TRUE BASIC

The Original BASIC

Complex Arithmetic Toolkit

Reference Guide

Complex Arithmetic Toolkit

by Thomas E. Kurtz
Co-inventor of BASIC

Introduction

This toolkit allows you to write True BASIC programs in the usual way except that certain numeric variables and arrays may be declared to be **complex**.

A **DO** program then revises your True BASIC program into one that can be run directly and that will perform the calculations using complex arithmetic, if necessary.

You may use most of the True BASIC statements and structures including modules, internal and external subroutines, public and shared variables, etc. There are some restrictions.

1. The numeric variable 'i' always stands for the square root of -1. Therefore, 'i' may not be used as a looping variable in a **FOR NEXT** loop.
2. All variables and arrays that are intended to be complex must be so declared early on in the main program, subroutine, or module, and before any other statements, such as **LOCAL** or **PUBLIC**, that contain these variables. There is a special requirement for subroutines. A complex declaration must be included just after the **SUB** statement in the definition if some or all of the parameters in the **SUB** statement are to be complex. The same is true for multiple-line **DEF** structures. (Single-line **DEF**s cannot be of type complex nor can their arguments be complex. The solution is to make them into multiple-line **DEF**s.)
3. Only simple **MAT** statements may be used.

The toolkit operates as a **DO** program, revising the contents of the source program in the current editing window. The subroutines of the toolkit may be pre-loaded, using

```
SCRIPT loadcomplex
```

but this is not necessary. After a successful revision, the resulting modified source program can be run directly, or saved.

This toolkit combines that portion of the former **Mathematician's Toolkit** that dealt with complex arithmetic for simple and matrix entities.

Example:

```
REM  Quadratic equation solver

DECLARE COMPLEX r1, r2

FOR example = 1 to 3
    READ a, b, c          ! Coefficients of equations
    CALL quad(a,b,c,r1,r2)    ! Solve
    PRINT r1, r2
NEXT example

DATA 1, 4, 3
DATA 1, 4, 4
DATA 1, 4, 5

END

SUB quad(a,b,c,r1,r2)     !  Equation solver, assumes a<>0

    DECLARE COMPLEX r1, r2, s

    LET discr = b^2 - 4*a*c   ! Discriminant
    LET s = sqr(discr)        ! Complex square root

    LET r1 = (-b+s)/(2*a)
    LET r2 = (-b-s)/(2*a)

END SUB
```

Notice that all you really have to do is to insert a

```
DECLARE COMPLEX r1, r2
```

into the main program, early on, to notify the toolkit that the variables 'r1' and 'r2' are going to be complex-valued.

A similar complex declaration must appear in the external subroutine. It must appear right after the **SUB** statement as it applies to two of the parameters in that statement.

The next step is to revise the program using

```
do complex
```

After the revision, this will be your program in the current editing window:

```
DECLARE DEF c_out$
LIBRARY "CompLibs.trc"
REM  Quadratic equation solver

! declare complex r1, r2
```

```
FOR example = 1 to 3
    READ a, b, c          ! Coefficients of equations
    CALL QUAD (A, B, C, c_R1$, c_R2$)  ! Solve
    PRINT c_out$(c_R1$), c_out$(c_R2$)
NEXT example

DATA 1, 4, 3
DATA 1, 4, 4
DATA 1, 4, 5

END

SUB QUAD (A, B, C, c_R1$, c_R2$)
DECLARE DEF c_sqr1$,c_sum1$,c_quot2$,c_diff1$
    ! declare complex r1, r2, s

    LET discr = b^2 - 4*a*c     ! Discriminant
    LET c_S$ = c_sqr1$(DISCR)   ! Complex square root

    LET c_R1$ = c_quot2$(c_sum1$(-B,c_S$),2 * A)
    LET c_R2$ = c_quot2$(c_diff1$(-B,c_S$),2 * A)


END SUB
```

Notice these changes:

1. Two statements (**DECLARE DEF** and **LIBRARY**) have been added near the top of the program. If the complex toolkit has been "loaded," these statements are not needed. The **DECLARE DEF** statement names all the defined functions used by the program. (A defined function is defined by a **DEF** statement, in contrast with, for example, the **SIN** function, which is builtin.) The only special function used in the main program is the output formatting function "**c_out$**"

2. The variables r1 and r2 have been declared to be of type **COMPLEX**. The **DO** program changes them to c_R1$ and c_R2$, respectively. All numeric variable names declared to be complex are preceded with "c_" and followed by "$".

