April 20, 2018
Geena Smith

TCL Scripting in Avizo

The console in Avizo uses a modified TCL scripting language that can be used to control modules, and is useful for automation. This document outlines some basic TCL syntax, including variable use, loops, expressions and writing to a file, as well as Avizo specific commands. Included at the end of the document are two sample scripts.

Variables

In TCL, data can be stored in variables in the form of integers, decimal values, strings. It is also possible to create sets of variables, called lists. To create or change a variable, use the following syntax:

set *variable_name value*

Ex. set x 12
set y 1.05
set name “hello world”
set list_name {“val1” “val2” “val3”}

Note that elements in a list should be separated by a space and not a comma.

In order to use these values in your script, the variable name needs to be dereferenced using a “$”. If the variable name is more than one letter in length, it is best to use braces (ie. “{“ and “}”) around the variable name to avoid errors.

$*x*
${*variable_name*}

A new variable can also be created by concatenating two or more variables.

Ex. set x “hello”
set y “world”
set z xy → returns “helloworld”

set a 1
set b 2
set c ab → returns 12

And elements can be added onto a list using:

lappend *list_name value*

Ex. set x {1 2 3}
lappend x 4 → x is now {1 2 3 4}

To access a value from a list, we use the command “lindex”. This is short for “list index” and uses the following syntax:

lindex $*list_name n*

This returns the nth value in the list, with indexing beginning at 0 (ie. First element is index 0, second is index 1, etc). Lists are useful for processing multiple image files in an Avizo file that have unique names.

Ex. Set lst {“val1” “val2” “val3”}
lindex ${lst} 0 → returns “val1”
lindex ${lst} 1 → returns “val2”

If you want to determine what value your variable is set to, type “echo” followed by your dereferenced variable name, and the value will be printed to the console.

echo $*variable_name*

Ex. echo $x
echo [lindex lst 0]

Expressions

To do any mathematical expressions or increment a variable, you must use the command “expr” followed by the expression.

Ex. echo [expr 1+2] → prints 3 to the screen
set x [expr 2*5] → x is now set to 10

Note that the brackets (ie. “[“ and “]”) are used to substitute the return variable from inside the brackets.

Writing to Files

Data from Avizo can easily be written to text files through the console. In order to do so, you must first open a file in either write or append mode, and set a variable to point to the desired file. In write mode, only the values you write to the file while it is open will appear in the file; any previous data will be overwritten. In append mode, the values you write to the file will be added to the end of the file, conserving any previous data.

set *variable_name* [open *“file_name.format”* w] → write to file (w)

Or

set *variable_name* [open *“file_name.format”* a] → append to file (a)

Note that the brackets (ie. "[" and "]") are used to substitute the return variable from inside the brackets.

To add to a file, use the following syntax:

```
puts $variable_name data
```

The data will be written to the file on a new line.

```
Ex.   puts $myfile "Hello World"
      puts $yourfile 42
      puts $anotherfile [expr 1+2]
      puts $lastfile $var
```

Once all data is written to the file, the file must be closed using the command "close" and the variable pointing to the file. The syntax is as follows:

```
close $file_name
```

Control Structures

One form of control structures are loops. In order to automate samples with a time step, loops are needed to change to each new time. Loops can also be used for repeated tasks, incrementing naming convention, etc.

While-loops are a type of loop where only a condition is checked to determine if the lines within the loop should be executed. If a variable is being checked in the condition, this variable needs to be initialized before the loop or an error will occur. The while loop has the following syntax:

```
while {condition} {
        Code to be executed if condition is met
}
```

Usually the while-loop is used with the condition checking a numerical value, often with the value increasing or decreasing such as to complete a loop x number of times. In this case, the variable in the condition needs to be changed at the bottom or top of the loop.

```
Ex.   set x 1
      while {$x < 5} {
            Do something
            incr x                    note: "incr" increments x by 1
      }

      set y 10
      while {$y > 0} {
```

```
        Do something
        Set y [expr $y – 1]
    }
```

For-loops are a better option for numeric looping, as the loop declaration includes a variable declaration, condition and the next value for the variable. This eliminates the need to initialize and increment/decrement a variable in a numeric loop. The for-loop has the following syntax:

```
for {initialize_variable} {condition} {next_value} {
    Code to be executed if condition is met
}
```

```
Ex.  for {set n 1} {$n < 5} {incr n} {
        Do something
    }
```

