

# Programming with Dual Numbers and its Applications in Mechanisms Design

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](mailto:hhcheng@ucdavis.edu)  
World Wide Web: <http://iel.ucdavis.edu>

## Abstract

Dual numbers are in the form of  $x + \varepsilon y$  with  $\varepsilon^2 = 0$ . Dual metanumbers are defined in this paper as DualZero, DualInf, and DualNaN. The extended dual plane and extended finite dual plane are introduced in this paper to describe dual numbers and dual metanumbers. Handling of dual numbers in the  $C^H$  programming language is presented in this paper. The I/O, arithmetic and relational operations, and built-in mathematical functions are defined for both dual numbers and dual metanumbers. Due to polymorphism, the syntaxes of dual arithmetic and relational operations, and built-in dual functions are the same as those for real and complex numbers in the  $C^H$  programming language. The valid lvalues related to dual numbers in  $C^H$  are defined. The computation of the motion screw for a rigid body displacement is used as an example to illustrate the creation of user's dual functions. The efficacy of  $C^H$  programming with dual numbers is demonstrated by displacement analysis of a RCCC mechanism. For the first time, dual data is handled as a built-in data type in a general-purpose computer programming language. Programming with dual numbers in  $C^H$  is simpler than in any other computer programming languages.

## 1 Introduction

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 noneuclidean geometry. Study (1903) defined dual numbers as dual angles to specify the relation between two lines in the Euclidean space. McAulay (1898) used dual quaternions to describe finite displacement of rigid and deformable bodies. Dimentberg (1950) and Denavit (1958) pioneered kinematic analysis of spatial mechanisms via dual numbers. Using dual numbers, Yang and Freudenstein (1963, 1964) studied kinematic analysis of spatial mechanisms; and Ravani and Roth (1984) studied rigid body displacements via kinematic mapping. Recently, dual numbers have been applied to study the kinematics, dynamics, and calibration of open-chain robot manipulators (Pennock and Yang, 1985; 1983; Dooley and McCarthy, 1991; Ravani and Ge, 1991). Dual numbers are useful for analytical treatment in kinematics and dynamics of spatial mechanisms. 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 will diminish when numerical computations are needed. Therefore, the

application of dual numbers in the study of problems in science and engineering has not been fully appreciated yet.

In this paper, we will demonstrate that programming with dual numbers can be as simple as programming with real or complex numbers. FORTRAN was designed in the 1950s for real or complex FORMula TRANslation, but not for formulas with dual numbers. This language design reflects the state-of-the-art of computer science and technology in the 1950s and the perception of scientific computation at that time although many advances in the field have been incorporated into its standard in 1977 as FORTRAN77 (ANSI 1978). C, a modern computer language, has gained its wide acceptance as it was standardized recently (ANSI, 1989). C is most popular for system programming because it was originally designed for Unix system programming (Kernighan and Ritchie, 1978). The numerically-oriented scientific programming was not the original design goal; needless to say, the handling of dual numbers. Recently, we have designed and implemented  $C^H$  which is a general-purpose block-structured interpretive programming language. It retains most features of ANSI C, but with many High-level extensions.  $C^H$  combines features of many computer languages and software packages, and it is designed to be especially suitable for applications in the areas of scientific and system programmings as well as mechanical systems engineering applications. For example, high-level matrix computation features of MATLAB are available in  $C^H$  [27]. In most of its applications, the size of a program written in  $C^H$  is significantly smaller than that written in Fortran or C. Scientific  $C^H$  programmings with real and complex numbers have been reported in (Cheng, 1993a; 1993b). Real and complex data types in  $C^H$  are designed and implemented in the spirit of ANSI C, IEEE 754 standard for binary floating-point arithmetic (IEEE, 1985), and Fortran.  $C^H$ , I believe, is the first general-purpose computer programming language which has provisions for consistent handling of signed zeros, signed infinities, and unsigned NaN in real numbers; and consistent handling of complex zero, complex infinity, and complex NaN as well as multiple-valued complex functions with optional arguments. Handling of complex numbers in  $C^H$  and other proposals (Kahan, 1986; MacDonald, 1989; Kahan and Thomas, 1991; Knaak, 1992; Tydeman, 1992) are currently being considered by the Numerical C Extension Group (NCEG), subcommittee X3J11.1 of the ANSI C committee X3J11, as complex extension to C.

It should be pointed out that dual numbers in  $C^H$  is not a just simple application of C++ (Stroustrup, 1987). It is not implemented as a class of C++. Like float, double, and complex; dual is also implemented as a built-in data type, one of primitives of the  $C^H$  language. Many features described in this paper are not feasible using a dual class implmented using a C++. For example, I/O, implicit data type conversion, metanumbers, lvalues, and arrays related to dual numbers presented in this paper cannot be implmented in C++ at the user's level. As a result, dual numbers in  $C^H$  are much more convenient to use than in C++.  $C^H$  is the first computer programming language, I believe, that provides dual as a basic data type. Formulas formulated in dual numbers can be translated into  $C^H$  programs as easy as those expressed in real or complex numbers. Handling of dual numbers in the  $C^H$  programming language in the spirit of ANSI C, as it is currently implemented, will be described in this paper.

The rest of the paper is organized as follows. Section 2 describes the geometric interpretation of dual numbers by introducing the extended dual plane and extended finite dual plane. It explains how dual variables, dual arrays, and dual pointers are declared. It also describes the data conversion rules between real and complex numbers and dual numbers, as well as the I/O for dual numbers. Sections 3 and 4 define the dual operations and dual functions for dual numbers and dual metanumbers in the  $C^H$  language syntax, respectively. Section 5 gives the valid lvalues related to dual numbers. Section 6 computes the motion screw of a rigid body displacement, which illustrates how user's dual functions can be easily created in  $C^H$ . Section 7 demonstrates the efficacy of  $C^H$  programming with dual numbers by displacement analysis of a RCCC mechanism. Unless

explained otherwise, all code fragments presented in this paper follow the interpretation of the ANSI C standard.

## 2 Dual Numbers

### 2.1 Dual Numbers and Dual Constructor

Dual numbers  $d \in \mathcal{D} = \{(x, y) | x, y \in \mathcal{R}\}$  can be defined as ordered pairs

$$d = (x, y) \quad (1)$$

with specific addition and multiplication rules (Clifford, 1873; Guggenheimer, 1963; Bottema and Roth, 1979). 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  $\mathcal{R}$  is a subset of  $\mathcal{D}$ , i.e.,  $\mathcal{R} = \{(x, y) | x \in \mathcal{R}, y = 0\}$  and  $\mathcal{R} \subset \mathcal{D}$ . If a real number is considered either as  $x$  or  $(x, 0)$  and let  $\varepsilon$  denote the *pure dual* number  $(0, 1)$  with the property of  $\varepsilon^2 = 0$ , dual numbers can be written as

$$d = x + \varepsilon y \quad (2)$$

Resembling the notation (1), a dual number can be created in  $C^H$  by the *dual constructor* `dual(x, y)` with  $x, y \in \mathcal{R}$ . For example, a dual number with its real part of 1.2 and dual part of 3.4 can be constructed by `dual(1.2, 3.4)`. Internally, a dual number consists of two floats at the current implementation. Hence, arguments of the dual constructor `dual(x,y)` will always be cast to floats internally if they are not floats. As described in (Cheng, 1993a), all floating-point constants without suffix in  $C^H$  are floats by default. This default mode for floating-point constants can be switched from float to double by function `floatconst(FALSE)`. The double constants can be obtained by suffixing a floating-point constant with `D` or `d`. When double dual data type is implemented in the future, the dual constructor shall return dual or double dual polymorphically, depending on the data types of the input arguments. The polymorphic dual constructor `dual(x,y)` shall return a double dual if any argument of  $x$  or  $y$  is double. For example, `dual(3, 4.0d)`, `dual(3.0f, 4.0d)`, `dual(0.3e1D, 40e-1F)`, and `dual(3.0D, 4.0D)` shall return a double dual number of `dual(3.0D, 4.0D)`. The internal data representation of a dual number is similar to that of a complex number. For the convenience of presentation and programming, the second element of a dual number is, therefore, also referred to as *imaginary part* in  $C^H$ . The names of *dual part* and *imaginary part* are used interchangeably in this paper.

It should be emphasized that dual numbers are extension of real numbers. If the dual parts of dual numbers are identically zeros, results of dual operations and dual functions should be dual numbers with identical zero dual parts. This design goal has been achieved in  $C^H$ . For example, `sqrt(dual(9.0, 0.0))` equals `dual(3.0, 0.0)` and `pow(dual(2.0,0.0), dual(3.0,0.0))` becomes `dual(8.0, 0.0)`, all with identically zero dual parts. These dual functions will be discussed in details in section 4.

### 2.2 Dual Metanumbers and Geometric Interpretation

A dual number  $d = x + \varepsilon y$  can be associated with a point in the plane whose Cartesian coordinates are  $x$  and  $y$  as shown in Figure 1. Each dual number corresponds to just one point in the plane, and vice versa. The number  $d$  can also be thought of as the vector from the origin to the point  $(x, y)$ . When used for the purpose of displaying numbers  $d = x + \varepsilon y$  geometrically, we call the  $XY$  plane the *dual plane*, or *d plane*, the  $X$ -axis the *real axis*, and the  $Y$ -axis the *dual axis*.

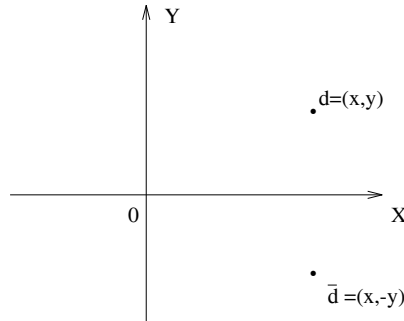


Figure 1: The dual number  $d$  and its conjugate  $\bar{d}$  in the dual plane.

