QuIP Manual

A system for QUick Image Processing
with simple scripts or quips

Jeffrey B. Mulligan, NASA Ames Research Center

QuIP Manual for version 1.0.02, generated 27 October 2018

Table of Contents

1 Overview

QuIP stands for QUick Image Processing, and is also a reference to its scripting language which allows the user to implement a variety of useful operations with short fragments of script or "quips."

1.1 History

QuIP has been developed continously over a period of several decades, under a series of flavors of UNIX. The QuIP interpreter is a front end for a number of libraries; some of these are system libraries, such as X Windows and OpenGL for image display, and Motif for GUI widgets. Other libraries are distributed as part of the QuIP package, and provide support for various specialized functions, including support for a variety of hardware devices, and psychophysical experimentation. It provides an environment for visual and numerical computing similar in functionality to Matlab and the like.

The QuIP interpreter has its roots in a simple terminal-based system to interactively control experiments. It was designed to facilitate the interaction and minimize the amount of typing required of the user. It only gradually grew into a full-fledged programming language, and still exhibits various shortcomings. At various times, switching to another interpreter (such as Tcl) was considered, but the loss of some of QuIP's unique features seemed too high a price to pay. You can be the judge.

1.2 First steps

Here we present the simplest possible QuIP script. Invoke QuIP by typing 'quip' at your command shell prompt. (QuIP command prompts typically end with the string '> '; the top level prompt is 'quip> '.) Characters typed by the user are appear as *user_input*.

```
quip> echo "Hello world" RET
Hello world
quip>
```

If you type this example in, you might notice that after the first 'e' is typed, the interpreter completes the word 'expressions', but that the completion disappears as soon as the following 'c' is typed.

Here is a slightly longer version:

```
quip> view RET
quip/view> viewers RET
quip/view/viewers> new v 300 100 RET
quip/view/viewers> quit RET
quip/view> draw v RET
quip/view/draw> string "Hello world" 20 20 RET
quip/view/draw> quit RET
quip/view> quit RET
quip>
```

The first command `view` causes the interpreter to enter a submenu, which is reflected in the prompt. By convention, submenus are exited with the `quit` command.

2 Command Language

The simplest way to interact with QuIP is via the command language.

2.1 Interactive Features

QuIP has many unique interactive features.

2.1.1 Prompts

Prompts and prompt suppression

When QuIP is waiting for user input, a prompt is printed. Command prompts typically end in "> ", preceded by a string indicating the menu hierarchy. Parameter prompts are typically of the form "Enter <parameter description>: ".

Prompts are only printed when QuIP is starved for input. So if multiple commands are entered on a single line, no prompt will be printed before the second command is read.

The fact that commands prompt for missing arguments makes them somewhat self-documenting. If you forget the number or order of parameters to a given command, you can simply enter it interactively and note the prompts. The transcripting facility (LINK) provides a convenient way of recording the results.

2.1.2 Response completion

As you start to type in the name of a command, QuIP will complete the word for you if it can.

2.1.3 Input redirection

You can read the contents of a file by using the '<' builtin command. The effect is the same as if you were to type in the contents of the file, with the exception that commands and command arguments are not added to the respective history lists.

2.1.4 Transcripts

You can save a transcript of your typing for later re-execution. Usually this is not done without a bit of editing of the transcript. A common use of this feature is that you are unsure of what arguments are required for a particular command, so you execute it interactively, using the prompts as guides for the arguments. Transcripting offers an alternative to paper-and-pencil notes concerning the order and types of arguments.

A transcript file is started using the '>' builtin command.

2.2 Menus

QuIP's command menus

Commands in QuIP are grouped into menus. The menu hierarchy is designed so that all of the commands in a menu can be displayed on a single screen (i.e., fewer than 20 commands per menu).

2.2.1 Displaying the menu

How to display the current menu

The commands in the current menu can be shown using the command `?`. This is a builtin command, i.e. it is universally available in any menu.

2.2.2 Builtin commands

Commands available from any menu

Some commands are so useful that it is desireable to have them available regardless of the menu context. These include commands for manipulating variables, loop control, and file redirection. The complete list of builtin commands can be displayed using the `??` command.

2.2.2.1 Loop control

These commands are useful for creating control structures.

If <condition> <command>
If <condition> Then <command> Else <command>

When an `If` is encountered, the next word is read and interpreted as a boolean condition using the scalar expression parser. If the word following the condition is the reserved word `Then`, then three words are read, the second of which must be the reserved word `Else`.

Multi-word commands (or multiple commands) may be executed following an `If` by enclosing the entire group in single or double quotes. `If` commands can be nested, but because the command language does not use braces for grouping, it is often better to write a macro to encapsulate the predicate of an `If` statement.

do <commands> while <condition>

repeat <count> <commands> end

foreach <varname> (<wordlist>) <commands> end

These three forms provide for looping. After the loop is opened, commands are read until the closing word is encountered. Unlike command shells such as (t)csh, the commands are immediately executed as they are read. When the loop is scanned for the first time, the commands are saved to a buffer before each one is executed. When the end of the loop is reached, the condition is evaluated, and when appropriate the stored loop body is pushed back onto the input.

