ASME Trans., Journal of Mechanica Design, Vol. 117, No. 3, Sept. 1995, pp. 390-395

Extending C and FORTRAN for Design Automation

Dr. Harry H. Cheng, Assistant Professor
Director of Integration Engineering Laboratory
Department of Mechanical & Aeronautical Engineering
and Graduate Group in Computer Science
University of California
Davis, CA 95616
Email: hhcheng@ucdavis.edu
World Wide Web: http://iel.ucdavis.edu

Abstract

The C^H programming language is designed to be a superset of C. C^H bridges the gap between C and FORTRAN; it encompasses all the programming capabilities of FORTRAN 77 and consists of features of many other programming languages and software packages. Unlike other general-purpose programming languages, C^H is designed to be especially suitable for applications in mechanical systems engineering. Because of our research interests, many programming features in C^H have been implemented for design automation, although they are useful in other applications as well. In this paper we will describe these new programming features for design automation, as they are currently implemented in C^H in comparison with C and FORTRAN 77.

1 INTRODUCTION

Computers have brought the revolutionary changes to design automation. Numerous design problems that used to be intractable have been solved using powerful computers. Computers were programmed in the machine code in their earliest days. Machine code consists of highly inscrutable sequences of 0's and 1's. The programming paradigm of the assembly language is at a significantly higher level than that of machine code. Unlike machine code, with terms such as 0011000011011100, assemblers allow users to program a computer with more meaningful symbols, such as ADD and MOVE, as the processor's built-in machine instructions in a one-to-one symbolic mnemonic form. Like machine code, an assembly language program offers precise control over the way the processor will carry out the instruction. Although an assembly language program can be very compact and efficient, it is machine-dependent, tedious, hard to read, and difficult to change.

To make the computer accessible to users other than computer specialists, high-level programming languages have been developed. The source program of high-level programming languages uses readable English-like syntax so that users can concentrate on the problem rather than on the underlying machine architecture of the computer. All currently existing general-purpose high-level programming languages were not originally designed for applications in mechanical systems engineering. For example, FORTRAN was the first general-purpose high-level computer programming language ever developed. It was invented for FORmula TRANslation in the late 1950s by John Bakus at IBM. Real and complex formulas, but not dual formulas, can be conveniently translated into FORTRAN programs. Although some people think that it has lived considerably beyond its

time, FORTRAN 77 (ANSI, 1978) is still the primary language used by the professionals in mechanical engineering both in academia and industry. The second most popular language used by the professionals in mechanical engineering is the C programming language. Like other languages, C was not designed for applications in mechanical system engineering. The C programming language was invented by Dennis M. Ritchie at AT&T in the 1970s for Unix system programming (Kernighan and Ritchie, 1978). C is commonly considered as a *mid-level* language, and it can provide precise control over machines in the same manner as assembly languages. The user-base and its popularity of C are continuously growing. Compared with FORTRAN 77, C contains modern data structures, a wide variety of data types, modern control structures, dynamic memory allocation capabilities, capability of communication with the program's environment, and more operators. Although these desirable linguistic features provided in C are not present in FORTRAN 77, there exist serious deficiencies in C for scientific programming. Some deficiencies for numerical computing in the original K&R C (Kernighan and Ritchie, 1978) have been resolved in the ISO C standard (ISO, 1990). For example, ISO C honored parentheses, defined additional floating-point arithmetic operations so that floating-point operations for floats will be performed in float instead of in double, defined the standard mathematical functions in the library header `math.h`, and provided the characteristics of different floating-point data types in the library header `float.h`. These enhanced features make ISO C more desirable for numerical computing than K&R C. However, many severe deficiencies still remain in ISO C comparing with FORTRAN. For example, there are no intrinsic functions and complex data type in C; passing multi-dimensional arrays to functions is cumbersome.

Since the development of FORTRAN 77, computer software has undergone revolutionary changes, although they are not as dramatic as those of computer hardware. Many modern programming features of C and other programming languages have been incorporated into a new standard Fortran 90 (ISO, 1991). In light of the ISO C standard, the single most important enhancement of Fortran 90 is to allow the programmer to manipulate array objects. Many other features introduced in Fortran 90 are familiar to C programmers. For example, major additional features in Fortran 90, such as control structures, data structures, pointers, user-defined types, recursive procedures, and additional I/O functions were available in C at its earliest days. Because these modern features are being added more than 30 years after its original design, the syntax and semantics of Fortran 90 become quite complicated in comparison with FORTRAN 77 and C.

Both C and FORTRAN contain features that do not represent good programming practice. FORTRAN is based on the state-of-the-art of science and technology in the 1950s whereas C reflects the needs of system programmers. They are not best suitable for applications in design automation. Researchers have investigated the programming requirements for design automation. For example, a design process description language currently under development by Shah et al. (1993) is based upon Prolog. They have investigated the high-level representation of design processes and extensions of vocabulary to the base language. Neville and Joskowicz (1993) are developing a representation language for automating mechanism design. In general, developing a brand new language starting from scratch for design automation may not be justified economically. There is a tremendous investment in both C and FORTRAN programs as well as in training programmers and perfecting their skills in programming these two languages. Therefore, to narrow the historical gap, we will merge these two most popular languages, C and FORTRAN, into a single programming language — the C^H programming language. C^H is designed to be a superset of ISO C with incorporation of all the programming features of Fortran 90. The C^H programming language has been enhanced for design automation. It has been used as a teaching and learning tool at UC Davis for an undergraduate course, Computer-Aided Mechanism Design (Cheng, 1994a). In this paper we will describe the new programming features that are desirable for design automation, as they are currently implemented in C^H, in comparison with C and FORTRAN 77.

