Console Programmable RPN Calculator CPRCA

By
Namir Clement Shammas

Contents

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Copyright © 2015, 2016 by Namir Clement Shammas

Dedication

To my son Joey who is my pride and joy!
To my beloved wife Sherry. What an awesome lady!
To my beloved grandchildren, Tyler, Oshby, Owen, and Gracie

Copyright © 2015, 2016 by Namir Clement Shammas

Introduction

This technical presentation discusses the operations and features of a *console programmable RPN calculator application.* I wrote it in Visual Basic 2015 Community Edition. I call it *CPRCA* for short. These initials should not be confused with the existing disease *Constitutional Pure Red Cell Aplasia*! I often refer to CPRCA application as simply, *the application*. The core of the CPRCA is based on the HP–41C calculator. I have added many new features, enhanced many existing ones, and left out a few HP–41C functions and commands. Due to time limitation, this document is *relatively* brief. I hope that I am providing you with enough information and sample programs to make it worth tinkering with.

I have put a lot of work and my best effort in creating the CPRCA application. However, I make no warranties explicit or implied about the fitness of the work. This application is developed for programming fun. You are using the CPRCA application at your own risk. I am not responsible for any damages that may result from using the CPRCA application.

Basic Modes of Operations

The application runs as a console program with text menus typical of legacy DOS programs. The application displays a main menu with a numbered option list. You can exit the program by choosing option 0 or select one of the operations that appear in the next subsections.

RPN Calculator Mode

In the RPN calculator mode you enter an RPN expression on a single line and press the Enter key. The application remains in calculator mode until you just press Enter when prompted for an RPN expression. Use the space character to delimit the terms in an RPN expression. The CPRCA application parses your RPN expression and evaluates it. The RPN expressions must be void of labels, GTO commands, GSB commands, and any other program flow control commands. You have a generous 10,000 memory registers to use. They are numbered 0 to 9999. It is worth pointing out that an RPN expression can, within the same session, use the values stored in the stack and memory registers left by evaluating previous RPN expressions. This feature allows you to break down the calculations of a long equation into parts. Here are a few examples of RPN expressions:

1. 355 113 /
2. 4 3 2 1 + + +
3. Pi 2 * x^2
4. 355 113 / pi – pi / 1e9 *
5. 2 sqrt ln sto 0 x^2 rcl 0 + 5 –
6. Sto 0 sin x^2 rcl 0 cos x^2 +
7. Enter sin x^2 x<>y cos x^2 +

As with physical HP calculators, the CPRCA application replaces the arguments of a function, found in the stack, with the value of the function.

Copyright © 2015, 2016 by Namir Clement Shammas

You enter a sequence of numbers in an RPN expression by simply separating each two numbers with a space. This space replaces using the ENTER key, between two numbers, in physical HP RPN calculators. Only use the ENTER command to push the number in the X register up into the other stack registers. Using a sequence of three ENTER commands fills the four–register stack with the same number.

File Query Mode

The CPRCA application does not include an IDE (that would be another project all by itself) or an editor to enter the source code for your programs. You need to key in your source code separately, as text, using your favorite text editor. The program files can have any extension. I use .txt and that works fine for me. Using file extensions like .41, .hp41, .cprca, .con41, and the like, is just fine. In case you don't remember the names of the program files, you can ask the application for help. Select option 2 from the main menu to ask the application for help in viewing existing files. The application prompts you to enter the search path (or just press Enter if you want to search in the application's current directory) and a single file extension. The program displays the matching files in batches of ten files. Since this option does not use a GUI File Open dialog box, you will need to know the full path of the source code programs, especially in case they are stored in a directory that differs from the executable files of the CPRCA application. To avoid typing long directory names you can visit the target directory, copy its full path, and then paste that it in the console application in response to the prompt for a path. This is a method that is easy and free of typos.

Program View and Edit Mode

The CPRCA allows you to view program source code and edit its lines using Windows Notepad text editor. The application provides a menu option for this purpose, prompting you to enter the filename you want to access. The input must include the full path if the file is not located in the same folder with the CPRCA's executable files. If the filename you specify has a .txt extension, you can omit that extension. With the correct filename supplied to the application, it invokes Notepad, in normal view mode, with the target file in view. You can then view the program's code, search for text, edit, and replace text.

It is worth pointing out that after you have successfully executed a CPRCA program, the application adds a special menu option. This option allows you to view and edit the file that you ran without having to re-enter its name. You can repeatedly view, edit, save, and rerun the program file.

Copyright © 2015, 2016 by Namir Clement Shammas

Program Execution Mode

The CPRCA application run program files that are text files with no line numbers. Each program line contains one command, a single number, data text, or a comment. In the program execution mode, the application prompts you to enter a program file name to execute. This input requires a full path for the file, if it is not located in the same folder as the application's executable files. If your program has the .txt file extension, you need not enter it. If this operation is successful, the program loads in memory. The application prompts you to select from the following options:

1. Simply press the Enter key to start executing the application at the first line.
2. Enter the question mark (?) to ask the application to display a list of program labels. You can then select the label to execute by enter the label's number in the list.
3. Type in the exclamation character (!) to skip program execution and return to the main prompt.

If you have successfully loaded a program, the application will add an additional menu option that allows you to reload and run the last program.

Copyright © 2015, 2016 by Namir Clement Shammas

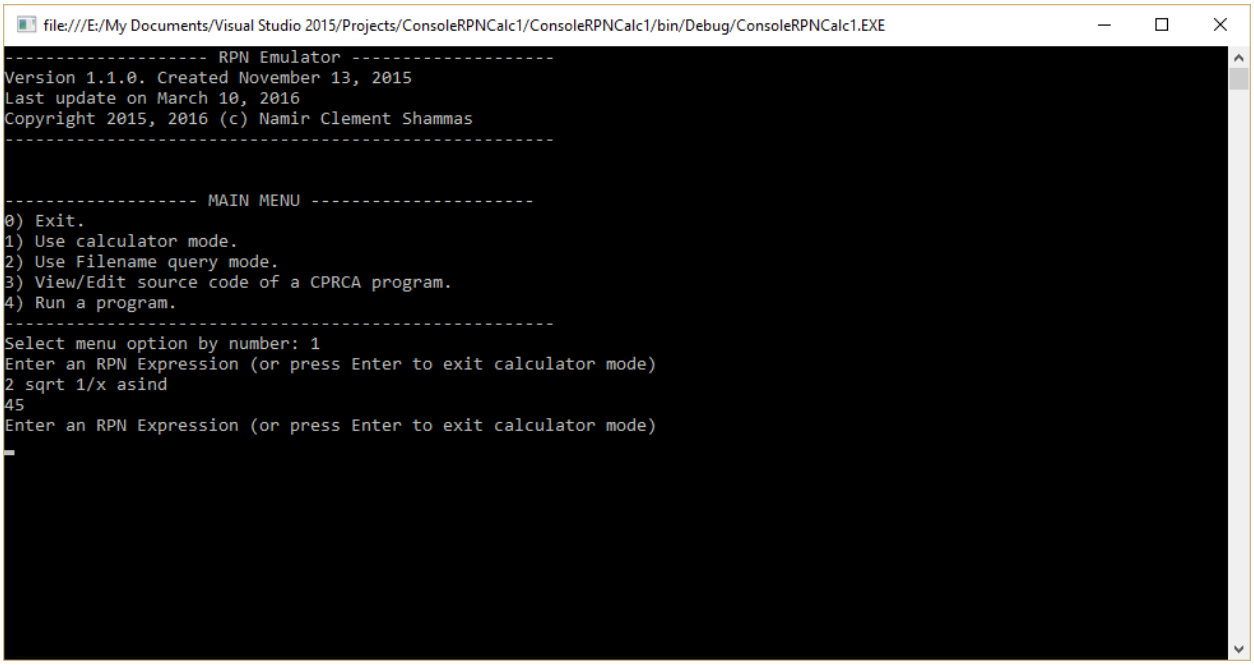


Figure 1. Sample calculator mode session with the Console RPN Calculator application.

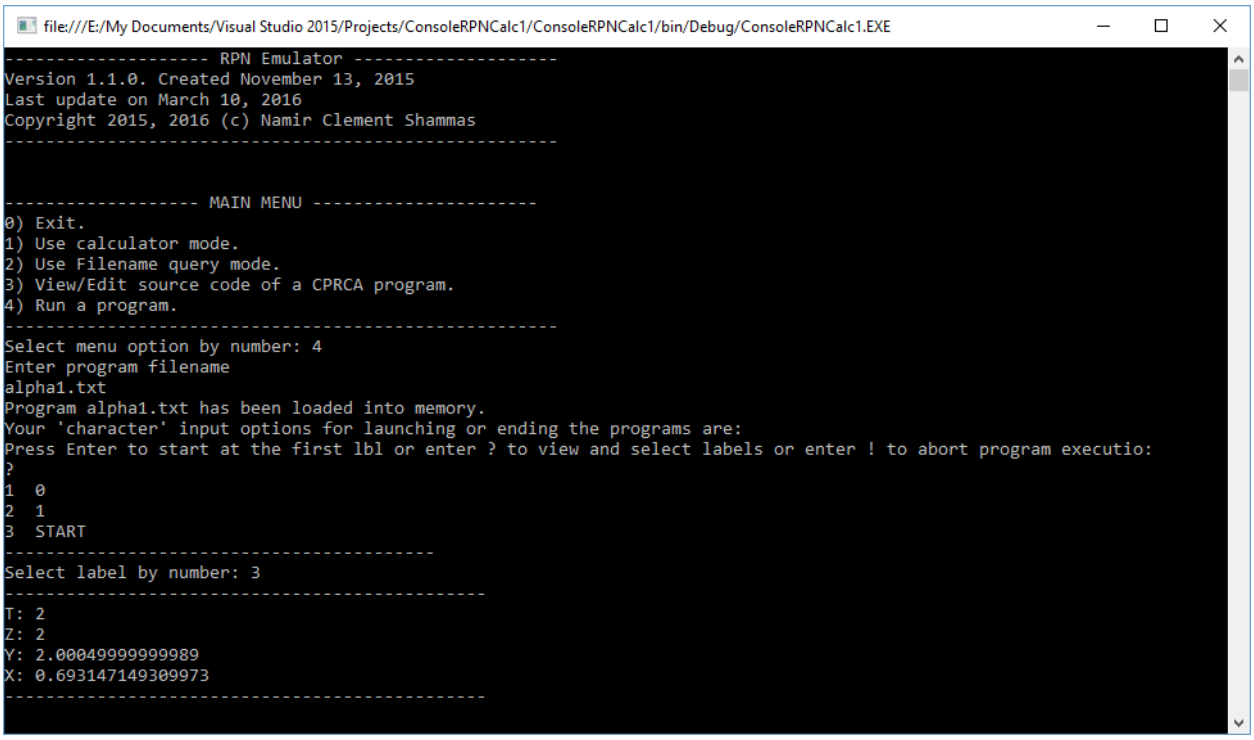


Figure 2. Sample program mode session with the Console RPN Calculator application.

Figure 1 shows a sample session that uses the calculator mode. In this mode we calculate the arc sine, in degrees, of $1/\sqrt{2}$, which is 45 degrees. The RPN expression that obtains this result is "2 sqrt 1/x asind".

Copyright © 2015, 2016 by Namir Clement Shammas

Figure 2 shows a session that executes the program alpha1.txt (which calculates the integral of 1/X from X=1 to X=2) and uses the question mark to display the program's labels. The session shows the selection of the third label, START, as a program execution starting point. This feature works like using the buttons A to J on the HP–41C calculator (in User mode). You can select to trigger specific parts of your program that starts with an LBL and ends with a RTN. In other words, using this feature does not limit the loaded programs from always starting at the first program line. The application displays the contents of all four stack registers when it ends execution. This approach permits you to view multiple results that your program can conveniently place in the stack. To suppress displaying the stack at the end of the program include the command NOSTACK in your program.

The names of commands, labels, and variables are case **insensitive** in the console RPN calculator. Thus, for example, Lbl Start, lbl start, LBL start, Lbl STart, and LBL START all represent the same label declaration. Likewise, the commands, Vsto Xarr. vsto xarr, and VSTO XARR, all store the value of the X register in the same variable, XARR.

The application makes a reasonable effort to detect run–time errors and report them. Your program can use error–handling statements that allow you to:

- Turn on error handling, using ONERRGOTO, and specify a label that acts as an error handler.
- Request to ignore the run–time error, using ONERRRESUME, and execute the next program line.
- Turn off an error–handling mechanism, set by the above two options, using ONERROFF.

The above mechanism for error handling is inspired by Excel VBA error handling. It does not use special flags as does the HP–41C.

Display Formatting

The application supports different display formats. Appendix A lists the display formats that work with the application. The display formats are based on Visual Basic .Net formats, since the application is written in that programming language. Unfortunately, Visual Basic does not support engineering notation. The DSP command allows you to specify and use a display format listed in Appendix A. The F*n* and E*n* formats display numbers in fixed and scientific notations using *n* decimal places. The commands FIX n and SCI n are shortcut commands to set the display to a fixed or scientific mode with n number of decimal places. The commands X*n* and D*n* display *n* integers of the integer values in hexadecimal and integer formats, respectively. If the value of *n* exceeds the number of digits, the application displays leading zeros. The application prefixes hexadecimal integers with the letter x, such as x11. This prefix is very useful for displaying

Copyright © 2015, 2016 by Namir Clement Shammas

hexadecimal integers that lack the hexadecimal digits A to F, since they can easily be mistaken for decimal integers. The command DSP? works both in programs and with RPN expressions. It both cases, it displays the current format string. In programming mode, it also copies the display format to the Alpha register. The command CLRDSP turns off the display format and causes the output to appear unformatted.

Types of Memory Registers

The application supports a wealth of different registers and arrays that well extend beyond the capacity of vintage programmable HP calculators. The next subsections discuss the various types of registers and memory storage.

Floating–Point and Integer Values

Since the application handles integers and real numbers, I need to point out the range of floating–point numbers. The CPRCA application does not support complex numbers. The application supports double–precision floating–point numbers. Since the application is written in Visual Basic 2015 it supports the ranges of the Double type in that language. This type ranges in values from 4.94065645841246544E–324 through 1.79769313486231570E+308 for positive values. Negative values range from –1.79769313486231570E+308 through –4.94065645841246544E–324. Entering numbers, in the calculator and program modes, that lie outside these ranges yield zero values! Dividing small positive numbers that result in values below 4.94065645841246544E–324 end up being zero. A similar statement is true for small negative numbers. Regarding integers, the application uses them mostly as indices for memory registers. It takes the integer part of a number in a stack register. As such you will certainly be within the supported range of the Visual Basic Integer type which is –2,147,483,648 through 2,147,483,647.

The Stack

As expected, the RPN application has a legacy 4–register stack with a full set of backup stack registers. The CPRCA application supports the ability to store and recall values in various stack registers. The application also supports register arithmetic with and without indirect addressing for the stack registers storage commands. The CPRCA application also offers commands to swap between any two stack registers, and not just the X and Y registers. The application backs up the entire stack when performing calculations. Consequently, you can recall the last X, Y, Z, and T registers using the LastX, LastY, LastZ, and LastT commands, respectively. You can display any and all stack registers using an output format. Consult Appendix A for the output formats.

The Alpha Register

Like the HP–41C, the application supports an Alpha register. This register can store a sizable text. The text size is not as limited as the Alpha register in the HP–41C calculator. The application allows you to store and recall text between the Alpha register and special named registers that store text. The contents of the Alpha register (and its copies, stored in special text variables) are case sensitive. The application also offers commands to manipulate the Alpha register as follows:

Copyright © 2015, 2016 by Namir Clement Shammas

1. Replace the Alpha register with leading, trailing, and middle characters.
2. Trim leading and trailing spaces in the Alpha register.
3. Convert the text in the Alpha register to either lowercase or uppercase.
4. Toggle the character case for the text in the Alpha register.
5. Reverse the text in the Alpha register.
6. Search for the first or last occurrence of a substring in the Alpha register.
7. Replace characters in the Alpha register.
8. Filter out user-defined characters from the Alpha register.

The Statistical Registers

The CPRCA application supports a set of statistical registers, which are separate from any other memory group. These registers keep track of the number of observations and the sum of X, sum of X squared, sum of Y, sum of Y squared, and sum of X*Y. You can perform the following operations with these registers:

- Clear the statistical registers.
- Add or subtract (X, Y) values in the statistical registers.
- Calculate the mean values for variables X and Y.
- Compute the standard deviations for variables X and Y.
- Calculate the covariance for the variables X and Y.
- Calculate the confidence intervals for the mean and standard deviation for the variables X and Y.
- Calculate the slope, intercept, and coefficient of determination (R^2) for the linear regression model Y = intercept + slope * X. Of course you can transform the original values of X and/or Y to perform a linearized regression.
- Compute projections of X onto Y and of Y onto X based on the most recent linear regression calculations.
- Calculate the confidence intervals for the slope, intercept, coefficient of determination, and the projections of X onto Y.
- Recall the values for the individual statistical summations using the following commands:
    - SUMN returns the number of observations.
    - SUMX returns the sum of X values.
    - SUMX2 returns the sum of X squared.
    - SUMY returns the sum of Y values.
    - SUMY2 returns the sum of Y squared.
    - SUMXY returns the sum of X*Y.
- Write and read the statistical register to/from a text file. The commands SAVESTAT and GETSTAT support these features. These features are only available in programs.
- Merge statistical registers' values store in a file with those in memory. You can divide a large set of data into small groups, get the statistics for the small groups first, then

Copyright © 2015, 2016 by Namir Clement Shammas

merge the small groups into bigger ones and obtain the statistics for the bigger groups. You can apply this scheme for two, three, or more layers of data, depending on the type of research you are doing. The approach for using MERGESTAT, to manage a double layer of data, follows the subsequent method:

- Divide your data into N small groups.
- For each small group i:
  - Clear the statistical registers.
  - Accumulate values of group I in the statistical registers using the command S+.
  - Perform statistical calculations and record the results.
  - Save the values in the current statistical registers to a file using the command SAVESTAT.
- Clear the statistical registers.
- Read the values for the statistical registers in group 1 using the command GETSTAT.
- Merge the statistical registers' data from groups 2 to N, using the command MERGESTAT.
- Perform the calculations on the merged values and record the results.
- Compare the results of the statistical calculations for the N small groups with the results of the large group.

The Main Memory Registers Pool

The application offers ten thousand (10,000) memory registers with indices in the range of 0 to 9999. You access these registers using STO *nn* and RCL *nn*. The application also supports register arithmetic versions of the basic STO and RCL commands. In addition, you can use indirect addressing in the STO and RCL sets of commands. Simply append the letters IND (in any character case) to the STO and RCL commands. Register arithmetic is also available for the STOIND and RCLIND commands. The application uses the absolute values of the memory registers and indirect memory registers in all the STO, STOIND, RCL, and RCLIND commands.

The CPRCA application offers storage and recall commands between various data sources and destinations. Consult appendices B and C for a summary of the basic STO and RCL commands. Using these appendices will help you to correctly select the type of STO and RCL commands for your programs.

Filling Memory Registers with Values

The application offers several MFILLxxx commands to fill a range of memory registers with various kinds of values. All these command use a stack register to specify the range of memory registers. The value that controls the range has the numeric format of aaaa.bbbbcc. This means that you can specify both the range of registers to work with, and the number of registers in that range to skip over. The MFILL command fills a range of memory register with a fixed value. The MFILLSEQ command fills a range of memory registers with a sequence of values. This sequence may contain increasing, decreasing, or even fixed values. These values can be integers

Copyright © 2015, 2016 by Namir Clement Shammas

or floating-point numbers. The commands MFILLRND and MFILLINTRND fill a range of memory registers with uniformly-distributed floating-point and integers, respectively. The command MFILLNROMRND fills a range of memory registers with normally-distributed floating-point values.

Loop Control HP–41C Style

The application supports the ISG41 and DSE41 loop controlling commands. They mimic the ISG and DSE commands that first appeared in the HP–41C calculator. The ISG41 and DSE41 use the three–integers and five–decimals index format, aaa.bbbcc. The ISG41 can loop from aaa (in the range of 0 to 999) to and including bbb (in the range of 0 to 999) in increments of cc (with a minimum of 1). The DSE41 decrements from aaa (which can have more than 3 digits) down to bbb (between 0 and 999) in steps of cc. You are reminded that the DSE41 command has less restriction in the range of values it handles than the ISG41.

The New and Extended Loop Control Commands

The application also supports the ISG and DSE loop controlling commands. They use the wider memory register range of 0 to 9,999 and work with the four–integers and six–decimals format aaaa.bbbbcc. They allow you to loop between aaaa and bbbb in increments/decrements of cc. The ISG can loop from aaaa (in the range of 0 to 9999) to and including bbbb (in the range of 0 to 9999) in increments of cc (with a minimum of 1). The DSE decrements from aaaa (which can have more than 4 digits) down to bbbb (between 0 and 9999) in steps of cc. Please keep this difference in mind when programming with DSE and ISG. I also added the command DSL which is *Decrement and Skip If Less*. This command allows a loop control register to count down from a positive value until and including zero. This scheme makes the value of zero accessible, to the loop control register, in a loop. The DSE command can access all of the memory registers from 9999 down to 1 in that order. By comparison, the DSL command can include 0 to the DSE's lower limit. I also added the DSL41 command that provides an HP–41C flavor of the DSL command.

Copyright © 2015, 2016 by Namir Clement Shammas

Remember that the bbbb values in commands ISG, DSE, and DSL require AT LEAST FOUR DECIMAL PLACES. As an HP-41C programmer you are used to dividing integer limits for ISG and DSE by 1000. You must now divide them by 10000. The application has a predefined constants called SCALE and SCALE41 that you can use. SCALE is equal to 10,000. SCALE41 is equal to 1000. So, for example to loop within memory registers 10 through 5000 using memory register 0 for indirect addressing, perform the following steps:

10	LBL 0
5000	RCLIND 0
SCALE	…
/	ISG 0
+	GTO 0
STO 0	

The HP-41C style version of the above loop that accesses memory registers 10 through a smaller upper limit of 500 is:

10	LBL 0
500	RCLIND 0
SCALE41	…
/	ISG41 0
+	GTO 0
STO 0	

A For–Like Loop Control

The application also supports a BASIC–like FOR loop control command. The BASIC FOR loop has the following syntax:

```
FOR varI = initialValue TO finalValue STEP incValue
```

Where varI is the loop control variable which stores the initial value and its updates. The loop iterates, by changing the value in the loop control variable, from the initialValue to the finalValue in steps of incValue. If the latter value is positive, then the initialValue must be less than or equal to the finalValue for the iterations to occur. Conversely, if the incValue is negative, then the initialValue must be greater than or equal to the finalValue for the iterations to occur. The initialValue, finalValue, and incValue are usually integers and can be floating points. Here is an upward counting FOR loop in BASIC:

```
FOR I = 1 TO 100 STEP 2
```

Copyright © 2015, 2016 by Namir Clement Shammas

The above loop initializes the loop control variable, I, with the value of 1. The loop iterates with values of I progressing in the sequence, 1, 3, 5, 7, …, 97, and 99. The last iteration starts with value in variable I becoming 101. Since it is greater than the final value of 100, the FOR loop bypasses executing the statements in the loop. Program execution resumes after the end of the FOR loop. Here is a downward–counting loop which is the counterpart of the one above:

```
FOR I = 100 TO 1 STEP -2
```

The above loop initializes the loop control variable, I, with the value of 100. The loop iterates with values of I progressing in the sequence, 100, 98, 96, ..., 6, 4, and 2. The last iteration starts with value in variable I becoming 0. Since it is less than the final value of 1, the FOR loop bypasses executing the statements in the loop. Program execution resumes after the end of the FOR loop.

The application offers the command FOR that uses a text variable as a loop control variable. This variable contains the comma–delimited string images of the values for the initialvalue (which become the currentValue after the first loop iteration), finalValue, and incValue. Each time you use command FOR it performs the following tasks:

1. Obtain the values of the initialvalue, finalValue, and incValue from the text variable.
2. Increment/decrement the loop control variable by adding the incValue to the initialValue/currentValue.
3. If the incValue is positive, the command checks if the updated initialValue/currentValue is less than or equal to the finalValue. When this condition is true, the application updates the text in the loop control variable and executes the next program line—usually a GTO command. Otherwise, the application skips the next program line without updating the text in the loop control variable.
4. If the incValue is negative, the command checks if the updated initialValue/currentValue is greater than or equal to the finalValue. When this condition is true, the application updates the text in the loop control variable and executes the next program line—usually a GTO command. Otherwise, the application skips the next program line without updating the text in the loop control variable.

Thus each loop alters the current value (which starts out as the initial value) while maintaining the final and increment values. The application also offers the command FORCL to obtain the current value of the loop control variable. The name of this variable is the argument of the command FORCL. The CPRCA application also offers the command FORSET that allows you to obtain the values for the initialValue, finalValue, and incValue from the stack and store them as comma-delimited text in a string variable.

As an example of using the commands FOR and FORCL to control loop iteration, consider the next program that adds the integers from 1 to 100, in increments of 1. To set up the loop control variable we have 1 as the initial value, 100 as the final value, and 1 as the increment.

Copyright © 2015, 2016 by Namir Clement Shammas

This means that the loop control variable, which I will call incVar, must be initialized with the text “1,100,1” that I first place in the Alpha register using the following program lines:

```
'1,100,1
asto incVar
```

I can also use the command FORSET instead of the command ASTO to assemble the text in variable incVal using the following code:

```
1
100
1
forset incVal
```

The example requires a loop that adds the integers from 1 to 100. Here is the semi–finished code for that loop:

```
lbl start
'1,100,1
asto incVar
0
lbl 0
# get the current value of the loop control variable
+
for incVar
gto 0
end
```

The comment inside the loop (between LBL 0 and GTO 0) states that the loop needs to access the current value of the loop control variable incVar. The command FORCL incVar satisfies this requirement. The complete code is:

```
lbl start
'1,100,1
asto incVar
0
lbl 0
forcl incVar
+
for incVar
gto 0
end
```

The FOR command is more powerful than the ISG, DSE, and DSL commands since it overcomes the limitations of these commands. The initial, final, and increment values used with the FOR command can be integers or floating point numbers and can be either positive or negative numbers! You can also nest FOR commands.

Searching and Sorting Memory Registers

The CPRCA application offers the SORTA and SORTD commands that allow you to sort the memory registers and search for values in them. The sort command allows you to treat some or all of the memory registers as a virtual table or matrix. You define the number of rows and

Copyright © 2015, 2016 by Namir Clement Shammas

columns for that virtual table, as well as the first memory register where it starts. You also select the column in the virtual table whose values are used as key values in the sorting process. The sort command copies the targeted memory registers to a local matrix, sorts the elements of that matrix, then copies the matrix elements back to the source memory registers.

As for searches, the application offers the commands SEARCH, MSEARCH, CSEARCH, and HSEARCH supports forward sequential, backward sequential, middle–first, forward circular, backward circular, forward heuristic and backward heuristic searching. The latter two searches allow you to move the matching element forward or backward, in support of optimistic or pessimistic search schemes, respectively. The search commands (except the heuristic ones) allow you to search for the first memory register that is equal to (or approximately equal to), less than, or greater than a searched value.

The Logical Flags

Logical flags. The application supports 100 flags, indexed 0 to 99. You can set, clear, and test the flags. The tests can be simple—testing if a flag is set or clear. You can also test a flag AND then set it, clear it, or simply flip its state. The application also supports versions of the flag test commands that use indirection. These command work with the values in the memory registers. You can also store and recall all the flags into/from text variables. The program stores the flags as strings of 1s and 0s. Examples for working with flags are:

1. SF 0
2. CF 1
3. FS? 0
4. FC? 4
5. FS?C 3
6. FS?S 2
7. FC?FLIP 0
8. FLIP 3

In addition to being able to test individual flags, the CPRCA application allows you to test a group of contiguous flags for having any combination of set and clear states. The command GFTEST? accepts an argument which is a string of 1s and 0s (such as 10011). The 0 stands for a clear flag state and 1 stands for set flag state. This argument defines a logical pattern used in comparing the states of the targeted flags. If the states match the logical equivalent of the tested pattern, the test succeeds —the application executes the next program line. The application also supports the commands GFTEST?C, GFTEST?S, and GFTEST?FLIP that will clear, set, and toggle, respectively, the flags involved in the test IF the test succeeds. The integer part of the X stack register provides the index of the first flag involved. The GFTEST?x commands serve to save you from resorting to using somewhat convoluted code (filled with LBLs and GTOs) to achieve the same purpose. My guess is that this new group flag test feature finds application in testing two or three flags.

Copyright © 2015, 2016 by Namir Clement Shammas

Here is an example of the above group flag test feature. Suppose flags 0, 1, 2, 3, and 4 have the states clear, set, set, set, and clear, respectively. The logical pattern is equivalent to the test string of 01110. The command GFTEST? 01110 (with a zero in the X stack register) succeeds since the string pattern 01110 matches the string equivalent of the logical states of the tested flags. By contrast, commands such as GFTEST? 01010 and GFTEST? 01001 fail because their patterns are not equivalent to 01110.

Named Individual Variables

The application dynamically creates named variables when you apply the VSTO. Using named individual variables enhanced the readability of your source code. This is even more true when read by others or when you read your code months or years later. The application also supports register arithmetic with the VSTO and VRCL commands.

Named Individual Registers That Store Text

The application allows you to save and recall the contents of the Alpha register. The XEQ *stringVarName* command allows you to evaluate an RPN expression stored in a named string variable. All by itself, the XEQ command evaluates an RPN expression stored in the Alpha register. This feature supports a dynamic user–defined function that you can enter and execute at runtime as a simple subroutine.

Array Variables

The CPRCA application supports numerical arrays that are accessed using names. The array variables bring with them much power and many features. The next subsections discuss the various aspects of array variables.

Basic Access

The application requires that your program create each individual array variable by specifying its name and number of elements. The RSTO and RRCL commands allow you to access elements of the array variables by specifying the array name, index, and stored value (in the case of the RSTO set of commands). Attempting to store a value beyond the current array size expands the array size to accommodate the stored value. The application supports commands that include register arithmetic with RSTO and the RRCL commands.

Creating Array variables

The application offers you these options to create new arrays:

1. The ARNEW command creates new array variables. All of the elements of the new array are set to zero.
2. The ARNEWRND command creates new array variables with uniformly–distributed random values.
3. The ARNEWINTRND command creates new array variables with uniformly–distributed random integers.
4. The ARNEWNORMRND command creates new array variables with normally–distributed random values.
5. The ARNEWSEQ command creates new array variables with a sequence of values.

Copyright © 2015, 2016 by Namir Clement Shammas

The CPRCA application supports commands to copy data between all or part of the memory registers and array variables. These commands make it easy to make copies (or backups, if you like) of the memory registers and then restore them.

You can create local array variables within subroutines and use these array variables as the repertoire of local values. You can use the ARVARRESET command to clear the memory of an array variable that store the values local to a subroutine. With the aid of flags, you can choose to create the array variable the first time a subroutine is called and then maintain the values in the supporting array variable between subroutine calls. This approach implements static local values for such subroutines.

Sorting and Searching

The CPRCA application supports commands that allow you to sort and search array variables. The sort command treats the array variable as a single row or column of values. As for searches, the application supports sequential forward, sequential backward, middle–first, forward circular, backward circular, forward heuristic and backward heuristic searching. The latter two searches allow you to move the matching element forward or backward, in support of optimistic and pessimistic search schemes. The search commands (except the heuristic ones) allow you to search for the first memory register that is equal to, less than, or greater than a searched value. The application also supports the command BINSEARCHA for binary searching in sorted array variables. This command returns the index of the matching array element AND the number of duplicate values that may be located both above and below the matching element!

The circular searching that I mentioned above supports forward and backward search versions. The forward circular search uses the following algorithm:

1. Start the search in the array at a selected index, call it *First*.
2. Search in the elements starting with index *First* and until the end of the array.
3. If step 2 finds a match, return the index of the matching array element, and end the search.
4. Search from the first array element and up to the element at index *First*–1.
5. If step 4 finds a match, return the index of the matching array element.
6. If step 4 fails to find a match, return –1.

The backward circular search uses the following algorithm:

1. Start the search in the array at a selected index, call it *First*.
2. Search in the elements starting with index *First* and until the beginning of the array.
3. If step 2 finds a match, return the index of the matching array element, and end the search.
4. Search from the last array element and down to the element at index *First*+1.

Copyright © 2015, 2016 by Namir Clement Shammas

5. If step 4 finds a match, return the index of the matching array element.
6. If step 4 fails to find a match, return –1.

In the pairs of sequential forward and backward searches, the application uses the same command for each pair. What determines the search direction is the *sign* of the index of the first searched element. If that index is zero or positive, the search command performs a forward search. By contrast, if the index is negative, the search command performs a backward search.

The Middle–First search is a good statistical search for an unsorted array. The search starts with the middle element and maintains two search indices. The command uses these indices to repeatedly alternate searching above and below the middle element. With each iteration, these two indices gradually point to elements that become further from the middle element and closer to the first and last array elements.

Efficient Sequential Access of Array Elements

The application supports a special feature to simplify the sequential access of values in an array variable. Each array variable has two indices that track the sequentially stored and recalled elements. First, you employ the command SIDX to initialize the storage index of an array variable. Likewise, you use the command RIDX to initialize the recall index of an array variable. With either or both indices set, you can use a loop iteration controlled by another variable to store or recall values in an array variable without the explicit need to use an index. All you need is the name of the array. The commands that support this feature are:

- The command RST+ stores the value of the X stack register in an array variable using the array storage index. This command then increments the array storage index by 1. The command is useful especially when you want to access the elements of an array variable starting with low indices and moving up.
- The command RST– stores the value of the X stack register in an array variable using the array storage index. This command then decrements the array storage index by 1. The command is useful especially when you want to access the elements of an array variable starting with high indices and moving down.
- The command RST$ stores the value of the X stack register in an array variable using the array storage index. This command does not alter the value of the array storage index.
- The command RRC+ recalls the value in an array variable using the array recall index. This command then increments the array recall index by 1. This command is useful especially when you want to access the elements of an array variable starting with low indices and moving up.
- The command RRC– recalls the value in an array variable using the array recall index. This command then decrements the array recall index by 1. This command is usefully

Copyright © 2015, 2016 by Namir Clement Shammas

especially when you want to access the elements of an array variable starting with high indices and moving down.

- The command RRC$ recalls the value in an array variable using the array recall index. This command does not alter the value of the array recall index.
- The command +RIDX allows you to increment or decrement the current value of the array variable recall index.
- The command +SIDX allows you to increment or decrement the current value of the array variable storage index.
- The command GETSDIX returns the value of the storage index of an array variable.
- The command GETRDIX returns the value of the recall index of an array variable.

Normally using the commands RST+ with RRC+ or the commands RST– with RRC– will do fine in sequentially accessing the elements of an array variable in a loop. You can combine using the RST+, RST–, and RST$ commands if you need to access the array elements in a neo–sequential method. By neo–sequential I mean that you are generally moving up or down the array elements but may want to revisit the same or a previous element, or may want to peek at the next element. The same comments apply for combining the use of the commands RRC+, RRC–, and RRC$. The program subsection titled *Efficient Access of Array Variable Elements* has a program that shows you how to implement this feature using the commands SIDX, RIDX, RST+, and RRC+.