3. In the subroutine the **DECLARE COMPLEX** statement converts several of the parameters in the immediately preceding SUB statement.

4. The **DECLARE DEF** statement in the subroutine names a number of functions. For example, **c_sum1$** is used to add a real number and a complex number, in that order. There are two other adding functions: **c_sum2$** is used to add a complex number and a real number, while **c_sum$** is used to add two complex numbers.

5. Since the square root of a real number could be complex, all occurrences of **SQR** are replaced by **c_SQR1$**. There is also **c_SQR$** which is used to take the (principal) square root of a complex number.

How it is Done

Complex numbers have a real and an imaginary part. Thus, each complex number really consists of two numbers. A complex number is stored in a 16-byte string. The first 8 bytes contain the real part and the second 8 bytes contain the imaginary part. The advantage is that a single complex number can be stored and manipulated as a single string rather than as a vector having two elements.

The True BASIC functions **num** and **num$** can be used to go back and forth. Suppose **real** and **imag** are the real and imaginary parts of a complex number **com$**. Creating the complex number **com$** can be done by:

```
LET com$ = num$(real) & num$(imag)
```

The reverse operation can be done with two statements:

```
LET real = num(com$[1:8])
LET imag = num(com$[9:16])
```

You write your program in the usual way. To convert it to complex, type

```
do complex
```

This 'do program' converts the current program in the editing window into a complex one.

If you are not using the **Gold Edition**, you should now type

```
rename _complx_
```

or something similar, to prevent accidently saving the revised program over your original saved version.

Now type the command

```
run
```

or simply select **Run** from the Run menu.

The complex tool kit runs equally well whether set up as a loaded workspace or not. If a loaded workspace, the startup time for the do program is much shorter. To load the complex toolkit workspace, enter the complex toolkit directory and type

```
script loadcomplex
```

If your complex toolkit is not loaded as a workspace, just make sure that the compiled toolkit file **CompWork.trc** is in your current directory, or that there is an 'alias' to allow True BASIC to find it.

If you load the file **CompWork.trc**, do not also load the library file **CompLibs.trc**, as the latter file is included. Also, you cannot pre-load the conformal mapping library **ConfLib.trc**.

```
MAT Operations
```

The following **MAT** operations are permitted. (These form a subset of the **MAT** operations allowed in True BASIC.)

```
MAT C = A + B
MAT C = A - B
MAT C = A * B
MAT C = K * A          ! k a scalar variable or constant
MAT C = INV(A)
MAT C = TRN(A)
MAT C = CONJ(A)
MAT C = IDN
MAT C = CON
MAT C = ZER
```

It is a strict requirement that the dimensions and subscript ranges must match.

Adding and subtracting are allowed for vectors and matrices. Products are allowed for two matrices, a vector and a matrix, a matrix and a vector, and for two vectors. Again, the subscript ranges must conform! If you multiply a vector times a matrix, the vector will be interpreted as a row vector. If you multiply a matrix times a vector, the vector will be interpreted as a column vector. If you multiply two vectors, the dot product will be assumed.

Scalar multiplication is allowed for both vectors and matrices. The complex conjugate (**CONJ**) is also allowed for both vectors and matrices.

INV, **TRN**, and **IDN** are allowed only for matrices.

The **PUBLIC** variable **c_det$** will be the complex determinant of the most recently successfully inverted matrix.

