 Scientific Programming, Vol. 2, No. 3, Fall, 1993, pp. 76-106.

Handling of Complex Numbers in the C^{H} Programming Language

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 handling of complex numbers in the C^{H} programming language will be described in this paper. Complex is a built-in data type in C^{H}. The I/O, arithmetic and relational operations, and built-in mathematical functions are defined for both regular complex numbers and complex metanumbers of ComplexZero, ComplexInf, and ComplexNaN. Due to polymorphism, the syntax of complex arithmetic and relational operations, and built-in mathematical functions are the same as those for real numbers. Besides polymorphism, the built-in mathematical functions are implemented with a variable number of arguments, which greatly simplies computations of different branches of multiple-valued complex functions. The valid lvalues related to complex numbers are defined. Rationales for the design of complex features in C^{H} are discussed from language design, implementation, and application points of views. Sample C^{H} programs show that a computer language which does not distinguish the sign of zeros in complex numbers can also handle the branch cuts of multiple-valued complex functions effectively so long as it is appropriately designed and implemented.

1 Introduction

Cheng (1993) presented the extension of C to C^{H}, a general-purpose block-structured interpretive programming language, for the numerical computation of real numbers. The extension of scientific programming from the one-dimensional real line to the two-dimensional extended complex plane will be described in this paper. Complex number, an extension of real number, has wide applications in science and engineering. Due 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 (ANSI, 1978), a language mainly for scientific computing, has provided complex data type since its earliest days. Ada has introduced complex data in its new proposed standard recently (Hodgson, 1991a,b; Squire, 1991a,b). C, a modern language originally invented for the Unix system programming (Ritchie and Thompson, 1974; Kernighan and Ritchie, 1978), does not have complex as a basic data type because the 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 this

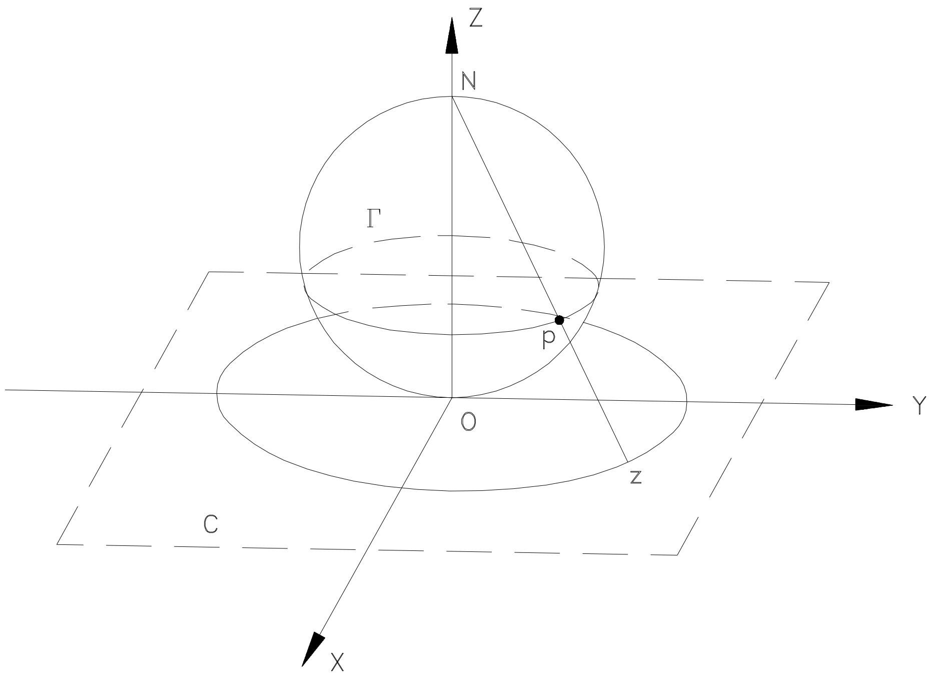

Figure 1: The Riemann sphere Γ and extended complex plane.

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. Besides, many features described in this paper cannot be conveniently implemented at user's level.

C^H retains most features of C for scientific computing and extends C's capabilities in many aspects. Providing complex as a basic data type is one of its extensions. The reason for providing complex as a basic data type is not only for programming convenience, but also due to design considerations. Design considerations such as automatic data conversion, handling of metanumbers, and optional arguments in a function 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. As described in (Cheng, 1993), 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 (IEEE, 1985) easily available to the programmer. This paper 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 ANSI C (ANSI, 1989). Mathematically, complex numbers can be represented in the extended complex plane shown in Figure 1 (Churchill and Brown, 1984; Marsden, 1973). In Figure 1, there is a one-to-one correspondence between the points on the Riemann sphere Γ and the points on the extended complex plane $\mathcal{C}$. The point p on the surface of the sphere is determined by the intersection of the line through the point z and the north pole N of the sphere. There is only *one* complex infinity in the extended complex plane. The north pole N corresponds to the point at infinity. Due to the finite representation of floating-point numbers, *the extended finite complex plane* shown in Figure 2 is introduced in this paper. Any com-

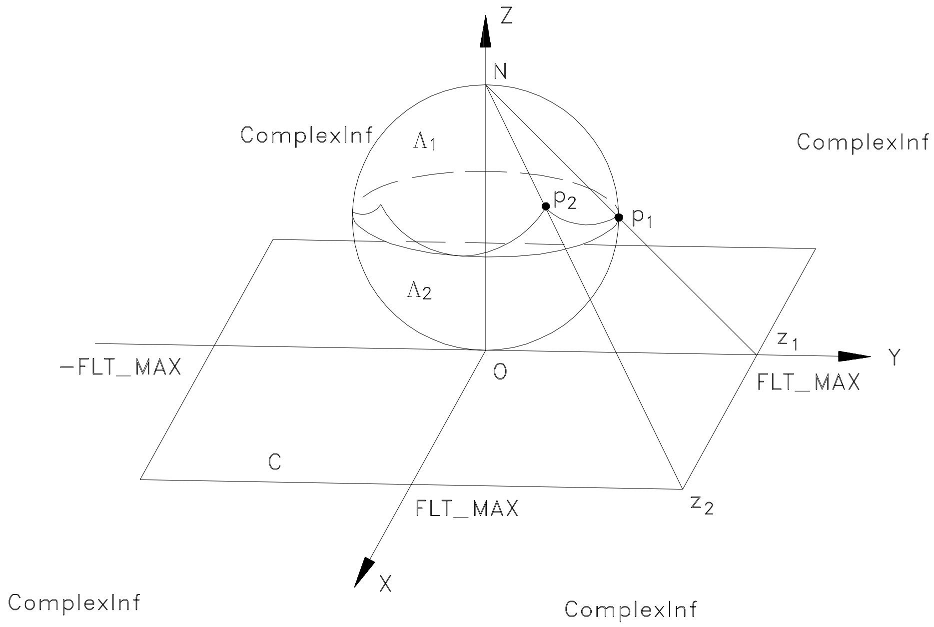

Figure 2: The unit sphere Λ and extended finite complex plane.

plex 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. Outside this rectangular area, a complex number is treated as a Complex-Infinity represented as ComplexInf or complex(Inf,Inf) in $\mathrm{C^H}$. The one-to-one correspondence between points on the Riemann sphere Γ and the extended complex plane is no longer valid for the unit sphere Λ and the extended finite complex plane. All points on the surface of the upper part Λ_1 of the unit sphere correspond to the complex infinity. Points on the lower part Λ_2 of the sphere and points in the extended finite complex plane are in one-to-one correspondence. The boundary between surfaces Λ_1 and Λ_2 corresponds to the threshold of overflow. For example, points p_1 and p_2 on the unit sphere Λ correspond to points $z_1 =$ complex(FLT_MAX, 0.0) and $z_2 =$ complex(FLT_MAX, FLT_MAX) in the extended finite complex plane shown in Figure 2, respectively. The origin of the extended finite complex plane is complex(0.0, 0.0) or ComplexZero which stands for Complex-Zero. In $\mathrm{C^H}$, an undefined or 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 directions 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 $\mathrm{C^H}$ do not distinguish 0.0 from -0.0 for real and imaginary parts of complex numbers. Due to these differences, some operations and functions need to be handled differently for real and complex numbers, especially for real metanumbers and complex metanumbers. For example, following the IEEE 754 standard, the addition of two real positive infinities is a value of infinity in $\mathrm{C^H}$ (Cheng, 1993). But, the addition of two complex infinities is indeterminate, according to complex analysis, although the value of ComplexInf is represented internally as two positive infinities of Inf. As another example, following the ANSI C standard (ANSI, 1989), the mathematical function **atan2**(y, x) in $\mathrm{C^H}$ returns a value in the range of $[-\pi, \pi]$. The value of the expression **atan2**$(-0.0, -1)$ is $-\pi$. Using this result as the phase angle for complex number $-1.0 - i0.0$, the square root of $-1.0 - i0.0$, expressed in $\mathrm{C^H}$ as **sqrt**(complex$(-1.0, -0.0)$), becomes complex$(0.0, -1.0)$ which is obtained by $\cos(-\pi/2) + i\sin(-\pi/2) = 0.0 - i$. In our definition, this is the second branch of the square root function for the complex number of complex$(-1.0, -0.0)$, obtained by the expression **sqrt**(complex$(-1.0, -0.0), 1)$ where the second argument of the function **sqrt**() indicates the branch number with the default value of 0. As illustrated in this example, the mathematical functions in $\mathrm{C^H}$ are not only polymorphic, but also with variable number of arguments so that the function **sqrt**() cannot only be used to compute the square root of a real number, but also to calculate the different branches of the square root of a complex number. Due to polymorphism and variable number of arguments for mathematical functions, scientific computing with complex numbers in $\mathrm{C^H}$ is much simpler in comparison to Fortran and other languages. In Fortran, there are only few standard mathematical functions and they can only calculate the principal branches of multiple-valued functions.

Manipulation of complex numbers in $\mathrm{C^H}$, as it is currently implemented, will be described in this paper. The rest of the paper is organized as follows. Section 2 discusses how complex numbers and variables are created in $\mathrm{C^H}$. The data conversions between real numbers and complex numbers are explained. The I/O for complex numbers is also illustrated in this section. Sections 3 and 4 present complex operations and complex functions, respectively. Section 5 defines valid objects on the left hand side of an assignment statement related to complex numbers. Section 6 demonstrates how user's complex functions in $\mathrm{C^H}$ can be conveniently created through the example for computation of the logarithm of the complex gamma function. Section 7 provides rationales for the handling of complex numbers in $\mathrm{C^H}$, in comparison with existing systems and current research efforts in the design of languages with a complex data type.

2 Complex Numbers

2.1 Complex Constants and Complex Variables

Complex numbers $z \in \mathcal{C} = \{(x, y) \mid x, y \in \mathcal{R}\}$ can be defined as ordered pairs

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

with specific addition and multiplication rules (Churchill and Brown, 1984; Marsden, 1973). The real numbers x and y are called the *real* and *imaginary parts* of z. If we identify the pair of $(x, 0.0)$ as the real numbers. The real number $\mathcal{R}$ is a subset of $\mathcal{C}$, i.e., $\mathcal{R} = \{(x, y) \mid x \in \mathcal{R}, y = 0.0\}$ and $\mathcal{R} \subset \mathcal{C}$. If a real number is considered either as x or $(x, 0.0)$ and let i denote the *pure imaginary number* $(0, 1)$, complex numbers can be mathematically represented as

$$z = x + iy \tag{2}$$

Both mathematical notations (1) and (2) can be implemented for complex numbers in a computer language. General-purpose computer programming languages such as Fortran, Ada, Common Lisp (Steele, 1990) tend to use notation (1) whereas software packages such as Mathematica (Wolfram, 1988) and MATLAB (MathWorks, 1990) incline to notation (2). Following the lead of Fortran in scientific programming, a complex number can be created in $\mathrm{C^H}$ by the complex constructor **complex**(x, y) with $x, y \in \mathcal{R}$. For example, a complex number with its real part of 3.0 and imaginary part of 4.0 can be constructed by **complex**(3.0, 4.0). Internally, a complex number consists of two floats at the current implementation. Therefore, if arguments of a complex constructor are not floats, they will be cast to floats internally. As described in (Cheng, 1993), all floating-point constants in C$^{\mathrm{H}}$ are floats by default. The double constants can be obtained by suffixing a floating-point constant with D or d. When double complex data type is implemented in the future, the complex constructor shall return complex or double complex polymorphically, depending on the data types of the input arguments. For example, **complex**(3, 4.0d), **complex**(3.0f, 4.0d), **complex**(3.0D, 4.0F), and **complex**(3.0D, 4.0D) shall return a double complex number of **complex**(3.0D, 4.0D). By default, `complex, ComplexZero, ComplexInf`, and `ComplexNaN` are keywords in C$^{\mathrm{H}}$. However, as described in (Cheng, 1993), these keywords can be changed at user's discretion. For example, one can add `CMPLX` to the keyword list and remove `complex` from the list by `addkey("CMPLX","complex")` and `remkey("complex")`, respectively. With this keyword change from `complex` to `CMPLX`, `CMPLX` will act the same as `complex` in a C$^{\mathrm{H}}$ program in both syntax and semantics. Hence, porting code related to complex numbers from Fortran to C$^{\mathrm{H}}$ is relatively easy. Many C programs have defined `complex` as the definition of a structure for complex numbers. With the keyword changeability, reserved word conflict can be avoided when porting C programs to C$^{\mathrm{H}}$.

