

# Notes on Complex Numbers for Computer Scientists

K.A. Hawick and D.P. Playne  
Institute of Information and Mathematical Sciences  
Massey University – Albany  
North Shore 102-904, Auckland, New Zealand  
Email: k.a.hawick@massey.ac.nz and daniel.playne@gmail.com  
Tel: +64 9 414 0800 Fax: +64 9 441 8181

July 2008

## Abstract

Complex numbers are a useful piece of mathematical apparatus that play a critical role in understanding many applications such as quantum computation. This note summarises some key properties as well as some systems and packages that support programming with complex numbers. We discuss complex number support in Fortran, Python and D as well as library and package support in other languages such as C/C++, Java and Ruby.

**Keywords:** complex numbers; numerical precision; Java; C++; D; FORTRAN; Ruby; Python

## 1 Complex Numbers

Complex numbers consist of a pair of ordinary Real numbers. Sometimes written as  $c = (a, b)$  or  $z = (x, y)$  :  $x, y \in \mathbb{R}$ , where  $a$  or  $x$  is the “real part” - written  $\text{Re } c$  or  $\text{Re } z$  and  $b$  or  $y$  is the “imaginary part”, written  $\text{Im } c$  or  $\text{Im } z$ . The use of the word “imaginary” is historical and perhaps misleading. There is nothing mystical or weird about complex numbers and the so-called imaginary part is just an ordinary real number but with a specific contextual meaning. The two ordinary numbers that make up the ordered pair that is a complex number are often used to denote some entity that has a magnitude and a phase or angle. This is shown in figure 1.

The pair of numbers specifies a vector in two-dimensional space and is shown drawn on the conventional Cartesian x-y axes. We can think of a complex number as having its real part represented horizontally on the conventional real number line taking values  $-\infty, +\infty$  and the imaginary part represented vertically and also taking on  $-\infty, +\infty$  values. We can also represent the complex number vector using a length (or magnitude) drawn as  $r$  and an angle - drawn as  $\theta$ . We often use lower case Greek letters such as theta  $\theta$  or phi  $\phi$  for angles.

Pythagoras tells us what the magnitude or length  $r$  of the vector is and elementary trigonometry tells us the

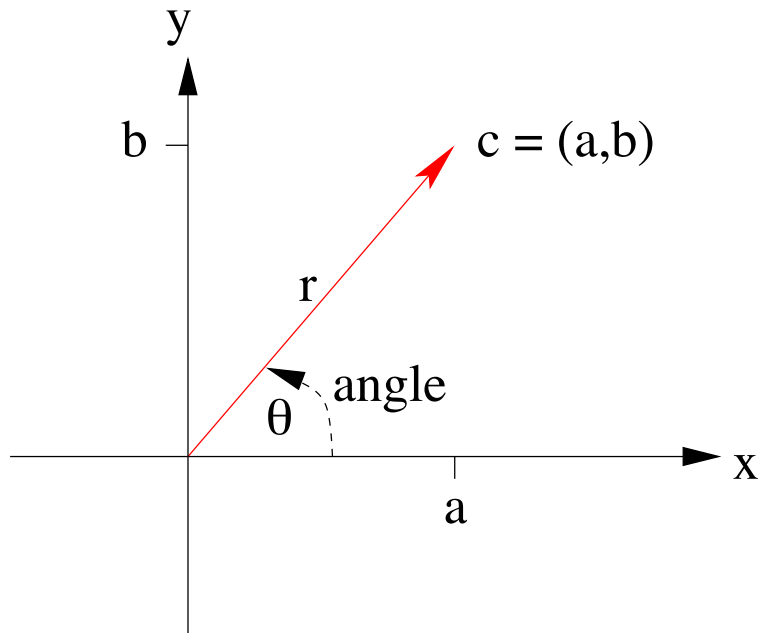


Figure 1: A Complex number  $c = (a, b)$  drawn as a vector on conventional Cartesian x-y axes.