2 PROGRAMMING IN THE ENTIRE REAL DOMAIN

The IEEE 754 standard (IEEE, 1985) for binary floating-point arithmetic is a significant milestone on the road to consistent floating-point arithmetic with respect to real numbers. This standard has significantly influenced the design of C^H. The IEEE 754 standard distinguishes $+0.0$ from -0.0, which introduces an extra complexity for programming. The rationale for this extra complexity is not well understood and accepted by all computer scientists. Many have challenged the necessity for the sign of zeros. Apparently, how to best handle "the sign of nothing" is still a topic to be further investigated. Another important feature of the IEEE 754 standard is the internal representations for the mathematical infinities and invalid value. The mathematical infinity ∞ is represented by the symbol of Inf. A mathematically indeterminate or an undefined value such as division of zero by zero is represented by NaN, which stands for Not-a-Number. Many computer hardware support signed zeros, infinity, and NaN. Information about low-level and limited high-level instruction sets provided by hardware vendors may not be relevant to the application programmer and most features of a final system depend on the software implementation. Even for IEEE machines, if there is no provision for propagating the sign of zeros, infinity, and NaN in a consistent and useful manner through the software support, they will have to be programmed as if zeros were unsigned, without infinity and NaN. Based on IEEE machines, some vendors provide software support for the IEEE 754 standard through libraries. However, these special values in libraries are not transparent to the programmer. Although the application of symbols such as Inf and NaN can be found in some software packages, their handling of these special numbers is often full of flaws. For example, one can find ComplexInfinity in the software package Mathematica, and Inf and NaN in MATLAB. In Mathematica, there is no distinction between complex infinity and real infinities, nor between -0.0 and 0.0; therefore, many operations defined in C^H cannot be achieved in this package. In MATLAB, there is no complex infinity, and one will be surprised by some of its results. At one point, the sign of a zero is honored, but at an other point, it may not be. For example, **acosh**(Inf) equals NaN whereas **acos**(Inf) is a complex NaN. Results of mathematical functions in many cases are not consistent with mathematical conventions. It is in these grey areas that the IEEE 754 standard is not supported in many implementations of hardware and software systems.

To make the power of the IEEE 754 standard easily available to the programmer, the floating-point numbers of -0.0, 0.0, Inf, $-$Inf, and NaN, referred to as *metanumbers*, are introduced in C^H (Cheng, 1993a). These metanumbers are transparent to the programmer. Signed zeros $+0.0$ and -0.0 in C^H behave like correctly signed infinitesimal quantities 0_+ and 0_-, whereas symbols Inf and $-$Inf correspond to mathematical infinities ∞ and $-\infty$, respectively. The manipulation capabilities of Inf and NaN in C^H go far beyond the scope used in mathematical software packages such as Mathematica and MATLAB. The IEEE 754 standard only addresses the arithmetic involving these metanumbers. These metanumbers are extended in C^H consistently to commonly used mathematical functions in the spirit of the IEEE 754 standard. There are provisions for consistent handling of metanumbers in I/O, arithmetic, relational and logic operations, and polymorphic mathematical functions in C^H (Cheng, 1993a).

The metanumbers $0.0, -0.0$, Inf, $-$Inf, and NaN are very useful for design automation. For example, the discontinuity at the origin can be expressed using signed zeros. The infinity of mechanical advantage at a toggling position for a four-bar linkage can be written as Inf. If there exists no solution for output link corresponding to a given input link position of a four-bar linkage, the solution can be represented symbolically as NaN (Cheng, 1994a).

3 PROGRAMMING IN THE EXTENDED FINITE COMPLEX PLANE

Complex number, an extension of real number, has wide applications in science and engineering. It is very useful for analysis and design of planar mechanisms (Erdman and Sandor, 1991). Owing to its importance in scientific programming, numerically oriented programming languages and software packages usually provide complex number support in one way or another. For example, FORTRAN has provided complex data type since its earliest days. C does not have complex as a basic data type because numerically oriented scientific computing was not its original design goal. Computations involving complex numbers can be introduced as a data structure in C. However, programming with such a structure is somewhat clumsy, because for each operation a corresponding function has to be invoked. Using C++ (Stroustrup, 1987), a class for complex numbers can be created. By using operator and function overloadings, built-in operators and functions can be extended so as to also apply to the complex data type. Therefore, the complex data type can be achieved. However, it may be too involved for novice users to create such a class.

