*Dr. Dobb's Journal*, March 2002, pp. 28-34.

# C99 & Numeric Computing

## — Scientific Computing in Real and Complex Domains

Harry H. Cheng

### 1 Introduction

Although C is a powerful systems programming language that can deliver much the same control over devices as assembly language, it has deficiencies when it comes to scientific and engineering applications that require extensive numerical computing. While some numerical computing deficiencies in the original K&R C [1] were addressed in C90 [2], many limitations still remain. C99, ratified as the ANSI/ISO C Standard (ISO/IEC IS 9899) [3], is a milestone in C's evolution into a viable programming language for scientific and numerical computing. Among other features, C99 supports IEEE floating-point arithmetic, complex numbers, and variable-length arrays (VLAs) for numerical programming. Complex numbers and VLAs were added mainly based on the prior art of implementation of Ch from SoftIntegration (http://www.softintegration.com) [4], SCC from Cray Research, (http://www.cray.com), gcc from Free Software Foundation (http://www.gnu.org), and some others.

Although C99 support is limited, more compilers are adding these new features. For example, Comeau C 4.2.45.2 from Comeau Computing (http://www.comeaucomputing.com/features.html#c99) supports VLAs without complex numbers. The C compiler from Hewlett-Packard supports complex numbers. GCC 2.95 and later (http://gcc.gnu.org/c99status.html) provides limited support of VLAs and complex numbers. The Dinkum C99 Library from Dinkumware (http://www.dinkumware.com/) is a complete library for C99.

Complex numbers are handled as built-in data types in C, Unlike C, complex numbers are treated as classes in C++. For example, Forte C++ 6 Update 2 (formerly Sun Visual WorkShop C++; http://www.sun.com/forte/cplusplus/index.html) provide some support for complex arithmetic. There is no provision for IEEE floating-point arithmetic for both real numbers and complex numbers in C++.

In this article, I examine what's involved with C99 compliance by looking at Ch, a C virtual machine produced by SoftIntegration (http://www.softintegration.com/). Ch has provisions for consistent handling of numerical numbers in the entire real and complex domains with VLAs under the framework of IEEE floating-point arithmetic. As a superset of C interpreter, Ch also supports classes in C++. In particular, I focus here on the design and implementation of IEEE floating-point arithmetic and complex numbers. Ch was designed for script computing in the framework of C/C++. C or C++ conforming programs with complex numbers will run in this virtual machine without modification. I've tested all C programs presented here in both Ch and GCC 2.96, and C++ programs in Ch and G++ 2.95.

### 2 Computing in the Entire Real Domain

The IEEE 754 standard [5] for binary floating-point arithmetic is significant for consistent floating-point arithmetic with respect to real numbers. The IEEE 754 standard distinguishes +0.0 from

$-0.0$, which introduces an extra complexity for programming. Another important feature of the IEEE 754 standard is the internal representations for the mathematical infinities and invalid value. The mathematical infinity $\infty$ is represented by the symbol of Inf. A mathematically indeterminate or an undefined value such as division of zero by zero is represented by NaN, which stands for Not-a-Number. Many computer hardware support signed zeros, infinity, and NaN. Information about low-level and limited high-level instruction sets provided by hardware vendors may not be relevant to the application programmer and most features of a final system depend on the software implementation. Even for IEEE machines, if there is no provision for propagating the sign of zeros, infinity, and NaN in a consistent and useful manner through the software support, they will have to be programmed as if zeros were unsigned, without infinity and NaN. Based on IEEE machines, some vendors provide software support for the IEEE 754 standard through libraries. However, these special values in libraries are not transparent to the programmer. Although the application of symbols such as Inf and NaN can be found in some mathematical software packages and libraries, their handling of these special numbers is often full of flaws. It is in these grey areas that the IEEE 754 standard is not supported in many hardware and software systems.

