1969

Manipulation of Data Structure in a Numerical Analysis Problem Solving Systems

Lawrence R. Symes

Report Number:
69-045
MANIPULATION OF DATA STRUCTURES IN A
NUMERICAL ANALYSIS PROBLEM SOLVING SYSTEM
NAPSS

Lawrence R. Symes

CSD TR 45
November 1969
L. R. Symes

-ABSTRACT-

This paper presents the internal construction of an on-line interactive system NAPSS designed for solving numerical analysis problems. The aims and requirements of the system, along with a brief description of its language, are given in the first section. The second section presents the basic components of the system and how they are used by the system. In the final section a description is given of the various data structures that arise in such a system and how they are manipulated.
L. R. Symas

AIMS OF SYSTEM

During the past several years considerable effort has been expended designing and implementing systems which are intended to provide extended capabilities for persons with mathematical problems to solve. Some of them in addition to NAPSS are CULLER FRIED, KLEROER MAT, MAP, RECKONER, AMTRAN, and POSE. These systems can be classified as problem solving systems for applied mathematics.

Before the advent of these systems the research scientist or engineer used a procedural language such as FORTRAN or ALGOL when he employed the computer to aid him in solving a problem. Both of these languages, although they resemble mathematical notation more closely than machine language, are somewhat artificial and contain many unnecessary, from the user's point of view, rules. The artificial appearance and the rules must be mastered before the language can be used. Therefore the scientist or engineer is diverted from his main purpose into becoming a programmer. Even after he has learned the language, its complexity increases the probability of error, and thus reduces his efficiency.

In addition to those difficulties, the user with a mathematical problem had to use program libraries in order to obtain routines for solving commonly occurring problems. These libraries frequently were inadequate and almost always confusing. The routines often were poorly documented and
L. R. Symes

performed little or no monitoring of the accuracy of the results. Thus the user of such a library had to know enough numerical analysis to select the best method for solving problems and to determine the accuracy of the results.

NAPSS has been designed to remove some or all of these problems and to offer several other desirable features. It, in some sense, endeavors to have man do what he is best equipped to do and to have the computer do what it is best equipped to do.

Six general techniques have been utilized to assist the user in stating and solving his problem.

First, the source language used to present a problem to the computer is similar to normal "text book" mathematical notation. This permits one to use the system without first having to intensively study the input language. It also reduced the probability of user programming errors, because the user is familiar with the notation.

Second, clerical statements used for dimensioning arrays and declaring variables are removed from the source language. These are tasks which the computer can easily perform but which are a constant source of errors if the user does them.

Third, NAPSS permits the direct manipulation of quantities other than scalars. These include numeric arrays, symbolic expressions, functions, and arrays of functions. This further allows the source language to resemble more closely
"text book" form, and thereby leads to fewer statements; hence fewer opportunities for programming errors.

Fourth, solve statements are included in the source language. These statements permit the user to state a problem he wishes to solve in a concise, natural form. The user may include parameters such as initial values, the accuracy desired, the method he would like used, or he may omit any or all of the additional parameters. The solve statements invoke routines, polyalgorithms [4][5], from a built-in library. They attempt to solve the user's problem automatically. They request additional information as needed and monitor the accuracy of the results in order to insure that it remains within the specified limits. The inclusion of these solve statements greatly reduces the burden normally imposed on the user. To solve commonly occurring problems with the aid of the solve statements, the user is only required to know how to define the equations for the problem; he is not required to know the numerical analysis involved or even the method used. The method is selected by the system and the accuracy of the results is assured.

Fifth, on-line communication between the system and the user is provided by either a teletype or graphic display device. The use of these terminals bring the computer and the user closer together and consequently improve the user's efficiency.
Sixth, incremental execution of a program is allowed. This, combined with the use of on-line terminals, creates a closed loop between the user and the system. The user is able to monitor his program during execution and the system is able to request information from the user and point out errors when they arise. This eliminates much of the time that is wasted in preparing and submitting runs of a program which are unproductive because the user tried several fruitless cases, has an incorrect program, or has forgotten to initialize a variable.
NAPSS LANGUAGE

Rather than present a detailed description of the NAPSS language [10],[12] we describe a sampling of the allowable assignment statements.