The reason for providing complex as a basic data type is not only for programming convenience, but also for design considerations. Design considerations such as implicit data type conversion, handling of metanumbers, computational complex arrays, lvalues, and optional arguments in multiple-valued complex functions used as branch indicators are difficult to implement at a user's program level even for a language like C++ with operator and function overloading capabilities. They should best be handled as language primitives. Most textbooks avoid issues related to the complex infinity. Likewise, none of the currently existing general-purpose computer programming languages has provisions for consistent handling of complex infinity. C^H provides real metanumbers of 0.0, -0.0, Inf, $-$Inf, and NaN, which makes the power of the IEEE 754 standard for binary floating-point arithmetic easily available to the programmer. C^H extends the idea of metanumbers to complex numbers not only for arithmetic, but also for commonly used mathematical functions in the spirit of the IEEE 754 standard and C (Cheng, 1993b). Because of the finite representation of floating-point numbers, complex numbers in C^H are programmed in *the extended finite complex plane*. Any *complex* or *float complex* values inside the ranges of $|x| <$ FLT_MAX and $|y| <$ FLT_MAX are representable in finite floating-point numbers. Variable x is used to represent the real part of a complex number and y the imaginary part; FLT_MAX, a predefined system constant, is the maximum representable finite floating-point number in the float data type. For *double complex*, the boundary for the extended finite complex plane will be expanded from FLT_MAX to DBL_MAX. A variable of double complex occupies 16-bytes; real and imaginary part each takes 8-bytes of double data type. A variable of *long double complex* is treated as if it was a double complex. Outside the extended finite complex plane, a complex number is treated as a Complex-Infinity represented as ComplexInf or complex(Inf,Inf) in C^H. The origin of the extended finite complex plane is complex(0.0, 0.0) or ComplexZero, which stands for Complex-Zero. In C^H, an undefined or a mathematically indeterminate complex number is denoted as complex(NaN, NaN) or ComplexNaN, which stands for Complex-Not-a-Number. The special complex numbers of ComplexZero, ComplexInf, and ComplexNaN are referred to as *complex metanumbers*. Because of the mathematical infinities of $\pm\infty$, it becomes necessary to distinguish a positive zero 0.0 from a negative zero -0.0 for real numbers. Unlike the real line, along which real numbers can approach the origin through the positive or negative numbers, the origin of the complex plane can be reached in any direction in terms of the limit value of $\lim_{r\to 0} re^{i\theta}$ where r is the modulus and θ is the phase of a complex number. Therefore, complex operations and complex functions in C^H do not distinguish 0.0 from -0.0 for real and imaginary parts of complex numbers. It is believed that C^H is the first general-purpose computer programming language that has provisions for consistent handling of signed zeros, signed infinities, and unsigned NaN in real numbers, and it also has consistent

handling of complex zero, complex infinity, and complex NaN as well as multiple-valued complex functions with optional arguments. Applications of complex numbers in C^{H} for planar mechanism design are presented by Cheng (1994a).

4 PROGRAMMING IN THE EXTENDED FINITE DUAL PLANE

Clifford (1873) introduced dual numbers in the form of $x + \varepsilon y$ with $\varepsilon^2 = 0$ to form bi-quaternions (called dual quaternions nowadays) for studying the non-Euclidean geometry. *Dual numbers* $d \in \boldsymbol{\mathcal{D}} = \{(x, y) | x, y \in \boldsymbol{\mathcal{R}}\}$ can be defined as ordered pairs

$$d = (x, y) \tag{1}$$

with specific addition and multiplication rules. The real numbers x and y are called the *real* and *dual parts* of d, respectively. If both x and y of a dual number are not zero, it is called a *proper dual* number. If we identify the pair of $(x, 0)$ as real numbers, the real number $\boldsymbol{\mathcal{R}}$ is a subset of $\boldsymbol{\mathcal{D}}$; that is, $\boldsymbol{\mathcal{R}} = \{(x, y) | x \in \boldsymbol{\mathcal{R}}, y = 0\}$ and $\boldsymbol{\mathcal{R}} \subset \boldsymbol{\mathcal{D}}$. If a real number is considered either as x or $(x, 0)$ and let ε denote the *pure dual* number $(0, 1)$ with the property of $\varepsilon^2 = 0$, dual numbers can be written as

$$d = x + \varepsilon y \tag{2}$$

Dual numbers are useful for analytical treatment in kinematics and dynamics of spatial mechanisms (Yang and Freudenstein, 1964; Ravani and Roth, 1984; Pennock and Yang, 1985; Dooley and McCarthy, 1991). Succinct formulas and equations can be derived by using dual numbers. However, the concise dual formulas cannot be conveniently implemented in a computer program using commonly used computer programming languages. The conciseness of a symbolic formula diminishes when numerical computations are needed. Therefore, the application of dual numbers in the study of problems in science and engineering has not yet been fully appreciated.

For the similar reasons of complex, dual is also implemented as a built-in data type, one of the primitives of the C^{H} language (Cheng, 1994b). Dual numbers and dual metanumbers are handled in the same manner as complex ones. For example, data types of *float dual* or *dual*, *double dual*, and *long double dual* are available. C^{H} is the first computer programming language, to the best of our knowledge, that provides dual as a basic data type. Formulas formulated in dual numbers can be translated into C^{H} programs as easily as those expressed in real or complex numbers (Cheng and Thompson, 1995).