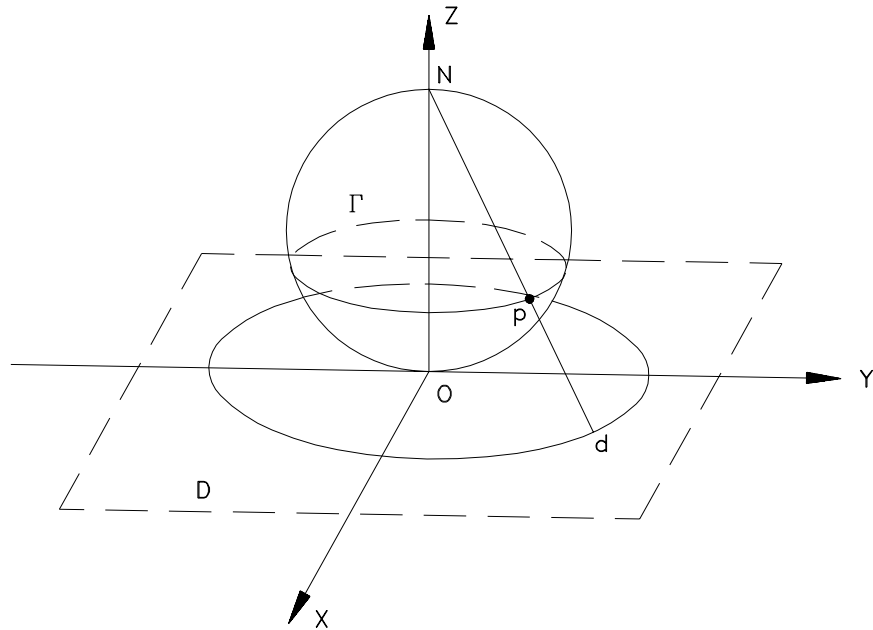
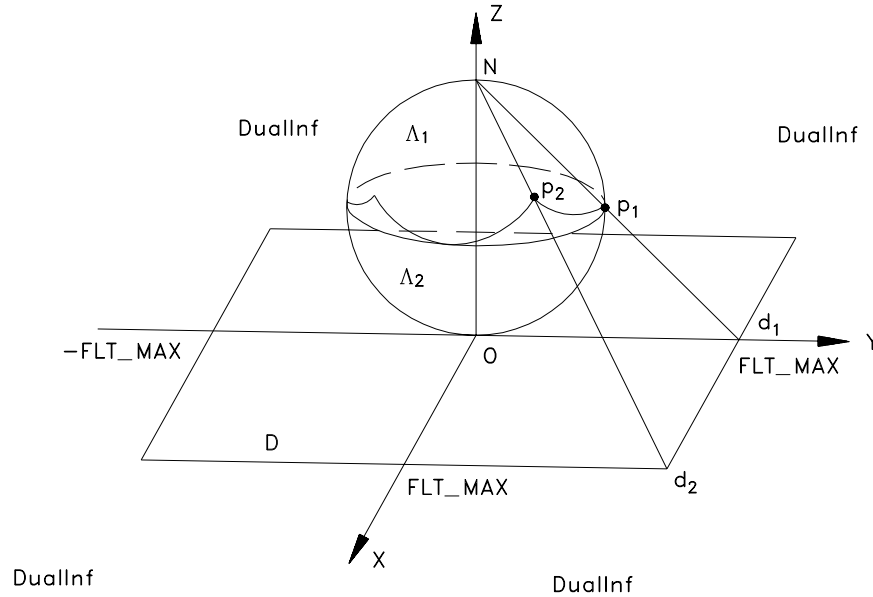


Figure 2: The extended dual plane and unit sphere  $\Gamma$ .

We define the *dual conjugate* of the dual number  $d = x + \varepsilon y$  as the dual number  $x - \varepsilon y$ , denoted as  $\bar{d}$ . As shown in Figure 1, the dual number  $\bar{d}$  represented by the point  $(x, -y)$  in the dual plane is the reflection in the real axis of the point  $(x, y)$  representing  $d$ .

As described in (Cheng, 1993a),  $C^H$  provides real metanumbers of 0.0,  $-0.0$ , Inf,  $-\text{Inf}$ , and NaN for scientific computing. Because of metanumbers of Inf and  $-\text{Inf}$ ,  $C^H$  distinguishes 0.0 from  $-0.0$  for real numbers in arithmetic operations and elementary mathematical functions. For proper handling of complex numbers, Cheng (1993b) introduced complex metanumbers of ComplexZero, ComplexInf, and ComplexNaN along with the extended finite complex plane. In this paper, we introduce the *extended dual plane* shown in Figure 2. The extended dual plane is the dual plane plus the point at infinity denoted as DualInf. In Figure 2, there is an one-to-one correspondence between the points on the unit sphere  $\Gamma$  and the points in the extended dual plane  $\mathcal{D}$ . The point  $p$  on the surface of the sphere is determined by the intersection of the line through the point  $d$  in the extended dual plane and the north pole  $N$  of the sphere. There is only *one* dual infinity in the extended dual plane, the north pole  $N$  corresponds to the point at infinity. Due to the finite representation of floating-point numbers, we also introduce the *extended finite dual plane* shown in Figure 3. Any dual values inside the ranges of  $|x| < \text{FLT\_MAX}$  and  $|y| < \text{FLT\_MAX}$  in the extended finite dual

Figure 3: The extended finite dual plane and unit sphere  $\Lambda$ .

plane are representable in finite floating-point numbers. FLT\_MAX, a predefined system constant, is the maximum representable finite floating-point number in the float data type. Outside this rectangular area of the extended finite dual plane, a dual number is treated as a Dual-Infinity represented as DualInf or dual(Inf,Inf) in  $C^H$ . The one-to-one correspondence between points on the unit sphere  $\Gamma$  and extended dual plane is not valid for the unit sphere  $\Lambda$  and extended finite dual plane. All points on the surface of the upper part  $\Lambda_1$  of the unit sphere correspond to the dual infinity. Points on the lower part  $\Lambda_2$  of the sphere and points in the extended finite dual plane are in one-to-one correspondence. The boundary between surfaces  $\Lambda_1$  and  $\Lambda_2$  corresponds to the threshold of overflow. For example, points  $p_1$  and  $p_2$  on the unit sphere  $\Lambda$  correspond to points  $d_1 = \text{dual}(\text{FLT\_MAX}, 0.0)$  and  $d_2 = \text{dual}(\text{FLT\_MAX}, \text{FLT\_MAX})$  in the extended finite dual plane shown in Figure 3, respectively. The origin of the extended finite dual plane is dual(0.0, 0.0) or DualZero which stands for Dual-Zero. It should be pointed out that, by defining the dual unit sphere as  $\mathbf{d} \cdot \mathbf{d} = 1$  where  $\mathbf{d} = \mathbf{a} + \varepsilon \mathbf{a}_0$  is a dual unit vector with  $|\mathbf{a}| = 1$  and  $\mathbf{a} \perp \mathbf{a}_0$ , Study (1903) established the one-to-one correspondence between the oriented lines in space and the points of the dual unit sphere (Guggenheimer, 1963). However, that one-to-one correspondence is different from the one-to-one correspondence between points on the unit sphere and the extended dual plane presented in this paper.

Like a real number which can be considered as an angle, in differential geometry and motion analysis of spatial mechanisms (Study 1903; Guggenheimer, 1963; Yang and Freudenstein, 1964), a dual number is also commonly referred to as a *dual angle*

$$\hat{\theta} = \theta + \varepsilon s \quad (3)$$

between two lines  $L_1$  and  $L_2$  in space as is shown in Figure 4. The real part  $\theta$  of the dual angle is the projected angle between lines  $L_1$  and  $L_2$ , and the dual part  $s$  is the length along the common normal of two lines. In general, the dual angle between two non-parallel or non-intersecting lines in space is a proper dual number. The dual angle subtended by two intersecting lines is a real number. The dual angle between two parallel lines is a pure dual number. A dual angle can also be represented by

$$d = \hat{\theta} = \theta(1 + \varepsilon p) \quad (4)$$

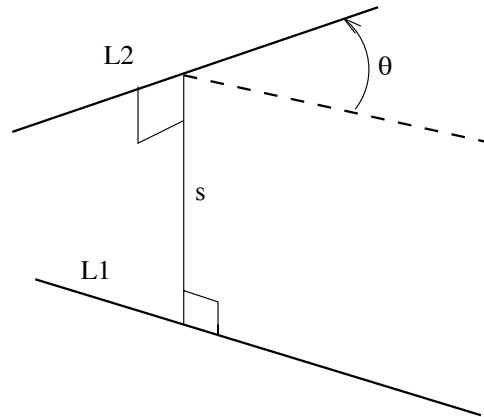


Figure 4: The dual angle  $\hat{\theta} = \theta + \varepsilon s$  between lines  $L_1$  and  $L_2$ .

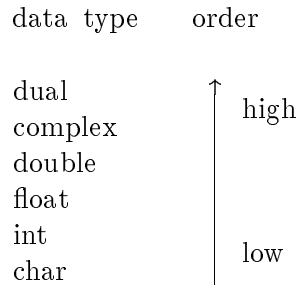
where the ratio  $p = s/\theta$  is referred to as the *pitch* of the dual angle  $\hat{\theta}$ . If the pitch  $p$  is zero, it is a pure rotation; if  $p$  is infinity, it is a pure translation. The values of  $\pm\infty$  are represented by real metanumbers of  $\pm\text{Inf}$  in  $C^H$  (Cheng, 1993a).

In  $C^H$ , an undefined or mathematically indeterminate dual number is denoted as `DualNaN` or `dual(NaN, NaN)` which stands for Dual-Not-a-Number. To ensure the proper flow of the program, all operations and functions are defined over the entire domain of the extended finite dual plane. Furthermore, To guarantee the delivery of correct numerical results, a built-in dual operation or dual function in  $C^H$  will deliver `DualNaN` if it is mathematically indeterminate, e.g., `sqrt(dual(-3.0, 0.0))` is `DualNaN`. We call special dual numbers of `DualZero`, `DualInf`, and `DualNaN` as *dual metanumbers*. Because of the mathematical infinities of  $\pm\infty$ , it becomes necessary to distinguish the positive zero of `0.0` from the negative zero of `-0.0` for real numbers (Cheng, 1993a). Unlike the real line along which real numbers can approach the origin only in the positive or negative numbers, the origin of the dual plane can be reached in any directions. Therefore, dual operations and dual functions in  $C^H$  do not distinguish `0.0` from `-0.0` for real and dual parts of a dual number. Due to these differences, some operations and functions need to be handled differently for real and dual numbers, especially for real metanumbers and dual metanumbers. For example, the addition of two real numbers of positive infinities is a value of infinity in  $C^H$  (Cheng, 1993a). But, the addition of two dual infinities is indeterminate although the value of `DualInf` is represented internally as two positive infinities of `Infs`.