Input and Printed Output

Input is always in terms of real numbers. Ordinary INPUT and READ statements can be used. To construct a complex number, such as “1 + i”, simply do so in an ordinary **LET** statement, as in

```
DECLARE COMPLEX c
...
LET c = 1 + i
```

To input a complex number, use something like

```
DECLARE COMPLEX c
INPUT prompt "Enter constant: ": re, im
LET c = re + i*im
...
```

The same approach can be used to **READ** a complex constant

```
DECLARE COMPLEX c
READ re, im
DATA 1, 2
LET c = re + i*im
```

To build a complex matrix, you can use the **READ** and **DATA** statements to create the real and imaginary parts, and then use the subroutine **c_MatComp** (or **c_VecComp**.) For example,

```
DECLARE COMPLEX C
DIM A(3,3), B(3,3), C(3,3)
MAT READ A, B
DATA 1, 2, 3    ! A is the real part
DATA 2, 3, 4
DATA 3, 4, 5
DATA 2, -1, 4   ! B is the imaginary part
DATA 0, 5, 1
DATA 3, 0, -4
CALL c_MatComp (A, B, C) ! C is the complex composition
```

You can output real numbers in the usual way.

```
PRINT x, y
```

If the value is complex, then

```
DECLARE COMPLEX z
...
PRINT z
```

will work. This is converted into **PRINT c_out$(z)**, where **c_out$** is a function that converts a complex number into a string, suitable for printing, of the form **a + i*b**.

You can control the number of decimals places of accuracy with the **DECIMALS** statement.

```
DECIMALS 4
```

will cause all subsequent uses of **c_out$** to round the real and imaginary parts to four decimals places.

You can have any number of **DECIMALS** statements in your program.

If you want to revert to the default (eight significant figures, more or less,) use

```
DECIMALS 999
```

In addition, there is another function **c_outd$(z,d)** that allows you to convert to a string a complex value with both real and imaginary parts rounded to ‘d’ decimal places.

Plotted Output

If the expressions in a **PLOT** statement are complex, they will be plotted on the complex plane. That is,

```
PLOT z
```

will plot a point **(real(z),imag(z))** in the x-y plane. You must, of course, declare z to be of type complex.

Conformal Mapping

The subroutine **ConformalMap** is found in the file **ConfLib.trc** (source in **ConfLib.tru**). The function to be mapped must be named **f** (as in f(x)), and must be included as an external, multiple-line defined function after the **END** statement. The calling sequence is

```
CALL ConformalMap (ll, ur, dx)
```

where **ll** is the lower-left corner of the domain rectangle in the complex plane, **ur** is the upper-right corner, and **dx** is the incremental spacing for both the real and imaginary parts of the argument.

The file **ConfLib.trc** cannot be loaded ahead of time. Thus, the program must contain a library statement

```
LIBRARY "ConfLib.trc"
```

The defined function **f** must include, in its definition, a **DECLARE COMPLEX** statement that names both the function name f and all its complex arguments.

See the demontration program **DemConf.tru**.

Demonstration Programs

DemSines.tru

Verifies that sin^2 + cos^2 = 1 for random complex arguments.

DemQuad.tru

Solves the general quadratic equation with real coefficients, yielding possibly complex roots.

DemRoot.tru

Finds a root of a polynomial equation with possibly complex coefficients using the Newton Raphson method. The user supplies the initial point.

DemConf.tru

Demonstrates conformal mapping for an arbitrary complex function of a complex variable.

DemInv.tru
Finds the inverse and determinant of a square matrix with complex coefficients.

DemMat1.tru
Illustrates certain **MAT** operations (constructing a matrix of complex coefficients, multiplication, addition, subtractions.)

DemMat2.tru
Illustrates other **MAT** operations (vectors with complex coefficients, matrix transpose, vector-matrix and matrix-vector multiplication, complex conjugate, dot product.)

