TECHNICAL RESEARCH REPORT

Structural Matrix Computations with Units: Data Structures, Algorithms, and Scripting Language Design

by Mark Austin, Wane-Jang Lin, Xiaoguang Chen

T.R. 99-63

Abbreviated Version to Appear in ASCE Journal of Computing in Civil Engineering

STRUCTURAL MATRIX COMPUTATIONS WITH UNITS : DATA STRUCTURES, ALGORITHMS, AND SCRIPTING LANGUAGE DESIGN

By Mark A. Austin,[1]

Wane-Jang Lin[2] **and Xiaoguang G. Chen**[3]

ABSTRACT : Despite the well known benefits of physical units, matrices, and matrix algebra in engineering computations, most engineering analysis packages are essentially dimensionless. They simply hold the engineer responsible for selecting a set of engineering units and making sure their use is consistent. While this practice may be satisfactory for the solution of self-contained and well established problem solving procedures, where the structure of the output is well known and understood, identifying and correcting unintentional errors in the solution of new and innovative computations can be significantly easier when units are an integral part of the computation procedure. This report begins with a description of the data structures and algorithms needed to represent and manipulate physical quantity variables, and matrices of physical quantities. The second half of this report focuses on the implementation of Aladdin, a new computational environment for matrix and finite element calculations. Aladdin employs a novel combination of system programming languages, scripting language concepts, and stack machine technology. The result is a high-level scripting language that offers enhanced type checking for expressions and assignments, problem oriented scaling of variables, automatic conversion of systems of units, and program control structures for the solution of engineering problems. Functionality of the Aladdin stack machine is illustrated by working step by step through the parsing and execution of a simple statement involving units. The capabilities of Aladdin are demonstrated through the deflection analysis of a cantilever beam.

Keywords : Structural Analysis. Matrix Computations, Physical Units, Scripting Language Design, System Programming Languages, Finite Element Analysis.

[1] Associate Professor, Department of Civil Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742, USA.

[2] Design Engineer, Bridge Design Division, Maryland State Highway Administration, Baltimore, MD 21203, USA.

[3] Chief Technologist of Risk Management, Tuttle Decision System, Inc. 655 Redwood Highway, Suite 200, Mill Valley, CA, 94941

STRUCTURAL MATRIX COMPUTATIONS WITH UNITS : DATA STRUCTURES, ALGORITHMS, AND STACK MACHINE DESIGN

INTRODUCTION

Now that high-speed computers with Internet connectivity are readily available, many structural engineering companies see opportunities for using this technology to expand their domain of business activities, and enhance business productivity via computer-based support for engineering analysis (including design code rule checking, optimization, and interactive computer graphics), contract negotiations, procurement, management of project requirements, and access to engineering services. The participating computer software programs and networking systems must be integrated, fast and accurate, flexible, reliable, and of course, easy to use. The pathway from automated information management and ease-of-use to productivity gains is well defined (Austin, Chen, Lin 1995; Goudreau 1994).

With these opportunities in mind, a fundamental tenet of our work is that matrices, matrix algebra, and physical units are key ingredients of structural engineering systems integration. Matrices and linear matrix algebra are an essential part present-day structural analysis because they enable large families of equations and their solution procedures to be specified at a relatively high level of abstraction, and because matrix operations are ideally suited for step-by-step solution procedures on computers. In traditional approaches to problem solving, engineers employ units of measure to help relate the properties of an object to the real world, thereby generating problem descriptions that are easier to understand, maintain, and validate than if the units were omitted. Since the selection of a system of units is most often based on convenience and tradition, units of measure are just as important as the numerical quantity itself (Cmelik, Gehani 1988; Gehani, 1982; Karr, Loveman 1978).

Despite the well known benefits of matrices and matrix algebra, and physical units, most engineering analysis packages are essentially dimensionless. They simply hold the engineer responsible for selecting a set of engineering units and making sure their use is consistent. For example, MATLAB supports matrix calculations but does not support units of measurement at all (Mathworks 1995). Among the engineering analysis programs that do incorporate physical units, most handle them at either the program input and output stage (i.e., the input and output will be presented and displayed in a certain set of units) or at the physical quantity level. A number of engineering analysis programs allow a user to specify a set of units to be used (e.g., SI or US), and then assumes that all of the subsequent calculations will conform to the selected units set. Programming environments like MathCad (Mathsoft 1998) and Mathematica (Wolfram 1999), which support variables with physical units (e.g., x = 2 in), discard the same units when families of equations are written in matrix format. Indeed, we have not been able to find any computer programs that systematically integrate units into the definition of physical quantities, matrices, and finite element analysis.

While this practice may be satisfactory for the solution of self-contained and established problem solving procedures, where the structure of a problem's input/output is well understood, identifying and correcting unintentional errors in the solution of new and innovative computations can be significantly easier when units are an integral part of the computation procedure. In the evaluation of a new equation, for example, an environment includes handling of units can check all sub-expressions are dimensionally consistent before proceeding the equation evaluation. Similarly, the ability of a computational environment to represent and scale units facilitates interpretation of results – an engineer can choose, for example, to represent forces in either Newtons (N) or Kilo Newtons (kN), and distances in meters (m), centimeters (cm), or millimeters (mm). Failure to correctly identify the appropriate units in a calculation can lead to embarrassing failures. A case in point is NASA's $125 million dollar Mars Climate Orbiter which recently plowed into the Martian atmosphere because engineers failed to make a simple conversion from English units to metric in their navigation calculations (Sawyer 1999).

OBJECTIVES AND SCOPE

The objectives of this research are to use systems development methodologies and ideas from computer science to design and implement a scripting language and computational toolkit for engineering matrix and finite element analysis, with units an integral part of the scripting language. The integration of units into a scripting language offers enhanced type checking for expressions and assignments during compilation (and passing of parameters in function calls), and allows for problem oriented scaling of variables, and automatic conversion of systems of units (Manner 1986). The target application areas for this work are static and dynamic analysis of multi-story buildings and highway bridge structures, including earthquake resistant structures.

From an engineering perspective, measures of success include the degree to which matrix and finite element problem descriptions and their solution procedures are textually descriptive, extensible, and computational efficient. In the latter sections of this report we will see that these measures of success can be balanced with a judicious choice and implementation of scripting language and system programming language technologies. From a long-term business perspective, measures of success include the extent to which matrix and finite element computations can be integrated with other disciplines.

This report describes the architecture of a computational toolkit called Aladdin (Austin, Chen, Lin 1995), and the data structures, algorithms, high-level scripting language, and stack machine needed to represent and manipulate units, physical quantity variables, and matrices of physical quantities in an efficient manner. In Aladdin, finite element computations are viewed as a specialized form of matrix computations, matrices are viewed as rectangular arrays of physical quantities, and numbers are viewed as dimensionless physical quantities. In the second half of this report, functionality of the Aladdin stack machine is illustrated by working step by step through the parsing and execution of a simple statement involving units. The capabilities of Aladdin are demonstrated through the deflection analysis of a cantilever beam.

ALADDIN TOOLKIT

The preliminary result of this work is Aladdin (Version 2), a computational toolkit for interactive matrix and finite element analysis (Austin, Chen, Lin 1995). Our implementation has been inspired in part by the systems integration methods developed for the European ESPRIT Project (Kronlof 1993), namely that a system specification should contain four key components:

1. **A Model.** The model will include data structures for the information (i.e., physical quantities, matrices of physical quantities) to be stored, and a stack machine for the conversion of high-level problem descriptions into low-level stack machine operations, followed by the execution of these operations in a precise and unambiguous manner.
2. **A Language.** The language acts as an interface between the engineer and the underlying computational model. For our purposes, this meana designing a scripting language for the definition and manipulation of physical quantities, matrices of physical quantities, finite element meshes and numerical solution procedures. The scripting language should be a composition of data, control structures, and functions (Salter 1976).
3. **Defined Steps and Ordering of the Steps.** The steps will define the transformations that can be carried out on system components (e.g., nearly all engineering processes will require iteration and branching).
4. **Guidance for Applying the Specification.** Guidance includes factors such as easy-to-understand problem description files, documentation, and example problems.

The upper half of Fig. 1 is a high-level schematic of the Aladdin architecture; its three main parts are: (1) the input file(s), (2) the kernel, and (3) software libraries of units, matrix, and finite element functions. Matrix and finite element problems are written as blocks of statements in an input file. Solutions to matrix/finite element problems are computed using a combination of interpreted code from the input file, and compiled code located in the matrix/finite element software libraries. The interface between these systems is handled by Aladdin's kernel, implemented as a stack machine.

The lower half of Fig. 1 shows that the development of Aladdin is a composition of two problems, the development of the matrix and finite element libraries, and their integration into a working system. On the system libraries side of the problem, the main challenge lies in the design of data structures and algorithms for the underlying matrix and finite element operations that are computationally efficient. The main challenge in designing the kernel and input file is in finding a language syntax and semantics that will enable a number of disciplines involved in a problem solving procedure to be integrated in a seamless manner.

To be useful to engineers, the problem description language should be reasonably high level and yet precise, perhaps resulting in hundreds/thousands of low-level native machine instructions per Aladdin statement. It is important to keep in mind that as the speed of desktop computers increases, the time needed to prepare a problem description

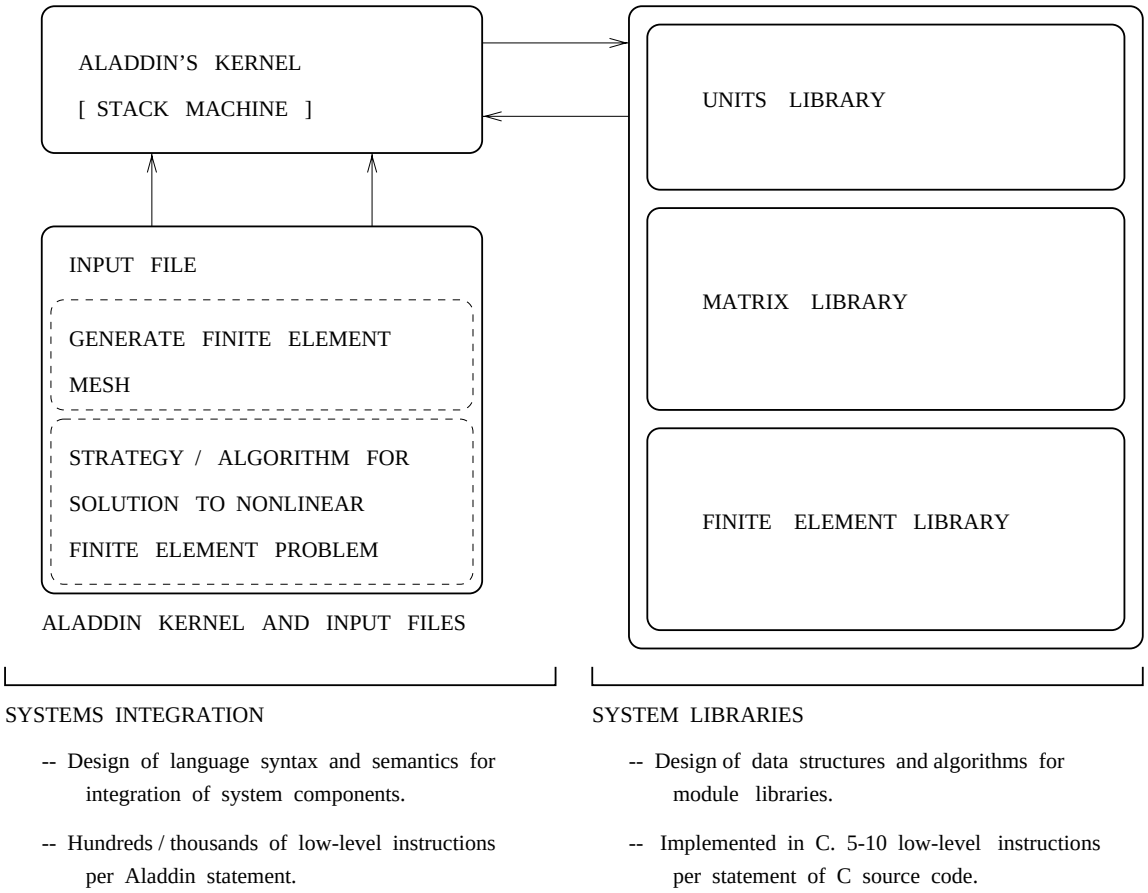


FIG. 1. : High Level Architecture for Aladdin

increases relative to the total time needed to work through an engineering analysis. Hence, clarity of an input file's contents is of paramount importance. The Aladdin scripting language achieves these goals with: (1) liberal use of comment statements (as with the C programming language, comments are inserted between `/* .... */`), (2) consistent use of function names and function arguments, (3) use of physical units in the problem description, and (4) consistent use of variables, matrices, and structures to control the flow of program logic. Aladdin's scripting language is C-like in the sense that it uses only a small number of keywords, and supports a variety of familiar looping and branching constructs. However, unlike C, the scripting language is designed from the bottom-up with definition and manipulation of physical quantities, and matrices of physical quantities in mind.