Foreach-loops are good for using each item in a list of values, which can be an integer, float or string. This loop executes the instructions for each item in the list, and has the following syntax:

```
foreach name  {var1 var2 var3 …} {
    Do something
}
```

Or

```
set list_name {val1 val2 val3}
foreach name $list_name {
    Do something
}
```

The foreach-loop is also possible with words in a string:

```
Ex.  set string “One Two Three”
     foreach s $string {
        echo $s
     }
```

This will print “One”, “Two”, and “Three” each on a new line to the screen

In the case that either you need to exit a loop early or skip the remaining lines of code in the loop and continue, insert the commands “break” or “continue” at the desired line.

Another control structure are if-then conditions. These are conditional statements and allows for execution of a block of code if a condition is met. The “if” condition is always required, but the “elseif”

and “else” are optional, with the requirement that there can only be a single “if” and “else”. These structures have the structure:

```
if {condition1} {
        Do something
} elseif {condition2} {
        Do something
}
.
.
.
} else {
        Do something
}
```

In this structure, the conditions are checked sequentially and only the first block of code corresponding to a true condition is completed. In the event that no condition is met, the block of code in the “else” statement is completed (provided there is an else statement).

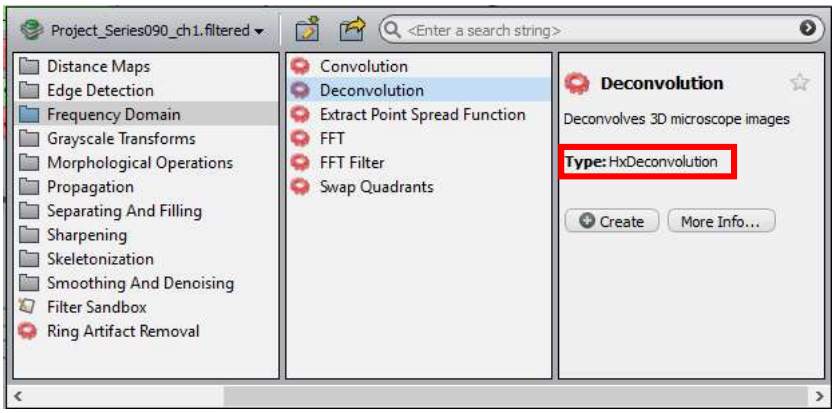
Ex.
```
set x 7
for {set y 0} {$y < 10} {incr y} {
        if {$y < $x} {
                echo “y is less than x”
        } elseif {$y > $x } {
                echo “y is greater than x”
        } else {
                echo “y is equal to x”
        }
}
```

Avizo Specific

The modules in Avizo can be controlled through the TCL console, which is beneficial for automating the image processing of samples.

Creating Modules

To create a module in Avizo, we use the command “create” and the type of module we want. The type can be found by right clicking on the image we want to apply the module to and selecting the desired module. There will be the type name near the above the “create” and “more info” buttons.



The type can also be found from an existing module by hovering your cursor over the collapse icon (white, square box with a horizontal line).

To create the module, use the following command:

create *type_name*

Ex. create HxDeconvolution

This will create a module with default naming (ex. “Deconvolution”, “Deconvolution 2”, …) for each version of the same module, but you can also customize the module name by using:

create *type_name “my custom name”*

Ex. create HxDeconvolution “Filter1”

Connecting Modules

Once a module is created, it needs to be connected to another image. This is done using the “connect” command followed by the name of the image it should be connected to. To do this you need the name of the module, the setting name (usually either “inputImage” or “data”) and the image that the module should be connected to.