The arithmetic expression in NAPSS permits the direct manipulation of numeric scalars, vectors, arrays, symbolic functions and variables which denote symbolic expressions. The user need not worry about the type or mode of the operands; rather, all that need concern him is whether or not the arithmetic expression is mathematically correct.

Several examples of arithmetic expressions and assignment statements appear below:

i) \[ D - ( B + C ) ! D = E ! \]

ii) ARRAY \( \{ [3,0:2? 1,2,...,9 ] / 10 \)

iii) E = V1 + V2 * 2

iv) \( F(X) = A X + 2 + B X + C, \{ X < 0 \} = A X + 2 - B X + C \)

v) \( G(X) = A X + 2 + B X + C, \{ X < 0 \} = A X + 2 - B X + C \)

vi) \( X(X)[1,1] = X + 2 - B, \{ X < C \} = - ( X + 2 ) - B, \{ X > C \} \)

vii) \( H(X,Y) [5,-2] = G'(X) + X + A \{ X - 0 TO Y \} \)

viii) S = " I AM A WAPPS STRING"

ix) R[1,1] = S || "ARRAY ELEMENT "

The left arrow operator (-) indicates that the arithmetic expression on the right is to be evaluated and its value is to be assigned to the variable on the left. The value assigned to D is either a scalar or an array depending upon the operands.
L. R. Symes

in the expression on the right; while the value assigned to ARRAY is a 3 by 3 array.

The equals sign (=) has the more mathematical meaning. Statement three establishes that a future occurrence of E is equivalent to the expression V1 + V2 + 2. Values are only substituted for the variables in the expression on the right of the = when a value of the variable on the left is needed. Thus if the value of V1 or V2 should change between the definition of E and the use of E this is reflected in the value of E. Variables defined to the left of an = are referred to as equals variables, and variables defined to the left of an ← are called left arrow variables, or simply variables.

Statements four and five illustrate that a symbolic functions may be assigned different definitions on different domains. The difference between statements four and five is similar to the difference between statements two and three. In the definition of F the variables A, B, and C have their current values substituted for them, while in the definition of G they do not. Values are only substituted for A, B, and C when a value of the function G is needed. Functions defined to the left of an = sign are called equals functions and functions defined to the left of an ← are called left arrow functions.

Statements six and seven illustrate how arrays of functions are defined. All the elements in array of functions must have the same number of arguments and they all must be either left
arrow or equals functions.

Statement eight assigns to S a string, and statement
nine assigns a string to an element of an array.

Although NAPSS is intended primarily as a problem state-
ment language, the features of a procedural language have been
included to increase its power for the user who wishes to
create a personal library of NAPSS routines. External and
internal procedures may be written in NAPSS. The use of
these facilities is optional. The casual user need not be
concerned with the rules that procedures introduce, for he can
employ the system on what is called console level.

On console level the user does not set up any procedures.
Statements are entered without having to go through any
initial set up, and are normally executed as they are received.
OVER-ALL STRUCTURE OF THE SYSTEM

The NAPSS system currently running on the Control Data 6500 at Purdue University consists of four main modules: the supervisor, the compiler, the interpreter and the editor. These modules are composed of 115 different routines, which are combined into 28 overlays. Almost all of the system is written in FORTRAN, with the exception of a few machine dependent operations which are restricted to "black-box" modules coded in assembly language. This is done to aid the goal of machine independence for the system.

The supervisor controls the flow into each of the three other modules. It distinguishes between NAPSS sources statements, which are processed by the compiler and edit statements, which are processed by the editor. The supervisor is also responsible for invoking the interpreter when a NAPSS statement is to be executed.

NAPSS source statements are transformed by the compiler into an internal text which the interpreter processes. This scheme was adopted for several reasons. First, the complexity of the elements to be manipulated and the absence of declarations require execution time decoding of operands. Second, it easily allows for extensions to the system. Third, it gives the user incremental execution. Fourth, it permits extensive error diagnostics and permits error corrections without having to recompile the whole program. Fifth,
L. R. Symes

statements which are repeatedly executed are only translated once into internal text.