The functional components of Aladdin provide hierarchy to the solution of our matrix and finite element processes, and are located in libraries of compiled C code shown along the right-hand side of Fig. 1. Aladdin has library functions for basic matrix operations, the numerical solution to linear matrix equations, the symmetric eigenvalue problem, and a variety of finite element operations. The finite element library contains functions for adding nodes, elements, and external loads to the Aladdin database, specifying boundary conditions, forming the global stiffness matrix, and printing the system displacements and

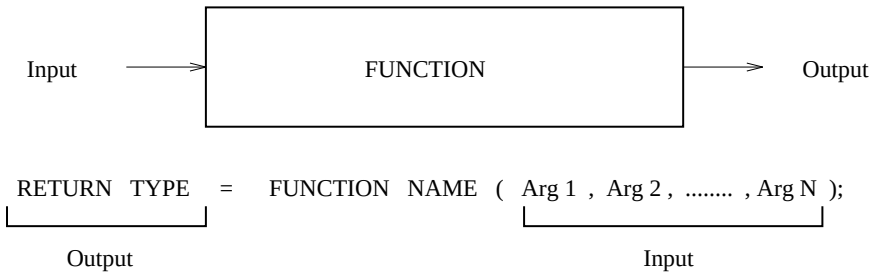


FIG. 2. : Schematic of Functions in Aladdin

stresses.

Fig. 2 shows the general components of a function call, including the input argument list, the function name, and return type. The strategy we have followed in Aladdin's development is to keep the number and type of arguments employed in library function calls small. Whenever possible, a function's return type and arguments should be of the same data type, thereby allowing the output from one or more functions to act as the input to following function calls. This approach to language development enables integration of application areas, as illustrated in the fragment of code:

```
returntype1 = Function1();                              /* <= area 1 */
returntype2 = Function2();                              /* <= area 1 */

returntype3 = Function3( returntype1, returntype2 );  /* <= area 2 */
```

If `Function1()` and `Function2()` belong to application area 1 (e.g., matrix analysis), and `Function3()` belongs to application area 2 (e.g., finite element analysis; optimization), then this language structure allows application areas 1 and 2 to be combined in a natural way.

PHYSICAL QUANTITIES

Aladdin supports three data types, "character string" for variable names, physical quantities, and matrices of physical quantities for engineering data. A physical quantity is a measure of some quantifiable aspect of the modeled world. In Aladdin, basic engineering quantities such as length, mass, and force, are defined by a numerical value (number itself) plus physical units. Fig. 3 is a subset of units presented in the Unit Conversion Guide (AICHE 1990), and shows the primary base units, supplementary units, and derived units that occur in structural analysis. The four basic units needed for engineering analysis are: length unit L; mass unit M; time unit t; and temperature unit T. Planar angles are represented by the supplementary base unit rad.

Derived units are expressed algebraically in terms of base and supplementary units by means of multiplication and division, namely:

$$\text{units} = k \cdot L^{\alpha} M^{\beta} t^{\gamma} T^{\delta} \cdot \text{rad}^{\epsilon} \tag{1}$$

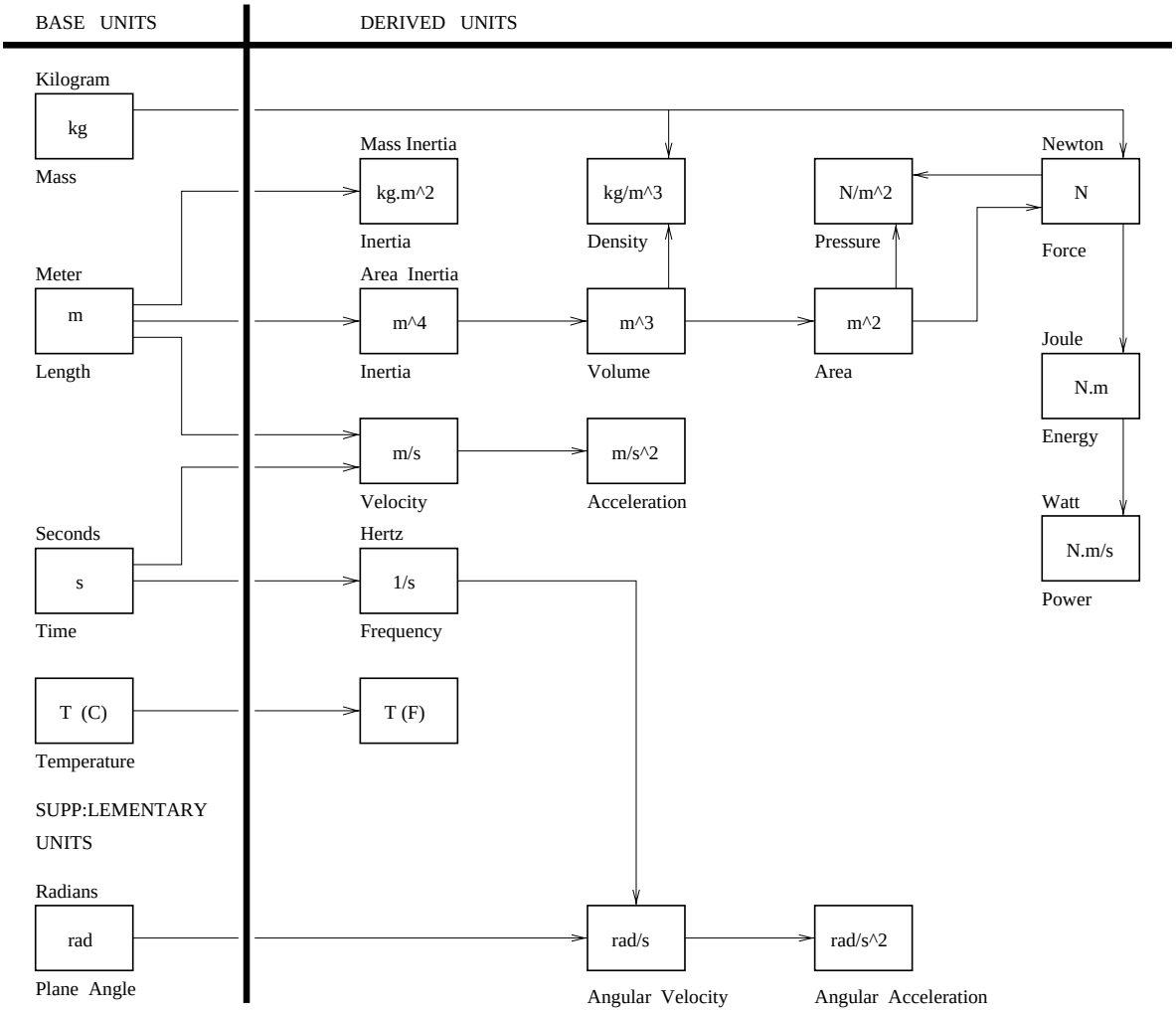


FIG. 3. Primary Base and Derived Units in Structural Analysis

where $\alpha, \beta, \gamma, \delta$ and ϵ are exponents, and k is the scale factor. Numbers are simply non-dimensional quantities represented by the family of zero exponents $[\alpha, \beta, \gamma, \delta, \epsilon] = [0, 0, 0, 0, 0]$.

The four basic units play the primary role in determining compatibility of units in physical quantity and matrix operations. Because a radian represents the ratio of two distances (i.e., distance around the perimeter of a circle divided by its radius), most software implementations deal with radians as if they were dimensionless entities. Aladdin departs from this trend by explicitly representing radians, and employing a special set of rules for their manipulation during physical quantity and matrix operations. The result is enhanced readability of program output, especially for computations involving displacements and rotations of structural components.

Physical Quantity Data Structure

With this basic units model in place, we can use the C programming language to define the physical quantity and units data structures:

```
#define  SI     100
#define  US     200
#define  SI_US  300

typedef struct {
        char    *cpUnitsName;    /* units name               */
        int       iUnitsType;    /* US or SI units           */
        double  dScaleFactor;    /* scale/conversion factor  */
        double  dLengthExpnt;    /* exponent for length      */
        double    dMassExpnt;    /* exponent for mass        */
        double    dTimeExpnt;    /* exponent for time        */
        double    dTempExpnt;    /* exponent for temperature */
        double  dRadianExpnt;    /* exponent for radians     */
} UNITS;

typedef struct {
        double    dValue;
        UNITS   *spUnits;
} QUANTITY;
```

Fig. 4 shows the layout of memory for one physical quantity.

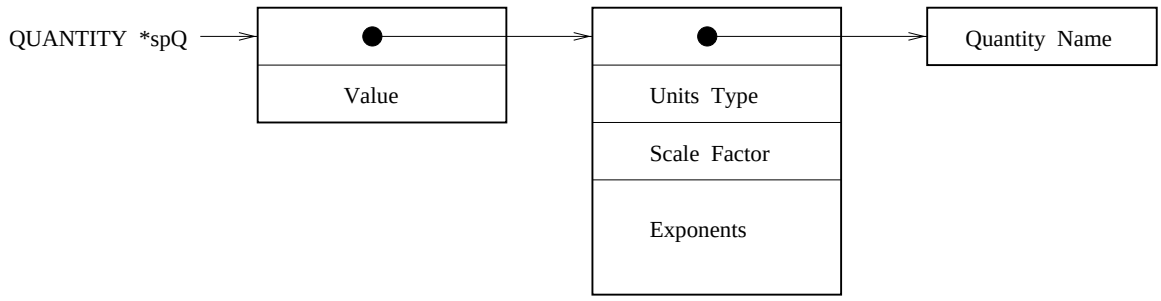


FIG. 4. : Layout of Memory in a Physical Quantity

`cpUnitsName` is a pointer to a character string storing the units name. `dScaleFactor` is the scale factor with respect to a basic unit. The `iUnitsType` flag allows for the representation of various systems of units (e.g., SI, US, and combined SI_US). SI_US units are those shared by both US and SI systems, the most important examples being time, frequency, and planar angle. We use variables of data type double to represent exponents, thereby avoiding mathematical difficulties in manipulation of quantities.

The Aladdin units package provides facilities for dynamic allocation of units, units copying, consistency checking and simplification, and units printing. Operations for units

conversion are provided, as are facilities to turn units operations on/off. In an effort to keep the scripting language usage and implementation as simple as possible, all physical quantities are stored as floating point numbers with double precision accuracy, plus units. Floating point numbers are viewed as physical quantities without units. There are no integer data types in Aladdin.