To make the power of the IEEE 754 standard easily available to the programmer, the floating-point numbers of INFINITY, −INFINITY, NAN and signed zeros $-0.0$ and 0.0 are introduced in C99. In Ch, INFINITY and NAN corresponding to built-in metanumbers Inf and NaN are defined as macros in header file math.h. For notational convenience and brevity, metanumbers Inf and NaN will be used in the presentation in the rest of this article. These metanumbers are transparent to the programmer. Signed zeros $+0.0$ and $-0.0$ in C99 behave like correctly signed infinitesimal quantities $0_+$ and $0_-$, whereas symbols Inf and −Inf correspond to mathematical infinities $\infty$ and $-\infty$, respectively. The IEEE 754 standard only addresses the arithmetic involving these metanumbers. These metanumbers are extended in C99 to commonly used mathematical functions in the spirit of the IEEE 754 standard. There are provisions for consistent handling of metanumbers in I/O, arithmetic, relational and logic operations, and polymorphic mathematical functions in the implementation of Ch [6]. An NaN is propagated consistently through subsequent computations. Many believe that C99 made mistakes in handling some of mathematical functions. For example, the values of function calls for hypot(Inf, NaN), hypot(-Inf, NaN), pow(1, NaN), and pow(NaN, +/-0.0) are defined in C99 as Inf, Inf, 1.0, and 1.0, respectively. They are implemented in Ch to return NaN because these functions are mathematically undefined for the arguments with the above mentioned values.

C99 distinguishes $-0.0$ from 0.0 for real numbers. The metanumbers 0.0, $-0.0$, Inf, −Inf, and NaN are very useful for scientific computing. For example, the function $f(x) = e^{\frac{1}{x}}$ is not continuous at the origin as shown in Figure 1. This discontinuity can be handled gracefully in C99. The evaluation of the expression $\mathbf{exp}(1/0.0)$ will return Inf and $\mathbf{exp}(1/(-0.0))$ gives 0.0, which corresponds to mathematical expressions $e^{\frac{1}{0_+}}$ and $e^{\frac{1}{0_-}}$ or $\lim_{x\to 0_+} e^{\frac{1}{x}}$ and $\lim_{x\to 0_-} e^{\frac{1}{x}}$, respectively. In addition, the evaluation of expressions $\mathbf{exp}(1.0/\text{Inf})$ and $\mathbf{exp}(1.0/(-\text{Inf}))$ will get the value of 1.0. As another example, the function `finite(x)` recommended by the IEEE 754 standard is equivalent to the expression `-Inf < x && x < Inf`, where $x$ can be a float/double variable or expression. If $x$ is a float, `-Inf < x && x < Inf` is equivalent to `-FLT_MAX <= x && x <= FLT_MAX`; if $x$ is a double, `-Inf < x && x < Inf` is equivalent to `-DBL_MAX <= x && x <= DBL_MAX`; The mathematical statement "*if* $-\infty <$ *value* $<= \infty$, *then y becomes* $\infty$" can be easily programmed as follows

```
if(-Inf < value && value <= Inf) y = Inf;
```

However, a computer can only evaluate an expression step by step. Although the metanumbers are limits of the floating-point numbers, they cannot replace mathematical analysis. For example,