Reference List

Here is a list of all available complex functions.

c_abs(z$)	Absolute value of a complex value
c_chs$(z$)	Changes the sign of its complex argument
c_sum$(z1$,z2$)	Adds two complex numbers
c_sum1$(x,z$)	Adds a real and a complex
c_sum2$(z$,x)	Adds a complex and a real
c_diff$(z1$,z2$)	Subtracts two complex numbers
c_diff1$(x,z$)	Subtracts a complex from a real
c_diff2$(z$,x)	Subtracts a real from a complex
c_prod$(z1$,z2$)	Multiplies two complex numbers
c_prod1$(x,z$)	Multiplies a real and a complex
c_prod2$(z$,x)	Multiplies a complex and a real
c_quot$(z1$,z2$)	Divides two complex numbers
c_quot1$(x,z$)	Divides a real by a complex
c_quot2$(z$,x)	Divides a complex by a real
c_pwr$(z1$,z2$)	Raises a complex to a complex power
c_pwr1$(x,z$)	Raises a real to a complex power
c_pwr2$(z$,x)	Raises a complex to a real power
c_sqr$(z$)	Square root of a complex number
c_sqr1$(x)	Square root of a real
c_exp$(z$)	e to a complex power

c_log$(z$)	Natural logarithm of a complex number
c_log2$(z$)	Log base 2 of a complex number
c_log10$(z$)	Log base 10 of a complex number
c_sin$(z$)	Sine of a complex number
c_cos$(z$)	Cosine of a complex number
c_tan$(z$)	Tangent of a complex number
c_conj$(z$)	Complex conjugate

The following subroutines carry out the complex matrix operations. In all cases, the last argument is the result matrix or value.

SUB c_MatComp (A(,), B(,), C$(,))	Complex composition, A is the real part, B the imaginary part
SUB c_MatSum (A$(,), B$(,), C$(,))	Add matrices
SUB c_MatDiff (A$(,), B$(,), C$(,))	Subtract matrices
SUB c_MatProd (A$(,), B$(,), C$(,))	Multiply matrices
SUB c_MatPrint (A$(,))	Print matrix
SUB c_MatIdn (A$(,))	Make identity square matrix
SUB c_MatCon (A$(,))	Set entries to one
SUB c_MatZer (A$(,))	Set entries to zero
SUB c_VecCon (A$())	Set entries to one
SUB c_VecZer (A$())	Set entries to zero
SUB c_VecConj (V$(), W$())	Complex conjugate of elements
SUB c_MatConj (V$(,), W$(,))	Complex conjugate of elements
SUB c_MatTRN (A$(,), B$(,))	Transpose
SUB c_MatScmC (k$, A$(,), B$(,))	Scalar multiply by k$, complex
SUB c_MatScmR (k, A$(,), B$(,))	Scalar multiply by k, real
SUB c_MatInv (A$(,), B$(,))	Invert square matrix
SUB c_VecComp (V(), W(), V$())	Convert vector
SUB c_VecSum (V$(), W$(), Z$())	Add vectors
SUB c_VecDiff (V$(), W$(), Z$())	Subtract vectors
SUB c_VecScmC (k$, V$(), Z$())	Scalar multiply by k$, complex
SUB c_VecScmR (k, V$(), Z$())	Scalar multiply by k, real

SUB c_VecPrint (V$())	Print vector
SUB c_MatVecProd (A$(,),W$(),Z$())	Multiply matrix times vector
SUB c_VecMatProd (W$(),A$(,),Z$())	Multiply vector times matrix
SUB c_DotProd (V$(), W$(), n$)	Dot product

The following array functions are also available.

c_MatIsEqual (A$(,), B$(,))	Returns 1 if A$ = B$, 0 otherwise
c_VecIsEqual (V$(), W$())	Returns 1 if V$ = W$, 0 otherwise

Error Messages

210, "Division by 0."

220, "0 to a complex power."

230, "0 to negative power"

240, "Can't get Angle from (0,0)."

400, "Illegal number for decimals."

500, "Dimensions do not match for matrix arithmetic."

501, "Lower bounds not equal for vector operation."

505, "Must be square matrix"

510, "Determinant is 0"

December 11, 2000