6.6 A user function

The calling sequences for the four subroutines required to create a user component are:

```
      SUBROUTINE UINFO(IPAR, CNAME, NPAR)
      INTEGER   IPAR, NPAR
      CHARACTER CNAME*(*)
C---
C IPAR    I    The parameter number.
C CNAME     O  The name of the parameter IPAR.  Note if IPAR=0, then
C              -return the name of the model.
C NPAR      O  The number of parameters in your user model.
C---
C*********
      SUBROUTINE ULIMIT(PVAL, PLIM, NT, NPAR)
      REAL      PVAL(*), PLIM(3,*)
      INTEGER   NT, NPAR
C---
C PVAL(*)   I/O  The current parameter values
C PLIM(1,*) I    If <0 then the corresponding parameter is frozen
C NT        I    Pointer to first parameter value in array PVAL(*)
C NPAR      I    Number of parameters
C---
C*********
      REAL FUNCTION UFNY(X, PVAL, NT, NPAR)
      REAL      X, PVAL(*)
      INTEGER   NT, NPAR
C---
C X       I    The current X value
C PVAL    I    The current parameter values
C NT      I    Pointer to first parameter value in array PVAL(*)
C NPAR    I    Number of parameters
C---
C*********
      SUBROUTINE UDERIV(X, PVAL, PLIM, DERIV, NT, NPAR)
      REAL      X, PVAL(*), PLIM(3,*), DERIV(*)
      INTEGER   NT, NPAR
C---
C X       I    The current X value
C PVAL    I    The current parameter values
C PLIM    I    The constraints array
C DERIV     O  The calculated derivative
C NT      I    Pointer to first parameter value in array PVAL(*)
C NPAR    I    Number of parameters
C---
```

The file `XANADU:[LIB.UFNY]UFNYDEMO.FOR` contains the source code for the built-in `DEMO` user component. You should copy that file and use it as a template for any file that you create.

When FIT starts, it calls UINFO with `IPAR` set equal to 0 to obtain the name of the user component and the number of parameters. This component name will be included in the names of the built-in components, and therefore, should not match any existing component name (such as `CONS`, `LINE`, *etc.*). If the user component is selected, then UINFO will be called for each parameter to obtain the name of that parameter.

ULIMIT is always called after any parameter values have been changed and before UFNY is called. The purpose of ULIMIT is twofold. First, it should check the parameter values in `PVAL` and adjust any that may cause a problem in UFNY (for example, if UFNY divides by a parameter value, then ULIMIT should ensure that the parameter does not equal zero). Second, ULIMIT can be used to set up any initial data that UFNY needs. Since, UFNY is often called many times with the same parameter set, this can result in an increase in speed. The parameter values are stored in `PVAL(NT)` to `PVAL(NT+NPAR-1)`. The `PLIM` array contains `SIG`, `PLO`, and `PHI`. If `PLIM(1,I)` is less than zero, then that parameter is frozen and you should not adjust the parameter value. Also, if `PLIM(2,I)<PLIM(3,I)`, then an effective range is active and you should not adjust a parameter outside that range.

UFNY is the function that actually calculates user component at location `X` with parameter values given by `PVAL`.

UDERIV should calculate the derivative of the UFNY function with respect to each parameter. The version contained in `UFNYDEMO.FOR` evaluates the derivative numerically and hence you may be able to use it without modification. If you use that method you should try to scale the problem so that parameter values are in the range .1-100; values outside this range work, but the convergence can be slower.

If `PLIM(1,I)=-1` then that parameter is frozen and hence you do not need to calculate the derivative. If `PLIM(1,I)<-1` then the parameter has been set equal to another parameter and you should calculate the derivative in the normal manner (the FIT routine assumes that the derivative has been correctly calculated).

If you are able to compute the analytic derivative of your function with respect to the parameter values, then you should use it, because an accurate derivative can greatly improve the fitting process. NOTE: slow convergence is most often due to the derivative being incorrectly calculated. If you find that χ^2 drops slowly, and that FIT is unable to precisely locate the minimum, then you should carefully check both your equations and the UDERIV implementation for typical errors, such as an incorrect sign.