### 2.3 Dual Variables

Declarations of *dual variables* in  $C^H$  are similar to declarations of any other data types in C. One can declare not only a simple dual variable, but also *pointer to dual*, *array of dual*, and *array of pointer to dual*, etc.. The array and pointer of dual in  $C^H$  are manipulated in the same manner as the floating-point float and double. The following code fragment illustrates how dual variables are declared and manipulated in  $C^H$ .

```
dual d1;          /* declare d1 as dual variable */
dual *dptr1;
  /* declare dptr1 as pointer to dual variable */
dual d2[2];      /* declare d2 as an array of dual
dual *dptr2[2][4];
  /* declare dptr2 as an array of pointer to dual
```

Figure 5: The order of basic data types in C<sup>H</sup>.

```

dptr1 = &d1;    /* dptr1 point to the address of d1
*dptr1 = dual(1,2);
    /* d1 with real part of 1.0 and dual part of 2.0

```

Like C, comments in a C<sup>H</sup> program are enclosed in a pair of delimiter signs of /\* and \*/. In addition, the symbol /\* in C<sup>H</sup> will comment out a subsequent text terminated at the end of a line.

## 2.4 Dual Functions

A dual function  $f(d) = f(x + \varepsilon y)$  with a dual variable can be obtained by expanding it formally in a Taylor series. Because  $\varepsilon^n = 0$  for  $n > 1$ , then,

$$f(d) = f(x + \varepsilon y) = f(x) + \varepsilon y f'(x) \quad (5)$$

where  $f'(x)$  in the dual part is the derivative of function  $f(x)$ . Similarly, a dual function with two dual variables can be obtained as

$$\begin{aligned} f(d_1, d_2) &= f(x_1 + \varepsilon y_1, x_2 + \varepsilon y_2) \\ &= f(x_1, x_2) + \varepsilon(y_1 f'_{d_1}(x_1, x_2) + y_2 f'_{d_2}(x_1, x_2)) \end{aligned} \quad (6)$$

where  $f'_{d_1}(x_1, x_2)$  is the partial derivative with respect to the first variable  $d_1$  and  $f'_{d_2}(x_1, x_2)$  with respect to the second. Using these definitions, all identities for ordinary elementary functions hold true for dual functions with dual variables as well. For example,  $\sin^2 d + \cos^2 d = 1$ ;  $\cosh^2 d - \sinh^2 d = 1$ ,  $d_1^{d_2} = \exp(d_2 \log(d_1))$ . The built-in dual functions in C<sup>H</sup> will be described in details in section 4.

## 2.5 Conversion between Dual and Other Basic Data Types

C<sup>H</sup> is a loosely typed language. All arguments of calling functions will be checked for compatibility with the data types of the called functions. The data types of operands for an operation will also be checked for compatibility. If data types do not match, the system will signal an error and print out some informative messages for the convenience of program debugging. However, unlike languages such as Pascal (ANSI, 1983) which prohibits automatic type conversion, some data type conversion rules have been built in C<sup>H</sup> so that they can be invoked whenever necessary. This will save many type conversion statements for a program. The order of the data types in C<sup>H</sup> is shown in Figure 5. Char is the lowest data type and dual the highest. The default conversion rules are briefly discussed in this section as follows:

- Char, int, float, and double can be converted according to ANSI C conversion rules. The ASCII value of a character will be used in conversion for a char data type. Demotion of data may lose the information.
- Char, int, float, and double can be promoted to complex or dual with its imaginary part being zero. When casting a real number into a complex or dual number, the values of Inf and -Inf become DualInf; and the value of NaN becomes DualNaN. Conversion from double to complex or dual may lose the information. A real number can be cast into a complex by the complex construction function **complex**(x,y). Likewise, a real number can also be cast into a dual explicitly by the dual construction function **dual**(x,y), which will be discussed in details in section 4.
- A complex or dual number can be converted to a real number implicitly in the assignment statement and function interface. When a complex or dual is converted to char, int, float, or double, if the imaginary part of the complex or dual number is identically zero, its real part is used and the imaginary part will be discarded. If the imaginary part of the complex or dual number is not identically zero, the converted real number becomes NaN. The metanumbers of ComplexInf, ComplexNaN, DualInf, and DualNaN are converted as NaNs in real number. The real and imaginary components of a dual number can be obtained explicitly by the functions **real**(d) and **imaginary**(d), respectively. When a dual number is converted to a real number either implicitly by the assignment statement and function interface or explicitly by **real**(d), **imaginary**(d), **float**(d), **double**(d), **(float)**d, and **(double)**d; the sign of a zero will not be carried over. If the imaginary part of the complex or dual number is identically zero, converting a dual number to an integral value, such as char and int, is equivalent to the conversion of **real**(d) to an integral value. For example, `i = DualInf` will make i equal to INT\_MAX. However, if **real**() or **imaginary**() is used as a lvalue, the sign of zeros from rvalue will be preserved. A *lvalue* is any object that occurs on the left side of an assignment statement. The lvalue refers to a memory such as a variable or pointer, not a function or constant. On the other hand, the *rvalue* refers to the value of the expression on the right hand side of an assignment statement. Details about the lvalue related to dual will be discussed in section 5.
- When a complex number is converted to a dual, the real and imaginary parts of the complex become the real and dual parts of the dual number, respectively. Likewise, the real and dual parts of a dual number will become the corresponding real and imaginary parts of a complex number when the dual number is converted into a complex.
- In binary operations such as addition, subtraction, multiplication, and division, etc. with mixed data types, the result of the operation will carry the higher data type of two operands. For example, the result of addition of an int and a double will result in a double. When one of the two binary operands is dual and the data type of the other operand is a real or complex number, the real or complex number will be promoted to a dual before the operation is carried out. This conversion rule is also valid for an assignment statement when data types of the lvalue and rvalue are different.
- In a pointer assignment statement, the pointer types of lvalue and rvalue can be different. They will be reconciled internally. To comply with the ANSI C standard, the data type of the rvalue can also be explicitly cast into that of the lvalue in an assignment in C<sup>H</sup>. For example, the statement `fp = (float*)intptr` will cast the integer pointer `intptr` to float pointer before its address is assigned to float pointer `fp`. But, the content pointed to by `intptr` will not be

changed by this data type casting operation. For example, if *\*intptr* is 90, the value of *\*fp* will not be equal to 90 because of the difference in their internal representations for int and float. The memory of a dual variable can be accessed by pointers. If real or imaginary part of a dual variable is obtained by a float pointer, the sign of a zero will be carried over, which will be discussed in section 5.

The following code fragment illustrates how different data types are automatically converted in C<sup>H</sup>.

```
int i;           /* declare i as int
float f;        /* declare f as float
complex;       /* declare c as complex
dual d, *dptr;  /* declare d as dual,
                /* dptr as dual pointer

i = 1;         /* i is 1
f = i+1;       /* f is 2.0
c = complex(i,f); /* c is complex(1,2)
d = c + 2;     /* d is dual(3,2)
d = dual(f*50, c); /* d is dual(100,3)
f = d;         /* f is 100
c = d;         /* c is complex(100,3)
d = i;         /* d is dual(1, 0)
d = NaN;       /* d is DualNaN
d = dual(Inf, Inf); /* d is DualInf
d = Inf;       /* d is DualInf
d = -Inf;      /* d is DualInf
c = d;         /* c is ComplexInf
dptr = &d;     /* dptr points to address of d
dptr = dptr +1; /* dptr points to memory
                /* at address of d plus 8 bytes
```

## 2.6 I/O for Dual Numbers

Since dual is a basic data type in C<sup>H</sup>, it is desired that the I/O for this data type is also handled in the same manner as real numbers. Like complex, the real and imaginary parts of a dual number can be treated as two individual floats by the functions **real(d)** and **imaginary(d)** which will be discussed in details in sections 3 and 4. Then, all standard I/O functions such as **printf()** and **scanf()** for real numbers presented in (Cheng, 1993a) can be readily used. In this section, how a dual number is treated as a single object by the standard I/O function will be discussed. Due to the space limit, only the enhancement related to the function **printf()** will be explained in the following discussions. However, the underlining principle can be applied to all other I/O functions as well. The format of function **printf()** in C<sup>H</sup> is as follows

```
int printf(char *format, arg1, arg2, ...)
```

The function **printf()** prints output to the standard output device under the control of the string pointed to by *format* and returns the number of characters printed. If the format string contains two types of objects: ordinary characters and conversion specifications beginning with a character of % and ending with a conversion character, the ANSI C rules for **printf()** will be used. If the format string in **printf()** contains only ordinary characters, the subsequent numerical constants or variables will be printed according to preset default formats. For function **printf()**, a single

conversion specification for a float will be used for both real and dual parts of a dual number. The default format for dual is "%.3f", which will be applied to both real and dual parts of a dual number. The dual metanumbers DualZero, DualInf and DualNaN are treated as regular dual numbers in I/O functions. For the debugging purpose, the default output for DualInf and DualNaN are dual(Inf, Inf) and dual(NaN, NaN), respectively. The default output for DualZero is dual(0.000,0.000), the number of zeros can be controlled by a format specifier. The following C<sup>H</sup> program will illustrate how dual numbers are handled by the I/O functions **printf()** and **scanf()**.

```
dual d1, d2, *dptr;
dptr = &d2; /* dptr points to d2's memory location
printf("Please type in two duals \n");
scanf(&d1, dptr);
printf("The first dual number is ", d1, "\n");
printf("The second dual number is %f \n", d2);
```

The result of the interactive execution of the above program is shown as follows

Please type in two duals

*1 2.0 3.0 4*

The first dual number is dual(1.000,2.000)

The second dual number is dual(3.000000,4.000000)

where the second line in italic is the input and the rest are the output of the program. In the future implementation, **scanf()** will accept constants such as DualInf, DualNaN, dual(2, 3.8F), etc..

### 3 Dual Operations

The arithmetic and relational operations for dual numbers are treated in the same manner as those for real and complex numbers in C<sup>H</sup>. This section will discuss how these operations are defined and handled by C<sup>H</sup>.

#### 3.1 Dual Operations with Dual Numbers

The negation, arithmetic and relational operations for dual numbers are defined in Table 1, where dual numbers  $d$ ,  $d_1$ , and  $d_2$  are defined as  $x + \varepsilon y$ ,  $x_1 + \varepsilon y_1$ , and  $x_2 + \varepsilon y_2$ , respectively.