The internal and source text for each statement is stored in secondary storage. When a statement is to be executed, a copy of the internal text is passed to the interpreter. This reduces considerably the core storage required for a user's programme. Since the system is intended for use in an incrementally executing mode, no reference to secondary storage is normally required to obtain the internal text of a statement.

The system operates in one of two modes: suppress mode or execute mode. In the suppress mode, each statement is compiled into internal text and the internal and source text is saved on secondary storage for later execution. Suppress mode is entered by typing the statement .SUPPRESS. A block of statements which have been compiled in suppress mode may be executed at any time by typing the statement .GO.

The normal mode of execution is execute mode. Here, each statement is executed immediately after it has been compiled and a copy of its internal and source text saved in secondary storage. The system automatically enters suppress mode when the user starts a compound statement (a FOR statement) or a procedure. This is necessary because a compound statement cannot be executed until the whole statement is received and a procedure is only executed when invoked. The system re-enters
execute mode automatically as soon as the compound statement or procedure is completed.

The memory of a NAPSS program is made up of a few pages of real memory which reside in core and a larger number of virtual pages of virtual memory which reside in secondary storage and are brought in and out of real memory. Two vectors (one dealing with virtual and the other with real memory) and several pointers are used to keep track of real and virtual memory.

Each element in the virtual memory vector is subdivided into three twenty-bit bytes. The first byte contains a flag indicating what type of information is stored in the page. The second byte is a switch, used when a page is in real memory to indicate whether or not a copy of the page also resides in secondary storage. The third byte contains the real page number the virtual page is in, when it is in real memory.

The elements of the virtual memory vector which denote available pages are linked together. Initially, the element for virtual page one points to the element for virtual page two and the last element contains a zero. When a page of virtual memory is returned to the system its element is again linked to the top of the list of available virtual pages.

The real memory vector elements contain one entry per real page. This entry is the number of the virtual page occupying it (zero if it is free). This pointer from real memory to
virtual memory is used when a new virtual page is placed in real memory. The virtual page currently in the real page must be copied out into secondary storage if a copy of it is not already there.

The amount of core assigned to real memory is dynamic. Pages are removed from the top and bottom of real memory in order to obtain contiguous blocks of storage. Pages are removed from the top of real memory for two purposes: first, to expand the name table, and second, to obtain space for the work pool. Pages are removed from the bottom of real memory to obtain space for local name control blocks during the evaluation of left arrow functions. See figure 1.

The work pool is used to hold arrays when performing array arithmetic. Requests for work pool space are always made in terms of words. However, the amount of real memory assigned to the work pool is always an integral number of pages. When a request is made for work pool space and the work pool is empty, the space supplied is zeroed. When space is requested for the work pool and the work pool is not empty, one of two situations arises. First, the space requested is less than the current size of the work pool. If the difference between the space requested and the current size of the work pool amounts to one or more pages, a corresponding number of pages is returned to real memory from the bottom of the work pool.

Second, the space requested exceeds the current size of the work
Figure 1
NAPSS Memory Organization
pool. If additional pages are obtained from real memory to satisfy the request, they are zeroed.

Virtual pages are assigned to real pages sequentially. Thus a virtual page is not removed until all real pages are assigned a virtual page. This sequential process may be broken whenever space is assigned to the work pool or to hold the local name control blocks for a left arrow function, since, after the space request is satisfied, the next real page to receive a virtual page may no longer belong to real memory. When this occurs the pointer to the next real page to receive a virtual page is reset to the first page now in real memory.

The algorithm for bringing virtual pages into real memory is further modified when the work pool returns a page to real memory. Since the page returned is empty, a virtual page may be placed in it directly, avoiding the possibility of having to save the virtual page currently there in secondary storage. Thus the normal sequential process is interrupted until all pages returned to real memory by the work pool are re-used.