One can declare not only a *simple complex variable*, but also *pointer to complex*, *array of complex*, and *array of pointer to complex*, etc.. Declarations of these complex variables are similar to the declarations of any other data types in C. The array and pointer of complex in C$^{\mathrm{H}}$ are manipulated in the same manner as the floating-point float and double. The following code segment will illustrate how complex is declared and manipulated in C$^{\mathrm{H}}$:

```
complex z1;             /* declare z1 as complex variable */
complex *zptr1;         /* declare zptr1 as pointer to complex variable */
complex z2[2], z3[2,3];/* declare z2 and z3 as arrays of complex */
complex *zptr2[2][4];   /* declare zptr2 as an array of pointer to complex */
zpt1r = &z1;            /* zptr1 point to the address of z1 */
*zptr1 = complex(1,2); /* z1 becomes 1+i2 */
```

2.2 Data Conversion Rules

C$^{\mathrm{H}}$ is a loosely typed language. All arguments of calling functions will be checked for compatibility with the data types of the called functions. The 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 into C$^{\mathrm{H}}$ so that they can be invoked whenever necessary. This will save many explicit type conversion commands for a program. The order of the data type in C$^{\mathrm{H}}$ is arranged as

data type	order
complex	↑ high
double	
float	
int	
char	low

with char being the lowest data type and complex the highest data type. The default conversion rules will be briefly discussed in this section as follows:

- Char, int, float, and double can be converted according to ANSI C data 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 converted to complex with its imaginary part being zero. When casting a real number into a complex number, the values of Inf and −Inf become ComplexInf; and the value of NaN becomes ComplexNaN. Conversion from double to complex may lose the information. A real number can be cast into a complex explicitly by the complex construction function **complex**(x,y), which will be discussed in details in section 4.

- When a complex is converted to char, int, float, and double; only its real part is used, and the imaginary part will be discarded. If the value of a complex is ComplexInf, the converted real number becomes NaN. The real and imaginary components of a complex number can be obtained explicitly by the functions **real**(z) and **imaginary**(z), which will be discussed in details in section 4. When a complex number is converted to a real number either implicitly by assignment statement such as f = z or explicitly by **real**(z), **imaginary**(z), **float**(z), **double**(z), (**float**)z, and (**double**)z; the sign of a zero will not be carried over. Converting a complex number to an integral value such as char and int is equivalent to conversion of **real**(z) to an integral value. For example, `i = ComplexInf` 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, which will allow experimentation with signed zeros in computations of complex numbers. A *lvalue* is any object that occurs on the left hand 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 will be discussed in section 5.

- 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 complex and the data type of other operand is a real number, the real number will be cast into a complex 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$^{\text{H}}$. 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 contents 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 complex variable can be accessed by pointers. If real or imaginary part of a complex 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 segment will illustrate how different data types are automatically converted in C$^{\text{H}}$.

```
char c;
int i;
```

```
float f;
double d;
complex z, *zptr;
c = 'a';                /* c is  'a' */
i = c;                  /* i is 97, ASCII number of  'a' */
f = i;                  /* f is 97.0 */
d = i;                  /* d is 97.0 */
z = complex(ch+1, f);   /* z is 98.0 + i 97.0 */
z = complex(Inf, Inf);  /* z is ComplexInf */
z = Inf;                /* z is ComplexInf */
z = -Inf;               /* z is ComplexInf */
f = z;                  /* f is NaN, since real(ComplexInf) is NaN */
d = z;                  /* d is NaN, since real(ComplexInf) is NaN */
i = Inf;                /* i is 2147483647 = INT_MAX, */
i = z;                  /* i is 2147483647, int of NaN is 2147483647
                           plus warning message */
z = complex(d+1, 3);    /* z is 98.0 + i 3.0 */
c = z;                  /* c is 'b', ASCII number of 'b' is 98 */
i = z;                  /* i is  98 */
f = z;                  /* f is  98.0 */
d = d;                  /* d is  98.0 */
z = NaN;                /* z is ComplexNaN */
zptr = &z;              /* zptr points to address of z */
zptr++;                 /* zptr points to memory at address of z plus 8 bytes */
```

2.3 I/O for Complex Numbers

Since complex is a basic data type in C^H, it is desired that the I/O for this data type is also handled in the same manner as real numbers. Similar to Fortran, the real and imaginary parts of a complex number can be treated as two individual floats by the functions **real**(z) and **imaginary**(z) as will be discussed in sections 3 and 4. Then, all standard I/O functions such as **printf**() and **scanf**() for real numbers presented in (Cheng, 1993) can be readily used. In this section, how a complex 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 other I/O functions as well. The format of function **printf**() in C^H 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 imaginary parts of a complex number. The default format for complex is %f, which will be applied to both real and imaginary parts of a complex number. The metanumbers ComplexZero, ComplexInf and ComplexNaN are treated as regular complex numbers in I/O functions. For the debugging purpose, the default output for ComplexInf and ComplexNaN are complex(Inf, Inf) and complex(NaN, NaN), respectively. The default output for ComplexZero is complex(0.000000,0.000000). The format for real and imaginary parts can be controlled by a format specifier. The following C$^{\mathrm{H}}$ program will illustrate how complex numbers are handled by the I/O functions **printf**() and **scanf**().

```
complex z1, z2, *zptr;
zptr = &z2;          /* zptr points to z2's memory location  */
printf("Please type in real and imaginary of two complex numbers \n");
scanf(&z1, zptr);
printf("The first complex is ", z1, "\n");
printf("The second complex is  %.3f \n", z2);
```

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

```
Please type in real and imaginary of two complex numbers
```

1 2.0 3.0 4

```
The first complex is complex(1.000000,2.000000)
The second complex is complex(3.000,4.000)
```

where the second line in italic is the input and the rest are the output of the program. Function **printf**() in C$^{\mathrm{H}}$ is in full compliance with ANSI C. Function **scanf**() in C$^{\mathrm{H}}$ at its current implementation has a minor difference from ANSI C. In the future implementation, **scanf**() will comply with ANSI C; besides, it will accept the input constants such as ComplexInf, ConmplexNaN, complex(2, 3.8F), etc..

3 Complex Operations

The arithmetic and relational operations for complex numbers are treated in the same manner as those for real numbers in C$^{\mathrm{H}}$. This section will discuss how these operations are defined and handled by C$^{\mathrm{H}}$.

3.1 Complex Operations with Regular Complex Numbers

The negation of a complex number, and arithmetic and comparison operations for two complex numbers are defined in Table 1, where the complex numbers z, z_1, and z_2 are defined as $x + iy, x_1 + iy_1$, and $x_2 + iy_2$, respectively.

The negation of a complex number will change the sign of both real and imaginary parts of the complex number. The addition of two complex numbers will add real and imaginary components of two complex numbers, separately. The subtraction of two complex numbers will subtract real and imaginary parts of the second complex number from real and imaginary of the first complex number, respectively. Treating the imaginary number i as a complex number of complex(0, 1), the multiplication and division for two complex numbers are defined in Table 1. For binary operations with a real operand and a complex operand, the regular real operand will be cast into a complex before the operation. Complex numbers are not ordered, one cannot compare whether one complex number is larger or smaller than other. But two complex numbers can be tested whether they are equal or not. Two complex numbers are equal to each other iff both real and imaginary parts of two complex numbers are equal to each other, separately. If real or imaginary parts of two complex numbers are not equal to each other, two complex numbers are said not equal.

Table 1: The complex operations.

Definition	$\mathrm{C^H}$ Syntax	$\mathrm{C^H}$ Semantics
negation	−z	$-x - iy$
addition	z1 + z2	$(x_1 + x_2) + i(y_1 + y_2)$
subtraction	z1 − z2	$(x_1 - x_2) + i(y_1 - y_2)$
multiplication	z1 ∗ z2	$(x_1 * x_2 - y_1 * y_2) + i(y_1 * x_2 + x_1 * y_2)$
division	z1 / z2	$\dfrac{x_1 * x_2 + y_1 * y_2}{x_2^2 + y_2^2} + i\dfrac{y_1 * x_2 - x_1 * y_2}{x_2^2 + y_2^2}$
equal	z1 == z2	$x_1 == x_2$ and $y_1 == y_2$
not equal	z1 != z2	$x_1 != x_2$ or $y_1 != y_2$

Table 2: Negation results.

Negation −				
operand	complex(0.0, 0.0)	z	ComplexInf	ComplexNaN
result	complex(0.0, 0.0)	−z	ComplexInf	ComplexNaN

3.2 Complex Operations with Complex Metanumbers

In the above definitions of complex operations, we assume that all operands are regular complex numbers. The real and imaginary parts of a complex number are then treated as two regular floating-point floats. If the values of operands involve complex metanumbers, the definitions defined in Table 1 may not be valid. For example, ComplexInf is represented internally as complex(Inf, Inf). According to the complex addition definition defined in Table 1 and addition rule for real numbers discussed in (Cheng, 1993), the result of addition of two ComplexInfs would be complex(Inf, Inf). But, addition of two complex infinities is mathematically indeterminate. Therefore, the results for arithmetic and relational operations with both regular complex numbers and complex metanumbers are defined in Tables 2 to 7.

From a programmer's point of view, values of complex(±0.0, ±0.0) are the same as complex(0.0, 0.0) or ComplexZero when they are used as operands or arguments in $\mathrm{C^H}$. In the following discussions, the positive zero 0.0 and the negative zero −0.0 for real and imaginary components of a complex number are considered the same. Therefore, although the negation of complex(0.0, 0.0) returns complex(−0.0, −0.0), the result listed in Table 2 is complex(0.0, 0.0). Negation of a complex infinity is still a complex infinity. Of course, negation of a complex Not-a-Number is ComplexNaN.

For binary operations in Tables 3 to 5, if any one of operands is ComplexNaN, the result is ComplexNaN. If one of two operands is ComplexInf and other is a finite complex number, the result of addition and subtraction is ComplexInf. Unlike real numbers, addition and subtraction of two ComplexInfs are ComplexNaNs. Multiplication of ComplexInf with complex(0.0, 0.0) is ComplexNaN; multiplication of ComplexInf with a finite nonzero number is ComplexInf; and multiplication of two ComplexInfs becomes ComplexInf. Like real numbers, divisions of complex(0.0, 0.0) by complex(0.0, 0.0) and ComplexInf by ComplexInf are ComplexNaNs. A finite number or infinity divided by complex(0.0, 0.0) becomes ComplexInf. The division of ComplexInf by a finite number gives ComplexInf. Theoretically, two complex infinities cannot be compared with each other, they may or may not be equal to each other. In $\mathrm{C^H}$, however, two ComplexInfs are considered the same from the programming point of view as shown in Table 6. Likewise, the comparison of two

Table 3: Addition and subtraction results.

Addition and Subtraction ±				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	complex(0.0, 0.0)	±z2	ComplexInf	ComplexNaN
z1	z1	z1 ± z2	ComplexInf	ComplexNaN
ComplexInf	ComplexInf	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 4: Multiplication results.

Multiplication ∗				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
z1	complex(0.0, 0.0)	z1∗z2	ComplexInf	ComplexNaN
ComplexInf	ComplexNaN	ComplexInf	ComplexInf	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

ComplexNaNs will get a logic TRUE. This design consideration is also reflected in the `not equal` relational operation shown in Table 7.

4 Complex Functions

Besides the polymorphism, the mathematical functions implemented in C^H can have a variable number of arguments, which is very convenient for calculations of complex mathematical functions with multiple branches. If a mathematical function, as a real function, has only one real argument, the additional second argument will render the function to a complex function unless explained otherwise. The integral value of the second argument will indicate the branch of the complex function. When this second argument presents, the first argument will be cast into a complex number according to the previously discussed data type conversion rules when the order of its data type is lower than complex. For a mathematical function with two arguments as a real function, if either one of two input arguments is a complex, the mathematical function becomes a complex

Table 5: Division results.

Division ÷				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
z1	ComplexInf	z1/z2	complex(0.0, 0.0)	ComplexNaN
ComplexInf	ComplexInf	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 6: Equal comparison results.

Equal comparison ==				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	1	0	0	0
z1	0	z1 == z2	0	0
ComplexInf	0	0	1	0
ComplexNaN	0	0	0	1

Table 7: Not equal comparison results.

Not equal comparison !=				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	0	1	1	1
z1	1	z1 != z2	1	1
ComplexInf	1	1	0	1
ComplexNaN	1	1	1	0

function. If an additional third argument as a branch indicator is provided, the function becomes a complex function if data types of the first two arguments are lower than or equal to complex. If their data types are lower than complex, they will be cast into complex numbers.

4.1 Results of Complex Functions with Regular Complex Numbers

The built-in functions related to the complex numbers are listed in Table 8 along with their definitions. The input arguments of these functions can be complex numbers, variables, or expressions. For the presentation purpose, the complex numbers z, z_1, and z_2 are defined as $x+iy, x_1+iy_1$, and x_2+iy_2, respectively. The integer values of k, k_1, and k_2 are the branch numbers of complex functions. If arguments for these branch numbers of the calling function are not integers, they will be cast into integers internally. For mathematical expressions in the second column in Table 8, if the arguments of mathematical functions are regular real numbers, the mathematical functions are real mathematical functions which have been described in (Cheng, 1993). The results of complex functions involving complex metanumbers will be discussed in the next section. In Table 8, the principal value Θ of the argument of a complex number is in the range of $-\pi < \Theta \leq \pi$. The definition of the principal value Θ for various complex numbers are given in Table 9. Note that the trigonometric function atan2(y,x) is in the range of $-\pi \leq \mathrm{atan2}(y, x) \leq \pi$. Normally, through complex arithmetic and complex functions, one shall not get a complex number with its real or imaginary part being the value of −Inf, Inf, or NaN while the other part is a regular real number. This kind of result can be obtained only explicitly by functions **real**(z) and **imaginary**(z), and float pointer variables through lvalues, which will be discussed in section 5.

The first four functions in Table 8 return real numbers. The **sizeof**() function returns, in bytes, an integer of the variable, type-specifier, or expression that it precedes. Since $\mathrm{C^H}$ does not have unsigned data types at its current implementation, the returned data type is signed integer, which slightly differs from ANSI C. If the argument is a complex, it will return the value of 8 which is

Table 8: The syntax and semantics of built-in complex functions.