Array Mathematics

The application supports the commands VCA+, VCA–, VCA*, and VCA/ to add, subtract, multiply, and divide, respectively, a range of the individual elements of two array variables. These commands take two array variables and modify the elements of the first array using the values of the second array. The stack provides information that determines the maximum number of elements to process and the first indices of the arrays involved. These indices give you the ability to work with ranges of indices that are different in each of the paired arrays. The CPRCA application has a scalar/array version of the above commands. They allow you to add, subtract, multiply, and divide array elements and scalar values. The commands VCAS+ and VCAS* add and multiply the elements of an array with a scalar value. Since subtraction and division are not communicative, the application offers two versions for these math operations. The VCAS– and VCAS/ subtract by and divide by scalar values. The following equations show the math performed by these commands:

$$ArrayVar(i) = ArrayVar(i) - X$$
$$ArrayVar(i) = \frac{ArrayVar(i)}{X}$$

Conversely, the commands VCASA– and VCASA/ subtract by and divide by the array elements. The following equations show the math performed by these commands:

$$ArrayVar(i) = X - ArrayVar(i)$$

Copyright © 2015, 2016 by Namir Clement Shammas

$$ArrayVar(i) = \frac{X}{ArrayVar(i)}$$

In addition to the above VCAxxx commands, the application offers the VCAFX command that allows you to alter a range of elements in an array variable using an RPN expression. The Alpha register contains the RPN expression and the stack registers hold the information related to the location and range of array elements to alter. This command offers much power and flexibility in manipulating the values in an array variable.

Local Memory Registers for Subroutines

The application supports local numerically–indexed memory registers inside subroutines. The LSTO *n* command stores the value of the X stack register in the local memory register at index n. Likewise, the LRCL *n* command recalls the value in the local memory register at index n. The application supports register arithmetic for the commands LSTO and LRCL. How does the local memory register system work? The application creates an initial number of special array variables to simulate local registers. Each array variable has an integer–name and has an initial number of registers. The application can expand both the number of array variables and/or the memory registers held by each array. As you call each subroutine, the application increments the subroutine–call counter. The CPRCA uses that counter to access the appropriate array variable, using the counter's value as the unique name of the array. The code resets the value in the array variable when a subroutine is called. You use the commands LSTO and LRCL with only the index of the local memory register. The application knows which integer–named array variable stores the targeted register, with the help of the subroutine–call counter.

What happens if you use the commands LSTO and LRCL in a main routine (before calling any subroutine or after all the subroutine calls have terminated)? The answer may surprise you! The main routine can also access its own local memory registers that are separate from the large pool of 10,000 memory registers! The programs localregs1.txt and localregs2.txt that I present in the programming examples section illustrate the local memory register features in general and also how the main routine can access its own local memory registers!

Another question you may have concerns recursion. The application supports subroutines that call itself and allows each call to maintains its own version of local memory registers. This feature works using the subroutine–call counter as a unique id for each subroutine's local memory registers. Please use recursions reasonably, since excessive recursive calls will drain memory resources and may cause a program crash.

For this system of local memory registers to work properly you must be disciplined in calling subroutines using the GSB command and exiting subroutines using the RTN command. Terminating subroutine execution using a GTO command will confuse the

Copyright © 2015, 2016 by Namir Clement Shammas

local memory register system and create chaos, because the system requires a correct update for the subroutine–call counter. The local memory register system works correctly as long as the subroutine–call counter is not out of synch or corrupted by *cute programming tricks*.

Statistical Regression Calculations

Using the Statistical Registers

The CPRCA application supports two separate sets of commands and approaches to perform linear regression. The first feature is similar to HP calculators that maintain statistical registers and perform statistical calculations. These calculations include evaluating the mean, standard deviation, regression coefficient of determination, regression slope, and regression intercept. You can use the S+ and S– commands to add or remove pairs of (X, Y) observations, respectively. Using these commands is basically slow and is meant for relatively small sets of observations. You can use the DATAS command to accelerate adding statistical data. The application allows you to recall the individual statistical registers using the commands SUMN, SUMX, SUMX2, SUMY, SUMY2, and SUMXY.

You can also use the command DATAM to read data into the memory registers and then use the S++ command to add the pairs of (X, Y) in that block of memory registers to the statistical registers. If you wish to transform the data just before adding them to the statistical registers, then use the S+T command. This command places the (X, Y) values in the X and Y stack registers and allows you to transform these values using an RPN expression in the Alpha register. This expression can use functions like LN, SQRT, 1/X, as well as the command X<>Y to switch values back and forth between the X and Y stack registers. Once the transformation is accomplished, the S+T commands accesses the values in the X and Y registers and adds them to the statistical registers.

The application also supports calculating the confidence intervals for the means, standard deviations, regression slope, intercept, coefficient of determination, and the projections of X onto Y. The commands for these calculations start with the prefix CI.

Using the READDATA Commands

In the case that you have a relatively large set of observations for two or three variables, you can use the READDATA2 and READDATA3 commands. These commands read values from comma–delimited files into the memory registers. The application designates the memory registers that hold these observations as special blocks for statistical calculations. The application supports a variety of linearized regressions between two and three variables. Among these commands are BESTLR and BESTMLR which perform the best linearized and best multiple regression between two and three variables, respectively. Thus, you can reuse the values in these special memory register blocks for different linearized models.

Copyright © 2015, 2016 by Namir Clement Shammas

Probability and Cumulative Distribution Functions

The CPRCA application supports the following functions related to common statistical distributions. When I coded these functions, I checked their results against those given by the HP–Prime calculator. The functions are by group:

1. The commands Q_PDF, T_PDF, CHI_PDF, and F_PDF calculate the probability distribution functions for the Normal, Student–t, Chi–Squared, and Fisher–Snedecor F functions, respectively.
2. The commands Q_CDF, T_CDF, CHI_CDF, and F_CDF calculate the cumulative distribution functions (CDF) for the Normal, Student–t, Chi–Square, and Fisher–Snedocor F distributions, respectively.
3. The commands Q_ICDF, T_ICDF, CHI_ICDF, and F_ICDF calculate the inverse CDF functions for the Normal, Student–t, Chi–Square, and Fisher–Snedocor F distributions, respectively. These functions help you in carrying out various statistical testing and calculations for confidence intervals.

File I/O Support

Since file I/O is much easier on computer applications than with the original HP–41C, it would be a missed opportunity not to use file I/O with the console application. This is true especially when the application supports a generous number of memory registers that can store different groups of data. The application supports three types of file I/O:

1. Reading and writing memory registers. You can read and write all or part of the memory registers. The application ignores the attempt of reading values meant for indices higher than 9999. It displays a warning message to that effect and proceeds with program execution.
2. Reading data blocks of two and three variables from comma–delimited files. The application stores these blocks of data in the memory registers. It uses them to perform various types of linear and multiple regression. It is very important to point out that the related regression calculations do not use the block of statistical registers that I mentioned earlier. Instead, they use internal variables to track the various statistical summations. The application ignores the attempt of reading values meant for indices higher than 9999. It displays a warning message to that effect and proceeds with program execution.
3. Reading and writing the values in array variables. Since these arrays have flexible sizes, the application will read all the values from the source file. When using this kind of file I/O, you read/write data from/to different dynamic variables. The only limitation here, in the case of multiple variables, is that they must have the same number of elements with file output commands. Reading data that expands the size of a dynamic array requires additional execution time to resize the dynamic array.

Copyright © 2015, 2016 by Namir Clement Shammas

Within the regular memory registers, you can load special blocks containing sets of 2 or 3 variables. The READDATA2 and READDATA3 commands allow you to read relatively large data sets from comma–delimited text files. The application uses these special blocks in linear regression between 2 or 3 variables. They reside in memory registers, starting at an index, call it IDX, of your choosing. Table 1 shows the format for storing the special data blocks used for regression calculations.

Memory Register	*Content*
Mem(IDX)	A counter for the number of variables. This is either 2 or 3.
Mem(IDX+1)	The number of data points that are stored in subsequent memory registers.
Mem(IDX+2)	The first observation of the first row, call it X(1).
Mem(IDX+3)	The second observation of the first row, call it Y(1).
Mem(IDX+4)	Other observations on the same row, call it Z(1) in the case of reading data for three variables, or on the following row, call it X(2) in the case of reading data for two variables.
Mem(IDX+5) to Mem(IDX+…)	More data.

Table 1. The format for storing special data blocks used for regression calculations.

As for array variables, you can load data from a comma–delimited text file into several existing array variables. The READ1VAR, READ2VARS, and other similar commands, allow you to get data from one source and distribute the values in separate array variables. Using the WRITE1VAR, WRITE2VARS, and other similar commands, allow you to collect data from different named variables (with the same number of values) and store them in a single comma–delimited text file.

Using file I/O allows you to type in data in text files or save Excel data into comma–delimited text files. Once the data is stored in text files, you can use the appropriate commands to read them while maintaining relatively short source code.

Program Flow Control

The application supports using labels to direct subroutines and program flow control. The labels are all stored as text—even those with only digits. You can use leading zeros in declaring labels, just like with the HP–41C. In fact, the application allows you to use multiple leading zeros, if your heart so desires! The application internally removes leading zeros in storing labels and in searching for them. You can reference a label having a leading zero with or without that zero. For example, the commands GTO 01 and GTO 1 direct program flow control to LBL 1 OR to LBL 01. Keep in mind that the application regards leading zeros as superfluous. Consequently, declaring labels like LBL 01 and LBL 1 in the same program generate an error since the application regards these two labels as duplicates. The error message that informs you about the duplicate numeric labels will refer to that label without the leading zero.

Copyright © 2015, 2016 by Namir Clement Shammas

The CPRCA offers the following program flow control constructs:

- The GTO and GTOIND commands that jump to a label. While the GTO can jump to alphanumeric and numeric labels, the GTOIND can only jump to numeric labels. This limitation gives you greater flexibility in jumping to any numerical label.
- The GSB and GSBIND that execute a subroutine at designated label. While the GSB can execute subroutines with alphanumeric and numeric labels, the GSBIND can only execute subroutines with numeric labels. This limitation gives you greater flexibility in jumping to any numerical label.
- The JUMP, JUMPIND, JUMPX, JUMPY, JUMPZ, and JUMPT commands support jumping to program lines by specifying the number of lines to jump. If that number is positive, the program jumps forward. By contrast, if that number is negative, the program jumps backward.
- The DSE41 and ISG41 commands control looping, HP–41C style.
- The DSE and ISG commands that offer a wider range of looping.
- The DSL command that controls looping. I added this command to be able to process memory registers starting at a positive index and moving down to and including 0.
- The DSL41 command that supports and HP–41C version of the DSL command.
- The FOR BASIC-like commands that allows you to perform powerful loop iterations.

Extended Logical Testing

The application supports a complete set of commands to compare the values in the X and Y stack registers as well as comparing the values of the X stack register with zero. In addition, the application offers the following sets of testing commands:

- Testing commands that compare the values in the X and Y stack registers AND specify the number of program lines to skip if the test fails. These commands extend the standard tests, such as X>Y? And X=Y? by appending a single digit between 2 and 9 right after the question mark. This digit specifies the number of program lines to skip if that test fails. For example, the command X=Y?4 causes the runtime system to skip the next four program lines if the values of the X and Y stack registers are not equal. By appending the number of program lines to skip, if the test fails, you are able to maintain good readability of the code. This approach is more readable than using a scheme where you use a special and separate command to specify, ahead of time, the number of program lines to skip if the next test fails.
- Testing commands to compare the values in the X stack register with those in a memory register. For example X?=5 tests if the value in the X stack register equals that in memory register 5. The other commands have the general syntax X?LogicalTestMemoryRegisterIndex. Check Table 3 for the complete members of this set of commands.

Copyright © 2015, 2016 by Namir Clement Shammas

- Testing commands to compare the values in the Y stack register with those in a memory register. For example Y?<>5 tests if the value in the Y stack register does not equal the value in memory register 5. The other commands have the general syntax Y?LogicalTestMemoryRegisterIndex. Check Table 3 for the complete members of this set of commands.
- Testing commands to compare the values in the Z stack register with those in a memory register. For example Z?<>5 tests if the value in the Z stack register does not equal the value in memory register 5. The other commands have the general syntax Z?LogicalTestMemorZRegisterIndex. Check Table 3 for the complete members of this set of commands.
- Testing commands to compare the values in the T stack register with those in a memory register. For example T?<>5 tests if the value in the T stack register does not equal the value in memory register 5. The other commands have the general syntax T?LogicalTestMemorTRegisterIndex. Check Table 3 for the complete members of this set of commands.
- The IS_BETWEEN command for testing if the value in the X stack register is at or within a range defined by the values in the Y and Z stack registers.
- The IS_OUTSIDE command for testing if the value in the X stack register is at or outside a range defined by the values in the Y and Z stack registers.
- The IS_WITHIN command for testing if the value in the X stack register is *strictly inside* a range defined by the values in the Y and Z stack registers.
- The IS_WITHOUT command for testing if the value in the X stack register is *strictly outside* a range defined by the values in the Y and Z stack registers.

Solve and Integrate

The CPRCA application supports two versions for both the SOLVE and INTEGrate commands. The first version allows you to use an RPN expression in the Alpha register as the function to solve or integrate. Such expressions must be void of labels, GTOs, GSBs, and all other program flow control constructs. For example, you can store the string “x^2 1 –“ in the Alpha register to define the simple function $f(x)=x^2–1$ to be solve or integrated.

The second version allow you to use a function defined by a program label (and use other labels, GTOs, GSBs, and all other program flow control constructs). Simply place in the Alpha register the keyword GSB, followed by a space, followed by the label that implements the target function. For example, placing the string “gsb myfx” in the Alpha register tells the commands SOLVE and INTEG to call the code after label myfx to obtain values of the targeted function.

The CPRCA offers three flavors of SOLVE:

1. The SOLVE command that uses Newton’s method.

Copyright © 2015, 2016 by Namir Clement Shammas

2. The SOLVHAL command that uses Halley's method, which is in general converges faster than Newton's method.
3. The SOLVEBIN method that uses the Bisection method. This method is slow to converge but is guaranteed to work if the limits A, and B of the rook bracketing range [A, B] have functions values with opposite signs.

The application offers several commands to perform Gaussian quadrature. The GAUSSLEGQUAD, GAUSSCHEBQUAD, GAUSSKRONQUAD, GAUSSKRONQUAD2, GAUSSLAGQUAD, GAUSSHERQUAD, and GAUSSCHEBQUAD perform the Gauss-Legendre quadrature, Gauss-Laguerre quadrature, Gauss-Kronrod quadrature, Gauss-Hermite quadrature, and Gauss-Chebyshev quadrature, respectively. The Gaussian-Legendre quadrature performs a numerical integration for the following integral:

$\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w(i)f\big(x(i)\big)$

Where *w(i)* is a weight associated with each *x(i)* root of the Legendre function. The above integral can be mapped to the range [a, b] using the following equations:

$\int_{a}^{b} f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{(b-a)x}{2} + \frac{a+b}{2})dx$

$\int_{a}^{b} f(x)dx = \frac{b-a}{2}\sum_{i=1}^{n} w(i)f\left(\frac{(b-a)x(i)}{2} + \frac{a+b}{2}\right)$

The Gauss-Kronrod quadrature is an enhanced version of the Gaussian-Legendre quadrature and uses about twice as many weights and nodes. Therefore, it is more accurate than the Gaussian-Legendre quadrature.

The Gauss-Chebyshev quadrature performs a numerical integration for the following integral:

$\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w(i)f\big(x(i)\big)$

Where *w(i)* is a weight associated with each *x(i)* root of the Chebyshev function. The value of each w(i) is equal to π/n. The value of x(i) is calculated using:

$$x(i) = \cos\left(\frac{2i-1}{2n} * \pi\right)$$

The integral is calculated using:

$\int_{a}^{b} f(x)dx = (\frac{b-a}{2})(\frac{\pi}{n})\sum_{i=1}^{n} f\left(\frac{(b-a)x(i)}{2} + \frac{a+b}{2}\right) * \sqrt{(1 - x(i) * x(i))}$

It is worth pointing out that the Gauss-Legendre quadrature is more accurate than the Gauss-Chebyshev quadrature when both methods use the same order of their respective orthogonal polynomials. Good results are obtained with the Gauss-Chebyshev quadrature when using high

Copyright © 2015, 2016 by Namir Clement Shammas

polynomial orders (something like in order of 100). This quadrature method is very easy to implement in (vintage) programmable calculators and vintage BASIC pocket computers. The Gauss-Kronrod quadrature is more accurate than the Gauss-Legendre quadrature.

The Gauss-Laguerre quadrature performs a numerical integration for the following integral:

$$\int_0^{\infty} e^{-x} f(x)dx = \sum_{i=1}^{n} w(i)f(x(i))$$

Where *w(i)* is a weight associated with each *x(i)* root of the Laguerre function. The Gauss-Hermite quadrature performs a numerical integration for the following integral:

$$\int_{-\infty}^{\infty} e^{-x\text{^}2} f(x)dx = \sum_{i=1}^{n} w(i)f(x(i))$$

Where *w(i)* is a weight associated with each *x(i)* root of the Hermite function. The GAUSSxxxQUAD commands permit you to specify the order of the associated polynomial used. The higher the order, the more accurate the calculated integral. I use a special algorithm that I devised to calculate the roots of the Legendre, Laguerre, and Hermite polynomials for a polynomial order that your programs specify. This approach is better than using one, two, or a few different quadrature routines for fixed orders of these orthogonal polynomials.

The Gauss-Laguerre quadrature, and Gauss-Hermite quadrature are able to calculate areas not (easily) possible with the other numerical integration methods supported by CPRCA. Remember to include the exponential term (as shown in the above equations) with your own function f(x) when you supply the commands GAUSSLAGQUAD and GAUSSHERQUAD with the expression to integrate.

The routines that support the GAUSSKRONQUAD are implemented using several helper functions and a backup function. The command uses a main function that dynamically calculates the nodes and weights needed for the quadrature. Should this routine experience a logical or runtime error, the command invokes a backup version that supplies the weights and abscissa for 99 points. The code then examines the range for integration and determines if dividing that range into 1, 10 or 100 smaller parts is necessary to maintain a good accuracy. This programming scheme guarantees that the command GAUSSKRONQUAD yield a good answer.

The Application also offers the command GAUSSKRONQUAD2 that works just like GAUSSKRONQUAD. The two commands differ slightly in their internal algorithms. The difference lies in the way the weights and abscissa are calculated for different integration ranges. Unlike the command GAUSSKRONQUAD, the command GAUSSKRONQUAD2 does not divide the integration range into smaller ranges to perform multiple quadratures.

Copyright © 2015, 2016 by Namir Clement Shammas

Scanning a function for Roots, Minima, and Maxima

The application offers the command SCAN to scan the values of a function in a given range. The command locates and reports the roots, minima, and maxima in that range. The command SCAN actually needs the following six input values:

1. The index of the memory block that will store the results. This index, call it K, will store the number of results obtained by the command SCAN. Following index K, is 3*K memory registers that store one or more data triplets. The first memory register at triplet n (stored in index *K + 3n-2*) stores the root/minimum/maximum. The second memory register in triplet n (stored in index *K + 3n-1*) stores the value of the function at the root/minimum/maximum. The third memory register in the triplet n (stored in index *K + 3n*) is an integer code that indicates the type of the locus—0 for a root, 1 for a maximum, and -1 for a minimum.
2. The function tolerance value.
3. The tolerance for the root/minimum/maximum values.
4. The search step value.
5. The starting point for the range of values scanned.
6. The ending point for the range of values scanned.

The trick used to pass the above six parameters is to enter the first two in the stack, use the command COPYSTACK to copy these values in the backup stack, and then enter the last four values in the stack. Once the above parameters are in place you can execute the SCAN command. This command will display the results for the root/minimum/maximum values and their function values. The command also stores these results in the memory registers indicated by the first parameter.

Best Linear and Multiple Regression Models

The CPRCA application supports the BESTLR and BESTMLR commands that use data blocks read using the READDATA2 and READDATA3 commands, respectively. The BESTLR command searches for the best model describing the relation between variables X and Y. This is possible, because the application retains the original observations in the memory registers. This scheme allows the BESTLR command to reprocess, as many times as needed, the original data with different transformations. The command applies different powers, to each variable, that range between –4 and 4, in increments of 0.5. The command interprets the power of 0 as a special request to calculate the natural logarithm. The command returns, in the stack, the coefficient of determination, the intercept, and the slope, for the best curve fit. The command places a string describing the general equation for the best linearized regression model in the Alpha register. To get the values for the powers associated with the best model use the command BESTLRPWRS. This command places the best powers for the variables Y and X in the Y and X stack registers, respectively. What about the runner up regression models and all the other models? Declaring the best model as the absolute winner and discarding the regression

Copyright © 2015, 2016 by Namir Clement Shammas

statistics for the other models is, in most cases, not a wise move. This is true, because random error/noise in the observations may artificially favor a certain model over other ones. The command writes the regression results, for each successfully calculated model, to the comma-delimited file BESTLR_*date_stamp*.CSV. The application uses the current date stamp as part of the filename. This approach creates a unique filename and thus avoid writing over files generated by previous sessions. You can open the .CSV files with Excel. The first row contains the header for the columns. You can easily sort the data using the values in the first column (the coefficient of determination) as the key values for sorting the data in a descending order. Each row contains the following values:

- The coefficient of determination.
- The transformation power for variable Y.
- The transformation power for variable X.
- The intercept.
- The slope.

The BESTMLR command searches for the best model describing the relation between variables Z, X, and Y. The command applies different powers, to each variable, that range between –4 and 4, in increments of 0.5. The command interprets the power of 0 as a special request to calculate the natural logarithm. The command returns in the stack the coefficient of determination, the intercept, the slope for Y, and the slope for X, for the best curve fit. The command places a string describing the general equation for the best regression model in the Alpha register. To get the values for the powers associated with the best model use the command BESTMLRPWRS. This command places the best powers for the variables Z, Y, and X in the Z, Y, and X stack registers, respectively. What about the runner up regression models and all the other models? Declaring the best model as the absolute winner and discarding the regression statistics for the other models is, in most cases, not a wise move. This is true, because random error/noise in the observations may artificially favor a certain model over other ones. The command writes the regression results, for each successfully calculated model, to the comma-delimited file BESTMLR_*date_stamp*.CSV. The application uses the current date stamp as part of the filename. This approach creates a unique filename and thus avoid writing over files generated by previous sessions. You can open the .CSV files with Excel. The first row contains the header for the columns. You can easily sort the data using the values in the first column (the coefficient of determination) as the key values for sorting the data in a descending order. Each row contains the following values:

- The coefficient of determination.
- The transformation power for variable Z.
- The transformation power for variable Y.
- The transformation power for variable X.
- The intercept.

Copyright © 2015, 2016 by Namir Clement Shammas

- The slope for variable Y.
- The slope for variable X.

Armed with the above tables of data (for the best 2 or 3 variables), you can then study and compare the best models. Keep in mind that arbitrary errors in the observations may favor some models that do not represent the *true* relationship between the variables. If you process several sets of data (or have enough observations to organize them into several groups), then you should (hopefully) see a single model that repeatedly takes its place among the elite models. That would be the model you seek.

Writing the regression models to a .CSV file simplifies things for the CPRCA application. It shifts the task for sorting the results to Excel (or other advanced text editors). Excel can sort through hundreds and thousands of rows almost effortlessly. Thus the CPRCA application is absolved from storing the regression statistics for all of the hundreds and thousands of models and then sorting them. Even if we adopt a scheme of writing the regression statistics of, say the top 50 best models, it will still require the application to do a lot of bookkeeping.

The application offers the commands NORMDATA2 and NORMDATA3 that allow you to normalize the data blocks, read with commands READDATA2 and READATS3, using the following equations:

$X_i = 1 + (Xi - X_{min}) / (X_{max} - X_{min})$
$Y_i = 1 + (Y_i - Y_{min}) / (Y_{max} - Y_{min})$
$Z_i = 1 + (Z_i - Z_{min}) / (Z_{max} - Z_{min})$

The above transformations place the results in the range of [1, 2] and prepare the data for *empirical* curve fitting. This range is suitable for all of the transformation available in the commands BESTLR and BESTMLR. The normalizing commands also provide the option to transform any or all of the X, Y, and Z variables into their logarithm values as long as all of the values for the transformed variable are positive. Such transformation can be helpful when the values for a variable changes several orders of magnitudes. My advice is to use the commands BESTLR or BESTMLR on the raw data first, then apply command NORMDATA2 or NORMDATA3, and then reapply commands BESTLR or BESTMLR on the normalized data. You end up with two sets of best curve fits, giving you more options for better empirical fits.

The DATA Statement is Back!

I enjoyed using the DATA statement in legacy BASIC programs. This statement offered a convenient way to read constants into arrays and single variables. Even FORTRAN had a similar construct! I was sad to see the demise of the DATA statement when line numbers were laid to rest. I decided to bring back the DATA statement, albeit with a different twist, in the CPRCA application. It supports three types of DATA statements allowing you to concentrate entering

Copyright © 2015, 2016 by Namir Clement Shammas

data using fewer lines. The family of DATA commands all include lists of comma–delimited data. The commands are:

1. The DATAS command allows you to add pairs of (X, Y) values to the statistical registers. The S after DATA stands for statistics. You can use multiple DATAS commands to accumulate statistical data in multiple batches. You need not tell the DATAS how many pairs of (X, Y) are present. The command does the counting for itself. If there are an odd number of values in the list, the command ignores the last value.
2. The DATAM command permits you to store data in the memory registers. The M after DATA stands for memory registers. The first value in the list of data is the index of the first memory register that receives the data. The command ignores values that violate the indexing limit of the memory registers.
3. The DATAR command empowers you to store values in an array variable. The R after DATA stands for array variables. The first value in the list of data is the name of the array that stores the data. This name is optional. When you omit it from the list, the command uses the contents of the Alpha register to specify the array variable. The second value in the list is the index where the insertion in the array variable begins. The remaining list values are the data to be inserted. You can use multiple DATAR commands to store data, in batches, in an array variable.

The general syntax for using the DATAS command is:
DATAS X(1), Y(1), X(2), Y(2),…, X(n), Y(n)

Here is a simple example of using the DATAS command:
DATAS 10,50,25,77,30,86,35,95,100,212

The general syntax for using the DATAM command is:
DATAM index, value1, value2, value3, …, value_n

It is possible to use the DATAM command to emulate the READDATA2 and READDATA3 commands. This approach allows you to hard code the data inside a program. The list for the DATAM command involves the following information:

1. The index of the first memory registers that store the rest of the data.
2. The value of 2 or 3 when emulating commands READDATA2 or READDATA3, respectively.
3. The number of data sets (for 2 or 3 variables).
4. The data sets for 2 or 3 variables.

Here is a simple example of using the DATAM command:
DATAM 11,1,2,3,4,5,6

The above command stores the values 1 through 6 in memory register 11 through 16.

Copyright © 2015, 2016 by Namir Clement Shammas

Here is a simple example of using the DATAR command:
DATAR Xarr,0,1,2,3,4,5,6

The above example makes the command store, in the array variable Xarr, the values from 1 to 6, starting at index 0. Here is another example for the DATAR command is:
'Xarr
DATAR 0,1,2,3,4,5,6

The above example first places the identifier for the array variable in the Alpha register then uses the DATAR command to insert the values from 1 to 6, starting at index 0 of the array variable Xarr.

Comments in Source Code

You can insert comment lines in your source code. These comments MUST EACH OCCUPY an entire line. You can use REM or any of the characters !, @, #, $, and % as a comment line designators. These designators are located the start of a comment line. It is a good idea to use the same comment line designator, even though you can mix at will between the various designators.

Use comments to document various steps in your program, the program version, update history, copyright notice, and so on. Here are examples of comment lines:

```
REM This is a comment line
! This is also a comment line
# and so is this line
$ Same here!
% and here too!
```

Swapping Between Any Two Stack Registers

The CPRCA application supports swapping between any two stack registers. In addition to the traditional X<>Y command that swaps the values in the X and Y registers, the application allows you to swap between the following stack registers:

- The X and Z registers, using the X<>Z command.
- The X and T registers, using the X<>T command.
- The Y and Z registers, using the Y<>Z command.
- The Y and T registers, using the Y<>T command.
- The Z and T registers, using the Z<>T command.

The general format for the *reg1*<>*reg2* command lists the lower stack register first, followed by the characters <>, followed by the higher stack register. Using these stack register swap commands saves you from having to roll the stack registers up or down to access the Z and T stack registers.

Copyright © 2015, 2016 by Namir Clement Shammas

The application also supports commands that swap between any stack register and any memory registers. The general syntax for these commands is *stack_register*<>*nnn*, where *stack_register* is either X, Y, Z, or T. The entity *nnn* is the numerical index of the targeted memory register. For example, the command Z<>10 swaps the value in the Z stack register and memory register with index 10.

It's About Time!

The application offers commands that return the current system date, time, and their individual components. Consult Table 2 and look for commands GETDATE, GETTIME, GETYEAR, GETMONTH, GETDAY, GETHR, GETMIN, and GETSEC.

The application also supports storing date/time information, located in the Alpha register, in text variables using the command ASTO. The format used for this storage is:

```
#mm/dd/yyy hh:mm:ss#
```

The leading and trailing # characters are important in telling the runtime system that the text represents date/time information. Moreover, Visual Basic, the language used to create the CPRCA application, also uses the above syntax to store date/time information in strings.

You can store floating-point representations of the date, yyyy.mmdd, and time, hh.mmss, that you place in the stack to text variables. The command DSTO takes the floating-point values for the date and time in the Y and X stack registers, respectively, and store them in the text variable specified with the DSTO command. The command DRCL reverses this operation. It takes the name of a text variable that contains date/time information and calculates the floating-point representation of the date and time. It then places the floating-point representation of the date and time in the Y and X stack registers, respectively. You can view the date and time stored in a text variable using the command VIEWDT or the command DVIEW. The argument for each one of these commands is the name of a text variable.

The CPRCA application also supports adding and subtracting date/time units (i.e. components). The ADDDT allows you to add either years, months, days, hours, minutes, or seconds to a date/time stored in a text variable. The Alpha register contains the name of the date/time unit you wish to add. You can only add one unit of date/time at a time. The command updates the date/time in the targeted text variable. The application also offers the command DIFFDT that subtracts date/time units from two dates stored in text variables. The Alpha register contains the same type of information, regarding the selected date/time unit, that I mentioned earlier. The DIFFDT command returns in the stack the difference in the date/time units, as selected by the Alpha register. The DIFFDT does not alter the date/time information in its text variables. Consult the reference for commands ADDDT and DIFFDT, in Table 3, to learn about the date/time units used to add or subtract date/time values. To subtract between dates only, use

Copyright © 2015, 2016 by Namir Clement Shammas

the same time-of-day in both date/time values subtracted. Conversely, to subtract time only, use the same date in both date/time values subtracted.

Copyright © 2015, 2016 by Namir Clement Shammas

Basic Functions

Table 2 shows the functions that work in both RPN calculations and programming modes. Table 3 shows the commands used for programming and include some advanced functions. I chose the headings of these two tables to have a different color to make it easy to distinguish between them when scrolling back and forth though the document.