The system does not assign all of real memory to either the work pool or to space for left arrow function's local name control blocks. A request for real memory space is honored as long as two pages remain in real memory after the request is satisfied. If more space is requested than can be supplied, the request is modified to correspond to the maximum amount of space available. This permits the system to continue if this
L. R. Symes

is adequate.

Two pages are required in real memory to facilitate the linking of virtual pages. With two pages in real memory the above algorithm guarantees that the previous and current virtual pages referenced remain in real memory. Thus they may be linked together if necessary, without having to save pointers and re-read a virtual page to fill in link information.

Associated with each procedure is a name table containing entries for each variable, label and constant in that procedure. The entries, called name control blocks, are created during compilation when the name or constant is introduced. At this time it contains the name of the variable, and some basic attributes describing how the variable appears in the program. During execution the name control block is used to hold values, pointers to values and a complete set of attributes for the variable.

This double usage of the name control block entries poses no problem if compilation and execution are performed separately. But in NAPSS the normal mode of operation is to execute each statement as soon as it is compiled. Thus, three situations are possible when a variable is entered in the name table. First, the variable may never have been used before in the program. Second, the variable may have appeared before in the program but have no value assigned to it. Therefore, it is just as it was when the compiler last saw it.
L. R. Symes

Here a limited compatibility check is made between the two
uses of the variable in the program. For example, the use
of a name as a label and as a variable in an arithmetic
expression is illegal. Third, the variable has appeared before
in the program and has been assigned a value and a complete
set of attributes. This enables more checking to be performed.
However, the name table routine must not disrupt any of the
attribute flags, for if any of them are changed the attribute
may no longer correspond to the value associated with the
name control block.

The name table is constructed sequentially. This method
requires a minimum amount of space, and permits the name table
to grow dynamically. The name table is expanded by removing
pages permanently from real memory. This method of name
table construction does require that the name table be searched
sequentially. The search goes through the name table from
bottom to top. This is done because frequently the greatest
percentage of references to a variable occur in the immediate
vicinity of its definition.

A variable which is declared to be global in \( N \) different
procedures has \( N+1 \) name control blocks associated with it.
There is a name control block for the variable in the name
table of each of the procedures in which it appears. Only
compile time information and a pointer to the \( N+1 \)st copy is
contained in these name control blocks. The \( N+1 \)st copy is in
the global variable name table and contains a complete set of attributes for the variable and its value or a pointer to its value.

The N+1st copy of a global variable's name control block is placed in the global name table when the first procedure is invoked in which the global variable appears, or when the variable is declared global on the console level (the portion of the program not contained in a procedure). When a global variable is added to the global name table and it already appears there, a check is made on the compatibility of the attributes. An error results when they conflict. Otherwise a pointer to the N+1st copy is placed in the procedure's copy of the variable's name control block.

A count is kept in the global name control block of the number of procedures referencing the global variable. When a global variable is no longer referenced, then its name control block is removed from the global name table and the storage associated with it is returned to the system.

A procedure is compiled when it is defined. To permit it to be linked into the program, the text generated uses only relative pointers to name table entries, and all linking between entries in a procedure's name table is done with relative pointers. This allows procedure A, for example, to be compiled as an external procedure and to be invoked either directly from the console level or from another procedure.
which itself is invoked from the console level. The name
table for procedure A is placed in the name table after the last
entry presently there when it is invoked and a base address
is set up.

Variables which are not declared to be either local or
global in an internal procedure are assumed to be known in
the containing block. After the procedure is compiled and a
copy of its name table saved, a pass is made through the
procedure's name table. This pass goes through the name
table from top to bottom and places a copy of the name control
block for each variable not declared to be either local or glob-
al, in the name table of the containing block. If the variable
has appeared in the containing block, a compatibility check
is made between the attributes.

During execution only one name control block is used for
the value and attributes of a variable which is not declared
to be local or global. This is the name control block entry
in the outermost block. The name control block in the internal
procedure is linked to this when the interval procedure is
invoked. The linkage is constructed so that only one step is
required to obtain the value of the variable regardless of the
depth of the procedure.