C^H Syntax	C^H Semantics
sizeof(z)	8
abs(z)	$\text{sqrt}(x^2+y^2)$
real(z)	x
imaginary(z)	y
complex(x, y)	$x+iy$
conjugate(z)	$x-iy$
polar(z)	$\text{sqrt}(x^2+y^2)+i\Theta;\ \Theta=\text{atan2}(y,x)$
polar($r, theta$)	$r\cos(theta)+ir\sin(theta)$
sqrt(z)	$\text{sqrt}(\text{sqrt}(x^2+y^2))(\cos\frac{\Theta}{2}+i\sin\frac{\Theta}{2});\ \Theta=\text{atan2}(y,x)$
sqrt(z, k)	$\text{sqrt}(\text{sqrt}(x^2+y^2))(\cos\frac{\Theta+2k\pi}{2}+i\sin\frac{\Theta+2k\pi}{2});\ \Theta=\text{atan2}(y,x)$
exp(z)	$e^x(\cos y+i\sin y)$
log(z)	$\log(\sqrt{x^2+y^2})+i\Theta;\ \Theta=\text{atan2}(y,x)$
log(z, k)	$\log(\sqrt{x^2+y^2})+i(\Theta+2k\pi);\ \Theta=\text{atan2}(y,x)$
log10(z)	$\frac{\log(z)}{\log(10)}$
log10(z, k)	$\frac{\log(z,k)}{\log(10)}$
pow(z_1, z_2)	${z_1}^{z_2}=e^{z_2\ln z_1}=\exp(z_2*\log(z_1))$
pow(z_1, z_2, k)	${z_1}^{z_2}=e^{z_2\ln z_1}=\exp(z_2*\log(z_1,k))$
ceil(z)	$\text{ceil}(x)+i\ \text{ceil}(y)$
floor(z)	$\text{floor}(x)+i\ \text{floor}(y)$
fmod($z1, z2$)	$z;\quad \frac{z_1}{z_2}=k+\frac{z}{z_2},\ k\ge 0$
modf($z1$, &$z2$)	$\text{modf}(x_1,\&x_2)+i\ \text{modf}(y_1,\&y_2)$
frexp($z1$, &$z2$)	$\text{frexp}(x_1,\&x_2)+i\ \text{frexp}(y_1,\&y_2)$
ldexp($z1, z2$)	$\text{ldexp}(x_1,x_2)+i\ \text{ldexp}(y_1,y_2)$
sin(z)	$\sin x\cosh y+i\cos x\sinh y$
cos(z)	$\cos x\cosh y-i\sin x\sinh y$
tan(z)	$\frac{\sin z}{\cos z}$
asin(z)	$-i\log(iz+\text{sqrt}(1-z^2))$
asin(z, k)	$-i\log(iz+\text{sqrt}(1-z^2,k))$
asin($z, k1, k2$)	$-i\log(iz+\text{sqrt}(1-z^2,k_1),k_2)$
acos(z)	$-i\log(z+i\text{sqrt}(1-z^2))$
acos(z, k)	$-i\log(z+i\text{sqrt}(1-z^2,k))$
acos($z, k1, k2$)	$-i\log(z+i\text{sqrt}(1-z^2,k_1),k_2)$
atan(z)	$\frac{1}{2i}\log(\frac{1+iz}{1-iz})$
atan(z, k)	$\frac{1}{2i}\log(\frac{1+iz}{1-iz},k)$
atan2($z1, z2$)	$\frac{1}{2i}\log(\frac{1+iz_1/z_2}{1-iz1_{/}z_2})$
atan2($z1, z2, k$)	$\frac{1}{2i}\log(\frac{1+iz_1/z_2}{1-iz1_{/}z_2},k)$
sinh(z)	$\sinh x\cos y+i\cosh x\sin y$
cosh(z)	$\cosh x\cos y+i\sinh x\sin y$
tanh(z)	$\frac{\sinh x\cos y+i\cosh x\sin y}{\cosh x\cos y+i\sinh x\sin y}$

continued on next page

Table 8: continued.

$\mathrm{C^H}$ Syntax	$\mathrm{C^H}$ Semantics
asinh(z)	$\log(z + \mathrm{sqrt}(z^2+1))$
asinh(z, k)	$\log(z + \mathrm{sqrt}(z^2+1, k))$
asinh($z, k1, k2$)	$\log(z + \mathrm{sqrt}(z^2+1, k_1), k_2)$
acosh(z)	$\log(z + \mathrm{sqrt}(z+1)\mathrm{sqrt}(z-1))$
acosh(z, k)	$\log(z + \mathrm{sqrt}(z+1, k)\mathrm{sqrt}(z-1, k))$
acosh($z, k1, k2$)	$\log(z + \mathrm{sqrt}(z+1, k_1)\mathrm{sqrt}(z-1, k_1), k_2)$
atanh(z)	$\frac{1}{2}\log(\frac{1+z}{1-z})$
atanh(z, k)	$\frac{1}{2}\log(\frac{1+z}{1-z}, k)$

Table 9: The principal value Θ $(-\pi < \Theta \leq \pi)$ of the argument for complex(x,y).

Θ						
y value	x value					
	$-$x1	$-$0.0	0.0	x2	Inf	NaN
y2	$\mathrm{atan2}(y_2, -x_1)$	pi/2	pi/2	$\mathrm{atan2}(y_2, x_2)$		
0.0	pi	0.0	0.0	0.0		
$-$0.0	pi	0.0	0.0	0.0		
$-$y1	$\mathrm{atan2}(-y_1, -x_1)$	$-$pi/2	$-$pi/2	$\mathrm{atan2}(-y_1, x_2)$		
Inf					Inf	
NaN						NaN

the number of bytes required for storing two floats of real and imaginary parts of a complex. The **abs**(z) function computes the modulus of a complex number. The returned data type is float. The functions **real**(z) and **imaginary**(z) returns the real and imaginary parts of a complex number, respectively. The results of **real**(z) and **imaginary**(z) are always floats. If the data type of the argument for **real**() is lower or equal to double, the input data will be cast into a float. If the data type of the argument for **imaginary**() is lower than or equal to double, the value of zero will be returned. The sign of a zero will be ignored in **real**(z) and **imaginary**(z) functions. For example, **real**(complex($-$0.0,0.0)) will return 0.0.

A complex number can be created from two real numbers by the complex construction function **complex**(x,y). If the input arguments are not floats, they will be cast into floats according to the internal data conversion rules. The sign of a zero for x or y will be carried over to the complex number.

The **conjugate**(z) function returns the complex conjugate $\overline{z}$ of z. The complex number $\overline{z}$ represented by the point $(x, -y)$ is the reflection in the real axis of the point (x, y) representing z.

The function **polar**() is implemented mainly for the convenience of transformation between Cartesian and polar representations of a complex number. If there is only one input argument, a complex number with its real and imaginary parts being the modulus and argument, respectively, of the input complex number will be returned. If there are two input arguments, the complex number z in the polar form will be returned. The first and second input arguments are the modulus and argument of z, respectively. According to the definition $re^{i\theta}$ for the polar function, negative values for r are valid.

For the square root function **sqrt**(), whenever there are two arguments, the first argument is treated as a complex number. In case it is not a complex number and cannot be cast into a complex number, a syntax error message will be reported by the system. If the second argument is not an integer, it will be cast into an integral value according to internal data conversion rules. For the complex square root, there are only two distinct branches because of the periodic natures of the sine and cosine functions. In general, for taking the nth root, there are n distinct branches. If the function **sqrt**() is invoked with a single complex argument, the default branch value of 0 will be used.

The **exp**(z) function will calculate the exponential function of the complex number z.

Like the square root function, the natural logarithmic function **log**() has multiple branches. The branch number is provided by the second argument of the function. For the convenience, the function **log10**() will calculate the base-ten logarithmic function of a complex value.

The exponential function with a complex base can be calculated by the function **pow**(), which is accomplished by the exponential function and logarithmic function as is shown in Table 8. The branch of the logarithmic function determines the branch of the function **pow**(). Unlike its corresponding real function, the complex function **pow**() is always well-defined. If any one of two arguments of **pow**(z1, z2) is complex, the result is complex which is obtained by the principal branch of the expression **exp**(z2***log**(z1)). The result of the expression y^x equals the real part of the expression **pow**(complex(y,0.0), complex(x,0.0)) with its imaginary part being zero. For the function **pow**(z1, z2, k), z1 and z2 can be any data type lower than or equal to complex, and k is an integer. Whenever there are three arguments for the function **pow**(), the first and second arguments are treated as complex numbers. If z2 is an integer, all branches will have the same result; thus the solution is unique.

For functions **ceil**(z), **floor**(z), and **ldexp**(z1, z2), the real and imaginary parts are treated as if their were two separate real functions. The functions **modf**(z1, &z2) and **frexp**(z1, &z2) are handled in the same manner. For these two functions, when the data type of the first arguments are complex, the data type of the second argument must be a pointer to complex. The **fmod**(z1,z2) function computes the complex remainder of z1/z2.

The complex trigonometric functions **sin**(z), **cos**(z), and **tan**(z) and complex hyperbolic functions **sinh**(z), **cosh**(z), and **tanh**(z) have unique values. But, the complex inverse trigonometric functions **asin**(z), **acos**(z), and **atan**(z) and complex inverse hyperbolic functions **asinh**(z), **acosh**(z), and **atanh**(z) have multiple branches for a given input complex value. The second argument of these inverse functions indicates the branch number. For functions **asin**(), **acos**(), **asinh**(), and **acosh**(), the second and third arguments specify the branches of the related square root and logarithmic functions, respectively. The function **atan2**() is implemented similar to the function **atan**().

4.2 Results of Complex Functions with Complex Metanumbers

Like complex arithmetic operations, the definition for regular complex functions may not be valid when the input arguments are complex metanumbers. The results of the built-in complex functions with complex metanumbers as their input arguments are given in Table 10. In Table 10, complex($\pm$0.0, $\pm$0.0) in $\mathrm{C^H}$ is treated as complex(0.0, 0.0). When the input argument of a function is ComplexNaN, the returned result is always ComplexNaN except for the function **sizeof**(). As shown in Figure 2, a complex infinity is different from the real infinities of $\pm\infty$. When either real or imaginary part of a complex value is outside the range of the representable floating-point number, it becomes ComplexInf. Therefore, the absolute value of ComplexInf is a real number of Inf. The real and imaginary parts of ComplexInf are NaN. But, the conjugate of ComplexInf is still a complex

Table 10: Results of complex functions for complex(0.0, 0.0), ComplexInf, and ComplexNaN.

function	z value and results		
	complex(0.0, 0.0)	ComplexInf	ComplexNaN
sizeof(z)	8	8	8
abs(z)	0.0	Inf	NaN
real(z)	0.0	NaN	NaN
imaginary(z)	0.0	NaN	NaN
conjugate(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
polar(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
sqrt(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
exp(z)	complex(1.0, 0.0)	ComplexNaN	ComplexNaN
log(z)	ComplexInf	ComplexInf	ComplexNaN
log10(z)	ComplexInf	ComplexInf	ComplexNaN
ceil(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
floor(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
modf(z, &z2)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
z2	complex(0.0, 0.0)	ComplexInf	ComplexNaN
frexp(z, &z2)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
z2	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
ldexp(z, z2)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
sin(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
cos(z)	complex(1.0, 0.0)	ComplexNaN	ComplexNaN
tan(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
	Note: tan(complex($\pi/2 + k * \pi, 0.0$)) = ComplexInf		
asin(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
acos(z)	complex(pi/2, 0.0)	ComplexInf	ComplexNaN
atan(z)	complex(0.0, 0.0)	complex(pi/2, 0.0)	ComplexNaN
	Note: atan(complex($0.0, \pm 1.0$)) = ComplexInf; atan(ComplexInf, k) = complex(pi/2 + k*pi, 0.0)		
sinh(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
cosh(z)	complex(1.0, 0.0)	ComplexNaN	ComplexNaN
tanh(z)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
	Note: tanh(complex($0.0, \pi/2 + k * \pi$)) = ComplexInf		
asinh(z)	complex(0.0, 0.0)	ComplexInf	ComplexNaN
acosh(z)	complex(0.0, pi/2)	ComplexInf	ComplexNaN
atanh(z)	complex(0.0, 0.0)	complex(0.0, pi/2)	ComplexNaN
	Note: atanh(complex($\pm 1.0, 0.0$)) = ComplexInf; atanh(ComplexInf, k) = complex(0.0, pi/2 + k*pi)		

Table 11: Results of the function complex(x, y) for $0.0, \pm\infty$, and NaN.

complex(x, y)						
x value	y value					
	−Inf	−y1	0.0	y2	Inf	NaN
Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
x2	ComplexInf	complex(x2, −y1)	complex(x2, 0.0)	complex(x2, y2)	ComplexInf	ComplexNaN
0.0	ComplexInf	complex(0.0, −y1)	complex(0.0, 0.0)	complex(0.0, y2)	ComplexInf	ComplexNaN
−x1	ComplexInf	complex(−x1, −y1)	complex(−x1, 0.0)	complex(−x1, y2)	ComplexInf	ComplexNaN
−Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
NaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

infinity. The result **polar**(complex(0.0,0.0)) is defined as complex(0.0,0.0) because the principal value Θ for complex(0.0, 0.0) equals 0.0 as defined in Table 9. The result of **polar**(ComplexInf) is defined as complex(Inf, Inf). Therefore, if z equals complex(0.0,0.0) or ComplexInf, the equality of z = **polar**(real(polar(z)), imaginary(polar(z))) will still be satisfied. Like a real function, the square root of ComplexInf is ComplexInf.

As a real function, **exp**(Inf) = Inf whereas **exp**(−Inf) = 0.0. However, both values of ±Inf become ComplexInf if they are cast into complex numbers. Therefore, the complex exponential function **exp**(z) is ComplexNaN when the input argument is ComplexInf. The complex logarithmic function **log**(z) with the input argument of complex(0.0,0.0) or ComplexInf will return ComplexInf. With complex metanumbers as their input arguments, the real and imaginary parts of functions **ceil**(z), **floor**(z), and **ldexp**(z1, z2) are handled equivalent to two individual real functions. Like real functions, the complex trigonometric functions **sin**(z), **cos**(z), and **tan**(z) are undefined when the input arguments are ComplexInfs. The irrational number π is not representable in a computer program. If we had the value of π, the expression of **tan**($k\pi+\pi/2$) would return ComplexInf. Unlike real functions, the complex inverse trigonometric functions **asin**(z) and **acos**(z) return Complex-Infs when the input arguments are ComplexInfs. As an inverse function of **tan**(z), the function **atan**(z,k) has different branches when the first input value is ComplexInf. According to the definition, **atan**($\pm i$) equals ComplexInf. The results of complex hyperbolic functions **sinh**(z), **cosh**(z), and **tanh**(z), and complex inverse hyperbolic functions **asinh**(z), **acosh**(z), and **atanh**(z) are implemented similar to those of complex trigonometric functions and complex inverse trigonometric functions.