angle  $\theta$ :

$$a = \operatorname{Re} c \quad (1)$$

$$b = \operatorname{Im} c \quad (2)$$

$$r = |c| = \operatorname{abs} c = \sqrt{a^2 + b^2} \quad (3)$$

$$a = r \cos \theta \quad (4)$$

$$b = r \sin \theta \quad (5)$$

$$\theta = \arg c = \arctan \frac{b}{a} \quad (6)$$

We can use the notion of the square root of  $-1$  denoted as  $i : i^2 \equiv -1$  (or sometimes  $j$  in engineering texts) to exploit the algebra of complex numbers, that will let us use them more effectively.

We represent  $c = a + ib$  and use various algebraic properties including:

$$c = a + ib = r e^{i\theta} = r (\cos \theta + i \sin \theta) \quad (7)$$

We need to define  $+, -, \times, \div$  operations for complex numbers. Sensible definitions all arise from the master equation 7:

$$c_3 = c_1 c_2 = (a_1 + ib_1)(a_2 + ib_2) = (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + b_1 a_2) = a_3 + ib_3 \quad (8)$$

$$c_4 = c_1 + c_2 = a_1 + ib_1 + a_2 + ib_2 = (a_1 + a_2) + i(b_1 + b_2) = a_4 + ib_4 \quad (9)$$

It is useful to define the complex conjugation operation that changes the sign of the imaginary part so that  $c = a + ib : c^* \equiv a - ib = r e^{-i\theta}$ . This implies various other useful relations:

$$\operatorname{Re} c = \frac{1}{2}(c + c^*) = a \quad (10)$$

$$\operatorname{Im} c = \frac{1}{2i}(c - c^*) = b \quad (11)$$

$$|c|^2 = c c^* = \operatorname{norm} c \quad (12)$$

$$e^{2i \arg c} = \frac{c}{c^*} \quad (13)$$

$$(c_1 c_2)^* = c_1^* c_2^* \quad (14)$$

$$|c_1 c_2| = |c_1| \cdot |c_2| \quad (15)$$

$$|c_1 + c_2| \leq |c_1| + |c_2| \quad (16)$$

We can define an inverse (to multiplication)  $c^{-1}$  so that:

$$c^{-1} = \frac{c^*}{c c^*} = \frac{c^*}{|c|^2} = \frac{a - ib}{a^2 + b^2} \quad (17)$$

Using the  $r, \theta$  form, note also that:

$$r_1 e^{i\theta_1} \cdot r_2 e^{i\theta_2} = r_1 r_2 e^{i(\theta_1 + \theta_2)} \quad (18)$$

$$(r e^{i\theta})^n = r^n e^{in\theta} \quad (19)$$

$$(r e^{i\theta})^{\frac{1}{n}} = r^{\frac{1}{n}} e^{i\frac{\theta}{n}} \quad (20)$$

The angle  $\theta$  is periodic - a complete rotation of 360 degrees or  $2\pi$  radians leaves the trigonometrical functions unchanged, so that:

$$e^{i\theta} = e^{i(\theta+2\pi)} = e^{i(\theta+4\pi)} = e^{i(\theta+6\pi)} = \dots \quad (21)$$

$$(r e^{i\theta})^{\frac{1}{n}} = r^{\frac{1}{n}} e^{i\frac{\theta}{n}} = r^{\frac{1}{n}} e^{i\frac{\theta+2\pi}{n}} = \dots \quad (22)$$

and so every complex number has  $n$  n-th roots:

$$\sqrt[n]{c} = c^{\frac{1}{n}} = (r e^{i\theta})^{\frac{1}{n}} = r^{\frac{1}{n}} e^{i\frac{\theta+2\pi k}{n}}, k = 0, 1, 2, \dots, n \quad (23)$$

## 2 Programming with Complex Numbers

Various programming languages and systems support complex numbers directly but in some languages a separate package or module must be used to enable their use.