2.2.2.2 Generating output

Printing messages and warnings

2.2.2.3 Interacting with the system

OS-specific commands

2.2.3 Exiting the menu

Returning to the previous menu

By convention, menus are exited using the `quit` command. This command is not a builtin, it is specified individually for each menu. Most menus will use the default menu exit method, but occasionally it is necessary to perform cleanup actions on menu exit.

2.3 Startup

QuIP initialization

Before interpreting user input, QuIP attempts to read a startup file. QuIP looks for a startup file with the same name as by which the program was invoked, with the .scr (script) extension added. Thus, the default startup file is named quip.scr. Creating links to the QuIP executable with different names provides a way to produce different behavior on startup through the use of specialized startup scripts.

QuIP searches for a startup script in the following series of directories: 1) the current directory; 2) $QUIPSTARTUPDIR (if defined in the environment); 3) $HOME/.quip/startup (if it exists); 4) the macro installation directory (/usr/local/share/quip/macros/startup).

Options 2 and 3 happen only when the user customizes his or her environment appropriately. The installation directory used in the last case can be modified during the configuration process (XREF building).

2.4 Variables

Using script variables

The QuIP has a set of string variables that have a usage similar to command interpreters such as bash and csh.

2.4.1 The variables submenu

Commands to manipulate variables are found in the variables submenu, accessed via the `variables` builtin command. The commonly used functions such as setting a variable are generally invoked via a macro which hides the descent into the submenu.

From the variables submenu, the `list` command lists the names of all variables. The `find` command lists the names of variables containing a given text string, while the `search` command lists the names of variables whose values contain a given string.

Other commands are described in the following sections.

2.4.2 Setting variables

Variables are normally set using the `Set` macro:

```
Set <var_name> <value_string>
```

Variable names can be made up of alphanumeric characters and the underscore '_'.

By convention, normal variable names begin with an alphabetic character, and numeric variables are reserved for macro arguments. This is not strictly enforced, however. If a variable with name `1` is created and set to a value, then that will prevent the first macro argument from being accessed. This is not a good thing.

2.4.3 Using variables

Variable expansion is peformed when lines are read. Variable substitution is triggered by the dollar sign character `$`. As in the UNIX command shells, variable expansion is inhibited by enclosing a string in single quotes, and always performed for strings in double quotes.

Nested quotes can become rather complicated. It is the outermost set of quotes that matter; In the following example, the variable will be expanded when the line is first read; the text interpreted after evaluation of the `If` condition will be `echo 'John is a minor'`.

```
Set name John
Set age 15
If $age>=18
Then "echo '$name is an adult'"
Else "echo '$name is a minor'"
```

On the other hand, the quotes could be switched:

```
If $age>=18
Then 'echo "$name is an adult"'
Else 'echo "$name is a minor"'
```

In this case, the text to be interpreted will be `echo "$name is a minor"`. This difference can become important if such a statement is encountered in a loop in which the value of the variable changes from one iteration to the next.

Another example where immediate variable expansion must be inhibited is when the variable may not exist:

```
If var_exists(test)
Then 'echo "variable test exists and has value = $test"'
Else 'echo "variable test does not exist"'
```

In this example, the first echo command after the `Then` must be enclosed in single quotes, because all of the text is scanned and subjected to variable expansion before evaluation of the condition. Enclosing the first `echo` command in double quotes would generate an undefined variable warning in the case where the variable does not exist, even though the command will not be executed. This might be considered a design flaw.

2.4.4 Special variables

Some variables are preset by the system. In particular, command line arguments given when the program is invoked may be accessed as `$argv1`, `$argv2`, etc. The may also be accessed using the abbreviated form used for macro arguments, e.g. `$1`, `$2`, etc., but the longer form is useful inside a macro where the short form accesses the macro arguments.

Scripts may refer to external environment variables, but they are loaded as needed. For example, `HOME` will not show up on the initial list of variables, but an attempt to dereference `$HOME` will cause the environment variable `HOME` to be imported if it exists.

Other special variables may be set by particular functions. These are described in the description of the functions.

2.4.5 Numeric variables

It is not uncommon to use a variable to hold a numeric value. The interpreter functions which fetch numeric arguments to commands process their arguments through a scalar expression parser, but sometimes we would like to have the result of the expression evaluation stored in the variable, instead of the expression. This is accomplished using the `Assign` macro. `Assign` is like `Set` in that it sets the value of a variable, but it does so after running the value through the scalar expression parser, and formatting the result as a number. For example:

```
quip> Set pi 4*atan(1)
quip> Print pi
pi = 4*atan(1)
```

```
quip> Assign pi 4*atan(1)
quip> Print pi
pi = 3.141593
```

Several macros are provided to facilitate the manipulation of numeric variables:

```
Add_Var <var_name> <number>
Mul_Var <var_name> <number>
Increment <var_name>
Decrement <var_name>
```

These correspond to the C language operators `+=`, `*=`, `++` and `--`, respectively.