The results of the complex construction function **complex**(x,y) are given in Table 11. For constructing a complex number, if either its real or imaginary part is NaN, the result is a complex Not-a-Number. Likewise, if either one is a value of $\pm\infty$, the result is ComplexInf. For the function **polar**(r, theta) shown in Table 12, when the modulus is infinitely large, the resultant complex number is ComplexInf even if the provided argument of a complex number is infinity, which is compatible with the result of **polar**(ComplexInf) = complex(Inf, Inf). This also follows the rule that, through complex arithmetic and complex functions, one shall not get a complex number with its real or imaginary part being the value of −Inf, Inf, or NaN while the other part is a regular real number. Like the exponential function **exp**(z), the function **pow**(z1,z2) is undefined when the second argument is ComplexInf as is shown in Table 13. When the imaginary part y2 of z2 is a finite value, the results of the function depends on the value of its real part x2 when the value of z1 is complex(0.0, 0.0) or ComplexInf. Like the real function, the following expressions **pow**(complex(0.0,0.0), complex(0.0,0.0)), **pow**(complex(0.0,0.0), complex(0.0,y2)), **pow**(ComplexInf, complex(0.0,0.0)), and **pow**(ComplexInf, complex(0.0,y2)) are ComplexNaN.

Table 12: Results of the function polar(r, theta) for $0.0, \pm\infty$, and NaN.

polar(r, theta)						
r value	theta value					
	−Inf	−theta1	0.0	theta2	Inf	NaN
Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
r2	ComplexNaN	polar(r2, −theta1)	complex(r2, 0.0)	polar(r2, theta2)	ComplexNaN	ComplexNaN
0.0	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
−r1	ComplexNaN	polar(−r1, −theta1)	complex(−r1, 0.0)	polar(−r1, theta2)	ComplexNaN	ComplexNaN
−Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
NaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 13: Results of the function pow(z1, z2) for complex$(0.0, 0.0)$, ComplexInf, and ComplexNaN.

pow(z1, z2)						
z1 value	z2 value					
	complex(0.0, 0.0)	z2; ($\lvert y2 \rvert < \infty$)			ComplexInf	ComplexNaN
		$-\infty < x2 < 0.0$	x2 = 0.0	$0 < x2 < \infty$		
complex(0.0, 0.0)	ComplexNaN	ComplexInf	ComplexNaN	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
z1	complex(1.0, 0.0)	$z_1^{z_2}$	$z_1^{z_2}$	$z_1^{z_2}$	ComplexNaN	ComplexNaN
ComplexInf	ComplexNaN	complex(0.0, 0.0)	ComplexNaN	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Because **pow**(0.0, Inf) = 0.0 and **pow**(0.0, −Inf) = Inf, and both Inf and −Inf are considered as ComplexInf, **pow**(complex(0.0,0.0),ComplexInf) is defined as ComplexNaN. The results of function **fmod**(z1,z2) for complex metanumbers are given in Table 14.

5 Lvalue Related to Complex Numbers

As defined before that a lvalue is any object that occurs on the left hand side of an assignment statement. The valid lvalues related to complex numbers are listed in Table 15. The assignment operations `+=`, `-=`, `*=`, `/=`, as well as increment operation `++` and decrement operation `--` described in (Cheng, 1993) can be applied to all these lvalues. Besides the simple variable in case (1), an element of a complex array can be a lvalue which is case (2) in Table 15. In case (3), pointer to complex is used as a lvalue to get the memory or to point to a memory of a complex object. In case (4), the memory pointed to by the pointer *zptr* is assigned the value of the expression on the

Table 14: Results of the function fmod(z1, z2) for complex$(0.0, 0.0)$, ComplexInf, and ComplexNaN.

fmod(z1, z2)				
z1 value	z2 value			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
z1	ComplexNaN	fmod(z1,z2)	z1	ComplexNaN
ComplexInf	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 15: The valid lvalues related to complex numbers.

Case	Meaning of lvalue	Example
1	simple variable	`z = complex(1.0, 2);`
2	an element of a complex array	`zarray[i] = complex(1.0, 2)+ ComplexInf;`
3	complex pointer variable	`zptr = malloc(sizeof(complex) * 3;`
		`zptr = &z;`
4	address pointed to by a complex variable	`*zptr = complex(1.0, 2) + z;`
5	an element of a complex pointer array	`zarrayptr[i] = malloc(sizeof(complex) * 3;`
		`zarrayptr[i] = &z;`
6	address pointed to by an element of a complex pointer array	`*zarrayptr[i] = complex(1.0, 2);`
7	real part of a complex variable	`real(z) = 3.4;`
	real part of a complex variable	`real(*zptr) = 3.4;`
	real part of a complex variable	`real(*(zptr+1)) = 3.4;`
	real part of a complex variable	`real(*zarrayptr[i]) = 3.4;`
8	imaginary part of a complex variable	`imaginary(z) = complex(1.0, 2);`
	imaginary part of a complex variable	`imaginary(*zptr) = 3.4;`
	imaginary part of a complex variable	`imaginary(*(zptr+1)) = 3.4;`
	imaginary part of a complex variable	`imaginary(*zarrayptr[i]) = 3.4;`
9	float pointer variable	`fptr = &z;`
		`fptr = zptr;`
	pointer to real part of a complex variable	`*fptr = 1.0;`
	pointer to imaginary part of a complex variable	`*(fptr+1) = 2.0;`

right hand side of an assignment statement. In addition to a single pointer variable, one can have an array of complex pointers. Cases (5) and (6) show how an element of a complex 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 complex variable, or address pointed to by a complex pointer or pointer expression. A constant complex number or expression can be used as an input argument of the function **real**() only when it is a rvalue or an operand. In case (8), the imaginary part of a complex is accessed by the function **imaginary**() in the same manner as the function **real**(). Since a complex 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 *fptr* is a pointer to float. For cases (7)-(9), a real number, including ±0.0, ±Inf, and NaN, on the right hand side will be assigned to lvalue formally without filtering. Therefore, abnormal complex numbers such as complex(Inf, NaN), complex(Inf,0.0), etc. may be created. For example, two C^H commands **real**(z) = NaN and **imaginary**(z) = Inf make z equal to complex(NaN,Inf); and **real**(z) = −0.0 and **imaginary**(z) = NZero gives z the value of complex(−0.0, −0.0).

6 Creation of User's Complex Functions

User's complex functions in C^H can be created in the spirit of ANSI C, which will be demonstrated by the computation of the gamma function $\Gamma(z)$. $\Gamma(z)$ can be defined by the integral

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \tag{3}$$

One of approximation formulas for the function $\Gamma(z)$ derived by Lanczos (1964) is as follows

$$\Gamma(z) = (z+4.5)^{z-0.5} e^{-(z+4.5)} \sqrt{2\pi} \left(1.0 + \frac{76.18009173}{z} - \frac{86.50532033}{z+1}\right.$$

$$+\frac{24.01409822}{z+2} - \frac{1.231739516}{z+3} + \frac{0.120858003e-2}{z+4} - \frac{0.536382e-5}{z+5} + \varepsilon\Big); \qquad (4)$$

with the error smaller than $|\varepsilon| < 2 * 10^{-10}$. The above formula is valid for the complex gamma function in the half complex plane of $\mathbf{real}(z) > 0$. To avoid the overflow, $\log(\Gamma(z))$ rather than $\Gamma(z)$ is usually computed. For example, the logarithm of the above approximation formula for the gamma function with a float argument has been programmed by Press et al (1990) in C. The complex version of the logarithm of the gamma function can be easily programmed in $\mathrm{C^H}$ as follows.

```
complex gammalog(complex z)
{
  complex zz, sum;
  sum = 1.0 + 76.18009173/z - 86.50532033/(z+1) + 24.01409822/(z+2)
        -1.231739516/(z+3) + 0.120858003e-2/(z+4) -0.536382e-5/(z+5);
  zz = (z-0.5)*log(z+4.5) - (z+4.5) + log(sqrt(2*pi)*sum);
  return zz;
}
```

Using the above programmed external gamma function, the commands of

```
printf("gammalog(-2) = %f \n", gammalog(-2));
printf("gammalog(complex(1,2)) = %f \n", gammalog(complex(1,2)));
printf("gammalog(2) = %f \n", gammalog(2));
```

will produce the following output.
gammalog(-2) = complex(Inf,Inf)
gammalog(complex(1,2)) = complex(-1.876079,0.129647)
gammalog(2) = complex(0.000000,0.000000)
Note that the Gamma function gets ComplexInf at the singular point -2. For a real argument as in gammalog(2), the complex gamma function returns a complex number with an identically imaginary zero.

7 Rationale behind $\mathrm{C^H}$

This section will provide rationales for some of scientific programming features of $\mathrm{C^H}$. The following philosophies guided me through the design and implementation of language features presented in this paper and the companion paper (Cheng, 1993).

1. Follow conventional mathematics; define values for all operations and functions over the entire domain;

2. Preserve ANSI C and Fortran styles. The interpretation will follow C whenever there is a syntax conflict.

3. Make language easy to use.

4. Make language easy to implement.

These principles are reflected in many features of the $\mathrm{C^H}$ programming language. Some features like *polymorphism* of built-in functions and *optional arguments* for multiple-valued complex functions are obvious. Others are less obvious and even contentious, which are briefly explained in the following discussions.

7.1 No Unanticipated Complex Values

Unlike conventional languages, there is no reserved words in C^H. The *keyword changeability* in C^H makes the language very flexible. C^H is a programming environment, it can be configured at the programmer's discretion. Whenever appropriate, the decision in C^H will be made by the programmer, not the language implementor. This policy is also reflected in features related to complex numbers.

In some programming languages like Common Lisp and software packages such as Mathematica and MATLAB, values are typed, but variables are typeless. As a result, a function could return different data types even with same data types for input arguments. For example, a square root function may return a real value at one point as in **sqrt**$(9.0) = 3.0$ and may return a complex value at other point as in **sqrt**$(-9.0) = -i3$. In C^H, if x is a negative real number, function **sqrt**(x) will return NaN as described in (Cheng, 1993). At the first sight, their policy seems more generous than C^H's because functions like **sqrt**() can deliver some meaningful results. But, numerical computation experiences indicate that typeless variable is not a good language design for scientific programming as demonstrated by Kahan (1992). For example, the unanticipated creation of complex results due to roundoff errors in solving a nonlinear equation may not allow a numerical algorithm to find solutions in real numbers. Another serious problem for typeless languages such as MATLAB is that all computations are performed in double precision, which is apparently not applicable for many applications as discussed in (Cheng, 1993).

Following the lead of Fortran and C in scientific programming, C^H is also designed as a loosely typed language. Different data types can be mixed in arithmetic operations according to built-in data conversion rules as described in section 2.2. Unlike C, its standard mathematical functions can only return double value, functions in C^H can deliver different data types. But, the output data type of a function is still deliberated by the application programmer through the data types and numbers of the input arguments polymorphically. Extending definition of functions beyond their valid domains is the programmer's responsibility. The decision whether an expression delivers a real value or complex value are made by the application programmer, not by the C^H language implementor, which can be illustrated by numerically solving the quadratic equation $x^2+2x+2 = 0$. In (Cheng, 1993), we have shown that, if the results are restricted to real numbers, the solutions to this equation is NaNs. With a little modification, the following C^H program

```
complex root[2], a1;
float a,b,c;
a1=1; a = 1; b = 2; c = 2;
root[0] = (-b+sqrt(b*b-4*a1*c))/(2*a);
root[1] = (-b+sqrt(b*b-4*a1*c, 1))/(2*a);
printf("Solutions are %f and %.1f \n", root[0], root[1]);
```

will produce the output
Solutions are complex(-1.000000,1.000000) and complex(-1.0,-1.0)
Comparing with the program in (Cheng, 1993), one can see that integral value of 1 assigned to the complex variable a_1 is first cast to complex(1.0,0.0) internally. The complex variable a_1 then promotes the argument of **sqrt**() to a complex value, which finally results in a complex square root function in the calculation of $root[0]$. The complex result in $root[1]$ is achieved by switching the mode of function **sqrt**() through its auxiliary second argument directly.

7.2 Deliver Correct Numerical Values or NaN/ComplexNaN

In many computer systems, if operations such as 1/0.0 and **sqrt**(−9.0) are encountered, a computer program will be halted and the system will invoke exception routines to report invalid instructions such as *division by zero* or *domain error*. In $\mathrm{C^H}$, to ensure the correct flow of a program, all instructions will be executed. However, If a computer program at one point can deliver a correct numerical result while at other point it may deliver an erroneous result, the user will lose his/her confidence in the computer program immediately. To guarantee the delivery of correct numerical results, mathematically indeterminate expressions are defined as NaN in real operations and ComplexNaN in complex operations in $\mathrm{C^H}$. For example, 0^0 is defined as NaN in $\mathrm{C^H}$. If 0^0 is otherwise defined as 1 as suggested in (Kahan, 1986; Thomas, 1993), the expression of **pow**(x, 1/log(x)) with $x = 0$ would deliver 1.0; whereas $\lim_{x\to 0} x^{1/\log(x)}$ is e according to mathematical analysis. The result of ComplexNaN for **exp**(ComplexInf) is another example. In general, all built-in operations and functions in $\mathrm{C^H}$ will either deliver correct numerical results or NaN in real numbers and ComplexNaN in complex numbers if they are mathematically indeterminate.