An easy way to determine the setting name needed to connect the module is to first create the module and then in the console type

“module name” getState

This will list all the current settings for the module and should have a line of the form

"*module name*" data disconnect

Or

"*module name*" inputImage disconnect

```
>"Deconvolution" getState
create HxDeconvolution "Deconvolution"
"Deconvolution" setIconPosition -1183 -201
"Deconvolution" setVar "CustomHelp" {HxDeconvolution}
"Deconvolution" data disconnect
"Deconvolution" kernel disconnect
"Deconvolution" fire
"Deconvolution" borderWidth setMinMax 0 -16777216 16777216
"Deconvolution" borderWidth setValue 0 0
"Deconvolution" borderWidth setMinMax 1 -16777216 16777216
"Deconvolution" borderWidth setValue 1 0
```

To connect the module to an image, use the line above that matches and replace "disconnect" with connect and the name of the image, such as:

"*module_name*" data connect *"image_to_connect"*

Or

"*module_name*" inputImage connect *"image_to_connect"*

For a more robust command, use

"*module_name*" data connect ["*previous module name*" getResult]

Or

"*module_name*" inputImage connect ["*previous module name*" getResult]

which will connect the current module to the output image from a previous module. This is good if you can't predict the naming convention of the output images. However, the first set of commands should be used for the original file, as it is not an output from a module.

Changing Module Settings

Changing the settings in a module follows a similar syntax as creating a module, but can be more difficult to determine the commands needed.

A good habit to start is to use the command

"*module name*" fire

after connecting a module. This will update some of the settings related to the connected image such as the colour map values.

The easiest way to proceed in changing settings is to use the same command as before,

"*module name*" getState

which will bring up a list of the current settings in the form that is used to set them.

```
>"Deconvolution" getState
create HxDeconvolution "Deconvolution"
"Deconvolution" setIconPosition -1183 -201
"Deconvolution" setVar "CustomHelp" {HxDeconvolution}
"Deconvolution" data connect "Project_Series090_ch2"
"Deconvolution" kernel disconnect
"Deconvolution" fire
"Deconvolution" borderWidth setMinMax 0 -16777216 16777216
"Deconvolution" borderWidth setValue 0 0
"Deconvolution" borderWidth setMinMax 1 -16777216 16777216
"Deconvolution" borderWidth setValue 1 0
"Deconvolution" borderWidth setMinMax 2 -16777216 16777216
"Deconvolution" borderWidth setValue 2 0
"Deconvolution" iterations setMinMax 0 100
"Deconvolution" iterations setButtons 1
"Deconvolution" iterations setEditButton 1
"Deconvolution" iterations setIncrement 1
"Deconvolution" iterations setValue 10
"Deconvolution" iterations setSubMinMax 0 100
"Deconvolution" initialEstimate setValue 0
"Deconvolution" overrelaxation setValue 1
"Deconvolution" regularization setValue 0
"Deconvolution" penaltyWeight setMinMax 0 -3.40282346638529e+038 3.40282346638529e+038
"Deconvolution" penaltyWeight setValue 0 9.99999974737875e-005
"Deconvolution" method setValue 1
"Deconvolution" parameters setMinMax 0 -3.40282346638529e+038 3.40282346638529e+038
"Deconvolution" parameters setValue 0 1.39999997615814
"Deconvolution" parameters setMinMax 1 -3.40282346638529e+038 3.40282346638529e+038
"Deconvolution" parameters setValue 1 0.620000004768372
"Deconvolution" parameters setMinMax 2 -3.40282346638529e+038 3.40282346638529e+038
"Deconvolution" parameters setValue 2 1.51800000667572
"Deconvolution" mode setValue 1
"Deconvolution" applyTransformToResult 1
"Deconvolution" fire
"Deconvolution" setViewerMask 16383
"Deconvolution" setPickable 1
```

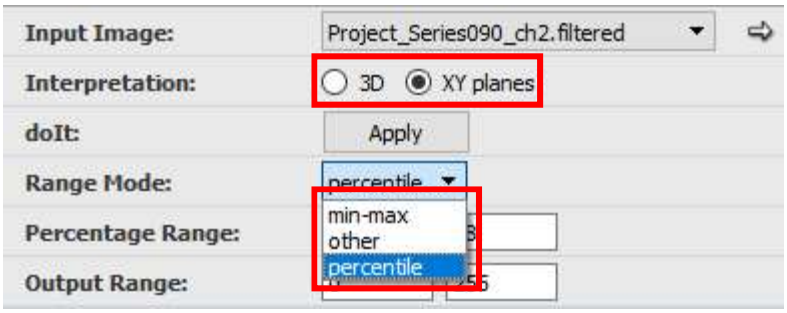
You can match the current setting value and command to the setting you want to change, and use the same command with a new value in your script.