Physical Quantity and Variable Declarations

Aladdin stores physical quantities as "unscaled" values with respect to a set of SI units – meter "m" is the reference for length, kilogram "kg" for mass, second "sec" for time and "deg_C" for temperature. For example, the Aladdin statement

```
prompt >> print 20 kg, "\n";
```

generates the output

```
20 kg
```

Aladdin parses and stores the character sequence "20 kg" as the physical quantity twenty kilograms. The units name is "kg," the units scale factor `dScaleFactor` is 1, and the $[\alpha, \beta, \gamma, \delta, \epsilon]$ exponents are $[0, 1, 0, 0, 0]$. Notice how `20` juxtaposed with `kg` implies multiplication; we have hard-coded this interpretation into the Aladdin language because `20 kg` is more customary and easier to read than `20 * kg`. This quantity is discarded once the statement has finished executing.

We use the equals character (=) and the syntax:

```
identifier = expression ;
```

to assign a physical quantity to an identifier – each statement is terminated by a semicolon. Consider, for example, the block of statements:

```
t       = 3  sec;         /* time equals three seconds     */
x       = 2    m;         /* distance equals two meters    */
y       = 200 cm;         /* distance equals two meters    */
gravity = 981 cm/sec^2;   /* acceleration due to gravity   */
```

Table 1 shows the `QUANTITY` and `UNITS` data structure settings for each variable. Variable t represents three seconds. The numerical value for this quantity is 3 and the units name is "sec". While variables x and y both represent distances of two meters, and have a numerical value of 2, their units names are m and cm, respectively. When Aladdin prints a quantity, the output numerical value is the quantity numerical value divided by the scale factor associated with the units name. Therefore, the statements:

Variable Name	Quantity Value	Quantity Name	Units Type	Scale Factor	Length Expnt	Mass Expnt	Time Expnt	Temp Expnt	Radian Expnt
x	2.0	m	SI	1	1	0	0	0	0
y	2.0	cm	SI	0.01	1	0	0	0	0
t	3.0	sec	SI_US	1	0	0	1	0	0
gravity	9.81	m/sec^2	SI	1	1	0	-2	0	0

TABLE 1. Data Structure Contents for x, y, t, and *gravity*.

```
print "Distance x = ", x , "\n";
print "Distance y = ", y , "\n";
```

will generate the output:

```
Distance x =          2 m
Distance y =        200 cm
```

For variable x, the printed value is 2 because $2/1 = 2$. For variable y, the printed value is 200 because $2/0.01 = 200$. Finally, when Aladdin parses the assignment statement for *gravity*, the character sequence "`m/sec`$\wedge$2" is translated to the set of unit exponents $[\alpha, \beta, \gamma, \delta, \epsilon] = [1, 0, -2, 0, 0]$.

Special Names for Derived Units

A number of derived units have special names that may themselves be used to express other derived units in a simpler way than in base units. For example, Aladdin uses the symbol `N` to represent one Newton (i.e., $[\alpha, \beta, \gamma, \delta, \epsilon] = [1, 1, -2, 0, 0]$), and so the force

```
25 kg*m/sec/sec
```

can be simply written `25 N`. See Fig. 3.

Simplification and Casting of Physical Quantity Output

Because Aladdin stores all quantities internally with SI units, quantities specified in US units must be converted into their SI counterparts before they can be stored and used in calculations. Fortunately, for most sets of units (the important exception being temperature units), only a single scale factor is needed to convert a quantity from US into SI and vice versa (e.g., 1 in = 25.4 mm = 25.4E-3 m). Aladdin automates conversion between different but equivalent units, and provides features in its language for physical quantities to be cast and displayed in a desirable set of units. For example, with the definition of `gravity` in place, the statements:

```
print "Gravity g = ", gravity (cm/sec^2), "\n";
print "Gravity g = ", gravity (in/sec^2), "\n";
```

generates the output

```
Gravity g =        981 cm/sec^2
Gravity g =      386.2 in/sec^2
```

Both sets of units have the same dimension (i.e., length divided by time squared), with the numerical value of the physical quantity differing only by a scale factor. The stream of characters between `( ... )` indicates the desired format for the physical quantity units.

Of course, casting of physical quantity output also works with special names for derived units. This feature is useful for situations where a particular set of units exponents has more than one interpretation. For example, the unit exponents $[\alpha, \beta, \gamma, \delta, \epsilon] = [2, 1, -2, 0, 0]$ can be interpreted as a moment (force times distance) and as work done (force times distance). Casting of units allows for these cases to be distinguished, as in:

```
force    = 25 N;
distance = 1 m;

print "Moment    = ", force*distance, "\n";
print "Work done = ", force*distance (Jou), "\n";
```

The program output is:

```
Moment    =         25 N.m
Work done =         25 Jou
```

PHYSICAL QUANTITY ARITHMETIC

Aladdin supports the construction and evaluation of physical quantity expressions involving arithmetic, relational, and logical operators. Units and physical quantities are integrated into the Aladdin scripting language via equation (1) and the aforementioned data structures. Together, they enable the construction of arithmetic, relational, and logical operands that include a powerful check for the dimensional consistency of formulas.

Physical quantity expressions are evaluated in two steps. First, units within the expression are examined for compatibility with respect to the operation to be performed. Incompatible units will trigger the printing of an error message and termination of the program execution. Expressions containing compatible units proceed to step two, evaluation of the numerical expression with appropriate units for the result.

To see how operations of physical quantities works in practice, let q_1 be a physical quantity with unit scale factor k_1, and unit exponents $[\alpha_1, \beta_1, \gamma_1, \delta_1, \epsilon_1]$. And let q_2 be a physical quantity with unit scale factor k_2, and unit exponents $[\alpha_2, \beta_2, \gamma_2, \delta_2, \epsilon_2]$.

Expressions involving Arithmetic Operators

Table 2 contains a summary of arithmetic expressions together with the rules for computing the scale factor and unit exponents.

Description	Expression	Scale Factor	Unit Exponents
Addition	$q_1 + q_2$	k_1	$[\alpha_1, \beta_1, \gamma_1, \delta_1, \epsilon_1]$
Subtraction	$q_1 - q_2$	k_1	$[\alpha_1, \beta_1, \gamma_1, \delta_1, \epsilon_1]$
Multiplication	$q_1 * q_2$	$k_1 \cdot k_2$	$[\alpha_1 + \alpha_2, \beta_1 + \beta_2, \gamma_1 + \gamma_2, \delta_1 + \delta_2, \epsilon_1 + \epsilon_2]$
Division	q_1/q_2	k_1/k_2	$[\alpha_1 - \alpha_2, \beta_1 - \beta_2, \gamma_1 - \gamma_2, \delta_1 - \delta_2, \epsilon_1 - \epsilon_2]$
Exponential	$q_1 \,\hat{}\, q_2$	$k_1^{N\dagger}$	$[N\alpha_1, N\beta_1, N\gamma_1, N\delta_1, N\epsilon_1]^\dagger$

Note:

1. † N is the value of q_2.

TABLE 2. Units Arithmetic in Arithmetic Operations

The operations of addition and subtraction, $q_1 \pm q_2$, are defined when the units of q_1 and q_2 are equivalent. The rules for evaluation of units are as follows:

1. If the operands all have the same units type (e.g., let us say all are SI or US type), then the result units will be set according to the first operand.

2. If the operands have different units type, then the result units will be set according to the operand which has the same units type as the environmental units type.

3. When there are more than one operands which have the same units type as the environmental units type, the first one will be the basis for the result units.

For example, the script of code:

```
x = 2 in; y = 2 ft;
print "*** x     = " , x , "\n";
print "*** y     = " , y , "\n";
print "*** x + y = " , x + y , "\n";
print "*** y + x = " , y + x , "\n";
```

generates the output

```
*** x     =          2 in
*** y     =          2 ft
*** x + y =         26 in
*** y + x =      2.167 ft
```

Operator	Description	Example	Result
$<$	less than	$x < y$	true
$>$	greater than	$x > y$	false
$<=$	less than or equal to	$x <= y$	true
$>=$	greater than or equal to	$x >= y$	false
$==$	identically equal to	$x == y$	false
$!=$	not equal to	$x != y$	true
&&	logical and	$(x < y)$ && $(x <= y)$	true
$\|\|$	logical or	$(y < x)$ $\|\|$ $(x <= y)$	true
!	logical not	$!y$	false

TABLE 3. Expressions involving relational and logical operators

and shows rule 1 in action. For multiplication and division, the units of $q_1 \cdot q_2$ may be obtained by simply adding the exponents in the units of q_1 and q_2. The units of q_1/q_2 are computed by subtracting from the exponent coefficients in q_1, the exponent values in q_2.

The expression ${q_1}^{q_2}$ only makes sense when q_1 and q_2 are dimensionless, and cases where one of the two operands, but not both, has units. For example, if variable y has units and $x = y^{0.5}$, then the unit exponents of x will be half those of y.

Expressions involving Relational and Logical Operators

Table 3 summarizes the logical and relational operators that can be applied to physical quantities. Briefly, expressions involving the relational operators (e.g., $q_1 < q_2$, $q_1 \leq q_2$) are defined only when the units of q_1 and q_2 are equivalent. A units compatibility check is made before the operation proceeds, and the result of the operation is either true (1) or false (0). Assuming that variables `x` and `y` are as defined in the previous section, the Aladdin statements

```
print "(x >  y) evaluates to : ",  x > y, "\n";
print "(y >= x) evaluates to : ", y >= x, "\n";
```

generate the output

```
(x >  y) evaluates to :          0
(y >= x) evaluates to :          1
```

Similar Aladdin statements can be written for the logical operands, and expressions involving combinations of arithmetic, relational, and logical operands. See Table 3.

MATRICES OF PHYSICAL QUANTITIES

Matrices of physical quantities correspond to blocks of physical quantity elements. For example, the statement:

```
displ = [ 0 cm; 1 cm; 0.1 rad ];
```

defines an (1×3) matrix of displacements. Matrix elements `displ[1][1]` and `displ[2][1]` have units of length, and `displ[3][1]`, units of planar angle rotation.

Aladdin's engineering units module supports operations on engineering quantities and matrices for both US and SI systems of units. The matrix module supports storage of double precision floating point matrix elements, the specification of units for each matrix element, and a wide range of matrix operations. Indirect storage patterns are used for small general matrices. Skyline storage techniques are employed for large symmetric matrices.