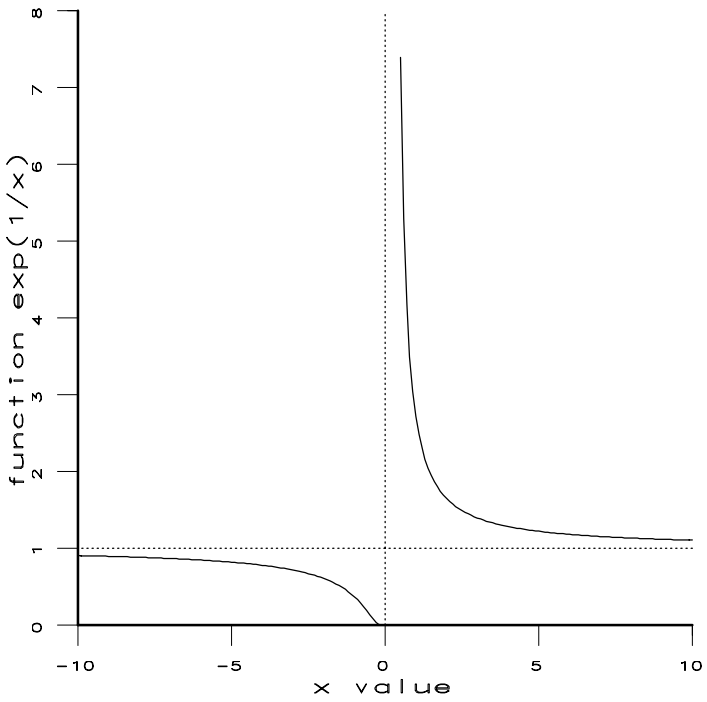


Figure 1: Function $f(x) = e^{\frac{1}{x}}$.

the natural number $e$ equal to 2.718281828... is defined as the limit value of the expression

$$\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x = e.$$

But, the value of the expression **pow**(1.0 + 1.0/Inf, Inf) is NaN. The evaluation of this expression is carried out as follows:

$$\left(1.0 + \frac{1.0}{Inf}\right)^{Inf} = (1.0 + 0.0)^{Inf} = 1.0^{Inf} = NaN$$

If the value FLT_MAX instead of Inf is used in the above expression, the result is obtained by

$$\left(1.0 + \frac{1.0}{FLT\_MAX}\right)^{FLT\_MAX} = (1.0 + 0.0)^{FLT\_MAX} = 1.0^{FLT\_MAX} = 1.0$$

Because metanumber NaN is unordered, a program involving relational operations should be handled cautiously. For example, the expression `x > y` is not equivalent to `!(x <= y)` if either x or y is a NaN. As another example, the following code fragment

```
if(x > 0.0)
    function1();
else
    function2();
```

is different from the code fragment

```
if(x <= 0.0)
    function2();
else
    function1();
```

The second `if`-statement should be written as `if(x <= 0.0 || isnan(x))` in order to have the same functionality for these two code fragments.

The metanumbers 0.0, −0.0, Inf, −Inf, and NaN are very useful for applications in engineering. For example, the discontinuity at the origin can be expressed using signed zeros. The infinity of mechanical advantage at a toggling position for a four-bar linkage can be written as Inf. If there exists no solution for output link corresponding to a given input link position of a four-bar linkage, the solution can be represented symbolically as NaN, which will be described in detail later.

## 3 Computing in the Entire Complex Domain

*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. 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* $(0, 1)$ with $i * i = -1$, complex numbers can be mathematically represented as

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

Complex number has wide applications in engineering and science. 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 has provided complex data type since its earliest days. Computations involving complex numbers can be accomplished using a data structure in C90. However, programming with such a structure is somewhat clumsy, because a corresponding function has to be invoked for each operation. Using C++, classes 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. Such complex classes have been added in the C++ standard. Without overloading features of classes, C99 supports complex numbers with built-in data types. There are differences for complex numbers in C and C++. However, a typical C or C++ program with complex numbers can run in Ch without any modification. A complex number $z = (x, y) = re^{i\theta}$ can be constructed by `x+I*y`, `complex(x,y)`, `polar(r,theta)`, `float_complex(x,y)`, or `double_complex(x,y)` in Ch. Complex numbers are supported in generic polymorphic mathematical functions in Ch.

Most textbooks and software avoid issues related to the complex infinity. Likewise, no other general-purpose computer programming language has provisions for consistent handling of complex infinity. Primarily, there are two models for implementation of complex numbers under the framework of the IEEE floating-point arithmetic. One is to follow the conventional mathematics, which will be described later. The other is invented in informative Annex G in C99 with imaginary types. In this model, all these metanumbers (+-Inf, x) (x,+-Inf), (+-Inf, +-0.0) (+-0.0, +-Inf), (+-Inf, NaN), and (NaN, +-Inf) can represent a complex infinity. The sign of zeros is honored. Not-a-number is no longer not a number. NaN is essentially treated as *any number*. Non-conventional numerical results are invented.

There are major problems in this model.

