Complex arithmetic in FORTH

Contents

One of the most crucial features of FORTRAN for scientific computation is the ease with which it embeds complex arithmetic into formulae. No other compiled language has this feature[1]. From time to time someone gets a bright idea, and adds complex arithmetic to Pascal *via* subroutines (since Pascal functions can return only a single number, and complex functions must return two numbers)[2]. The same problem would afflict C and the new structured BASICs.

Recently, a mechanism for adding complex arithmetic to Pascal by defining a stack and using postfix notation has been proposed[3]. Does this sound at all familiar?

This chapter deals with complex arithmetic and its implementation in FORTH. We do not consider complex numbers whose real

1. APL gracefully admits complex numbers and functions, but is an *interpreted* language.
2. D.D. Clark, "Simple Calculations with Complex Numbers", *Dr. Dobb's Journal*, October 1984, p. 30.
3. D. Gedeon, "Complex Math in Pascal", *Byte Magazine*, July 1987, p.121.

and imaginary parts are integers, since these are virtually useless in scientific computing. Therefore we perform complex arithmetic solely in single– or double-precision floating point format. We begin with a brief review of the complex number system.

The complex number system

The algebraic equation

$$x^2 - 1 = 0 \tag{1}$$

has 2 roots, ± 1, in the standard number system (real numbers). But the (otherwise quite similar) equation

$$x^2 + 1 = 0 \tag{2}$$

has **no** roots. That is, there is no ordinary number which, when squared, gives a negative result. Eighteenth century mathematicians misliked a state of affairs wherein some polynomial equations of n'th degree had n roots, whereas others had n-2, n-4, *etc*. This seemed disorderly and unpredictable. How much simpler life would be if every polynomial of n'th order could **always** be factored into n primitives of the form (x_n are **roots**)

$$\begin{aligned} &a_0 + a_1x + a_2x^2 + \ldots + a_nx^n \\ &\quad \equiv a_n(x - x_1)(x - x_2) \ldots (x - x_n) \end{aligned} \tag{3}$$

§§1 Cartesian representation of complex numbers

In order that every polynomial have n roots it was necessary to extend the idea of **number** to include objects of the form

$$z = x\,\mathbf{1} + y\,\mathbf{i} \tag{4}$$

where **i** is –by definition– a "number" whose square is –1; **1** is a "number" whose square is +1, and x and y are ordinary numbers. Conventionally, x is called the **real part** of z, and y the **imaginary part**:

$$x = \mathrm{Re}(z)\ ,\ y = \mathrm{Im}(z). \tag{5}$$

Thus the solutions of the polynomial equation

$$z^2 + 1 = 0 \tag{6}$$

are

$$z = 0\mathbf{1} \pm 1\mathbf{i} \tag{7}$$

(that is, $x = 0, y = \pm 1$).

To reassure readers who find uncomfortable the notion of a "number" $\mathbf{i} = \sqrt{-1}$ whose square is negative, it is possible to find a 2×2 **matrix representation** for 1 (unit matrix) and i:

$$\mathbf{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathbf{i} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \tag{8}$$

It is easy to verify that —in the sense of matrix multiplication—

$$\mathbf{i} \times \mathbf{i} = -1, \quad \mathbf{1} \times \mathbf{i} = \mathbf{i} \times \mathbf{1} = \mathbf{i}, \text{ and } \mathbf{1} \times \mathbf{1} = \mathbf{1}. \tag{9}$$

These are just the usual multiplication rules for complex numbers. Note that $x\,\mathbf{1}$ and $y\,\mathbf{i}$ then mean multiplication of a matrix by a scalar:

$$x\mathbf{1} \equiv \begin{pmatrix} x & 0 \\ 0 & x \end{pmatrix}, \quad y\mathbf{i} \equiv \begin{pmatrix} 0 & y \\ -y & 0 \end{pmatrix}$$

$$z = x + iy \equiv \begin{pmatrix} x & y \\ -y & x \end{pmatrix}$$

The complex numbers obey all the algebraic rules of ordinary arithmetic — commutative and associative laws of multiplication and addition. They are **complete** in the sense that when we multiply or add two complex numbers we get a **complex** number, not some other kind of number:

$$(a + ib)(x + iy) = (ax - by) + i(ay + bx) \tag{10}$$

$$(a + ib)(x + iy) = (ax - by) + i(ay + bx) \qquad (10)$$

For every complex number, $z = x + iy$, there is a corresponding **complex conjugate** complex number,

$$z^* = x - iy. \qquad (11)$$

Each non-zero complex number z has a **multiplicative inverse** $1/z$. To see this, multiply numerator and denominator of $1/z$ by the complex conjugate Eq. 11, and note that

$$z z^* \equiv z^* z = x^2 + y^2$$

is a non-zero **real** number. We can therefore calculate its inverse by ordinary division, and (scalar-) multiply z^* by the result:

$$\frac{1}{z_2} \equiv \frac{z_2^*}{z_2 z_2^*} \equiv \left(\frac{x_2}{x_2^2 + y_2^2} - i \frac{y_2}{x_2^2 + y_2^2} \right). \qquad (12)$$