Matrix Data Structure

Our development strategy enables any combination of data type and storages scheme to be implemented. Details of the matrix data structure are as follows:

```
/* Data structures for matrices of physical quantities */

typedef enum {
        IntegerArray = 1,
        DoubleArray = 2,
} DATA_TYPE;

typedef enum {
        Indirect = 1,
        Skyline  = 2,
} INTERNAL_REP;

typedef struct {
        char      *cpMatrixName;    /*  *name           */
        int             iNoRows;    /*  no_rows         */
        int          iNoColumns;    /*  no_columns      */
        UNITS       *spRowUnits;    /*  *row_units_buf  */
        UNITS       *spColUnits;    /*  *col_units_buf  */
        INTERNAL_REP       eRep;    /*  storage type    */
        DATA_TYPE         eType;    /*  data type       */
        union {
           int           **ipp;
           double        **dpp;
       } uMatrix;
} MATRIX;
```

Fig. 5 shows a high-level layout of memory for the matrix data structure. Memory is provided for a character string containing the matrix name, two integers for the number of matrix rows and columns, and integer flags for the basic data type and storage scheme

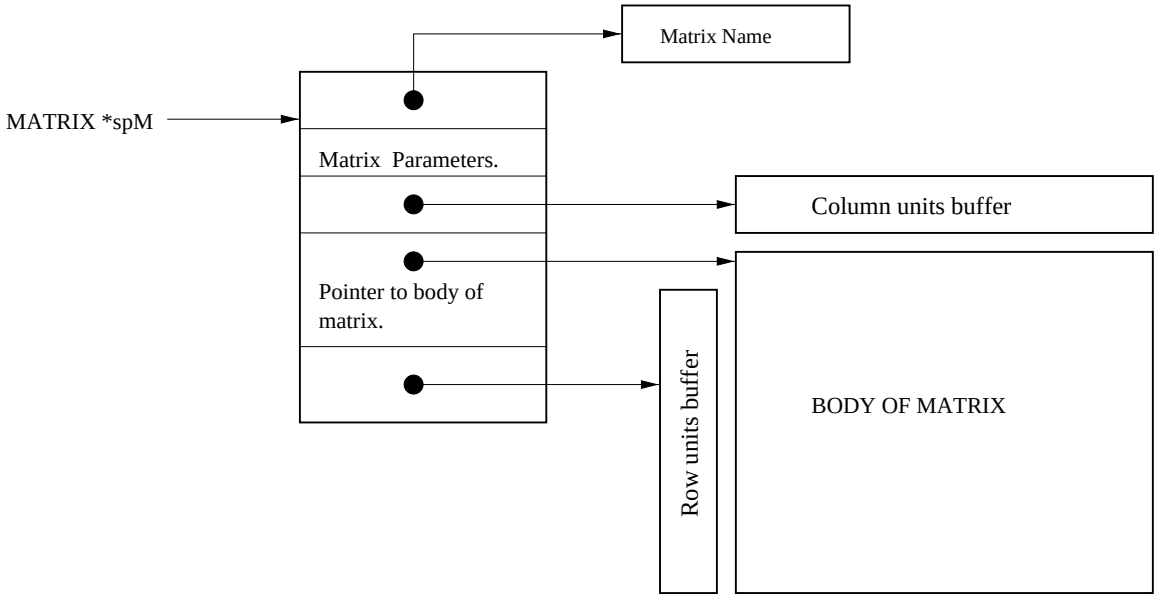


FIG. 5. Layout of Memory in Matrix Data Structure

for matrix elements. The union `uMatrix` contains pointers to matrix bodies of integers and double precision floats (only the latter have been implemented in Aladdin 2.0). The matrix element units are stored in two one-dimensional arrays of type `UNITS`. One array stores the column units, and a second array the row units. The units for matrix element at row `i` and column `j` are simply the product of the `i-th` element of the row units buffer and the `j-th` element of column units buffer.

Our use of row and column units matrices means that this model does not support the representation of matrices of quantities having arbitrary units. For most engineering applications, however, matrices are simply a compact and efficient way of describing families of equations of motion and equilibrium, and collections of data. Engineering considerations usually dictate that the terms within an equation be dimensionally consistent. Similarly, consistency of dimensions in large collections of engineering data also tends to hold. In practical terms, the assumptions made by this model not only have minimal impact on our ability to solve engineering problems with matrices, but requires much less memory than individual storage of units for all matrix elements. More precisely, for an $(n \times n)$ matrix, our storage scheme requires memory for $2n$ units. Individual storage of units for all matrix elements requires memory for n^2 units.

DYNAMIC ALLOCATION OF MATRICES

The step-by-step procedure for constructing a matrix with units is as follows:

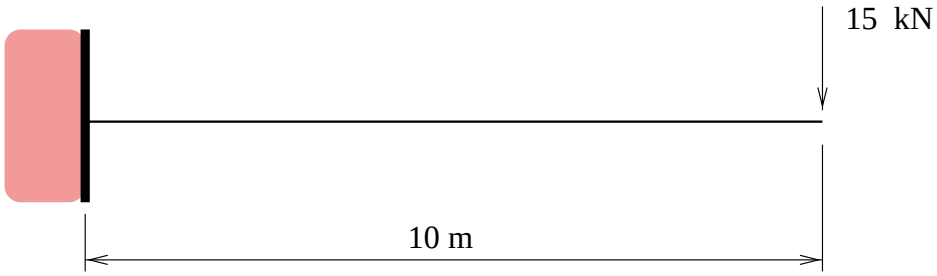
1. Dynamically allocate memory for `MATRIX`, and a character string for the matrix name.
2. Dynamically allocate memory for the matrix body.
3. Dynamically allocate memory for the row and column units matrices.

4. Initialize values for the matrix elements.

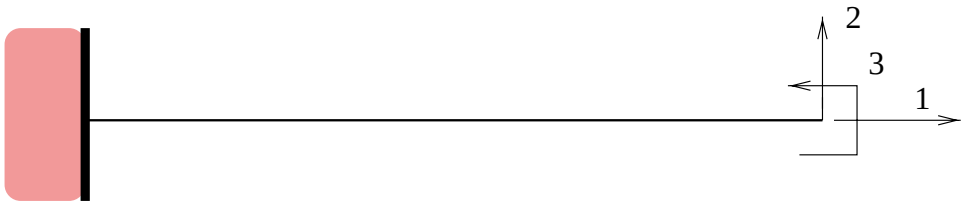
For small matrices, steps 2 and 3 can be combined by explicitly declaring a complete matrix of physical quantity elements.

Example

The purpose of this example is to see how units and matrices of physical quantities can be integrated into the displacement analysis of a cantilever beam carrying a vertical load of 15 kN at its tip. See Fig. 6a. The cantilever is 10 meters long and is constructed from mild structural steel (i.e., Young's Modulus of Elasticity is 200 GPa). The beam's moment of inertia and cross section area are 6.66e+08 mm^4 and 1.40e+04 mm^2, respectively.



(a) : Cantilever Beam and Point End load



(b) : Structural Model for Cantilever Beam

FIG. 6. Cantilever Beam with Supported End Point

A simple structural model for the cantilever beam is shown in Fig. 6b. Displacements in the cantilever system will be modeled with two translational degrees of freedom and one rotational degree of freedom at the cantilever tip. The statement:

```
eload = [ 0.0 N ; -15 kN; 0.0 N*m ];
```

defines a three-by-one matrix for the external loads applied to degrees of freedom 1 through 3. It is important to note that even though the external loads are zero for degrees of freedom one and three, we still specify the zero-sized quantity with units. Printing the external loads matrix with `PrintMatrix(eload);` gives

```
MATRIX : "eload"

row/col                 1
        units
   1        N   0.00000e+00
   2       kN  -1.50000e+01
   3      N.m   0.00000e+00
```

The script of code:

```
E = 200 GPa;
I = 6.66e+08 mm^4;
A = 1.40e+04 mm^2;
L = 10 m;
```

defines variables for the beam material and section properties, and its length. Details of the stiffness matrix for this problem can be found in almost any undergraduate text in structural engineering. Hence, we will focus exclusively on its specification here. The statement:

```
stiff = Matrix([3,3]);
```

dynamically allocates the layout of memory shown in Fig. 5, initializes all matrix elements to zero and the matrix name to "stiff," and zeros out all exponents in the row and column units matrices. The appropriate row and column units for this problem are specified with:

```
stiff = ColumnUnits( stiff, [ N/m, N/m, N/rad ]);
stiff = RowUnits( stiff, [m], [3] );
```

The first and second columns of `stiff` have column units `N/m`, and the third column, units of `N/rad`. The third row of `stiff` has row units `m`. Now the layout of memory and contents of the units buffers and matrix body are:

```
MATRIX : "stiff"

row/col                 1            2            3
        units         N/m          N/m        N/rad
   1            0.00000e+00  0.00000e+00  0.00000e+00
   2            0.00000e+00  0.00000e+00  0.00000e+00
   3        m   0.00000e+00  0.00000e+00  0.00000e+00
```

You should observe that while all of the matrix elements have a numerical value of zero, the matrix elements are not equal because they have different sets of units. For example, `stiff[1][1]` equals `0.0 N/m` and `stiff[3][3]` equals `0.0 N.m/rad`. With the exception of the radian units in the third column, the matrix units are symmetric. The matrix elements can be initialized with:

```
stiff[1][1] =  E*A/L;
stiff[2][2] = 12*E*I/L^3;
stiff[2][3] = -6*E*I/L/L;
stiff[3][2] = -6*E*I/L/L;
stiff[3][3] =  4*E*I/L;
```

Before the arithmetic expression on the right-hand side of each statement is assigned to the matrix element, the units of the evaluated expression are compared to the matrix element units, as defined by the row and column units buffers. Radians are excluded from the units compatibility check. The assignment will proceed if and only if units of the right- and left-hand sides of the assignment are compatible. (Otherwise, an error message will be printed, indicating an incompatibility of units and the file line number where the error occurs.)

Casting Units in Matrix Output

The Aladdin statement:

```
PrintMatrix ( stiff );
```

will generate:

```
MATRIX : "stiff"

row/col                  1            2            3
        units          N/m          N/m        N/rad
   1           2.80000e+08  0.00000e+00  0.00000e+00
   2           0.00000e+00  1.59840e+06 -7.99200e+06
   3        m  0.00000e+00 -7.99200e+06  5.32800e+07
```

By default, Aladdin will print the row/column units using appropriate base units for the SI or US system of units. Our techniques for casting units in physical quantities extend naturally to casting of units in matrices of physical quantities. For example,

```
PrintMatrixCast( stiff, [ kN/cm, kN/rad ]);
```

generates the output:

```
MATRIX : "stiff"

row/col                  1            2            3
        units         kN/cm        kN/cm       kN/rad
   1           2.80000e+03  0.00000e+00  0.00000e+00
   2           0.00000e+00  1.59840e+01 -7.99200e+03
   3        m  0.00000e+00 -7.99200e+01  5.32800e+04
```

The bracketed argument `[ kN/cm, kN/rad ]` tells Aladdin to search along the column units array, and scale and display matrix columns having compatible units in the new set of units. In this example, changing N/m to kN/cm requires a matrix element scale factor of 10^{-5}.

MATRIX OPERATIONS

Aladdin supports a wide variety of matrix operations, including addition and subtraction, matrix multiplication, solution of linear systems of equations, and the symmetric eigenvalue problem. Units are carried along with all matrix operations.

Addition and Subtraction

Addition and subtraction of matrices is a direct extension of addition and subtraction of physical quantities. The operation will proceed if and only if the matrices have the same number of rows and columns, and, the row and column buffers of both matrices are dimensionally consistent. Now let matrix `Z` be the result of `X` $\pm$ `Y`. Matrix `Z` will assume the same row and column units as `X`, with element values of `Z` scaled accordingly.

Multiplication

The handling of units in multiplication of two dimensional matrices needs special attention. Let `X` be a $(m \times n)$ matrix with row units buffer $[a_1, a_2, \cdots, a_m]$ and column units buffer $[b_1, b_2, \cdots, b_n]$. And let `Y` be a $(n \times r)$ matrix with row units buffer $[\, c_1, c_2, \cdots, c_n\,]$ and a column units buffer $[d_1, d_2, \cdots, d_r]$. The units for matrix elements x_{ik} and y_{kj} are $a_i * b_k$ and $c_k * d_j$, respectively. Moreover, let `Z` be the product of `X` and `Y`. From basic linear algebra we know that

$$z_{ij} = \sum_{k=1}^{n} x_{ik} * y_{kj} \tag{2}$$

Due to the consistency condition, all terms in the matrix element summation must have same units. That is, the exponents of the units must satisfy the product constraint $a_i b_1 c_1 d_j = a_i b_2 c_2 d_j = \cdots = a_i b_q c_q d_j$. Matrix element z_{ij} is assigned the units $a_i b_1 c_1 d_j$. The units buffers for matrix `Z` are written as a row units buffer $[\, a_1 c_1, a_2 c_1, \cdots, a_m c_1]$, and a column units buffer is $[\, d_1 b_1, d_2 b_1, \cdots, d_r b_1]$. It is important to notice that although the units for matrix `Z` are unique, the solution for the units buffers is not unique.