Negation of a dual number changes the sign of both real and dual parts of the dual number. Addition of two dual numbers adds real and dual components of two dual numbers, separately. Subtraction of two dual numbers subtracts real and dual parts of the second dual number from real and dual parts of the first dual number, respectively. Treating the pure dual  $\varepsilon$  as a dual number of dual(0, 1), multiplication and division for two dual numbers are defined in Table 1. Dual numbers form an algebra, not a field. The pure dual numbers are zero divisors such that dual(0.0,y1)\*dual(0.0,y2) = dual(0.0,0.0). For binary operations with a real or complex operand and a dual operand, the real or complex operand will be promoted to a dual before the operation. One cannot compare whether one dual number is larger or smaller than other. But two dual numbers can be tested whether they are equal or not. Two dual numbers are equal to each other iff both real and dual parts of two dual numbers are equal to each other, separately. If real parts or dual parts of two dual numbers are not equal to each other, two dual numbers are said not equal.

Table 1: Dual operations.

Definition	C <sup>H</sup> Syntax	C <sup>H</sup> Semantics
negation	-d	$-x - \varepsilon y$
addition	d1 + d2	$(x_1 + x_2) + \varepsilon(y_1 + y_2)$
subtraction	d1 - d2	$(x_1 - x_2) + \varepsilon(y_1 - y_2)$
multiplication	d1 * d2	$x_1 * x_2 + \varepsilon(y_1 * x_2 + x_1 * y_2)$
division	d1 / d2	$\frac{x_1}{x_2} + \varepsilon \frac{y_1 * x_2 - x_1 * y_2}{x_2^2}$
equal	d1 == d2	$x_1 == x_2$ and $y_1 == y_2$
not equal	d1 != d2	$x_1 != x_2$ or $y_1 != y_2$

### 3.2 Dual Operations with Dual Metanumbers

In the above definitions of dual operations, we assume that all operands are dual numbers. The real and dual parts of a dual number are then treated as two regular floating-point floats. If the values of operands involve dual metanumbers, the definitions defined in Table 1 may not be valid. For example, DualInf is represented internally as dual(Inf, Inf). According to the dual addition defined in Table 1 and addition rule for real numbers discussed in (Cheng, 1993a), the result of addition of two DualInfs would be dual(Inf, Inf). But, addition of two dual infinities is mathematically indeterminate, which is the value of DualNaN in C<sup>H</sup>. Division of a dual number by a pure dual can be either DualNaN or DualInf, depending on the value of the numerator. Results for arithmetic and relational operations with both dual numbers and dual metanumbers are defined in (Cheng, 1993c).

## 4 Dual Functions

C is a small language, there is no intrinsic mathematical function in C. All functions are external. For example, functions `sinf()`, `sin()`, and `sinl()` are reserved for float, double, and long double sine functions, respectively. 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. External functions in C<sup>H</sup> can be created in the same manner as C functions are created, which will be discussed soon in section 6. Unlike C, however, the commonly used mathematical functions are built internally in the language itself in C<sup>H</sup>. The intrinsic mathematical functions in C<sup>H</sup> can handle different data types of arguments gracefully. The output data type and algorithm of a function depend on the data types of the input arguments, which is called *polymorphism*. Like arithmetic operators, the built-in commonly used mathematical functions in C<sup>H</sup> are polymorphic. For example, if the order of the data type of the input argument `x` is less than float, function `sin(x)` computes the sine of `x` in float. If `x` is double, the returned data is in double. The same function `sin(x)` will give complex or dual result if `x` is complex or dual, respectively. Hence, unlike programming with C, users of C<sup>H</sup> do not need to remember many arcane names.

For portability, all mathematical functions included in the ANSI C header `math.h` have been implemented polymorphically in C<sup>H</sup>. Names of built-in mathematical functions in C<sup>H</sup> are based upon the ANSI C header `math.h`. However, these mathematical functions and all mathematical operators in C<sup>H</sup> can be changed, added, or removed at user's convenience as delineated in (Cheng, 1993a).

The built-in polymorphic mathematical functions related to the dual numbers are listed in Table 2 along with their definitions. The input arguments of these dual functions can be dual numbers, dual variables, or dual expressions. For presentation purpose, the dual numbers  $d$ ,  $d_1$ , and  $d_2$  are defined as  $x + \varepsilon y$ ,  $x_1 + i\varepsilon y_1$ , and  $x_2 + \varepsilon y_2$ , respectively. Like dual arithmetic operations, definitions for dual functions may not be valid when the input arguments are dual metanumbers. The results of the built-in dual functions with dual metanumbers as their input arguments are given in (Cheng, 1993c).

## 5 Lvalues Related to Dual Numbers

As mentioned before that a lvalue is any object that occurs on the left hand side of an assignment statement. The valid lvalues related to dual numbers are listed in Table 3. The assignment operations `+=`, `-=`, `*=`, `/=`; address operation `&` and indirection operation `*`; as well as increment operation `++` and decrement operation `--` described in (Cheng, 1993a) can be applied to all these lvalues. Besides the simple variable in case (1), an element of a dual array can be a lvalue which is case (2) in Table 3. In case (3), pointer to dual is used as a lvalue to get the memory or to point to a memory of a dual object. In case (4), the memory pointed to by the pointer `dp_ptr` is assigned the value of the expression on the right hand side of a assignment statement. In addition to a single pointer variable, one can have an array of dual pointers. Cases (5) and (6) show how an element of a dual pointer array is used to access the memory. The function `real()` cannot only be used as a rvalue or an operand, but also used as a lvalue to access the memory of its argument. In case (7), the argument of `real()` must be a dual variable, or address pointed to by a dual pointer or pointer expression. A constant dual number or expression can be used as an input argument of function `real()` only when it is a rvalue or an operand. In case (8), the imaginary part of a dual is accessed by function `imaginary()` in the same manner as function `real()`. Since a dual number occupies two floats internally, this memory storage can be accessed not only by the functions `real()` and `imaginary()`, but also by a pointer-to-float as is shown in case (9) where the variable `f_ptr` is a pointer to float. For cases (7)-(9), a real number, including  $\pm 0.0$ ,  $\pm \text{Inf}$ , and `NaN`, on the right hand side will be assigned to lvalue formally without filtering. Therefore, abnormal dual numbers such as `dual(Inf, NaN)`, `dual(Inf, 0.0)`, etc. may be created. For example, two `CH` commands `real(d) = NaN` and `imaginary(d) = Inf` makes `d` equal to `dual(NaN, Inf)`; and `real(d) = -0.0` and `imaginary(d) = NZero` gives `d` the value of `dual(-0.0, -0.0)`. The following code fragment illustrate how lvalues are used in a `CH` program.

```
int i;
dual d1, d2, d3, da[3], *dp_ptr;
float f, *f_ptr;
d1 = dual(3,4);      /* d = dual(3,4);
d1 += 1;            /* d1 = d1+1;
d1++;              /* d1 = d1+1;
--d1;              /* d1 = d1-1;
d2 -= d1;          /* d2 = d2-d1;
d3 = d2++++*d1     /* d3 = d2*d1; d2 = d2+2;
d3 = d2+ ----d1;   /* d1 = d1-2; d3 = d2+d1;
d3 *= ++d1--;      /* d1 = d1+1; d3 = d3*d1;
                  /* d1 = d1-1;
d3 /= f-d3;        /* d3 = (f-d3)/d3;
da[i++] = ++da[2]; /* da[2] = da[2]+1;
```

Table 2: Syntax and semantics of built-in dual functions.

C <sup>H</sup> Syntax	C <sup>H</sup> Semantics
sizeof( $d$ )	8
abs( $d$ )	$\text{sqrt}(x^2 + y^2)$
real( $d$ )	$x$
imaginary( $d$ )	$y$
dual( $x, y$ )	$x + \varepsilon y$
conjugate( $d$ )	$x - \varepsilon y$
polar( $d$ )	$\text{sqrt}(x^2 + y^2) + i \Theta$
sqrt( $d$ )	$\text{sqrt}(x) + \varepsilon \frac{y}{2\text{sqrt}(x)}$
exp( $d$ )	$\exp(x)(1 + \varepsilon y)$
log( $d$ )	$\log(x) + \varepsilon \frac{y}{x}$
log10( $d$ )	$\frac{\log(d)}{\log(10)}$
pow( $d_1, d_2$ )	$d_1^{d_2} = x_1^{x_2} + \varepsilon(y_1 x_2 x_1^{x_2-1} + y_2 x_1^{x_2} \log(x_1))$
sin( $d$ )	$\sin(x) + \varepsilon y \cos(x)$
cos( $d$ )	$\cos(x) - \varepsilon y \sin(x)$
tan( $d$ )	$\tan(x) + \varepsilon \frac{y}{\cos(x) \cos(x)}$
asin( $d$ )	$\text{asin}(x) + \varepsilon \frac{y}{\text{sqrt}(1-x^2)}$
acos( $d$ )	$\text{acos}(x) - \varepsilon \frac{y}{\text{sqrt}(1-x^2)}$
atan( $d$ )	$\text{atan}(x) + \varepsilon \frac{y}{1+x^2}$
sinh( $d$ )	$\sinh(x) + \varepsilon y \cosh(x)$
cosh( $d$ )	$\cosh(x) + \varepsilon y \sinh(x)$
tanh( $d$ )	$\tanh(x) + \varepsilon \frac{y}{\cosh(x) \cosh(x)}$
asinh( $d$ )	$\text{asinh}(x) + \varepsilon \frac{y}{\text{sqrt}(x^2+1)}$
acosh( $d$ )	$\text{acosh}(x) - \varepsilon \frac{y}{\text{sqrt}(x^2-1)}$
atanh( $d$ )	$\text{atanh}(x) + \varepsilon \frac{y}{1-x^2}$
ceil( $d$ )	$\text{ceil}(x) + \varepsilon \text{ceil}(y)$
floor( $d$ )	$\text{floor}(x) + \varepsilon \text{floor}(y)$
ldexp( $d1, d2$ )	$\text{ldexp}(x_1, x_2) + \varepsilon \text{ldexp}(y_1, y_2)$
fmod( $d1, d2$ )	$d; \frac{d_1}{d_2} = k + \frac{d}{d_2}, k \geq 0$
modf( $d1, \&d2$ )	$\text{modf}(x_1, \&x_2) + \varepsilon \text{modf}(y_1, \&y_2)$
frexp( $d1, \&d2$ )	$\text{frexp}(x_1, \&x_2) + \varepsilon \text{frexp}(y_1, \&y_2)$

Table 3: Valid lvalues related to dual numbers.