Once you have a working function, you should test it in PLT. Use the `MOdel ?` command to see whether your component is listed. If not make sure you have linked a new version, and that your are running that new version. Next define a model that is composed only of your new component, and enter a resonable set of parameters. Do not attempt to fit at this time, but rather just plot the data and model. Use the `Fit Plot 200` command to ensure that the function is evaluated at 200 points over the visible range. Is the plotted function what you expected? If it is not then you should carefully examine your code.

Once the function is doing what you expect, then you can try to fit it. If certain parameter values can cause a program crash, then you should write a version of ULIMIT that prohibits these values.

Chapter 7

COD

7.1 Introduction

The `COD` program has been designed to fill two roles. First, it can be used as a programmable calculator. In this mode you can use the computer to do simple calculations (on days that your calculator is down). Second, it is designed to assist in developing and testing COD functions that can be used as components in PLT models.

This chapter assumes that you want to create a COD file that can be used with PLT. If you have no previous experience with COD, then you should start by running the `COD` program and learning how to use the stack and various built-in functions. (In COD, functions are sometimes called *words.*) Next you should create and use some simple *colon definitions* within `COD` itself. Colon definitions are the way one creates new functions. COD files contain ASCII text in the same form as you would type in interactive mode. A file that can be used as a model component by PLT is nothing more than a COD file containing a colon definition and supporting code. The `COD` program provides tools for reading and testing functions contained in COD files.

7.2 Interactive mode

The best way to learn about COD is to run the `COD` program and experiment. This can be done with

```
$ COD
Type HELP for help.

COD>
```

When COD starts, it first prints information on how to get help, followed by a blank line. The blank line actually displays the contents of the stack, which is initially empty. Finally, you get the "`COD>`" prompt. At this prompt you can type `HElp` to obtain interactive help on the various commands and how to use them.

The first thing you will want to do is to enter a number into the stack. This is done by typing the number and then pressing the Return key. For example, to enter the number 2 into the stack,

```
COD> 2
 2.0
```

COD echos the stack and then returns the COD prompt (in this documentation the final prompt is not shown). To execute a simple mathematical function, enter all the numbers required by the function and then the function itself. The following sequence shows how to multiply the previously entered `2` by the number `3` to obtain `2*3`:

```
 2.0
COD> 3
 2.0 3.0
COD> *
 6.0
```

With COD it is not necessary to enter one token (number or function) per line. Tokens may be entered, separated by spaces, on a single line. Hence, to divide the result of the previous calculation by `0.5`, enter

```
 6.0
COD> .5 /
 12.0
```

COD contains several commands to manipulate the stack. Thus `SWap` will swap the top two numbers on the stack, and `DUP` will duplicate the top number on the stack. When using COD interactively, you will sometimes wish to clear out the stack. The can be done using the `ABOrt` command. Thus,

```
 12.0
COD> ABOrt
```

and the blank line indicating an empty stack will again appear just before the following COD prompt. There are a large number of built-in COD functions. To obtain a list of the functions, you may use the `List Dictionary` command. If you see a function and would like more information on what it does, you should use the `HElp Dictionary` command. For a complete list of built-in COD commands, consult Appendix A.

7.3 Colon definitions

When running COD interactively, it is sometimes necessary to enter the same sequence of tokens several times. For such cases, you should create a colon definition that contains the sequence. A colon definition consists of a colon `:` , followed by the function name, followed by the sequence of COD functions that you wish to execute when the function name is typed, and terminated with a semi-colon `;` . For example, although there is no built-in COD function to square a number, you can create one with

```
COD> : X2 DUP * ;
```

which will have the effect of multiplying the top number on the stack by itself. After you have defined `X2`, it may be used in exactly the same way as any built-in function; thus,

```
COD> 3.0 X2
 9.0
```

A previously-defined colon function may be used in the definition of a new colon function. An `X3` function, for example, can be constructed from the `X2` function with

```
COD> : X3 DUP X2 * ;
 9.0
COD> 3.0 X3
 9.0 27.0
```

The name of a colon function is not allowed to match the name of any built-in function or other colon function. Hence, a colon function cannot be used to redefine the action of any existing COD keyword.