The list of commands in tables 2 and 3 is extensive. Please take the time to familiarize yourself with the commands. You will notice the following patterns with commands that require arguments. You can either include the argument(s) with the command or omit them and rely on the Alpha register to provide the argument(s). Using the Alpha register provides you with more runtime flexibility since you are not stuck with names hard coded with the commands.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
+, –, *, /, ^	Perform a basic math operation. The ^ operator is a shorthand for the command Y^X.	2 5 / Returns 0.4
–>HMS	Convert a real value to H.MS (hours, minutes, and seconds). The result has the format hh.mmss.	12.5 –>HMS Returns 12.3
1/X	Calculate the reciprocal value.	10 1/x Replaces 10 in the X stack register with 0.1
ABS	Replace the value of the X register with its absolute value.	–10 abs Replaces –10 in the X stack register with 10
ASIN, ACOS, ATAN	Calculate the common inverse trigonometric functions. The results are angles in radians.	0.5 asin Returns 0.5235987755
ASIND, ACOSD, ATAND	Calculate the common inverse trigonometric functions. The results are angles in degrees.	2 sqrt 1/x asind Returns 45.
ASINH, ACOSH, ATANH	Calculate the common inverse hyperbolic functions.	2 asinh Returns 1.4436354751
BESSELJ0, BESSELJ1, BESSELJN, BESSELY0, BESSELY1, BESSELYN.	Calculate the Bessel functions. All functions use the value of the X register to supply the value for x in the function call. The commands BESSELJN and BESSELYN use the integral value in the Y register to supply the value of the function's order n.	4 1.2 BesselJn Returns 0.0050226662
BETA	Calculate the Beta function. The function uses the values in the X and Y registers as the first and second arguments, respectively.	4 5 beta Returns Beta(5,4) as 0.00357142857142857
CHEBYSHEV, HERMITE, LEGENDRE, LAGUERRE	Calculate the values for the named polynomial $P_n(x)$. The Y register supplies the value for the polynomial order. The X register supplies the value for the argument x.	5 2.5 Legendre Calculates the 5^{th} order Legendre function at x=2.5. The result is 637.0117187.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
CHI_CDF and CHI_ICDF	Calculate the Chi–Squared cumulative distribution function and its inverse. The stack provides the following input: T: Z: Y: The degrees of freedom. X: The argument X. In the case of the command for the inverse pdf, the value of the X stack register is the probability expressed as a fraction.	5 6 chi_cdf Returns 0.6937810815 5 0.7 chi_icdf Returns 6.05720105
CHI_PDF	Calculate the Chi–Squared probability distribution function. The stack provides the following input: T: Z: Y: The degrees of freedom. X: The argument X.	5 1 chi_pdf Returns 0.0806569081
CHS	Change the sign of the value in the X stack register.	1 chs Makes the X stack register store the value –1.
CI	Calculate the cosine integral function. The X stack register provides the argument for this command.	2.3 ci Returns 0.3471756175

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
CIINTERCEPT	Calculate the upper and lower limits of the confidence interval for the regression intercept. The value in the X stack register provides the confidence level as a fraction (usually 0.95). The command returns the following values in the stack: T: Z: The upper limit of the confidence interval for the intercept. Y: The intercept. X: The lower limit of the confidence interval for the intercept. This command assumes that you have used the command LRXY recently. If not, it returns the values of 1E+99 in the Z, Y, and X stack registers.	0.95 ciintercept Returns, for example, the following results: Z: 17.7532966286244 Y: 10.8726619398323 X: 3.04670337137564
CIMEANX, CIMEANY	Calculate the confidence intervals for the mean values of variables X and Y, respectively. You must have used the S+ command at least twice (actually more to get reasonable results) for the commands to display values other than 1E+99. The value in the X stack register provides the confidence level as a fraction (usually 0.95). The command returns the following values in the stack: T: Z: The upper limit of the confidence interval for the mean of X or Y. Y: The mean value for X or Y. X: The lower limit of the confidence interval for the mean of X or Y. This command assumes that you have used the command S+ recently. If not, it returns the values of 1E+99 in the Z, Y, and X stack registers.	0.95 cimeanx Returns, for example, the following values: T: Z: 0.75 Y: 0.5 Z: 0.25

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
CIR	Calculate the upper and lower limits of the confidence interval for the correlation coefficient, *r* (calculated as sign(slope)*$\sqrt{R^2}$). This command uses the inverse normal CDF function to calculate the confidence interval. The value in the X stack register provides the confidence level as a fraction (usually 0.95). The command returns the following values in the stack: T: Z: The upper limit of the confidence interval for *r*. Y: The value of *r*. X: The lower limit of the confidence interval for *r*. This command assumes that you have used the command LRXY recently. If not, it returns the values of 1E+99 in the Z, Y, and X stack registers.	0.95 cir Returns, for example, the following results: Z: 0.995392185251632 Y: 0.955214936062091 X: 0.629724598174932

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
CIR2, CIRSQR	Calculate the upper and lower limits of the confidence interval for the coefficient of determination, R^2. This command uses the inverse Student–t CDF function to calculate the confidence interval. The value in the X stack register provides the confidence level as a fraction (usually 0.95). The command returns the following values in the stack: T: Z: The upper limit of the confidence interval for R^2. Y: The value of R^2. X: The lower limit of the confidence interval for R^2. This command assumes that you have used the command LRXY recently. If not, it returns the values of 1E+99 in the Z, Y, and X stack registers.	0.95 cir2 Returns, for example, the following results: Z: 0.970671584877132 Y: 0.912435574076105 X: 0.854199563275077
CISDEVX, CISDEVY	Calculate the confidence intervals for the standard deviation values of variables X and Y, respectively. You must have used the S+ command at least twice (actually more to get reasonable results) for the commands to display values other than 1E+99. The value in the X stack register provides the confidence level as a fraction (usually 0.95). The command returns the following values in the stack: T: Z: The upper limit of the confidence interval for the standard deviation for X or Y. Y: The standard deviation for X or Y. X: The lower limit of the confidence interval for the standard deviation for X or Y.	0.95 cisdevx Returns, for example, the following values: Z: 0.75 Y: 0.5 Z: 0.25

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
CISLOPE	Calculate the upper and lower limits of the confidence interval for the regression slope. The value in the X stack register provides the confidence level as a fraction (usually 0.95). The command returns the following values in the stack: T: Z: The upper limit of the confidence interval for the slope. Y: The slope. X: The lower limit of the confidence interval for the slope. This command assumes that you have used the command LRXY recently. If not, it returns the values of 1E+99 in the Z, Y, and X stack registers.	0.95 cislope Returns, for example, the following results: Z: 9.37629910935212 Y: 5.87600014783627 X: 3.50581803398191

Copyright © 2015, 2016 by Namir Clement Shammas

CIYHAT, CIYHATFIT	The command CIYHAT calculates the upper and lower limits of the confidence interval for the projection of an of X onto Y. The value in the X stack register supplies the value of X used to calculate Y^. The value in the Y stack register provides the confidence level as a fraction (usually 0.95). The command returns the following values in the stack: T: Z: The upper limit of the confidence interval for Y^. Y: The value of Y^. X: The lower limit of the confidence interval for Y^. The command CYHATFIT is similar to the CIYHAT command, except it expects the projected value X to be part of the *original observations* that was used by commands S+ and LRXY to calculate the regression statistics. It is your responsibility to use the commands CIYHAT and CIYHATFIT correctly, since the application does not store the original observations processed by the command S+. The commands CIYHAT and CIYHATFIT yield slightly different confidence intervals for Y^. Keep in mind that if you had transformed the X values for the regression calculations, then you need to also transform the value for X before calling this command. Likewise, if you had transformed the Y values for the regression calculations, you need to perform inverse transformations for the two values that define the confidence interval.	0.95 1 ciyhat Returns, for example, the following results: Z: 18.6952228289432 Y: 15.0909090909091 X:1 1.486595352875

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	The commands CIYHAT and CIYHATFIT assume that you have used the command LRXY recently. If not, it returns the values of 1E+99 in the Z, Y, and, X stack registers.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
CLEARSIGMA	Clear the statistical registers.	clearsigma
CLRDSP	Clear the display format. Subsequent output appears unformatted.	`clrdsp`
CLREG	Clear the memory registers.	clreg
CLREGX, CLREGY, CLREGZ, CLREGT	Clear a range of the memory registers using one of the four stack registers. The value of the range is aaaa.bbbb.cc. A value of 0 or less clears all of the memory registers and ends up doing the same task as command CLREG.	0.001002 clregt Clear the memory registers 0, 2, 4, 6, 8, and 10 using the range defined in the T stack register.
CLST, CLSTK, CLRSTK	Clear the stack registers.	clst
COMB, PERM	Calculate the combination and permutation for arguments in the X and Y registers.	4 6 comb Returns Comb(6,4) as 15.
COPYRIGHT	Display the application's copyright.	
COPYSTACK	Copy the values of the regular stack into the backup stack. This command is used with the SCAN command.	
COV	Calculates the covariance of the X and Y variables using values in the satirical registers This command returns the result calculated as: $$\sum xy - [\sum x \cdot \sum y]/n$$	cov Returns something like 2.345982.
D–>R	Convert degrees to radians.	60 d–>r Returns 1.047197551
DSP	Set the display format. See Appendix A for the display formats. The command checks that the format starts with a valid format character followed by one or more digits. If the specified format is valid, the command places 1 in the X stack register. Otherwise, it places 0 in the X stack register.	Dsp f3 Sets the display format to fixed with three decimal places.
DSP?	Show the current display format.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
ENTER	Push the value of the X register up the stack. Unlike with the physical HP RPN calculators, you need not use the command ENTER to separate the input of a sequence of multiple numbers. In calculation mode, an RPN expression can separate a sequence of numbers using single spaces. In program mode, the separate lines are able to input a sequence of values.	3 enter enter enter Fill the stack with the number 3.
ERF, ERFC	Calculate the error and complementary error functions. The X stack register provides the argument for this command.	2 erf Returns 0.99532226
EULER	Push the Euler constant in the stack.	
EXP, 10^	Calculate the exponential and power of 10, respectively. The X stack register provides the argument for these commands.	0 exp Returns 1.
EXPOINTEG	Calculate the exponential integral function. The X stack register provides the argument for this command.	1.4 expointeg Returns 3.00720746
F_CDF and F_ICDF	Calculate the Fisher–Snedecor F cumulative distribution function and its inverse. The X stack register provides the argument for this command. The stack provides the following input: T: Z: The degrees of freedom df1. Y: The degrees of freedom df2. X: The argument X. In the case of the command for the inverse pdf, the value of the X stack register is the probability expressed as a fraction.	8 6 0.6 f_cdf Returns the value for F(8,6,0.6) which equals 0.2461533829 8 6 0.24615 f_icdf Returns Finv(8, 6, 0.24615) which equals 0.599999

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
F_PDF	Calculate the Fisher–Snedecor F probability distribution function. The stack provides the following input: T: Z: The degrees of freedom df1. Y: The degrees of freedom df2. X: The argument X.	7 4 2 f_pdf Returns 0.1613689198
FACT	Calculate the factorial. The integer part of the value in the X stack register provides the argument for this command.	6 fact Returns 120.
FIB	Calculate the Fibonacci number. The integer value in the X register provides the sequence number for the Fibonacci number.	10 fib Returns 55 which is the 10^{th} Fibonacci number.
FIX	Set the display format to be fixed with a specific number of decimal places.	fix 3
FLIP	Toggle the logical value of a flag, without testing the state of that flag.	flip 1 Flips flag(1)
FRC, FRAC	Return the fractional part of the value in the X register.	Pi frac Returns 0.1415926535
GAMMA	Calculate the Gamma function. The X register provides the argument for the Gamma function.	5.5 gamma Returns 52.3427777.
GETDATE	Return the current system date in the stack calculated as yyyy.mmdd using: Year + Month/100 + Day/10000	getdate Returns the current system date.
GETHR, GETMIN, GETSEC	Return the current system hour, minute, and second in the stack.	
GETMAXMEM	Return the maximum number of memory registers in the X stack register. The highest memory register index would be that number minus one.	Getmaxmem Returns 10000.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GETNOW	Return the current system date and time in the Y and X stack registers, respectively.	Getnow -stk- Display the stack showing the current date and time in the Y and X stack registers, respectively.
GETTIME	Return the current system time in the stack calculated as hh.mmss using: Hour + Minute/100 + Second/10000	gettime Returns the current system time.
GETYEAR, GETMONTH, GETDAY	Return the current system year, month, and day, respectively, in the stack.	getyear Returns 2016
HMS–	Subtract H.MS values located in the Y and X registers.	12.3 4.25 hms– Returns 8.056
HMS+	Add H.MS values located in the Y and X registers.	2.3 4.25 hms+ Returns 16.536
HMS–>	Convert an H.MS value to a decimal value.	12.3012 hms–> Returns 12.503333333
IERF, IERFC	Calculate the inverse error and inverse complementary error functions. The X stack register provides the argument for this command.	0.5 ierf Returns 0.4766780021
IFACT	Calculate the product of integers between n to m. This is equal to m!/n!. The integer parts of the values in the Y and X stack registers supply the values for n and m, respectively.	5 6 ifact Returns 30.
IGAMMA	Calculate the incomplete Gamma function *P*(a, x). The X and Y stack registers provide the values for parameter x and a, respectively. Note: $P(a, x) = \gamma(a, x)/\Gamma(a)$ $= 1/\Gamma(a) \int_0^x e^{-t} t^{a-1} dt \ \ where\ a > 0$	2 1 igamma Returns 0.264241117.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
IICGAMMA	Calculate the inverse incomplete Gamma function inv*P*(a, x). The X and Y stack registers provide the values for parameter x and a, respectively. Note: *P*(a, x)= $\gamma(a,x)/\Gamma(a)$ $= 1/\Gamma(a) \int_0^x e^{-t}t^{a-1}dt \ where\ a > 0$	2 0.26 iigamma Returns 0.999095857.
INT	Return the integer part of the value in the X register.	pi int Returns 3.
LASTX, LASTY, LASTZ, LASTT	Recall the last X, last Y, last Z, and last T registers, respectively.	355 113 / lastx * Returns 355.
LN, LOG	Calculate the natural and common logarithms, respectively. The value in the X register provides the argument for the logarithms.	100 ln Returns 4.605170185
LNFACT	Calculate the natural logarithm of the factorial. The X register provides the argument for the factorial. If the argument is zero, the command returns –1 in the X stack register.	1000 lnfact Returns 5912.128178.
LNGAMMA	Calculate the natural logarithm of the Gamma function. The X register provides the argument for the logarithm of the Gamma function. If the command encounters a runtime error, it returns –1 in the X stack register.	1000 lngamma Returns 5905.220423.
LRXY	Perform a simple linear regression between X and Y, using the data already in the statistical summations. This command returns the following values in the stack: T: Z: The coefficient of determination (R^2). Y: The intercept. X: The slope.	clearsigma 2.5 1 S+ 4 2 S+ 5.5 3 S+ lrxy showstk Returns the following stack values: Z: T: 1 Y: 1 X: 1.5

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
MEAN	Calculate the means of Y and X variables based on the values in the statistical registers and store them in the Y and X registers, respectively.	Mean Returns the mean value based on the data in the statistical registers.
MEANSDEV	Calculate the mean and standard deviation for values in the memory registers. The X register defines the indices of the first and last memory register using aaaa.bbbbcc format. Here aaaa and bbbb are the indices of the first and last memory registers to process, respectively. The value of cc is the increment (a zero increment is translated into an increment of 1).	1.002002 MEANSDEV Finds the mean and standard deviation for values in register 1 through 20, skipping every other register.
MEMCOPY	Copy values from one block in the memory registers to another. The value of the X register, aaaa.bbbb, defines the first indices of source and target memory registers. The integer value of the Y register specifies the number of registers to copy. The command returns the number of elements that were actually copied, in the X stack register.	10 0.0050 MEMCOPY Copies 10 values from Mem(0)…Mem(9) to Mem(50)…Mem(59).
MEMSWAP	Swap values between two blocks in the memory registers. The value of the X register, aaaa.bbbb, defines the first indices of the two blocks of memory registers to swap. The integer value of the Y register specifies the number of registers to swap. The command returns the number of elements that were actually swapped, in the X stack register.	10 0.0050 MEMSWAP Swaps 10 values between Mem(0)…Mem(9) and Mem(50)…Mem(59).

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
MFILL	Fill a range of memory registers with a fixed value. The stack provides the following information: T: Z: Y: The range of registers affected. This value has the format aaaa.bbbbcc. X: The value to fill in the range of memory registers.	.0010 1 mfill Fills the memory registers Mem(0) to Mem(10) with the value of 1.
MFILLINTRND	Fill a range of memory registers with uniformly-distributed random integers. The stack provides the following information: T: Z: The range of registers affected. This value has the format aaaa.bbbbcc. Y: The high integer limit. X: The low integer limit.	.0010 10 1 mfillintrnd Fills the memory registers Mem(0) to Mem(10) with the random integers in the range of [1, 10].
MFILLNORMRND	Fill a range of memory registers with normally-distributed random values. The stack provides the following information: T: Z: The range of registers affected. This value has the format aaaa.bbbbcc. Y: The standard deviation for the random numbers generated. X: The mean value for the random numbers generated.	.0010 10 1 mfillnormrnd Fills the memory registers Mem(0) to Mem(10) with the normally-distributed random values with a mean of 1 and standard deviation of 10.
MFILLRND	Fill a range of memory registers with uniformly-distributed random values. The stack provides the following information: T: Z: The range of registers affected. This value has the format aaaa.bbbbcc. Y: The high limit. X: The low limit.	.0010 10 1 mfillrnd Fills the memory registers Mem(0) to Mem(10) with the random values in the range of [1, 10].

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
MFILLSEQ	Fill a range of memory registers with a sequence of values. The stack provides the following information: T: Z: The range of registers affected. This value has the format aaaa.bbbbcc. Y: The increment value. Can be positive, zero, or negative. X: The initial value. Can be positive, zero, or negative. This command fills the sequence of values expressed by: $$Value(i) = initialValue + i \cdot incrementValue$$ For *i* =0, 1, 2, 3, …	.0010 10 1 mfillrnd Fills the memory registers Mem(0) to Mem(10) with the values thathave the sequence of 1, 11, 21, 31, and so on.
MOD	Perform the modulus operation.	113 13 mod Returns 9.
NEXTPRIME, PREVPRIME	Calculate the next and previous prime, respectively. The integer value in the X register specifies the starting integer which may or may not be a prime itself.	5 nextprime Returns 7. 11 prevprime Returns 7.
P–>R	Perform a polar–to–rectangular conversion. The command obtains the values for the radian-angle and radius from the Y and X stack registers, respectively. The command places the Y and X values in the Y and X stack registers, respectively.	Pi 4 / 25 p–>r showstk Displays: Y: 17.6776695 X: 17.6776695
PI	Insert the value of pi into the X register and push up the stack.	Pi 4 / Returns 0.78539816

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
Q_CDF and Q_ICDF	Calculate the Normal cumulative distribution function and its inverse. The X stack register provides the argument for this command. In the case of the inverse Normal pdf, the value of the X stack register is the probability expressed as a fraction.	0.6 q_cdf Returns 0.7257469354 0.75 q_icdf Returns 0.67418914
Q_PDF, Q_STD	Calculate the normal probability distribution function. The stack provides the following input: T: Z: The standard deviation. Y: The mean. X: The argument X. The Q_STD command calculates the standard normal pdf with a mean of 0 and a standard deviation of 1. This command only requires a value from the X stack register.	1 0 .6 q_pdf Returns 0.333224602 .6 q_std Returns 0.333224602
R–>D	Convert radians to degrees.	Pi 4 / r–>d Returns 45.
R–>P	Perform a rectangular–to–polar conversion. The command obtains Y and X values from the Y and X stack registers, respectively. The command places the values for the radian-angle and radius in the Y and X stack registers, respectively.	3 4 r–>p showstk Displays: Y: 0.64350110 X: 5
RAND	Return a uniform random value between 0 and 1 (exclusive).	Rand
RANDNORM	Return a normally–distributed random number. This function uses the values in the X and Y registers to represent the mean and standard deviation, respectively. The command uses these values to calculate the random number.	1.0 0.0 randnorm Generates a standard normal *N*(0, 1) random number.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
RCL, RCL+, RCL–, RCL*, RCL/	Recall the value from the specified memory register. This command offers options for register arithmetic.	rcl 03 rcl+ 44
RCLIND, RCLIND+, RCLIND–, RCLIND*, RCLIND/	Indirectly recall the value from the specified memory register. This command offers options for register arithmetic.	rclind 03 rclind+ 44
RCLSTX, RCLSTY, RCLSTZ, RCLSTT. RCLSTINDX, RCLSTINDY, RCLSTINDZ, RCLSTINDT.	Recall the value in a stack register and push it in the X register. The application also supports using indirect address with the various stack recall commands.	RCLSTZ
RDN	Roll down the stack registers.	Rdn rdn rdn Uses the RDN command, the long way, to display the T stack register.
RUP	Roll up the stack registers.	Rup Uses the command RUP to display the T stack register.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
SCALE	Return the value 10000. Use this constant in preparing loop control variables that work with the commands ISG, DSE, and DSL.	1 2000 SCALE / + Sto 0 … Lbl 0 … Isg 0 Gto 0 Prepares the memory register 0 to be a loop control variable. This loop control variable iterates between the values of 1 to 2000.
SCALE41	Return the value 1000. Use this constant in preparing loop control variables that work with the commands ISG41, DSE41, and DSL41.	1 100 SCALE41 / + Sto 0 … Lbl 0 … Isg41 0 Gto 0 Prepares the memory register 0 to be a loop control variable. This loop control variable iterates between the values of 1 to 100.
SCI	Set the display format to a scientific notation with a specific number of decimal places.	sci 3 Sets the output to a SCI 3 format mode.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SDEV	Calculate the standard deviations of the Y and X variables, using the values in the statistical registers, and store them in the Y and X registers, respectively.	
SHOWSTK	Display the values of the stack registers.	
SI	Calculate the sine integral function. The X stack register provides the argument for this command.	2.4 si Returns 1.7524855
SIGMA–, S–	Subtract the values of the Y and X registers from the statistical registers. This command assumes that the data you are removing matches one that you previously entered using the command SIGMA+ or S+. The application has no way to verify the truth of this assumption.	Clearsigma 1 1 S+ 2 2 S+ 3 3 S+ 3 3 S– 4 4 S+ LRXY showstk Returns the following stack values: Z: T: 1 Y: 0 X: 1
SIGMA+, S+	Add the values of the Y and X registers to the statistical registers.	clearsigma 1 1 S+ 2 2 S+ 3 3 S+ LRXY showstk Returns the following stack values: Z: T: 1 Y: 0 X: 1

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SIGMA++, S++	Add values from the memory registers to the statistical registers. The X stack register contains the value aaaa.bbbb that defines the range of memory registers aaaa to bbbb where the pairs of (X, Y) values reside. This command does not support transforming the pairs of (X, Y) values. If you wish to transform the data use the command S+T in Table 3. This command does not clear the statistical registers before adding the data. You are responsible for separately initializing the statistical registers. This feature allows you to add values from different areas of the memory registers using multiple calls to this command.	1.0012 S++ Adds the data in memory registers Mem(1) to Mem(12) as pairs of (X, Y) values to the statistical registers.
SIGN	Return the sign of the value in the X stack register. The command returns 1, 0, and -1, for positive values, zero, and negative values, respectively, in the X stack register.	3 sign Returns 1.
SIN, COS, TAN	Calculate the common trigonometric functions. The arguments are angles in radians.	PI 4 / sin Returns 0.7071067811
SIND, COSD, TAND	Calculate the common trigonometric functions. The arguments are angles in degrees.	45 sind Returns 0.7071067811
SINH, COSH, TANH	Calculate the common hyperbolic functions.	1 sinh Retiurns 1.175201193
SQRT	Calculate the square root value.	144 sqrt Returns 12.
STO, STO+, STO–, STO*, STO/	Store with optional arithmetic operations. The X register contains the value to store.	sto 04 sto+ 05 sto– 10

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
STOIND, STOIND+, STOIND–, STOIND*, STOIND/	Perform an indirect store with optional arithmetic operations. The X register contains the value to store. The argument contains the memory register number to use for indirect addressing.	stoind 10 stoind+ 43
STOSTX, STOSTY, STOSTZ, and STOSTT STOSTINDX, STOSTINDY, STOSTINDZ, and STOSTINDT	Store the value of the X register in the X, Y, Z, or T registers. The STOSTX is practically a NOP (no operation) command. The application also supports register arithmetic with these commands. Examples are STOSTY+, STOSTZ*, and STOSTT/. In addition, the application supports indirect addressing for the STOSTX, STOSTY, STOSTZ, and STOSTT commands. Examples are STOINDX+, STOSTINDY, STOSTINDY/, and STOSTINDT/.	stostz*
SUMNEGPWR	Calculate the following summation: $\sum_{i=j}^{n} 1/x\text{^}i$ The stack provides the following information: T: Z: The (positive) value for the upper summation limit n. Y: The (positive) value for the lower summation limit j. X: The value for x.	10 1 1.1 sumnegpwr Returns 6.144567105. 10 1 –1.1 sumnegpwr Returns –0.29259843.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SUMNEGPWRTOL	Calculate the following summation until the absolute value of an added term reaches or falls below a tolerance limit: $$\sum_{i=j,x>1}^{\left\lvert\frac{1}{x^i}\right\rvert<Toler} \frac{1}{x^i}$$ The stack provides the following information: T: Z: The tolerance value. Y: The (positive) value for the lower summation limit j. X: The value for x which must be greater than 1. If the argument for x is 1 or less, the command returns a large value.	1e-8 1 1.1 sumnegpwrtol Returns 9.999999906.
SUMPWR	Calculate the following summation: $$\sum_{i=j}^{n} x\text{^}i$$ The stack provides the following information: T: Z: The (positive) value for the upper summation limit n. Y: The (positive) value for the lower summation limit j. X: The value for x.	10 1 1.1 sumpwr Returns 17.53116706. 10 1 –1.1 sumpwr Returns 0.834817479.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	***Purpose***	***Example***
SUMPWRTOL	Calculate the following summation until the absolute value of an added term reaches or falls below a tolerance limit: $$\sum_{i=j,x<1}^{\lvert x^i \rvert<Toler} x^i$$ The stack provides the following information: T: Z: The tolerance value. Y: The (positive) value for the lower summation limit j. X: The value for x which must be less than 1. If the argument x is 1 or greater, the command returns a large value.	1e-8 1 .9 sumpwrtol Returns 8.999999911.
SUMX, SUMX2, SUMY, SUMY2, SUMXY, SUMN	Push the sum of X, sum of X squared, sum of Y, sum of Y squared, sum of X*Y, and the number of statistical observations, respectively, in the stack.	Sumn X>0? Gto there Returns the number of observations in the stack and test its value for being positive.
T_CDF and T_ICDF	Calculate the Student–t cumulative distribution function and its inverse. The stack provides the following input: T: Z: Y: The degrees of freedom. X: The argument X. In the case of the command for the inverse pdf, the value of the X stack register is the probability expressed as a fraction.	5 0.5 t_cdf Returns 0.6808505641 5 0.7 t_icdf Returns 0.5589837309

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
T_PDF	Calculate the Student–t probability distribution function. The stack provides the following input: T: Z: Y: The degrees of freedom. X: The argument X.	10 1 t_pdf Returns 0.2303619892
VERSION	Display the application's version.	
X^2	Calculate the squared value.	12 x^2 Returns 144.
X<>nnn, Y<>nnn, Z<>nnn, T<>nnn	Swap the value in the X, Y, Z, or T stack register with a memory register at index nnn. The index of the memory register is part of the command.	X<>10 Swaps the X register with the memory register Mem(10). T<>55 Swaps the T register with the memory register Mem(55).
X<>Y, X<>Z, X<> Z, Y<>Z, Y<>T, Z<>T	Swap values between various stack registers.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
XHAT	Project the value of the Y on X using the value in the X register and the results of a recent invocation of the LRXY command. If the application Is unable to provide you with the correct projected value of X, it places a number higher than 1E99 in the X register. Please keep in mind that if you transformed the original X and/or Y data, then you need to transform the projected Y value before using this command. In addition, if you transformed the original values for variable X, then you need to apply an inverse transformation to the result of the XHAT command. Failing to observe these rules yield meaningless results!	12 Xhat 1E99 X<>Y X>Y? Gto err
Y^X	Raise the value of the Y register to power found in the X register	2 3 y^3 Returns 8.
YHAT	Project the value of the X on Y using the value in the X register and the results of a recent invocation of the LRXY command. If the application Is unable to provide you with the correct projected value of Y, it places a number higher than 1E99 in the X register. Please keep in mind that if you transformed the original X and/or Y data, then you need to transform the projected X value before using this command. In addition, if you transformed the original values for variable Y, then you need to apply an inverse transformation to the result of the YHAT command. Failing to observe these rules yield meaningless results!	12 Yhat 1E99 X<>Y X>Y? Gto err

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ZETA	Calculate the Riemann Zeta function. The X register contains the argument for the function and the Y register contains the tolerance value. If this value is less than 1E-8, the command uses 1E-8 as the working tolerance value. The Riemann Zeta function is defined by the following equation: $$\zeta(s) = \sum_{n=0}^{\infty} 1/(n+1)\text{^}s$$	1e-8 2 1e-8 zeta Returns 1.644834071.
ZETA2	Calculate the Hurwitz Zeta function, ζ(s, q), (a generalized form of the Riemann Zeta function, ζ(s, 1)). The Hurwitz Zeta function is defined by the following equation: $$\zeta(s,q) = \sum_{n=0}^{\infty} 1/(n+q)\text{^}s$$ The stack provides the following input data: T: Z: The q factor. Y: The tolerance value. X: The argument s.	1 1e-8 2 1e-8 zeta2 Returns 1.644834071.

Table 2. Functions used in calculation and programming modes.

Programming and Advanced Functions

Table 3 shows the list of commands that support programming and advanced functions and operations. Appendices B and C contain summaries for the various types of store and recall commands that you can use in writing your programs that run under the CPRCA application.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
' (single quote) " (double quote)	Copy text after the single or double quote to the Alpha register. You can optionally end the text with a matching single or double quote to insert trailing spaces in the Alpha register. Otherwise, the trailing single or double quote character is not needed.	'Hello "Hello World "
|– (bar and dash/minus characters)	Append text to the alpha register. You can optionally end the text with a single or double quote to insert trailing spaces in the Alpha register. Otherwise, the trailing single or double quote character is not needed	|– world!
+RIDX, +SIDX	Increment or decrement the value of the recall or storage index. The integer value of the X stack register (which can be positive or negative) specifies the shift in the index value. The command may include the name of the array. If omitted, the command uses the Alpha register to select the array variable.	‘Xarr 0 Sidx … 4 +sidx
ADDDT	Add a date unit stored in a text variable. The name of the text variable is the argument for this command. The integer value of the X register contains the value to add. The Alpha register contains the unit of time to add. These units are: • *Year* to add years. • *Month* to add months. • *Day* to add days. • *Hour* to add hours. • *Minute* to add minutes. • *Second* to add seconds. • *Week* to add week days. • *Weekofyear* to add weeks.	'#1/1/2016 2:00:00 PM#" asto dt1 1 'Day adddt dt1 viewdt dt1 Adds 1 day to the date 1/1/2016 stored in text variable dt1. The last command displays “1/2/2016 2:00:00 PM” that is currently stored in text variable dt1.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
AINPUT	Display the contents of the Alpha register as a prompt message and replace the content of the Alpha register with the user's input.	'Enter filename ainput asto filename
ARCL	Copy text from a named text register into the Alpha register.	Arcl Hi Copies text stored in text variable Hi to the Alpha register.
ARCL–	Prepend text from a named text register to the Alpha register.	Arcl– Hi Prepends text stored in text variable Hi to the Alpha register.
ARCL+	Append text from a named text register to the Alpha register.	Arcl+ Hi Appends text stored in text variable Hi to the Alpha register.
ARCLIX, ARCLIY, ARCLIZ, ARCLIT	Append the integer part of the selected stack register to the Alpha register.	Pi sqrt 'X= ARCLIX AVIEW Displays "X=3".
ARCLS–	Prepend text from a named text register to the Alpha register. The command inserts a space between the two merged strings.	Arcls– Hi Prepends text stored in text variable Hi to the Alpha register.
ARCLS+	Append text from a named text register to the Alpha register. The command inserts a space between the two merged strings.	Arcls+ Hi Appends text stored in text variable Hi to the Alpha register.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
ARCLX, ARCLY, ARCLZ, ARCLT	Append the value in the selected stack register to the Alpha register.	Pi sqrt 'X= ARCLX AVIEW Appends the square root of pi (located in the X stack register) to the text “X=” in the Alpha register, and then displays the result.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ARCOPY	Copy one array variable into another. The target array variable may or may not already exist. In the latter case, the command creates the target array variable. This command copies the data and other information (such as the array size, sorted–elements status, and recall/storage index), making an exact replica (except for the name) of the array variable. The command can have four types of arguments: 1. Two arguments separated by a comma. The first argument is the name of the source array variable. The second argument is the name of the target array variable. 2. One argument (the source array variable). The command uses the contents of the Alpha register to obtain the name of the target array variable. 3. One argument that starts with a comma to indicate that it is the target array variable. The command uses the contents of the Alpha register to obtain the name of the source array variable. 4. No arguments. The command expects the Alpha register to contain the names of the two array variables, separated by a comma.	# Option 1 Arcopy xarr,yarr # Option 2 'yarr Arcopy xarr # Option 3 ‘Xarr Arcopy ,yarr # Option 4 ‘Xarr,yarr arcopy Each of the four examples copies the data from the array Xarr into the array Yarr.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
ARDEL	Delete an array variable. This command may include the name of the array to be removed. If omitted, the command uses the Alpha register to obtain the name of the targeted array.	Ardel xarr Deletes the array Xarr.
ARFILLINTRND	Fill an array variable with random integers. The command can include the name of the array. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The Y and X stack registers provide the high and low integer limits, respectively, for the uniformly distributed random numbers.	‘Xarr 100 1000 ARFILLINTRND Fills the elements of array Xarr with uniformly distributed random numbers in the range of 100 to 1000.
ARFILLNORMRND	Fill an array variable with normally–distributed random numbers. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The Y and X stack registers provide the standard deviation and mean values, respectively, for the normally–distributed random numbers.	‘Xarr 1 0 ARFILLNORMRND Fills the elements of array Xarr with normally–distributed random numbers with a mean of 0 and a standard deviation of 1.
ARFILLRND	Fill an array variable with uniformly–distributed random values. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The Y and X stack registers provide the high and low limits, respectively, for the uniformly distributed random numbers.	‘Xarr 1 10 ARFILLRND Fills the elements or array Xarr with uniformly–distributed random numbers in the range of 1 to 10.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ARFILLSEQ	Fill the elements of an array variable with a sequence of values. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The Y and X stack registers provide the increment in values and the initial value, respectively, for the sequence of numbers. The initial values and the increment in values can be zero, negative values, positive values, or any combination of these values.	‘Xarr 1 10 ARFILLRND Fills the array Xarr with a sequence of numbers that starts with 10 and increases by 1.
ARGETSIZE	Return the number of array elements in an array variable. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The command stores the array size in the X stack register.	‘Xarr Argetsize Returns the number of elements in array Xarr.
ARNEW	Create a new array variable. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The X register specifies the number of array elements. These elements are numbered from 0 and up to the size of the array minus one. The command stores zeros in the array elements. If the array variable already exists, the command resizes and reinitializes the array variable.	100 ‘Xarr ARNEW Creates an array of 100 elements and name it Xaerr.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ARNEWINTRND	Create a new array variable and fill it with uniformly–distributed random integers. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The X register specifies the number of array elements. The Z and Y stack registers provide the high and low integers, respectively, for the uniformly–distributed random numbers. If the array variable already exists, the command resizes and reinitializes the array variable.	‘Xarr 1000 100 50 ARNEWINTRND Creates an array of 50 elements and name it Xarr. Fills the array with uniformly–distributed random numbers in the range of 100 to 1000.
ARNEWNORMINTRND	Create a new array variable and fill it with normally–distributed random integers. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The X register specifies the number of array elements. The Z and Y stack registers provide the standard deviation and mean, respectively, for the normally–distributed random integers. If the array variable already exists, the command resizes and reinitializes the array variable.	‘Xarr 10 0 50 ARNEWNORMINTRND Creates an array of 50 elements and name it Xarr. Fills the array with normally–distributed random integers with a mean of 0 and a standard deviation of 10.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ARNEWNORMRND	Create a new array variable and fill it with normally–distributed random numbers. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The X register specifies the number of array elements. The Z and Y stack registers provide the standard deviation and mean, respectively, for the normally–distributed random numbers. If the array variable already exists, the command resizes and reinitializes the array variable.	‘Xarr 1 0 50 ARNEWNORMRND Creates an array of 50 elements and name it Xarr. Fills the array with normally–distributed random numbers with a mean of 0 and a standard deviation of 1.
ARNEWRND	Create a new array variable and fill it with uniformly–distributed random values. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The X register specifies the number of array elements. The Z and Y stack registers provide the high and low limits, respectively, for the uniformly–distributed random numbers. If the array variable already exists, the command possibly resizes and reinitializes the array variable.	‘Xarr 10 1 100 ARNEWRND Creates an array of 100 elements and name it Xarr. Fills the array with uniformly–distributed random numbers in the range of 1 to 10.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ARNEWSEQ	Create a new array variable and fill it with a sequence of values. The command can include the name of the array variable. Otherwise, the command uses the contents of the Alpha register to obtain the name of the array variable. The stack provides the following information: T: Z: The increment value for the sequence. Y: The initial value for the sequence. X: The number of array elements. The initial values and the increment in values can be 0, negative values, positive values, or any combination of these values. If the array variable already exists, the command possibly resizes and reinitializes the array variable.	‘Xarr 1 10 100 arnewseq Creates an array of 100 elements and name it Xarr. Fills the array with a sequence of numbers that starts with 10 and increases by 1. The sequence stored would be 10, 11, …, 109.
ARRESET	Clear the memory associated with an array variable. The command can specify the array variable or use the contents of the Alpha register to obtain the array variable.	arrreset Xarr Resets the memory of the array variable Xarr.
ARSWAP	Swap the data of two array variables. The command takes two space–delimited array names. If the command has no arguments, it expects the Alpha register to contain the names of the two arrays. Including just one array name or no names generates a runtime error. This command simply swaps the names of the array objects. As such, there is no need to swap the values in the targeted arrays.	Arswap Xarr Yarr Swaps the data associated with the array variables Xarr and Yarr.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ARVAR2MEM	Copy values from an array variable to the memory registers. The command can specify the array variable or use the contents of the Alpha register to select the array variable. The stack contains the following parameters: T: Z: The index of the first array variable element that is copied from. Y: The index of the last memory registers that is copied into. X: The index of the first memory registers that is copied into. The command does it's best to comply with the number and location of the copied data. The command stores the number of actual values copied in the X register.	‘Xarr 0 22 0 arvar2mem Copies values from the array variable Xarr, starting with the first array element, into memory registers Mem(0) to Mem(22).
ASTO	Store the content of the Alpha register in a named text register.	'Hello World asto Hi
ATOX	Convert the number stored in the Alpha register into a number and store it in the X register.	'1.234 Atox Returns 1.234.
AVIEW	Display the content of named variable that stores text. If you omit this argument, the command displays the contents of the Alpha register. The program resumes execution after displaying text.	'Hello Aview Displays Hello
AVIEW2	Display the content of named variable that stores text. If you omit this argument, the command displays the contents of the Alpha register. This command prompts the user to press Enter to continue program execution.	'Hello Aview2 Displays: Hello Press Enter to resume …

Copyright © 2015, 2016 by Namir Clement Shammas

BESTLR	Search for the best linearized regression model for X and Y in powers ranging from –4 to 4, in increments of 0.5 for each variable. The command interprets the power of zero as a request to take the logarithm of the variable. This command uses a block of data read using the READDATA2 command. The stack provides the following input data: T: Z: Y: X: Index for the first register in the block read using READDATA2. The program returns the following values: T: Z: Coefficient of determination, R^2, for the best curve fit. Y: Intercept for the best curve fit. X: Slope for the best curve fit. The Alpha register contains a string that describes the best model. Use the BESTLRPWRS to obtain the powers of variables X and Y for the best curve fit. The command writes the regression results for each model to the comma-delimited file BESTLR_*date_stamp*.CSV. The application uses the current date stamp as part of the filename. This approach creates a unique filename and thus avoid writing over files generated by previous sessions. You can open the .CSV files with Excel. The first row contains the header for the columns. You can easily sort the data	See examples.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	using the values in the first column (the coefficient of determination) as the key values for sorting the data in a descending order. Each row contains the following values: • The coefficient of determination. • The transformation power for variable Y. • The transformation power for variable X. • The intercept. • The slope.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
BESTLRPWRS	Return the best powers for the regression variables Y and X in the Y and X registers, respectively.	See examples.