7.3 Programming Complex Numbers over the Extended Finite Complex Plane

Most textbooks avoid issues related to complex infinity (Churchill and Brown, 1984; Marsden, 1973). Likewise, all currently existing general-purpose computer programming languages do not have provisions for consistent handling of complex infinity. For example, the recent proposed standard for Ada only spells out the behaviors when the absolute values for both real and imaginary parts of a complex number are less than or equal to FLT_MAX. In an effort to extend the IEEE 754 standard to complex arithmetic, Kahan (1986) and Tydeman (1991) explored the handling of complex numbers with components of $\pm 0.0, \pm\infty$, and NaN. In their proposed complex system, complex numbers are manipulated in the Cartesian plane, somewhat similar to the implementation of mathematical software package MATLAB. Pairs of real numbers such as $(\pm\mathrm{Inf}, y)$ and $(x, \pm\mathrm{Inf})$ with $-\mathrm{Inf} \le x \le \mathrm{Inf}$ and $-\mathrm{Inf} \le y \le \mathrm{Inf}$ are considered as valid complex numbers. It is true that these values can be represented by two floating-point data in a computer program. But, they are in conflict with the convention of mathematics in many situations. For example, according to complex analysis, there is only *one* complex infinity in an extended complex plane which corresponds to the north pole of the Riemann sphere as shown in Figure 1. Addition of two complex infinities is indeterminate and division of complex infinity by zero is complex infinity. $\mathrm{C^H}$ is in full compliance with mathematical conventions regarding complex numbers. Therefore, (Inf, Inf) + (Inf, Inf) is defined as (NaN, NaN) and (Inf, Inf)/(0.0, 0.0) as (Inf, Inf) in $\mathrm{C^H}$. However, if real and imaginary parts of a complex number are treated as two completely separate objects, according to the arithmetic operation rules given in Table 1, (Inf, Inf) + (Inf, Inf) will become (Inf,Inf) and (Inf,Inf)/(0.0,0.0) will become (NaN, NaN) as given in (MathWorks, 1990; Tydeman, 1991). Similarly, according to complex analysis, complex numbers such as (NaN, NaN), (NaN, y), (x, NaN), (Inf, NaN), (NaN, Inf), (Inf, y), and (x, Inf), (−Inf, NaN), (NaN, −Inf), (−Inf, y), and (x, −Inf) are not valid complex numbers. Therefore, complex operations and complex functions in $\mathrm{C^H}$ shall not produce these abnormal complex numbers. We try to make $\mathrm{C^H}$ simple. In $\mathrm{C^H}$, there is no negative NaN because not-a-number is not a number as is discussed in (Cheng, 1993). For the same reason, there is no need for having so many different formats of complex-not-a-number although these formats can be stored in complex data. There are serious problems for allowing existence of these abnormal complex data in a computer program. For example, let $z = 0 + i\infty$, if one attempts to get $z * z = -\infty + i0$, the result may end up with complex(0.0, Inf)∗complex(0.0, Inf) = complex(−Inf, NaN), not complex(0.0, Inf)∗complex(0.0, Inf) = complex(−Inf, 0.0) (Kahan and Thomas, 1991).

After getting numerous such surprising results during the testing of the implemented $\mathrm{C^H}$ complex features, we decided to use ComplexInf and ComplexNaN in $\mathrm{C^H}$. These two complex metanumbers make the implementation of a whole complex system much simpler. In $\mathrm{C^H}$, there is only *one* Complex-Infinity in the extended finite complex plane and *one* Complex-Not-a-Number, which follows mathematical conventions with respect to the complex number and Riemann sphere. This will avoid the delivery of a complex number with its real or imaginary part being Inf, −Inf, or NaN while the other part has a different value in complex operations and complex functions. The rules for coercion of two real numbers into ComplexInf and ComplexNaN are given in Table 11. The only way to get abnormal complex numbers such as complex(Inf, NaN) is by assigning real metanumbers to the address of a float pointer variable which points to the memory of a complex variable, or use functions **real**(z) and **imaginary**(z) as lvalues, as shown in section 5. The memory accessibility, which is inherited from C, makes many impossible tasks in other languages possible in $\mathrm{C^H}$.

7.4 Distinguish −0.0 from 0.0 in Real Numbers, not in Complex Numbers

In the domain of real numbers, there are ±Inf and ±0.0 which are very useful for scientific programming as is shown in (Cheng, 1993). Shall the sign of zeros also be honored in complex numbers as in real numbers. The immediate answer seems to be "yes". As illustrated by Kahan (1986), the sign of zeros can be useful in complex numbers, especially for handling of branch cuts of multiple-valued complex functions. Indeed, we tried to implement complex operations and complex functions which would treat 0.0 and −0.0 as two different objects. It turns out that , if 0.0 and −0.0 were treated as two different objects as in real numbers (Cheng, 1993), not only would the implementation of the language become a very difficult task, but also programming of the language would be very tedious. The programmer would have to struggle with the sign of zeros. It would be almost impossible to write a program without constantly consulting a manual because everything would be so complicated. The programmer may lose the sight of the forest for trees. For example, the square root of the complex zero is only defined as **sqrt**(complex(0.0,0.0)) = complex(0.0, 0.0) in $\mathrm{C^H}$, irrespective of the sign of zeros. The same function in a language with the sign of zeros being respected may be defined as **sqrt**(complex(0.0, 0.0)) = complex(0.0, 0.0), **sqrt**(complex(−0.0, 0.0)) = complex(0.0, 0.0), **sqrt**(complex(0.0, −0.0)) = complex(0.0, −0.0), **sqrt**(complex(−0.0, −0.0)) = complex(0.0, −0.0) as is listed in (Squire, 1991a), depending on the implementation. As one can see that, for a function with multiple input arguments, the definition will be even more complicated. Although the proposed standard for Ada provides some guidelines for handling the sign of zeros in complex numbers, many critical issues related to the implementation of such a system have not been addressed in the documentations.

Implementing complex data with a respected sign of zeros is not as easy as in real numbers. There are some tradeoffs have to be compromised. In the spirit of ANSI C, real numbers and complex numbers can be mixed in arithmetic operations and elementary functions in $\mathrm{C^H}$. For example, r+complex(x, y) becomes complex(r+x, y+0.0) in $\mathrm{C^H}$ at its current implementation. For the convenience of implementation, a real number is first coerced into a complex number with a zero imaginary part prior to addition operation. If y is −0.0, its sign will be coerced such that r+complex(x, −0.0) becomes complex(r+x,0.0). The minus sign of y will be lost in the addition operation, and other arithmetic operations and functions. The sign of a zero has a serious affect on results. For example, **sqrt**(complex(−4.0, −0.0) = complex(0.0, −2.0) whereas **sqrt**(complex(−4.0, 0.0) = complex(0.0, 2.0) in a complex system with signed zeros. Of course, there is no difficulty to implement the addition of a real number and a complex number in r+complex(x, y) as complex(r+x, y) without data coercion so that the sign of the imaginary part of the complex number will be preserved. But, how do we handle arithmetic operations for a pair of imaginary and complex numbers? Most computer languages such as Fortran, Ada, Common Lisp, etc. treat a complex number as a pair of real numbers. An imaginary number $z = iy$ is conventionally represented as a complex number complex(0.0, y) or (0.0, y) in a computer program, i.e., an imaginary number is treated as a complex number with zero real part both internally and externally. Such handling of imaginary numbers cannot prevent data coercions. Hence, in order for a language to effectively handle the sign of zeros in a complex system, a new imaginary data type is needed (Kahan and Thomas, 1991), which will result in a completely new syntax and semantics of the language. The new language will have a style different from ANSI C and Fortran. For example, a complex number will be created by $x + i * y$ where i is an imaginary data; multiplication of an imaginary number and a real number delivers an imaginary number, and addition of (real)+(imaginary) will be promoted to a complex number rather than actually add two operands. This kind of language is not difficult to implement; but, experienced C programmers will have to be adapted to a new language paradigm. To preserve the ANSI C and Fortran styles and ease the implementation and programming, we choose to do not distinguish 0.0 from $-0,0$ for complex numbers in $\mathrm{C^H}$. This is why 0.0 and -0.0 in relational operations in $\mathrm{C^H}$ are considered the same in $\mathrm{C^H}$ (Cheng, 1993). Otherwise, they would be regarded as two different objects in comparison operations because, in real operations and real functions, the sign of zeros does make a difference.

It should be pointed out in passing that, in Fortran, a complex number is represented as a pair of real numbers. For example, (3.0, 4.0) in Fortran stands for 3.0+i4.0. This representation of a complex number is intentionally avoided in the design of $\mathrm{C^H}$ because not only complex number, but also dual number are basic data types in $\mathrm{C^H}$. A dual number also consists of a pair of real numbers. A dual number is defined as $x + \epsilon y$ with $\epsilon^2 = 0$. For consistent handling of aggregate data types, complex and dual numbers are created by complex constructor **complex**() and dual constructor **dual**(), respectively. These data constructors are also refereed to as explicit type conversion functions. Details about the handling of dual number and its applications in mechanical systems analysis and design can be found in (Cheng, 1993b).

In $\mathrm{C^H}$, the rules for the determination of the sign of a zero resultant when it is produced and rules for the use of the sign of a zero operand or argument have been spelled out for real numbers in (Cheng, 1993). Although the sign of zeros are not honored in complex numbers in $\mathrm{C^H}$, the sign of zeros will be carried over when signed zeros of real numbers are converted to complex numbers either implicitly or explicitly. For the convenience of implementation, many complex operations and functions are implemented through real operations and real functions which treat -0.0 and 0.0 as two different objects. For example, the $\mathrm{C^H}$ expression complex(-0.0, -0.0)+ complex(0.0, -0.0) actually delivers complex(0.0, -0.0). In many applications, real and imaginary parts of a complex zero are used as real numbers for real operations. It seems that it is necessary to keep track of the sign of complex zeros when it is generated. However, to make programmers' life easier, when -0.0 of real or imaginary part of a complex value is coerced into a real number either implicitly or explicitly, the sign of a zero will be discarded as described in section 2.2, which simplifies mixed mode application significantly. The programmer shall not worry about the sign of each complex zero delivered by complex operations and complex functions. However, if one must carry the sign of a zero in a complex value over to a real number, pointing a pointer-to-float to the memory location of the complex variable can achieve this goal.

Real numbers have two infinities $+\infty$ and $-\infty$, and the origin in a real line can be approached through both positive and negative directions represented by 0.0 and -0.0, respectively. Unlike real number, there is only one complex infinity and the origin of the complex plane can be reached in any directions 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 in the range of $-\pi < \theta \leq \pi$. Therefore, it seems that distinction of the sign of zeros only along the real and imaginary axes does not make much sense. If the origin is approached from directions other than the Cartesian coordinate axes, points in the complex plane will be obtained by functions like **polar**(r, theta) instead of complex(x, y); then, roundoff errors introduced in the computations of **polar**(r, theta) and other operations and functions will overpower the sign of zeros. If the one-to-one correspondence between the origin and infinity of the complex plane is concerned, there is certainly no need for recognition of the sign of zeros.

7.5 The Principal Value Θ lies in the Range of $-\pi < \Theta \le \pi$

But, unlike complex operations, complex functions are much more complicated than real functions. It is branch cuts that make complex functions so complicated. The sign of zeros can play a role in the computation of complex functions along branch cuts. There is a general consensus about where the slits should be placed for commonly used complex mathematical functions (Penfield, 1981; Kahan, 1986; Steele, 1990; Squire, 1991a,b). The slits for functions **sqrt**(), **log**(), and **pow**() are along the negative real axis. The slits for **asin**(), **acos**(),and **atanh**() are the real axis excluding the segment between -1 and 1. Similarly, the slits for functions **atan**() and **asinh**() lie on the imaginary axis not between $-i$ and i. The slit for function **acosh**() is along the real axis where $z \le 1$. Since all slits of elementary functions lie on either the real or imaginary axis, Kahan (1986) proposed that every point z along a slit be represented in two ways: once with a +0.0 and once with -0.0 for whichever the real and imaginary parts of z vanishes. This can be easily achieved by the complex constructor such as complex(x, ±0.0) or complex(±0.0, y) in $\mathrm{C^H}$. Accordingly, the principal value Θ of the argument of a complex number should be in the range of $-\pi \le \Theta \le \pi$. Although defining the principal value Θ in this range is not conventional, it does provide a nice treatment for branch cuts, especially, many identities for real numbers can be preserved for complex numbers along slits as well. But, if a slit does not lie on the Cartesian coordinate axis, such as in the conformal map $dz/dw = (w + \alpha)/(w^2 - \beta)$ or $z = w^\beta/(1 - \beta)$ with $\alpha > 0$ and $0 < \beta < 1$ (Kober, 1957), the roundoff errors will be inevitably introduced during computations so that the sign of zeros intended for handling the slits can be easily lost. In the situation like this, there is no payoff for the distinction of 0.0 and -0.0.

The handling of branch cuts in $\mathrm{C^H}$ is similar to what was proposed for APL by Penfield (1981). On the branch cut, the function is "clockwise continuous" in the sense that the discontinuity associated with the branch cut occurs just below or left of the cut so that the function is continuous with values above or right of the slit. Under this treatment of branch cuts, the principal value of the argument of a complex number will be in the range of $-\pi < \Theta \le \pi$, which follows the convention of mathematics (Churchill and Brown, 1984; Marsden, 1973). In $\mathrm{C^H}$, there is no distinction of 0.0 and -0.0 for components of a complex number, every point z on a slit is represented with 0.0 or -0.0 for whichever of the real and imaginary parts of z vanishes. The discontinuity of a slit along the real axis can be represented by setting the imaginary component of a complex number to −FLT_MINIMUM or −FLT_MIN which are the maximum denormalized and normalized negative numbers (Cheng, 1993), respectively, depending on if the computer system is in conformance with the IEEE 754 standard or not. If it is an IEEE machine, −FLT_MINIMUM can be used; otherwise, FLT_MIN should be used. Similarly, the discontinuity of a slit along the imaginary axis can be treated by setting the real part of a complex number to −FLT_MINIMUM or −FLT_MIN. For example, the principal value of $\mathrm{Log}(z) = \log(\sqrt{x^2 + y^2}) + i\Theta$ can be obtained by **log**(z) or **log**(z,0) in $\mathrm{C^H}$ as given in Table 8. This logarithm function is a single-valued function defined over the extended finite complex plane. This single-valued function is not analytic in its domain $r > 0, -\pi < \Theta \le \pi$. When z lies on the negative real axis with z =complex($-x$, ±0.0), Θ is π; whereas just below the real axis of FLT_MINIMUM, i.e., z =complex($-x$, −FLT_MINIMUM), Θ

is near $-\pi$. For example, **log**(complex(−1.0, −FLT_MINIMUM)) becomes complex(0.0, −pi). Due to the finite representation, the system constant pi will be used in C$^{\rm H}$. When $-\pi < \Theta \leq \pi$, each branch of a multiple-valued function will have unique value. For example, in C$^{\rm H}$, **log**(complex(−1.0, 0.0)) = complex(0.0, π) and **log**(complex(−1.0, 0.0), 1) = complex(0.0, $3 * \pi$). If the sign of zeros was respected and $-\pi \leq \Theta \leq \pi$, the above expression would be evaluated as **log**(complex(−1.0, 0.0)) = complex(0.0, π), **log**(complex(−1.0, −0.0)) = complex(0.0, $-\pi$), **log**(complex(−1.0, 0.0), 1) = complex(0.0, 3π), and **log**(complex(−1.0, −0.0), 1) = complex(0.0, π). Different branches **log**(complex(−1.0, 0.0)) and **log**(complex(−1.0, −0.0), 1) would have the same result of complex(0.0, π).