- First, many results in Annex G are pure invention. they are mathematically incorrect according to the conventional complex analysis. For example, it is a well known textbook example in complex analysis that exp(ComplexInf) is undefined, because the complex infinity can be approached from different directions, which lead to different numerical results. But, Annex G invented exp(-Inf, NaN)==(+-0,+-0), exp(Inf, NaN)==(+-Inf,NaN), exp(-Inf, Inf) == (+-0,+-0), exp(Inf, Inf)==(Inf, NaN), etc. In this model, real and imaginary parts of a complex number is treated as a whole in some cases. In other cases, they are treated as if they were completely independent. For example, imag(complex(NaN, x)) will gives x.

- Second, it has to introduce a new data type of imaginary in the type system to achieve the afore-mentioned non-conventional numerical results. In fact, three new data types float imaginary, double imaginary, and long double imaginary have to be added to the type system. It is very expensive to add a type system to C. Many operators and type conversion rules have to be overloaded internally, which will slow compilation and execution speed even for programs without using imaginary types.

- Third, the new imaginary type is not orthogonal with other types. The set of imaginaries is closed under addition but not multiplication, so it isn't a proper number system under the field operations. They do not form a self-contained subspace. For example, the square root of an imaginary does not return a pure imaginary. Very few if any real-world applications use pure-imaginary numbers alone. Imaginaries are typically embedded in the whole complex plane. Nevertheless, additional functions are needed to handle mathematical functions with arguments of imaginary type.

- Forth, it complicates complex arithmetic operations and complex functions dramatically, which will lead to inefficiency of C for numerical computing.

- Fifth, this model will also shift the burden of keeping track of sign of zeros to users. It is almost impossible to get the correct computing results without consulting a manual constantly, especially for complex functions with multiple arguments. If the sign of zero is ignored in complex numbers, a programmer can ignore the sign of zeros. If the sign is honored, a programmer however has to worry about the sign of zeros.

Most importantly, many inventions such as exp(-Inf, NaN)==(+-0,+-0) in Annex G have no practical use in engineering and science. These inventions, against the conventional complex mathematical analysis, are even harmful as shown in the example below.

```
double x, y, r;
double complex z;
double tolerance = DBL_EPSILON;
/* ... */
z = cexp(complex(x/y, sqrt(r));
if(cabs(z) < tolerance)
    move_needle_one_inch_for_robot_brain_surgery();
else
    error_handling_for_some_thing_wrong();
```

If it happens that the numerical values for x is any positive number such as 3.0, y is -0.0, and r is NaN at a singularity point of a robot. The above code works fine for implementation such as