In summary:

Language	Mechanism	Data type
C/C++	include <complex.h>	complex_t
Fortran	built-in	COMPLEX
D	built-in	cdouble
Python	built-in	complex
Ruby	require complex	complex

Table 1: Support for Complex Number type in Different Programming Languages

### 3 Programming with Complex Numbers in C++

In C++ the package `complex.h` provides a mechanism and library functions based on template classes. We have:

- a template class `complex` of float, double and long double
- `real`, `imag` returning a real
- functions: `abs`, `arg` and `norm` returning a real
- functions: `conj` and `polar(r, theta)` returning a complex
- `cos`, `cosh`, `sin`, `sinh`, `tan`, `tanh`, `exp`, `log`, `log10`, `pow`, `sqrt`
- the usual operators : `+`, `-`, `*`, `/` are all defined as well as the C style operators: `+=`, `*=`, ... etc.
- C++ stream insertion and extraction operators.

Use of these functions is illustrated in the C++ code segment below. The package defines a template class for `complex` - it is convenient in this present age of 64 bit processor architectures to decide to use complex numbers that consist of a pair of doubles. A double is likely to be a 64 bit floating point number on many systems, although note that the language standard does not mandate this. It is convenient to use the modern name convention for a type definition and define the data type `complex_t` accordingly.

The code segment in figure 2 will produce output like:

```
(1,0) = 1 + i. 0
(2,-2) = 2 + i. -2
(2,-2) = 2 + i. -2 = 2.82843 e^ i -0.785398
(0.25,0.25) = 0.25 + i. 0.25 = 0.353553 e^ i 0.785398
```

Note that 0.785398 corresponds to 45 degrees, and that 2.82843 is approximately  $2\sqrt{2}$ .

### 4 Complex numbers in Java<sup>TM</sup>

The Java<sup>TM</sup> programming language does not have the in-built capability to work with complex numbers. To solve this problem we must define our own class (`Complex`) which will allow us to to perform complex arithmetic. The class `Complex` stores the real and imaginary values of the complex number and provides a number of methods to allow us to work with these `Complex` objects. The methods (similar to the C++ methods) that should be provided are:

```

#include <iostream>
#include <complex>

using namespace std;

typedef complex<double> complex_t;

int main( int argc, char *argv[] ){

    complex_t  c1, c2, c3, c4;

    double a1, a2, a3, a4, b1, b2, b3, b4;
    double r3, r4, theta3, theta4;

    a1 = 1.0;
    b1 = 0.0;

    a2 = 2.0;
    b2 = -2.0;

    c1 = complex_t( a1, b1 );
    c2 = complex_t( a2, b2 );

    cout << c1 << "=" << a1 << "+i." << b1 << endl;
    cout << c2 << "=" << a2 << "+i." << b2 << endl;

    c3 = c1 * c2;

    a3 = real( c3 );
    b3 = imag( c3 );

    r3 = abs( c3 );
    theta3 = arg( c3 );

    cout << c3 << "=" << a3 << "+i." << b3 << "="
        << r3 << "e^i" << theta3 << endl;

    c4 = c1 / c2;

    a4 = real( c4 );
    b4 = imag( c4 );

    r4 = abs( c4 );
    theta4 = arg( c4 );

    cout << c4 << "=" << a4 << "+i." << b4 << "=" << r4 << "e^i" << theta4 << endl;

    exit(1);
}

```

Figure 2: C++ Code Example showing use of typical `complex_t` data type from `complex.h`.