In C$^{\rm H}$, all familiar identities will be satisfied in the domain where the concerned functions in an identity are analytic. Some identities will not hold if the values of functions lie on the slit. For example, $\log(1/z) = -\log(z)$ will not be valid if z is on the negative real axis because $\log(1/(-x)) = \mathrm{Log}(1/x) + i\pi = -\mathrm{Log}(\mathrm{x}) + i\pi$, whereas $-\log(-x) = -\mathrm{Log}(x) - i\pi$. As another example, entire functions like $z^2 + 1$ and $\sin(z)$ satisfy the reflection identity of $w(\overline{z}) = \overline{w(z)}$ over the extended finite complex plane, but not functions like $z^2 + 1$ and iz. The Principle of Reflection requires that the function $w(z)$ be analytic in its domain in order to hold the identity of $w(\overline{z}) = \overline{w(z)}$ (Churchill and Brown, 1984). Indeed, identities like $\sin(\overline{z}) = \overline{\sin(z)}$ and $\overline{z}^2 + 1 = \overline{z^2 + 1}$ are valid in C$^{\rm H}$ over the extended finite complex plane, including z = ComplexInf and z = ComplexNaN. But, the identity of $\sqrt{\overline{z}} = \overline{\sqrt{z}}$ may not hold if $z < 0$ because the square root function has a slit along the negative real axis. For comparison, if a complex system distinguishes 0.0 from −0.0 with $-\pi \leq \Theta \leq \pi$, these two identities can be preserved along the slits (Kahan, 1986). However, if identities along slits really matter, C$^{\rm H}$ has mechanisms for these equivalent identities. Using the above examples, if $z < 0$, the following two C$^{\rm H}$ identities along the slits are valid: **conjugate**(**sqrt**(complex($-x$, 0.0), k) = **sqrt**(**conjugate**(complex($-x$, 0.0)), $k-1$), **log**(1/complex($-x$, 0.0), k) = − **log**(complex($-x$, 0.0), $-k-1$)) where k is an integral value as a branch indicator. However, programmers are cautioned that the C$^{\rm H}$ expression **conjugate**(**sqrt**(complex($-x$, 0.0), k) == **sqrt**(**conjugate**(complex($-x$, 0.0)), $k-1$) will always evaluate to TRUE whereas **log**(1/complex($-x$, 0.0), k) == − **log**(complex($-x$, 0.0), $-k-1$)) may not return TRUE because of roundoff errors as in any other computer languages. As demonstrated in these examples, values of different branches of a multiple-valued function can be easily obtained in C$^{\rm H}$.

If one wants to distinguish −0.0 from 0.0 in complex numbers and place Θ in the range of $-\pi \leq \Theta \leq \pi$, he/she can achieve this objective through C$^{\rm H}$'s real operations and real functions in which the sign of zeros is respected and real function **atan2**() will readily provide the value for Θ. Although the programmer cannot directly access the function overloading features in C$^{\rm H}$ at its current implementation, one can replace built-in mathematical functions by external functions through functions `addkey()`, `chkey()`, and `remkey()` (Cheng, 1993). For this experimentation purpose, when **real**() and **imaginary**() are used as lvalues, the sign of zeros of rvalues will be honored as described in sections 2.2 and 5. Program examples using **real**() and **imaginary**() as lvalues with signed zeros will be given in section 7.7.

7.6 $F(x + i0) = F(x) + i0$, If x Is within the Valid Domain of $F(x)$

Taking roundoff errors into consideration, the sign of zeros in complex number appears an accuracy issue. In many cases, the fiddling of ±0.0 in the proposed complex system (Kahan, 1986) and fiddling of ±FLT_MINIMUM or ±FLT_MIN in C$^{\rm H}$ have essentially the same perturbation effect. Most important for a complex system is to preserve the correct sign for the results of complex operations and functions so that an intended branch of a multiple-valued function can be invoked.

If a language is not carefully implemented, the sign of zeros and FLT_MINIMUM both can be lost. From a language implementation point of view, preserving the sign of a result is less demanding than preserving both sign and zero in complex operations and complex functions. It is even more true from an application programmer's point of view. In modern languages such as Fortran, Common Lisp, and proposed complex systems (Kahan, 1986; Tydeman, 1992), some commonly used complex functions are defined or implemented in terms of other complex operations and complex functions. This is not only computationally inefficient, but also may introduce some potential errors. For example, **atan**(x+i0.0), ideally, should deliver a result whose imaginary part is identically zero. If it is implemented in terms of complex operations and complex functions, due to roundoff errors of the complex implementation, its imaginary part may not be identically zero. Propagation of this non-zero imaginary part may lead to severe problems near branch cuts. Complex operations and functions in $\mathrm{C^H}$ are defined in the conventional mathematical forms. No function is defined in terms of complex trigonometric and hyperbolic functions. Although formulas given in Tables 1 and 8 may not be the direct implementation, they do indicate that all operations and functions in $\mathrm{C^H}$ are achieved through real operations and real functions. Quite a few algorithms for complex operations and complex functions are available [4,7,11,12, 13,14,17,18, 23,25,30,32,34,38]. But, due to the unique design of the $\mathrm{C^H}$ programming language, some of these algorithms have to be slightly modified. The detailed description of the implementation of algorithms used in $\mathrm{C^H}$ is beyond the scope of this paper. A point should be made is that a complex number in $\mathrm{C^H}$ is stored internally in a Cartesian format rather than in a polar form which involves trigonometric functions. When the imaginary part of a complex operand or a complex argument is identically zero, $\mathrm{C^H}$ will deliver a complex result with an identically zero imaginary part, if mathematically possible, as demonstrated in section 6. As another example, the computation of **sqrt**(complex$(0.0, 0.6) - 6*$complex(6, 0.1)) will surely deliver complex(0.0, 6.0) with identically zero real part, not complex(0.0, -6.0). $\mathrm{C^H}$ is designed to be deterministic. There is no fuzzy around branch cuts. Essential to the built-in complex functions are the principal phase angle Θ. When a point $z = x + iy$ lies slightly below the negative real axis, Θ is $-$pi so long as y in the $\mathrm{C^H}$ expression `y < -FLT_MINIMUM` evaluates to TRUE, which is critical for many multiple-valued functions with branch cuts.

7.7 Application Examples Involving Branch Cuts

So long as built-in complex operations and complex functions are appropriately implemented, complicated problems can be solved using these language primitives. This section will be concluded by solving two complex conformal map problems. Kahan (1986) showed that these two problems are difficult to handle by a computer language which does not respect the sign of zeros in complex numbers. Through these two interesting examples, we will illustrate how $\mathrm{C^H}$ which does not respect the sign of zeros in complex numbers can conveniently solve complex problems which involve branch cuts. We will also demonstrate how to handle signed zeros in complex numbers in a $\mathrm{C^H}$ program.

Example 1

The first example is the conformal map $w(z) = z - i\sqrt{iz+1}\sqrt{iz-1}$ which maps the complex $z-$plane of $z = x + iy$ onto the w-plane of $w = u + iv$. Note that function $w(z)$ has a slit along the imaginary axis from $-i$ to i which will be mapped onto the unit circle $|w| = 1$. Vertical lines in the z-plane are mapped to the stream-lines of a vertical flow around the unit circle in the w-plane. A $\mathrm{C^H}$ program that can calculate stream-lines in the w-plane is given below:

```
FILE *stream; complex i, z, w; float x, y;
stream = fopen("diskflow.out","w");
i = complex(0,1); x = -0.5;
```

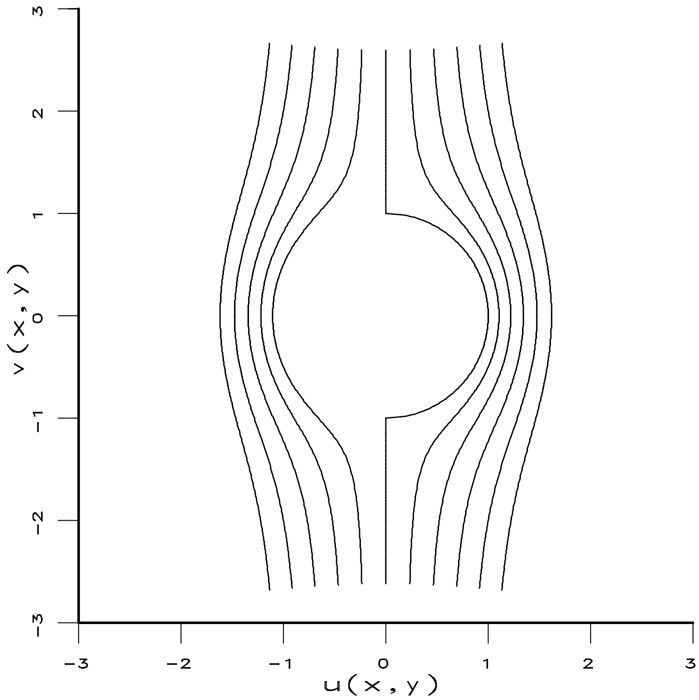

Figure 3: The flow around an unit disk without $x = -$FLT_MINIMUM.

```
while(x <= 0.5)
{
  y = -1.5;
  while(y <= 1.5)
  {
    z = complex(x, y);
    w = z - i* sqrt(i*z+1)*sqrt(i*z-1);
    fprintf(stream, "%f %f \n", real(w), imaginary(w));
    y += 0.01;
  }
  x += 0.1;
}
fclose(stream);
```

The program runs vertical lines in the z-plane with y from -1.5 to 1.5 and x from -0.5 to 0.5, both at the interval of 0.1. The stream-lines in the w-plane through the output file `diskflow.out` produced by execution of the above $\mathrm{C^H}$ program is shown in Figure 3. Note that the slit $-1 < z^2 < 0$ is mapped to the right-hand arc of the circle by $w = iy - i\sqrt{-y+1}\sqrt{-y-1} = -iy + \sqrt{-y^2+1} = u + iv$. In order to get the left-hand arc of the circle, the following $\mathrm{C^H}$ code fragment should be added to the above program before the last statement `fclose(stream)`.

```
x = -FLT_MINIMUM;
y = -1.5;
while(y <= 1.5)
```

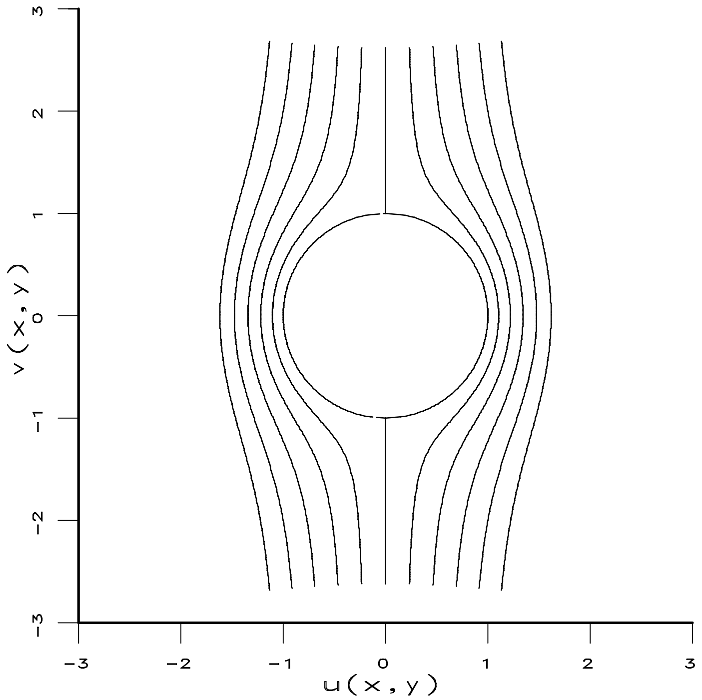

Figure 4: The flow around an unit disk with $x = -$FLT_MINIMUM.

```
{
  z = complex(x, y);
  w = z - i* sqrt(i*z+1)*sqrt(i*z-1);
  fprintf(stream, "%f %f \n", real(w), imaginary(w));
  y += 0.01;
}
```

With this additional vertical line $x = -$ FLT_MINIMUM in the z-plane, the corresponding image in the w-plane is shown in Figure 4, where the dashed line corresponds to the vertical line $x = -$ FLT_MINIMUM. The images in the w-plane for vertical lines $x = 0.0$ and $x = -$ FLT_MINIMUM with $|y| > 1$ in the z-plane are overlapped in Figure 4. Note that $x = -$ FLT_MINIMUM in the program can be replaced by any small negative numbers such as $-$FLT_MIN or -0.0001; graphically, the result will be the same as Figure 4. It appears that the singularity of the slit along the imaginary axis between $1 < z^2 < 0$ is the source of the problem because, as shown in Figure 4, the images for the segments $|y| > 1$ along the imaginary axis in the z-plane are well-behaved.