External Load Analysis for Displaced Cantilever Beam

Suppose that the cantilever beam shown in Fig. 6 has its tip displaced by 2 cm in the transverse directions, 0 cm in the horizontal direction, and zero radians in the rotational degree of freedom. With the aforementioned stiffness matrix in place, the block of Aladdin code:

```
displ = [ 0.0 cm; 2.0 cm; 0.0 rad ];
```

```
loads = stiff*displ;
PrintMatrix ( displ, loads );
```

defines a three-by-one displacement vector with appropriate units, and computes the matrix product `stiff*displ` for the loads needed to hold the cantilever in the displaced configuration. The third line of Aladdin code generates the output:

```
MATRIX : "displ"

row/col                 1
        units
   1       cm   0.00000e+00
   2       cm   2.00000e+00
   3      rad   0.00000e+00

MATRIX : "loads"

row/col                 1
        units
   1        N   0.00000e+00
   2        N   3.19680e+04
   3      N.m  -1.59840e+05
```

When Aladdin computes `stiff*displ`, exponents in the row and column units buffers of `stiff` are combined with the units of `displ`, namely:

$$[\,\mathrm{N/m} \quad \mathrm{N/m} \quad \mathrm{N/rad}\,] \cdot \begin{bmatrix} \mathrm{cm} \\ \mathrm{cm} \\ \mathrm{rad} \end{bmatrix} \Longrightarrow \begin{bmatrix} \mathrm{N} \\ \mathrm{N} \\ \mathrm{N.m} \end{bmatrix}.$$

As expected, the result is a three-by-one vector of two translational forces and one moment.

Linear Systems of Matrix Equations

The Aladdin commands:

```
displ = Solve( stiff, eload );
PrintMatrix( displ );
```

compute the solution to the linear equations `stiff*displ = force`, and generate the output:

```
MATRIX : "displ"

row/col                 1
        units
   1        m   0.00000e+00
   2        m  -2.60417e-06
```

```
3      rad  -7.81250e-07
```

Compatibility of units in the matrix equation `stiff*displ = force` dictates that:

$$[\,\mathrm{N/m} \quad \mathrm{N/m} \quad \mathrm{N/rad}\,] \cdot \begin{bmatrix} \text{displ[1][1] units} \\ \text{displ[2][1] units} \\ \text{displ[3][1] units} \end{bmatrix} = \begin{bmatrix} \mathrm{N} \\ \mathrm{kN} \\ \mathrm{N.m} \end{bmatrix} \Longrightarrow \begin{bmatrix} \text{displ[1][1] units} \\ \text{displ[2][1] units} \\ \text{displ[3][1] units} \end{bmatrix} = \begin{bmatrix} \mathrm{m} \\ \mathrm{m} \\ \mathrm{rad} \end{bmatrix}.$$

In other words, the column units of `stiff` multiplied by the row units of `displ` must match the row units of `eload`. By explicitly representing units for the third column as `N/rad` (versus N), radian units can be assigned to `displ[3][1]`. Including radians in the solution vector enhances its readability, and allows for appropriate conversion among units for planar angles.

PROGRAM CONTROL

Program control is the basic mechanism in a programming languages for using the outcome of logical and relational expressions to guide the pathway of a program execution. See Fig. 7.

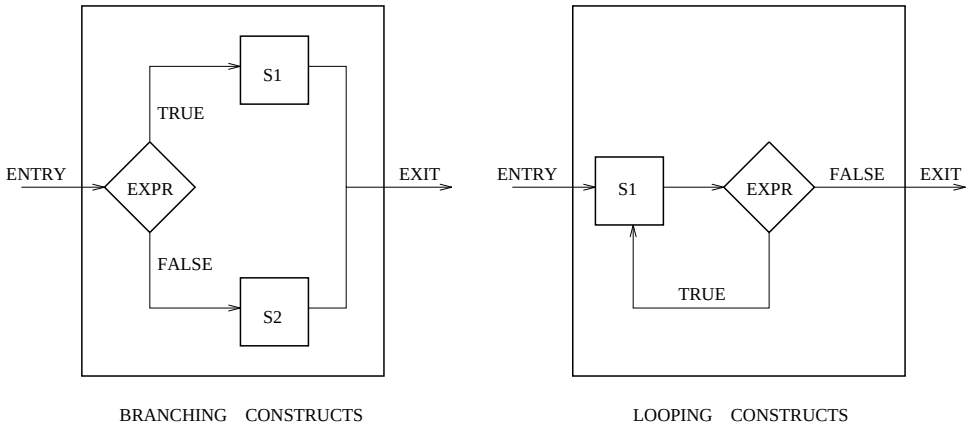


FIG. 7. : Branching and Looping Constructs in Aladdin

Aladdin supports the `if` and `if-then-else` branching constructs, and the `while` and `for` looping constructs, with logical and relational operations being computed on physical quantities.

If-then-else Branching Construct

The syntax for the if-then-else construct is as follows:

```
if ( conditional ) then {
```

```
     statements1;
} else {
     statements2;
}
```

`statements1` will be executed when the `conditional` expression evaluates to true. Otherwise, `statements2` will be executed. For example, the fragment of code:

```
x = 3 in;

if ( x < 2 in ) then {
   xnew = x + 1 in;
} else {
   xnew = x + 2 in;
}
print "x    =",x   , "\n";
print "xnew =",xnew, "\n";
```

generates the output:

```
x    = 3 in
xnew = 5 in
```

The relational expression x $<$ 2 in evaluates to true, hence, `xnew` is assigned the value `3 in` plus `2 in`.

For-looping Construct

The syntax for the for-loop construct is as follows:

```
for ( initializer ; condition ; increment ) {
      statements;
}
```

The `initializer`, `condition`, and `increment` statements can be zero or more statements separated by a comma. Similarly, zero or more statements may be located in the for-loop body. Of course, all elements in the for-loop statement are dimensionally consistent. For example, the block of statements

```
for ( x = 1 cm; x <= 5 cm; x = x + 2 cm ) {
      print  "x =",  x, "\n";
}
```

generates the sequence of activities and output shown in Table 4. Execution of the looping construct begins with the `initializer` statement, where we assign `1 cm` to variable `x`. The `condition` statement is then evaluated to see if it is true or false. In this particular example, `x <= 5 cm` evaluates to true and so statements inside the body of the loop are

Loop No.	x	x <= 5 cm	Output
1	1 cm	true	x = 1 cm
2	3 cm	true	x = 3 cm
3	5 cm	true	x = 5 cm
4	7 cm	false	

TABLE 4. : Activities and Output in for-looping Construct

executed (i.e., we print out the value of `x`). When all of the statements inside the `for-loop` body have finished executing, the `increment` statement is executed (i.e., we increase the value of `x` by `2 cm`) and `condition` is re-evaluated to see if another loop of computations is required. For this example, looping continues for 4 iterations, until the `condition` evaluates to false. At the end of the for-loop execution, the value of `x` will be `7 cm`. You should notice that all of the elements in the for-loop statement are dimensionally consistent.

ALADDIN SCRIPTING LANGUAGE DESIGN

Using grammatical rules and compiler construction tools based on the work of linguist Niam Chomsky, Aladdin's scripting language and stack machine enables translation of high-level engineering design and analysis concepts into mathematical and computational models. Working out the details of language translation requires a combination of design and artistic skills – the language aims to be textually descriptive and strike a balance between simplicity and extensibility. It uses a small number of (physical quantity and matrix) data types and control structures, incorporate physical units, and yet, is descriptive enough so that the "pencil and paper" and "problem description" files are almost the same.

Fig. 8 shows how the scripting language is connected to the stack machine and matrix and finite element applications libraries. System programming languages and scripting languages are complementary in their intended use and approach to program development, and Aladdin employs the best features of both language types to achieve the objectives stated in the previous section.

System Programming Languages and Scripting Languages

System programming languages, such as C, C++ and Java, are designed for building applications from data structures and algorithms from scratch. In an effort to make the development of large and complex programs more manageable, many system programming languages are strongly typed, meaning that a programmer declares how each piece of information will be used. Compilers can use this information to detect certain types of errors, and generate low-level code with performance optimized for the specified data type and underlying computer architecture. The need for error detection at run time is minimized. On the downside, computer program development with system programming

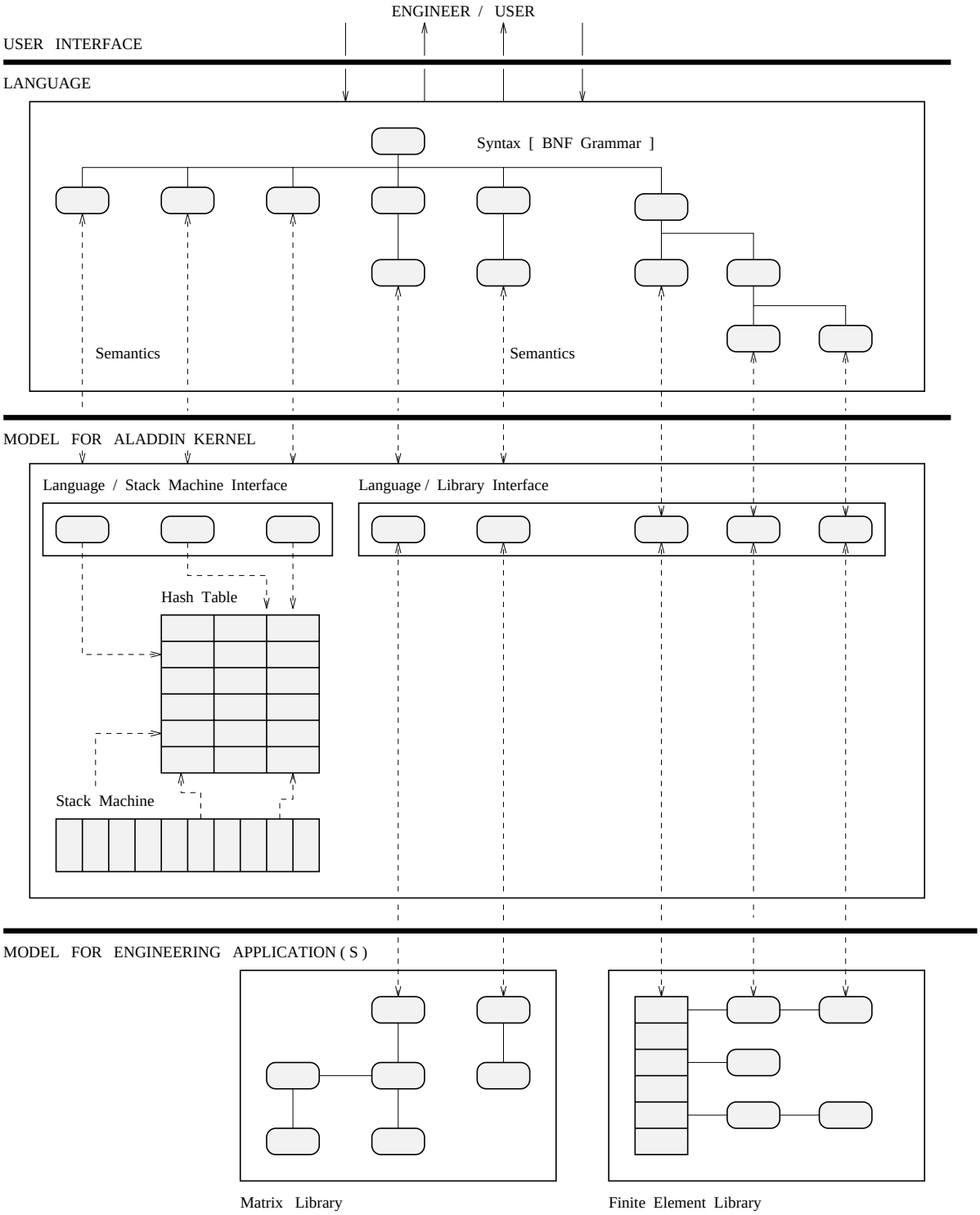


FIG. 8. : Interaction of Language and Underlying Model

languages typically takes place in a edit-compile-run cycle. Strong typing increases the number of interfaces needed to ensure component compatibility. This trend is at odds with principles of good systems design, namely, minimizing the complexity of system interfaces. As we have already seen, Aladdin's physical quantity and matrix libraries are implemented in C.