C^{H} is a loosely typed language. All arguments of calling functions will be checked for compatibility with the data types of the called functions. The order of the data types in C^{H} is in the following sequence: char, int, float, double, complex, and dual. Char is the lowest data type and dual the highest. The data types of operands for an operation will also be checked for compatibility. However, unlike languages such as Pascal; which prohibits automatic type conversion, some data type conversion rules have been built into C^{H} so that they can be invoked whenever necessary. This will save many type conversion statements for a program.

5 POLYMORPHISM

A computer language with no mathematical functions is not suitable for design automation. Unlike FORTRAN, in which commonly used mathematical functions are handled as intrinsic functions, C is a small language — it does not provide mathematical functions internally. All functions in C are external. Because C does not provide mathematical functions internally, like arithmetical

operations in K&R C, the return value from a standard mathematical function is a double floating-point number regardless of the data types of the input arguments. In some of C implementations, if the input arguments are not doubles, the mathematical functions may return erroneous results without warning. The ISO C mathematical standard library does not provide any float functions. If a different return data type is desired for a mathematical function, a new function with a different name will be needed. For example, functions `sinf`(), **sin**(), and `sinl`() are reserved for float, double, and long double sine functions, respectively. The expression **sin**(1) appears correct in C. Indeed, most C programs will execute this operation calmly, but maybe with an erroneous result because the input data type of integer does not match with the data type for the argument of the **sin**() function. Possible proliferations of the sine function could be `csinf`(), `csin`(), and `csinl`() for float, double, and long double complex; `dsinf`(), `dsin`(), and `dsinl`() for float, double, and long double dual, respectively. As a result, one has to remember many arcane names for different functions.

The external functions of C^H can be created in the same manner as in C. Unlike C, however, the commonly used mathematical functions are built internally into C^H. The mathematical functions in C^H can handle different data types of the arguments gracefully in the same manner as the intrinsic functions in FORTRAN. The output data type of a function depends on the data types of the input arguments, which is called *polymorphism*. Like arithmetic operators, the built-in commonly used mathematical functions in C^H are polymorphic. For example, the intrinsic function **pow**(x,y) is defined as x^y. The variables x and y can be char, int, float, double, complex, or dual data types. If both x and y are integral values, the return data type is an integer value which is obtained using a more efficient algorithm than for exponentiation of real numbers. In this case, function **pow**() is equivalent to the exponentiation operator $**$ in FORTRAN. Note that there is no integer exponentiation operator or integer exponentiation function in ISO C. If x or y is a complex number, function **pow**(x,y) returns a complex value polymorphically. Unique to C^H is its optional argument for multiple-valued complex functions. For example, function **pow**(x,y,i) will calculate the ith branch of the multiple-valued complex exponential function (Cheng, 1993b);

For portability, all mathematical functions included in the C header `math.h` have been implemented polymorphically in C^H. The names of built-in mathematical functions of C^H are based upon the C header `math.h`. However, one can change, add, or remove these functions and operators in C^H if necessary (Cheng, 1993a). These mathematical functions are C compatible. If the arguments of these functions have the data types of the corresponding C mathematical functions, there is no difference between the C and C^H functions from a user's point of view. Besides the polymorphic nature, the mathematical function in C^H is more powerful owing to its abilities to handle metanumbers.

6 REFERENCES

A program written in a procedural programming language such as FORTRAN or C is generally formed by a set of functions, which subsequently consist of many programming statements. Using functions, a large computing task can be broken into smaller ones; a user can develop application programs based on what others have done instead of starting from scratch. The performance and user friendly interface of functions are critical to a programming language. The user may not need to know details inside functions that were developed by others. But, to use the functions effectively, the user has to understand how to interface functions through their arguments and return values. In general, arguments can be passed to functions in one of two ways: *call-by-value* and *call-by-reference*. In the call-by-value model, when a function is called, the values of the actual parameters are copied into formal parameters local to the called function. When a formal parameter is used

as an lvalue, only the local copy of the parameter will be altered. If the user wants the called function to alter its actual parameters in the calling function, the addresses of the parameters must be passed to the called function explicitly. In the call-by-reference method, however, the address of an argument is copied into the formal parameter of a function. Inside the function, the address is used to access the actual argument used in the calling function. This means that when the formal parameter is used as an lvalue, the parameter will affect the variable used to call the function. FORTRAN uses the call-by-reference model, whereas C uses the call-by-value. When a FORTRAN subroutine or function is ported as a function in C, the formal arguments of the subroutine are generally treated as arguments of pointer type in the function of C. All variables of arguments inside a subroutine then have to be modified accordingly, which may degrade the clarity of the original algorithm and code readability. This is also a point where beginners of C who have prior FORTRAN experience get confused. C^H is designed to be a superset of C, but it encompasses all the programming capabilities of FORTRAN 77. To bridge the gap between C and FORTRAN, references in C^H are designed and implemented in the spirit of C, C++, and FORTRAN. For example, following is a valid C^H code:

```
int a = 5, b = 6;
void swap(int &x, &y) /* pass-by-reference */
{
  int temp;
  temp = x; x = y; y = temp;
}
swap(a, b);           /* swap a and b */
```