It is possible to enter a multi-line colon definition interactively. While in the midst of a multi-line colon definition, the stack will not be printed just before the COD prompt. The following example shows one way to enter a colon definition that prints the integers from one to five:

```
COD> : COUNT5
COD>   5 1 FOR
COD>          I .
COD>       LOOP
COD> ;
 9.0 27.0
COD> COUNT5
 1.0
 2.0
 3.0
 4.0
 5.0
 9.0 27.0
```

The last line, just before the next COD prompt, is the stack. This allows us to verify that the original stack has been changed, and therefore, `COUNT5` is not altering the stack.

7.4 COD files

COD provides a way to read commands from a disk file. These files contain the same commands that you would enter *via* the interactive mode. Commands not contained in a colon definition will execute as the file is read. In order to use a COD file as a model component in PLT, it is necessary that the file contain at least one colon definition and it is the last colon definition that will be called when the component is evaluated. As an example of a COD file, assume the file `LINE.COD` contains the following lines:

```
 ! COD program to calculate a line.
 ! P1 + P2*X
 : LINE  P1 X P2 * + ;
```

All lines that begin with '!' are considered comment lines and are ignored by COD. The third line contains the program itself. `P1` and `P2` refer to the parameters that will be adjusted to minimize χ^2. When these words execute, they will load the value of the corresponding parameter into the stack. When writing COD programs that use parameters, you must use consecutive numbers starting with one — *i.e.*, do not leave any holes in the sequence. The keyword `X` is used to push the current value of x into the stack.

It is also possible to use the COD program to read and test the code found in a COD file. The following example demonstrates how this can be done:

```
COD> GET LINE  ! Read the test file, note comment lines are echoed.
 ! COD program to calculate a line.
 ! P1 + P2*X
 NTERMS= 2
COD> NEW 1 2.  ! Set parameter 1 to 2.0

COD> NEW 2 1.  ! Set parameter 2 to 1.0

COD> 5         ! Place the number 5.0 in the stack
 5.0
COD> RUN       ! Run the program with an X value of 5.0
 7.0           ! The final result
```

The RUN command reads the top number on the stack, makes it the x value, clears the stack, and then runs the last program in memory. All these steps ensure that COD is in the same state as it will be when called from the PLT routine.

The Single step command can be used to debug a COD function. In the above example, instead of typing RUN, you could have entered Single Init. This would have read the top number in the stack, made it the x value, cleared the stack, and then executed the first step in the LINE colon definition. When using Single step, COD echos a line that contains three columns of information. The first column is the memory location that is about to be executed. The second column contains the encoded command which can be ignored. The third column is the decoded command that is to be executed. When taking single steps, the stack is still be printed just before the COD prompt appears. Hence, you can watch each step and its effect on the stack.

Code not contained in a colon definition will execute as the COD file is being read. Assume the FUNC.COD file contains the following lines:

```
 VAR 2PI
 2 PI * 2PI STO
 : FUNC X 2PI RCL * P1 / COS ;
```

While this file is being read, the variable 2PI is created and loaded with the value of 2π. The function FUNC can now access and use this variable.

7.5 Other stack-oriented languages

People who have used the Forth, Postscript, and/or HP calculators will recognize certain similarities with COD. This is partly by accident, since all these languages were designed to solve the problem of making a very fast interpreted language. Given the similarity, it would be pointless for similar functions to be implemented differently in COD. Although there is no standard stack-oriented language, when compared to HP calculator languages, or to the Postscript language, the deficiencies/limitations of Forth are serious. In addition, it is the author's opinion that more people know about HP calculators and Postscript, then about Forth. For these reasons, Forth will no longer be considered to be a model for COD. It is the intention of the current author to pattern new COD functions after existing functions from other languages. Currently, HP calculator language, as implemented on the HP48, is the most powerful of such languages and will be examined first for models of new COD functions.

Appendix A

COD Command summary

*

Multiply the top two numbers in the stack.

Example:

```
 5.0 2.0
COD> *
 10.0
```

+

Add the top two numbers in the stack.

Example:

```
 5.0 2.0
COD> +
 7.0
```

+LOOP

Terminate a COD `FOR` loop. When this statement executes, the number at the top of the stack is added to the current index. The loop terminates when the index passes the limit value. The `+LOOP` statement allows for loops in which the index value can either increase or decrease. This word can only be used in colon definitions.