- **re**, **im** fields, accessing a real (double)
- functions: **abs phase** returning a real
- functions: **conjugate reciprocal** returning a Complex
- functions: **log exp sqrt** returning a Complex
- functions: **cos sin tan acos asin atan** returning a Complex
- functions: **cosh sinh tanh acosh asinh atanh** returning a Complex
- functions: **plus minus times divides** returning a Complex

These functions will allow us to perform arithmetic and use Complex numbers in our calculations. The basic arithmetic  $+*$  for our Complex class must be performed by methods such as *plus(Complex)* and *minus(Complex)* due to the fact that Java<sup>TM</sup> does not provide us with any way to overload operators. The signature for our Complex class is shown in figure 3.

```
public class Complex {
    private double re;
    private double im;

    public Complex(double real, double imag);
    public String toString();

    public double re();
    public double im();

    public double abs();
    public double phase();

    public Complex plus(Complex number);
    public Complex minus(Complex number);
    ...
}
```

Figure 3: Java Interface specification for a complex class.

One interesting dilemma this style causes is the order in which the expressions are evaluated. *c1.plus(c2)* quite clearly represents  $c1 + c2$  and *c1.minus(c2)* should mean  $c1 - c2$ . The order of multiplication is not important however it is vital for division; it is not entirely clear whether *c1.divides(c2)* should represent  $\frac{c1}{c2}$  or  $\frac{c2}{c1}$ . For this reason it was decided that it a better naming convention would be to have the methods *times(Complex)* and *over(Complex)*. These names leave no ambiguity in the meaning of equations. *c1.over(c2)* quite clearly represents  $\frac{c1}{c2}$ .

Since Java<sup>TM</sup> lacks the operator overloading features that are available in C++ we end up with a complex arithmetic capability that is much more complicated to construct and less directly related to the normal mathematics syntax than we would be able to attain with a truly built in type. For example the relatively simple equation  $c1 = c2/c3 + c4/c5$  must be written in the form  $c1 = c2.over(c3).plus(c4.over(c5))$ . This can make writing more complex equation extremely tricky and error prone, however there is a possible alternative. Although Java<sup>TM</sup> does not support operator overloading there is a preprocessor available called JFront [1] which can provide similar functionality. JFront processes the Java<sup>TM</sup> code before it is compiled into Java<sup>TM</sup> byte code and replaces defined operators with standard Java<sup>TM</sup> code. For example, the  $+$  operator can be defined by:

```

public class Complex {
    ...
    Complex operator+(Complex c2) {
        Complex result = new Complex();
        result.re = this.re + c2.re;
        result.im = this.im + c2.im;
        return result;
    }
    ...
}

```

This operator can then be used in the form:

$$c1 = c2 + c3$$

Before the code is sent to Java<sup>TM</sup>, JFront will analyse the code to find the operators defined in each class. After this step JFront will have changed the Complex class code into:

```

public class Complex {
    ...
    Complex operatorplus(Complex c2) {
        Complex result = new Complex();
        result.re = this.re + c2.re;
        result.im = this.im + c2.im;
        return result;
    }
    ...
}

```

After the class operators have been detected and changed, it will look for the usage of these operators within the rest of the program and replace them with the appropriate method calls. The example equation will be replaced by:

$$c1 = c2.operatorplus(c3)$$

This effectively automates the process of converting the equation in mathematical notation into the Java<sup>TM</sup> code. JFront provides a simple way of easily expressing Complex arithmetic in the Java<sup>TM</sup> programming language.

## 5 Complex Numbers in Fortran

Many scientists may have first used complex numbers in programs using Fortran [2]. Fortran has supported a built in COMPLEX data type consisting of a pair of single precision floating point numbers since its early language standards. No special compiler directives or include packages are required to use complex numbers and perform complex arithmetic within Fortran.