We have extended the linguistic features of references in C++ and FORTRAN for design automation. Variables of basic data type and variables of pointer type can be used as references. In C^H variables of different data types can be passed to arguments of functions by reference, which transcends a long-standing tradition of scientific programming in typed programming languages. Furthermore, references can be used as both arguments and local variables of nested and recursively nested functions. Functions in C^H can be called either by value or by reference. Many restrictions in FORTRAN 77 about call-by-reference are relaxed in C^H. For example, the data types of actual arguments of a function should be the same as those of the formal arguments of the function in FORTRAN. In C^H, the data types of actual arguments of functions can be different from those of the corresponding formal arguments of references in functions. In FORTRAN, when an argument of a subroutine is used as an lvalue inside a subroutine, the actual argument in the calling subroutine must be a variable. Unlike in FORTRAN, a reference variable in C^H can be used as an lvalue inside a function even if the actual argument is not an lvalue. More details about linguistic features of references in C^H and its applications in mechanism design can be found in (Cheng 1993c; 1994a).

7 NESTED FUNCTIONS

As described in the previous section, a program written in C or FORTRAN is formed by a set of functions, which subsequently consist of many programming statements. All functions, including the function `main()`, in C are at the same level; functions cannot be defined inside other functions. In other words, there are no internal procedures in C. It has been pointed out by the C inventor that "They have been forbidden not for any philosophic reason, but only to simplify the implementation" (Ritchie, et al., 1978). C^H extends C with nested functions. Functions in C^H can not only be recursive, but also be nested, which means that a function not only can call itself, but also can define other functions inside the function (Cheng, 1993c).

With nested functions, details of one functional module can be hidden from the other modules which do not need to know about them. Each module can be studied independently of others. Nested functions modularize a program, thus clarifying the whole program and easing the pain of making changes to modules written by others. This programming feature is very important for writing modular and resuable code. This feature has been proved very useful for developing teaching tool-boxes in assisting student learning of mechanism design (Cheng, 1994a).

8 COMPUTATIONAL ARRAYS

Arrays in C are intimately tied with pointers. Treating array variables as pointers in C is very elegant for system programming; it is one of C's major strengths. But numerically oriented scientific computing was not the original design goal of C, as is reflected in an assumption made by the C inventor that "C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers" (Kernighan and Ritchie, 1978). This assumption is true only when C is used for system programming, because multi-dimensional arrays are essential for design automation.

Elements of an array in C are stored row-wise so that the second subscript (column) varies faster than the first subscript (row) as elements are accessed in the storage order. The row-wise storage pattern for elements of arrays and each index of arrays with the lower bound of zero is used mainly because variables of array type are also treated as pointers in C. The storage for elements of an array in C is different from that in FORTRAN, which stores elements of an array column-wise. In order to facilitate the integration of C and FORTRAN, the best solution is to treat the array as a single object in a program.

Regular arrays in C^H are fully ISO C compatible. *Computational arrays* in C^H can be manipulated as single objects. Computational arrays can be specified by type qualifier *array*. The default keyword can be changed if there is a conflict with the existing C code. What distinguish the computational array in C^H from other languages is its simplicity and similarity to mathematical formulas. Not only computational arrays of real and complex numbers, but also computational dual arrays are available in C^H.

9 AN APPLICATION EXAMPLE

The nested functions, polymorphism, and computational arrays in C^H can be demonstrated by the computation of the forward kinematics of general six-degree-of-freedom robot manipulators. The forward kinematics refers to the process that computes the position and orientation of the end-effector of a robot manipulator when joint angles of the manipulator are given. For a six-degree-of-freedom robot kinematic chain, the body coordinate system $X_iY_iZ_iO_i$ is attached to body i which is the physical link between joints $i-1$ and i. The transformation matrix which transforms the position specified in the body coordinate system $X_{i+1}Y_{i+1}Z_{i+1}O_{i+1}$ to the body coordinate system $X_iY_iZ_i$ can be formulated using Denavit-Hartenberg notation as follows:

$$\mathbf{D}_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

where α_i is the twist angle between Z_i and Z_{i+1} measured about axis X_{i+1}, a_i the shortest distance between Z_i and Z_{i+1} measured along the common normal X_{i+1}, d_i the shortest distance between

X_i and X_{i+1} along Z_i, and θ_i the angle between X_i and X_{i+1} measured along Z_i. According to the above definition, body 0 is the base of the robot manipulator and the world inertial frame is $X_0Y_0Z_0O_0$. There are seven coordinate systems $X_0Y_0Z_0O_0$, ..., $X_6Y_6Z_6O_6$ for a six-degree-of-freedom robot manipulator, its forward kinematics then becomes the computation of products of transformation matrices as

$$\mathbf{T}_6 = \begin{bmatrix} \mathbf{R}_6 & \mathbf{p}_6 \\ \mathbf{O} & 1 \end{bmatrix} = \mathbf{D}_1\mathbf{D}_2\mathbf{D}_3\mathbf{D}_4\mathbf{D}_5\mathbf{D}_6 \tag{4}$$

where the 3x3 rotation matrix $\mathbf{R}_6$ gives the orientation of the end-effector and 3x1 vector $\mathbf{p}_6$ locates the tool center point of the end-effector represented in the base coordinate system $X_0Y_0Z_0O_0$; the coordinate transformation matrix $\mathbf{D}_i$ is defined in equation (3).