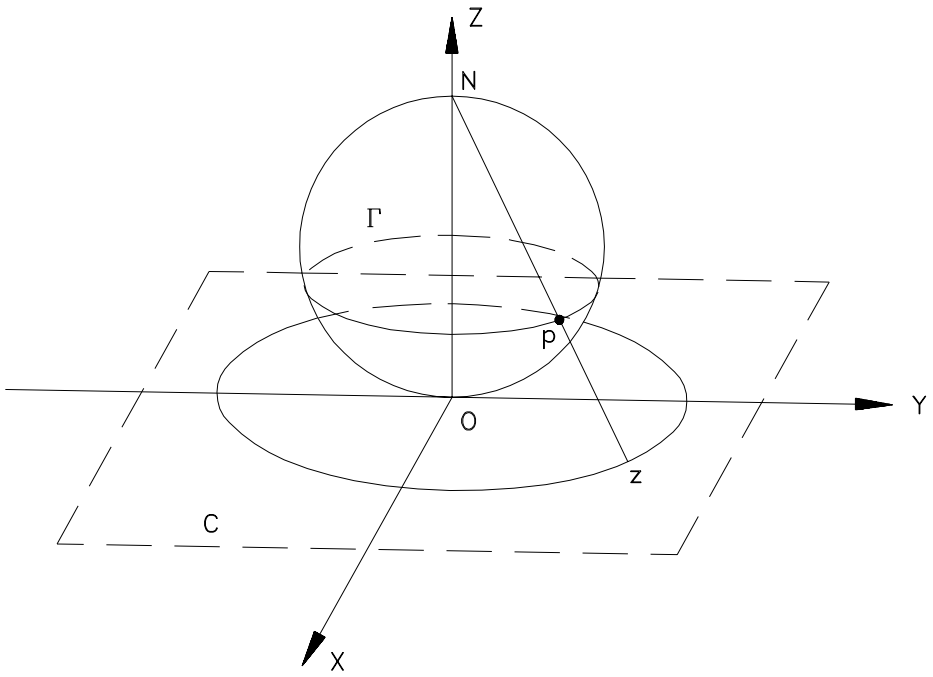


Figure 2: The Riemann sphere Γ and extended complex plane.

Ch based on an alternative complex model described in next paragraphs. Based on the invention in Annex G, a patient's brain might be damaged. Propagating NaN to a valid, but erroneous, numerical result according to Annex G has a serious consequence for applications in engineering and science.

In short, Annex G is neither normative nor mandated for C99-conforming implementations. Consequently, C99-conforming implementation of Ch uses a different complex model in the spirit of IEEE 754 floating-point arithmetic. This model follows the conventional complex analysis under a Riemann sphere. In this model, there is only *one* complex infinity (Inf, Inf)==ComplexInf in the extended complex domain, *one* Complex-Not-a-Number (NaN, NaN)==ComplexNaN, and zero is unsigned. This complex model is easy to implement and use.

In conventional complex analysis under a Riemann sphere, complex numbers can be represented in the extended complex plane shown in Figure 2. In Figure 2, there is a one-to-one correspondence between the points on the Riemann sphere Γ and the points on the extended complex plane $\boldsymbol{C}$. 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.

Because of the finite representation of floating-point numbers, complex numbers in Ch are programmed in *the extended finite complex plane* shown in Figure 3. Any *complex* or *float complex* values inside the ranges of $|x| <$ FLT_MAX and $|y| <$ FLT_MAX are representable in finite floating-point numbers. Variable $x$ is used to represent the real part of a complex number and $y$ the imaginary part; FLT_MAX, a predefined system constant in header file float.h, is the maximum representable finite floating-point number in the float data type. For *double complex*, the boundary for the extended finite complex plane will be expanded from FLT_MAX to DBL_MAX. Outside the extended finite complex plane, a complex number is treated as a Complex-Infinity represented

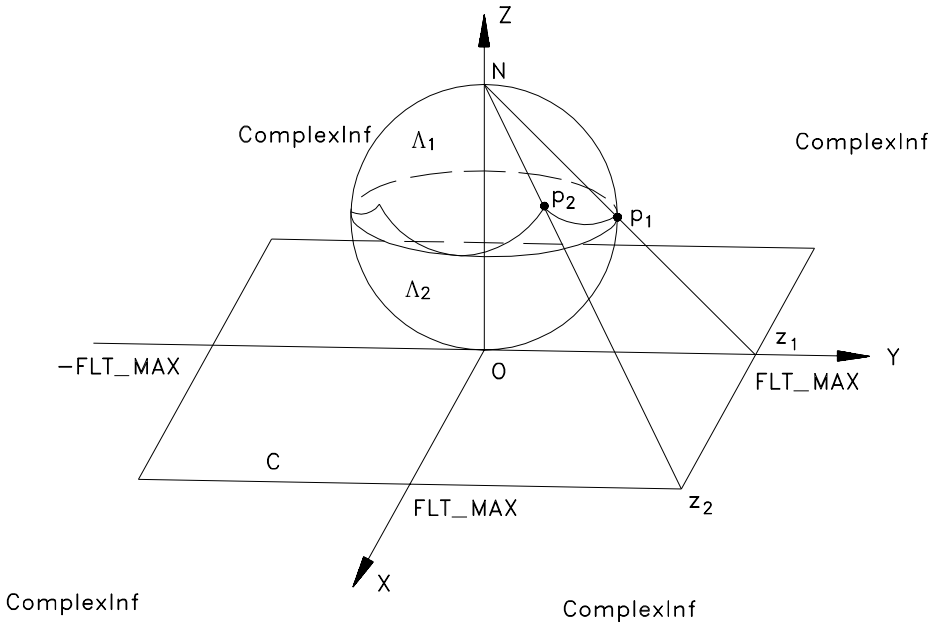


Figure 3: The unit sphere $\Lambda$ and extended finite complex plane.