Copyright © 2015, 2016 by Namir Clement Shammas

BESTMLR	Search for the best multiple regression model for variables X, Y and Z in powers ranging from –4 to 4, in increments of 0.5 for each variable. The command interprets the power of zero as a request to take the logarithm of the variable. This command uses a block of data read using the READDATA3 command. The stack provides the following input data: T: Z: Y: X: Index for the first register in the block read using READDATA3. The program returns the following values: T: Coefficient of determination, R^2, for the best curve fit. Z: Intercept for the best curve fit. Y: Slope for Y for the best curve fit. X: Slope for X for the best curve fit. The Alpha register contains a string that describes the best model. Use the BESTMLRPWRS to obtain the powers of variables X, Y and Z for the best curve fit. The command writes the regression results for each model to the comma-delimited file BESTMLR_*date_stamp*.CSV. The application uses the current date stamp as part of the filename. This approach creates a unique filename and thus avoid writing over files generated by previous sessions. You can open the .CSV files with Excel. The	See examples.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	first row contains the header for the columns. You can easily sort the data using the values in the first column (the coefficient of determination) as the key values for sorting the data in a descending order. Each row contains the following values: • The coefficient of determination. • The transformation power for variable Y. • The transformation power for variable X. • The intercept. • The slope.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
BESTMLRPWRS	Return the best powers for the regression variables Z, Y and X in the Z, Y and X registers, respectively.	See examples.

Copyright © 2015, 2016 by Namir Clement Shammas

BINSEARCHA	Perform an efficient binary search for a value in a sorted array variable. The command may include the name of the array. If this argument is omitted, the command uses the Alpha register to specify the array variable to search. The Stack provides the following search parameters: T: Z: Y: Tolerance value. X: The search value. This command will sort the array variable if its elements are not already sorted. The supporting software keeps track of the in–order state of each array variable, as you manipulate its elements. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values. The command returns the following results in the stack registers: T: Z: The number of *equal* (within the specified tolerance value) values located above the matching element. Y: The number of *equal* (within the specified tolerance value) values located below the matching element. X: The index of the searched value. The command returns the array index of the matching element in register X. Since this index does not necessarily	‘Xarr 1E-6 1.5 binsearcha X<0? Gto notFound Performs a binary searche in the array Xarr for the element that stores 1.5. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	point to the first matching item (when duplicates exists) it returns additional information in the other stack registers. Registers Y and Z contain the number of duplicates (if any), that are located below and above the matching array elements, respectively. If there is no matching element in the array variables, then registers Y and Z will contain zeros. If the matching element is unique in the array variable, you also get zeros in the Y and Z stack registers. Otherwise, the values in registers Y and Z depend on the number of duplicates in the array and their location in the array with respect to the matching element. If the supplied array name is valid but the searched value is not found, the command returns –1. If the supplied array name does not exist in the collection of array names, the command returns –2.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
CLA	Clear the Alpha register.	
CLRFLGS	Clear all flags.	

Copyright © 2015, 2016 by Namir Clement Shammas

CSEARCH	Search for a value in the memory registers, using a forward or backward circular search scheme. The search begins at a chosen element and moves to either ends of the memory registers. If the search does not find a match, it resumes at the other end of the memory registers until it reaches the point where the search started. The Stack provides the following search parameters: T: Search mode. Z: Tolerance value. Y: Index of the first memory register to search. X: The search value. The search mode instructs the command how to compare the search value with the memory registers: 1. Positive to search for the first memory register with a value that is greater than the search value. 2. Zero to search the first memory register with a value that is equal (within the specified tolerance limit) to the search value. 3. Negative to search for the first memory register with a value that is less than the search value. The sign of the Index, of the first memory register to search, determines the direction of the circular search. A zero and positive values move the search in increasing register indices. By	0 1E-6 9 1.5 cSearch X<0? Gto notFound Searches the memory registers for the element that stores 1.5, starting with the 10th element. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	contrast, negative values direct the search in decreasing register indices. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values. The command returns the index of the matching element in the memory register. If the searched value is not found, the command returns –1.	

Copyright © 2015, 2016 by Namir Clement Shammas

CSEARCHA	Search for a value in an array variable, using a forward or backward circular search scheme. The search begins at a chosen element and moves to either ends of the array. If the search does not find a match, it resumes at the other end of the array until it reaches the point where the search started. The command may include the name of the array. If this argument is omitted, the command uses the Alpha register to specify the array variable to search. The Stack provides the following search parameters: T: Search mode. Z: Tolerance value. Y: Index of the first array variable element to search. X: The search value. The search mode instructs the command how to compare the search value with the array variable elements: 1. Positive to search for the first array variable element with a value that is greater than the search value. 2. Zero to search the first array element with a value that is equal (within the specified tolerance limit) to the search value. 3. Negative to search for the first array element with a value that is less than the search value. The sign of the Index, of first array element to search, determines the direction of the circular search. A zero and positive values direct the search in	‘Xarr 0 1E-6 9 1.5 csearcha X<0? Gto notFound Searches the array Xarr for the element that stores 1.5, starting with the 10th element. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	increasing array indices. By contrast, negative values move the search in decreasing array indices. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values. The command returns the index of the matching element in the array variable. If the supplied array name is valid but the searched value is not found, the command returns –1. If the supplied array name does not exist in the collection of array names, the command returns –2.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
DATAM	Insert a sequence of values in the memory registers from a comma–delimited list of values. The first value after the DATAM keyword is the index of the first memory registers that receives the data. The remaining values are sequentially stored in the memory registers.	DATAM 10,1,2,3,4,5 Stores the values 1 to 5 starting at Mem(10).
DATAR	Store a sequence of values in an array variable. The command requires a list of comma–separated values. The first value in the list of data is the name of the array that stores the data. This argument is optional. When omitted, the command looks to the Alpha register to supply the name of the array. The second value in the list is the index where the insertion in the array variable begins. The remaining list values are the data to be inserted.	‘Xarr DATAR 0,0,1,2,3,4,5 ‘Xarr DATAR 6,6,7,8,9,10 The above two DATAR commands store values in the array variable Xarr at indices 0 to 10.
DATAS	Add pairs of (X, Y) values to the statistical registers from a comma–delimited list of values. If the list has an odd number of values, the command ignores the last value.	DATAS 1,2,3,4,5,6

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
DIFFDT	Subtract date units stored in two text variables. The space-delimited names of the text variables are the arguments for this command. The command subtracts the first date from the second one. The command places in the X register the difference in the time units. The Alpha register contains the unit of time to add. These units are: • *Year* to subtract years. • *Month* to subtract months. • *Day* to subtract days. • *Hour* to subtract hours. • *Minute* to subtract minutes. • *Second* to subtract seconds. • *Week* to subtract week days. • *Weekofyear* to subtract weeks.	'#1/1/2016 2:00:00 PM#" asto dt1 '#1/1/2017 2:00:00 PM#" asto dt2 1 'Day diffdt dt1 dt2 viewx Stores two dates in the text variables dt1 and dt2, subtracts the number of days between these dates, and displays the difference which is 366 days.
DRCL	Recall the values of the date and time from to a text variable and into the stack. The Y register contains the floating-point value of the date (as yyyy.mmdd) and the X stack register contains the floating-point value of time (hh.mmss). The name of the text variable is the argument for this command.	drcl dtVar Recalls the date and time from the text variable dtVar. The Y and X stack registers will contain the floating-point format of the date and time, respectively.
DSE, DSEIND	Implement the *Decrement and Skip if Equal*, somewhat like in the HP–41C calculator, but with an extended range that handles 0 to 9999. The DSEIND uses indirect addressing.	lbl 0 …. dse 10 gto 0
DSE41, DSEIND41	Implement the *Decrement and Skip if Equal*, just like in the HP–41C calculator. The DSEIND41 uses indirect addressing.	lbl 0 …. dse41 10 gto 0

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
DSL, DSLIND	Implement the *Decrement and Skip if Less*, which is similar to the DSE command. The command DSLIND uses indirect addressing.	lbl 0 …. dsl 10 gto 0
DSL41, DSLIND41	Implement an HP–41C version of the *Decrement and Skip if Less*, which is similar to the DSE41 command. The command DSLIND41 uses indirect addressing.	lbl 0 …. dsl41 10 gto 0
DSP?	Display the current display format and also copy it to the Alpha register.	
DSTO	Store the values of the date and time found in the stack to a text variable. The Y register contains the floating-point value of the date (as yyyy.mmdd) and the X stack register contains the floating-point value of time (hh.mmss). The name of the text variable is the argument for this command.	2016.0101 6.3000 dsto dtVar Stores the date and time values found in the stack to the text variable dtVar.
DVIEW	View the date text that is stored in a text variable (minus the leading and trailing # characters). The name of the variable is the argument for this command. This command is the same as the command VIEWDT.	'#1/1/2016 2:00:00 PM#" asto dt1 dview dt1 Displays the date in text variable dt1, which is “1/1/2016 2:00:00 PM”.
END	End program execution.	
EXISTALPHAVAR	Test if a named text variable exists. The Alpha register contains the name of the tested text variable. Register X contains the index of the flag that receives the Logical result. The flag will be set if the text variable exists, and cleared if it does not.	‘MyText 1 Existalphavar Tests if there exists a text variable named MyText. The command stores the Boolean result in flag 1.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
EXISTARRNAME	Test if an array variable exists. The Alpha register contains the name of the tested array variable. Register X contains the index of the flag that receives the Logical result. The flag will be set if the array variable exists, and cleared if it does not.	‘MyX 0 Existarrname Tests if MyX is an existing array variable. The command stores the Boolean result in flag 0.
FILTER	Filter out (remove) characters in Alpha register. The argument for this command specifies the name of a text variable that contains the set of characters to filter.	‘!@#$%^&* Asto myFilterStr ‘He$l!lo Wo&rld Filter myFilterStr Updates the Alpha register with the string “Hello World”.

Copyright © 2015, 2016 by Namir Clement Shammas

FOR	Offer a command that supports BASIC–like FOR iterations that can replace the ISG, DSE, and DSL loop control commands. The command uses the name of a text variable. If omitted, the command uses the Alpha register to obtain the name of that text variable. The text variable (which I will also call the *loop control variable*) has text that contains images of numbers organized using the following format: *Current_Value*, *Final_Value*, *Increment* The first parameter is the current value and also serves as the initial value. The second parameter specifies the upper limit for the iterations. The increment parameter specifies the non–zero increment in the current value. Each call to command INC perform the following tests and updates: 1. Increment the current value. 2. If the increment value is positive, the command tests if the current value is less than or equal to the final value. 3. If the increment value is negative, the command tests if the current value is greater than or equal to the final value. 4. If either condition in steps 2 or 3 is true, the command updates the text in the loop control variable. This update reflects the new current value of the loop control variable. 5. If both conditions in steps 2 and 3 is false, the command skips the next program line.	lbl start '1,100,1 asto incVar 0 lbl 0 forcl incvar + for incvar gto 0 end Calculate the sum of integers from 1 to 100 in increments of 1. The above code returns the value 5050.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
FORCL	Recall the current value in a text variable that is used as a loop control variable with the command FOR. The command requires the name of the loop control variable. If omitted, the command uses the Alpha register to obtain the name of the loop control variable.	lbl start '1,100,1 asto incVar 0 lbl 0 forcl incvar + for incvar gto 0 end Calculate the sum of integers from 1 to 100 in increments of 1. The above code returns the value 5050.
FORSET	Set the initial, final, and increment values stored as a string in a text variable. The name of this variable is the argument for this command. If omitted, the command uses the Alpha register to obtain the name of the text variable. The stack provides the following data input: T: Z: The increment value. Y: The final value. X: The initial value.	1 100 1 Foset loopVar Sets the text in the variable loopVar to “1,100,1”.
FRMT	Append a formatted value of the X stack register to the Alpha register. See Appendix A for a summary of the formatting strings used by this command. The command can include the output format. If omitted, the command uses the current display format.	'pi= pi FRMT F2 Displays “pi=3.14”.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
FS? IND, FS?C IND, FS?S IND, FS?FLIP IND, FC? IND, FC?C IND, FC?S IND, FC?FLIP IND	Support for indirect versions of the commands that tests the flags. The space character that appears between any flag test (such as FS?C) and the word IND is optional. Inserting that space enhances readability.	Fc?c ind 0 Gto there Test if the flag specified by memory register 0 is clear and then set that flag.
FS?, FS?C, FS?S, FS?FLIP, FC?, FC?C, FC?S, FC?FLIP	Test if a flag is set or clear. The command set has the option to set, clear, or flip the flag after it is tested to be set or clear.	FC?S 0 Test if flag 0 is clear and then set that flag.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSCHEBQUAD (VER 1)	Integrate the RPN expression in the Alpha register using the Gauss-Chebyshev Quadrature. The Gaussian-Chebyshev quadrature performs a numerical integration for the following integral: $\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w(i)f(x(i))$ Where *w(i)* is a weight associated with each *x(i)* root of the Chebyshev function. The value of each w(i) is equal to π/n. The value of x(i) is calculated using: $x(i) = \cos\left(\frac{2i-1}{2n} * \pi\right)$ The integral is calculated using: $\int_{a}^{b} f(x)dx =$ $(\frac{b-a}{2})(\frac{\pi}{n})\sum_{i=1}^{n} f\left(\frac{(b-a)x(i)}{2} + \frac{a+b}{2}\right) *$ $\sqrt{(1 - x(i) * x(i))}$ The stack provides the following input: T: Z: The order of the Chebyshev polynomial used in the quadrature Y: The upper limit for the integral. X: The lower limit for the integral. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	LBL Start '1/X 100 1 2 gausschebquad end Will integrate 1/X from X=1 to X=2 using the roots of Chebyshev polynomial of order 100.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSCHEBQUAD (VER 2)	Integrate the function defined by a label in the Alpha register. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the function to be integrated. The Gaussian- Chebyshev quadrature performs a numerical integration for the following integral: $\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w(i)f\big(x(i)\big)$ Where *w(i)* is a weight associated with each *x(i)* root of the Chebyshev function. The value of each w(i) is equal to π/n. The value of x(i) is calculated using: $x(i) = \cos\left(\frac{2i-1}{2n} * \pi\right)$ The integral is calculated using: $\int_{a}^{b} f(x)dx =$ $(\frac{b-a}{2})(\frac{\pi}{n})\sum_{i=1}^{n} f\left(\frac{(b-a)x(i)}{2} + \frac{a+b}{2}\right) *$ $\sqrt{(1 - x(i) * x(i))}$ The stack provides the following input: T: Z: The order of the Chebyshev polynomial used in the quadrature Y: The upper limit for the integral. X: The lower limit for the integral. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	LBL Start 'gsb reciprocal 100 1 2 gausschebquad end lbl reciprocal 1/X rtn Will integrate 1/X from X=1 to X=2 using the roots of Chebyshev polynomial of order 100.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSHERQUAD (VER 1)	Integrate the RPN expression in the Alpha register using the Gauss-Hermite Quadrature. This quadrature performs a numerical integration for the following integral: $\int_{-\infty}^{\infty} e^{-x\wedge 2} f(x)dx = \sum_{i=1}^{n} w(i)f(x(i))$ Where *w(i)* is a weight associated with each *x(i)* root of the Hermite function. The stack provides the following input: T: Z: The tolerance value for the result. Y: The step size used in searching for the roots of the Hermite polynomial. X: The order of the Hermite polynomial used in the quadrature. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	‘cos 1E–6 .01 10 gaussherquad Will integrate exp(–x^2)*cos(x) between minus and plus infinity, using a tolerance value of 1E–6 and the roots of Hermite polynomial of order 10. The result should be close to the exact integral of 1.38039.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSHERQUAD (VER 2)	Integrate the function defined by a label in the Alpha register using the Gauss-Hermite quadrature. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the function to be integrated. The Gaussian- Hermite quadrature performs a numerical integration for the following integral: $\int_{-\infty}^{\infty} e^{-x\hat{}2} f(x)dx = \sum_{i=1}^{n} w(i)f(x(i))$ Where *w(i)* is a weight associated with each *x(i)* root of the Hermite function. The stack provides the following input: T: Z: The tolerance value for the result. Y: The step size used in searching for the roots of the Laguerre polynomial. X: The order of the Laguerre polynomial used in the quadrature. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	Lbl start 'gsb fx 1E–6 .01 10 gaussherquad end lbl fx cos rtn Will integrate exp(–x^2)*cos(x) between minus and plus infinity, using a tolerance value of 1E–6 and the roots of Hermite polynomial of order 10. The result should be close to the exact integral of 1.38039.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSKRONQUAD (VER 1)	Perform a Gauss-Kronrod quadrature that is similar to the Gauss-Legendre quadrature, but more accurate. The stack provides the following input: T: The order of the Legendre polynomial used in the quadrature Z: The tolerance value that may be needed by helper function of this commands. Y: The upper value of the integration limit. X: The lower value of the integration limit.	LBL Start 5 1e–7 100 1 '1/X gausskronquad end Displays the integral of 1/X between 1 and 2, using a fifth order polynomial.
GAUSSKRONQUAD (VER 2)	Perform a Gauss-Kronrod quadrature that is similar to the Gauss-Legendre quadrature, but more accurate. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the function to be integrated. The stack provides the following input: T: The order of the Legendre polynomial used in the quadrature Z: The tolerance value that may be needed by helper function of this commands. Y: The upper value of the integration limit. X: The lower value of the integration limit.	LBL Start 5 1e–7 100 1 'gsb fx gausskronquad end lbl fx 1/X rtn Displays the integral of 1/X between 1 and 2, using a fifth order polynomial.
GAUSSKRONQUAD2 (VER 1)	Similar to command GAUSSKRONQUAD. This command differs in the algorithm that handles a wide variety of integration ranges to calculate the weights and abscissa.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSKRONQUAD2 (VER 2)	Similar to command GAUSSKRONQUAD. This command differs in the algorithm that handles a wide variety of integration ranges to calculate the weights and abscissa.	
GAUSSLAGQUAD (VER 1)	Integrate the RPN expression in the Alpha register using the Gauss-Laguerre Quadrature. The Gaussian-Laguerre quadrature performs a numerical integration for the following integral: $\int_0^\infty e^{-x} f(x)dx = \sum_{i=1}^{n} w(i)f(x(i))$ Where *w(i)* is a weight associated with each *x(i)* root of the Laguerre function. The stack provides the following input: T: Z: The tolerance value for the result. Y: The step size used in searching for the roots of the Laguerre polynomial. X: The order of the Laguerre polynomial used in the quadrature. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	LBL Start 'cos 1e–8 0.01 10 gausslagquad end Will integrate exp(-x)*cos(x) from 0 to infinity using a tolerance value of 1E–8, step search of 0.01 and the roots of Laguerre polynomial of order 10. The integral is 0.5.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSLAGQUAD (VER 2)	Integrate the function defined by a label in the Alpha register using the Gauss-Laguerre Quadrature. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the function to be integrated. The Gaussian- Laguerre quadrature performs a numerical integration for the following integral: $\int_0^\infty e^{-x} f(x) dx = \sum_{i=1}^{n} w(i) f(x(i))$ Where *w(i)* is a weight associated with each *x(i)* root of the Laguerre function. The stack provides the following input: T: Z: The tolerance value for the result. Y: The step size used in searching for the roots of the Laguerre polynomial. X: The order of the Laguerre polynomial used in the quadrature. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	lbl start 'gsb one 1e–8 0.01 10 gausslagquad end lbl one cos rtn Will integrate exp(-x)*cos(x) from 0 to infinity using a tolerance value of 1E–8, step search of 0.01 and the roots of Laguerre polynomial of order 10. The integral is 0.5.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSLEGQUAD (VER 1)	Integrate the RPN expression in the Alpha register using the Gauss-Legendre Quadrature. The Gaussian-Legendre quadrature performs a numerical integration for the following integral: $\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w(i)f\big(x(i)\big)$ Where *w(i)* is a weight associated with each *x(i)* root of the Legendre function. The above integral can be mapped to the range [a, b} using the following equations: $\int_{a}^{b} f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{(b-a)x}{2} + \frac{a+b}{2})dx$ $\int_{a}^{b} f(x)dx = \frac{b-a}{2}\sum_{i=1}^{n} w(i)f\left(\frac{(b-a)x(i)}{2} + \frac{a+b}{2}\right)$ The stack provides the following input: T: The tolerance value for the result. Z: The order of the Legendre polynomial used in the quadrature Y: The upper limit for the integral. X: The lower limit for the integral. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	'1/X 1E-6 10 2 1 gausslegquad Will integrate 1/X from X=1 to X=2 using a tolerance value of 1E-6 and the roots of Legendre polynomial of order 10.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GAUSSLEGQUAD (VER 2)	Integrate the function defined by a label in the Alpha register. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the function to be integrated. The Gaussian-Legendre quadrature performs a numerical integration for the following integral: $\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w(i)f\big(x(i)\big)$ Where *w(i)* is a weight associated with each *x(i)* root of the Legendre function. The above integral can be mapped to the range [a, b} using the following equations: $\int_{a}^{b} f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{(b-a)x}{2} + \frac{a+b}{2})dx$ $\int_{a}^{b} f(x)dx = \frac{b-a}{2}\sum_{i=1}^{n} w(i)f\left(\frac{(b-a)x(i)}{2} + \frac{a+b}{2}\right)$ The stack provides the following input: T: The tolerance value for the result. Z: The order of the Legendre polynomial used in the quadrature Y: The upper limit for the integral. X: The lower limit for the integral. The command returns the value of the integral in the X stack register. If there is an error in the calculations, the command displays a warning message and inserts 0 in the X stack register.	lbl Start ‘gsb Recipocal 1E-6 10 2 1 gausslegquad end lbl Reciprocal 1/x rtn Will integrate 1/X from X=1 to X=2 using a tolerance value of 1E-6 and the roots of Legendre polynomial of order 10.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
GETRIDX, GETSIDX	Return the value of the recall or storage indices of an array variable. The command may include the name of the array. If omitted, the command uses the Alpha register to select the array variable.	‘Xarr getridx
GETSTAT	Read the statistical registers from a file. The command can include the input filename. If not, the command uses the contents of the Alpha register as the input filename. The filename must include a full path if it not located in the same directory as that of the CPRCA application.	‘C:\MyData\Data.txt getstat Reads the statistical registers from file Data.txt.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
GFTEST?, GFTEST?C, GFTEST?S, GFTEST?FLIP	Perform a logical test on a group of contiguous flags. The integer value of the X stack register designates the index of the first flag involved in the group test. The argument for this command is a string of 1s (where 1 is the set state of a flag) and 0s (0 is the clear state of a flag) that specify the state of the subsequent flags. The number of 1s and 0s also determine the number of flags involved. If the command has no argument, the application uses the Alpha register as source for the 1s and 0s needed for the test. If the test fails, the CPRCA application skips the next program line. The string pattern must contain 0s and 1s only. Including any other character causes the tests to fail and to generate a runtime error. The command GFTEST?C clears all involved flags when the test is true. The command GFTEST?S sets all involved flags when the test is true. The command GFTEST?FLIP flips the logical states of the flags involved when the test is true. If any one of these tests fails, the flags involved retain their current states.	10 GFTEST? 10010 Gto there Tests the states of the following flags: Flag(10): true → 1 Flag(11): false → 0 Flag(12): false → 0 Flag(13): true → 1 Flag(14): false → 0 If the above flags have states that match their expected logical values, then the command succeeds and the application executes the next program line.
GSB, GSBIND	The command GSB executes a subroutine in a label. The command GSBIND uses indirection to call a subroutine. The command GSB can call a subroutine that uses numeric and alphanumeric labels. By contrast, the command GSBIND can only call subroutines that have numeric labels.	Gsb fx Gsbind 03

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
GTO, GTOIND	command The GTOIND uses indirection to jump to a label. The command GTO can jump to numeric and alphanumeric labels. By contrast, the command GTOIND can only jump to numeric labels.	gto start gtoind 03

Copyright © 2015, 2016 by Namir Clement Shammas

HSEARCH	Perform a heuristic search for a value in the memory registers. The Stack provides the following search parameters: T: Tolerance value. Z: Index of first array variable element to search. Y: Heuristic step (a positive or negative integer) X: The search value. If the index of the memory register to search is negative, the command performs a backward search from the last memory register to the absolute value of the Z register. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values. The heuristic step value allows you to move the matching element up (pessimistic scheme) using a positive value, or down (optimistic scheme) using a negative value. The optimistic scheme places a matching element in a lower index in hope that it will be sought after more often. Placing it at a lower index would accelerate the search. The pessimistic scheme implements the reverse logic. Typical heuristic step values are –1 and 1 for the optimistic scheme and the pessimistic scheme, respectively. Increasing the magnitude of the heuristic step accelerates moving repeated matching elements towards the front or the back of the array.	1E-6 9 –1 1.5 hSearch X<0? Gto notFound Performs a heuristic search in the memory registers for the element that stores 1.5, starting with the 10^{th} element. The heuristic step is –1, which enforces an optimistic scheme. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	The command returns the index of the matching element in the memory registers. If the supplied array name is valid but the searched value is not found, the command returns –1. If the supplied array name does not exist in the collection of array names, the command returns –2. Because a successful search may well move the matching element, the command returns both the new and old indices of the matching element in the stack: T: Z: Y: New index of matching element. X: Old index of matching element. If the X register contains –1 (because the search did not find a matching element), the Y register will also contain –1.	

Copyright © 2015, 2016 by Namir Clement Shammas

HSEARCHA	Perform a heuristic search for a value in an array variable. The command may include the name of the array. If this argument is omitted, the command uses the Alpha register to specify the array variable to search. The Stack provides the following search parameters: T: Tolerance value. Z: Index of first array variable element to search. Y: Heuristic step (a positive or negative integer) X: The search value. If the index of the memory register to search is negative, the command performs a backward search from the last array element to the absolute value of the Z register. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values. The heuristic step value allows you to move the matching element up (pessimistic scheme) using a positive value, or down (optimistic scheme) using a negative value. The optimistic scheme places a matching element in a lower index in hope that it will be sought after more often. Placing it at a lower index would accelerate the search. The pessimistic scheme implements the reverse logic. Typical heuristic step values are –1 and 1 for the optimistic scheme and the pessimistic scheme, respectively.	‘Xarr 1E-6 9 –1 1.5 hsearcha X<0? Gto notFound Performs a heuristic search in the array Xarr for the element that stores 1.5, starting with the 10th element. The heuristic step is –1, which enforces an optimistic scheme. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	Increasing the magnitude of the heuristic step accelerates moving repeated matching elements towards the front or the back of the array. The command returns the index of the matching element in the array variable. If the supplied array name is valid but the searched value is not found, the command returns –1. If the supplied array name does not exist in the collection of array names, the command returns –2. Because a successful search may well move the matching element, the command returns both the new and old indices of the matching element in the stack: T: Z: Y: New index of matching element. X: Old index of matching element. If the X register contains –1 (because the search did not find a matching element), the Y register will also contain –1.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
INTEG (version 1)	Integrate the RPN expression in the Alpha register. The stack provides the following input: T: Z: The tolerance value for the result. Y: The lower limit for the integral. X: The upper limit for the integral. This command uses the Romberg integration method. The command returns the value of the integral in the X stack register.	‘1/X 1E-6 1 2 INTEG Will integrate 1/X from X=1 to X=2 using a tolerance value of 1E-6.
INTEG (version 2)	Integrate the function defined by a label in the Alpha register. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the function to be integrated. The stack provides the following input: T: Z: The tolerance value for the result. Y: The lower limit for the integral. X: The upper limit for the integral. This command uses the Romberg integration method. The command returns the value of the integral in the X stack register.	‘gsb Recipocal 1E-6 1 2 INTEG Will integrate the function defined by the label Reciprocal, from X=1 to X=2 using a tolerance value of 1E-6.
IS_BETWEEN	Test if the value in the X stack register is in the range defined by the values in the Y and Z stack registers. This command uses the following inequality: $Y \leq X \leq Z$ If the test fails, the program skips a line.	1 .5 rand Is_between Goto there Tests if the random number generated is between 0.5 and 1.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
IS_OUTSIDE	Test if the value in the X stack register is outside the range defined by the values in the Y and Z stack registers. This command uses the following inequalities: $$Y \leq X \ Or \ X \geq Z$$ If the test fails, the program skips a line.	0.75 .25 rand Is_outside Goto there Tests if the random number generated is outside the range [0.25, 0.75].
IS_WITHIN	Test if the value in the X stack register is *strictly inside* the range defined by the values in the Y and Z stack registers. This command uses the following inequality: $$Y < X < Z$$ If the test fails, the program skips a line.	1 .5 rand Is_within Goto there Tests if the random number generated is between 0.49999 and 0.99999.
IS_WITHOUT	Test if the value in the X stack register is *strictly outside* the range defined by the values in the Y and Z stack registers. This command uses the following inequalities: $$Y < X \ Or \ X > Z$$ If the test fails, the program skips a line.	1 .5 rand Is_without Goto there Tests if the random number generated is strictly outside the range [0.5, 1].
ISEVEN	Test if the integer part of the X register is an even value. If not, the program skips a program line.	2 Iseven Gto there Jumps to label “there” if the integer value in the X stack register is even.
ISG, ISGIND	Implement the *Increment and Skip if Greater*, somewhat like in the HP–41C calculator. The ISGIND uses indirect addressing.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
ISODD	Test if the integer part of the X register is an odd value. If not, the program skips a program line.	2 Isodd Gto there Jumps to label “there” if the integer value in the X stack register is odd.
ISPRIME	Test if the integer part of the value in the X register is a prime number. If the test fails, the program skips a program line.	5 Isprime gto 90 * 1 +
IX–>A, IY–>A, IZ–>A, IT–>A	Same as ARCLIX, ARCLIY, ARCLIZ, and ARCLIT.	
JUMP	Jump forward or backward by a specified number of program lines.	X<>0? Jump 3 If the value in the X register is 0, jump 3 steps.
JUMPIND	Perform an indirect jump forward or backward. This command obtains the number of lines to jump from the value of a specified memory register. This command allows you to jump forward or backward using positive and negative values, respectively.	X<>0? Jumpind 3 If the value in the X register is 0, jump the number of steps specified by Mem(3).
JUMPX, JUMPY, JUMPZ, and JUMPT	Jump by the number of program lines specified by the integer value in the corresponding stack register. This command allows you to jump forward or backward using positive and negative values, respectively.	X<>0? Jumpy If the value in the X register is 0, jump the number of steps specified by the integer value of the Y stack register.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
LASTPOS	Search for text in the Alpha register for the last occurrence of a substring. The command requires the name of a text variable that contains the searched substring. The command returns the position of the matching string, or –1 if no match is found or if the search string is empty.	‘456 ASTO str1 ‘123456789 lastpos str1 Returns 3 in the X register.
LBL	Define a label which flags how to direct the GTO, GSB, and other flow–controlling commands.	LBL Start lbl 0 Lbl 2
LCASE	Convert the characters of the Alpha register to lowercase.	‘Hello lcase Sets the Alpha register to “hello”.
LEFT	Replace the content of the Alpha register with the N leading characters. N is the integer value in the stack register X.	‘1234567 3 left Sets the Alpha register to “123”.
LOGXLR and LOGYLR	Similar to command POWERFIT where LOGXLR applies the logarithmic model: Y= a + b ln(X) And LOGYLR applies the exponential model: ln(Y) = a + b X The X register value specifies the first register in the block read using READDATA2.	10 LOGXLR Performs a logarithm regression on the data block that was read starting in Mem(10) using READDATA2

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
LR	Perform a linearized regression using a block of data read using the READDATA2 command. The stack provides the following data: T: Z: Index for first register in the block read using READDATA2. Y: Power for Y values. X: Power for X values. The program returns the following values: T: Z: The coefficient of determination. Y: The intercept. X: The slope. Note: The program uses the real–value power indices to raise the observed values to these indices. In case a power index is 0, the program applies the natural logarithm.	10 2 –1 LR Performs a linear regression on the data block that was read starting in Mem(10) using READDATA2. The model used is: $Y^2 = a + b/X$, since the Y register contains 2 as the power of Y, and the X register contains –1 as the power of X.
LRCL, LRCL+, LRCL–, LRCL*, LRCL/	Recall the value in a subroutine's local memory register. The command set also supports register arithmetic.	GSB Fx … LBL Fx LRCL 0 … RTN Recalls the value in the local memory register with the index 0.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
LSTO, LSTO+, LSTO–, LSTO*, ARTSO/	Store the value of the X stack register in a subroutine's local memory register. The command set also supports register arithmetic.	GSB Fx … LBL Fx LSTO 0 … RTN Store the value of the X stack register in the local memory register with the index 0.
MEM2ARVAR	Copy values from the memory registers to an array variable. This command starts copying to a specified element of the array variable. The command can specify the array variable or use the contents of the Alpha register to obtain the array variable. The stack contains the following parameters: T: Z: The index of the first array variable element that is copied into. Y: The index of the last memory register that is copied from. X: The index of the first memory register that is copied from. The command does it's best to comply with the number and location of the copied data. The command stores the number of actual values copied in the X register.	'Xarr 0 22 0 MEM2ARVAR Copies values from the memory registers Mem(0) to Mem(22) into the array variable Xarr, starting at Xarr(0).
MERGESTAT	Merge the statistical registers' values that are stored in a file with the current values in the statistical registers. The command can include the input filename. If not, the command uses the contents of the Alpha register as the input filename.	'Data.txt mergestat Reads the statistical registers from file Data.txt and merges them with the statistical register in memory.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
MID	Replace the content of the Alpha register with the N trailing characters, starting at the Ith original character. N and I are the integer values of the X and Y registers, respectively.	‘1234567890 1 4 Mid Sets the Alpha register to “2345”.
MLR	Perform a linearized regression using a block of data read using the READDATA3 command. The stack provides the following data: T: Index for first register in the block read using READDATA3. Z: Power for Z values. Y: Power for Y values. X: Power for X values. The program returns the following values in the stack registers: T: The coefficient of determination. Z: The intercept. Y: The slope for Y. X: The slope for X. Note: The program uses the real–value power indices to raise the observed values to these indices. In case a power index is 0, the program applies the natural logarithm.	10 0 –1 1 MLR Performs a multiple linear regression on the data block that was read starting in Mem(10) using READDATA3. The model used is: Ln(Z)=a + bX + c/Y Since the Z, Y, and X registers contain 0, –1, and 1 as the powers for variables Z, Y, and X, respectively.