Along with support for basic complex number arithmetic, Fortran also supports built in generic functions for ABS, EXP, LOG, SQRT, SIN, COS, TAN and so forth. The precision of the Fortran complex numbers can be improved within most modern Fortran compilers by the use of a mechanism such as KIND to specify different precisions and it is possible to have a double precision (64 bit) complex number type. While there is much to be said for the orthogonality of COMPLEX with the other types in Fortrans, it is likely that newer programming projects will use languages like D and its complex number facilities.

```

PROGRAM MYPROG
COMPLEX A, B, C
A = COMPLEX( 1234.0, 5678.0)
READ(5,*) B
C = A * B

WRITE(6,1000) C
1000 FORMAT(1H0, 'C IS: ', F9.4, ' + i ', F9.4)

C Make C equal to its complex conjugate:
C = CONJG(C)

WRITE(6,*) 'Imaginary part of C is: ', AIMAG(C)
WRITE(6,*) 'Real part of C is: ', REAL(C)

STOP
END

```

Figure 4: Uses of the COMPLEX data type in Fortran

## 6 Complex Numbers in D

The D programming language [3] supports complex numbers as full inbuilt language types. The `cdouble` type implements a complex number as a pair of 64-bit doubles - one each for the real and imaginary parts.

The D language also supports `real` and `creal` that are the native machine floating point types. On a MacPro workstation using Intel Xeon processors the `real` corresponds to an 80 bit floating point representation which can be exploited for extra precision, although the IEEE portable 64 bit `double` and `cdouble` may be the normal programmer choices.

The code in figure 5 produces the following output using `creal`:

```

2+1i
true
2
2
z 1.41421356237309505+1.41421356237309505i
abs 2
conj 1.41421356237309505+-1.41421356237309505i
norm 4
2
1.38629436111989062
54.5981500331442391

```

and the (different) output using `cdouble`:

```

2+1i
true
2
2
z 1.41421356237309515+1.41421356237309515i
abs 2.00000000000000014

```

```

import std.stdio;
import std.math;

real norm( creal z ){
    return z.re * z.re + z.im * z.im;
}

void main( char [][] args ){

    creal z1, z2;
    ireal i1 = 2.0i;

    z1 = 2.0 + 1.0i;

    writefln("%s", z1 );

    z1 = z1.re + 1.5i;
    z1 = z1.re + i1;

    z2 = z1.re + z1.im * li;
    writefln("%s", z1 == z2 );

    writefln("%s", z1.re );
    writefln("%s", z1.im );

    z1 /= 2.0;
    z1 *= 2.0;
    z1 = sqrt(z1.re) + sqrt(z1.im) *li;

    writefln("z    %.18s", z1 );

    writefln("abs  %.18s", abs( z1 ) );
    writefln("conj %.18s", conj( z1 ) );
    writefln("norm %.18s", norm( z1 ) );

    writefln("%.18s", sqrt( norm( z1 ) ) );
    writefln("%.18s", log( norm( z1 ) ) );
    writefln("%.18s", exp( norm( z1 ) ) );
}

```

Figure 5: A complete D program showing use of `creal`. Substituting `cdouble` for `creal` employs a pair of 64 bit IEEE floating point numbers.

```

conj 1.41421356237309515+-1.41421356237309515i
norm 4.000000000000000089
2.000000000000000022
1.38629436111989084
54.5981500331442876

```

Some precision is perceptibly lost during the calculations using 64 bits. Note in particular that to 18 decimal places the `abs` is no longer exact using 64 bits.

D is a relatively recent programming language and current implementations may in fact use macros and other apparatus hidden from the user to implement the D language on top of a C or C++ compiler. Consequently the complex package may in fact be implemented as some overloaded operators that invoke the library routines or other macros from a C/C++ “`complex.h`” package.