For illustrative purpose, we will use dual matrices to compute robot forward kinematics. Using dual angles

$$\hat{\theta}_i = \theta_i + \varepsilon d_i \tag{5}$$

and

$$\hat{\alpha}_i = \alpha_i + \varepsilon a_i, \tag{6}$$

the corresponding dual D-H transformation matrix can be derived by the successive screw transformations as follows (Cheng, 1994b)

$$\hat{\mathbf{D}}_i = \begin{bmatrix} \cos\hat{\theta}_i & -\sin\hat{\theta}_i\cos\hat{\alpha}_i & \sin\hat{\theta}_i\sin\hat{\alpha}_i \\ \sin\hat{\theta}_i & \cos\hat{\theta}_i\cos\hat{\alpha}_i & -\cos\hat{\theta}_i\sin\hat{\alpha}_i \\ 0 & \sin\hat{\alpha}_i & \cos\hat{\alpha}_i \end{bmatrix} \tag{7}$$

The computation of the forward kinematics can be performed using the dual D-H matrix as

$$\hat{\mathbf{T}}_6 = \hat{\mathbf{D}}_1\hat{\mathbf{D}}_2\hat{\mathbf{D}}_3\hat{\mathbf{D}}_4\hat{\mathbf{D}}_5\hat{\mathbf{D}}_6 \tag{8}$$

Acoording to the *theory of transference* (Hsai and Yang, 1981, Gupta, 1989; Martinez and Duffy, 1992) which establishes the relationship between the dual form and regular vector form of a transformation, the following equation can be obtained

$$\hat{\mathbf{T}}_6 = (\mathbf{I} + \varepsilon\mathbf{P_6})\mathbf{R_6} \tag{9}$$

where $\mathbf{I}$ is a 3x3 identity matrix, $\mathbf{R_6}$ is the rotational part of the transformation matrix $\mathbf{T}_6$ in equation (4), and $\mathbf{P_6}$ defined as

$$\mathbf{P_6} = \begin{bmatrix} 0 & -p_z & p_y \\ p_z & 0 & -p_x \\ -p_y & p_x & 0 \end{bmatrix} \tag{10}$$

is a skew-symmetric matrix consisting of components of the position vector $\mathbf{p}_6$ in equation (4). Given $\hat{\mathbf{T}}_6$, $\mathbf{R_6}$ is the real part of $\hat{\mathbf{T}}_6$, and $\mathbf{P}_6$ is the product of the dual part of $\hat{\mathbf{T}}_6$ and transpose of $\mathbf{R_6}$. $\mathbf{R_6}$ and $\mathbf{P}_6$ can be written symbolically as follows:

$$\mathbf{R_6} = \text{real}(\hat{\mathbf{T}}_6) = \acute{\mathbf{T}}_6 \tag{11}$$

$$\mathbf{P_6} = \text{dual}(\hat{\mathbf{T}}_6)\mathbf{R_6}^T = \grave{\mathbf{T}}_6\mathbf{R_6}^T \tag{12}$$

where the accent grave symbol over a variable "$\acute{}$" means to take the real part of the variable whereas the accent acute symbol "$\grave{}$" stands for the dual part of the variable. Equations (11) and

Program 1: FUNCTION forward_kinematics().

```
void forward_kinematics(array dual T6[3][3],
                        alpha[6], theta[6])
{
  void dual_DH(array dual D[3,3],dual alpha,theta)
  {
    D[0][0] = cos(theta);
    D[0][1] = -sin(theta)*cos(alpha);
    D[0][2] = sin(theta)*sin(alpha);
    D[1][0] = sin(theta);
    D[1][1] =  cos(theta)*cos(alpha);
    D[1][2] = -cos(theta)*sin(alpha);
    D[2][0] = 0.0;
    D[2][1] =  sin(alpha);
    D[2][2] = cos(alpha);
  }
  array dual D1[3][3], D2[3][3], D3[3][3],
             D4[3][3], D5[3][3], D6[3][3];
  /*  compute dual DH matrix for six links */
  dual_DH(D1, alpha[0], theta[0]);
  dual_DH(D2, alpha[1], theta[1]);
  dual_DH(D3, alpha[2], theta[2]);
  dual_DH(D4, alpha[3], theta[3]);
  dual_DH(D5, alpha[4], theta[4]);
  dual_DH(D6, alpha[5], theta[5]);
  /*  compute dual transformation matrix */
  T6 = D1*D2*D3*D4*D5*D6;
}
```

(12) can also be written in C$^{\mathrm{H}}$ notation as $\mathbf{R_6} = \mathbf{real}(\hat{\mathbf{T}}_6)$ and $\mathbf{P_6} = \mathbf{dual}(\hat{\mathbf{T}}_6) * \mathbf{transpose}(\mathbf{R_6})$, respectively.