Copyright © 2015, 2016 by Namir Clement Shammas

MSEARCH	Search for a value in the memory registers, using a middle–first search scheme. The search begins with the middle element and alternates in searching elements above and below the middle element. The Stack provides the following search parameters: T: Z: Search mode. Y: Tolerance value. X: The search value. The search mode instructs the command how to compare the search element with the memory registers: 1. Positive to search for the first memory register with a value that is greater than the search value. 2. Zero to search the first memory register with a value that is equal (within the specified tolerance value) to the search value. 3. Negative to search for the first memory register with a value that is less than the search value. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values. The command returns the index of the matching element in the memory register. If the searched value is not found, the command returns –1.	0 1E-6 1.5 mSearch X<0? Gto notFound Searches the memory registers for the element that stores 1.5, starting with the middle element. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

MSEARCHA	Search for a value in an array variable. The command performs a middle–first search. The search begins with the middle element and alternates in searching elements above and below the middle element. The command may include the name of the array. If this argument is omitted, the command uses the Alpha register to specify the array variable to search. The Stack provides the following search parameters: T: Z: Search mode. Y: Tolerance value. X: The search value. The search mode instructs the command how to compare the search value with the array variable: 1. Positive to search for the first array variable element with a value that is greater than the search value. 2. Zero to search the first array element with a value that is equal (within the specified tolerance value) to the search value. 3. Negative to search for the first array element with a value that is less than the search value. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values.	‘Xarr 0 1E-6 1.5 msearcha X<0? Gto notFound Searches the array Xarr for the element that stores 1.5, starting with the middle element. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	The command returns the index of the matching element in the array variable. If the supplied array name is valid but the searched value is not found, the command returns –1. If the supplied array name does not exist in the collection of array names, the command returns –2.	

Copyright © 2015, 2016 by Namir Clement Shammas

NORMDATA2	Normalizes the data in memory registers read using command READDATA2. These blocks contain sets of (X, Y) data. The NORMDATA2 command normalizes the values in variables X and Y by using these formulae: $X_i = 1 + (X_i - X_{min}) / (X_{max} - X_{min})$ $Y_i = 1 + (Y_i - Y_{min}) / (Y_{max} - Y_{min})$ The command also allows you to first transform the X, and/or Y values into their logarithm values. This kind of transformation may help if the values vary in multiple orders of magnitude. The transformation into logarithmic values are applied only if the data are positive! The stack provides the following input: T: Z: The index of the start of the memory block. Y: Numeric flag for requesting logarithmic transformations for Y values. A positive value signals a request for such transformation. X: Numeric flag for requesting logarithmic transformations for X values. A positive value signals a request for such transformation. If you request to transform a variable using logarithmic values, the command checks if all of its values are positive. If they are not all positive, the command ignores the request for such transformation. The command displays	100 0 0 NORMDATA2 The above code normalizes the (X, Y) data in the memory block starting at index 100. The command will transform the data WITHOUT applying logarithmic transformations.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	a warning message that the request cannot be fulfilled. If either or both variables X or Y have an array with the same value in all of elements, then command will generate a runtime error. The command returns an index value, in the X stack register, that indicates the status of the command's operation: 1. A value of 2 indicates complete success. 2. A value of 1 indicates that at least one request for logarithmic transformation has failed. 3. A value of zero signals the at least one variable could not be normalized.	

Copyright © 2015, 2016 by Namir Clement Shammas

NORMDATA3	Normalizes the data in memory registers read using command READDATA3. These blocks contain sets of (X, Y, Z) data. The NORMDATA3 command normalizes the values in variables X, Y and Z by using these formulae: $X_i = 1 + (X_i - X_{min}) / (X_{max} - X_{min})$ $Y_i = 1 + (Y_i - Y_{min}) / (Y_{max} - Y_{min})$ $Z_i = 1 + (Z_i - Z_{min}) / (Z_{max} - Z_{min})$ The command also allows you to first transform the X, Y and/or Z values into their logarithm values. This kind of transformation may help if the values vary in multiple orders of magnitude. The transformation into logarithmic values are applied only if the data are positive! The stack provides the following input: T: The index of the start of the memory block. Z: Numeric flag for requesting logarithmic transformations for Z values. A positive value signals a request for such transformation. Y: Numeric flag for requesting logarithmic transformations for Y values. A positive value signals a request for such transformation. X: Numeric flag for requesting logarithmic transformations for X values. A positive value signals a request for such transformation. If you request to transform a variable using logarithmic values, the command	100 0 0 1 NORMDATA3 The above code normalizes the (X, Y, Z) data in the memory block starting at index 100. The command will transform the data for X by applying logarithmic transformations.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	checks if all of its values are positive. If they are not all positive, the command ignores the request for such transformation. The command displays a warning message that the request cannot be fulfilled. If either or all variables X, Y, and/or Z have an array with the same value in all of elements, then command will generate a runtime error. The command returns an index value, in the X stack register, that indicates the status of the command's operation: 1. A value of 2 indicates complete success. 2. A value of 1 indicates that at least one request for logarithmic transformation has failed. 3. A value of zero signals the at least one variable could not be normalized.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
NOSTACK	Suppress displaying the stack when the program ends.	
ONERRGOTO	Set an error handler to jump to a label if an error occurs. I recommend that you set a flag in the error–handling routine to handle to signal an in–error state.	ONERRGOTO HandleErr Instructs the application to jump to label HandeErr in case of a runtime error.
ONERROFF	Display error handling.	
ONERRRESUME	Ignore a runtime error and resume execution with the next line.	
POS	Search for text in the Alpha register for the first occurrence of a substring. The command requires the name of a text variable that contains the searched substring. The command returns the position of the matching string, or –1 if no match is found or if the search string is empty.	‘456 ASTO str1 ‘123456789 pos str1 Returns 3 in the X register.
POWERLR	Perform a power fit between two variables using a block of data read using the READDATA2 command. The X register value specifies the first register in the block read using READDATA2. The program returns the following values in the stack registers: T: Z: The coefficient of determination. Y: The intercept. X: The slope.	10 POWERLR Performs a power fit using the data block read into Mem(10) using READDATA2.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
POWERMLR	A version of the MLR command where the natural logarithm is applied to all of the observations. The model is: Ln(Z)= a + b ln(X) + c ln(Y) The X register value specifies the first register in the block read using READDATA3. The program returns the following values in the stack registers: T: The coefficient of determination. Z: The value for coefficient a. Y: The value for coefficient c. X: The value for coefficient b.	10 POWERMLR Performs a multiple regression power fit using the data block read into Mem(10) using READDATA3.
PROMPT	Display the contents of the Alpha register and then prompts the user to press the Enter key to resume program execution.	'Enter X? Prompt vsto x

Copyright © 2015, 2016 by Namir Clement Shammas

PRTSTK	Display the formatted values of the stack register. The command can include one or more format strings. If no format string is included, the command uses the content of the Alpha register as the format strings. See Appendix A for a summary of the formatting strings used by this command. You can use 1 to 4 comma–delimited format string codes to output formatted stack values: 1. Using a single format string causes the command to apply that format to all stack registers. 2. Using two format strings causes the command to apply the first format to the X register and the second format to all other stack registers. 3. Using three format strings causes the command to apply the first format string to the X register, the second string format to register Y, and the third format string to both the Z and T stack registers. 4. Using four format strings causes the first, second, third, and fourth format strings to be applied to the X, Y, Z, and T registers, respectively. The above scheme uses the approach of “work with what you have” by reapplying the same format string to multiple registers when there are fewer format strings than registers.	rand 1000 * rand 100 * rand 1000 / rand prtstk f5,e5,f3,f5 Displays the stack register, each with its own format.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	If any or all the format strings are incorrect, the command displays the targeted stack register value unformatted. Program execution resumes since the format error is not considered critical.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
PRTX, PRTY, PRTZ, PRTT	Display the value of the corresponding stack register as a formatted number. The command can include a format string. If no format string is included, the command uses the content of the Alpha register as the format string. See Appendix A for a summary of the formatting strings used by this command. If the format string is incorrect, the command displays a warning and displays the targeted register value unformatted. Program execution resumes since the format error is not considered critical.	Pi PRTX F2 'F2 Pi PRTX Both commands display 3.14.
PSE	Pause the program execution for a specified duration in milliseconds. The command may include the pause duration. If not, it obtains the pause duration from the X stack register. If there is an error with obtaining a correct value for the pause duration, the command displays a warning message and resumes program execution.	PSE 3000 Pauses the program for 3 seconds.
QUADFIT	A special version of MLR command that uses data read with the READDATA2 command. The X stack register provides the Index for first register in the block read using READDATA2. The program returns the following values in the stack registers: T: The coefficient of determination. Z: The intercept. Y: The coefficient for the X^2 term. X: The coefficient for the X term.	10 QUADFIT Performs a quadratic fit using the data block read into Mem(10) using READDATA2.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
RCLFLGS	Recall all flags from a named text variable. The command may include the named text variable. If not, the command uses the Alpha register to select the named text variable.	RCLFLGS FlgsSto
READ1VAR, READ2VARS, READ3VARS, READ4VARS	Read the values of one, two, three, and four array variables from a comma–delimited text file. In case of multiple variables, the values MUST BE COMMA–DELIMITED! The command returns the of data rows read. If the filename and/or its path are incorrect, the command returns 0. If any or all of the array variables have not yet been creates, the READ commands will create them for you.	READ1VAR datafile.tx Xvar READ2VARS datafile2.txt Xvar Yvar READ3VARS datafile3.txt Xvar Yvar ZVar
READDATA2	Read a text file containing lines of comma–delimited values of two variables. The Alpha register contains the name of the source file. The X register contains the index for the first memory register where the data block begins. The memory registers are organized as follows: I=Fix(value in X register) Mem(I)=2 Mem(I+1)=number of observations Mem(I+2)=X(1) Mem(I+3)=Y(1) Mem(I+4)=X(2) Mem(I+5)=Y(2) … The command places, in the X register, the number of lines read.	'temp_readings.txt 10 Readdata2 Reads a block of say 20 observations and storing the results in: Mem(10)=2 Mem(11)=20 Mem(12)=X(1) Mem(13)=Y(1) ….

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
READDATA3	Read a text file containing lines of comma–delimited values of three variables. The Alpha register contains the name of the source file. The X register contains the index for the first memory register where the data block begins. The memory registers are organized as follows: I=Fix(value in X register) Mem(I)=3 Mem(I+1)=number of observations Mem(I+2)=X(1) Mem(I+3)=Y(1) Mem(I+4)=Z(1) … The command places, in the X register, the number of lines read.	'temp_readings.txt 10 Readdata3 Reads a block of say 20 observations and storing the results in: Mem(10)=3 Mem(11)=20 Mem(12)=X(1) Mem(13)=Y(1) Mem(14)=Z(1) ….

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
READMEM	Read values for the memory registers from a text file. Each line in that file contains at least a single value and may contain multiple comma–delimited values. The number of comma–delimited values per line can vary in an input file. The command ignores the following items: • Blank lines. • Lines with no commas and pure text or a mix of text and digits. • Comma–delimited items containing text. • Comma–delimited items containing a mix of text and digits, like this1is4forme. The command looks for string images of valid numbers (integers, reals with decimals, or numbers in scientific format) that appear either on an entire line or as a comma–delimited item. The command may include the name of the source file. If not, the command uses the Alpha register to specify the name of the source file. The X register specifies the first memory register that receives data from the source file. The command ignores extra input values that have no place in the memory registers. The command returns in the X stack register the number of values read from the file.	'mydata.txt 1 Readmem Reads data from the file mydata.txt and places the first input value in Mem(1). The command returns the number of values read from the file.
READNVARS	A general version of READ1VAR, READ2VARS, READ3VARS, and READ4VARS allowing you to read more than four variables.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
REM, !, @, #, $, %	Define the start of a commented line.	REM Just a comment % Comment also @ same here!
REPLACE	Replace the contents of the Alpha register. The command specifies two comma–delimited text variables. The first text variable stores the text to search in the Alpha register. The second text variable stores the replacement string.	‘456 Asto s1 ‘abc Asto s2 ‘1234567890 replace s1,s2 Replaces the Alpha register from “1234567890” into “123abc7890”.
RESIZE	Resize an array variable. The command can include the array variable. If not the command uses the Alpha register to specify the array variable. The X register contains the new size. You can expand or contract an array variable. When you shrink an array variable you may lose some of the data in that array.	‘Xarr 100 Resize Resizes the array Xarr to have 100 elements.
REV	Reverse the characters in the Alpha register.	‘12345 rev Sets the Alpha register to “54321”.
RIDX	Set the value for the array variable recall index. The integer value of the X stack register initializes the index value. This command may specify the name of the array. If omitted, the command uses the Alpha register as the name of the array.	‘Xarr 1 Sidx Sets the recall index of array variable XARR to 1.
RIGHT	Replace the content of the Alpha register with the N trailing characters. N is the value in the stack register X.	‘1234567 3 right Sets the Alpha register to “567”.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
RRC+, RRC–, RRC$	Recall an element of a specified array variable using its current recall index. The command may include the array name. If omitted, the command uses the Alpha register as the name of the array. The RRC+ also increments the recall index of the array variable. By contrast, RRC– decrements the recall index. The command RRC$ keeps the recall index unchanged.	See namarr1.txt program example and Figure 21.
RRCL, RRCL+, RRCL–, RRCL*, RRCL/	Recall the value in an element of an array variable. The command set also supports register arithmetic. The command has four possible versions: 1. With no arguments, the command uses the Alpha register to select the name of the array. It also uses the integer value of the X stack register as the index of the recalled element. 2. With just an index. The command uses that index value and uses the Alpha register to select the name of the array. 3. With just the name of the array. The command uses the integer value of the X stack register as the index of the recalled element. 4. With both the name of the targeted array and the targeted index.	1 rrcl Xarr Recalls the value in the second element of array variable Xarr.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
RST+, RST–, RST$	Store the value in the X stack register in an element of a specified array variable using its current storage index. The command may include the array name. If omitted, the command uses the Alpha register as the name of the array. The RST+ also increments the storage index of the array variable. By contrast, RST– decrements the storage index. The command RST$ keeps the storage index unchanged.	See namarr1.txt program example and Figure 21.
RSTO, RSTO+, RSTO–, RSTO*, RTSO/	Store the value in the X register in an element of an array variable. The command set also supports register arithmetic. The command has four possible versions: 1. With no arguments, the command uses the Alpha register to select the name of the array. It also uses the integer value of the Y stack register as the index of the stored element. 2. With just an index. The command uses that index value and uses the Alpha register to select the name of the array. 3. With just the name of the array. The command uses the integer value of the Y stack register as the index of the stored element. 4. With both the name of the targeted array and the targeted index.	Pi rsto Xarr 1 Stores the value of pi in the second element of array variable Xarr.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SAVESTAT	Write the statistical registers to a file. The command can include the output filename. If not, the command uses the contents of the Alpha register as the output filename. The filename must include a full path if it not located in the same directory as that of the CPRCA application.	'C:\MyData\Data.txt Savestat Writes the statistical registers to file Data.txt.
SCAN	Scan a range of values in an RPN expression to locate roots, minima, and maxima. The command can specify the name of a text variable that stores the RPN expression. If this variable is not specified, the command expects the RPN expression to be found in the Alpha register. The command requires six parameters. To achieve this, the command uses the backup stack. To set up the input, first enter the following values in the stack: T: Z: Y: Index of the first memory register that stores the results. X: The value for the tolerance for the function value. Use the COPYSTACK command to copy the above values into the backup stack, and then enter the remaining parameters: T: The tolerance for the roots, minima, and maxima. Z: The step value used to scan the range of values. Y: The start of the scanned range. B: The end of the scanned range.	lbl Start 'exp LastX x^2 3 * – # memIdx 0 # FxToler 1e–7 copystack # Toler 1e–8 # Step 0.1 # from A –1 # to B 4 scan -x- nostack end Displays the roots, minima, and maxima in the range of -1 to 4 for the function e^x-$3x^2$.

Copyright © 2015, 2016 by Namir Clement Shammas

SEARCH	Search for a value in the memory registers. The Stack provides the following search parameters: T: Z: Tolerance value. Y: Index of first memory register to search. X: The search value. The search mode instructs the command how to compare the search value with the memory registers: 1. Positive to search for the first memory register with a value that is greater than the search value. 2. Zero to search the first memory register with a value that is equal (within the specified tolerance value) to the search value. 3. Negative to search for the first memory register with a value that is less than the search value. If the index of the memory register to search is negative, the command performs a backward search from the last memory register to the absolute value of the Y register. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance value to match two floating–point values that are close enough in values. The command returns the index of the matching element in the memory	0 1E-6 9 1.5 Search X<0? Gto notFound Searches the memory registers for the element that stores 1.5, starting with the 10th element. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
	register. If the searched value is not found, the command returns –1.	

Copyright © 2015, 2016 by Namir Clement Shammas

SEARCHA	Search for a value in an array variable. The command may include the name of the array. If this argument is omitted, the command uses the Alpha register to specify the array variable to search. The Stack provides the following search parameters: T: Search mode. Z: Tolerance value. Y: Index of first array variable element to search. X: The search value. The search mode instructs the command how to compare the search value with the array variable element: 1. Positive to search for the first array element with a value that is greater than the search value. 2. Zero to search the first array element with a value that is equal (within the specified tolerance value) to the search value. 3. Negative to search for the first array element with a value that is less than the search value. If the index of the memory register to search is negative, the command performs a backward search from the last array element to the absolute value of the Y register. Since comparing floating–point values is not as exact as comparing integers, the command uses a small tolerance	'Xarr 0 1E-6 9 1.5 Searcha X<0? Gto notFound Searches the array Xarr for the element that stores 1.5, starting with the 10th element. The tolerance for the search is 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
	value to match two floating–point values that are close enough in values. The command returns the index of the matching element in the array variable. If the supplied array name is valid but the searched value is not found, the command returns –1. If the supplied array name does not exist in the collection of array names, the command returns –2.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
SIDX	Set the value for the array variable storage index. The integer value of the X stack register initializes the index value. This command may specify the name of the array. If omitted, the command uses the Alpha register as the name of the array.	‘Xarr 1 ridx Sets the storage index of array variable XARR to 1.
SIGMA+T, S+T	Similar to commands SIGMA++ and S++ except it uses the Alpha register as an RPN expression used to transform the values for X and/or Y. The expression can use commands like X<>Y to access the values. This command does not clear the statistical registers before adding the data. You are responsible for initializing the statistical registers. This feature allows you to add transformed values from different areas of the memory registers using multiple calls to this command.	‘ln X<>Y ln X<>Y 1.0112 S+T Adds the data in memory registers Mem(1) to Mem(112) as pairs of (X, Y) values to the statistical registers. The RPN Expression in the Alpha register tells the command to transform the original (X, Y) values into (ln(X), ln(Y)) values.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
SOLVE (version 1)	Solve the root of the RPN expression in the Alpha register. The stack provides the following parameters for seeking the root: T: Z: The tolerance value. Y: The maximum number of iterations. Z: The initial guess for the root. This command uses Newton's method. The command places the following results in the stack registers: T: The last difference in the refined guess for the root. Z: The function value at the best refined guess for the root. Y: The number of iterations. X: The best refined guess for the root. If the solution fails to converge, the command SOLVE displays a warning message for a visual notification. Programmatically, you can determine convergence by comparing the number of iterations (in the Y register) with the maximum number of iterations that you specified when invoking command SOLVE. This comparison will let you know if the solution converged (when the number of iterations does not exceed the maximum limit) or diverged (when the number of iterations exceeds the maximum limit).	'EXP LASTX X^2 3 * – 1E-6 55 4.1 SOLVE Will solve for the root of f(x)=exp(x)–3*x^2, using an initial guess of 4.1, 55 maximum iterations, and a tolerance value of 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SOLVE (version 2)	Solve the root of a function that is defined by a program label. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the nonlinear function. The stack provides the following parameters for seeking the root: T: Z: The tolerance value. Y: The maximum number of iterations. Z: The initial guess for the root. This command uses Newton's method. The command places the following results in the stack registers: T: The last difference in the refined guess for the root. Z: The function value at the best refined guess for the root. Y: The number of iterations. X: The best refined guess for the root. If the solution fails to converge, the command SOLVE displays a warning message for a visual notification. Programmatically, you can determine convergence by comparing the number of iterations (in the Y register) with the maximum number of iterations that you specified when invoking command SOLVE. This comparison will let you know if the solution converged (when the number of iterations does not exceed the maximum limit) or diverged (when the number of iterations exceeds the maximum limit).	'gsb myfx 1E-6 55 4.1 SOLVE Will solve for the root of f(x) defined by label myfx, using an initial guess of 4.1, 55 maximum iterations, and a tolerance value of 1E-6.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SOLVEBIN (version 1)	Solve the root of the RPN expression in the Alpha register. The stack provides the following parameters for seeking the root: T: Z: The tolerance value. Y: The guess A. Z: The guess B. This command uses the Bisection method. The command places the following results in the stack registers: T: Z: Y: The function value at the best refined guess for the root. X: The best refined guess for the root. The values of the function at A and B must have opposite signs. If not, the command displays a warning message and returns a large number.	‘exp lastx x^2 3 * 1 1e-7 3 4 Solvebin Finds the root of e^x-3x^2 between x= 3 and x=4, using a tolerance of 1e-7.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SOLVEBIN (version 2)	Solve the root of a function that is defined by a program label. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the nonlinear function. The stack provides the following parameters for seeking the root: T: Z: The tolerance value. Y: The guess A. Z: The guess B. This command uses the Bisection method. The command places the following results in the stack registers: T: Z: Y: The function value at the best refined guess for the root. X: The best refined guess for the root. The values of the function at A and B must have opposite signs. If not, the command displays a warning message and returns a large number.	Lbl start 'gsb fx 1e-7 3 4 solvebin end Lbl fx exp lastx x^2 3 * - Rtn Finds the root of e^x-3x^2 between x= 3 and x=4, using a tolerance of 1e-7.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SOLVEHAL (version 1)	Solve the root of the RPN expression in the Alpha register, using Halley's algorithm. The stack provides the following parameters for seeking the root: T: Z: The tolerance value. Y: The maximum number of iterations. Z: The initial guess for the root. This command uses Halley's method. The command places the following results in the stack registers: T: The last difference in the refined guess for the root. Z: The function value at the best refined guess for the root. Y: The number of iterations. X: The best refined guess for the root. If the solution fails to converge, the command SOLVEHAL displays a warning message for a visual notification. Programmatically, you can determine convergence by comparing the number of iterations (in the Y register) with the maximum number of iterations that you specified when invoking command SOLVEHAL. This comparison will let you know if the solution converged (when the number of iterations does not exceed the maximum limit) or diverged (when the number of iterations exceeds the maximum limit).	exp lastx x^2 3 * 1 1e-7 100 4 solvehal Finds the root of e^x-3x^2 near x=4, using a tolerance of 1e-7 and a maximum of 100 iterations.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
SOLVEHAL (version 2)	Solve the root of a function that is defined by a program label, using Halley's algorithm. The Alpha register must contain the keyword GSB, followed by a space, followed by the name of the label that implements the nonlinear function. The stack provides the following parameters for seeking the root: T: Z: The tolerance value. Y: The maximum number of iterations. Z: The initial guess for the root. This command uses Halley's method. The command places the following results in the stack registers: T: The last difference in the refined guess for the root. Z: The function value at the best refined guess for the root. Y: The number of iterations. X: The best refined guess for the root. If the solution fails to converge, the command SOLVEHAL displays a warning message for a visual notification. Programmatically, you can determine convergence by comparing the number of iterations (in the Y register) with the maximum number of iterations that you specified when invoking command SOLVEHAL. This comparison will let you know if the solution converged (when the number of iterations does not exceed the maximum limit) or diverged (when the number of iterations exceeds the maximum limit).	lbl start 'gsb fx 1e-7 100 4 solvehal end lbl fx exp lastx x^2 3 * - rtn Finds the root of e^x-3x^2 near x=4, using a tolerance of 1e-7 and a maximum of 100 iterations.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SORTA	Sort the data in the memory register in an ascending order. This command treats the memory registers as a virtual table or matrix. The stack registers provide the data for the sorting operation: T: The index of the first memory register to sort. Z: The number of rows. Y: The number of columns. X: The index of the column used to sort the data. This command treats the memory registers as a virtual table where the first element has the row and column indices of 1.	100 10 5 2 sorta Sorts the memory registers starting at Mem(100). The command sorts the data as having 10 rows and 5 columns. Column 2 is used to sort the values of the virtual table.
SORTARA	Sort all of the elements of an array variable in an ascending order. The arguments for this command are the name of the array. If there is no argument, the command uses the contents of the Alpha register as the name of the array.	Sortara Xarr Sorts the elements of the array variable Xarr in an ascending order.
SORTARD	Sort the all of elements of an array variable in a descending order. The arguments for this command are the name of the array. If there is no argument, the command uses the contents of the Alpha register as the name of the array.	Sortard Xarr Sorts the elements of the array variable Xarr in a descending order.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
SORTD	Sort the data in the memory registers in a descending order. This command treats the memory registers as a virtual table or matrix. The stack registers provide the data for the sorting operation: T: The index of the first memory register to sort. Z: The number of rows. Y: The number of columns. X: The index of the column used to sort the data. This command treats the memory registers as a virtual table where the first element has the row and column indices of 1.	0 10 5 2 Sortd Sorts the memory registers starting at Mem(0). The command sorts the data as having 10 rows and 5 columns. Column 2 is used to sort the values of the virtual table.
–STK–	Show the stack.	
STOFLGS	Store all the flags in a named text variable. The command may include the named text variable. If not, the command uses the Alpha register to select the named text variable.	STOFLGS FlgsSto
STOP and R/S	Display the stack and temporarily halt the program execution. You can optionally type in a number and press Enter, or just press Enter. The application pushes in the stack the (valid) number that you type in.	
SWAPCASE	Swap the character case of the text in the Alpha register.	‘Hello There! Swapcase Makes the Alpha register contain “hELLO tHERE!”

Copyright © 2015, 2016 by Namir Clement Shammas

Command	*Purpose*	*Example*
T?=nn, T?<>nn, T?!=nn, T?>nn, T?>=nn, T?<nn, T?>nn, T?<=nn	Compare the value in the T register with a value in a memory register.	T?<>03 Tests if the value in the T register differs from the value in memory register Mem(3).
TRIM	Trim the leading and trailing spaces in the Alpha register.	‘ 123 trim Sets the Alpha register to “123”.
UCASE	Convert the characters of the Alpha register to uppercase.	‘hello ucase Sets the Alpha register to “HELLO”.

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
VCA+, VCA–, VCA*, VCA/	Add, subtract, multiply, and divide the individual values of two array variables. The command requires two space–delimited array names. The command adds the individual values of the second array to those of the first one. The X stack registers provide the contains the following data: T: Z: The number of elements to process. Y: The index of the first element in the second array (i.e., the target array). X: The index of the first element in the first array (i.e., the source array). The command VCA/ handles division–by–zero by placing a very large number in the array element. Each of the VCAx command returns the number of elements that were actually processed. These commands work with elements within valid ranges and the requested number of elements to process.	Arcopy xarr1,xarr3 10 0 0 Vca+ xarr3 xarr2 First copies the values of array xarr1 into array xarr3. Then adds the 10 values of array xarr2 to array xarr3. The net effect is the following vector operation: **Xarr3(0..9)** = **Xarr1(0..9)**+**Xarr2(0..9)**

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
VCAFX	Evaluate the RPN expression in the Alpha register for a range of elements in an array variable. The command includes the name of the array. The X stack registers provide the contains the following data: T: Z: Y: The number of elements to process. X: The index of the first element in the array that will have its value altered. The VCAFX command returns the number of elements that were actually processed. The command works with elements within valid ranges and the requested number of elements to process.	'1/x 10 0 Vcafx xarr Applies the RPN expression of 1/x to the elements Xarr(0) to Xarr(9).

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
VCAS+, VCAS–, VCASA–, VCAS*, VCAS/, VCASA/	Add, subtract, multiply, and divide a scalar value to the individual values of an array variable. The command must contain the name of the array as an argument. The X stack registers provide the contains the following data: T: Z: The number of elements to process. Y: The index of the first element in the array. X: The scalar value. Notice that since subtraction and division are not communicative operations, the application offers two commands for subtraction and two for division. The VCAS– command performs Vect(i)=Vect(i)–X. The VCASA– command performs Vect(i)=X–Vect(i). The VCAS/ command performs Vect(i)=Vect(i)/X. The VCASA/ command performs Vect(i)=X/Vect(i). The commands VCAS/ and VCASA/ handle division–by–zero by placing a very large number in the array element. Each of the VCASx command returns the number of elements that were actually processed. These commands work with elements within valid ranges and the requested number of elements to process.	10 0 100 Vcas* Xarr Multiplies 10 elements in the array variable Xarr by 100. The elements affected are Xarr(0) to Xarr(9).

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
VIEWDT	View the date text that is stored in a text variable (minus the leading and trailing # characters). The name of the variable is the argument for this command.	'#1/1/2016 2:00:00 PM#" asto dt1 viewdt dt1 Displays the date in text variable dt1, which is "1/1/2016 2:00:00 PM".
VIEWMEM	Display the value in the designated memory register.	Viewmem 03 Displays the value in Mem(3).
VIEWREGS	Display the values the memory registers. The value of the X registers contains the value aaaa.bbbb, where aaaa and bbbb are indices that define the first and last memory registers to view, respectively.	0.0010 Viewregs Views the value in memory registers Mem(0) to Mem(10).
VIEWSTK	Display the values in the stack.	
VIEWVAR	Display the value is a named register. The command can include the name of the variable. If not, it uses the Alpha register to specify the named register.	VIEWVAR Guess Displays the value in the variable named "Guess".
VIEWX, VIEWY, VIEWZ, VIEWT	Display the value in the designated stack register	VIEWX
VRCL, VRCL+, VRCL–, VRCL*, VRCL/	Recall the value in a named register. The command set offers options to also perform basic math operations with the X register. The command can include the named register. If not, it uses the Alpha register to specify the named register.	Vrcl piVal
VSTO, VSTO+, VSTO–, VSTO*, VSTO/	Store the value of the X register in a named register. The command set offers options to also perform basic math operations with the targeted named variable register. The command can include the named register. If not, it uses the Alpha register to specify the named register.	Pi Vsto piVal

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
WRITE1VAR, WRITE2VARS, WRITE3VARS, WRITE4VARS	Write the values of one, two, three, and four array variables into a comma–delimited text file. The command returns the of data rows written.	WRITE1VAR datafile.tx Xvar WRITE2VARS datafile2.txt Xvar Yvar WRITE3VARS datafile3.txt Xvar Yvar ZVar
WRITEMEM	Write values of the memory registers to a data file. The command may include the name of the target file. If not, the command uses the Alpha register to specify the name of the target file. The stack provides the following information: T: Z: The number of comma–delimited values per line. Y: The number of memory registers to save to the output file. X: The index of the first memory register to copy. The command ignores request to write memory registers beyond what is available (i.e. index 9999). The command returns the number of written memory registers in the X stack register.	'data.txt 5 100 1 Writemem Writes 100 values in the memory registers to the text file data1.txt. The output start with Mem(1) and ends with Mem(100). The command writes each five values per line.
WRITENVARS	A general version of WRITE1VAR, WRITE2VARS, WRITE3VARS, WRITE4VARS allowing you to write more than four variables.	
–X–	Same as VIEWX.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
X?=nn, X?<>nn, X?!=nn, X?>nn, X?>=nn, X?<nn, X?>nn, X?<=nn	Compare the value in the X register with a value in a memory register.	X?<>03 Tests if the value in the X register differs from the value in memory register Mem(3).
X=0?, X<>0?, X!=0?, X>0?, X>=0?, X<0?, X<=0?	Compare the value in the X with 0. If the test is true, the program executes the next line. If not, the program skips the next line.	X>0? Gto There * +
X=0?n, X<>0?n, X!=0?n, X>0?n, X>=0?n, X<0?n, X<=0?n	Compare the value in the X with 0. If the test is true, the program executes the next line. If not, the program skips the next n lines. The value of n is in the range of 2 to 9. Including the number of program lines to skip if the test fails enhances the readability of the command.	X>0?2 * + Lbl 0 Jumps to LBL 0 if the X stack register does not contain a positive value.
X=Y?, X<>Y?, X!=Y?, X>Y?, X>=Y?, X<Y?, X<=Y?	Compare the values in the X and Y registers. If the test is true, the program executes the next line. If not, the program skips the next line.	X=Y? GTO 12 / +
X=Y?n, X<>Y?n, X!=Y?n, X>Y?n, X>=Y?n, X<Y?n, X<=Y?n	Compare the values in the X and Y registers. If the test is true, the program executes the next line. If not, the program skips the next n lines. The value of n is in the range of 2 to 9. Including the number of program lines to skip if the test fails enhances the readability of the command.	X=Y?2 / + Lbl 0 Jumps to LBL 0 if the values in the X and Y stack registers are not equal.
X–>A, Y–>A, Z–>A, T–>A	Same as ARCLX, ARCLY, ARCLZ, and ARCLT.	

Copyright © 2015, 2016 by Namir Clement Shammas

Command	Purpose	Example
XEQ	Perform one of the following tasks: 1. Execute a set of simple RPN commands stored in the Alpha register. This command uses the current contents of the Alpha register when you don't specify any argument. 2. Execute a set of simple RPN commands stored in a named string register	'355 113 / ASTO fx1 Xeq fx1 The above commands evaluate an approximation for pi store in the string variable fx1.
Y?=nn, Y?<>nn, Y?!=nn, Y?>nn, Y?>=nn, Y?<nn, Y?>nn, Y?<=nn	Compare the value in the Y register with a value in a memory register.	Y?<>03 Tests if the value in the Y register differs from the value in memory register Mem(3).
Z?=nn, Z?<>nn, Z?!=nn, Z?>nn, Z?>=nn, Z?<nn, Z?>nn, Z?<=nn	Compare the value in the Z register with a value in a memory register.	Z?<>03 Tests if the value in the Z register differs from the value in memory register Mem(3).

Table 3. Commands used for programming and advanced functions.

Sample Programs

This section presents a few short sample programs. You will notice that the code is case insensitive. You can type the names of commands, labels, and variables in upper, lower, or mixed case. Only the text (starts with single quotes) aimed for the Alpha register is case sensitive. The application internally translates these names into uppercase.
To avoid including the full (and most likely long) path name when invoking the program names, place the files for the programs in the same folder as the application's executable file. You can alternatively place the program files in a short path such as C:\MyCode.