Ex. If we want to change the iterations to 5,
"Deconvolution" iterations setValue 10
becomes
"Deconvolution" iterations setValue 5

Some settings are not a numeric value and thus have to be set slightly different. Drop down settings and interpretation selections are set according to their relative order. For example, the interpretation setting may have a selection for "3D" or "XY Planes". If "3D" is furthest left, this is option 0 and "XY Planes" is option 1. Then this is set with the command

"module name" interpretation setValue *option*

Drop down menus are set in a similar way with the selection nearest the top of the drop down as 0 and each successive option occurring below.

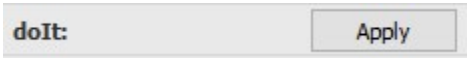


Lastly, to change the time step for an image, use the following commands:

"*time control name*" time setValue *time_step*
"*time control name*" fire

Applying a Module

Once the module is connected and the settings are correct, 2 commands are needed to apply the module. If you look in the properties window of your module, there should be a grey button that usually reads "Apply" or "DoIt".



If you cannot see the apply button, go to Edit-> Preferences-> Layout and make sure "Show 'DoIt' buttons" is selected. The name to the left of this button will be either "doIt" or "action", but can also be found by typing the module name into the console. This name will be used as a command in applying the module. The second command needed includes the module name followed by either "fire" or "compute" (in the case of the interactive thresholding module). The commands together should be:

"*module name*" doIt setValue 0
"*module name*" fire

Or

"*module name*" apply setValue 0
"*module name*" fire

Or

"*module name*" doIt setValue 0
"*module name*" compute

Snapshots and Saving Data

The image in the viewer window can be saved using the "snapshot" command in the following format:

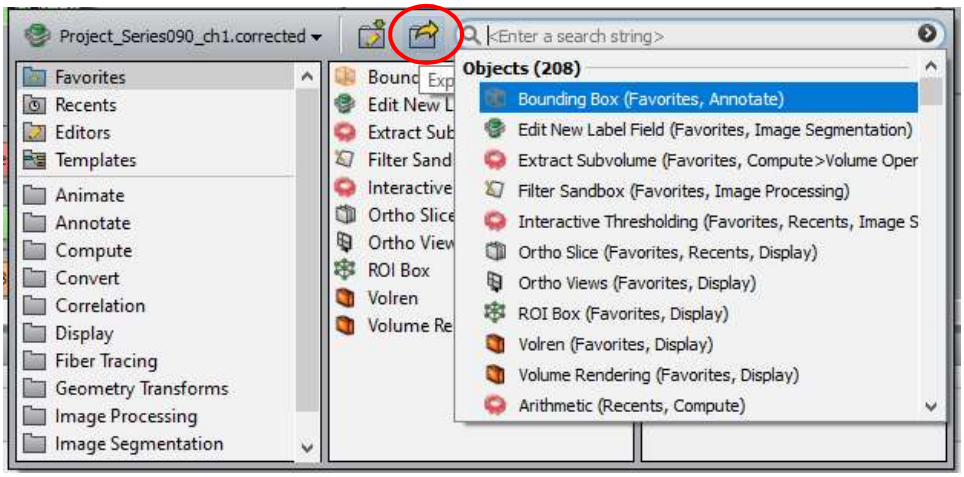
viewer *viewer_num* snapshot "*filename.format*"

Note that the viewer number is needed and is indexed from 0.

To turn on and off a module from view, use the following commands:

"*module name*" setViewerMask 16383 → turn on
"*module name*" setViewerMask 16382 → turn off

Some data and images can also be exported from Avizo using the "exportData" command. This is the same as right clicking on an image and clicking the export data icon.



To do so, use the command:

"*image name*" exportData "*file format*" "*filename.format*"

Where to Get Help

If more help is needed, there are a few ways to proceed.

If it is a problem with general TCL commands, there is a large amount of online documentation, including the following websites:

- https://www.tcl.tk/man/tcl/TclCmd/
- http://wiki.tcl.tk/

If there is a problem with controlling modules within Avizo, reading the user's guide and programmers manual may help. You can also type the module name in the console to get a list of setting names, or type part of a command followed by "help" which may give some information on how to use the command.

```
>["Area" getResult] getValue help
Usage: getValue [table:0] <col> <row>
```

Lastly, I have included two sample program below with some commonly used modules.