Example:

```
COD> : TMP 0 2 FOR I . -1 +LOOP ;

COD> TMP
 2.0
 1.0
 0.0

COD> : DOUBLE 100 1 FOR I . I +LOOP ;
```

```
COD> DOUBLE
 1.0
 2.0
 4.0
 8.0
 16.0
 32.0
 64.0
```

+STO

Add the previous number to the number stored at the address given at the top of the stack. Although it is easy to determine the address associated with a given variable, and hence use that address directly, it is advisable always to use a variable name to load an address into the stack before using `+STO`.

Example:

```
 0.0
COD> VAR TMP 5 TMP STO TMP RCL
 0.0 5.0
COD> TMP +STO
 0.0
COD> TMP RCL
 0.0 10.0
```

−

Subtract the top two numbers in the stack.

Example:

```
 5.0 2.0
COD> -
 3.0
```

.

Print the number at the top of the stack, and decrement stack pointer by one. The sequence "`DUP .`" can be inserted anywhere into COD functions to print the number at the top of the stack. This may help you figure out what the function is doing.

Example:

```
 1.0 2.0 3.0 4.0 5.0
COD> .
 5.0
 1.0 2.0 3.0 4.0
```

/

Divide the previously entered number by the number on the top of the stack.

Example:

```
 5.0 2.0
COD> /
 2.5
```

/MOD

Replace the top two numbers in the stack with the remainder and quotient of the previous number divided by the top number in the stack.

Example:

```
 23.1 10.0
COD> /MOD
 3.1  2.0
```

0<

Replace the top number in the stack with 1.0 if it is less than zero, 0.0 otherwise.

Example:

```
 -1.0
COD> 0<
 1.0
```

0=

Replace the top number in the stack with 1.0 if it equals zero, 0.0 otherwise.

Example:

```
 -1.0
COD> 0=
 0.0
```

0>

Replace the top number in the stack with 1.0 if it is greater than zero, 0.0 otherwise.

Example:

```
 -1.0
COD> 0>
 0.0
```

1+

Add one to the number at the top of the stack.

Example:

```
 3.0 5.0
COD> 1+
 3.0 6.0
```

1−

Subtract one from the number at the top of the stack.

Example:

```
 3.0 5.0
COD> 1-
 3.0 4.0
```

1/

Compute the inverse of the top number in the stack.

Example:

```
 5.0 2.0
COD> 1/
 5.0 0.50
```

2+

Add two to the number at the top of the stack.

Example:

```
 3.0 5.0
COD> 2+
 3.0 7.0
```

2−

Subtract two from the number at the top of the stack.

Example:

```
 3.0 5.0
COD> 2-
 3.0 3.0
```

:

Begin a new colon definition. In COD, colon definitions define new dictionary words (*i.e.*, new functions). The token following the : is taken to be the name of the function. All words typed after the : are compiled (stored) into memory. A semicolon ; terminates the colon definition and returns the state from compile to execute mode. The name of the function must not match any existing COD keyword. The interactive COD program does not print the stack when the internal state is compiling (*i.e.*, during a colon definition).

Example:

```
 0.0
COD> : X2 DUP * ;
 0.0
COD> 5
 0.0 5.0
COD> X2
 0.0 25.0
```

;

Terminate the current colon definition and return state from compile to execute mode. (See the : topic for an example.)

<

Replace the top two numbers in the stack with 1.0 if the previous number is less than the top number, 0.0 otherwise.

Example:

```
 1.0 2.0
COD> <
 1.0
```

=

Replace the top two numbers in the stack with 1.0 if the numbers are equal, 0.0 otherwise.

Example:

```
 1.0 2.0
COD> =
 0.0
```

>

Replace the top two numbers in the stack with 1.0 if the previous number is greater than the top number, 0.0 otherwise.

Example:

```
 1.0 2.0
COD> >
 0.0
```

?

Display the number stored at the address given at the top of the stack. Although it is easy to determine the address associated with a given variable, and hence use that address directly, it is advisable always to use a variable name to load an address into the stack before using `?` .