Sum of Reciprocal Power

This program calculates the sum of $1/X^2$ for X from 1 to 1000. The code uses memory registers to store the values. The program displays the result using an AVIEW command. The name of the file containing the program is sum001.txt. You are welcome to edit the program and change the upper limit for the summation, 1000, with another value. Figure 3 shows the output of the program.

```
LBL Sum
```

Copyright © 2015, 2016 by Namir Clement Shammas

```
0
sto 1
1000
sto 2
LBL 1
rcl 2
x^2
1/x
sto+ 1
1
sto- 2
rcl 2
x<>0?
gto 1
'Sum = '
rcl 1
ARCLX
AVIEW
Rtn
```

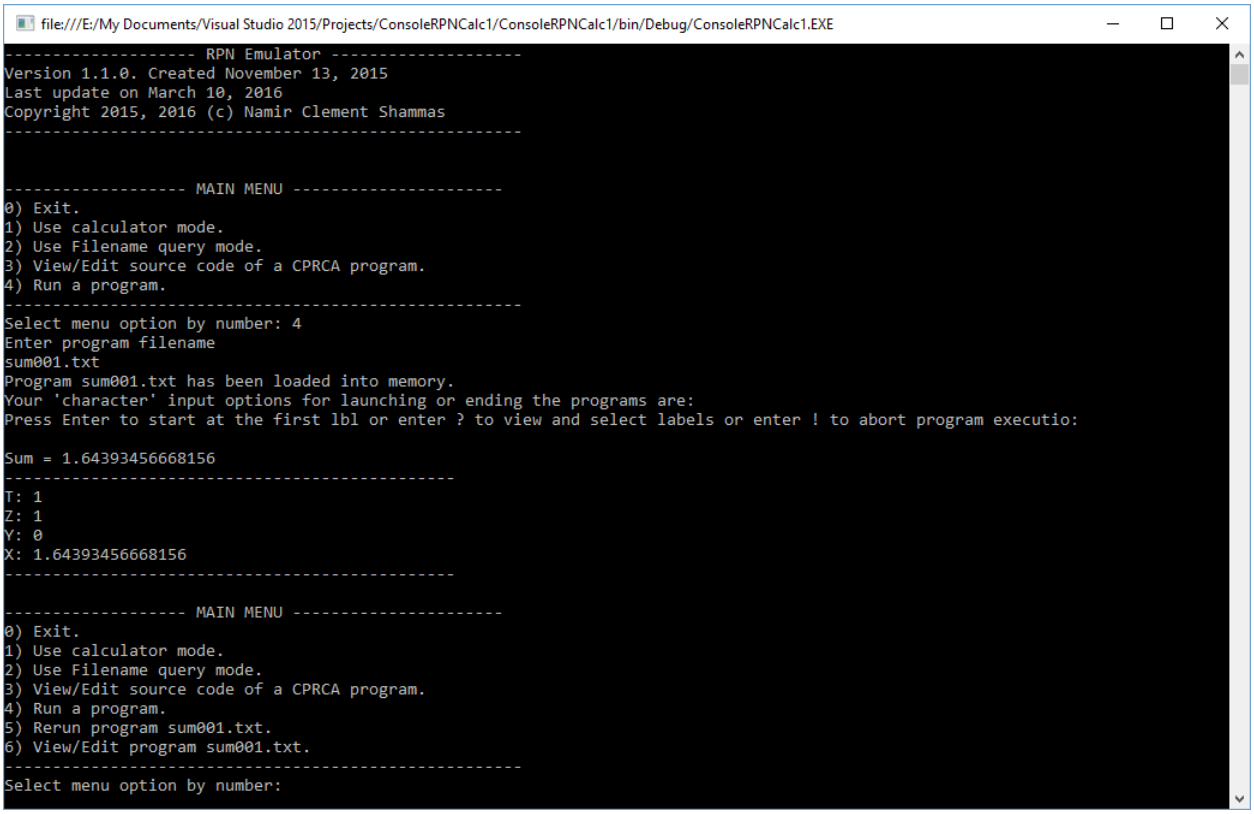


Figure 3. A sample session with the program in file sum001.txt.

Simple Integral of 1/X

This program calculates the integral of 1/X for X from 1 to 2. The program uses memory registers to store the data. The code uses a simple trapezoidal method to calculate the integral of 1/X, which is ln(X). The program is stored in file integ.txt.

```
LBL Integ
1
```

```
sto 1
2
sto 2
.01
sto 3
0
sto 0
rcl 3
2
/
sto+ 1
lbl 0
rcl 1
gsb Fx
sto+ 0
rcl 3
sto+ 1
rcl 2
rcl 1
x<=y?
gto 0
rcl 0
rcl 3
*
end
LBL Fx
1/X
RTN
```

The label Fx contains the program code that calculates the integrated function. You can change the code after that label to integrate other functions between X=1 and X=2. You can also change the values assigned to memory registers 1 and 2 to define a new integration range. Figure 4 shows a sample session with the program in file integ.txt.

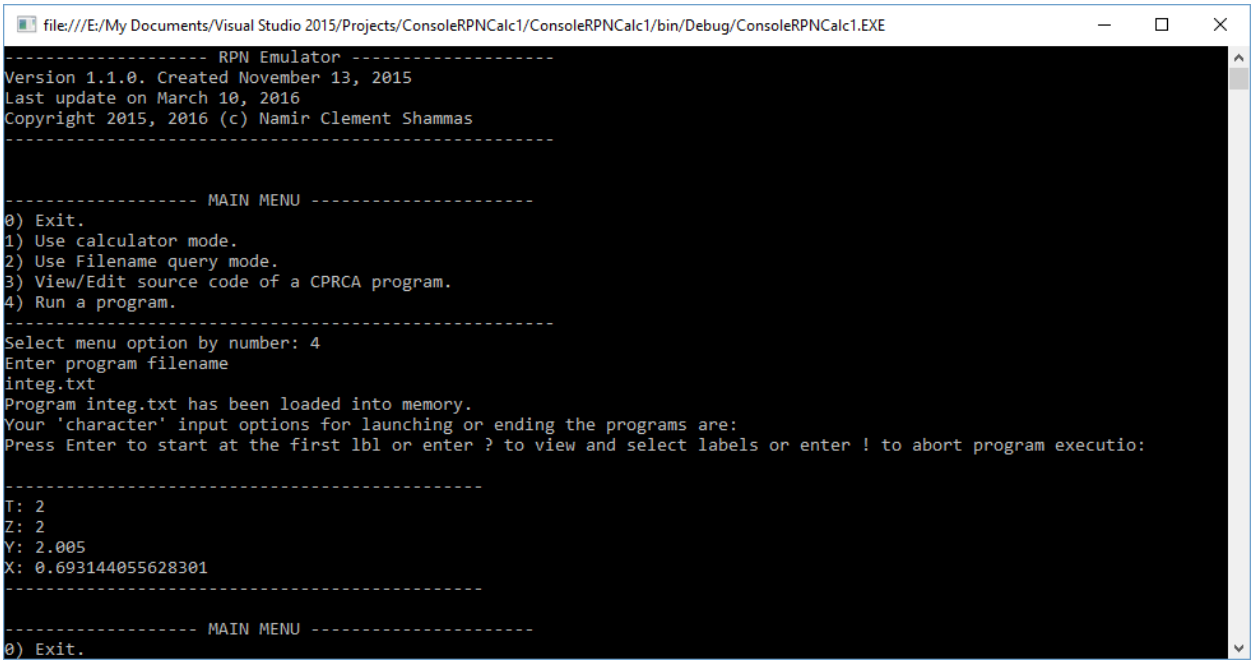


Figure 4. A sample session with the program in file integ.txt.

Simple Integral of 1/X Take Two

This program also calculates the integral of 1/X for X from 1 to 2. The program differs from the previous one in the following ways:

1. Uses named variables.
2. Use the Alpha register to store the expression “1/X” which represents the integrated function. The code uses the XEQ command to execute the RPN expression in a named text register.

The filename containing the program is alpha1.txt.

```
lbl start
'1/X
asto fx
0
vsto sum
1
vsto a
2
vsto b
.001
vsto h
2
/
vsto+ a
lbl 0
vrcl b
vrcl a
x>y?
gto 1
xeq fx
vsto+ sum
vrcl h
vsto+ a
gto 0
lbl 1
vrcl h
vrcl sum
*
rtn
```

To calculate integral for other functions, such as ln(x)/x, replace the string “’ 1/X” after the label start with the string “’ln lastx /”. You can store any other valid RPN expression that represents a custom function. Figure 5 shows a sample session with the program in file alpha1.txt.

Copyright © 2015, 2016 by Namir Clement Shammas

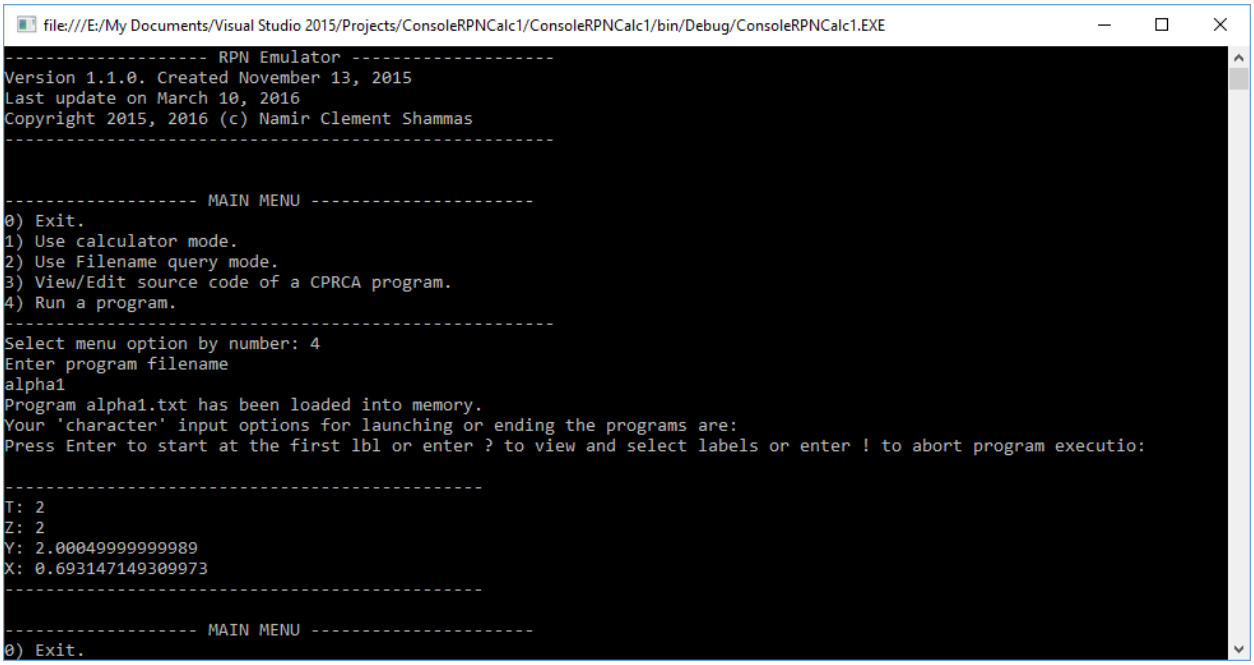


Figure 5. A sample session with the program in file alpha1.txt.

Newton's Method

Here is a short program that implements Newton's method for calculating the root of a function. The code solves the root for $f(x)=e^x - 3x^2$. The program prompts you to enter a guess for the root and a tolerance value. The program displays the current guess for the root in each iteration. The code uses named variables instead of the memory registers. As such, the code is easier to read. The filename containing the program is newton1.txt. The code after label fx calculates the function f(x). Figure 6 shows a sample session with the program in file newton1.txt.

```
# Newton's method version 1
LBL NEWTN
'Enter Guess?
PROMPT
VSTO X
'Enter Tolerance?
prompt
vsto toler
lbl 0
vrcl x
viewvar x
abs
1
+
.001
*
vsto h
vrcl x
gsb fx
vsto f0
vrcl x
```

```
vrcl h
+
gsb fx
vrcl f0
-
1/x
vrcl f0
*
vrcl h
*
vsto- x
abs
vrcl toler
x<y?
gto 0
vrcl x
end
lbl fx
exp
lastx
x^2
3
*
-
Rtn
```

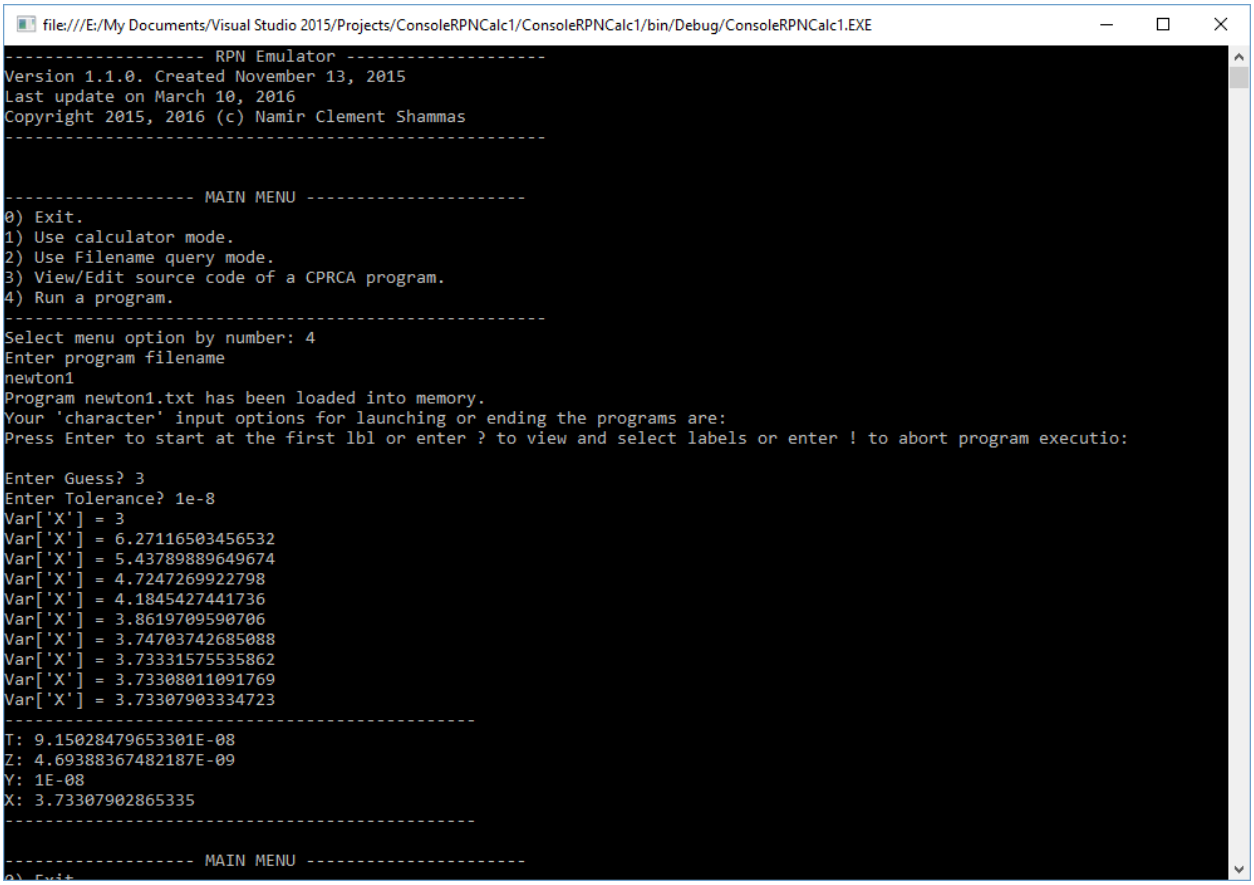


Figure 6. A sample session with the program in file newton1.txt.

Bisection Method

Here is a short program that implements the Bisection method for calculating the root of a function. The code solves the root for f(x)= f(x)=e^x – $3x^2$. The program prompts you to enter the two guesses that surround the root and a tolerance value. The code uses named variables instead of the memory registers. As such, the code is easier to read. The filename containing the program is bisection1.txt. The code after label fx calculates the function f(x). Figure 7 shows a sample session with the program in file bisection1.txt.

```
LBL Bisection
'Enter A?
prompt
vsto a
gsb fx
vsto fa
'Enter B?
prompt
vsto b
gsb fx
vsto fb
'Enter tolerance?
prompt
vsto toler
lbl startLoop
vrcl a
vrcl b
+
2
/
vsto c
gsb fx
vsto fc
vrcl fa
*
x>0?
gto replaceA
vrcl c
vsto b
vrcl fc
vsto fb
gto endLoop
lbl replaceA
vrcl c
vsto a
vrcl fc
vsto fa
lbl endLoop
vrcl a
vrcl b
-
abs
vrcl toler
x<y?
gto startLoop
vrcl a
```

Copyright © 2015, 2016 by Namir Clement Shammas

```
vrcl b
+
2
/
end
lbl fx
exp
lastx
x^2
3
*
-
Rtn
```

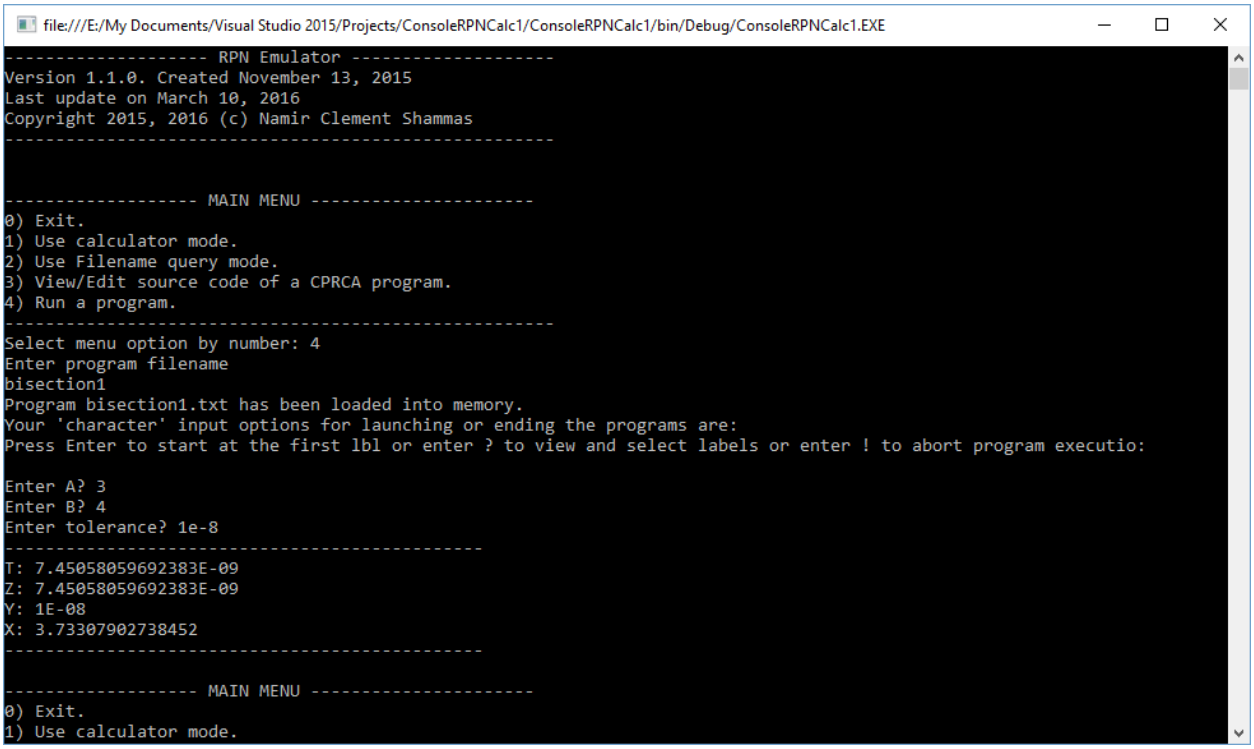


Figure 7. A sample session with the program in file bisection1.txt.

Program with Duplicate Labels

The next program has a duplicate label, preventing it from running. When you load the program, the CPRCA application will let you know that it detected a duplicate label. The application displays the name of the duplicate label and the program line number where the duplicate occurs. The filename containing the program is dupLbl.txt. Figure 8 shows a sample session with the program in file dupLbl.txt.

```
LBL Start
.1
Sto 1
0
sto 0
LBL Start
rcl 1
```

```
1000
*
int
sto+ 0
ISG 1
gto Start
rcl 0
end
```

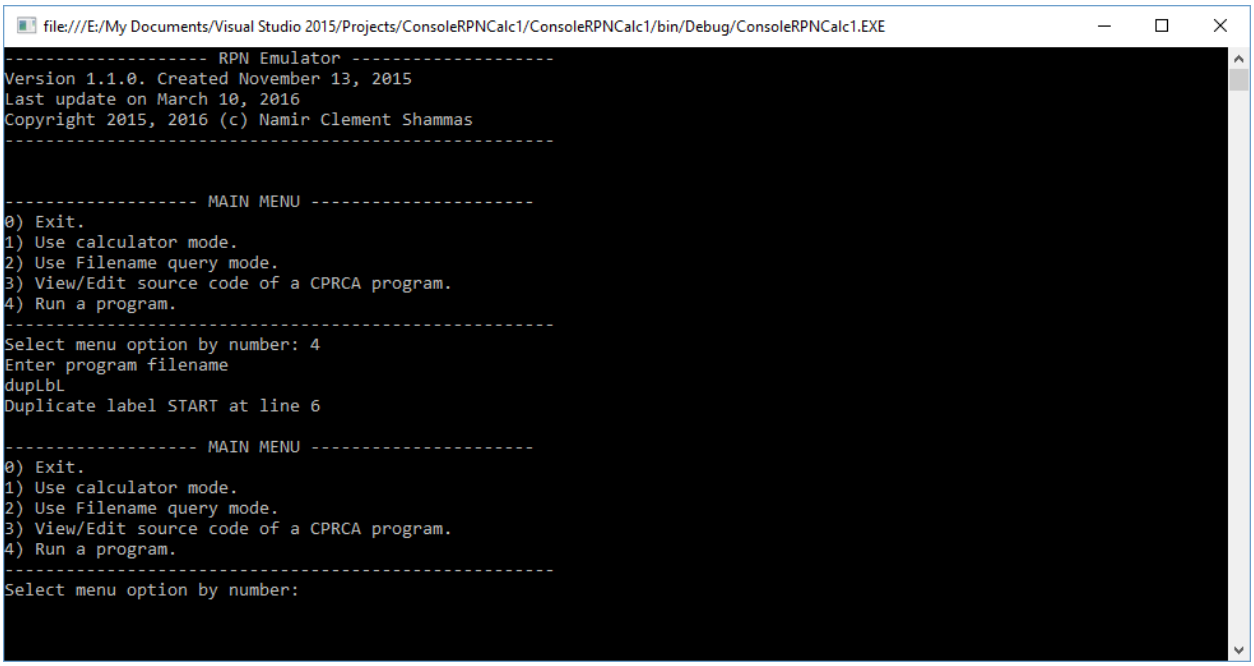


Figure 8. A sample session with the program in file dupLbl.txt.

Simple Linear Regression Program

The next program illustrates reading a data block for two variables, X, and Y, and then performing linear regression using the data in that block. The data file data2v1.txt must be located in the directory C:\MyData. The data file contains the following values:

```
10,50
25,77
30,86
35,95
40,104
100,212
```

The program in file reg2.txt performs a linear regression using the above data.

```
lbl lr
'C:\MyData\data2v1.txt
1
readdata2
1
1
1
lr
rtn
```

The output displays R^2, the intercept, and the slope in stack registers Z, Y, and X, respectively. Figure 9 shows a sample session with the program in file reg2.txt.

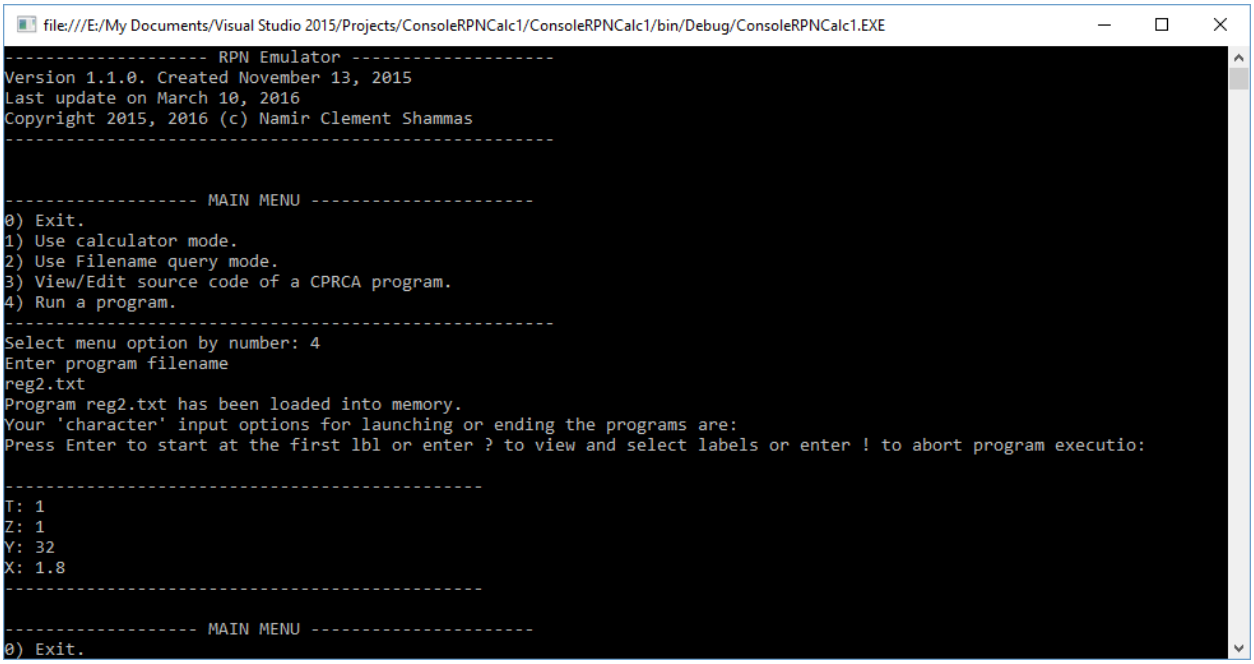


Figure 9. A sample session with the program in file reg2.txt.

Power Fit Program for Two Variables

The next program illustrates reading a data block for two variables, X, and Y, and then performing a power using the data in that block. The data file data2v2.txt must be located in the directory C:\MyData. The data file contains the following values:

```
1,1
2,4
3,9
4,16
5,25
9,81
10,100
12,144
```

The program in file reg2pwr.txt performs a linear regression using the above data.

```
Lbl start
'C:\MyData\data2v2.txt
1
readdata2
1
Powerlr
Rtn
```

The output displays R^2, the intercept, and the slope in stack registers Z, Y, and X, respectively. Figure 10 shows a sample session with the program in file reg2pwr.txt.

Copyright © 2015, 2016 by Namir Clement Shammas

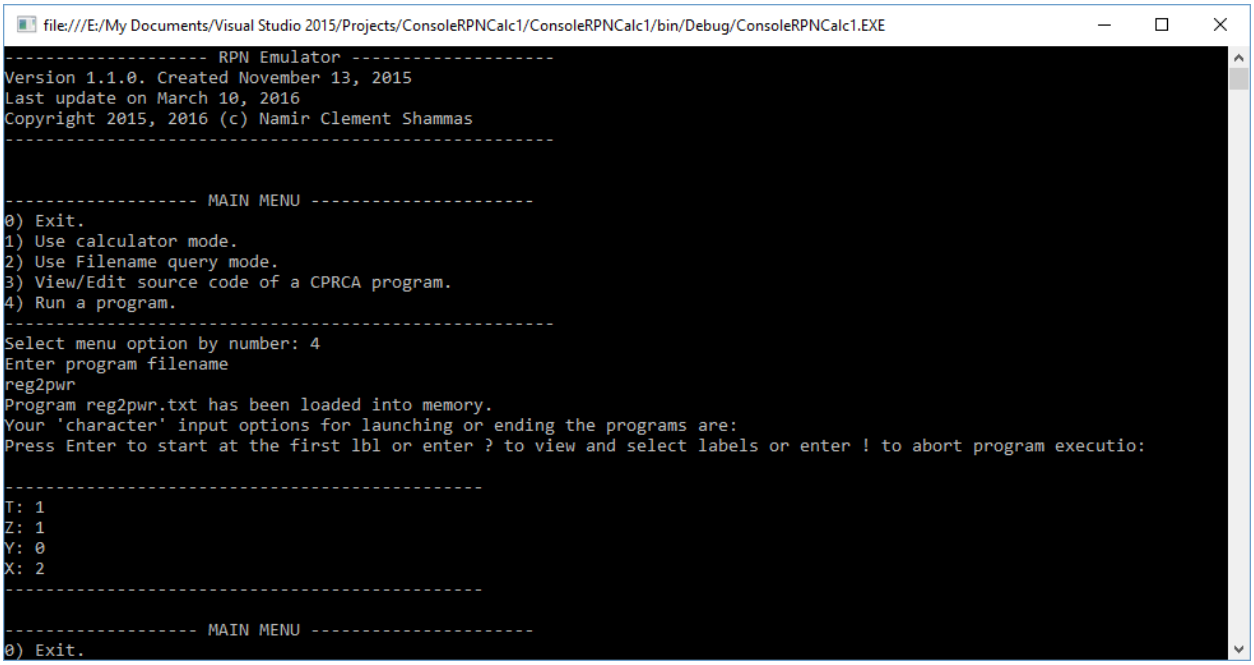


Figure 10. A sample session with the program in file reg2pwr.txt.

Multiple Linear Regression Program

The next program illustrates reading a data block for three variables, X, Y, and Z, and then performing multiple linear regression using the data in that block. The data file data3v1.txt must be located in the directory C:\MyData. The data file contains the following values:

```
1,10,21
2,22,43
5,10,61
7,30,101
1,1,12
6,40,101
```

The program in file reg3.txt performs a linear regression using the above data.

```
Lbl start
'C:\MyData\data3v1.txt
1
Readdata3
1
1
1
1
mlr
rtn
```

The output displays R^2, the intercept, the slope for Y, and the slope for X, in stack registers T, Z, Y, and X, respectively. Figure 11 shows a sample session with the program in file reg3.txt.

Copyright © 2015, 2016 by Namir Clement Shammas

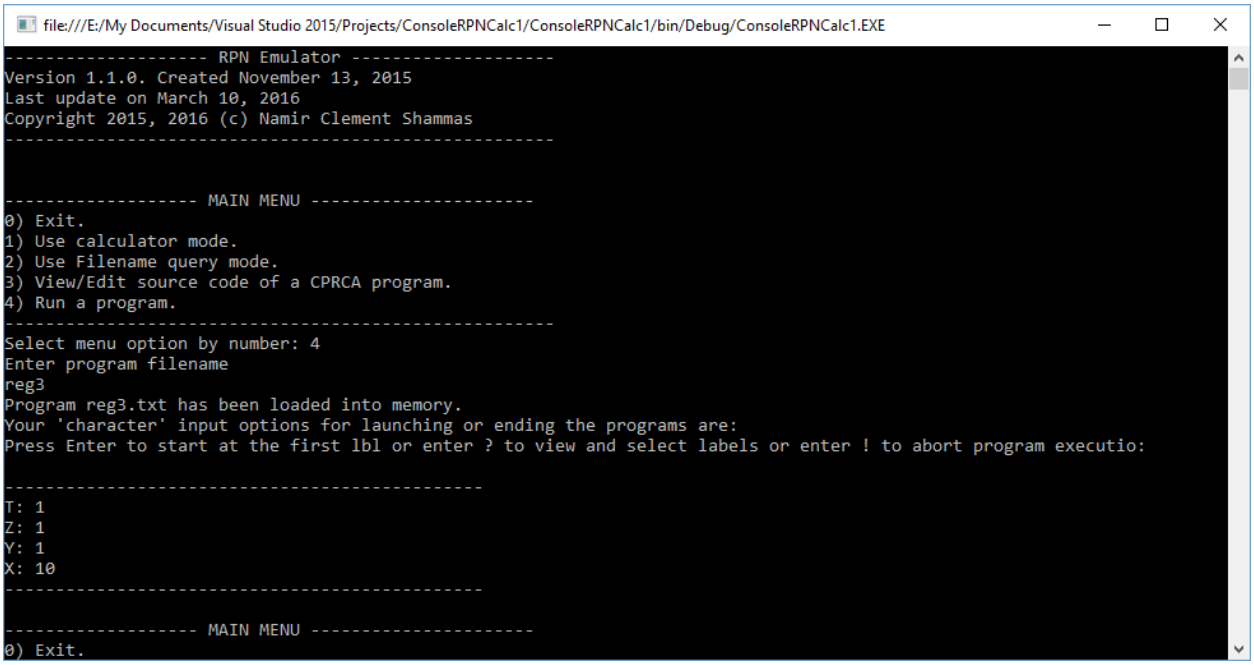


Figure 11. A sample session with the program in file reg3.txt.

Simple Linear Regression Program, Take 2

The next program uses the commands CLEARSIGMA, SIGMA+, and LRXY to perform a simple linear regression that does not involve reading data from a text file. The program in file reg2b.txt performs the simple linear regression. Figure 12 shows a sample session with the program in file reg2b.txt.

```
# Simple linear regression
LBL LR
clearsigma
50
10
sigma+
77
25
sigma+
86
30
sigma+
212
100
sigma+
lrxy
rtn
```

The program uses data supplied internally for the sake of demonstration. As such the steps that insert the data add to the program length. These input steps make the point that using file I/O is more practical, especially if you are dealing with a sizable number or data sets. The output shows a perfect linear fit that belongs to the equation that converse degrees Celsius into degree Fahrenheit:

F = 32 + 1.8 C

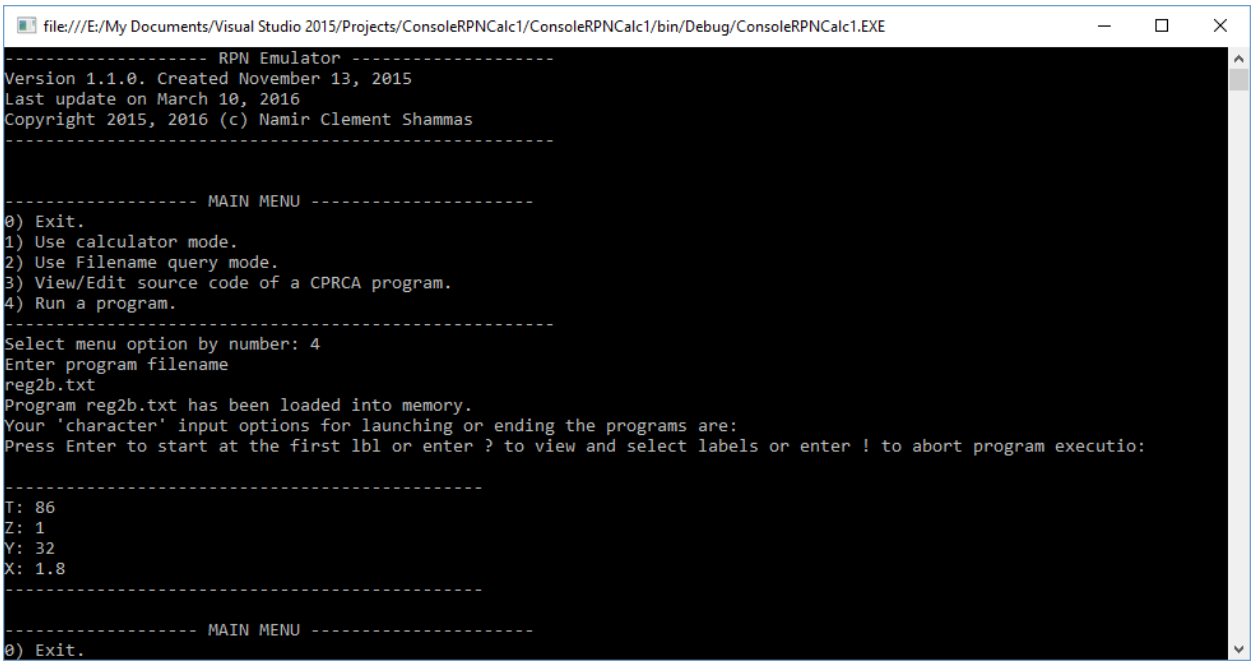


Figure 12. A sample session with the program in file reg2b.txt.