There are three types of name control blocks in different

---

1 A block is either a procedure or the console level routine.
L. R. Symes

memory areas: ordinary, local for left arrow functions, and temporary. See figure 1. Temporary name control blocks are used to hold temporary results during the evaluation of an arithmetic expression.

A central routine is used to decode variable name control blocks during execution. This routine determines the type of the name control block and handles the linkage between global, and non-local, non-global name control blocks. Three things are returned when a name control block is decoded: the attribute number, the data pointer field and the index of the array AENCBS of first word of the data pointer portion of the name control block. See figures 1 and 2.
DATA STRUCTURES

A name control block is the basic unit of all data structures in the system. In some cases it holds the actual values of the variable, and in others it contains a pointer to the actual values and descriptive information. A name control block is made up of seven sixty-bit words of twenty-one twenty bit bytes. See figure 2.

```
<table>
<thead>
<tr>
<th>ITERATION POINTER</th>
<th>DATA POINTER</th>
<th>ATTRIBUTE PLACES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA PORTION</td>
</tr>
</tbody>
</table>
```

The Layout of a Name Control Block

Figure 2

A name control block which denotes a numeric scalar contains the value of the scalar in its data portion. One or two words of the data portion are used depending upon whether the value is single precision real, double precision real or single precision complex.
When a name control block denotes a numeric array, the data portion of the name control block contains the actual bounds for the array, the declared bounds for the array (these may or may not have been specified by the programmer), and the number of dimensions in the array. The data pointer byte of the name control block points to where the actual array is stored, by rows, as a contiguous block. The array is stored as a contiguous block to speed up array operations.

If the data pointer byte of the name control block is non-zero, a copy of the array exists in secondary storage in the array file. The data pointer is then the number of the record used to store the array and an index in the vector AEVAR.

The vector AEVAR contains additional information about the array. Each word in AEVAR is subdivided into three bytes. The first byte contains the reference count for the array. This is incremented by one each time the array appears in a left arrow function definition. The values of all non-parameter variables are fixed when a left arrow function is defined. The use of a reference count for arrays permits only one copy of the actual array to be kept, and if the non-parameter array variable is assigned a new value the value of the function will not change. The second byte contains the number of dimensions in the array. And the third byte contains the number of words in the array. The number of words is equal
to the number of elements in the array times the number of words in each element. This factor is one for a single precision real array and two for a double precision real array or single precision complex array.

If the data pointer byte of the array's name control block is zero, the only copy of the array exists in the work pool, and the array is the result of the last array operation performed.

The work pool can contain anywhere from zero to three arrays. A counter is kept of the number of arrays in the work pool. In addition, for each array in the work pool the index of the first word of the array, the index of the first word of the data portion of the array's name control block, and the information contained in the array's AEPAR entry is kept. When an array operation is to be performed a check is made to see if any of the arrays involved already exist in the work pool. If they do, no reference to secondary storage needs to be made to obtain the operands. A check is also made to determine if the result of the previous array operation is an operand of the current array operation. If it is not the previous result array must be stored temporarily in secondary storage.

A name control block which denotes an equals variable contains the virtual page number of the first page used to store the internal text for the expression in its data pointer
byte. The first word of each virtual page is used for linkage. The link contains the virtual page number of the next page used to hold the text of the expression or zero if the page is the last. When an equals variable is an operand of an arithmetic expression this internal text is evaluated to obtain a value for the equals variable.

If a name control block denotes a scalar symbolic left arrow function, the data pointer contains the page number of the first virtual page used to store the internal text of the arithmetic expression for the first domain of definition. The first byte of the fourth word of the data portion contains the number of arguments of the function.