To **divide** one complex number z_1 by another, z_2, invert z_2 and multiply: $z_1/z_2 \equiv z_1 \times (1/z_2)$, *i.e.*

$$\frac{z_1}{z_2} \equiv (x_1 + iy_1) \times \left(\frac{x_2}{x_2^2 + y_2^2} - i \frac{y_2}{x_2^2 + y_2^2} \right) \qquad (13)$$

(Numbers that satisfy the addition, multiplication and closure properties of real or complex numbers are said to form a **field**.)

It is easy to prove that even when the coefficients of an n'th degree polynomial are **complex**, it has n roots that can be represented as complex numbers.

Complex numbers can be used to represent points in the x-y plane with y plotted vertically and x horizontally[4]. (Sometimes this graphical representation is called an **Argand plot** or **Argand diagram**. The x-y plane is then called the **Argand plane**.)

4. The 2-dimensional graphical representation of complex numbers makes complex arithmetic attractive for computer graphics.

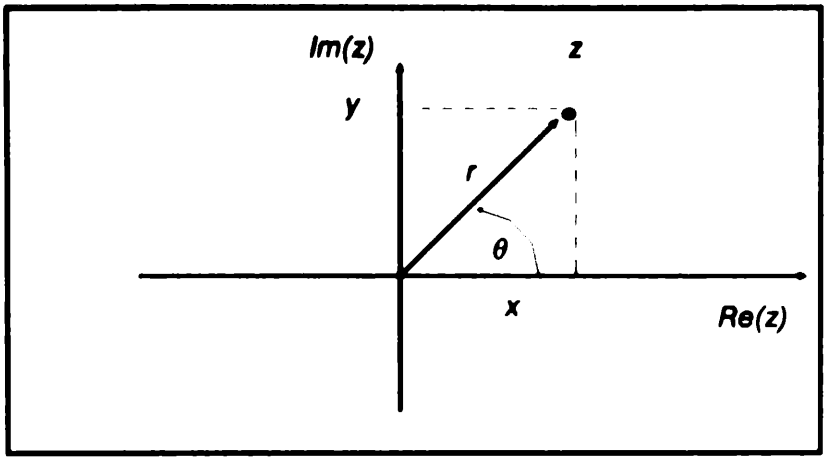

Fig. 7-1 *Representing a complex number as a point in the Argand plane, in Cartesian and polar coordinates*

§§2 Polar representation of complex numbers

The complex number $z = x + iy$ is said (remember the Argand plane) to be represented in **Cartesian** form (after René Descartes, the inventor of analytic geometry, *i.e.* the idea of graphing equations). However, it is equally valid to use the relationship (**Euler's theorem**)

$$e^{i\theta} \equiv \cos\theta + i\sin\theta \tag{14}$$

to write the **polar representation** of z,

$$z = re^{i\theta}\ ,\ r \geq 0\,. \tag{15}$$

Clearly, since $z^* \equiv re^{-i\theta}$, we have

$$r = \sqrt{zz^*} = \left(x^2 + y^2\right)^{\frac{1}{2}} \tag{16}$$

and[5]

$$\theta = \tan^{-1}(y/x). \tag{17}$$

5. See Ch. 3 §4§§1 for definitions of the inverse trigonometric functions.

Note that θ, sometimes called Arg(z), is defined only up to multiples of 2π – that is, adding 2π to an angle changes neither *sine* nor *cosine* because of their periodicity. The relations between (x,y) and (r, θ) are illustrated in Fig. 7- 1 above:

§2 Load, store, manipulate fstack

We now define FORTH words to perform complex arithmetic. All complex operations will be prefixed by the letter X[6]. We assume all complex operations are performed on the FPU for speed, hence we shall give 87stack diagrams.

In 87stack (or fstack) diagrams z stands for complex number, x for real part, and y for imaginary part. Where useful, the 87stack diagrams show the operations decomposed into real and imaginary parts. By convention the imaginary part is higher on the fstack than the real part.

By now most FORTH code should be self-explanatory. Many words have been coded in high-level FORTH because the overhead of threading is negligible compared with the time spent executing.