Using The SOLVE Command

The programs in file solve1.txt and SOLVE2.txt use the SOLVE command. The first program finds the roots of $f(x)=e^x - 3x^2$, while the second program permits you to specify a non–linear function, at runtime, by entering it as an RPN expression. Since the second program is more flexible, I will demonstrate it. The source code for SOLVE2.txt is:

```
# Test Solve command
LBL SolveIt
'Enter RPN expression
ainput
asto RPNexpr
'Enter Guess?
prompt
sto 0
arcl RPNexpr
1e-8
100
rcl 0
SOLVE
Rtn
```

Figure 13 shows a sample session with SOLVE2.txt that solves $f(x)=e^x–3x^2$, using an initial guess of 4.1. The program internally sets the tolerance to 1e–8 and the maximum number of iterations to 100.

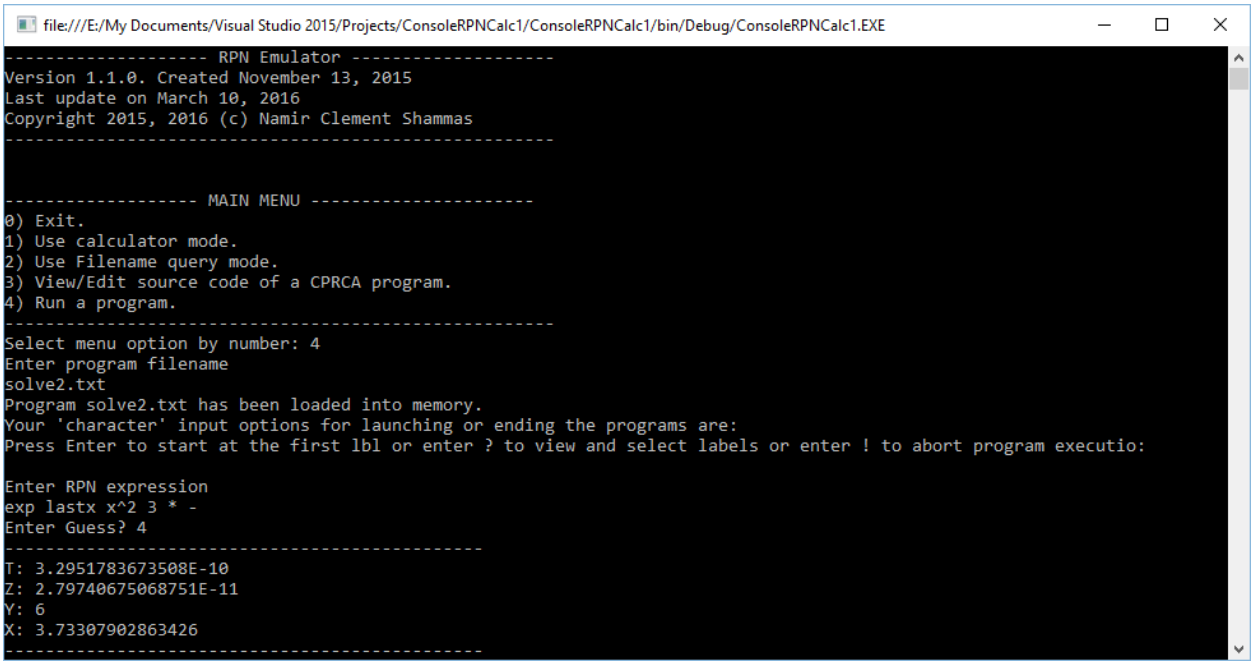


Figure 13. A sample session with SOLVE2.txt

Using The INTEG Command

The program file integ2.txt illustrates using the INTEG command to perform a numerical integration using the RPN expression in the Alpha register as f(x). Figure 14 shows a sample session where I integrate f(x)=1/x from x=1 to x=2, using a tolerance value of 1e–8.

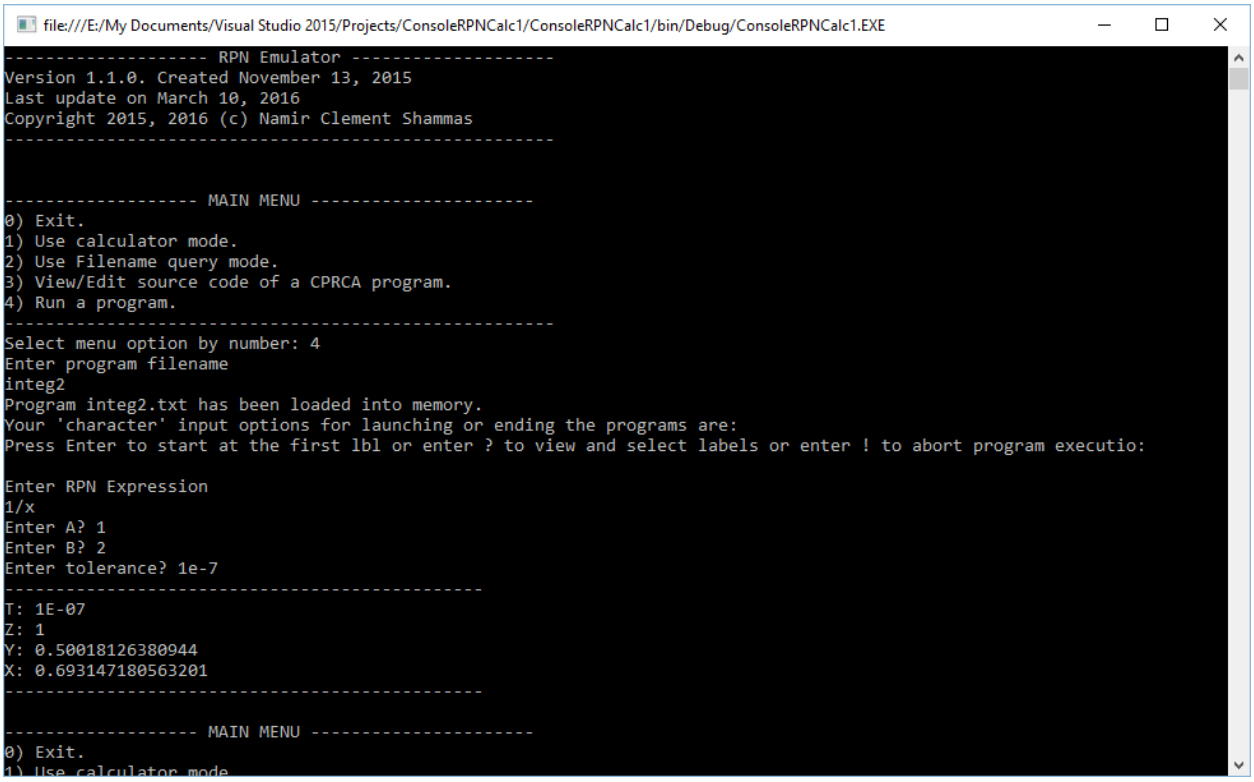


Figure 14. A sample session with program integ2.txt.

Best Linearized Regression Model

This program tests the BESTLR command. The source code for the bestlr.tx is:

```
# Best LR selection
'C:\MyData\data2v1.txt
1
readdata2
1
bestlr
aview
'Regression coefficients
aview
showstk
'Best powers
aview
bestlrpwrs
rtn
```

The program uses the data in file data2v1.txt. Figure 15 shows a sample session with program bestlr.txt. The results confirm that the linear model is the best one. The output displays R^2, the intercept, and the slope. The program also writes all of the regression model statistics to file BESTLRxxx.CSV. Figure 16 shows using Excel to view that file.

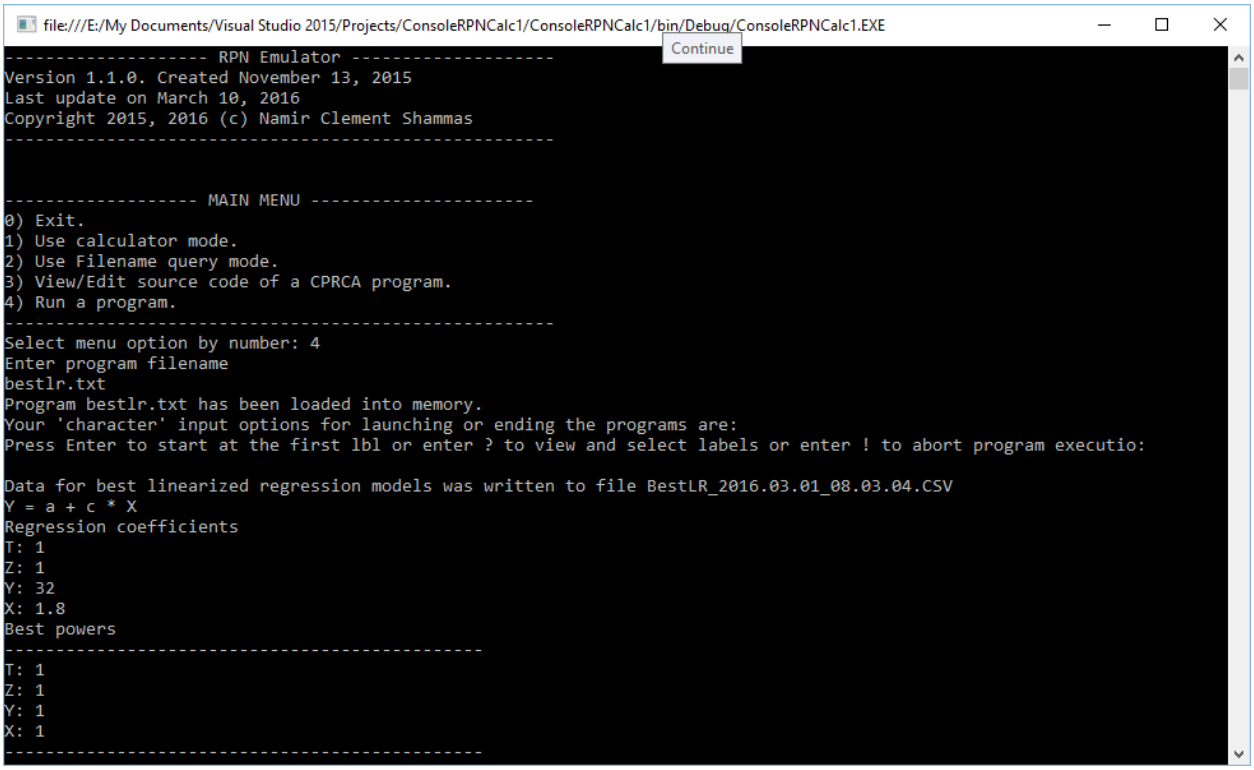
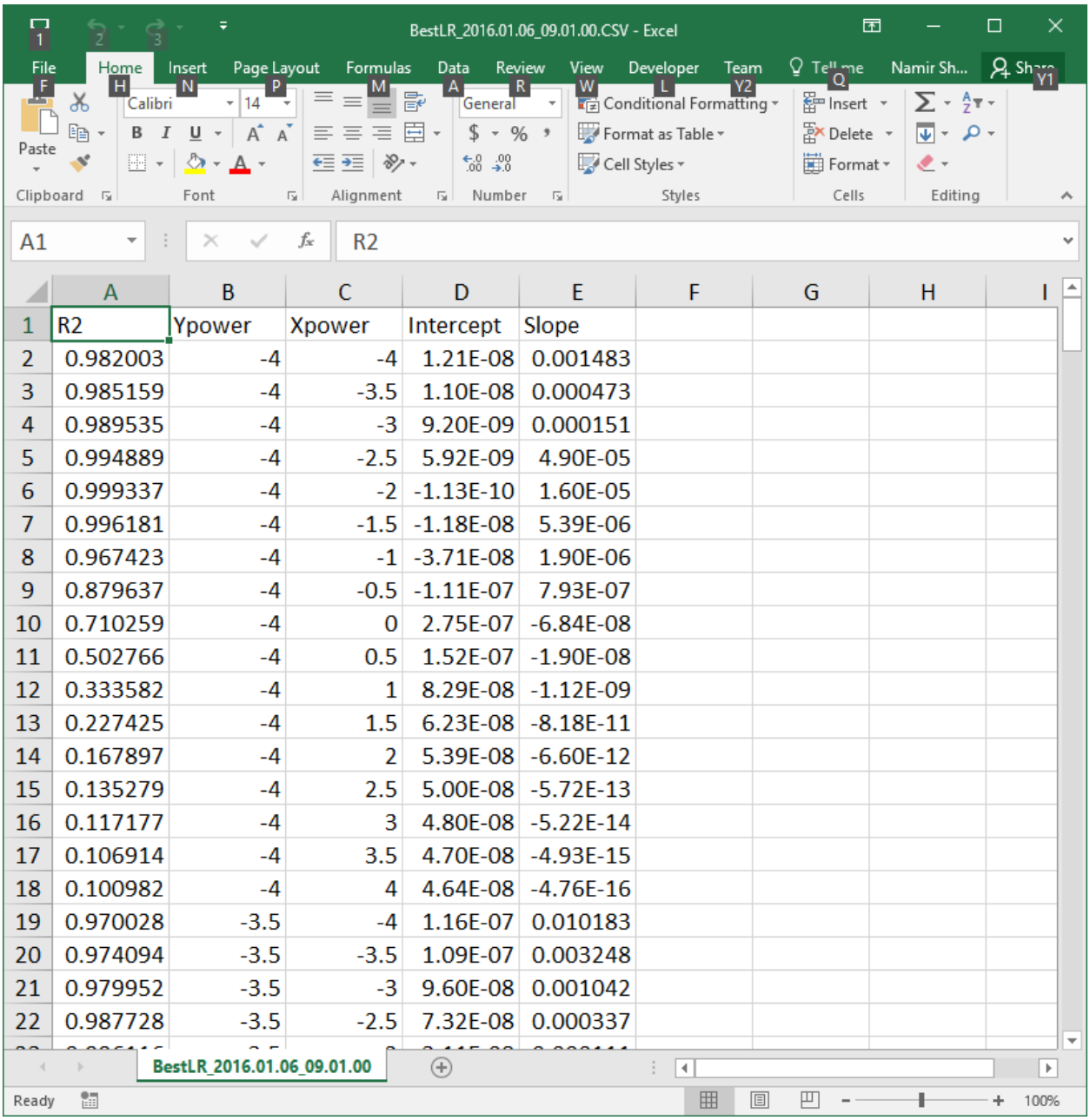


Figure 15. A sample session with program bestlr.txt.



	A	B	C	D	E
1	R2	Ypower	Xpower	Intercept	Slope
2	0.982003	-4	-4	1.21E-08	0.001483
3	0.985159	-4	-3.5	1.10E-08	0.000473
4	0.989535	-4	-3	9.20E-09	0.000151
5	0.994889	-4	-2.5	5.92E-09	4.90E-05
6	0.999337	-4	-2	-1.13E-10	1.60E-05
7	0.996181	-4	-1.5	-1.18E-08	5.39E-06
8	0.967423	-4	-1	-3.71E-08	1.90E-06
9	0.879637	-4	-0.5	-1.11E-07	7.93E-07
10	0.710259	-4	0	2.75E-07	-6.84E-08
11	0.502766	-4	0.5	1.52E-07	-1.90E-08
12	0.333582	-4	1	8.29E-08	-1.12E-09
13	0.227425	-4	1.5	6.23E-08	-8.18E-11
14	0.167897	-4	2	5.39E-08	-6.60E-12
15	0.135279	-4	2.5	5.00E-08	-5.72E-13
16	0.117177	-4	3	4.80E-08	-5.22E-14
17	0.106914	-4	3.5	4.70E-08	-4.93E-15
18	0.100982	-4	4	4.64E-08	-4.76E-16
19	0.970028	-3.5	-4	1.16E-07	0.010183
20	0.974094	-3.5	-3.5	1.09E-07	0.003248
21	0.979952	-3.5	-3	9.60E-08	0.001042
22	0.987728	-3.5	-2.5	7.32E-08	0.000337

Figure 16. Partial view of the contents of file BESTLRxxx.CSV using Excel.

Best Multiple Linearized Regression Model

This program tests the BESTMLR command to find the best multiple linearized regressions model. The source code for the bestmlr.txtis:

```
# Best MLR selection
'C:\MyData\data3v1.txt
1
readdata3
1
bestmlr
aview
```

Copyright © 2015, 2016 by Namir Clement Shammas

```
'Regression coefficients
aview
showstk
'Best powers
aview
bestmlrpwrs
rtn
```

The program uses the data in file data3v1.txt. Figure 17 shows a sample session with program bestmlr.txt. The results confirm that the multiple linear model is the best one. The output displays R^2, the intercept, the slope for variable Y, and slope for variable X. The program also writes all of the regression model statistics to file BESTMLRxxx.CSV. Figure 18 shows using Excel to view that file.

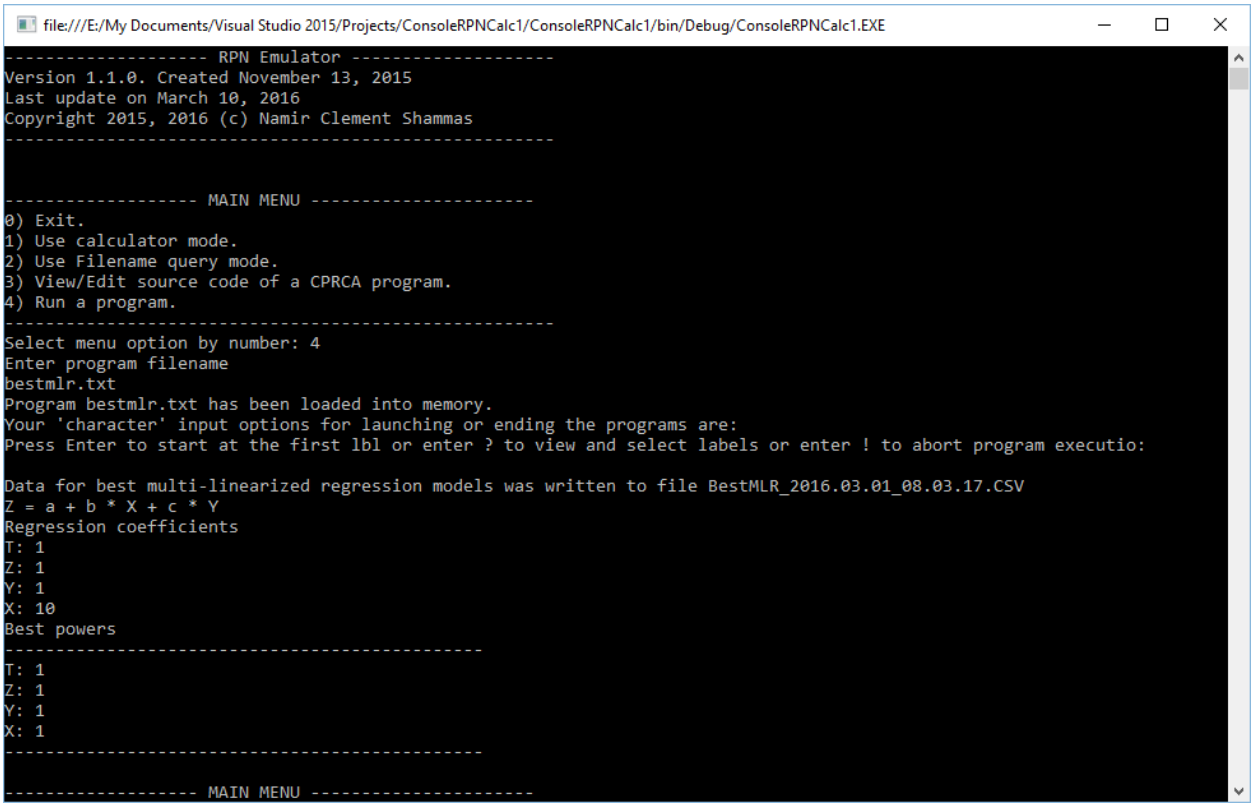
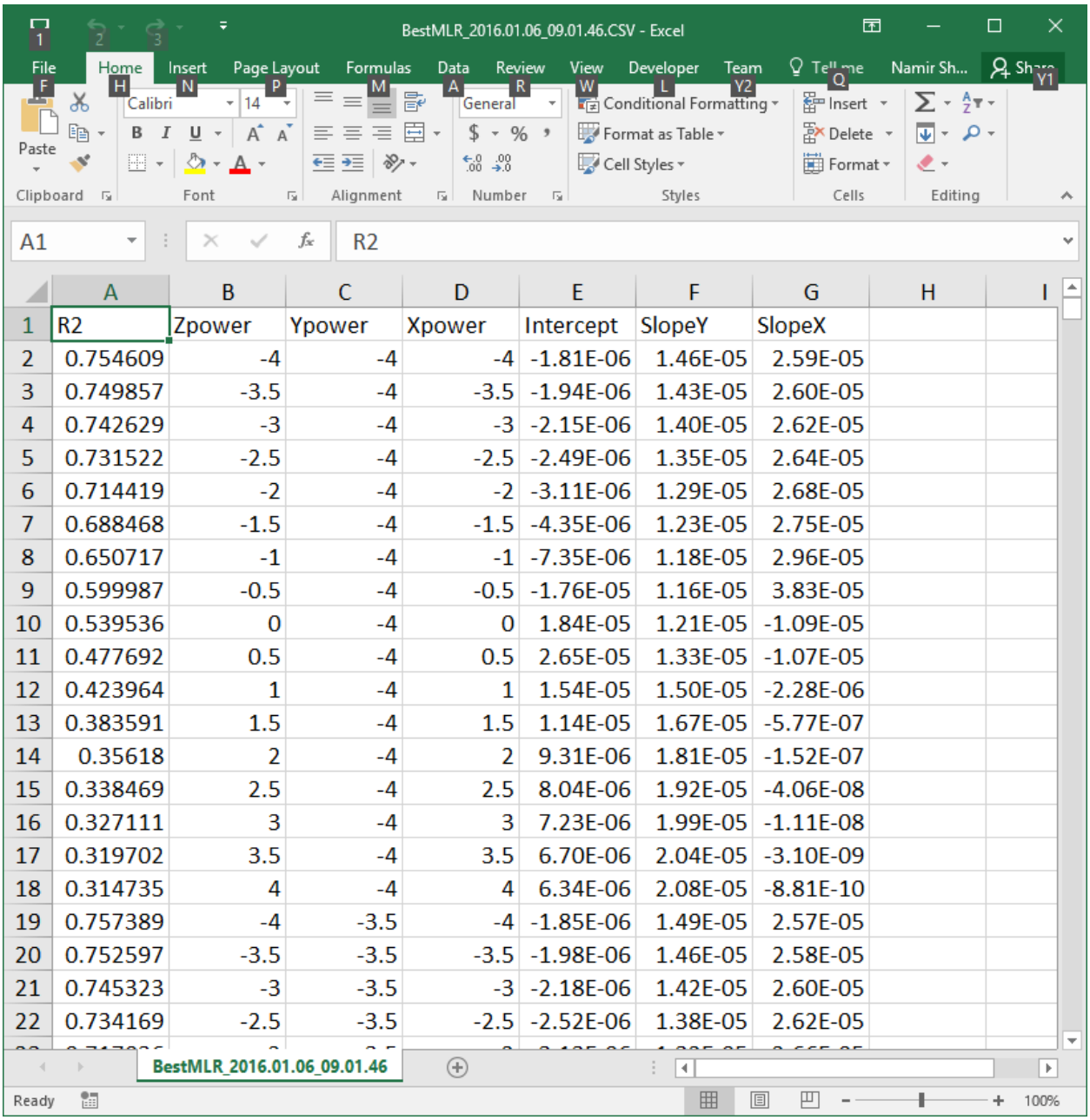


Figure 17. A sample session with program bestmlr.txt.



	A	B	C	D	E	F	G
1	R2	Zpower	Ypower	Xpower	Intercept	SlopeY	SlopeX
2	0.754609	-4	-4	-4	-1.81E-06	1.46E-05	2.59E-05
3	0.749857	-3.5	-4	-3.5	-1.94E-06	1.43E-05	2.60E-05
4	0.742629	-3	-4	-3	-2.15E-06	1.40E-05	2.62E-05
5	0.731522	-2.5	-4	-2.5	-2.49E-06	1.35E-05	2.64E-05
6	0.714419	-2	-4	-2	-3.11E-06	1.29E-05	2.68E-05
7	0.688468	-1.5	-4	-1.5	-4.35E-06	1.23E-05	2.75E-05
8	0.650717	-1	-4	-1	-7.35E-06	1.18E-05	2.96E-05
9	0.599987	-0.5	-4	-0.5	-1.76E-05	1.16E-05	3.83E-05
10	0.539536	0	-4	0	1.84E-05	1.21E-05	-1.09E-05
11	0.477692	0.5	-4	0.5	2.65E-05	1.33E-05	-1.07E-05
12	0.423964	1	-4	1	1.54E-05	1.50E-05	-2.28E-06
13	0.383591	1.5	-4	1.5	1.14E-05	1.67E-05	-5.77E-07
14	0.35618	2	-4	2	9.31E-06	1.81E-05	-1.52E-07
15	0.338469	2.5	-4	2.5	8.04E-06	1.92E-05	-4.06E-08
16	0.327111	3	-4	3	7.23E-06	1.99E-05	-1.11E-08
17	0.319702	3.5	-4	3.5	6.70E-06	2.04E-05	-3.10E-09
18	0.314735	4	-4	4	6.34E-06	2.08E-05	-8.81E-10
19	0.757389	-4	-3.5	-4	-1.85E-06	1.49E-05	2.57E-05
20	0.752597	-3.5	-3.5	-3.5	-1.98E-06	1.46E-05	2.58E-05
21	0.745323	-3	-3.5	-3	-2.18E-06	1.42E-05	2.60E-05
22	0.734169	-2.5	-3.5	-2.5	-2.52E-06	1.38E-05	2.62E-05

Figure 18. Partial view of the contents of file BESTMLRxxx.CSV using Excel.

Using Local Variables

The next program involves using Newton's method to calculate the root of a function. It is a variant of the program newton1.txt that I presented earlier. The label fx defines how the function f(x) is calculated and uses a single local register to store the argument x passed on the stack. Figure 19 shows a sample session with the program. The session shows the user's input of an initial guess for the root and a tolerance value. The program displays the refinement in the guess and then the final refinement (along with the other stack registers). The source code for the file localregs1.txt is:

Copyright © 2015, 2016 by Namir Clement Shammas

```
# Testing local memory registers
lbl newtn
'Enter guess?
prompt
vsto x
'Enter tolerance?
prompt
vsto toler
lbl 0
vrcl x
viewvar x
abs
1
+
.001
*
vsto h
vrcl x
gsb fx
vsto f0
vrcl x
vrcl h
+
gsb fx
vrcl f0
-
1/x
vrcl f0
*
vrcl h
*
vsto- x
abs
vrcl toler
x<y?
gto 0
vrcl x
end
lbl fx
lsto 0
exp
lrcl 0
x^2
3
*
-
rtn
```

The code after lbl fx stores the argument x in the local memory register at index 0, using the command lsto 0. The code later recalls the value of x using the command lrcl 0.

Copyright © 2015, 2016 by Namir Clement Shammas

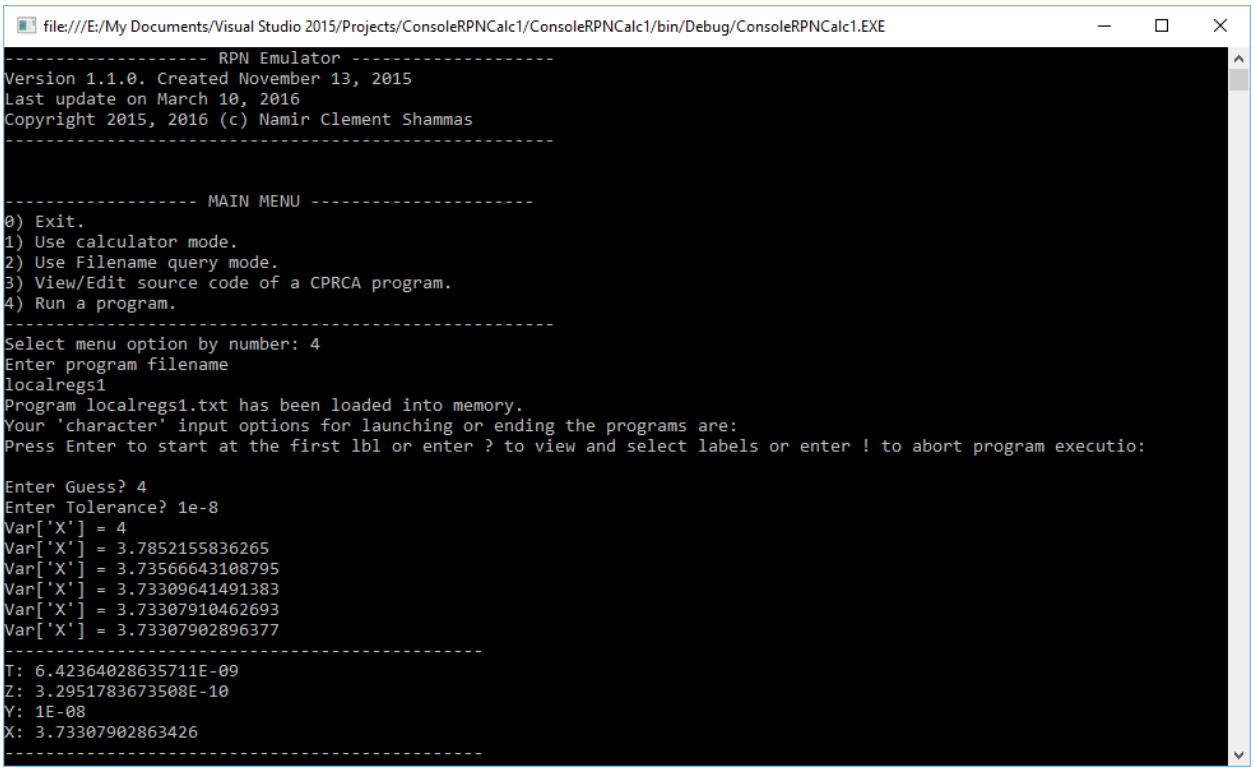


Figure 19. A sample session with program localregs1.txt.

Using Local Variables, Take 2

The next program illustrates using local memory registers in subroutines three levels deep. Figure 20 shows a sample session with the program, where the user has entered 1. The calculations done by the program are purely arbitrary. The listing for the localregs2.txt program is:

```
# More advanced example of local registers
lbl start
'Enter x?
prompt
lsto 5
x^2
lrcl 5
sqrt
/
gsb sub1
lrcl 5
+
end
lbl sub1
lsto 0
2
*
gsb sub2
lrcl 0
2
+
/
```

```
rtn
lbl sub2
lsto 0
1/x
gsb sub3
lrcl 0
ln
*
rtn
lbl sub3
lsto 1
x^2
lrcl 1
ln
/
Rtn
```

The above program uses three nested subroutines named sub1, sub2, and sub3. The subroutines sub1 and sub2 uses a local memory register at index 0. The subroutine sub3 uses a local memory register at index 1. You can also notice that the main routine (after label Start) also uses its own local memory register at index 5. It does not use the common memory registers!

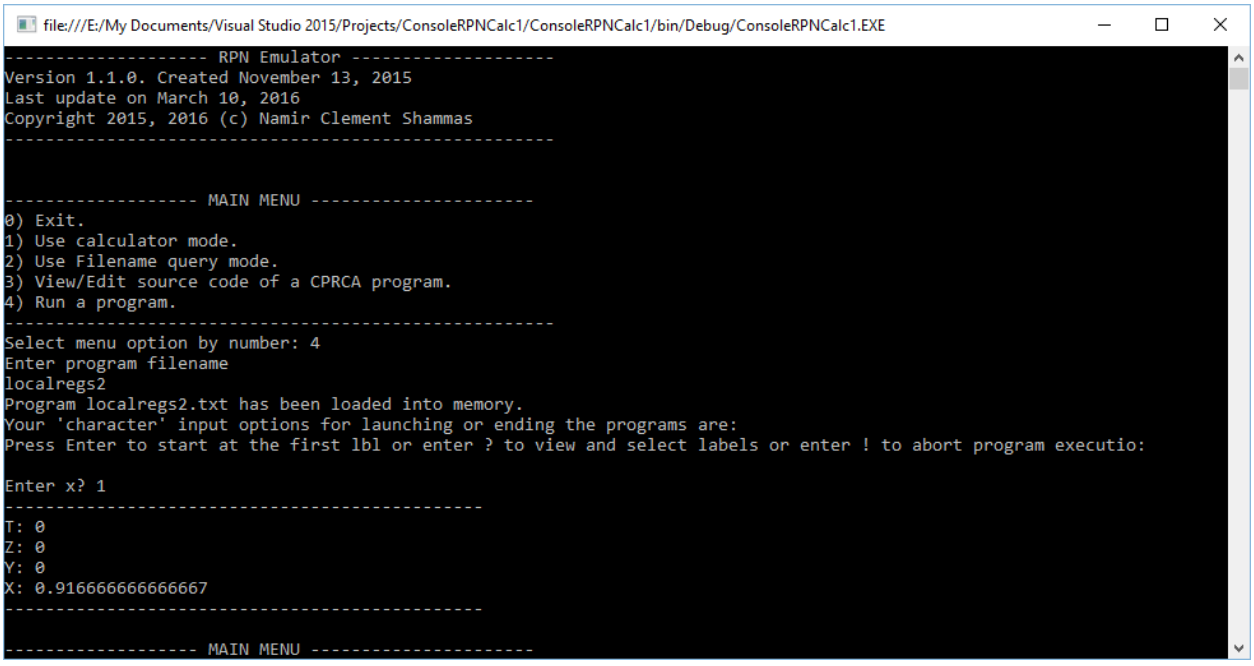


Figure 20. A sample session with program localregs2.txt.

Recursive Calls

The next program illustrates recursive calls to a subroutine that also uses a local memory register. The file recursion.txt contains a program that calculates the factorial recursively. Figure 21 shows a sample session with that program. The session shows an input of 15. The program calculates and displays the value of 15! In two ways. The first method uses the recursive subroutine. The second uses the command FACT. The second result confirms the correctness of the first one. The source code for the recursion.txt is:

```
# Illustrates recursive calls
lbl start
'Enter N?
prompt
abs
int
lsto 0
1
x<>y
gsb fact
x<>y
lrcl 0
fact
rtn
lbl fact
x=0?
rtn
lsto 0
*
lrcl 0
1
-
gsb fact
rtn
```

The label fact contains the recursive subroutine. Notice that the last statement before the rtn command calls the subroutine recursively. Also notice that the first statement in the subroutine tests the value of the X stack register with 0. If the two values match, the subroutine executes a rtn command. Otherwise, the subroutine uses a local memory register as part of the recursive calculation of the factorial. Also notice that the main routine uses a local memory register at index 0.