Case	Meaning of lvalue	Example
1	simple variable	d = dual(1.0, 2);
2	an element of a dual array	darray[i] = dual(1.0, 2)+ DualInf;
3	dual pointer variable	dptr = malloc(sizeof(dual) * 3); dptr = &d;
4	address pointed to by a dual variable	*dptr = dual(1.0, 2) + z;
5	an element of a dual pointer array	darrayptr[i] = malloc(sizeof(dual) * 3); darrayptr[i] = &d;
6	address pointed to by an element of a dual pointer array	*darrayptr[i] = dual(1.0, 2);
7	real part of a dual variable	real(d) = 3.4;
	real part of a dual variable	real(*dptr) = 3.4;
	real part of a dual variable	real(*(dptr+1)) = 3.4;
	real part of a dual variable	real(*darrayptr[i]) = 3.4;
8	imaginary part of a dual variable	imaginary(d) = dual(1.0, 2);
	imaginary part of a dual variable	imaginary(*dptr) = 3.4;
	imaginary part of a dual variable	imaginary(*(dptr+1)) = 3.4;
	imaginary part of a dual variable	imaginary(*darrayptr[i]) = 3.4;
9	float pointer variable	fptr = &d; fptr = dptr;
	pointer to real part of a dual variable	*fptr = 1.0;
	pointer to imaginary part of a dual variable	*(fptr+1) = 2.0;

```

                /* da[i] = da[2]; i = i+1;
dptr = (dual *)malloc(sizeof(dual)*10);
        /* allocate 10 elements of duals for dptr.
*dptr++ = d1;      /* *dptr = d1; dptr = dptr+1;
fptr = (float *)&d1; /* fptr = &d1;
f = **fptr;      /* f = imaginary(d1);
real(d1)++++ = ++imaginary(d2);/* imaginari(d2) +=1;
        /* real(d1) = imaginari(d2); real(d1) += 2;

```

## 6 Creation of User's Dual Functions

User's dual functions in  $C^H$  can be created like real or complex functions in the spirit of ANSI C, which is demonstrated in this section by computation of the motion screw of a rigid body displacement.

According to Chasles' theorem, a general displacement of a rigid body can be represented by a unique screw displacement of dual angle  $\hat{\theta} = \theta + \varepsilon s$  about the screw axis  $\mathbf{u}$  as shown in Figure 6 where point  $p$  is moved from its initial position  $\mathbf{p}_1$  to  $\mathbf{p}$  along the screw axis (Suh and Radcliffe, 1978; Yang, 1973). The position vectors  $\mathbf{q}_1$  and  $\mathbf{q}$  for a point  $q$  on the rigid body before and after the screw motion, respectively, can be formulated as

$$\begin{aligned}
 \begin{bmatrix} \mathbf{q} \\ 1 \end{bmatrix} &= \begin{bmatrix} \mathbf{R}(\theta) & s\mathbf{u} + (\mathbf{I} - \mathbf{R}(\theta))\mathbf{p}_1 \\ \mathbf{O} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{q}_1 \\ 1 \end{bmatrix} \\
 &= \mathbf{D}(\theta, s, \mathbf{u}, \mathbf{p}_1) \begin{bmatrix} \mathbf{q}_1 \\ 1 \end{bmatrix} \quad (7)
 \end{aligned}$$

where  $\mathbf{R}(\theta)$  is a 3x3 finite Euler rotation matrix (Cheng and Gupta, 1989) and  $\mathbf{I}$  is a 3x3 identity matrix.

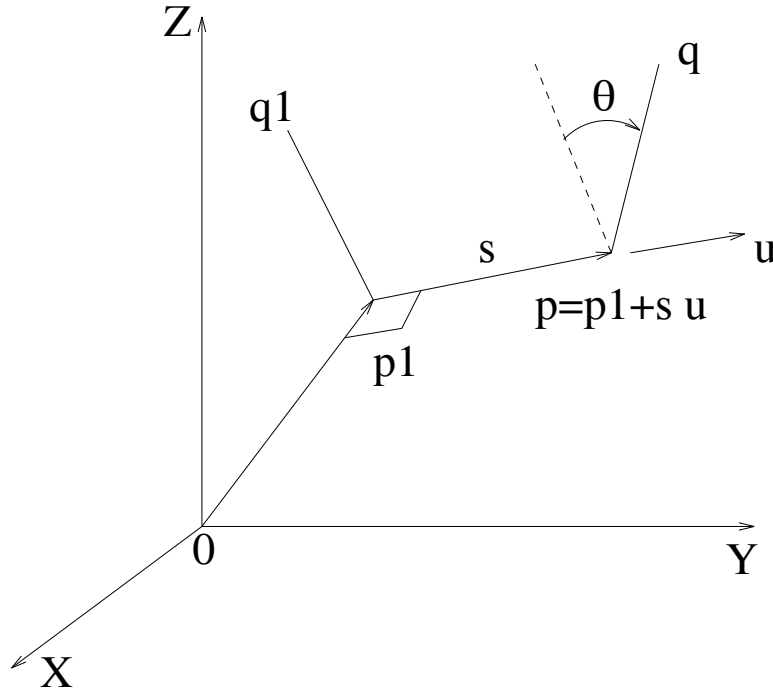


Figure 6: The screw displacement of a rigid body.

In many applications, parameters  $\theta$ ,  $s$ ,  $\mathbf{u}$ , and  $\mathbf{p}_1$  need to be computed from the displacement matrix  $\mathbf{D}(\theta, s, \mathbf{u}, \mathbf{p}_1)$ . This can be achieved through the following formulations.

$$\theta = \arccos\left(\frac{a_{11} + a_{22} + a_{33} - 1}{2}\right) \quad (8)$$

$$u_x = \frac{a_{32} - a_{23}}{2 \sin \theta}; \quad u_y = \frac{a_{13} - a_{31}}{2 \sin \theta}; \quad u_z = \frac{a_{21} - a_{12}}{2 \sin \theta} \quad (9)$$

$$s\mathbf{u} + (\mathbf{I} - \mathbf{R}(\theta))\mathbf{p}_1 = \mathbf{d} \quad (10)$$

where  $a_{ij}$  is an element of matrix  $\mathbf{D}$  and  $\mathbf{d}$  is the vector formed by the first three elements of the fourth column of matrix  $\mathbf{D}$ . For a general solution, we impose an additional constraint

$$\mathbf{u} \bullet \mathbf{p}_1 = u_x p_{1x} + u_y p_{1y} + u_z p_{1z} = 0 \quad (11)$$

to solve equation (10) simultaneously. Program 1 will calculate the motion screw for the displacement matrix

$$\mathbf{D}_i = \begin{bmatrix} -0.637 & 0.023 & 0.771 & 730.916 \\ 0.771 & 0.030 & 0.636 & 308.395 \\ -0.008 & 0.999 & -0.036 & 144.209 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12)$$

The output from the execution of the above program is

*The motion screw is dual(2.534833, 538.942871)*

In the above C<sup>H</sup> program, the input argument of the dual function `screw()` is a 4x4 displacement matrix  $\mathbf{D}$ , the returned dual number is the motion screw  $\hat{\theta}$ . We have used the built-in linear system solver `linearsolver()` to solve four linear equations (10) and (11) for  $s$  and  $\mathbf{p}_1$  simultaneously. The type qualifier `matrix` distinguishes C<sup>H</sup> arrays from regular arrays defined in ANSI C. An array declared as a matrix can be treated as a single object in C<sup>H</sup>. Details about array processing in C<sup>H</sup> will be reported elsewhere.

Program 1: The C<sup>H</sup> program for computation of the motion screw from a transformation matrix.

```

main()
{
float D[4][4] = {{-0.637, 0.023, 0.771, 730.916},
                { 0.771, 0.030, 0.636, 308.395},
                {-0.008, 0.999,-0.036, 144.209},
                { 0,    0,    0,    1    }};
dual screw(float D[4][4]);    /* dual function prototype
dual screwvalue;
screwvalue = screw(D);
printf("The motion screw is %f \n",screwvalue);
}

dual screw(float D[4][4])    /* define dual function
{
matrix float u[3], A[4][4], b[4], x[4];
float theta;
dual theta_s;
theta = acos((D[0][0] + D[1][1] + D[2][2] -1)/2);
u[0] = (D[2][1] - D[1][2])/(2*sin(theta));
u[1] = (D[0][2] - D[2][0])/(2*sin(theta));
u[2] = (D[1][0] - D[0][1])/(2*sin(theta));
b[0] = D[0][3]; b[1] = D[1][3]; b[2] = D[2][3]; b[3] = 0;
A[0][0] = u[0]; A[0][1] = 1-D[0][0]; A[0][2] = -D[0][1]; A[0][3] = -D[0][2];
A[1][0] = u[1]; A[1][1] = -D[1][0]; A[1][2] = 1-D[1][1]; A[1][3] = -D[1][2];
A[2][0] = u[2]; A[2][1] = -D[2][0]; A[2][2] = -D[2][1]; A[2][3] = 1-D[2][2];
A[3][0] = 0; A[3][1] = u[0]; A[3][2] = u[1]; A[3][3] = u[2];
x = linearsolver(A, b); /* x[0] is s; x[1],x[2],x[3] are p1
real(theta_s) = theta;
imaginary(theta_s) = x[0];
return theta_s;
}

```

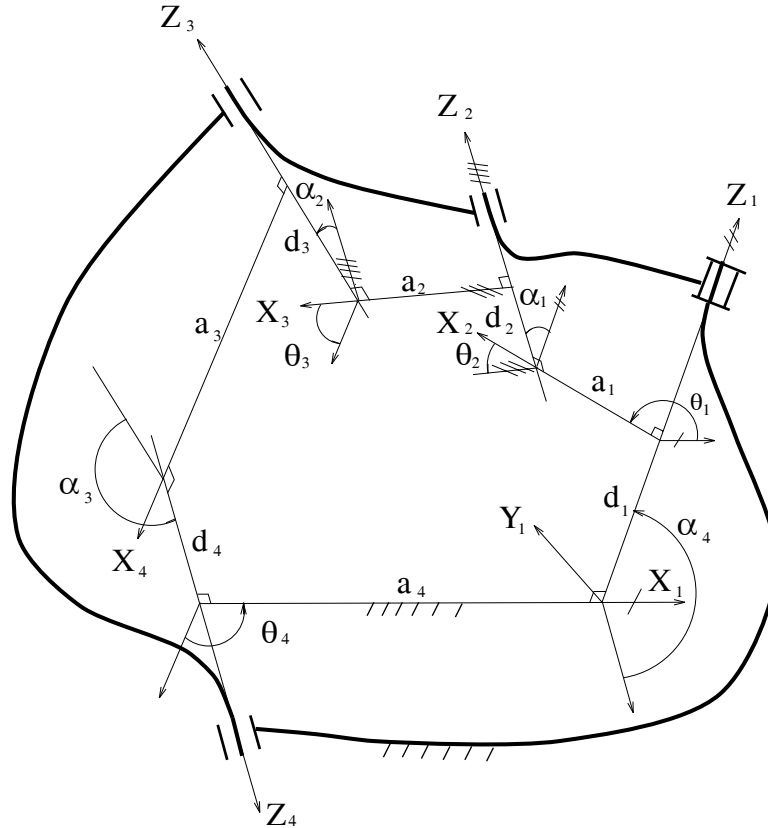


Figure 7: The RCCC mechanism.