With by-now obvious meanings we have

```
\ - - single & double-precision complex fetch and store.
: X@     DUP R32@  4+ R32@ ;  ( adr - - 87: - - z)
: X!     DUP  4+  R32! R32! ;  ( adr - - 87: z - -)
CODE  8+   BX  08  IW ADD.  END-CODE
: DX@    DUP  R64@ 8+ R64@ ;  ( adr - - 87: - - z)
: DX!    DUP  8+  R64! R64! ;  ( adr - - 87: z - -)
\ - - end complex fetch and store.
```

---

6. The letter **C** is more mnemonic, but FORTH conventionally reserves **C** for prefixing byte ("character") operations – as in **C@**, **C!**, *etc.* The complex extension supplied with HS/FORTH uses the prefix **C** when it is unambiguous (as in **C+**, **C–**, **C***, **C/**, *etc.*), and **CP** when **C** alone won't do, as in **CP@**, *etc.*

```
\ fstack manipulation
: REAL   FDROP ;                    ( 87: z - - x)
: IMAG   FPLUCK ;                   ( 87: z - - y)
: CONJG  FNEGATE ;                  ( 87: x y - - x , -y )

: XDROP  FDROP  FDROP  ;            ( 87: z - -)
: XSWAP  F4R   F4R  ;               ( 87: z1 z2 - - z2 z1)
: XDUP   FOVER FOVER  ;             ( 87: z - - z z)
```

§3 Arithmetic operations

The standard complex operations should be virtually self-explanatory. For efficiency we define multiplication and division of a complex number by a scalar, as well as complex×complex and complex/complex.

```
: CMPLX   F=0 ;                     ( 87: x - - x 0 )
: X+     FROT  F+  F-ROT   F+   FSWAP ;
: X-     FROT  FR- F-ROT   F-   FSWAP ;
: XOVER   F3P  F3P ;                ( 87: z1 z2 - - z1 z2 z1)
: X*F                               ( 87: x y a - - ax ay )
   FUNDER  F*                       ( 87: - - a x bx )
   F-ROT   F*   FSWAP  ;
: F*X     FROT   X*F  ;             ( 87: x a b - - ax bx )
\ CODE   X*F    2 FMUL".   1 FMULP.   END-CODE
: X*I    FNEGATE  FSWAP  ;
: X/F    1/F   X*F  ;
```

The critical operation of complex×complex can be defined in high level FORTH for portability:

```
: X*                                ( 87: z1 z2 - - z1*z2 )
   XOVER  X*I   FROT                ( 87: - - a b x -b a y)
   X*F                              ( 87: - - a b x -by  ay)
   F3X  FSWAP                       ( 87: - - a  ay x  b  -by)
   F4X  FSWAP                       ( 87: - -  -by  ay x  a  b)
   F*X                              ( 87: - -  -by  ay  xa xb)
   X+  ;
```

Actually, complex multiplication is sufficiently involved to be worth defining in code. Here is a code definition for the 80x87 family of FPUs. The equivalent for the Motorola 68881/2 family is virtually identical, allowing for minor differences in Intel and

Motorola assembler mnemonics, as well as for the fact that the 68881/2 has registers but no stack.

```
\ operation                    87stack contents

CODE  X*                          \ x y a b
   3  FLD.                        \ x y a b x
   2  FMUL.                       \ x y a b xa
   4  FXCH.                       \ xa y a b x
   1 FMUL.                        \ xa y a b xb
   1 FXCH.                        \ xa y a xb b
   3 FMUL.                        \ xa y a bx yb
   4 FSUBRP.                      \ xa-yb y a bx
   2 FXCH.                        \ xa-yb bx a y
   1 FMULP.                       \ xa-yb bx ay
   1 FADDP.                       \ xa-yb bx+ay
END-CODE                       ( x y a b - - xa–yb xb+ya)
```

Once we have multiplication, division is easy:

```
: XMODSQ                       ( 87: x y - - x**2+y**2)
   F**2  FSWAP  F**2  F+  ;
: 1/X    CONJG    XDUP  XMODSQ
   FDUP   F0=  ABORT" Can't divide by 0"   X/F  ;
: X/    1/X  X*  ;             ( 87: z1 z2 - - z1/z2 )
```

With the preceding discussion and referring to Fig. 7-1 on page 147 the FORTH words to accomplish Cartesian-polar transformation and *vice versa* should also be fairly transparent[7]:

```
: ARG   FSWAP  FPATAN  ; ( 87: x y - - atan[y/x])
SYNONYM  FATAN2   ARG
\ FORTRAN defines FATAN2 so we do also.

: XABS                         ( 87: z - - |z|)
   XMODSQ   FSQRT  ;
: >POLAR                       ( 87: x y - - r θ[radians])
   XDUP  XABS  F-ROT  ARG  ;
: POLAR>                       ( 87: r θ [radians] - - x y)
   FSINCOS  FROT  X*F  ;
```