The dual transformation matrix $\hat{\mathbf{T}}_6$ can be easily computed in C$^{\mathrm{H}}$ using the function `forward_kinematics()` shown in Program 1. In Program 1, the 3x3 computational dual array `T6[3][3]` corresponding to dual transformation matrix $\hat{\mathbf{T}}_6$ in equation (8) is the output of the function passed as an argument of the function; `theta[i]` and `alpha[i]` corresponding to $\hat{\theta}_i$ and $\hat{\alpha}_i$ in equations (5) and (6), respectively, are the input of the function. The dual D-H matrix $\hat{\mathbf{D}}_i$ in equation(7) is represented by computational dual array `Di[3][3]` in the program. Function `dual_DH()` computes the dual D-H matrix in equation (7). Inside the function, functions **sin**() and **cos**() will compute dual sine and cosine of input dual angles, respectively, because these are polymorphic mathematical functions. Because function `dual_DH()` is called only inside the function `forward_kinematics()`, it is defined as a nested function. The formula (8) is translated as `T6 = D1*D2*D3*D4*D5*D6` in the above C$^{\mathrm{H}}$ program, each computational dual array is treated as a single object.

For a Puma 560 robot manipulator, the screw parameters $\hat{\theta}_i$ and $\hat{\alpha}_i$ are given as follows:

Program 2: COMPUTATION OF FORWARD KINEMATICS FOR PUMA 560.

```
/* forward kinmatics for Puma 560*/
#include <linkage.h>
main()
{
  array dual T6[3,3], alpha[6], theta[6];
  array float R6[3,3], P6[3,3];
  float d2r = PI/180;
  /* DH parameters of dual alpha and theta */
  theta[0] = dual(10*d2r, 0.0);
  theta[1] = dual(20*d2r, 149.09);
  theta[2] = dual(30*d2r, 0.0);
  theta[3] = dual(40*d2r, 433.07);
  theta[4] = dual(50*d2r, 0.0);
  theta[5] = dual(60*d2r, 56.25);
  alpha[0] = dual(-90*d2r, 0.0);
  alpha[1] = dual(0.0, 431.8);
  alpha[2] = dual(90*d2r, -20.32);
  alpha[3] = dual(-90*d2r, 0.0);
  alpha[4] = dual(90*d2r, 0.0);
  alpha[5] = 0;
  forward_kinematics(T6, alpha, theta);
  R6 = real(T6);
  P6 = dual(T6)*transpose(R6);
  printf("The dual transformation matrix T6 is \n", T6);
  printf("The rotational matrix R6 is \n%10.3f", R6);
  printf("The skew-symmetric matrix for the position"
         " vector p6 is \n%10.3f", P6);
}
```

$$
\begin{array}{ll}
\hat{\theta}_1 = \theta_1 & \hat{\alpha}_1 = -\ 90 \text{ deg} \\
\hat{\theta}_2 = \theta_2 + \varepsilon 149.09 \text{mm} & \hat{\alpha}_2 = \varepsilon 431.8 \text{ mm} \\
\hat{\theta}_3 = \theta_3 & \hat{\alpha}_3 = 90 \text{ deg } -\varepsilon 20.32 \text{ mm} \\
\hat{\theta}_4 = \theta_4 + \varepsilon 433.07 \text{mm} & \hat{\alpha}_4 = -90 \text{ deg} \\
\hat{\theta}_5 = \theta_5 & \hat{\alpha}_5 = 90 \text{ deg} \\
\hat{\theta}_6 = \theta_6 + \varepsilon 56.25 \text{mm} & \hat{\alpha}_6 = 0
\end{array}
$$

Program 2 can compute the forward kinematics of a Puma 560 with joint values of (10,20,30,40,50,60) degrees.

In Program 2, polymorphic functions **real**() and **dual**() return the real and dual parts of dual transformation matrix $\hat{\mathbf{T}}_6$, respectively. The transpose of computational array $\mathbf{R_6}$ is obtained by the array function `transpose()`. The standard I/O function **printf**() in C has been enhanced to handle computational arrays in C^H. The following output for the dual transformation matrix $\hat{\mathbf{T}}_6$, orientation matrix $\mathbf{R_6}$, and skew-symmetric positional matrix $\mathbf{P_6}$ will be printed out when Program 2 is executed in a C^H programming environment.

The dual transformation matrix T6 is

dual(-0.637,-113.792) dual(0.023, 303.913) dual(0.771,-102.919)
dual(0.771, -85.681) dual(0.030,-727.131) dual(0.636, 137.743)
dual(-0.008, 759.981) dual(0.999, 14.626) dual(-0.036, 227.072)
The rotational matrix R6 is
-0.637 0.023 0.771
0.771 0.030 0.636
-0.008 0.999 -0.036
The skew-symmetric matrix for the position vector p6 is
0.000 -144.209 308.395
144.209 0.000 -730.916
-308.395 730.916 0.000

10 CONCLUSIONS

The C^H programming language bridges the gap between C and FORTRAN 77 with many extensions. It is designed to be especially suitable for applications in design automation, although programming features in C^H are useful for applications in many other areas as well. In this paper we have presented new programming features metanumbers, complex and dual types, polymorphism, references, nested functions, and computational arrays, as they are currently implemented in C^H in comparison with C and FORTRAN 77. These new programming features can be very useful for solving problems in design automation as shown in the application example.