In the above program, if x is set to -0.0 in complex(x,y) which is equivalent to complex(0.0, y) in a $\mathrm{C^H}$ program, the left-hand arc of the unit circle cannot be obtained. The picture will be the same as Figure 3. As pointed out in section 7.5, one can experiment with signed zeros in complex numbers in a $\mathrm{C^H}$ program which honors the sign of zeros of real numbers. The following program will illustrate the handling of signed zeros in complex numbers.

```
FILE *stream;
complex i,z,w;
float x,y;
```

```
int sign(float f);
stream = fopen("diskflow2.out","w");
complex SQRT(complex c)
{
  float *fp;
  complex comp;
  if(c == ComplexInf)
    return ComplexInf;
  fp = (float *)&c;     /# fp points the real part of c
  real(comp) = sqrt((abs(c)+real(c))*0.5);
  imaginary(c) = sign(*++fp)*sqrt((abs(c)-real(c))*0.5);
  return comp;
}
int sign(float f)       /# sign() works for char, int,
                        /# double or complex inputs also
{
  if(f<0.0 || isnegzero(f))
    return -1;
  else
    return 1;
}
i = complex(0.0, 1.0);
x = -0.0;
y = -1.5;
while(y <= 1.5)
{
  z = complex(x,y);
  w = z-i*SQRT(i*z+1)*SQRT(i*z-1);
  fprintf(stream, "%f %f \n", real(w), imaginary(w));
  y += 0.01;
}
fclose(stream);
```

When **real**() and **imaginary**() are used as lvalues, the sign of zeros of rvalues will be preserved. In the above program, the sign of zeros in the complex square root function `SQRT()` is implemented according to the definition given in (Friedland, 1980; Kahan, 1986; Squire, 1991b; Tydeman, 1992). The square root function is essentially defined as $\sqrt{r}(\cos\frac{\Theta}{2} + i\sin\frac{\Theta}{2})$ with $-\pi \le \Theta \le \pi$. The results of complex zeros are defined as SQRT(complex($\pm 0.0, 0.0$)) = complex($0.0, 0.0$) and SQRT(complex($\pm 0.0, -0.0$)) = complex($0.0, -0.0$). In addition, SQRT(ComplexInf) is ComplexInf and SQRT(ComplexNaN) equals ComplexNaN. The output of the above $\mathrm{C^H}$ program is the file `diskflow2.out`. The vertical flow in the complex w-plane mapped from the vertical line $x = -0.0$ in the complex z-plane is shown in Figure 5 with data from file `diskflow2.out`. Note that the horizontal straight-line is introduced purely due to graphical interpolation. This straight connection line is not mapped from points in the z-plane. Although the square root function is implemented with a proper design for the sign of zeros, the result is still not correct. The reason is that the multiplication $i * z$ of an imaginary number with a complex number in function `SQRT()` has coerced the sign of zeros in z as discussed in section 7.4. If we replace the statement `w = z - i*SQRT(i*z+1)*SQRT(i*z-1)` by `w = z-i*SQRT(complex(1-y, -0.0))*SQRT(complex(-1-y,`

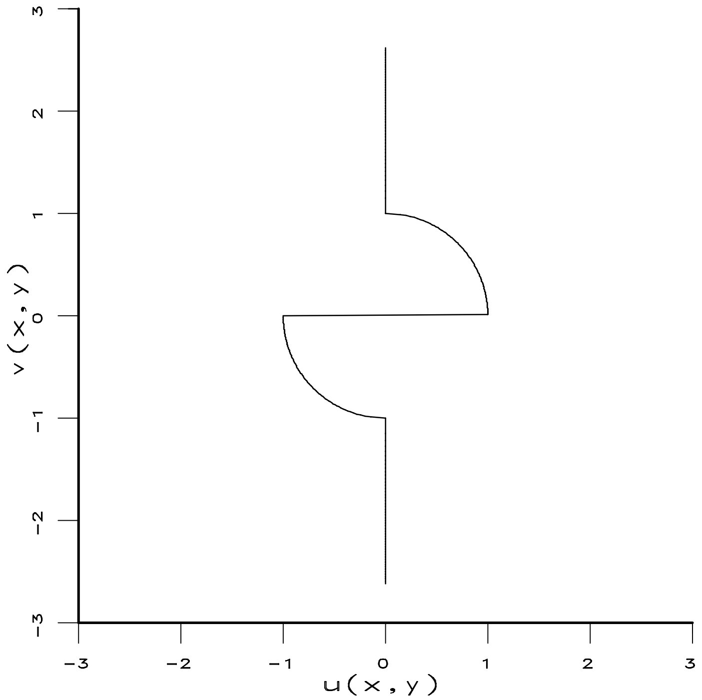

Figure 5: The flow around an unit disk for $x = -0.0$ with coercion of the sign of zeros

`-0.0))`, the correct vertical flow can be obtained. The corresponding conformal map is shown in Figure 6. According to the definitions for real operations and functions with signed zeros given in (Cheng, 1993), the difference of these two statements can be shown by two sample points. For point $z = \mathbf{complex}(-0.0, 0.36)$, the C$^{\mathrm{H}}$ expression `z - i*SQRT(i*z+1)*SQRT(i*z-1)` is computed as follows

$$\begin{aligned}
w &= (-0.0, 0.36) - (0.0, 1.0)\sqrt{(0.0, 1.0) * (-0.0, 0.36) + 1}\sqrt{(0.0, 1.0) * (-0.0, 0.36) - 1} \\
&= (-0.0, 0.36) - (0.0, 1.0)\sqrt{(-0.36, 0.0) + 1}\sqrt{(-0.36, 0.0) - 1} \\
&= (-0.0, 0.36) - (0.0, 1.0)\sqrt{(0.64, 0.0)}\sqrt{(-1.36, 0.0)} \\
&= (-0.0, 0.36) - (0.0, 1.0) * (0.8, 0.0) * (0.0, 1.166) \\
&= (0.933, 0.36)
\end{aligned}$$

whereas the C$^{\mathrm{H}}$ expression `z-i*SQRT(complex(1-y, -0.0))*SQRT(complex(-1-y, -0.0))` is computed by

$$\begin{aligned}
w &= (-0.0, 0.36) - (0.0, 1.0)\sqrt{(0.64, -0.0)}\sqrt{(-1.36, -0.0)} \\
&= (-0.0, 0.36) - (0.0, 1.0) * (0.8, -0.0) * (0.0, -1.166) \\
&= (-0.933, 0.36)
\end{aligned}$$

For point $z = \mathbf{complex}(-0.0, -0.36)$, the first expression is computed as follows

$$\begin{aligned}
w &= (-0.0, -0.36) - (0.0, 1.0)\sqrt{(0.0, 1.0) * (-0.0, -0.36) + 1}\sqrt{(0.0, 1.0) * (-0.0, -0.36) - 1} \\
&= (-0.0, -0.36) - (0.0, 1.0)\sqrt{(0.36, -0.0) + 1}\sqrt{(0.36, -0.0) - 1}
\end{aligned}$$

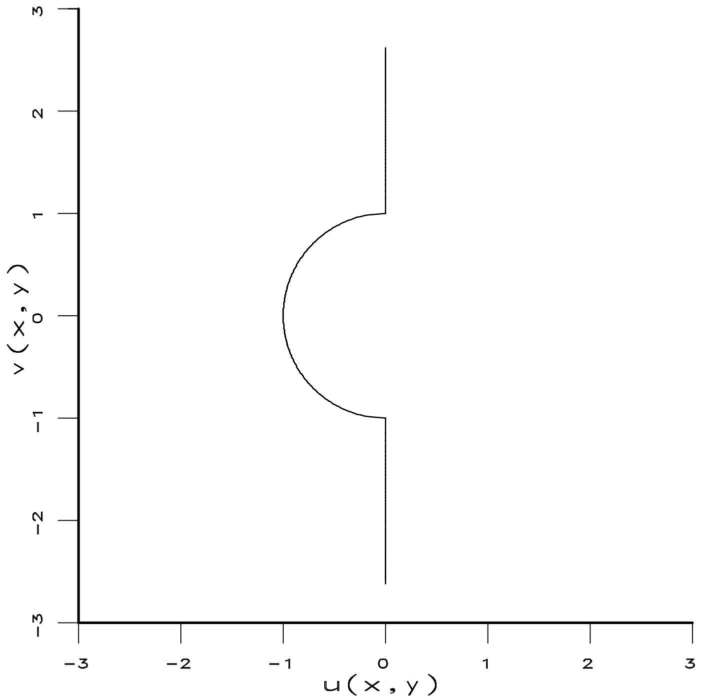

Figure 6: The flow around an unit disk for $x = -0.0$.

$$
\begin{aligned}
&= (-0.0, -0.36) - (0.0, 1.0)\sqrt{(1.36, 0.0)}\sqrt{(-0.64, -0.0)} \\
&= (-0.0, -0.36) - (0.0, 1.0) * (1.166, 0.0) * (0.0, -0.8) \\
&= (-0.933, -0.36)
\end{aligned}
$$

whereas the second expression is computed by

$$
\begin{aligned}
w &= (-0.0, -0.36) - (0.0, 1.0)\sqrt{(1.36, -0.0)}\sqrt{(-0.64, -0.0)} \\
&= (-0.0, -0.36) - (0.0, 1.0) * (1.166, -0.0) * (0.0, -0.8) \\
&= (-0.933, -0.36)
\end{aligned}
$$

In this simple example, we are able to avoid the coercion of sign of zeros by simplifying the multiplication of $i * z$ manually. This kind of reformulation will be quite difficult when a problem is complicated. Note that the coercion of sign of zeros by the imaginary number i outside the square root function will not make a difference in conformal mapping because no branch cuts are involved. With an appropriate design for the sign of zeros in complex operations and complex functions, the left-hand arc of the unit circle can also be obtained using $x = -0.0$ in a program. Therefore, one may also attribute much of the misbehavior in Figure 3 to the lack of recognition of the sign of zeros in a computer programming language (Kahan, 1986).

Example 2

The second example also deals with conformal map. If a liquid is forced by high pressure to jet into a slot, stream-lines will be formed around the opening of slot. The conformal map $w(z) = 1 + z^2 + z\sqrt{z^2+1} + \log(z^2 + z\sqrt{z^2+1})$ which will map radial straight-lines, including the imaginary axis, in the right half-plane onto these stream-lines. The following $\mathrm{C^H}$ program can compute these stream-lines with data saved in file `slotflow.out`.

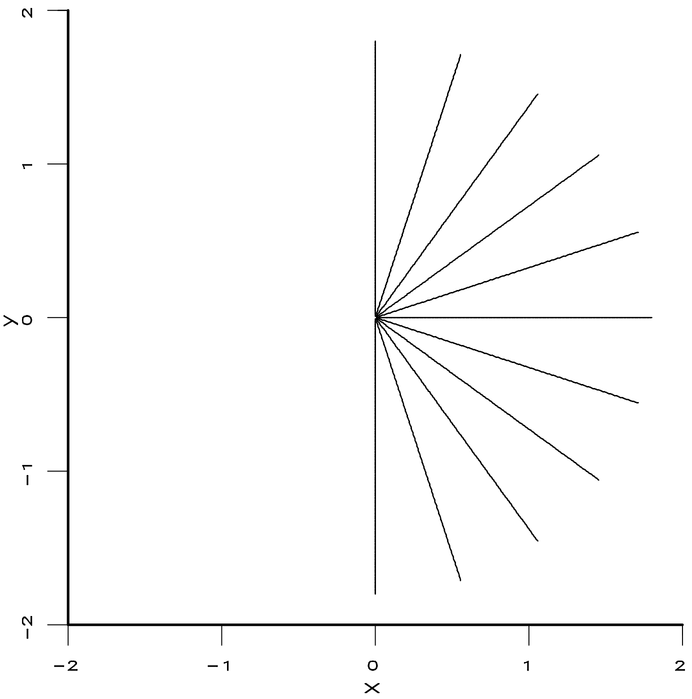

Figure 7: The radial straight-lines in the z-plane.

```
FILE *stream; complex z, w; float r; int k;
stream = fopen("slotflow.out","w");
k = -5;
while(k <= 5)
{
  r = 0.01;
  while(r <= 1.8)
  {
    z = polar(r, (pi/10)*k);
    w = 1 + z*z +z*sqrt(z*z +1) +log(z*z +z *sqrt(z*z +1))
    fprintf(stream, "%f %f \n", real(w), imaginary(w));
    r += 0.01;
  }
  k++;
}
fclose(stream);
```

The radial straight-lines in the z-plane and corresponding images in the w-plane are shown in Figures 7 and 8, respectively. Points along radial straight-lines in the z-plane are generated by the function **polar**(r,theta), where r runs from 0.01 to 1.8 with a step size of 0.01 and *theta* from $-\pi/2$ to $\pi/2$ with a step size of $\pi/10$. If r in the above program is 0, w will be ComplexInf; correspondingly, **real**(w) and **imaginary**(w) become NaNs. It appears in Figure 8 that complex infinities in the w-plane are located in $(u, v) = -\infty + iy$ with $-\pi/2 \le y \le \pi/2$ for different stream-lines; but, there is actually only one complex infinity. The origin of the z-complex plane is mapped to ComplexInf in the extended finite complex plane in an *one-to-one* manner with w(complex(0.0,

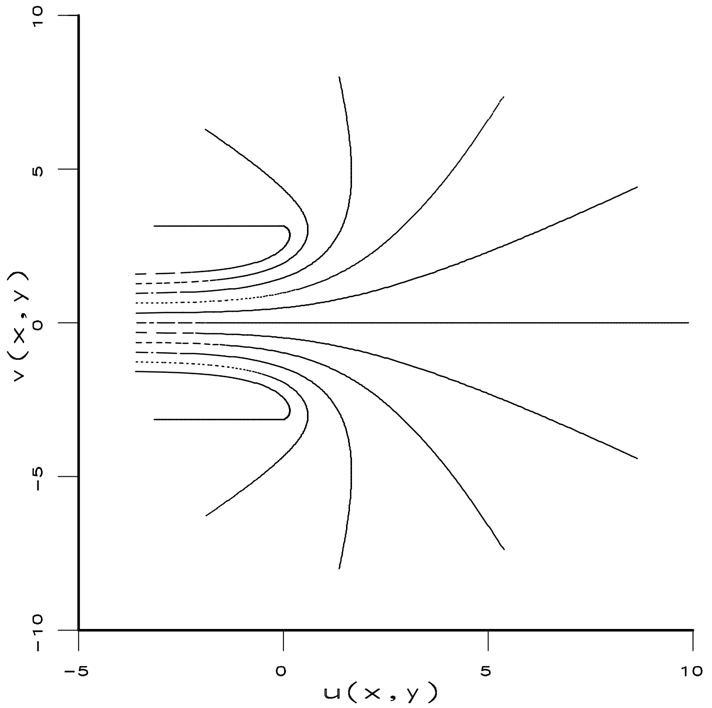

Figure 8: The flow forced into a slot in the w-plane with z = polar(r, θ) or z =complex(FLT_MINIMUM, y)

0.0)) = ComplexInf. In $\mathrm{C^H}$, all points on the w-plane with their real parts less than $-$FLT_MAX are treated as ComplexInf as shown in Figure 2. The picture of $w(z)$ is symmetrical about the real axis because $w(\overline{z}) = \overline{w(z)}$. The conformal map can be viewed as follows. Along the imaginary axis with $x = 0$, as y running from $-\infty$ through -1 toward 0, and then from 0 through 1 toward ∞, the image in the w-plane runs from left to right along the lower wall and back along the lower free boundary of the jet toward infinity, then from the infinity to the upper free boundary, finally, from left to right along the upper free boundary and back along the upper wall. Interestingly, lower and upper free boundaries and other stream-lines can be considered to merge at the infinity of ComplexInf in the w-plane which corresponds to the origin of complex(0.0,0.0) in the z-plane.