7. **SYNONYM** is an HS/FORTH innovation to save some code by avoiding : **FATAN2 ARG ;**

§4 Roots of complex numbers

The n'th root of a complex number z is that complex number, $z^{1/n}$, whose n'th power is the original number. That is, as for real numbers,

$$\left(z^{1/n}\right)^n \equiv z \tag{18}$$

It might seem almost trivial to define the n'th root of a complex number in polar representation:

$$z = re^{i\theta} \tag{19}$$

hence

$$z^{1/n} = r^{1/n} e^{i\theta/n} \tag{20}$$

Certainly we know what we mean by the n'th root of a positive real number, and from Euler's theorem we know how to evaluate

$$e^{i\theta/n} \equiv \cos\frac{\theta}{n} + i\sin\frac{\theta}{n} \tag{21}$$

However, we can also think of the n'th root as a solution of the polynomial equation in the variable w

$$w^n - z = 0. \tag{22}$$

The **fundamental theorem of algebra** proves that a polynomial of n'th degree has exactly n roots; this implies there are n distinct values of w that satisfy Eq. 22; *i.e.*, there are n distinct roots of z. How can we generate them? Let us call the root shown above (in Eq. 20)

$$w_0 = r^{1/n} e^{i\theta/n}$$

Then we can multiply w_0 by a factor

$$\chi_k = e^{2\pi ik/n}, \quad k = 0, 1, \ldots, n-1 \tag{23}$$

to obtain n different numbers,

$$w_k = w_0 \chi_k\,,\ k=0,\ldots,n-1$$

the n'th power of each of which is z. Clearly, if k increases past n-1, the numbers w_k simply repeat[8]. Since it is neither possible nor desirable for a complex function to return all n roots of z, we choose the **principal** one, w_0. All complex root- finding functions should obey this convention. In general the simplest algorithm to calculate the n'th root of a positive real number is

$$r^{1/n} = e^{ln(r)/n}\,. \tag{24}$$

Then to complete the job of evaluating w_0 we would need to calculate a sine and a cosine. Thus, 3 divisions, 2 multiplications and 4 transcendental function calls are generally needed to evaluate the n'th root of a complex number.

However, for square roots (n = 2) there is a much more efficient method based on ordinary square roots, which we shall now describe.

§§1 Complex square roots

Our phase convention means the phase θ of the square root of z must lie between 0 and π, since that of z lies between 0 and 2π.

The algorithm can be understood using the half-angle formulae for sines and cosines (we used these to develop the trigonometric functions in Ch. 4 §6):

$$\cos\left(\frac{\theta}{2}\right) = \left(\frac{1+\cos\theta}{2}\right)^{\frac{1}{2}} \tag{25a}$$

$$\sin\left(\frac{\theta}{2}\right) = \left(\frac{1-\cos\theta}{2}\right)^{\frac{1}{2}} \tag{25b}$$

8. Recall $e^0 \equiv e^{2\pi ni} = 1$.

Now we use the fact that if $z = x + iy$, and if $w = a + ib$ is its square root, then their polar representations are

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r\cos\theta \\ r\sin\theta \end{pmatrix} \tag{26a}$$

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sqrt{r}\cos(\theta/2) \\ \sqrt{r}\sin(\theta/2) \end{pmatrix} \tag{26b}$$

Therefore the principal root is

$$a = sgn(y)\left[\frac{1}{2}(r+x)\right]^{\frac{1}{2}}, \tag{27a}$$

$$b = \left[\frac{1}{2}(r-x)\right]^{\frac{1}{2}}. \tag{27b}$$

That is, as we can easily see from the Argand diagram, Fig. 7-2 below, if Im(z) is negative, then Re(w) will be negative, and *vice versa*.