## 7 Analysis of RCCC Mechanism

In this section, we will demonstrate the efficacy of  $C^H$  programming with dual numbers in spatial mechanisms design. We will perform the displacement analysis of the RCCC mechanism shown in Figure 7. A typical connection between adjacent links of a spatial mechanism is shown in Figure 8. The body coordinate system  $X_i Y_i Z_i O_i$  is attached to body  $i$  which is the physical link between joints  $i$  and  $i+1$ . Let  $\alpha_i$  be the twist angle between  $Z_{i-1}$  and  $Z_i$  measured along axis  $X_i$ ,  $a_i$  the shortest distance between  $Z_{i-1}$  and  $Z_i$  measured along the common normal  $X_i$ ,  $d_i$  the shortest distance between  $X_{i-1}$  and  $X_i$  along  $Z_{i-1}$ , and  $\theta_i$  the angle between  $X_{i-1}$  and  $X_i$  measured along  $Z_{i-1}$ . The transformation matrix which transforms the position specified in the body coordinate system  $X_i Y_i Z_i$  to the body coordinate system  $X_{i-1} Y_{i-1} Z_{i-1} O_{i-1}$  can be formulated by (Denavit-Hartenberg, 1955)

$$D_i = \begin{bmatrix} \cos \theta_i & -\sin \theta_i \cos \alpha_i & \sin \theta_i \sin \alpha_i & a_i \cos \theta_i \\ \sin \theta_i & \cos \theta_i \cos \alpha_i & -\cos \theta_i \sin \alpha_i & a_i \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (13)$$

Using dual numbers, the corresponding dual transformation matrix  $\hat{D}_i$  can be derived as follows. The screw motion

$$\hat{\theta}_i = \theta_i + \varepsilon d_i \quad (14)$$

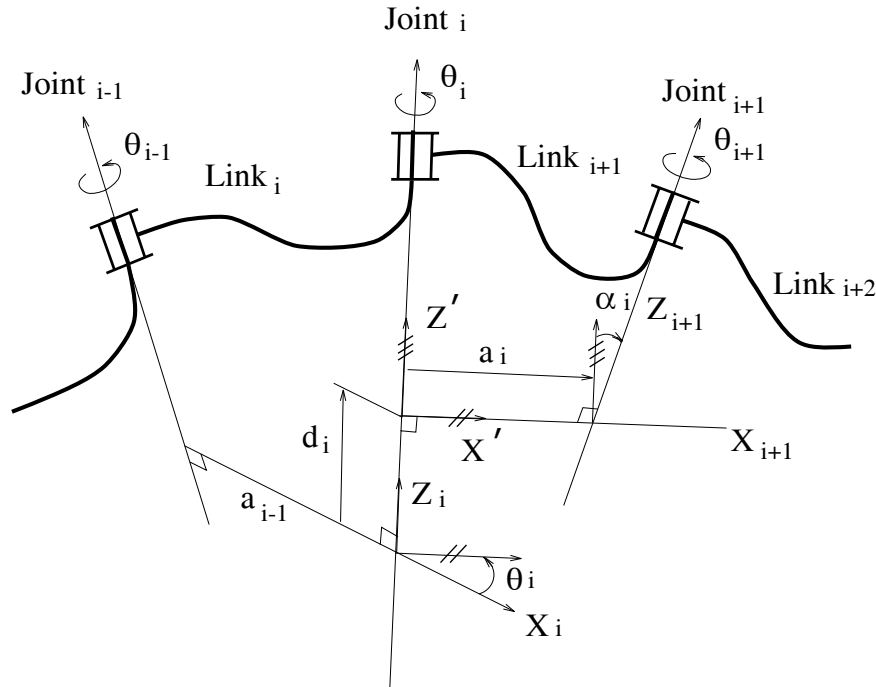


Figure 8: The Denavit-Hartenberg parameters for adjacent links.

along  $Z_i$  moves the coordinate system from  $X_{i-1}Y_{i-1}Z_{i-1}O_{i-1}$  to  $X'Y'Z'O'$  shown in Figure 8. The screw motion

$$\hat{\alpha}_i = \alpha_i + \varepsilon a_i \quad (15)$$

along  $X_i$  moves the coordinate system  $X'Y'Z'$  to  $X_iY_iZ_i$ . Let

$$\hat{\Theta}_i = \begin{bmatrix} \cos \hat{\theta}_i & -\sin \hat{\theta}_i & 0 \\ \sin \hat{\theta}_i & \cos \hat{\theta}_i & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (16)$$

$$\hat{\Lambda}_i = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \hat{\alpha}_i & -\sin \hat{\alpha}_i \\ 0 & \sin \hat{\alpha}_i & \cos \hat{\alpha}_i \end{bmatrix} \quad (17)$$

the corresponding dual D-H transformation matrix can be derived by the successive screw transformations as follows

$$\hat{\mathbf{D}}_i = \hat{\Theta}_i \hat{\Lambda}_i = \begin{bmatrix} \cos \hat{\theta}_i & -\sin \hat{\theta}_i \cos \hat{\alpha}_i & \sin \hat{\theta}_i \sin \hat{\alpha}_i \\ \sin \hat{\theta}_i & \cos \hat{\theta}_i \cos \hat{\alpha}_i & -\cos \hat{\theta}_i \sin \hat{\alpha}_i \\ 0 & \sin \hat{\alpha}_i & \cos \hat{\alpha}_i \end{bmatrix} \quad (18)$$

where  $\hat{\theta}_i$  and  $\hat{\alpha}_i$  are given in eqs(14) and (15), respectively.

In Figure 7, dual angles  $\hat{\alpha}_i = \alpha_i + \varepsilon a_i$  ( $i = 1, 2, 3, 4$ ) and  $d_1$  are given link parameters, and  $\theta_i, \alpha_i, a_i$ , and  $d_i$  follow the convention depicted in Figure 8. The revolute axis  $Z_1$  and cylindrical axis  $Z_4$  are fixed in the space. Coordinate systems  $O_1X_1Y_1Z_1$  and  $O_4X_4Y_4Z_4$  are attached to input and output links, respectively. The joint variable  $\theta_1$  is the input, the dual joint variable  $\hat{\theta}_4 = \theta_4 + \varepsilon d_4$  is the output. The other two dual joint variables are  $\hat{\theta}_2 = \theta_2 + \varepsilon d_2$  and  $\hat{\theta}_3 = \theta_3 + \varepsilon d_3$ . The displacement analysis of this mechanism is that, given link parameters  $\hat{\alpha}_i = \alpha_i + \varepsilon a_i$  ( $i = 1, 2, 3, 4$ ) and  $d_1$ , determine the output  $\hat{\theta}_4$  and joint variables  $\hat{\theta}_2$  and  $\hat{\theta}_3$  in terms of the input angle  $\theta_1$ .

The solution of displacements of output and moving joints can be derived as follows. According to screw motion matrices, we can write the loop closure equation for the RCCC mechanism shown in Figure 7 as

$$\widehat{\Theta}_3 \widehat{\Lambda}_3 \widehat{\Theta}_4 \widehat{\Lambda}_4 \widehat{\Theta}_1 \widehat{\Lambda}_1 \widehat{\Theta}_2 \widehat{\Lambda}_2 = \mathbf{I} \quad (19)$$

where screw motion matrices  $\widehat{\Theta}_i$  and  $\widehat{\Lambda}_i$  are defined in eqs(16) and (17), respectively, and  $\mathbf{I}$  is a 3x3 identity matrix. Eq(19) can be reformulated as

$$\widehat{\Lambda}_3 \widehat{\Theta}_4 \widehat{\Lambda}_4 \widehat{\Theta}_1 \widehat{\Lambda}_1 = \widehat{\Theta}_3^T \widehat{\Lambda}_2^T \widehat{\Theta}_2^T \quad (20)$$

where  $\widehat{\Theta}_i^T$  and  $\widehat{\Lambda}_i^T$  are the transposes of  $\widehat{\Theta}_i$  and  $\widehat{\Lambda}_i$ , respectively. After expanding the above matrix equation, element (3,3) gives us the following input-output relation of the mechanism

$$\begin{aligned} & (\sin \hat{\alpha}_1 \sin \hat{\alpha}_3 \sin \hat{\theta}_1) \sin \hat{\theta}_4 \\ & - \sin \hat{\alpha}_3 (\cos \hat{\alpha}_1 \sin \hat{\alpha}_4 + \sin \hat{\alpha}_1 \cos \hat{\alpha}_4 \cos \hat{\theta}_1) \cos \hat{\theta}_4 \\ & + \cos \hat{\alpha}_3 (\cos \hat{\alpha}_1 \cos \hat{\alpha}_4 - \sin \hat{\alpha}_1 \sin \hat{\alpha}_4 \cos \hat{\theta}_1) = \cos \hat{\alpha}_2 \end{aligned} \quad (21)$$

Let

$$\widehat{A} = \sin \hat{\alpha}_1 \sin \hat{\alpha}_3 \sin \hat{\theta}_1 \quad (22)$$

$$\widehat{B} = -\sin \hat{\alpha}_3 (\cos \hat{\alpha}_1 \sin \hat{\alpha}_4 + \sin \hat{\alpha}_1 \cos \hat{\alpha}_4 \cos \hat{\theta}_1) \quad (23)$$

$$\widehat{C} = \cos \hat{\alpha}_3 (\cos \hat{\alpha}_1 \cos \hat{\alpha}_4 - \sin \hat{\alpha}_1 \sin \hat{\alpha}_4 \cos \hat{\theta}_1) - \cos \hat{\alpha}_2 \quad (24)$$

Eq (21) becomes

$$\widehat{A} \sin \hat{\theta}_4 + \widehat{B} \cos \hat{\theta}_4 + \widehat{C} = 0 \quad (25)$$