In contrast, scripting languages are designed for rapid, high-level, solutions to software problems, ease of use, and flexibility in gluing application components together. They facilitate this process by being weakly typed and interpreted at run time. Weakly typed means that few restrictions are placed on how information can be used a priori – the meaning and correctness of information largely determined by the program at run time. A key advantage of interpreters is flexibility and reduced time needed to work through an iteration of the problem solving process. Problem parameters and algorithms may be modified without having to recompile source code. Moreover, because scripting languages omit many of the complicated programming constructs found in system programming languages (e.g., data structures, inheritance), they often have a simpler syntax and are easier to learn than system programming languages. And since much of the code needed to solve a problem using a system programming language is due to the language being typed, broadly speaking, weakly typed scripting languages require less code accomplish a task (Ousterhout 1998).

ALADDIN STACK MACHINE

The heart of the Aladdin kernel is a finite-state stack machine model that follows in the spirit of work presented by Kernighan and Pike (1983). Stack machines are suitable for modeling systems that have a finite number of internal configurations or states, and whose behavior evolves as a linear sequence of discrete points in time.

Fig. 9 shows that the stack machine is constructed from three connected data structures - an array of machine instructions, a program stack, and a symbol table. The stack machine reads blocks of command statements from either the input file, or keyboard, and processes them in two phases:

Phase 1

High-level input statements are converted into an array of equivalent low-level stack machine instructions. Machine instructions include calls to functions, and push/pop data operands to/from the program stack shown on the upper right-hand side of Fig. 9.

The scripting language specification and conversion process is enabled by the UNIX tool YACC, an acronym for Yet Another Compiler Compiler. As shown in Fig. 8, we have designed a Bacus-Naur form (BNF) grammar for the definition and solution of matrix and finite element problems (Johnson 1975). YACC takes a language description (i.e., grammar) and automatically generates C code for a parser that will match streams of input against the rules of the language. For a gentle introduction to YACC, see Chapter 8 of Kernighan and Pike (1983).

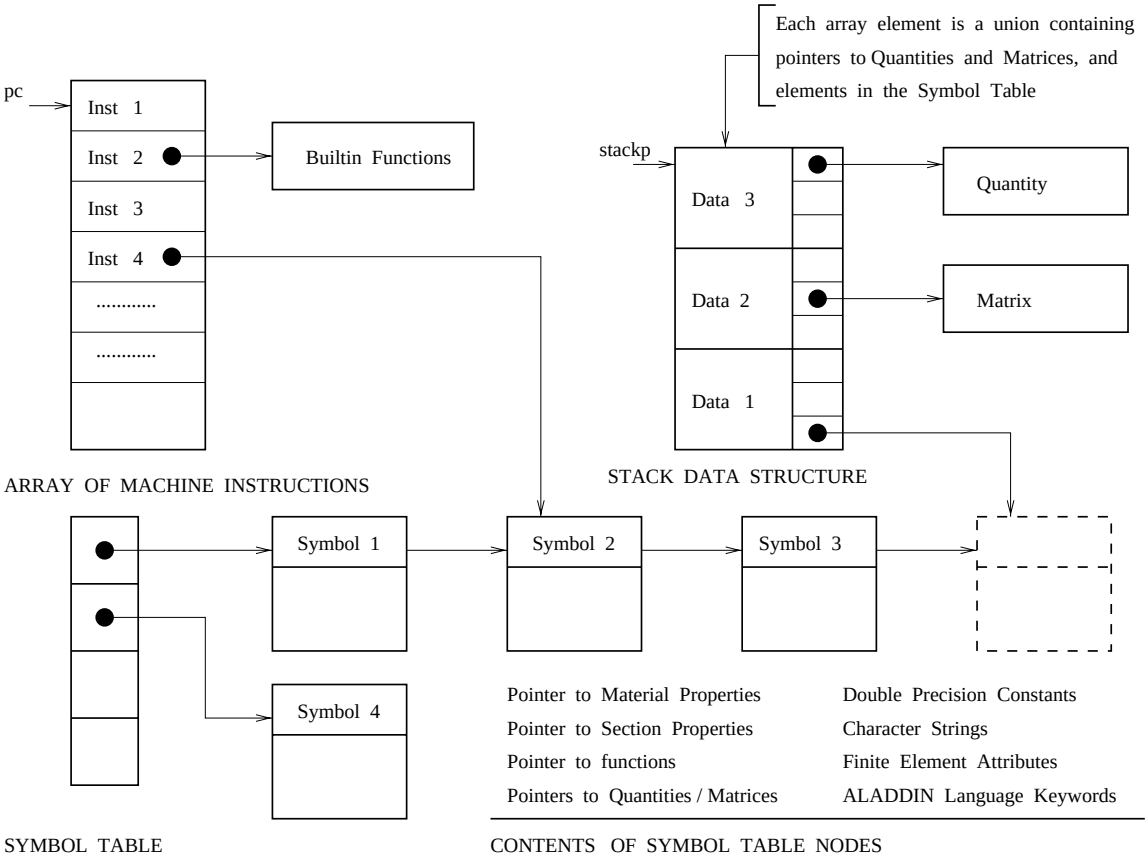


FIG. 9. : Data Structures in Aladdin's Stack Machine

Phase 2

The stack machine walks along the array of machine instructions and executes the functions pointed to by the machine instructions. Execution of the stack machine is handled by the C function:

```
int Execute( Inst *p ) {

    pc = p;
    while( *pc != STOP ) {
        pc = pc + 1;         /* Increment program counter   */
        if( Check_Break() )  /* Check for break condition   */
            break;
        (*(*(pc-1)))();      /* Execute machine instruction */
    }
}
```

The `while()` loop iterates until either a `break` command is encountered in the input, or a `STOP` instruction is reached. `pc` is a program counter that points to elements in the array of machine instructions. Most of the machine instructions are pointers to information

in the symbol table, or, pointers to C functions that handle low-level manipulation and transfer of data (e.g., pushing and popping data to/from the stack, evaluation of arithmetic, relational, and logical expressions, transfer of variables and matrices between the symbol table and program stack).

Example of Machine Stack Execution

We now demonstrate use of the stack machine by working step by step through the details of processing the assignment

```
x = 2 in;
```

Fig. 10 shows the parse tree for this statement.

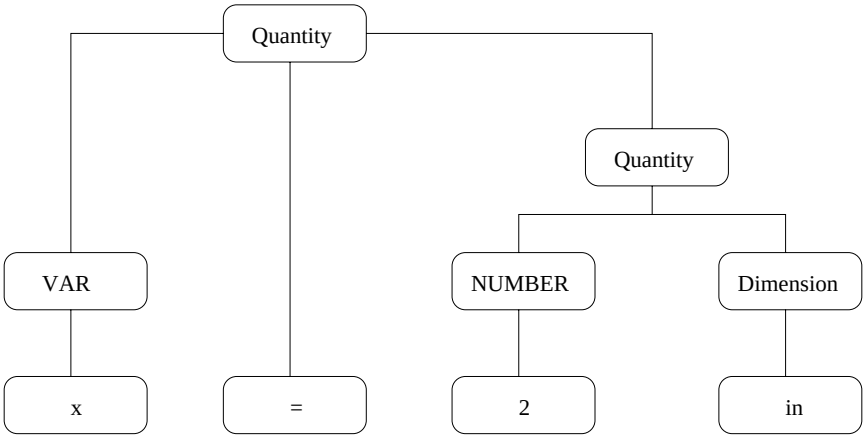


FIG. 10. : Parse tree for x = 2 in

Figures 11a through 11c show the relevant details of the machine array, symbol table, and program stack at critical stages of the statement evaluation.

Phase 1

The YACC parser recognizes that `x` is a variable, and the character sequence `2 in` is a quantity with number `2` and a units `in`. The grammatical rule for assignment (i.e., `"="`) has right associativity, meaning that when the machine is executed, `2 in` will be pushed onto the stack, and processed, before the assignment to `x` is made. At the conclusion of Phase 1, the array of machine instructions is as shown on the left-hand side of Fig. 11a.

Phase 2

The step-by-step procedure for execution of the stack machine is as follows:

1. `Push_Dimension()` allocates memory for a new quantity, `d1.q`, and initializes its contents with a copy of `in` from the symbol table. See Fig. 11a.