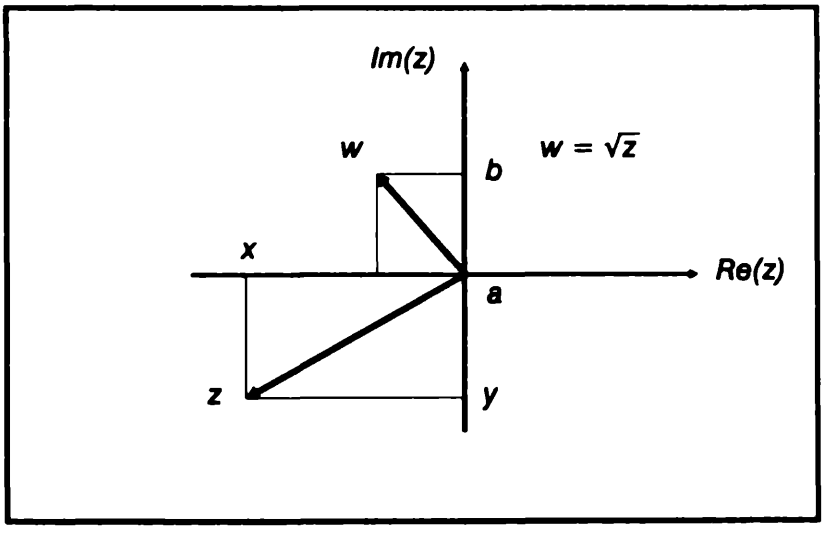

Fig. 7-2 *Complex number with negative imaginary part, and its principal square root*

§§2 The complex square root program

We now translate these equations into high-level FORTH with comments:

```
: XSQRT                                  ( 87: z - - z**1/2)
   FSWAP  XDUP  XABS                     ( 87: - - y x r)
   FDUP  F0=                             \ return 0 if |z| = 0
   IF     FDROP XDROP  X=0  EXIT    THEN
   XDUP  F+                              ( 87: - - y x r r+x )
   F2X  F-                               ( 87: - - y r+x r-x)
   F2/  FSQRT                            ( 87: - - y r+x b )
   F2X                                   ( 87: - - b r+x )
   F0<                                   \ get sign of Im(z)
   F2/  FSQRT                            ( - - f 87: - - b |a| )
   IF FNEGATE  THEN        \ fix sign of Re(z**1/2)
   FSWAP ;
```

We test to make sure $|z| > 0$, since we do not want the possibility that $|z| - x$ or $|z| + x$ works out to be negative (albeit small) through roundoff, thereby generating an error in the square root routine.

§5 Complex exponentials and trigonometric functions

Complex exponentials and trigonometric functions are nearly self-explanatory. They are based on **Euler's theorem**,

$$e^{i\theta} = \cos\theta + i\sin\theta\,.$$

Thus,

$$e^{z} \equiv e^{x+iy} \equiv e^{x}e^{iy} = e^{x}\left(\cos y + i\sin y\right) \tag{28}$$

Similarly,

$$\cos z = \frac{1}{2}\left(e^{ix-y} + e^{y-ix}\right) \tag{29a}$$

$$\sin z = \frac{1}{2i}\left(e^{ix-y} - e^{y-ix}\right). \tag{29b}$$

Hence the code for the complex trigonometric functions is

```
: FSINCOS              ( 87: x - - cos[x] sin[x] )
   F2/  FTAN           ( 87: - - a = tan[x/2] )
   FDUP  F**2          ( 87: - - a a**2 )
   F=1  XDUP  F+       ( 87: - - a a**2  1  1+a**2)
   F2X  F-             ( 87: - - a  1+a**2  1-a**2)
   FOVER  F/           ( 87: - - a  1+a**2  cos[x] )
   F-ROT  F/  F2*  ;  ( 87: x - - cos[x] sin[x] )
\ note: FSINCOS is microcoded on the 80387

: XEXP          ( 87: x y - - e**x*cos[y]  e**x*sin[y] )
   FSINCOS  FROT  FEXP  X*F  ;

: X2/    F2/  FSWAP  F2/  FSWAP  ;

: XSIN          ( 87: x y - - sin[x]cosh[y] cos[x]sinh[y])
   FNEGATE  FEXP  FSWAP  FSINCOS   F*X
   XDUP   1/X    X-  X2/  FSWAP  FNEGATE  ;

: XCOS          ( 87: x y - - cos[x]cosh[y]  -sin[x]sinh[y] )
   FNEGATE  FEXP  FSWAP  FSINCOS   F*X
   XDUP   1/X    X+   X2/  ;
```

§6 Logarithms

The logarithm of a complex number must be defined by the polar representation of the number. Thus, using the fact that

$$\log_e(ab) = \log_e(a) + \log_e(b)\,, \tag{30}$$

and that

$$\log_e(e^z) = z\,, \tag{31}$$

we find it consistent to **define** the complex logarithm as

$$\log_e(z) = \log_e\left(r\,e^{i\theta}\right) \equiv \log_e(r) + i\,\theta\,. \tag{32}$$

Thus,

```
: XLOG                  ( 87: x y - - ln[r] atan[y/x])
   >POLAR  FSWAP  FLN  FSWAP  ;
```

This completes our dissertation on complex arithmetic.