as ComplexInf or complex(Inf,Inf) in Ch. The origin of the extended finite complex plane is complex(0.0, 0.0). In Ch, an undefined or a mathematically indeterminate complex number is denoted as complex(NaN, NaN) or ComplexNaN, which stands for Complex-Not-a-Number. The special complex numbers of ComplexInf and and ComplexNaN are referred to as *complex metanumbers*. Because of the mathematical infinities of $\pm\infty$, it becomes necessary to distinguish a positive zero 0.0 from a negative zero $-0.0$ for real numbers. Unlike the real line, along which real numbers can approach the origin through the positive or negative numbers, the origin of the complex plane can be reached in any direction in terms of the limit value of $\lim_{r\to 0} re^{i\theta}$ where $r$ is the modulus and $\theta$ is the phase of a complex number. Therefore, complex operations and complex functions in Ch do not distinguish 0.0 from $-0.0$ for real and imaginary parts of complex numbers.

Under this complex model, the conventional numerical results such as ComplexInf+ComplexInf == ComplexNaN, ComplexInf-ComplexInf == ComplexNaN, ComplexInf*ComplexInf == ComplexInf, and exp(ComplexInf) == ComplexNaN can be obtained. The expression $(\infty i) * (\infty i)$ is equivalent to ComplexInf*ComplexInf with the result of ComplexInf.

## 4 Application Examples

In this section, we will describe how metanumbers and complex numbers can be used in applications. A second order polynomial equation

$$ax^2 + bx + c = 0 \tag{3}$$

can be solved by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{4}$$

```
#include <stdio.h>
#include <math.h>

int main() {
    double a = 1, b = -5, c = 6, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

Program 1: C program for solving $x^2 - 5x + 6 = 0$.

```
#include <stdio.h>
#include <math.h>

int main() {
    double a = 1, b = -4, c = 13, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

Program 2: C program for solving $x^2 - 4x + 13 = 0$ in the real domain.

According to the above formula, two solutions of $x_1 = 2$ and $x_2 = 3$ for equation

$$x^2 - 5x + 6 = 0 \tag{5}$$

can be obtained by Program 1. Executing Program 1 gives the following output:

```
x1 = 3.000000
x2 = 2.000000
```

For equation

$$x^2 - 4x + 13 = 0, \tag{6}$$

two complex solutions of $x_1 = 2 + i3$ and $x_2 = 2 - i3$ cannot be solved in the real domain. These complex numbers cannot be represented in double data type. In the domain of real numbers, the values of these solutions are Not-a-Number. Executing Program 2 gives the following output.

```
x1 = NaN
x2 = NaN
```

However, using complex numbers, equation (5) can be solved by a C99 conforming Program 3. Executing Program 3 gives the following output.

```
x1 = complex(2.000000,3.000000)
x2 = complex(2.000000,-3.000000)
```

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main() {
    double complex a = 1, b = -4, c = 13, x1, x2;
    x1 = (-b +csqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -csqrt(b*b-4*a*c))/(2*a);
    printf("x1 = complex(%f,%f)\n", creal(x1), cimag(x1));
    printf("x2 = complex(%f,%f)\n", creal(x2), cimag(x2));
}
```

Program 3: C program for solving $x^2 - 4x + 13 = 0$ in the complex domain.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main() {
    double_complex a = 1, b = -4, c = 13, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = complex(%f,%f)\n", real(x1), imag(x1));
    printf("x2 = complex(%f,%f)\n", real(x2), imag(x2));
}
```