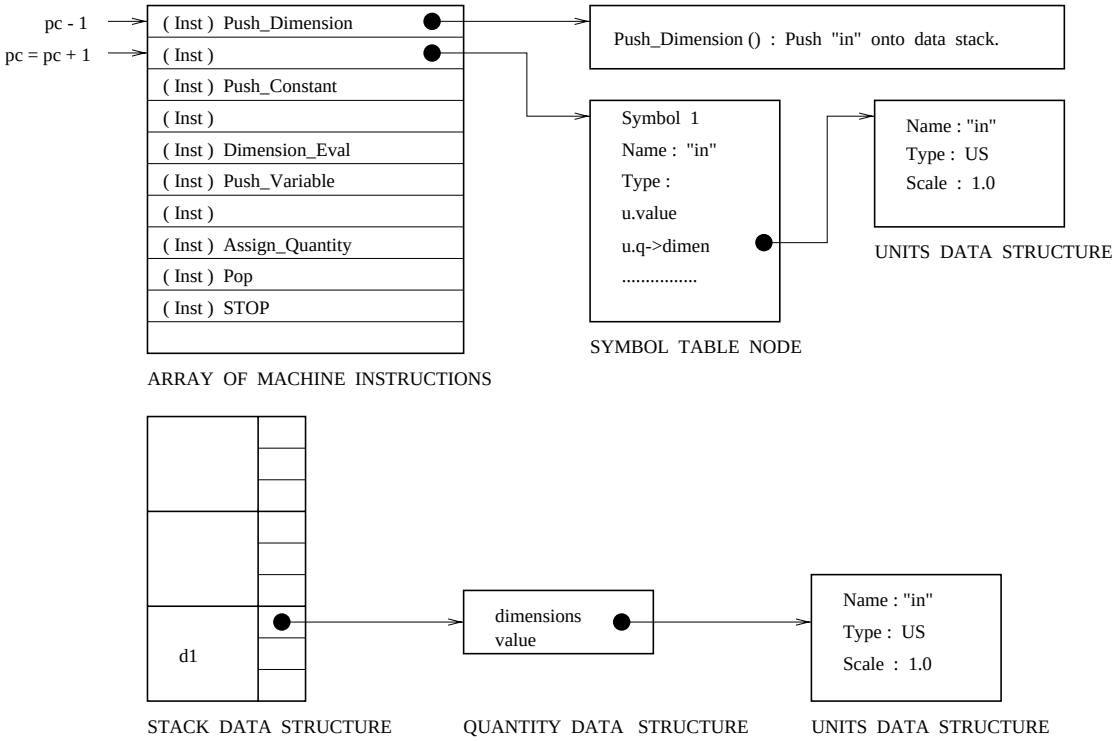


FIG. 11a. Step 1 : Push Unit onto Stack

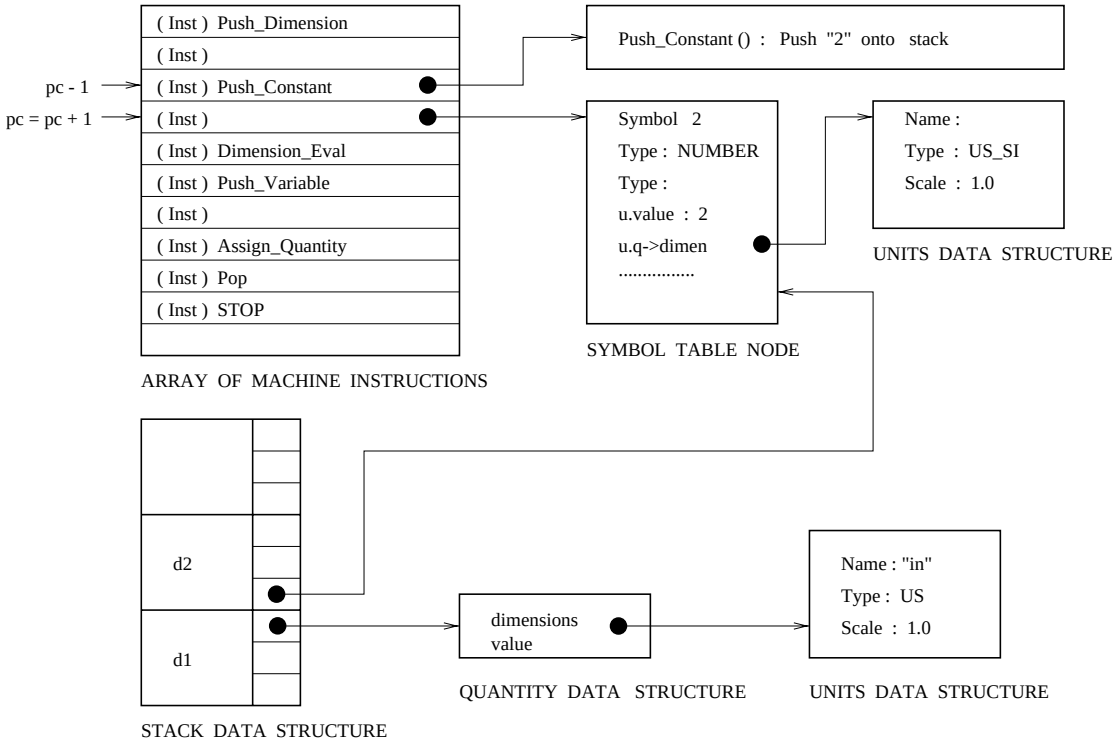


FIG. 11b. Step 2 : Push Number onto Stack

FIG. 11. : Data Structures in Aladdin's Stack Machine

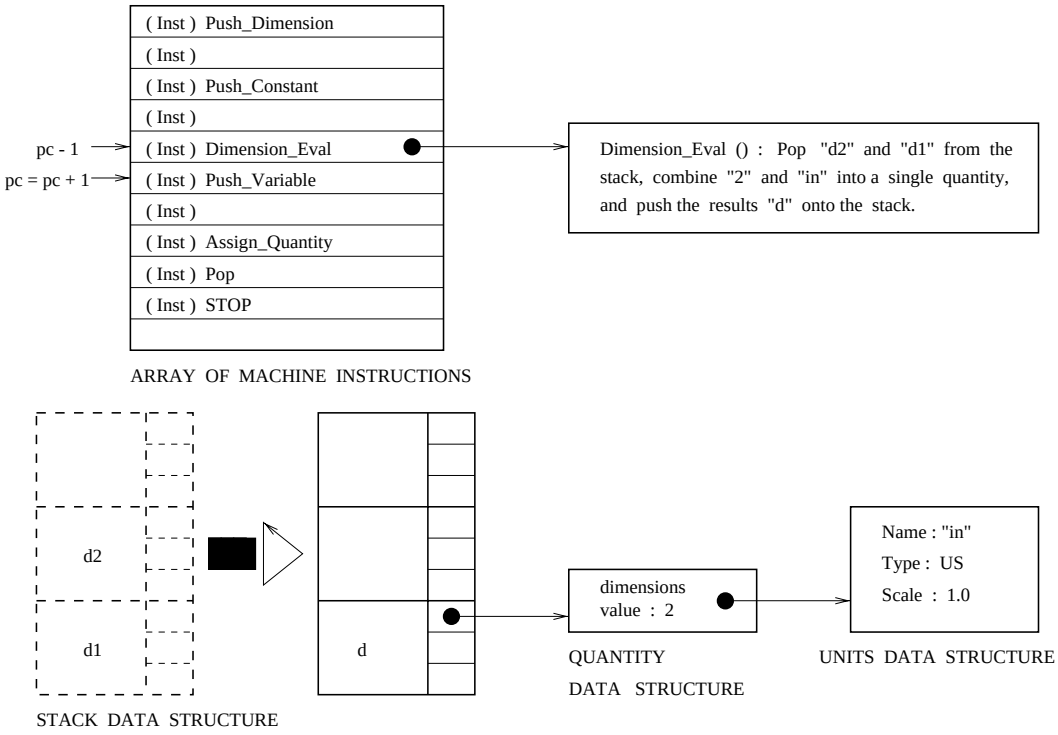


FIG. 11c. Step 3 : Combine Number and Unit into Quantity

FIG. 11. : Data Structures in Aladdin's Stack Machine

2. `Push_Constant()` allocates memory for a new quantity, `d2.q`, and initializes it with a copy the number `2` from the symbol table. See Fig. 11b.
3. `Dimension_Eval()` pops `2` and `in` off the stack, assigns the units `in` to number `2`, and pushes the quantity `2 in` back onto the stack. See Fig. 11c.
4. `Push_Variable()` pushes a symbol table pointer to variable `x` onto the stack.
5. `Assign_Quantity()` pops `x` and `2 in` from the stack, assigns `2 in` to `x`, and pushes `x` back onto the stack.
6. Data `x` is popped and cleared from the stack.
7. The looping mechanism within `Execute()` is halted by the instruction `(Inst) STOP`.

The program counter `pc` is incremented (i.e., `pc = pc + 1`) immediately before each of the steps 1 through 7.

Machine Execution for For-Loop

The implementation of branching and looping constructs is complicated by the need to store multiple blocks of machine code, and manage a pathway of execution governed by the outcome of relational expressions. To see how this works in practice, Fig. 12 shows the arrangement of stack machine instructions needed for the implementation of a for-looping

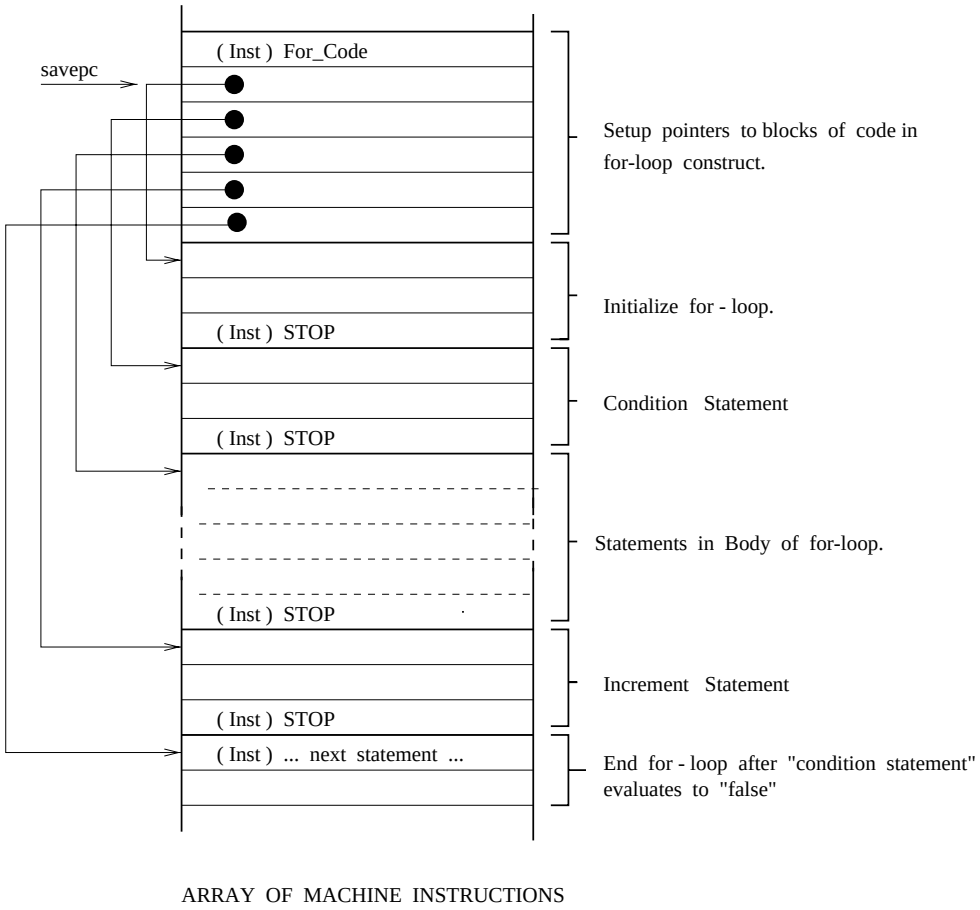


FIG. 12. : Stack Machine Instructions for For-loop Construct

construct. Four pointers are needed for the four segments of code that define a for loop (i.e., initialization, conditional statement, statements within the body of the for loop, increment statement). A fifth pointer is needed for the statement to be executed immediately after the for loop has finished its execution. The abbreviated details of `For_Code()` are as follows:

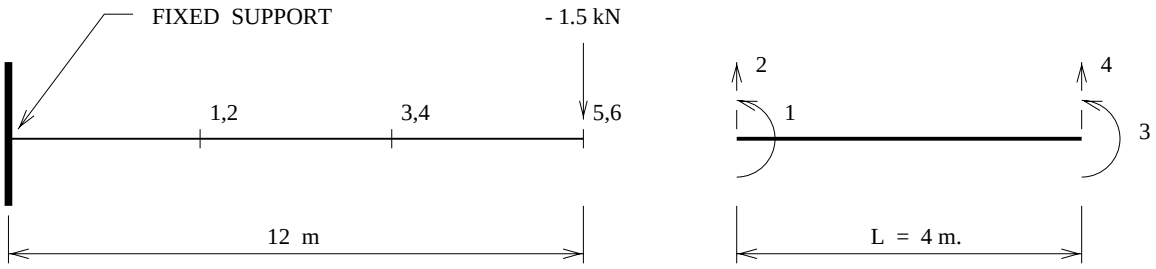
```
int For_Code() {
DATUM d;
Inst *savepc = pc;

    Execute(*((Inst **) savepc)) ;        /* Initialize variables     */
    Execute(*((Inst **) (savepc+1))) ;    /* Evaluate conditions      */

    d = Pop();
    while(d.q->value) {
```

```
        if( Check_Break() )                /* Check for break in code  */
            break;
        Execute(*((Inst **)(savepc+2))) ;  /* Execute body of code     */
        Execute(*((Inst **)(savepc+3))) ;  /* Increment variables      */

        if( Check_Break() )                /* Check for break in code  */
            break;
        Execute(*((Inst **) (savepc+1))) ; /* Evaluate conditions      */
        d = Pop();
    }

    pc = *((Inst **)(savepc + 4));         /* Set program pointer to   */
                                           /* next statement           */

    After_Break();                         /* Handle break conditions  */
}
```

The `while()` loop within `For_Code()` will iterate until `d.q->value` evaluates to false – that is, the condition expression(s) no longer evaluates to a value that is true. We have assembled similar structures for the `while` looping construct, and the `if` and `if-then-else` branching constructs.

EXAMPLE : DEFLECTION ANALYSIS OF CANTILEVER BEAM

In this example we use Aladdin's scripting language to compute displacements in the cantilever beam shown in Fig. 13. We have deliberately selected a problem with a small number of unknowns so that step-by-step details of the problem formulation and solution procedure can be illustrated using functions from the matrix library alone. Of course, the same example can be implemented in far fewer scripting statements by calling builtin functions from the finite element library.