11 ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation as a Research Initiation Award under Grant DMII-9309207, by the University of California, Davis through an undergraduate instructional improvement grant, a faculty research grant, and a junior faculty research fellowship, and by Motorola, Inc.

12 REFERENCES

1. ANSI, *ANSI Standard X3.9-1978, Programming Language FORTRAN*, (revision of ANSI X2.9-1966), American National Standards Institute, Inc., NY, 1978.
2. Cheng, H. H., Scientific Computing in the C^H Programming Language, *Scientific Programming*, vol. 2, No. 3, 1993a, pp. 49-75.
3. Cheng, H. H., Handling of Complex Numbers in the C^H Programming Language, *Scientific Programming*, vol. 2, No. 3, 1993b, pp. 79-106.
4. Cheng, H. H., Adding References and Nested Functions to C for Modular and Parallel Scientific Programming, NCEG, X3J11.1/93-044, October 22, 1993c.
5. Cheng, H. H., Pedagogically Effective Programming Environment for Teaching Mechanism Design, *Computer Applications in Engineering Education*, Vo. 2, No. 1, 1994a, pp. 23-39.
6. Cheng, H. H., Programming with Dual Numbers and its Applications in Mechanism Design, *Engineering with Computers*, Vol. 10, No. 4, 1994b, pp. 212-229.
7. Cheng, H. H., and Thompson, Computer-Aided Displacement Analysis of Spatial Mechanisms Using the C^H Programming Language, *Advances in Engineering Software*, 1995 (accepted).

8. Clifford, W. K., Preliminary Sketch of Bi-quaternions, *Proceedings of London Mathematical Society*, Vol. 4, 1873, pp. 381-395.
9. Dooley, J. R. and McCarthy, J. M., Spatial Rigid Body Dynamics Using Dual Quaternion Components, *Proceedings of IEEE International Conf. on Robotics and Automation*, vol. 1, Sacramento, CA, April 1991, pp.90-95.
10. Martinez, J., and Duffy, J., The Principle of Transference - History, Statement and Proof, *Mechanism and Machine Theory*, vol. 28, N0. 1, Jan. 1993, pp. 165-177.
11. Erdman, A. G., and Sandor, G. N, Mechanism Design, Analysis and Synthesis, vol. 1, 2nd edition, Prentice Hall, 1991.
12. Gupta, K, C., Spatial Kinematics, lecture notes, Department of Mechanical Engineering, University of Illinois at Chicago, June, 1989.
13. Hsia, L. M. and Yang, A. T., On the Principle of Transference in Three-Dimensional Kinematics, *Trans. ASME, J. of Mechanical Design*, Vol. 103, No.3, July 1981, pp. 652-656.
14. IEEE, *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1985.
15. ISO/IEC, *Information Technology, Programming Languages - FORTRAN*, 1539:1991E, ISO, Geneva, Switzerland.
16. ISO/IEC, *Programming Languages - C*, 9899:1990E, ISO, Geneva, Switzerland.
17. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1st edition, 1978; 2nd edition, 1988.
18. Neville, D., and Joskowicz, L., A Representation Language for Mechanical Behavior, *Proc. the 5th International Conf. on Design Theory and Methodology*, Albuquerque, NM, September 19-22, 1993, pp. 1-6.
19. Pennock, G. R. and Yang, A. T., Application of Dual-Number Matrices to the Inverse Kinematics Problem of Robot Manipulators, *Trans. ASME, J. of Mechanisms, Transmissions, and Automation in Design*, Vol.107, June 1985, pp. 201-208.
20. Ravani, B. and Roth B., Mappings of Spatial Kinematics, *Trans. ASME, J. of Mechanisms, Transmissions, and Automation in Design*, Vol.106, Sept. 1984, pp. 341-347.
21. Ritchie, D. M., Johnson S. C., Lesk, M. E., and Kernighan, B. W., The C Programming Language, *The Bell System Technical Journal*, vol. 57, No. 6, July-August 1978, pp. 1991-2020.
22. Shah, J. J., Bliznakov, P. I., and Urban, S. D., Development of a Machine Understandable Language for Design Process Representation. *Proc. the 5th International Conf. on Design Theory and Methodology*, Albuquerque, NM, September 19-22, 1993, pp. 15-24.
23. Stroustrup, B., *The C++ Programming Language*, Reading, MA, Addison-Wesley, 1987.
24. Yang, A. T. and Freudenstein, F., *Application of Dual-Number Quaternion Algebra to the Analysis of Spatial Mechanisms*, *Trans. ASME, J. of Applied Mechanics*, Vol. 86, No. 2, June 1964, pp. 300-308.