Sample Scripts

This script creates and applies the modules “Deconvolution”, “Normalize Grayscale”, and “Interactive Thresholding”. It then crops the image and saves each 2D slice as a separate image.

```
create HxDeconvolution
"Deconvolution" data connect "Project000_ch1"
"Deconvolution" fire
"Deconvolution" iterations setValue 10
"Deconvolution" method setValue 1                          set method to blind
"Deconvolution" parameters setValues 1.4 0.64 1.518
"Deconvolution" mode setValue 1                            set to confocal
"Deconvolution" action setValue 0
"Deconvolution" fire

create normalize
"Normalize Grayscale" inputImage connect ["Deconvolution" getResult]
"Normalize Grayscale" fire
"Normalize Grayscale" interpretation setValue 1            interpretation set to XY
"Normalize Grayscale" rangeMode setValue 2                 set range to percentile
"Normalize Grayscale" doIt setValue 0
"Normalize Grayscale" fire

create HxInteractiveThreshold
"Interactive Thresholding" data connect ["Normalize Grayscale" getResult]
"Interactive Thresholding" fire
"Interactive Thresholding" intensityRange setValue 0 80    lower threshold value
"Interactive Thresholding" intensityRange setValue 1 500   upper threshold value
"Interactive Thresholding" doIt setValue 0
"Interactive Thresholding" compute
"Interactive Thresholding" setViewerMask 16382             turn off viewer

[“Interactive Thresholding” getResult] crop 50 1000 50 1000 10 60      crop to xi xf yi yf zi zf

Export each slice as separate image. Will be named Z0.tiff, Z1.tiff, …
[“Interactive Thresholding” getResult] exportData “2D Tiff” “D:/Amaahzing/Sample0/Slices/Z.tiff”
```

For each time step (0-10), this script applies the modules “Deconvolution”, “Median Filter”, “Normalize Grayscale”, “Interactive Thresholding”, “Volume Fraction”, and “Generate Surface”. It then takes an image of the surface and adds the volume fraction value to a text file. This program is written assuming that each module is already created, but can be remedied by adding “create” commands for each of the modules before the for loop.

```
for {set i 0} {$i < 10} {incr i} {

        “Project001” time setValue $i                set time step
        “Project001” fire

        “Deconvolution” data connect “Project001_t0_ch1”
        “Deconvolution” fire
        "Deconvolution" iterations setValue 10
        "Deconvolution" method setValue 1
        "Deconvolution" parameters setValues 1.4 0.64 1.518
        "Deconvolution" mode setValue 1
        "Deconvolution" action setValue 0
        "Deconvolution" fire

        “Median Filter” inputImage connect [“Deconvolution” getResult]
        “Median Filter” fire
        “Median Filter” interpretation setValue 0
         “Median Filter” doIt setValue 0
        “Median Filter” fire

        "Normalize Grayscale" inputImage connect ["Median Filter" getResult]
        "Normalize Grayscale" fire
        "Normalize Grayscale" interpretation setValue 1
        "Normalize Grayscale" rangeMode setValue 2
        "Normalize Grayscale" doIt setValue 0
        "Normalize Grayscale" fire

        "Interactive Thresholding" data connect ["Normalize Grayscale" getResult]
        "Interactive Thresholding" fire
        "Interactive Thresholding" intensityRange setValue 0 65
        "Interactive Thresholding" intensityRange setValue 1 255
        "Interactive Thresholding" doIt setValue 0
        "Interactive Thresholding" compute
        "Interactive Thresholding" setViewerMask 16382

        "Volume Fraction" inputImage connect ["Interactive Thresholding" getResult]
        "Volume Fraction" interpretation setValue 0
        "Volume Fraction" doIt setValue 0
        "Volume Fraction" fire

        “Generate Surface” data connect ["Interactive Thresholding" getResult]
        "Generate Surface" action setValue 0
        "Generate Surface" fire

        “Surface View” data connect [“Generate Surface” getResult]
        "Surface View" setViewerMask 16383                    turn on viewer
```

```
        Save image of surface
        Viewer 0 snapshot "D:/Amaahzing/Sample1/Surfaces/${i}.tiff"

        Append volume fraction value to text file
        set $fo [open "D:/Amaahzing/Sample1/Result.txt" a]
        puts $fo "Volume Fraction for time $i"
        puts $fo [["Volume Fraction" getResult] getValue 1 0]        1 0 represents col and row of table
        close $fo

}
```