Example:

```
 1.0 2.0
COD> VAR TMP 5 TMP STO
 1.0 2.0
COD> TMP ?
 5.0
 1.0 2.0
```

?Dup

Duplicate the number at the top of the stack only if it is non-zero.

Examples:

```
 5.0 0.0
COD> ?DUP
 5.0 0.0
COD> 1.0 ?DUP
 5.0 0.0 1.0 1.0
```

A2tn

Compute the arctangent assuming the top two numbers in the stack represent an x,y pair.

Example:

```
 1.0 2.0
COD> A2tn
 1.570796
```

ABOrt

Reset the stack pointer. This deletes all numbers in the stack and can be very useful in the interactive mode to clean out the stack. If this command occurs when a COD function is executing, the function exits and a `NO` data function value is returned.

ABS

Take the absolute value of the top number in the stack.

Example:

```
 5.0 -2.0
COD> ABS
 5.0 2.0
```

ACos

Computer the arccosine (in radians) of the top number in the stack.

Example:

```
 5.0 0.5
COD> ACos
 5.0 1.047198
```

ALog

Compute 10. raised to the power of the top number in the stack.

Example:

```
 5.0 2.0
COD> ALog
 5.0 100.0
```

ASin

Compute the arcsine (in radians) of the top number in the stack.

Example:

```
 5.0 0.5
COD> ASin
 5.0 0.5235988
```

ATan

Compute the arctangent (in radians) of the top number in the stack.

Example:

```
 5.0 0.5
COD> ATan
 5.0 0.4636476
```

BEGIN

Begin a `BEGIN...UNTIL` or a `BEGIN...WHILE...REPEAT` structure. See either the `UNTIL` or `WHILE` topics for examples of use. This word can only be used in colon definitions.

COS

Compute the cosine of the top number (in radians) in the stack.

Example:

```
 5.0 0.5
COD> COS
 5.0 0.8775826
```

DDms

Convert a number in decimal degrees to the form DDDMMSS.S .

Example:

```
 12.5
COD> DDms
 123000.0
```

DEPth

Leave the number of values contained in the stack (not counting the result).

Example:

```
 5.0 4.0
COD> DEP
 5.0 4.0 2.0
```

DMsd

Convert a number of the form DDDMMSS.S to decimal degrees.

Example:

```
 123000.0
COD> DMsd
 12.5
```

DRop

Drop the stack pointer so that the number at the top of the stack is lost.

Example:

```
 5.0 4.0
COD> DRop
 5.0
```

DTor

Convert decimal degrees to radians.

Example:

```
 5.0 90.0
COD> DTor
 5.0 1.570796
```

DUP

Duplicate the number on the top of the stack (equivalent to `1 PICK`).

Example:

```
 5.0 2.0
COD> DUP
 5.0 2.0 2.0
```

ELSE

If the condition was false when the `IF` statement executed, then transfer control to the first statement following the `ELSE`. If the condition was true, then the function executes code down to the `ELSE` statement and then skips to the first statement following the `THEN` statement. This statement must be preceded by an `IF` statement. This word can only be used in colon definitions. See the IF topic for an example.

EXIT

Immediately terminate the current colon function. Since the stack is unaffected, check to verify that the stack is left in the same state no matter how the colon function is terminated (otherwise obscure and nasty bugs result). This word can only be used in colon definitions.

EXP

Compute the exponential of the top number in the stack.

Example:

```
 5.0 1.0
COD> EXP
 5.0 2.718282
```

FOR

Set up a `FOR...LOOP` or a `FOR...+LOOP` structure using the top two numbers in the stack to denote the range. The index for the `FOR` loop is an `INTEGER*2`, hence the maximum value is 32767. This word can only be used in colon definitions.

Example:

```
COD> : TMP 4 1 FOR I . LOOP ;

COD> TMP
 1.0
 2.0
 3.0
 4.0
```

Warning to people who know Forth: The COD `FOR` statement is similar to the Forth `DO` statement; however, there is an important difference concerning the two numbers that precede the `FOR`. In COD, these represent the upper and lower index values, respectively; in Forth, the first number is one greater than the upper index value.

HCos

Compute the hyperbolic cosine of the top number in the stack.