Note that there are slits in the imaginary axis with $z^2 < -1$ beyond $\pm i$. Points generated by `z = polar(r, pi/2)` may not be exactly on the imaginary axis due to roundoff errors. If we are cautious, we may want to examine the behavior of this conformal map without the effect of roundoff errors introduced by the **polar**() function. To suppress the contribution of the perturbation effect of roundoff errors, points along the imaginary axis in the z-plane can be generated by `z = complex(0.0, y)`. When the two radial lines along the imaginary axis are calculated by z =complex(0.0, y) with y running from 0.01 to 1.8 and from -0.01 to -1.8, the corresponding images in the w-plane are plotted in Figure 9. Like the previous example, the lower wall, part of the image, disappears, which can be explained as follows. The walls $v = i\pi$ and $v = -i\pi$ in the w-plane are the images of points in the imaginary axis $z = iy$ with $y > 1$ and $y < -1$, respectively. The curve forward from $v = i\pi$ then back to $w = -\mathrm{Inf} + i\pi/2$ is the image of the segment of $z = iy$ with $0 < y < 1$. When $z = -iy$ with $y > 1$, $w(z)$ becomes $1 - y^2 + y\sqrt{y^2-1} + \log(y^2 - y\sqrt{y^2-1}) + i\pi$ which will map the slit on the lower part of the imaginary axis onto the tiny segment of the upper wall as is shown in Figure 10. This result holds even if the sign of zeros are respected with

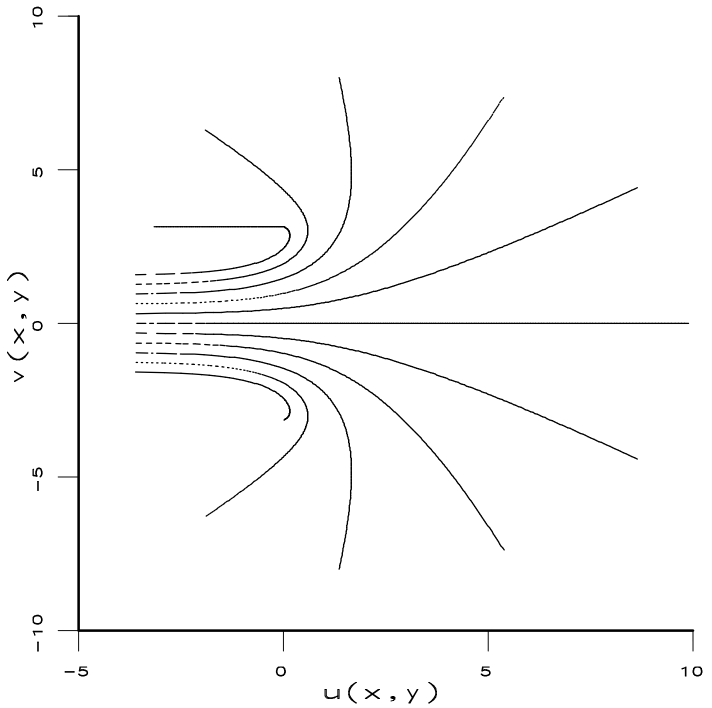

Figure 9: The flow forced into a slot in the w-plane with z = complex(0.0, y) for the imaginary axis in the z-plane.

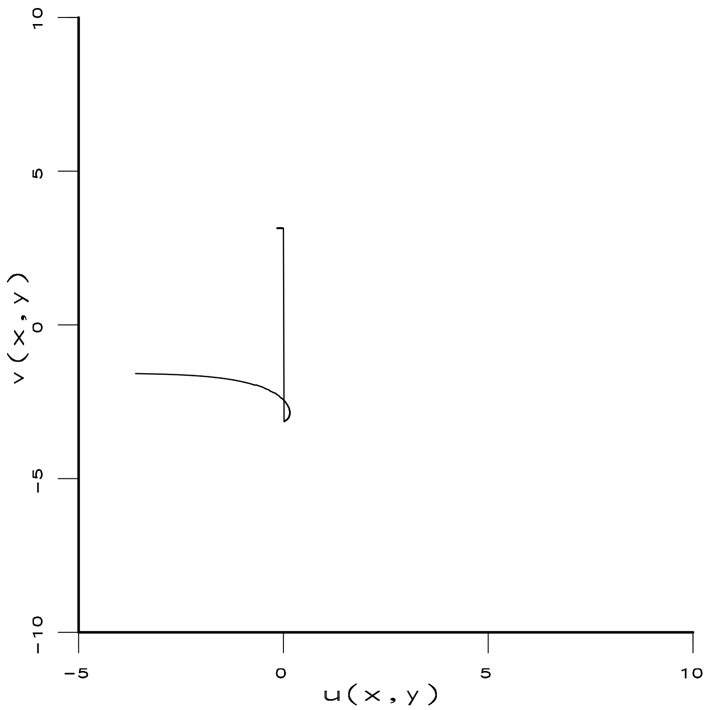

Figure 10: The true image in the w-plane for the negative imaginary axis in the z-plane.

$-\pi \leq \Theta \leq \pi$ as proposed in (Kahan, 1986). This phenomenon is due to the singularity along the slit. Note that, like the horizontal line in Figure 5, the vertical straight-line, which connects the tiny segment in $v = i\pi$ and the stream-line in the lower part of Figure 10, is produced by graphical interpolation. This straight connection line is not mapped from points in the z-plane. Nevertheless, a small perturbation of x, when y runs from $-\infty$ to -1, can bring back the missing lower wall. How tiny should x be is a nontrivial question in most computer languages as pointed out by Kahan (1986). In $\mathrm{C^H}$, however, there is a definitive answer to this question. If x for the imaginary axis in the program is set to any small positive number that is larger than or equal to FLT_MINIMUM, i.e. z =complex(FLT_MINIMUM, y), the missing lower wall in Figure 9 can be recovered. It is intersting to point out that, if the conformal mapping function is chosen as $w(z) = 1+z^2+z\sqrt{z+i}\sqrt{z-i}+\log(z^2+z\sqrt{z+i}\sqrt{z-i})$, the tiny segment of the wall that appears in Figure 10 will become a full wall, but other behaviors like locations of slits remain the same.

The above two examples demonstrate that conformal maps may misbehave along slits. However, a small perturbation normally will bring back the missing information. Decreasing the step size in a loop of a program may also recover the critical information, which resembles the handling of numerical solutions for stiff differential equations in some aspects.

8 Conclusions

Complex numbers are generalization of real numbers. Complex numbers are manipulated in the same manner as real numbers in $\mathrm{C^H}$. Since complex is implemented as a basic data type in the $\mathrm{C^H}$ programming language, the numerical computation can be handled in a much integrated fashion. $\mathrm{C^H}$ treats floating-point real numbers with signed zeros and complex numbers with unsigned zeros as well as Not-a-Number and infinities in an integrated consistent manner.

For scientific computing with complex numbers, the extended finite complex plane along with complex metanumbers of ComplexZero, ComplexInf, and ComplexNaN are introduced in this paper. The I/O for complex numbers and data conversion rules between real numbers and complex numbers are defined. The results of complex arithmetic and relational operations, and polymorphic mathematical functions with complex metanumbers as input arguments are also defined. These may be different from the results obtained directly according to their definitions for regular complex numbers. Besides the polymorphic nature in which the algorithms and resultant data types of arithmetic operations depend on the data types of operands, and the algorithms and resultant data types of mathematical functions are related to the input arguments; mathematical functions in $\mathrm{C^H}$ can have a variable number of arguments. In case a function becomes a complex function, additional integral arguments in comparing with its real counterpart indicate which branch of a multiple-valued complex function shall be invoked. It is the first time, I believe, that such a simplicity is introduced in a general-purpose computer programming language for scientific computing with complex numbers. Example programs show that the external complex functions in $\mathrm{C^H}$ can be programmed in the same syntax of the ANSI C and treated as if they were the internally built-in functions.

The rationales for the decision of the designed features, related to complex numbers in $\mathrm{C^H}$, have been provided from language design, implementation, and application points of views. We have demonstrated, in $\mathrm{C^H}$ programs, the effective handling of branch cuts of multiple-valued complex functions which are difficult to handle for most other computer programming languages.

9 Acknowledgement

The author would like to thank referees for their thoughtful comments which improved the quality of this paper and its companion paper (Cheng, 1993);

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. Baker, H., Less Complex Elementary Functions, *ACM SIGPLAN Notices*, vol 27, No. 11, Nov, 1992, pp. 15-16.
5. Cheng, H. H., Scientific Computing in the C^H Programming Language, Department of Mechanical, Aeronautical and Materials Engineering, University of California, Davis, Technical Report TR-MAME-93-101, February 18, 1993; also *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 49-75.
6. 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, 1993b, vol. 2, pp. 73-80.
7. Collens, D. S., Algorithm 243 Logarithm of a Complex Number, *Collected Algorithm from CACM*, Volume II, 1980.
8. Churchill, R. V. and Brown, J. W., Churchill, R. V. and Brown, J. W., *Complex Variables and Applications*, Fourth edition, McGraw-Hill Book Co., NY, 1984.
9. Dritz, K. W., Proposed Standard for a Generic Package of Elementary Functions for Ada, *Ada Numerics Standardization and Testing*, *ACM Ada Letters*, Vol. XI, No. 7, Fall, 1991, pp. 9-46.
10. IEEE, *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1985.
11. Friedland, P., Algorithm 312 Absolute Value and Square Root of a Complex Number, see Collens 1980.
12. Herndon, J. R., Algorithm 46 Exponential of a Complex Number, *Collected Algorithm from CACM*, Volume I, 1980a.
13. Herndon, J. R., Algorithm 48 Logarithm of a Complex Number, see Herndon 1980b.
14. Herndon, J. R., Algorithm 53 Nth Roots of a Complex Number, see Herndon 1980c.
15. Hodgson, G. S., Proposed Standard for Packages of Real and Complex Type Declarations and Basic Operations for Ada (including Vector and Matrix Types), see Dritz 1991, 1991a, pp. 91-130.
16. Hodgson, G. S., Rationale for Proposed Standard for Packages of Real and Complex Type Declarations and Basic Operations for Ada (including Vector and Matrix Types), see Dritz 1991, 1991b, pp. 131-139.
17. Johnson, M. L. and Sangren, W., Algorithm 106 Complex Number to a Real Power, see Herndon 1980a.
18. 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.

19. Kahan, W., Invalid Operations Deemed Advantageous, personal communication, Oct. 9, 1992.
20. Kahan, W. and J. W. Thomas, Augmenting a Programming Language with Complex Arithmetic, personal communication, Dec. 16, 1991.
21. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, first edition (K&R C), 1978; second edition (ANSI C), 1988.
22. Kober, H., *Dictionary of Conformal Representations*, Dover Publications, Inc., 1957.
23. Kuki, H., Algorithm 421 Complex Gamma Function with Error Control, see Herndon 1980a.
24. Lanczos, C., A Precision Approximation of the Gamma Function, *J. SIAM Numerical Analysis*, Ser. B, vol. 1, 1964, pp. 86-96.
25. Lucas, C. W., Jr. and Terrill, C. W., Algorithm 404 Complex Gamma Function, see Herndon 1980a.
26. Marsden, J. E., *Basic Complex Analysis*, W. H. Freeman and Company, San Francisco, 1973.
27. MathWorks, *Pro-MATLAB User's Guide*, The MathWorks, Inc., 1990.
28. Penfield, P., Jr., Principal Values and Branch Cuts in Complex APL. *APL Quote Quad*, vol. 12, No. 1, Sept. 1981.
29. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1990.
30. Relph, A. P., Algorithm 190 Complex Power, see Herndon 1980a.
31. Ritchie, D. M. and Thompson, K. L., The Unix Time-Sharing System, *Commun. ACM*, vol. 17, No. 7, July 1974, pp. 365-375.
32. Smith, R. L., Algorithm 116 Complex Division, see Herndon 1980a.
33. Squire, J. S., Proposed Standard for a Generic Package of Complex Elementary Functions, see Dritz 1991, 1991a, pp. 140-165.
34. Squire, J. S., Rationale for the Proposed Standard for a Generic Package of Complex Elementary Functions, see Dritz 1991, 1991b, pp. 166-179,
35. Steele, G. L., Jr., *Common Lisp: the Language*, 2nd edition, Bedford, MA, Digital Press, 1990.
36. Stroustrup, B., *The C^{++} Programming Language*, Addison-Wesley, Publishing Company, Inc., 1987.
37. Tydeman, F., Merging Complex and IEEE-745, Document X3J11.1 (NCEG) 92-061.
38. Van de Riet, R. P., Algorithm 186 Complex Arithmetic, see Herndon 1980a.
39. Wolfram, S., *Mathematica: a System for Doing Mathematics by Computer*, Addison-Wesley Pub. Co., 1988.