Program 4: C++ program for solving $x^2 - 4x + 13 = 0$ in the complex domain.

```
#include <stdio.h>
#include <math.h>
#ifndef INFINITY
#define INFINITY (1.0/0.0)
#endif

int main() {
    double a = 1, b = INFINITY, c = 13, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

Program 5: C program for solving $a^2 + bx + c = 0$ with b equal to $\infty$ in the real domain.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
#ifndef INFINITY
#define INFINITY (1.0/0.0)
#endif

int main() {
    double complex a = 1, b = INFINITY, c = 13, x1, x2;
    x1 = (-b +csqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -csqrt(b*b-4*a*c))/(2*a);
    printf("x1 = complex(%f,%f)\n", creal(x1), cimag(x1));
    printf("x2 = complex(%f,%f)\n", creal(x2), cimag(x2));
}
```

Program 6: C program for solving $a^2 + bx + c = 0$ with b equal to $\infty$ in the complex domain.

The same problem can be solved by a C++ conforming program shown in Program 4 with the same output as Program 3. Note that complex type is declared with the type specifier `double_complex` in C++ instead of `double complex` in C99. Real and imaginary parts of a complex number are obtained by functions `real()` and `imag()` in C++, instead of `creal()` and `cimag()` in C99.

If the coefficient `b` in equation (3) is $\infty$ which can be represented as `INFINITY` in C99, the solution can be calculated by Program 5 with the output given as follows:

```
x1 = NaN
x2 = -Inf
```

Note that the macro `INFINITY` in C99 for $\infty$ is not supported in gcc, it is obtained conveniently in the program by dividing 1.0 by 0. If complex numbers are used in the calculation by Program 6, which is C99 conforming, the output becomes

```
x1 = complex(NaN,NaN)
x2 = complex(NaN,NaN)
```

## 5 Conclusions

C99 makes the power of the IEEE 754 floating-point arithmetics easily available to the programmer for applications in the real domain. Complex numbers are generalization of real numbers. C99 supports complex numbers as built-in data types whereas C++ uses classes. There are differences for complex numbers in C and C++. But, a typical C or C++ conforming program using complex numbers can run in Ch, a C interpreter, without any modification. It is recommended that a commonly used complex construction function **polar**() be added to resolve the compatibility of C/C++.

IEEE floating-point extension for complex numbers invented in informative Annex G in C99 is in direct conflict with the conventional complex mathematical analysis. The complex model invented in Annex G in C99 is harmful and dangerous for applications in engineering and science. Fortunately, Annex G is not mandated in C99 conforming implementations.

An alternative complex model based on the conventional complex analysis under a Riemann sphere has been presented in this article. In this complex model, there is only one complex infinity (Inf, Inf) == ComplexInf in the extended complex domain, one complex Not-a-Number (NaN,

NaN)==ComplexNaN, and zero is unsigned. More information about this complex model including handling of branch cuts of multiple-valued complex functions can be found in the reference [7].

Ch conforms to the C99 standard related to the IEEE 754 floating-point arithmetic and complex numbers. Following the conventional mathematics, Ch handles numerical numbers consistently in the entire real and complex domains. Ch treats floating-point real numbers with signed zeros, signed infinities (Inf and -Inf), and a Not-a-Number (NaN); and complex numbers with unsigned zeros, a unique complex infinity (ComplexInf), and a unique complex Not-a-Number (ComplexNaN) in an integrated consistent manner. In addition, commonly used high-level functions with complex numbers for numerical analysis have been implemented in Ch. Numerical computing in the entire real and complex domains under the framework of C99 and IEEE 754 standards are very useful for solving problems in engineering and science.

## 6 References


1. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, first edition (K&R C), 1978.
2. ISO/IEC, *Programming Languages - C*, 9899:1990E, ISO, Geneva, Switzerland.
3. ISO/IEC, *Programming Languages - C*, 9899:1999, ISO, Geneva, Switzerland.
4. Cheng, H. H., *The Ch Language Environment, User's Guide*, version 2.0, SoftIntegration, Inc., September, 2001; http://www.softintegration.com.
5. IEEE, *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1985.
6. Cheng, H. H., Scientific Computing in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 49-75.
7. Cheng, H. H., Handling of Complex Numbers in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, 1993, pp. 79-106.