Example:

```
 5.0 0.5
COD> HCos
 5.0 1.127626
```

HSin

Compute the hyperbolic sine of the top number in the stack.

Example:

```
 5.0 0.5
COD> HSin
 5.0 0.5210953
```

HTan

Compute the hyperbolic tangent of the top number in the stack.

Example:

```
 5.0 0.5
COD> HTan
 5.0 0.4621172
```

I

Push the index value of the innermost `FOR` loop onto the stack. This statement can only be used inside `FOR` loops inside a colon definition. See the `FOR` topic for an example of use.

IF

If the condition is true, execute statements up to the corresponding `ELSE` or `THEN` statement. If the condition is false, pass control to the first statement following the `ELSE` if it exists, otherwise to the first statement following the `THEN`. `IF` structures can be nested. This word can only be used in colon definitions.

Example:

```
 X 1 <
 IF
  ! do these statements if X<1
 THEN

 X 1 >
 IF
  ! do these statements if X>1
 ELSE
  ! do these statements if X<=1
 THEN
```

INT

Compute the integer portion of the top number in the stack.

Example:

```
 5.0 0.9
COD> INT
 5.0 0.0
```

J

Push the index value of the next outer `FOR` loop onto the stack. This statement can only be used inside `FOR` loops inside a colon definition.

Example:

```
COD> : TMP 2 1 FOR
COD>           5 4 FOR
COD>                J . LOOP LOOP ;

COD> TMP
 1.0
 1.0
 2.0
 2.0
```

LEAVE

Cause an immediate exit from the current loop. No statements from the `LEAVE` statement to the loop terminator are executed. This statement resembles the Fortran 8x

EXIT statement. `LEAVE` can only be used inside `FOR` loops inside a colon definition.

LN

Take the natural logarithm of the top number in the stack.

Example:

```
 5.0 2.0
COD> LN
 5.0 0.6931472
```

LOG

Take the base-10 logarithm of the top number in the stack.

Example:

```
 5.0 2.0
COD> LOG
 5.0 0.3010300
```

LOOP

Terminate a COD `FOR` loop. When this statement executes, one is added to the current index value. If the current index is less than or equal to the maximum index, then control is transferred to the first statement following the corresponding `FOR` statement. Otherwise, control passes to the statement following the `LOOP` statement. This word can only be used in colon definitions. See the `FOR` topic for an example of use.

MAX

Replace the top two numbers in the stack with the maximum of the two numbers.

Example:

```
 1.0 3.0 5.0
COD> MAX
 1.0 5.0
```

MIN

Replace the top two numbers in the stack with the minimum of the two numbers.

Example:

```
 1.0 3.0 5.0
COD> MIN
 1.0 3.0
```

MOD

Compute the value of the previous number modulo the top number in the stack.

Example:

```
 2.0 3.0
COD> MOD
 2.0
COD> ABO 3.0 3.0
 3.0 3.0
COD> MOD
 0.0
```

NEG

Negate the top number in the stack.

Example:

```
 5.0 2.0
COD> NEG
 5.0 -2.0
```

NInt

Compute the nearest integer to the top number in the stack.

Example:

```
 5.0 0.9
COD> NInt
 5.0 1.0
```

NOT

Replace the top number in the stack with 1.0 if it is zero, 0.0 otherwise.

Example:

```
 1.0
COD> NOT
 0.0
```

OVer

Duplicate the second number on the stack (equivalent to `2 PICK`).

Example:

```
 5.0 4.0
COD> OVER
 5.0 4.0 5.0
```

PI

Push the value of π into the stack.

Example:

```
 5.0
COD> PI
 5.0 3.141593
```

PICK

Duplicate the nth number on the stack (not counting n itself), where n is the top number on the stack. Note: `1 PICK` does the same thing as `DUP`, and `2 PICK` does the same thing as `OVER`.

Example:

```
 100. 200. 300. 400.
COD> 3 PICK
 100. 200. 300. 400. 200.
```

RCL

Replace the address at the top of the stack with the number at that address. See the `STO` topic for an example of how `RCL` is used.