The first four words of the first virtual page used to store the internal arithmetic expression text for each domain contains a set of pointers. The first word is used to link together the pages required to store the internal text for the arithmetic expression for the domain. It contains the virtual page number of the next virtual page used. A zero link denotes the last page. The next three words are subdivided into nine bytes. The first byte contains the number of words of internal text in the boolean expression for the domain. This is used when the boolean expression is being moved prior to its evaluation. The second byte contains the reference count for the function. If the function appears in the definition of another left arrow function this is increased by
one so that only one copy of this function needs to be kept. The third byte is the virtual page number of the first page used to hold the text for the boolean expression for the domain. This byte is zero if the domain has no boolean expression. The fourth byte contains the number of virtual pages that are required to hold the local name table for the domain. The local name table contains a name control block for each non-parameter variable appearing in the boolean and arithmetic expression for the domain. This is necessary so that the value of these variables can be fixed when the function is defined. Byte five is unused. Byte six contains the virtual page number of the first page used to hold the local name table for the domain. Byte seven contains the number of words of internal text in the arithmetic expression for the domain. Byte eight is unused and byte nine contains the virtual page number of the first page of internal, arithmetic expression, text for the next domain. If this byte is zero, there is not another domain defined for the function.

The virtual pages used to store the text for a boolean expression or a local name table are linked together by the first word of each page. A zero link specifies the last page.

The name control block for a scalar symbolic equals function contains the same information as a scalar symbolic left arrow function. The text for the function is also stored in a similar fashion except that in the first virtual page
used to store the internal text for a domain's arithmetic expression bytes two, four and six are not used. There is no local name table required for an equals function since all non-parameter variables appearing in the function definition assume their current values when the function is evaluated. There is no reference count because if an equals function appears in the definition of a left arrow function, a copy of the equals function must be created. While the copy is being made the equals function is transformed into a left arrow function to insure that the values of all non-parameter variables are fixed.

If a name control block denotes an array of symbolic functions, it contains the same information as a numeric array name control block. In addition the first byte of the fourth word of the data portion contains the number of arguments in each of the functions.

The array is treated as if it is an array of real single precision numbers. Each element contains the virtual page number of the first page used to store the arithmetic expression text for the first domain of the element's definition. If an element is not defined, its value is zero. The text for the definition of each element is linked together in the same manner as a scalar symbolic function.

MAPSS is not designed for string processing but it does allow the user to create strings, concatenate them and assign
them to variables. This is done to permit the programmer to label his output. The data pointer byte of a string valued variable’s name control block contains the number of the string. The string number is the index of an entry in the string relocation vector. Each entry is subdivided into three bytes. Byte one contains the index of the start of the actual string description in the string picture table. The second byte contains the reference count for the string. The reference count designates the number of times the string variable has been concatenated to form another string, plus one. The third byte contains the index of the first word of the data portion of the name control block for the string variable.

The string picture table contains a description of each string. Several entries compose the description of a string. Each entry denotes either a literal string, a reference to a previously defined string variable, or the end of a string picture. An entry in the string picture table in subdivided into three bytes.

If byte one is not zero the entry describes a literal. Byte one is the number of characters in the literal, byte three is the number of the virtual page in which the literal is stored, and byte two is the displacement on that page to where the literal begins.

Each word in a virtual page used to hold string literals is subdivided into three bytes. A literal is divided into
segments of three characters. Each segment is stored in a byte. If a string literal will not fit in the current string page, the literal is broken. As many segments of the literal as possible are placed in the current page and the remainder are placed in a new string page. When this occurs two entries are placed in the string picture table. This avoids the problem of linking pages used to hold string literals. The maximum length of one string literal is 576 characters.

If byte one of a string picture table entry is 1313, then the entry denotes the null string. It has no length and does not require any storage, so byte two and three are unused.

If byte one is zero and byte three is not 501, the entry denotes a reference to a previously defined string variable. So that a new copy of the previously defined variable's string is not created, byte three contains the index of its entry in the string relocation table. When this occurs the reference count in the relocation table for the variable is increased by one.

If byte one is zero and byte three is 501, the entry denotes the end of a string picture.

When a name control block denotes an array of strings, it contains the same information as a numeric array. The array is treated as a single precision real array. The elements of the array contain the indices of the entries in
L. R. Symes

the string relocation table for the string descriptions. If an element is undefined, its value is zero.

As can be seen from the descriptions of the various data structures, the primary concerns in their design has been to facilitate their use as operands while at the same time reducing the amount of physical storage required.
ACKNOWLEDGEMENT

The work was supported in part by NSF Contract GP-05850.

REFERENCES