## 7 Precision and Numerical Issues

Although the definition of the complex modulus is:

$$|a + ib| = \sqrt{a^2 + b^2} \quad (24)$$

This is not a numerically safe way to compute it as this can result in drastic error if either the real or imaginary parts are comparable with the square root of the largest representable number. A safer way to perform this calculation is:

$$|a + ib| = \begin{cases} |a| \sqrt{1 + (b/a)^2}, & |a| \geq |b| \\ |b| \sqrt{1 + (a/b)^2}, & |a| \leq |b| \end{cases} \quad (25)$$

A similar issue arises with complex division and a numerically safe way to avoid potential overflow is to use (the somewhat cumbersome) formulation:

$$\frac{a + ib}{c + id} = \begin{cases} \frac{(a+b(d/c))+i(b-a(d/c))}{c+d(d/c)}, & |c| \geq |d| \\ \frac{(a+b(d/c))+i(b-a(d/c))}{c+d(d/c)}, & |c| \geq |d| \\ \frac{(a(c/d)+b)+i(b(c/d)-a)}{c(c/d)+d}, & |c| \leq |d| \end{cases} \quad (26)$$

Complex square roots need also be approached carefully with a multi-branch formula:

$$\sqrt{c + id} = \begin{cases} 0, & w = 0 \\ w + i \frac{d}{2w}, & w \neq 0, c \geq 0 \\ \frac{|d|}{2w} + iw, & w \neq 0, c \leq 0, d \geq 0 \\ \frac{|d|}{2w} - iw, & w \neq 0, c \leq 0, d \leq 0 \end{cases} \quad (27)$$

where:

$$w = \begin{cases} 0, & c = d = 0 \\ \sqrt{|c| \sqrt{\frac{1 + \sqrt{1 + (d/c)^2}}{2}}}, & |c| \geq |d| \\ \sqrt{|d| \sqrt{\frac{|c/d| + \sqrt{1 + (c/d)^2}}{2}}}, & |c| \leq |d| \end{cases} \quad (28)$$

Unfortunately these safe but slower algorithms are often not implemented in “complex.h” packages and therefore C++ and potentially D implementations that are built upon them are not necessarily safe against overflows. It is not; however, difficult to adapt the necessary overloaded operators as defined in the system “include files” for open compiler implementations.

More information on the effect of overflow in complex numbers is available in [4, 5].

## 8 Complex Numbers in Python and Ruby

Very modern interpreted environments such as the interpreted language Python [6] now typically come with built in support for complex arithmetic. In Python the numeric literal `j` is used as a suffix to denote an imaginary number. Figure 6 is a fragment of Python code illustrating the use of complex numbers.

```
#!/usr/local/bin/python

print 1 + 2j

print 1.0 * complex( 1.0, -2.0 )

a = 1234.0 + 5678.0j

print a.real

print a.imag
```

Figure 6: A short Python program showing the built in support for complex numbers using the numeric literal `j`.

The Python code in figure 6 produces the following output:

```
(1+2j)
(1-2j)
1234.0
5678.0
```

Other scripting languages such as Ruby [7] require a library package to support complex arithmetic, but do in fact have a richer set of associated functions and operators.

## 9 Summary

In summary, complex numbers are readily programmed in several modern programming languages although users must sometimes still give care and attention to precision and overflow issues, depending upon the needs of applications. Many systems will either support complex arithmetic using built in types or will simply require inclusion of a standard supplied library or package to enable them.

## References

- [1] Garnatz and Grovender Inc: Jfront (2008)
- [2] Metcalf, M., Reid, J., Cohen, M.: Fortran 95/2003 Explained. Oxford University Press (2004) ISBN 0-19-852693-8.
- [3] Bright, W.: D Programming Language. Digital Mars. (2008)
- [4] Knuth, D.: The Art of Computer Programming: Seminumerical Algorithms. 3rd edn. Volume 2. Addison-Wesley (1997)
- [5] Midy, P., Yakovlev, Y.: Computing some elementary functions of a complex variable. *Mathematics and Computers in Simulation* **33** (1991) 33–49
- [6] van Rossum, G.: Python essays. <http://www.python.org/doc/essays/> (2008)
- [7] Thomas, D., Fowler, C., Hunt, A.: Programming Ruby: The Pragmatic Programmer’s Guide. 3rd edn. Pragmatic Programmers (2004) ISBN 978-0-9745140-5-5.