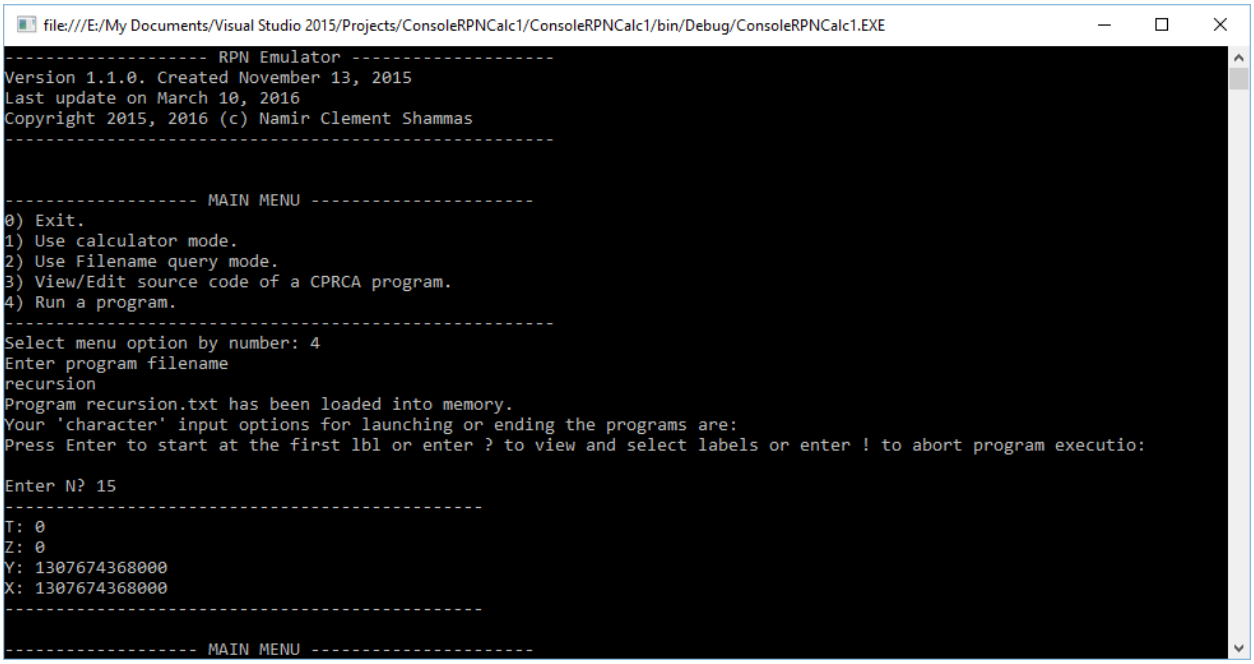


Figure 21. A sample session with program recursion.txt.

Formatted Output

The next program displays arbitrary values using different formats. The program does not require any user input. It internally supplies the numbers and the display formats. The listing for the frmt1.txt program is:

```
# Test formatting
'pi=
pi
FRMT F0
|- using F0 format string
AVIEW
'pi=
pi
FRMT F2
|- using F2 format string
AVIEW
'pi=
pi
FRMT F4
|- using F4 format string
AVIEW
'X=
123456
FRMT E3
|- using E3 format string
AVIEW
'X=
123456
FRMT E6
|- using E6 format string
AVIEW
'X=
123456
FRMT E9
|- using E9 format string
'N=
123456
FRMT D
|- using D format string
AVIEW
'N=
123456
FRMT D4
|- using D4 format string
AVIEW
'N=
123456
FRMT D8
|- using D8 format string
AVIEW
'N=
FRMT X4
|- using X4 format string
AVIEW
```

Copyright © 2015, 2016 by Namir Clement Shammas

```
NOSTACK
rtn
```

The program tests the F, E, D, and X formats. Figure 22 shows the output of the program. Notice how the F and E formats display fewer digits after the decimal place. Also notice the leading zeros when using the D8 format to display the decimal integer 123456 as 00123456. Likewise notice the leading zeros when using the X4 format to display the decimal integer 255 as x00FF.

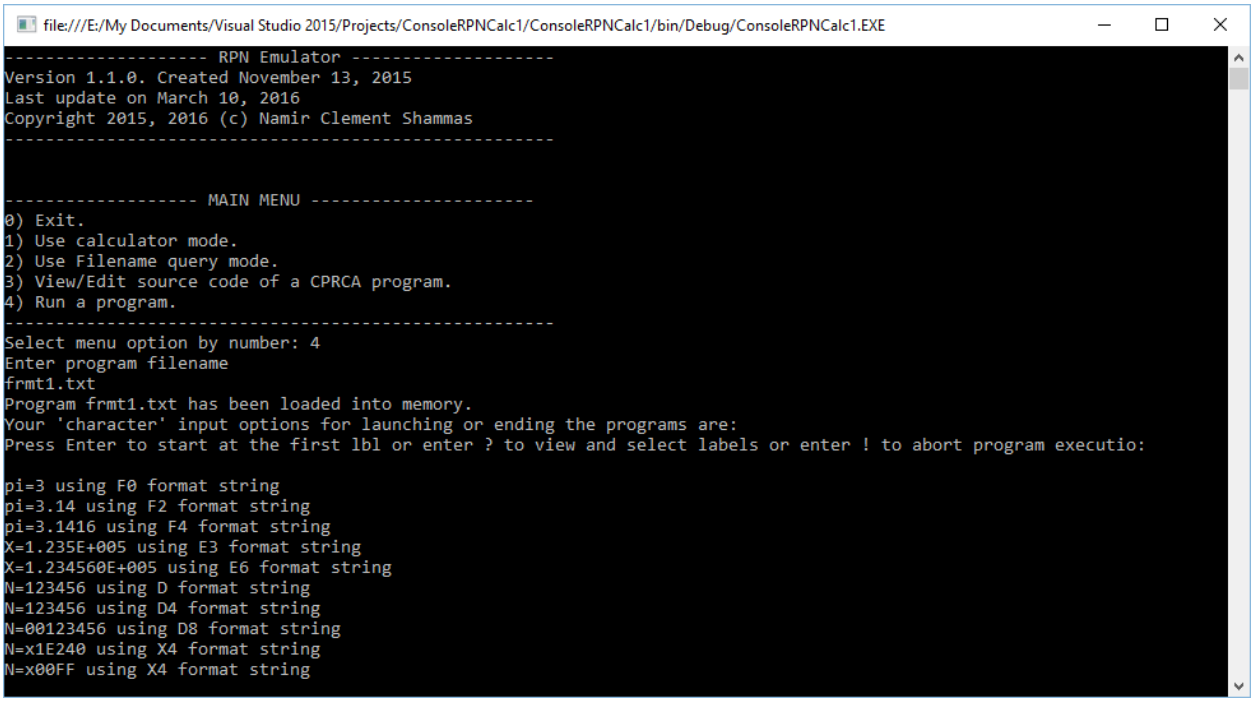


Figure 22. A sample session with program frmt1.txt.

Efficient Access of Array Variable Elements

The program namarr1.txt initializes the XARR array variable with 1000 elements with random values between 0 and 100. The program then scans the elements of the array variable to calculate statistical summations. The program uses these summations to calculate the mean and standard deviation of the random values. Figure 23 shows a sample session with the namarr1.txt. Here is the source code for the program:

```
# Testing array variable
lbl start
100
vsto xhi
0
vsto xlow
vsto sumx
vsto sumx2
vsto i
'Xarr
1000
vsto n
ARNEW
```

```
0
# set the automatic index of RSTO and RRCL
sidx
0
ridx
# first loop to store random numbers in array variable
lbl 0
rand
vrcl xhi
vrcl xlow
-
*
vrcl xlow
+
rst+
1
vsto+ i
vrcl n
vrcl i
x<=y?
gto 0
# second loop to calculate basic stats
0
vsto i
lbl 1
rrc+
vsto x
vsto+ sumx
x^2
vsto+ sumx2
1
vsto+ i
vrcl n
vrcl i
x<=y?
gto 1
vrcl n
'Num Obs = '
arclx
aview
vrcl sumx
vrcl n
/
'Mean = '
frmt f3
aview
vrcl sumx2
vrcl sumx
x^2
vrcl n
/
-
vrcl n
1
-
/
sqrt
```

Copyright © 2015, 2016 by Namir Clement Shammas

```
'Sdev = '
frmt f3
aview
nostack
end
```

The program uses named individual variables as well as the array variable XARR. The program places the name of this array in the Alpha register until it needs to display tagged results that use the Alpha register. The program contains two loops located between LBL 0 and GTO 0 and between LBL 1 and GTO 1. The four executable commands before LBL 0 assign zeros to the storage and recall indices of array XARR. These statements use the SIDX and RIDX commands. They use the contents of the Alpha register and the value in the X stack register. The loop after LBL 0 creates the random values (between 0 and 100) one by one and store them in the array variable using the command RST+. This command uses the contents of the Alpha register to obtain the targeted array variable. The second loop, located after LBL 1, recalls the values from the array variable XARR using the command RRC+. This command also uses the contents of the Alpha register to obtain the targeted array variable. The storage and recall indices of the array variable XARR work behind the scene to access the appropriate array elements. The contents of the Alpha register remain unchanged until the program moves beyond the end of the second loop. The iterations of both loops are explicitly controlled by the named variable I.

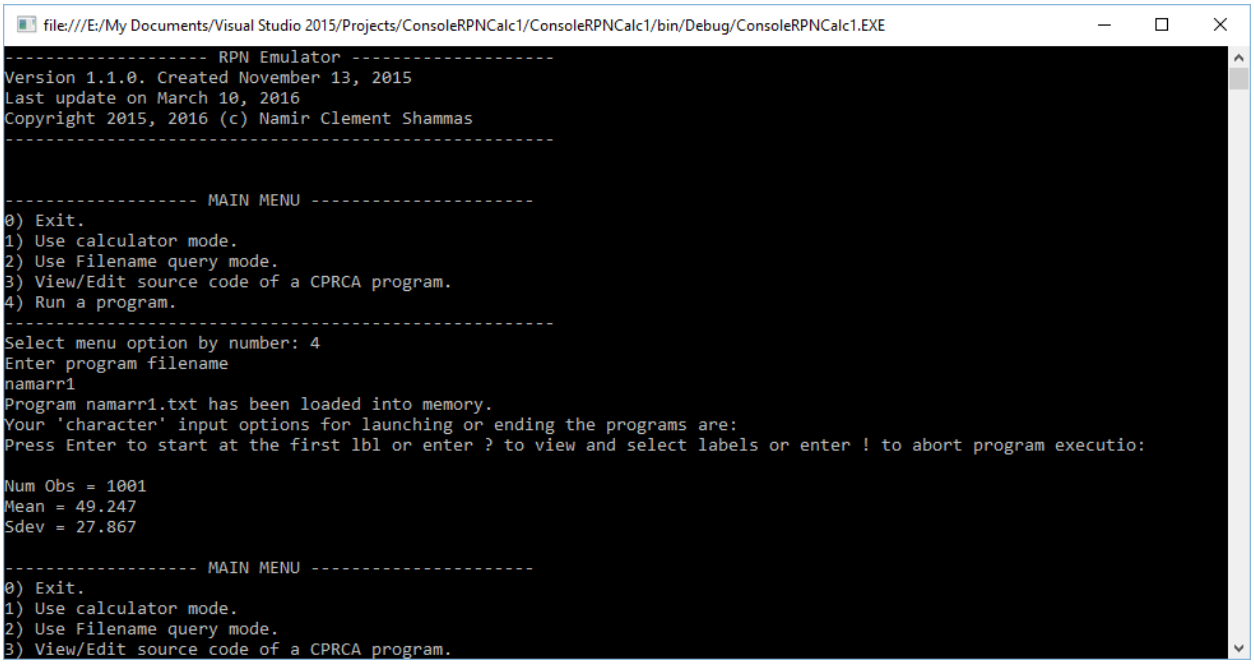


Figure 23. A sample session with program namarr1.txt

Adding Time Units

The next program date1.txt illustrates adding various types of time units to an arbitrary date/time of January 1, 2016, 2pm. The program displays comments for each use of the command ADDDT. The program also uses the commands DSTO, DRCL, and VIEWDT to store, recall, and view date/time values, respectively. Figure 24 shows the output of the program.

```
# Test date and time functions
lbl start
'#1/1/2016 2:00:00 PM#
asto dt1
1
'Starting date
aview
viewdt dt1
'Adding 1 day
aview
'Day
adddt dt1
viewdt dt1
'Adding 1 month
aview
'Month
adddt dt1
viewdt dt1
'Adding 1 year
aview
'Year
adddt dt1
viewdt dt1
'Adding 1 hour
aview
'Hour
adddt dt1
viewdt dt1
'Adding 1 minute
aview
'Minute
adddt dt1
viewdt dt1
'Adding 1 second
aview
'Second
adddt dt1
viewdt dt1
'Recalling date/time in variable dt1
drcl dt1
end
```

You can achieve the same date/time calculations if you replace the string literal for the date/time value with equivalent floating-point values in the X and Y stack registers. Here is the following code snippet that uses floating-point values for the date and time:

```
# Test date and time functions
lbl start
# The date/time is 1/1/2016 2:00:00 PM
# The decimal date goes in the Y stack register
2016.0101
# The decimal time goes in the X stack register
14.00000
dsto dt1
1
```

Copyright © 2015, 2016 by Namir Clement Shammas

```
'Starting date
…
end
```

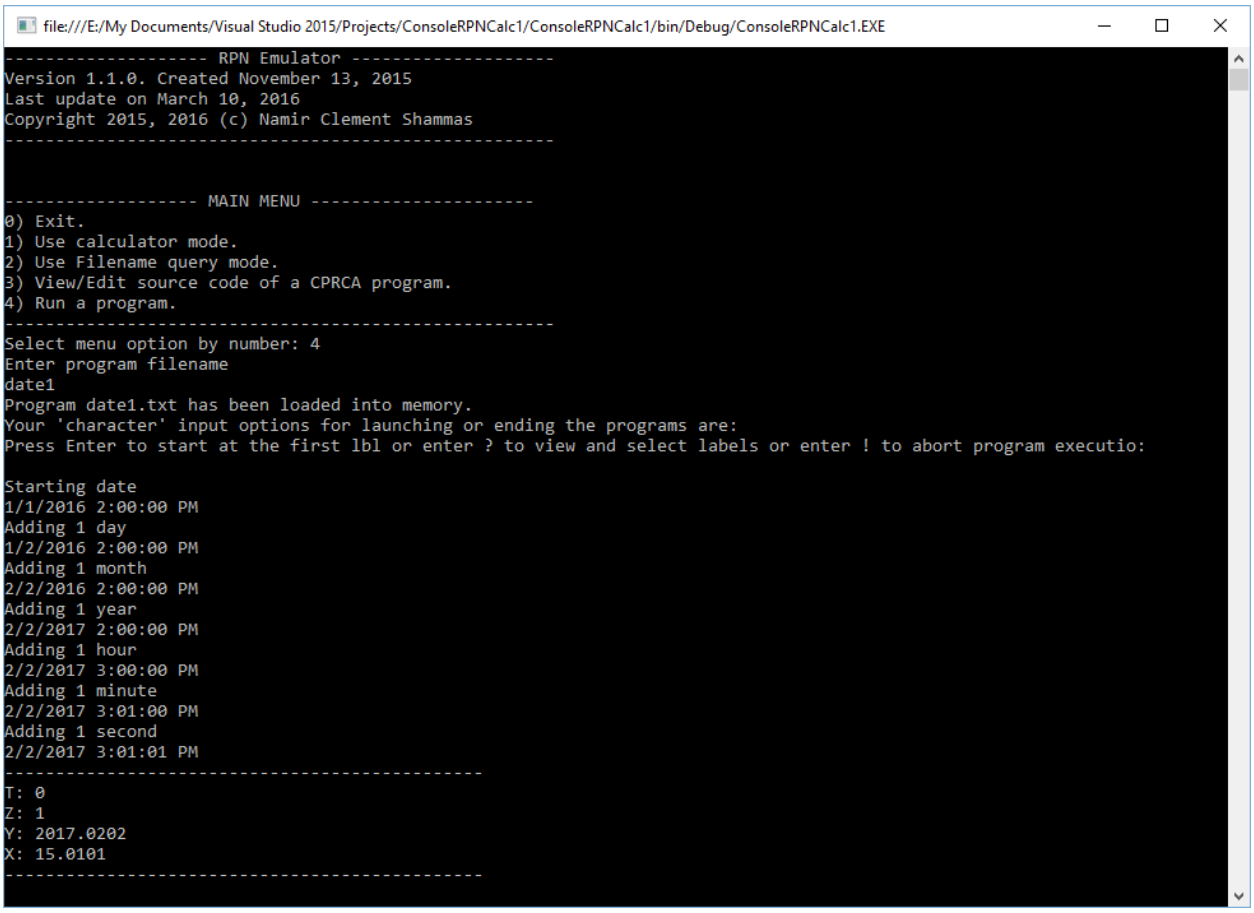


Figure 24. A sample session with program date1.txt.

Subtracting Date/Time Units

The next program date2.txt illustrates subtracting various types of time units between the arbitrary date/time values of January 1, 2017, 2pm and January 1, 2016, 2pm. The program displays comments for each use of the command DIFFDT. The program also uses the DSTO and VIEWDT commands to store and view date/time values, respectively. Figure 25 shows the output of the program.

```
# Test date and time functions
lbl start
'#1/1/2016 2:00:00 PM#"
asto dt1
'#1/1/2017 2:00:00 PM#"
asto dt2
1
'Starting dates
aview
viewdt dt1
viewdt dt2
'Subtracting days
```

```
aview
'Day
diffdt dt1 dt2
viewx
'Subtracting months
aview
'Month
diffdt dt1 dt2
viewx
'Subtracting years
aview
'Year
diffdt dt1 dt2
viewx
'Subtracting hours
aview
'Hour
diffdt dt1 dt2
viewx
'Subtracting minutes
aview
'Minute
diffdt dt1 dt2
viewx
'Subtracting seconds
aview
'Second
diffdt dt1 dt2
viewx
nostack
end
```

You can achieve the same date/time calculations if you replace the string literals for the date/time with equivalent floating-point values in the X and Y stack registers. Here is the following code snippet that uses floating-point values for the date and time:

```
# Test date and time functions
lbl start
# 1/1/2016 2:00:00 PM
# The decimal date goes in the Y stack register
2016.0101
# The decimal time goes in the X stack register
14.00000
DSTO dt1
# 1/1/2017 2:00:00 PM
# The decimal date goes in the Y stack register
2017.0101
# The decimal time goes in the X stack register
14.00000
dsto dt2
1
'Starting dates
…
end
```

Copyright © 2015, 2016 by Namir Clement Shammas

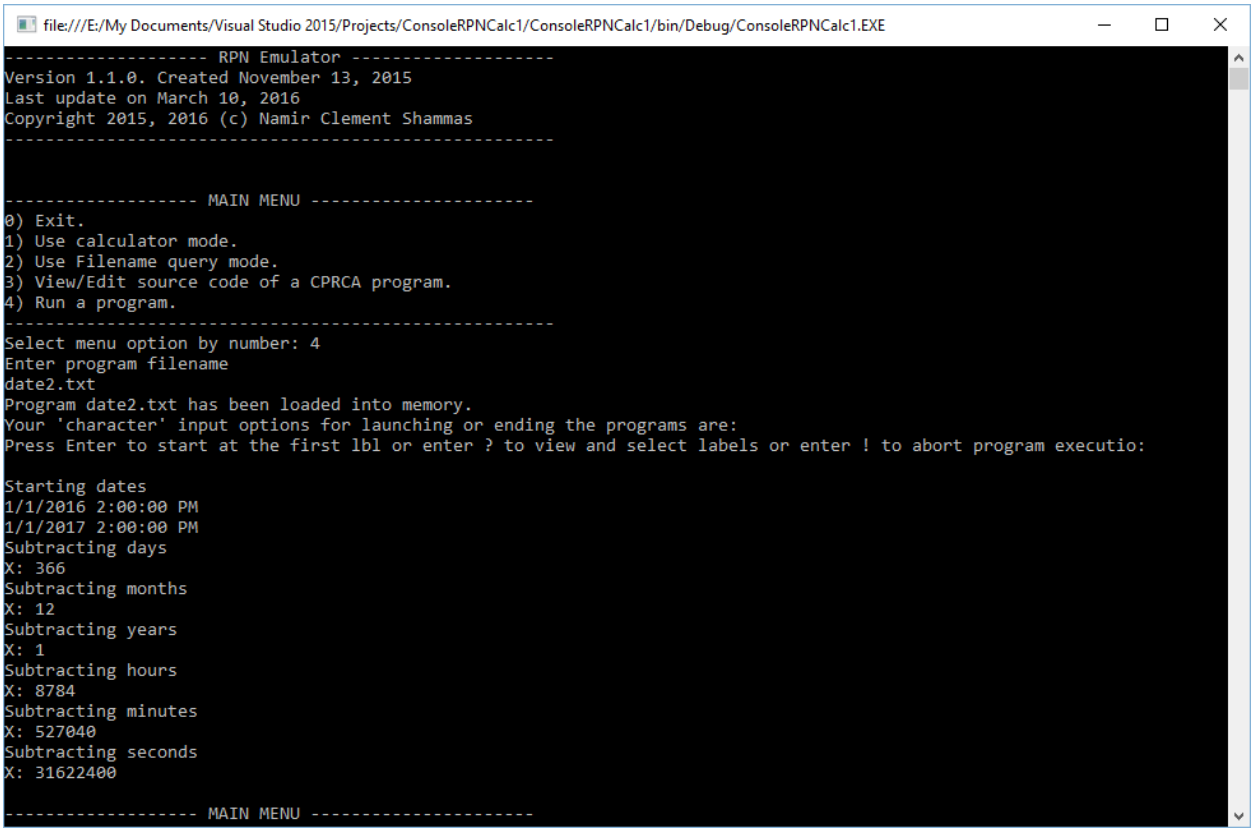


Figure 25. A sample session with program date2.txt.

Scanning a Function

The scandemo.txt program illustrates the SCAN command, introduced in version 1.1. This program provides internal input to scan the function $f(x)=e^x - 3x^2$ in the range of [–1, 4] in steps of 0.1. The command uses a tolerance of 1e–8 and function tolerance of 1e-8. Figure 26 shows the output of the scandemo.txt program.

```
# Test SCAn command
lbl Start
'Scanning f(x)=exp(x)-3*x^2
aview
'From x=-1 to x=4 in steps of 0.1 and tolerance of 1e-8
aview
'exp lastx x^2 3 * -
# memIdx
0
# FxToler
1e-8
copystack
# Toler
1e-8
# Step
0.1
# A
-1
# B
4
```

```
scan
-x-
nostack
end
```

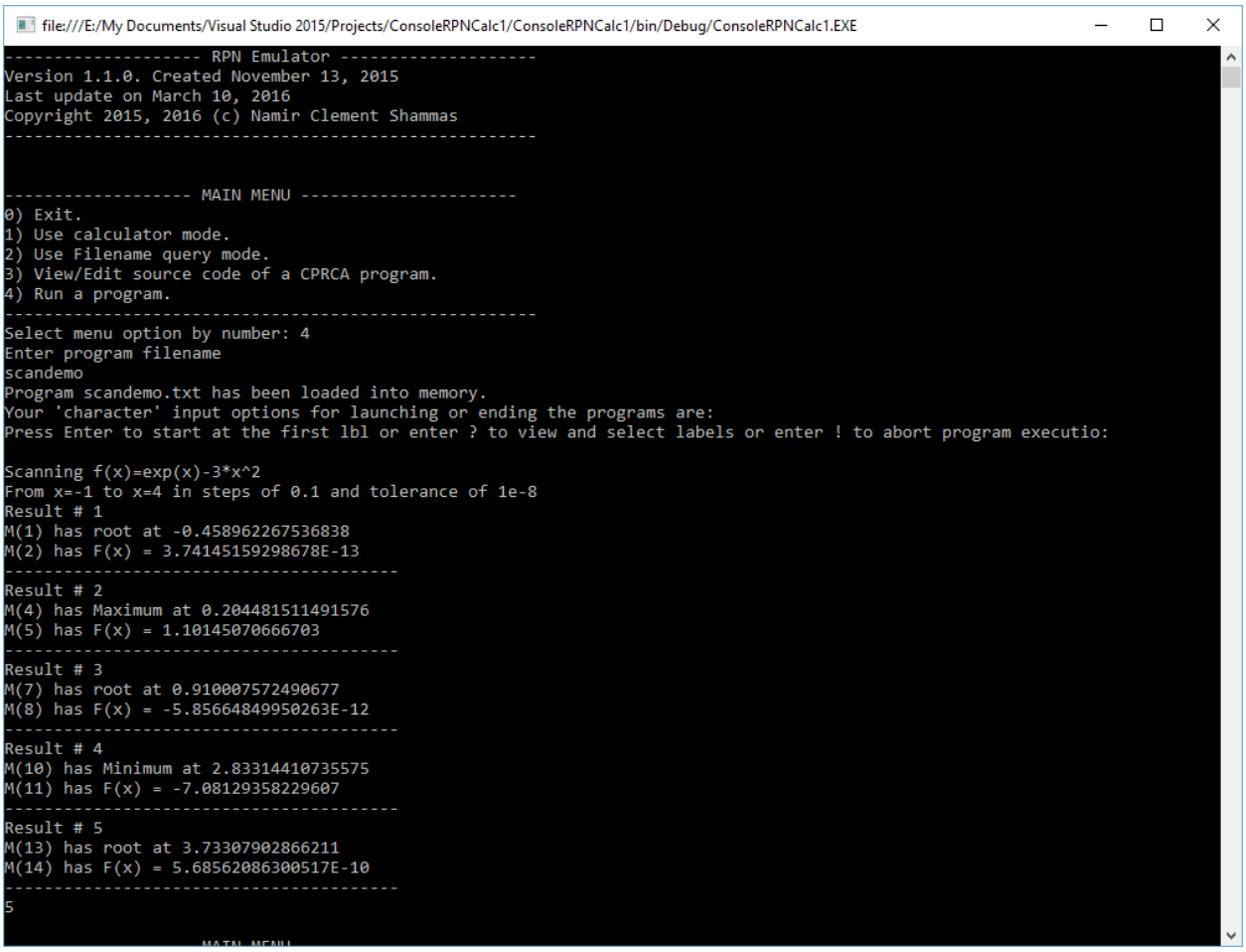


Figure 26. A sample session with program scandemo.txt.

Gauss-Kronrod Quadrature

The program gausskronquad.txt allows you to perform numerical integration for a function in a given range. The listing for the program is:

```
LBL GaussKronrodQuad
'Gauss-Kronrod Quadrature
aview
'Enter RPN Expression
AINPUT
ASTO RPNexpr
'Enter A?
prompt
vsto a
'Enter B?
prompt
vsto b
'Order?
```

```
prompt
vsto order
'Enter tolerance?
prompt
vsto toler
vrcl order
vrcl toler
vrcl b
vrcl a
arcl RPNexpr
gausskronquad
end
```

Figure 27 shows a sample session with the gausskronquad.txt program. The session integrates f(x)=1/x from x=1 to x=10, which is ln(10). The calculations use a Legendre polynomial order of 10 and tolerance value of 1e–7.

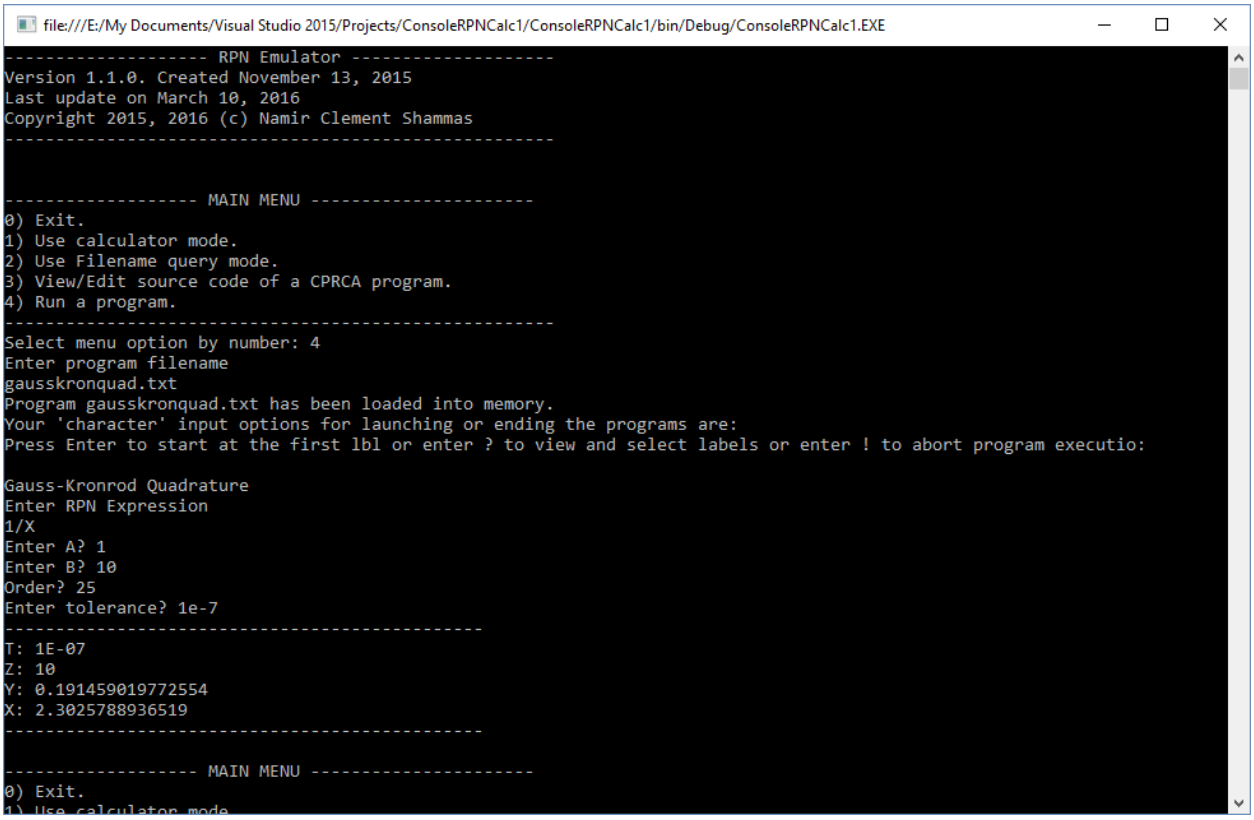


Figure 27. A sample session with program gausskronquad*.txt.*

Appendix A

This appendix presents the format strings used by the FRMT command. Table 4 shows a summary of the format strings supported by the application. Consult Microsoft web's site to learn more details about the tabulated formatting strings in Visual Basic. The programming language supports other format strings (for example, for date and time) which the CPRCA application does not support.

Format	Name	Description	Example
D or d	Decimal	Integer digits with optional negative sign.	D for 1234 displays 1234. D6 for 1234 displays 001234. D4 for 1234.5678 displays 1234.
E or e	Scientific	Exponential notations.	E4 for 123.4567 displays 1.2345E+2.
F or f	Fixed–point	Integral and decimal digits with optional negative sign.	F2 for pi displays 3.14.
G or g	General	The most compact of either scientific notation or fixed values.	G for –123.456 displays –123.456.
P or p	Percentage	The value is multiplied by 100 and displayed with a % character.	P2 for 2 displays 200%.
X or x	Hexadecimal	Displays an integer as a hexadecimal string.	X2 for 255 displays FF. X2 255.1234 displays FF.

Table 4. The format strings supported by the CPRCA application.

Appendix B

The next table offers a summary of the STO commands in the CPRCA application.

STO Command	Source	Destination
ASTO	Alpha register	Text variable
DSTO	Stack	Text variable (has text as a date/time value)
LSTO	Stack	Local memory register
RSTO	Stack	Array variable
STO	Stack	Memory
STOFLGS	Flags	Text variable
STOSTX, STOSTY, STOSTZ, STOSTT	Stack	Stack
VSTO	Stack	Variable

Appendix C

The next table offers a summary of the RCL commands in the CPRCA application.

RCL Command	Source	Destination
ARCL	Text variable	Alpha register
ARCLIX, ARCLIY, ARCLIZ, ARCLIT	Integer value of Stack	Alpha register
ARCLX, ARCLY, ARCLZ, ARCLT	Stack	Alpha register
DRCL	Text variable (has text as a date/time value)	Stack
FORCL	Text variable	Stack
LRCL	Local memory register	Stack
RCL	Memory	Stack

Copyright © 2015, 2016 by Namir Clement Shammas

RCL Command	*Source*	*Destination*
RCLFLGS	Text variable	Flags
RCLSTX, RCLSTY, RCLSTZ, RCLSTT	Stack	Stack
RRCL	Array variable	Stack
VRCL	Variable	Stack

Final Remarks

The CPRCA application grew gradually from a simple exercise, then to a more usable simple programmable calculator application, and then into a versatile application with many features. As each day came, I added and modified features. It went from a little exercise to a whopping mammoth! I felt my work exemplified the old saying "A cowboy's work is never done." If you enjoy programming the HP–41C and the HP–42S you will enjoy tinkering with the CPRCA application. Remember that it is a labor of love and work in progress. There as certainly bugs to be discovered and typo errors in this manual to be found. I appreciate your bug reports and patience. I hope you enjoy the new features that I added to the HP-41C programming language.

The application processes the RPN expressions and program statements in two different areas. As you might expect, the code uses IF–ELSEIF–ELSE statements that examine numerous ELSEIF clauses to find the right command or function to process. To speed up this process I adopted the following schemes:

1. Determining if a program line contains a command that is part of the RPN expressions or belong to the set of programming commands. This distinction speeds up program execution by avoiding fruitless lookup of RPN expression commands when the current command is a programing one.
2. Grouping commands and functions that share a common leading (or trailing) set of characters in their names. The grouping uses an ELSEIF clause to detect the common part of the command names. It then uses a nested IF–ELSEIF–ELSE statement to process each command in that group.
3. Offering *task-consolidating* commands that perform tasks on memory registers, flags, and the elements of array variables. These commands can be implemented using other simpler commands. However, the task-consolidating commands significantly reduce the number of commands processed by CPRCA's interpreter.

Copyright © 2015, 2016 by Namir Clement Shammas

Document Release and Updates History

Doc Version	*App Version*	*Date Released/Updated*	*Comment*
0.9.0	1.0.0	January 16, 2016	Initial release of beta version the document.
1.0.0	1.1.0	March 3, 2016	1. Updated the source code to use new VB 2015 features. 2. Implemented hash tables for faster access of names for array variables, named variables, and named text variables. 3. Fixed a few bugs. 4. Added additional SOLVE commands. 5. Added several Gaussian quadrature commands. 6. Added two more examples.

Copyright © 2015, 2016 by Namir Clement Shammas