The output  $\hat{\theta}_4$  can be obtained from the above equation as

$$\hat{\theta}_4 = 2 \operatorname{atan} \left( \frac{-\widehat{A} \pm \sqrt{\widehat{A}^2 + \widehat{B}^2 - \widehat{C}^2}}{\widehat{C} - \widehat{B}} \right) \quad (26)$$

where  $\widehat{A}$ ,  $\widehat{B}$ , and  $\widehat{C}$  are defined in eqs(22)-(24). Note that there are two branches of  $\hat{\theta}_4$  for a given  $\hat{\theta}_1$  in eq(26). Define elements (3,1) and (3,2) of the resultant matrix on the left hand side of eq(20) as

$$\begin{aligned} \widehat{E}_{31} &= \sin \hat{\alpha}_3 \cos \hat{\theta}_1 \sin \hat{\theta}_4 \\ &+ (\cos \hat{\alpha}_3 \sin \hat{\alpha}_4 + \sin \hat{\alpha}_3 \cos \hat{\alpha}_4 \cos \hat{\theta}_1) \sin \hat{\theta}_1 \end{aligned} \quad (27)$$

$$\begin{aligned} \widehat{E}_{32} &= -\sin \hat{\alpha}_3 [\cos \hat{\alpha}_1 \sin \hat{\theta}_1 \sin \hat{\theta}_4 \\ &+ (\sin \hat{\alpha}_4 \sin \hat{\alpha}_1 - \cos \hat{\alpha}_4 \cos \hat{\alpha}_1 \cos \hat{\theta}_1) \cos \hat{\theta}_4] + \\ &\cos \hat{\alpha}_3 (\cos \hat{\alpha}_4 \sin \hat{\alpha}_1 + \sin \hat{\alpha}_4 \cos \hat{\alpha}_1 \cos \hat{\theta}_1) \end{aligned} \quad (28)$$

respectively. Elements (3,1) and (3,2) of eq(20) give

$$\widehat{E}_{31} = \sin \hat{\alpha}_2 \sin \hat{\theta}_2 \quad (29)$$

$$\widehat{E}_{32} = -\sin \hat{\alpha}_2 \cos \hat{\theta}_2 \quad (30)$$

The dual joint variable  $\hat{\theta}_2$  can be solved from the above two equations

$$\hat{\theta}_2 = 2 \operatorname{atan} \left( \frac{\widehat{E}_{31}}{\sin \hat{\alpha}_2 - \widehat{E}_{32}} \right) \quad (31)$$

Similarly, define elements (1,3) and (2,3) of the resultant matrix on the left hand side of eq(20) as

$$\begin{aligned} \widehat{E}_{13} = & \sin \hat{\alpha}_1 \sin \hat{\theta}_1 \cos \hat{\theta}_4 \\ & + (\cos \hat{\alpha}_1 \sin \hat{\alpha}_4 + \sin \hat{\alpha}_1 \cos \hat{\alpha}_4 \cos \hat{\theta}_1) \sin \hat{\theta}_4 \end{aligned} \quad (32)$$

$$\begin{aligned} \widehat{E}_{23} = & \cos \hat{\alpha}_3 [\sin \hat{\alpha}_1 \sin \hat{\theta}_1 \sin \hat{\theta}_4 - \\ & (\sin \hat{\alpha}_4 \cos \hat{\alpha}_1 + \cos \hat{\alpha}_4 \sin \hat{\alpha}_1 \cos \hat{\theta}_1) \cos \hat{\theta}_4] - \\ & \sin \hat{\alpha}_3 (\cos \hat{\alpha}_4 \cos \hat{\alpha}_1 - \sin \hat{\alpha}_4 \sin \hat{\alpha}_1 \cos \hat{\theta}_1) \end{aligned} \quad (33)$$

respectively. Elements (1,3) and (2,3) in eq(20) give

$$\widehat{E}_{13} = \sin \hat{\alpha}_2 \sin \hat{\theta}_3 \quad (34)$$

$$\widehat{E}_{23} = \sin \hat{\alpha}_2 \cos \hat{\theta}_3 \quad (35)$$

The dual joint variable  $\hat{\theta}_3$  can be derived from the above two equations

$$\hat{\theta}_3 = 2 \operatorname{atan} \left( \frac{\widehat{E}_{13}}{\sin \hat{\alpha}_2 + \widehat{E}_{23}} \right) \quad (36)$$

As one can see that dual number is a powerful mathematical tool for spatial mechanisms design. Dual solutions can be elegantly derived from dual equations. But, dual formulas such as (22)-(24) and (26) are awkward to handle by currently existing computer programming languages. The conciseness of these symbolic dual solutions will diminish if numerical computations are needed.

The clarity and succinctness of dual formulas, however, can be preserved in a C<sup>H</sup> program. Let link parameters for a RCCC mechanism be  $\hat{\alpha}_1 = 30^\circ + \varepsilon 2$  in,  $\hat{\alpha}_2 = 55^\circ + \varepsilon 4$  in,  $\hat{\alpha}_3 = 45^\circ + \varepsilon 3$  in,  $\hat{\alpha}_4 = 60^\circ + \varepsilon 5$  in, and  $d_1 = 0$ . Program 2 can compute the output  $\hat{\theta}_4$  of this RCCC mechanism. In Program 2,  $\hat{\alpha}_1$ ,  $\hat{\alpha}_2$ ,  $\hat{\alpha}_3$ , and  $\hat{\alpha}_4$  are declared as dual variables `alpha1`, `alpha2`, `alpha3`, and `alpha4`;  $\hat{\theta}_1$ ,  $\hat{\theta}_2$ ,  $\hat{\theta}_3$ , and  $\hat{\theta}_4$  are `theta1`, `theta2`, `theta3`, and `theta4`;  $\widehat{A}$ ,  $\widehat{B}$ ,  $\widehat{C}$ ,  $\widehat{E}_{31}$ ,  $\widehat{E}_{32}$ ,  $\widehat{E}_{13}$ , and  $\widehat{E}_{23}$ , are `A`, `B`, `C`, `E31`, `E32`, `E13`, and `E23`; respectively. Dual formulas (22)-(24), (26), (27), (28), (31), (32), (33), and (36) are translated almost verbatim into C<sup>H</sup> expressions in the same manner as real or complex formulas. The input angle  $\theta_1$  of  $\hat{\theta}_1$  varies from 0 to 360 degrees with an interval of 20 degrees by using a Fortran-style do-loop (Cheng, 1993d). The output of the above program is a text file of *RCCC.out* shown in Figure 9. The file has seven columns. The first column is the input angle  $\theta_1$ ; the remaining are  $\theta_2$ ,  $d_2$ ,  $\theta_3$ ,  $d_3$ ,  $\theta_4$ , and  $d_4$  of dual angles  $\hat{\theta}_2$ ,  $\hat{\theta}_3$ , and  $\hat{\theta}_4$ , respectively. In Program 2, we have computed two sets of solutions by function `RCCC()`. The sign for a positive or negative square root in eq(26) for  $\hat{\theta}_4$  is passed as an argument of function `RCCC()`. The numerical data in Figure 9 matches with the result obtained by a much more elaborative FORTRAN program (Yang, 1963; Yang and Freudenstein, 1964). As is shown in this example, programming with dual numbers in C<sup>H</sup> is easier than in any other computer programming languages.

## 8 Conclusions

Dual numbers are generalization of real numbers. Similar to the extended complex plane, the extended dual plane is introduced. For scientific computing with dual numbers, the extended finite dual plane along with dual metanumbers of `DualZero`, `DualInf`, and `DualNaN` are also introduced in this paper. The dual arithmetic and relational operations, and dual functions are defined in the syntax of the C<sup>H</sup> programming language. The results of dual operations and dual functions with dual metanumbers may differ from those obtained directly according to formal definitions for dual numbers. Due to polymorphism, the algorithms and resultant data types of arithmetic operations

Program 2: The C<sup>H</sup> program for analysis of RCCC mechanism.

```

/* RCCC */
#include <linkage.h>
main()
{
    void RCCC(int branch, FILE *stream);
    FILE *stream;
    stream = fopen("RCCC.out", "w"); /* open file "RCCC.out"
    fprintf(stream, "theta1 theta2      d2(in)      theta3");
    fprintf(stream, "      d3(in)      theta4      d4(in)  \n\n");
    fprintf(stream, "                                  First solution \n\n");
    RCCC(1, stream);
    fprintf(stream, "\n\n                          Second solution \n\n");
    RCCC(-1, stream);
    fclose(stream);                /* close file "RCCC.out"
}

void RCCC(int branch, FILE *stream)/* routine for RCCC analysis
{
    int i;
    dual alpha1, alpha2, alpha3, alpha4, theta1, theta2, theta3, theta4;
    dual A, B, C, E31, E32, E13, E23;

    alpha1 = dual(30*PI/180, 2);
    alpha2 = dual(55*PI/180, 4);
    alpha3 = dual(45*PI/180, 3);
    alpha4 = dual(60*PI/180, 5);
    do i = 0, 360, 20
    {
        theta1 = dual(i*PI/180, 0);          /* 0 <= theta1 <= 360 degree */
        A = sin(alpha1)*sin(alpha3)*sin(theta1);
        B = -sin(alpha3) * (cos(alpha1)*sin(alpha4) +
            sin(alpha1)*cos(alpha4)*cos(theta1));
        C = cos(alpha3) * (cos(alpha1)*cos(alpha4) -
            sin(alpha1)*sin(alpha4)*cos(theta1)) - cos(alpha2);
        theta4 = 2*atan((-A+branch*sqrt(A*A + B*B - C*C))/(C-B));
        E31 = sin(alpha3)*cos(theta1)*sin(theta4) +(cos(alpha3)*sin(alpha4)+
            sin(alpha3)*cos(alpha4)*cos(theta4))*sin(theta1);
        E32 = - sin(alpha3)*(cos(alpha1)*sin(theta1)*sin(theta4) +
            (sin(alpha4)*sin(alpha1) - cos(alpha4)*cos(alpha1)*cos(theta1))*cos(theta4)
            +cos(alpha3)*(cos(alpha4)*sin(alpha1)+sin(alpha4)*cos(alpha1)*cos(theta1));
        theta2 = 2 *atan(E31/(sin(alpha2) - E32));
        E13 = sin(alpha1)*sin(theta1)*cos(theta4) +(cos(alpha1)*sin(alpha4)+
            sin(alpha1)*cos(alpha4)*cos(theta1))*sin(theta4);
        E23 = cos(alpha3)*(sin(alpha1)*sin(theta1)*sin(theta4) -
            (sin(alpha4)*cos(alpha1) + cos(alpha4)*sin(alpha1)*cos(theta1))*cos(theta4)
            -sin(alpha3)*(cos(alpha4)*cos(alpha1)-sin(alpha4)*sin(alpha1)*cos(theta1));
        theta3 = 2 *atan(E13/(sin(alpha2) + E23));
        fprintf(stream, "%3.0f %10.3f %10.3f %10.3f %10.3f %10.3f %10.3f \n",
            real(theta1)*180/PI, real(theta2)*180/PI, dual(theta2),
            real(theta3)*180/PI, dual(theta3), real(theta4)*180/PI, dual(theta4));
    }
}