2.4.6 Advanced variable usage

Unlike the UNIX command shells, QuIP does not provide for indexing of variables. In part, that is because QuIP uses braces for subscripting data objects, and variables are commonly used to hold the names of data objects.

QuIP does provide, however, a slightly cumbersome way to do indexing via double indirection:

```
Set person1 Larry
Set person2 Moe
Set person3 Curly
Set i 1
repeat 3
Set var_name person$i
echo "Person $i is $$var_name"
Increment i
end
```

2.5 Macros

A macro is a block of text that is substituted when the macro name is encountered. Macros provide one of the primary means of writing compact programs for QuIP.

2.5.1 Predefined macros

A number of useful macros are loaded on program startup. (XREF startup file) The predefined macros may be displayed by entering the `list` command in the `macros` submenu.

2.5.2 Writing new macros

While the predefined macros may access to the functionality desired by a user, most everyone will want to combine these building blocks to accomplish specific things. Any group of commands which is repeated more than once is a candidate for inclusion in a macro.

Macro definitions are begun the the command `Define` (which is itself a macro, albeit a very simple one). The syntax is as follows:

Define <macro_name> <n_args> [<prompt_specifications>]
<macro_body>
.

The macro body is terminated by a line containing a single period `.`. This is the one case where white space (or the lack of it) is important: this period must be on a line by itself with no leading white space.

The number of prompt specifications must match the number of arguments specified. Each prompt specification consists of a prompt string (quoted if it contains spaces), which may be preceded by an optional object type specifier enclosed in brackets `<` and `>`. The purpose of object type specifiers is to enable reponse completion when a macro is invoked interactively.

3 Data Objects

Data objects are blocks of memory containing data, and a small block of meta-data describing the object. Data objects can be scalars, vectors, matrices, images, or image sequences.

3.1 Dimensions

Data objects can have up to five dimensions. These are the component dimension or depth, the row dimension or width, the column dimension or height, the sequence dimension or number of frames, and hypersequence dimension or number of sequences. (Hypersequences were introduced to ease compatibility with file formats in which color images are stored as a sequence of color component frames; in this framework, a color image is a 3-frame sequence, thus a color movie is a hypersequence of 3-frame sequences.)

3.2 Creating Data Objects

Data objects can be created using the commands in the data submenu.

3.3 Subscripting

Subscripting allows the user to reference parts of an object. For example, if m is a matrix (image) of real numbers, then m[0] selects the first row of the image, while m{0} selects the first column.

4 Expression Language

Writing scripts using QuIP's expression language

The command language described in the previous chapter can be cumbersome when writing complicated image processing operations. Consider, for example, synthesizing a floating point sinusoidal grating image, scaling it to the range 0 to 255, and converting it to byte. In the command language, we would do it with the following script:

```
‘ Set h 256
Set w 256
Set period 64
Assign two_pi 8*atan(1)
Image f $h $w 1 float
Image b $h $w 1 u_byte
Ramp2D f 0 $two_pi/$period 0
VSin f f
VSAdd f f 1
VSMul f f 255/2
Convert b f’
```

Here is the exact same thing written using the expression language:

```
‘ expressions
read -
int h=256, w=256;
int period=64;
float two_pi=8*atan(1);
float f[h][w];
u_byte b[h][w];
f=ramp2d(0,two_pi/period,0);
b = (sin(f)+1)*255/2;
end
quit’
```

While the second version is not significantly shorter in terms of characters, it is arguably easier to read and understand. In the command language, each operation is one command, while in the expression language multiple operations can be packed into a complex vector expression.

5 Displaying Images

Displaying images using QuIP viewers
QuIP provides a simple interface to the X Windows system to enable display of images.

5.1 Creating a viewer

Creating a new viewing window

5.2 Loading a viewer

Displaying an image in a viewer

5.3 Manipulating viewers

Repositioning and hiding viewers

6 Image Files

Loading and storing images

7 Plotting Data

Using the plotting macros

8 Control Panels

Using a GUI to control the program

9 Advanced Graphics

Sythesizing images using OpenGL

10 Interfacing Hardware

Modules have been written to allow QuIP to control a variety of hardware devices. Most of these rely on LINUX drivers.

10.1 Frame grabbers

Getting images from analog cameras.

Support is provided for several frame grabbers.

10.1.1 Matrox meteor RGB

The matrox meteor is an older PCI frame grabber that allows capturing of RGB images via composite or component signals. The component inputs can be used to capture 3 monochrome signals provided that the 3 cameras are synchronized.

10.1.2 V4L2 devices

10.1.2.1 LML BT848

10.1.2.2 Sensoray 811

10.2 Video cameras

Getting images from digital cameras.

10.3 Analog I/O

Reading & sending signals.

10.4 Serial devices

Devices that communicate via RS232.

10.5 Parallel port

Using the PC parallel port.

11 Installing QuIP

This section describes how to build QuIP from the source tarball.
QuIP can be downloaded from: http://scanpath.arc.nasa.gov/quip

Index

C

I