IFIP Congress 1968, Morrell, A. J. H. (Ed.), vol. 1, North-Holland, Amsterdam, 1969, pp. 586-592.

[Black69] BLACKWELL, F. W. and ANDERSON, R. H. An on-line symbolic mathematics system using hand-printed two-dimensional notation. Proc. 24th ACM Natl. Conf., ACM, New York, 1969, pp. 551-557.

[Bry67] BRYAN, G. E. and SMITH, J. W. JOSS language RM-5377-PR, Rand Corp., Santa Monica, Calif. 1967.

[Car70] CARDENAS, A. F. and KARPLUS, W. J. PDEL-a language for partial differential equations. Comm. ACM 13, 3(Mar. 1970), 184-191.

[Dos71] DOSTERT, B. H. REL - an information system for a dynamic environment. REL report no. 3, California Institute of Technology, Pasadena, Calif., 1971.

[Eng69] ENGELMAN, C. MATHLAB 68. Information Processing 68: Proc. of IFIP Congress 1968, Morrell, A. J. H. (Ed.), vol. 1, North-Holland, Amsterdam, 1969, pp. 462-467.

[Eng71] _____. The legacy of MATHLAB 68. Proc. Second Symp. on Symbolic and Algebraic Manipulation, Petrick, S. R. (Ed.), ACM, New York, 1971, pp. 29-41.

[Frie69] FRIEDLANDER, S. K. and SEINFELD, J. H. A dynamic model of photochemical smog. Environmental Science and Technology 3, 11(Nov. 1969), 1175-1181.

[Gear66] GEAR, C. W. Numerical solution of ordinary differential equations at a remote terminal. Proc. 21st ACM Natl. Conf., Thompson, Washington, D. C., 1966, pp. 43-49.

[Gra64] GRAGG, W. B. and STETTER, H. J. Generalized multistep predictor-corrector methods. J. ACM 11, 2(April 1964), 188-209.

[Hull63] HULL, T. E. and CREEMER, A. L. Efficiency of predictor-corrector procedures. J. ACM 10, (1963), 291-301.

[IBM68] INTERNATIONAL BUSINESS MACHINES CORP. System/360 scientific subroutine package (360A-CM-03X) version III: programmer's manual. 4th ed. H20-0205-3, 1968.

[IBM70] _____. IBM System/360 operating system: FORTRAN IV (G and H) programmer's guide. 3rd ed. GC28-6817-2, 1970.

[Koh67] KOHFIELD, J. J. and THOMPSON, G. T. Multistep methods with modified predictors and correctors. J. ACM 14, 1(Jan. 1967), 155-166.

[Lap71] LAPIDUS, L. and SEINFELD, J. H. Numerical Solution of Ordinary Differential Equations. Academic Press, New York, 1971.

[Mar71a] MARTIN, W. A. Computer input/output of mathematical expressions. Proc. Second Symp. on Symbolic and Algebraic Manipulation, Petrick, S. R. (Ed.), ACM, New York, 1971, pp. 78-89.

[Mar71b] _____ and FATEMAN, R. J. The MACSYMA system. Proc. Second Symp. on Symbolic and Algebraic Manipulation, Petrick, S. R. (Ed.), ACM, New York, 1971, pp. 59-75.

[Mat70] MATSUMOTO, K. MODDEQ/Differential equation solver - 360. C1069-314-360, California Institute of Technology Computing Center, Pasadena, Calif., 1970.

[Mor68] MORRIS, S. M. and SCHIESSER, W. E. SALEM - a programming system for the simulation of systems described by partial differential equations. Proc. AFIPS 1968 Fall Joint Computer Conf., Vol. 33, Part I, Thompson, Washington, D. C., 1968, pp. 353-357.

[Naur63] NAUR, P. et al. Revised report on the algorithmic language ALGOL 60. Comm. ACM 6, 1(Jan. 1963), 1-17.

[Rein70] REINFELDS, J., ESKELSON, N., KOPETZ, H., and KRATKY, G. AMTRAN - an interactive computing system. Proc. AFIPS 1970 Spring Joint Computer Conf., Vol. 35, AFIPS Press, Montvale, New Jersey, 1970, pp. 537-541.

[Rice68] RICE, J. R. On the construction of polyalgorithms for automatic numerical analysis. In Interactive Systems for Experimental Applied Mathematics, Klerer, M. and Reinfelds, J. (Eds.), Academic Press, New York, 1968, pp. 301-313.

[Schl67] SCHLESINGER, S. and SASHKIN, L. POSE - a language for posing problems to the computer. Comm. ACM 10, 5(May 1967), 279-285.

[Sein69] SEINFELD, J. H. Mathematical models of air quality control regions. Paper presented at the Symp. on the Development of Air Quality Standards, Santa Barbara, Calif., Oct. 23-24, 1969.

[Seitz68] SEITZ, R. N., WOOD, L. H., and ELY, C. A. AMTRAN: Automatic mathematical translation. In Interactive Systems for Experimental Applied Mathematics, Klerer, M. and Reinfelds, J. (Eds.), Academic Press, New York, 1968, pp. 44-66.

[Stow66] STOWE, A. N., WIESEN, R. A., YNTEMNA, D. B., and FORGIE, J. W. The Lincoln Reckoner: An operation-oriented, on-line facility with distributed control. Proc. AFIPS 1966 Fall Joint Computer Conf., Vol. 29, Spartan Books, Washington, D. C., 1966, pp. 433-444.

[Sym68] SYMES, L. R. and ROMAN, R. V. Structure of a language for a numerical analysis problem solving system. In Interactive Systems for Experimental Applied Mathematics, Klerer, M. and Reinfelds, J. (Eds.), Academic Press, New York, 1968, pp. 67-78.

[Sym69] _____ and _____. Syntactic and semantic description of the numerical analysis programming language (NAPSS). CSD TR II (Revised), Purdue U., Dept. of Computer Science, 1969.

[Thomp69] THOMPSON, F. B., LOCKEMAN, P. C., DOSTERT, B. H., and DEVERILL, R. S. REL: a rapidly extensible language system. Proc. 24th ACM Natl. Conf., ACM, New York, 1969, pp. 399-417.

[Var71] VARAH, J. M. Problem set no. 2, AMa105b, California Institute of Technology, Pasadena, Calif., May 19, 1971.

[Will72] WILLIAMS, T. G. An on-line two-dimensional computation system. SP-3640, System Development Corp., Santa Monica, Calif., 1972.