REPEAT

Terminate a `BEGIN...WHILE...REPEAT` loop. When this statement executes, control is always passed to the first statement following the `BEGIN` statement. This word can only be used in colon definitions. See the `WHILE` topic for an example of use.

ROLL

Rotate the nth number (not counting n itself) to the top of the stack, where n is the top number in the stack. Note: `2 ROLL` does the same thing as `SWAP`, and `3 ROLL` does the same thing as `ROT`; and n must be greater than 1.0.

Example:

```
 100. 200. 300. 400.
COD> 4 ROLL
 200. 300. 400. 100.
```

ROT

Rotate the third number to the top of the stack (equivalent to `3 ROLL`).

Example:

```
 100. 200. 300. 400.
COD> ROT
 100. 300. 400. 200.
```

RTod

Convert radians to decimal degrees.

Example:

```
 5.0 1.0
COD> RTod
 5.0 57.29578
```

SIN

Compute the sine of the top number (in radians) in the stack.

Example:

```
 5.0 0.5
COD> SIN
 5.0 0.4794255
```

SQrt

Compute the square-root of the top number in the stack.

Example:

```
 5.0 2.0
COD> SQrt
 5.0 1.414214
```

STO

Store the previous number at the address given at the top of the stack. Although it is easy to determine the address associated with a given variable, and hence use that address directly, it is advisable always to use a variable name to load an address into the stack before using `STO`.

Example:

```
 1.0
COD> VAR TMP
 1.0
COD> 5 TMP STO
 1.0
COD> TMP RCL
 1.0 5.0
```

SWap

Swap the top two numbers on the stack (equivalent to `2 ROLL`).

Example:

```
 5.0 2.0
COD> SWap
 2.0 5.0
```

TAN

Compute the tangent of the top number (in radians) in the stack.

Example:

```
 5.0 0.5
COD> TAN
 5.0 0.5463025
```

THEN

Terminate an `IF` structure. This word can only be used in colon definitions. See the `IF` topic for an example of use.

TSig

Transfer the sign of the top number in the stack to the absolute value of the previous number.

Example:

```
 2.0 -5.0
COD> TSig
 -2.0
```

UNTIL

Terminate a `BEGIN...UNTIL` block. `UNTIL` pops one number off the stack. If that number is false (=0.0), the function jumps back to the first statement following the `BEGIN` statement. Otherwise execution continues with the statement following the `UNTIL` statement. This word can only be used in colon definitions.

Example:

```
COD> VAR Y                ! Create variable Y

COD> : TEST 2 Y STO       ! Initialize Y to 2
COD> BEGIN
COD>   Y RCL .            ! Print the value of Y
COD>   Y RCL DUP * Y STO  ! Square it and store new value
COD>   Y RCL 1.E10 >      ! Test condition
```

```
COD> UNTIL ;

COD> TEST
 2.0
 4.0
 16.0
 256.0
 4.2949673E+9
```

VAR

Define the following token to be a new variable name. The variable name is not allowed to match any existing COD keyword. When that variable is used in the future, it will cause the address of that variable to be loaded into the stack (for use with a following `STO` or `RCL` command).

Example:

```
 0.0
COD> VAR 2PI 2 PI * 2PI STO
 0.0
COD> 2PI RCL
 0.0 6.283185
```

WHILE

The `WHILE` statement pops one number off the stack. If that number is true ($\neq 0.0$), the statement following the `WHILE` will be executed. If the number is false, then control passes to the first statement following the `REPEAT` statement. This statement can only be used inside a `BEGIN...WHILE...REPEAT` loop in a colon definition.

Example:

```
COD> VAR Y                   ! Create variable Y

COD> : TEST 2 Y STO          ! Initialize Y to 2
COD> BEGIN
COD>   Y RCL .               ! Print the value of Y
COD>   Y RCL 1.E10 < WHILE   ! Recall number and test for <1.E10
COD>   Y RCL DUP * Y STO     ! Square and store new value
COD> REPEAT ;

COD> TEST
 2.0
 4.0
 16.0
 256.0
 4.2949673E+9
 1.8446744E+19
```

X

Push the current X value into the stack.

^

Raise the previously entered number to the power given by the top number in the stack.

Example:

```
 5.0 2.0
COD> ^
 25.0
```