```

theta1	theta2	d2(in)	theta3	d3(in)	theta4	d4(in)
First solution						
0	149.679	-0.210	45.556	-2.693	144.209	-0.115
20	130.460	-1.247	49.071	-2.512	131.899	-0.921
40	108.761	-2.288	58.311	-2.146	116.674	-1.771
60	86.600	-2.959	70.948	-1.817	101.195	-2.248
80	65.032	-3.192	85.270	-1.588	87.219	-2.259
100	44.087	-3.081	100.205	-1.463	75.723	-1.889
120	23.027	-2.748	114.907	-1.440	67.559	-1.262
140	0.332	-2.256	128.318	-1.525	64.214	-0.529
160	-26.466	-1.515	138.577	-1.701	68.597	0.011
180	-59.094	-0.301	142.648	-1.814	83.700	-0.173
200	-92.620	0.913	138.577	-1.701	105.330	-0.843
220	-119.185	1.384	128.318	-1.525	124.052	-1.086
240	-138.048	1.371	114.906	-1.440	136.989	-0.938
260	-151.899	1.220	100.205	-1.463	145.467	-0.663
280	-163.025	1.055	85.270	-1.588	150.868	-0.368
300	-173.011	0.902	70.948	-1.817	153.854	-0.084
320	176.810	0.732	58.310	-2.146	154.370	0.150
340	164.930	0.433	49.071	-2.512	151.599	0.220
360	149.679	-0.210	45.556	-2.693	144.209	-0.115
Second solution						
0	-149.679	0.210	-45.556	2.693	-144.209	0.115
20	-164.931	-0.433	-49.071	2.512	-151.599	-0.220
40	-176.810	-0.732	-58.311	2.146	-154.370	-0.150
60	173.010	-0.902	-70.948	1.817	-153.854	0.084
80	163.025	-1.055	-85.270	1.588	-150.868	0.368
100	151.899	-1.220	-100.205	1.463	-145.467	0.663
120	138.047	-1.371	-114.907	1.440	-136.989	0.938
140	119.184	-1.384	-128.318	1.525	-124.052	1.086
160	92.619	-0.912	-138.577	1.701	-105.329	0.843
180	59.092	0.301	-142.648	1.814	-83.699	0.173
200	26.465	1.515	-138.577	1.701	-68.596	-0.011
220	-0.333	2.256	-128.318	1.525	-64.214	0.529
240	-23.028	2.748	-114.906	1.440	-67.559	1.262
260	-44.088	3.081	-100.205	1.463	-75.724	1.889
280	-65.033	3.192	-85.270	1.588	-87.220	2.259
300	-86.601	2.959	-70.948	1.817	-101.195	2.248
320	-108.762	2.288	-58.310	2.146	-116.675	1.771
340	-130.461	1.247	-49.071	2.512	-131.900	0.921
360	-149.680	0.210	-45.556	2.693	-144.209	0.115

Figure 9: The output file from displacement analysis of a RCCC mechanism.

depend on the data types of operands, and the algorithms and resultant data types of mathematical functions are related to the data types of input arguments. Since dual is implemented as a basic data type in  $C^H$ , numerical computations can be handled in a much integrated fashion. Dual numbers, dual variables, dual operations, dual formulas, and dual functions are handled in the same manner as real or complex ones in  $C^H$  in the spirit of ANSI C. It is the first time, I believe, that such a simplicity is achieved in a general-purpose computer programming language for scientific computing with dual numbers. A sample  $C^H$  program for the computation of the motion screw of a rigid body displacement illustrates the handling of user's dual functions. Dual formulas for displacement analysis of a RCCC mechanism can be easily translated into a  $C^H$  program, which demonstrates the simplicity of programming with dual formulas in the  $C^H$  programming language. It is expected that, with conciseness, clarity, and programming simplicity of dual formulas, the recognition of dual numbers as a powerful mathematical tool, which is well-known to kinematicians, will reach wide audience in engineering and science.

## 9 Acknowledgement

The author would like to thank Prof. A. T. Yang for reading this paper and his comments and suggestions.

## 10 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. ANSI, *ANSI/IEEE Standard 770 X3.97-1983, IEEE Standard Pascal Programming Language*, IEEE, Inc., NJ, 1983.
3. ANSI, *ANSI Standard X3.159-1989, Programming Language C*, ANSI, Inc., NY, 1989.
4. Bottema, O. and Roth, B.: *Theoretical Kinematics*, North-Holland Publishing Comp., Amsterdam, 1979.
5. Cheng, H. H., Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP, (*IEEE*) *Computer*, Vol. 22, No. 9, September, 1989, pp. 31-46.
6. Cheng, H. H., Scientific Computing in the  $C^H$  Programming Language, Department of MAME, University of California, Davis, Technical Report TR-MAME-93-101, February 18, 1993; also NCEG, X3J11.1/93-033, June 11, 1993; also *Scientific Programming*, Vol. 2, No. 3, Fall, 1993a, pp. 49-75.
7. Cheng, H. H., Handling of Complex Numbers in the  $C^H$  Programming Language, Department of MAME, University of California, Davis, Technical Report TR-MAME-93-102, February 18, 1993; also NCEG, X3J11.1/93-020, June 11, 1993; also *Scientific Programming*, Vol. 2, No. 3, Fall, 1993b, pp. 76-106.
8. Cheng, H. H., Computations of Dual Numbers in the Extended Finite Dual Plane, *Proc. of 1993 ASME Design Automation Conference*, Albuquerque, NM, Sept. 19-22, 1993c, vol. 2, pp. 73-80.
9. Cheng, H. H., Adding Do-Loop to C for Scientific Computing, NCEG, X3J11.1/93-031, June 11, 1993d.
10. Cheng, H. H., Numerical Computations in the  $C^H$  Programming Language with Applications in Mechanisms and Robotics. Part I: Programming with Real Numbers; Part II: Programming with Complex Numbers; Part III: Programming with Dual Numbers. *Proc. of the Third National Conference on Applied Mechanisms and Robotics*, Cincinnati, OH, Nov. 8-10, 1993e.

11. Cheng, H. H., Passing Arrays to Functions under the Programming Paradigm of C, NCEG, X3J11.1/93-041, September 18, 1993f.
12. Cheng, H. H. and Gupta, K. C., An Historical Note on Finite Rotations, *Trans. of ASME, Journal of Applied Mechanics*, Vol. 56, No. 1, 1989, pp. 139-145.
13. Clifford, W. K., Preliminary Sketch of Bi-quaternions, *Proceedings of London Mathematical Society*, Vol. 4, 1873, pp. 381-395.
14. Denavit, J., Displacement Analysis of Mechanisms Based on (2x2) Matrices of Dual Numbers, VDI-Berichte, Vol. 29, 1958, pp. 81-89.
15. Denavit, J. and Hartenberg, R. S., A Kinematic Notation for Lower Pair Mechanisms Based on Matrices, *ASME Trans. J. Applied Mech.*, vol. 22, 1955, pp. 215-221.
16. Dimentberg, F. M., The Determination of the Positions of Spatial Mechanisms, *Izdatel'stvo Akademii Nauk*, Moscow, USSR, 1950.
17. Dooley, J. R. and McCarthy, J. M., Spatial Rigid Body Dynamics Using Dual Quaternion Components, *Proc. of IEEE International Conf. on Robotics and Automation*, vol. 1, Sacramento, CA, April 1991, pp.90-95.
18. Gerald, C. F. and Wheatley, P. O., *Applied Numerical Analysis*, Addison-Wesley Publishing Company, Inc., New York, 1984.
19. Guggenheimer, H. W., *Differential Geometry*, McGraw-Hill Book Co., NY, 1963.
20. 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.
21. Kahan, W., Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit, *The State of the Art in Numerical Analysis* (ed. Iserles & Powell), 1987, Oxford Univ. Press; *Proc. of the Joint IMA/SIAM Conference*, April 14-18, 1986.
22. Kahan, W. and J. W. Thomas, Augmenting a Programming Language with Complex Arithmetic, NCEG/91-039, November 15, 1991.
23. Knaak, D., Complex Extension to C, X3J11.1/92-075, November 17, 1992.
24. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, first edition, 1978; second edition, 1988.
25. Kotelnikov, A. P., Screw Calculus and Some of its Applications to Geometry and Mechanics, *Annals of Imperial University of Kazan*, 1895.
26. MacDonald, T., Adding Complex Arithmetic to C, *The Journal of C Language Translation*, Vol. 1, pp. 20-31, 1989.
27. MathWorks, *Pro-MATLAB User's Guide*, The MathWorks, Inc., 1990.
28. McAulay, A., *Octonions- A Development of Clifford's Bi-quaternions*, Cambridge University Press, 1898.
29. 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.
30. Pennock, G. R. and Yang, A. T., Dynamic Analysis of Multi-Rigid-Body Open-Chain System, *Trans. ASME, J. of Mechanisms, Transmissions, and Automation in Design*, Vol.105, March 1983, pp. 28-34.
31. 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.
32. Ravani, B. and Ge Q. J., Kinematic Localization for World Model Calibration in Off-Line Robot Programming Using Clifford Algebra, *Proc. of IEEE International Conf. on Robotics and Automation*, vol. 1, Sacramento, CA, April 1991, pp. 584-589.

33. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Publishing Company, Inc., 1987.
34. Study, E., *Geometrie der Dynamen*, Leipzig, 1903.
35. Suh, C. H. and Radcliffe, C. W.: *Kinematics and Mechanism Design*, John Wiley and Sons Inc., New York, 1978.
36. Tydeman, F., Merging Complex and IEEE-745, Document X3J11.1 (NCEG) 92-061.
37. Yang, A. T., *Application of Quaternion Algebra and Dual Numbers to the Analysis of Spatial Mechanisms*, Doctoral Dissertation, Columbia University, New York, 1963.
38. Yang, A. T., Calculus of Screw, *Basic Questions of Design Theory*, North Holland, Amsterdam, 1974, pp. 266-281.
39. 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.