DESCRIPTION OF CANTILEVER BEAM. BEAM FINITE ELEMENT

FIG. 13. : Single Span Cantilever Beam with Supported End Point

The cantilever has length 12 m, modulus of elasticity $E = 200\ GPa$, and constant moment of inertia $I = 15.5 \times 10^6\ mm^4$ along its length. We will assume that the cantilever is rigid in its axial direction (i.e., beam cross section area $\approx \infty\ m^2$), and that all other deformations are small. The boundary conditions are full-fixity at the base of the cantilever. Overall behavior of the cantilever will be modeled with three beam finite elements, and six global degrees of freedom, as numbered on the left-hand side of Fig. 13. Displacements along each beam element will be expressed as the superposition of four cubic interpolation functions, each weighted by a nodal displacement factor corresponding to the beam-end lateral displacements and rotations.

Definition of Element Stiffness Matrix

It is well known that if cubic interpolation functions are used to describe displacements along the beam element, then the beam element stiffness matrix will be

$$K_e = \left[\frac{2EI}{L}\right] \cdot \begin{bmatrix} 2 & 3/L & 1 & -3/L \\ 3/L & 6/L^2 & 3/L & -6/L^2 \\ 1 & 3/L & 2 & -3/L \\ -3/L & -6/L^2 & -3/L & 6/L^2 \end{bmatrix},$$

where EI is the flexural rigidity of the beam element, and L its length. The abbreviated fragment of Aladdin code:

```
/* [a] : Define section/material properties */

E    = 200000   MPa;
I    = 15.5E+6 mm^4;
L    = 4       m;

/* [b] : Define (4x4) stiffness matrix for beam element */

stiff = Matrix([4,4]);
stiff = ColumnUnits( stiff, [N/rad,  N/m,  N/rad,  N/m] );
stiff =    RowUnits( stiff, [m], [1] );
stiff =    RowUnits( stiff, [m], [3] );

stiff[1][1] =   4*E*I/L;     stiff[1][2] =  6*E*I/(L^2);
stiff[1][3] =   2*E*I/L;     stiff[1][4] = -6*E*I/(L^2);

.... details of array initialization removed .....

stiff[4][1] =  -6*E*I/(L^2); stiff[4][2] = -12*E*I/(L^3);
stiff[4][3] =  -6*E*I/(L^2); stiff[4][4] =  12*E*I/(L^3);

PrintMatrix(stiff);
```

defines the cantilever material and section properties, and then systematically assembles the element stiffness matrix with appropriate units. The last Aladdin statement generates the output:

```
MATRIX : "stiff"

row/col                  1            2            3            4
        units        N/rad          N/m        N/rad          N/m
   1        m   3.10000e+06  1.16250e+06  1.55000e+06 -1.16250e+06
   2            1.16250e+06  5.81250e+05  1.16250e+06 -5.81250e+05
   3        m   1.55000e+06  1.16250e+06  3.10000e+06 -1.16250e+06
   4           -1.16250e+06 -5.81250e+05 -1.16250e+06  5.81250e+05
```

Rows 1 and 3 of `stiff` represent equations of equilibrium when a moment is applied at the beam end to cause a unit rotation, and rows 2 and 4, equations of equilibrium for unit lateral displacements of the beam element. We distinguish these cases by applying units of `m` to rows 1 and 3.

Destination Array

A straightforward way of assembling these matrices is with the use of destination arrays, namely:

```
/* [c] : Setup destination array */
```

```
LD = [ 0, 0, 1, 2  ;
       1, 2, 3, 4  ;
       3, 4, 5, 6 ];
```

Element `LD[i][j]` contains the global degree of freedom for degree of freedom `j` of beam finite element `i`. A zero entry indicates that the beam element degree of freedom is fully fixed, and will not be mapped to the global stiffness matrix.

Assembly of Global Stiffness Matrix

The assembly process for the global stiffness matrix may be written

$$\mathbf{K} = \sum_{i=1}^{3} K_{ei}$$

where K_{ei} represents the stiffness matrix for beam element `i` mapped onto the global degrees of freedom. The following block of Aladdin code:

```
/* [d] : Allocate memory for global stiffness matrix */

gstiff = Matrix([6,6]);
gstiff = ColumnUnits( gstiff, [ N/rad, N/m,  N/rad,  N/m, N/rad, N/m ] );
gstiff =    RowUnits( gstiff, [m], [1] );
gstiff =    RowUnits( gstiff, [m], [3] );
gstiff =    RowUnits( gstiff, [m], [5] );

/* [e] : Assemble global stiffness matrix for three element cantilever */

no_elements = 3;
for( i = 1; i <= no_elements; i = i + 1) {
for( j = 1; j <= 4; j = j + 1) {

   row = LD [i][j];
   if( row > 0) {
       for( k = 1; k <= 4; k = k + 1) {
           col = LD [i][k];
           if( col > 0) {
               gstiff[ row ][ col ] = gstiff[ row ][ col ] + stiff[j][k];
           }
       }
   }
}
}
```

allocates memory for the (6×6) global stiffness matrix, assigns appropriate row and column units, and then systematically assembles the global stiffness matrix.

While blocks [d] and [e] do a nice job of illustrating the algorithm for assembling the global stiffness matrix, asking a programmer to manually assemble the global stiffness matrix via the LD matrix is tedious and error prone. Moreover, execution of the global stiffness assembly will be slow because the corresponding input file instructions are interpreted. All of these problems can be mitigated by calling the compiled C function, Stiff(), in Aladdin's finite element library, The single statement:

```
gstiff = Stiff();
```

replaces blocks of code [d] and [e] shown above.

External Load Vector/Solution of Linear Matrix Equations

An external load of -1.5 kN is applied to degree of freedom six. The required Aladdin code, with appropriate units, is as follows:

```
eload = [  0.0 N*m;
           0.0   N;
           0.0 N*m;
           0.0   N;
           0.0 N*m;
          -1.5  kN ];
```

Now we can compute and print the cantilever displacements by simply writing:

```
displ = Solve( gstiff, eload );
PrintMatrix(displ);
```

The displacements are:

```
MATRIX : "displ"

row/col                 1
        units
   1      rad  -1.93548e-02
   2        m  -4.12903e-02
   3      rad  -3.09677e-02
   4        m  -1.44516e-01
   5      rad  -3.48387e-02
   6        m  -2.78710e-01
```

The vertical deflection of the cantilever endpoint is 27.87 cm, downwards.

CONCLUSIONS AND FUTURE WORK

Aladdin (Version 2.0) is a computational toolkit for the interactive matrix and finite element analysis of large engineering structures. The toolkit views finite element computations as a specialized form of matrix computations, matrices as rectangular arrays of physical quantities, and numbers as dimensionless physical quantities. The data structures and algorithms described in this report have been used to solve engineering problems containing as many as 2500 matrix rows and columns. For a wide range of application examples and the program source code, see the Aladdin web site at http://www.isr.umd.edu/~austin/aladdin.html.

Our original goal for Aladdin was a computational environment that would help engineers setup and solve matrix problems in structural analysis and mechanics, and finite element problems in building and highway bridge analysis. However, it is now evident from the large number of Aladdin source code downloads (3,000+) that Aladdin is useful for the solution of a much wider range of engineering and scientific applications. To enhance the usefulness of Aladdin across engineering and business domains, the units package should be extended to include units of electric charge, and non-physical units, such as money/currency. The latter would be appropriate for costing programs that may need to operate in one or more currencies.

We are currently working on finite elements and numerical algorithms for the analysis of structures undergoing large geometric displacements, problems involving fluid-structure interaction, and numerical algorithms for the hybrid control of large structural systems. The results of this work will be released in Aladdin (Version 3).

ACKNOWLEDGMENTS

This work was supported in part by NSF's Engineering Research Centers Program : NSFD CDR 8803012, and by a series of Federal Highway Administration Fellowship Grants to Wane-Jang Lin. This support is gratefully acknowledged. The views expressed in this report are those of the authors, and not necessarily those of the sponsors.

APPENDIX I. REFERENCES

AICHE (1990). Unit Conversion Guide, Fuels and Petrochemical Division of AICHE.

Austin M.A., Chen X.G., and Lin W-J (1995). Aladdin: A Computational Toolkit For Interactive Engineering Matrix And Finite Element Analysis, *Technical Research Report TR 95-74*, Institute for Systems Research, University of Maryland, College Park, MD 20742.

Cmelik R.F., Gehani N.H. (1988). Dimensional Analysis with C++, *IEEE Software*, pp. 21-26.

Gehani N.H. (1982). Databases and Units of Measure, *IEEE Transactions on Software Engineering*", Vol. 8, No. 6, pp. 605-611.

Goudreau G.L. (1994). Computational Structural Mechanics : From National Defense to National Resource, *IEEE Computational Science and Engineering*, Vol. 1, No. 1, pp. 33-42.

Johnson S.C. (1975). YACC - Yet another Compiler Compiler, Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey.

Karr M., Loveman D. B. (1978). Incorporating Units into Programming Languages, *Communications of the ACM*, Vol. 21, No. 5, pp. 385-391.

Kernighan B.W., Pike R. (1983). The UNIX Programming Environment, Prentice-Hall Software Series.

Kronlof K. (1993). Method Integration : Concepts and Case Studies, John-Wiley and Sons.

Manner R. (1986). Strong Typing and Physical Units, *Association of Computing Machinery SIGPLAN Notices*, Vol. 21, No. 3, pp. 11-20.

MathCad 8 User's Guide (1998). *Mathsoft Inc*, Cambridge, MA.

Mathworks (1995). The Student Edition of MALAB, Version 4 User's Guide, MATLAB Curriculum Series, Prentice-Hall.

Ousterhout J. (1998). Scripting : Higher Level Programming for the 21st Century, IEEE Computer Magazine, March.

Salter K.G. (1976). A Methodology for Decomposing System Requirements into Data Processing Requirements, Proc. 2nd Int. Conf. on Software Engineering, San Francisco.

Sawyer K. (1999), Engineers' Lapse Led to Loss of Mars Spacecraft : Lockheed didn't tally Metric Units, The Washington Post, October 1, 1999.

Wolfram S. (1999). The Mathematica Book, Fourth Edition, Cambridge University Press.

APPENDIX II. NOTATION

The following symbols are used in this report:

`A` = cross section area of beam element;

`a` = element of row units buffer;

`b` = element of column units buffer;

`C` = degrees Centigrade;

`c` = element of row units buffer;

`d` = element of column units buffer;

`E` = Young's Modulus of Elasticity;

`displ` = matrix of structural displacements;

`distance` = length physical quantity;

`eload` = matrix of external loads;

`gravity` = gravitational acceleration;

`gstiff` = global stiffness matrix;

`F` = degrees Fahrenheit;

`force` = force physical quantity;

`I` = moment of inertia;

`i` = matrix row number;

`j` = matrix column number;

`K` = global stiffness matrix;

K_e = element stiffness matrix;

K_{ei} = stiffness matrix for beam element `i` mapped onto the global degrees of freedom;

`k` = units scale factor, and, element stiffness matrix;

`L` = length units, and, length of beam element;

`LD` = destination array;

`M` = mass units;

`m` = number of matrix rows/columns;

`n` = number of matrix rows/columns;

`pc` = stack machine program counter;

`q` = physical quantity;

`r` = number of matrix rows/columns;

`rad` = radians;

`SI` = International system of units;

`SI_US` = Units common to SI and US systems of units;

`stiff` = element stiffness matrix;

`T` = temperature units;

`t` = time units, and, physical quantity;

`US` = English system of units, used in United States;

`X` = matrix;

`x` = physical quantity, and, an element of matrix X;

`Y` = matrix;

`y` = physical quantity, and, an element of matrix Y;

`Z` = matrix;

`z` = element of matrix Z.

Superscripts:

α = exponent for length units;

β = exponent for mass units;

γ = exponent for time units;

δ = exponent for temperature units;

ϵ = exponent for radian units.

Subscripts:

`i,j,k` = matrix row and column indices, and, indices of units in matrix row and column units buffers;

`1,2,3,4` = matrix row and column, and physical quantity indices.