R: Creating Graphical User Interfaces (GUIs) using the tcltk

Introduction to Tcl

Adrian Waddell

University of Waterloo
Departement of Statistics and Actuarial Science

September 8, 2010

About these Slides

These slides were written on behalf of the Departement of Statistics and Actuarial Science at the University of Waterloo, Canada.

At the time of writing, the current software versions are

- GNU Emacs 23.1.1
- Eclipse SDK Version: 3.5.2
- R version 2.11.1
- ESS 5.11

You find an R source code file for these slides and more slides on our homepage.

About Tcl and Tk

Tcl (Tool Command Language) is a scripting language created by John Ousterhout at University of California around 1991. John Ousterhout also developed Tk, a platform independent graphical user interface (GUI) library, as an Tcl extension.

Tk gained popularity because many other programming languages such as Perl, Python, Ruby, Common Lisp and –for us of particular interest– R provide libraries to write Tk GUIs.

Peter Dalgaard, at University of Copenhagen, announced his `tcltk` package in 2001. Subsequently he announced some changes in the Rnews 2002, Vol. 3. However, he never wrote a complete documentation to his `tcltk` package. (I speculate that writing the documentation would have been more time intensive than writing the source code).

The absence of a complete `tcltk` library documentation makes the first steps into GUI creation with Tk in R a bit tough. We try to ease the initial learning curve with these slides.

Tk and R

Fortunately some understanding of Tcl and its naive Tk extension will be sufficient to make use of the `tcltk` package. This further allows you to make use of the excellent www.tcl.tk web page to get full Tcl/Tk documentation and plenty of examples.

The syntax of Tcl differs from that of R quite a lot. The `tcltk` library provides a whole set of Tk wrapper functions such that the user can integrate GUIs easily in R style into its R source code. Knowing Tcl and native Tk will however help to customize some GUIs for ones own special needs, as they open the door to the complete Tcl/Tk command set.

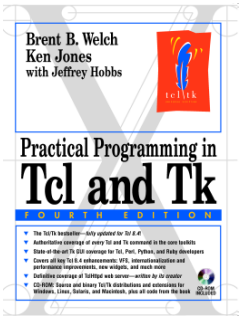
If you want an overview or a first impression of Tks capability, browse through this web page with examples, maintained by Philippe Grosjean, and also run some of the demos in R:

```
> library(tcltk)
> demo(package = "tcltk")
> demo(tkcanvas)
```

What you will learn from these Slides

As already mentioned, we will first go over some of the Tcl basics. We than cover the conceptual part of the Tk library, such as its syntax and some geometry managers. However individual widgets (graphical user interface elements) such as for example listboxes, scrollbars or sliders won't be discussed in detail because it would be too much to cover in these slides.

You eventually will need to get a good book about Tcl and Tk if you plan to frequently design GUIs. An excellent book is the following:



Practical Programming in Tcl and Tk
Brent B. Welch (Prentice Hall PTR)
A copy of an early draft can be downloaded for free from here. However getting a current edition (4th at this time) is highly recommended.

Brent B. Welch kindly gave me the permission to adapt some of his examples (the pack geometry manager) directly for these slides.

More advanced Tk programmers might want to take a look at this book.

"Hello World" in Tcl

Tcl commands, the equivalent of functions in R, get invoked as follows

```
commandName arg1 arg2 arg3 ...
```

The Tcl equivalent to the R prompt `>` is the Tcl shell (tclsh) promt `%`. R comes with Tcl/Tk by default and provides with the `.Tcl` function a basic interface between R and Tcl.

```
> library(tcltk)
> .Tcl('puts stdout "Hello World."')
```

This code example invokes the command `puts` with the arguments `stdout` and the character string `"Hello World."`.

Using the Tcl shell instead you can get the same output with the following code:

```
% puts stdout "Hello World."
```

Hence, in what follows, all the "pure" Tcl code can be executed in R by wrapping a `.Tcl('` and `')` around it.

You can check which Tcl version your R connects to by entering to R

```
> .Tcl('puts stdout $tcl_version')
```

Comments and Multiple Commands

Separation of commands and adding comments are done in Tcl the same way as in R with the only difference that Tcl expects a comment to be at the beginning of a command.

Hence, command separations are done with `;`

```
puts stdout "Hello"; puts stdout "World"
```

and comments are either at the beginning of a new line or after a `;`

```
# In the end, there are some similarities between R and Tcl
set stdout "BUT"    ;# but there are also many differences
```

In fact,

```
puts stdout Hello
```

would have also worked, as `Hello` appears to the Tcl interpreter as a single argument. However

```
puts stdout Hello World
```

Does not work because the Tcl things that `Hello World` are two arguments (separated by a white-space).

Variables and Math

- The `set` command assigns a value to a variable

```
set x 2
```

- The variable `x` can subsequently be accessed with `$x`, for example

```
puts stdout "Variable x is $x."     ;# or
set y $x
```

  In the Tcl world, accessing is called *variable substitution*.

- Variables are deleted with the `unset` command

```
unset x
```

- You can also do math with the variable `x` and `y`

```
expr $x+5*$y
```

  note that the math syntaxt is the same as used in C

```
set x [expr pow($y,2)]
```

- `[]` is used for a nested command. The previous example was an example of a command substitution.

More on Math in Tcl

The precision of math operations is by default set to 12 significant digits. You can change this by creating the variable `tcl_precision`:

```
set tcl_precision 5
```

When calling the `expr` command: put its argument into curly braces `{}` in order for `expr` to make all the substitutions in the expression. For example for

```
expr $x+5*$y
```

the Tcl interpreter will first substitute `$x+5*$y` and pass the *string!* to the `expr` command, which in turn has to converse it back to numeric values.

Better is to give the unsubstituted string `$x+5*$y` directly to `expr`, and let `expr` do the substitutions. Doing so results in a gain of speed and precision.

You can prevent the Tcl interpreter substitutions by putting the arguments in curly braces `{}`, for example

```
expr {$x+5*$y}
```

would be the best way to perform the computation of `$x+5*$y`.

Grouping

As we saw in some previous examples, we can either use *double quotes* to group words into one argument,

```
set x "Farewell, fair cruelty."
```

or curly braces, with the only difference that curly braces don't allow for substitution, try

```
puts stdout {He said \"$x\"}
```

and

```
puts stdout "He said \"$x\""
```

where the *backslash substitution* in the last code snippet works in the `tclsh` but not as an `.Tcl` argument, use

```
puts stdout "He said \'$x\'\"
```

instead.

Hence, using braces allows you to induce a substitution at a later time

```
set y {He said \"$x\"}
puts stdout [subst $y]
```

Procedures

Tcl procedures are the equivalent of R functions. The `proc` command is used to define procedures

```
proc procname arglist body
```

Note that, as opposed in R, procedure names do not conflict with variable names.

```
proc bmi {height weight} {
  set bmi [expr $weight/pow($height,2)]
  return $bmi
}
```

and you can call it with

```
bmi 1.82 75
```

As R functions do, Tcl procedures return their last expression evaluated.

The curly braces don't perform any different in the procedure syntax: they group arguments (space) or commands (new lines) and delay the substitution until the procedure is called.

Strings

"Within Tcl, the basic philosophy is that everything is a string. [...]. Other data structures (such as `proc`, `dict`, `list`, `handle`) are built on top of this fundamental assumption." [see wiki.tcl.tk]

The Tcl `string` command provides a collection of string operations.

- Its first argument determines the operation to perform [tcl.tk man]

A few examples are:

```
string length {Hello World!}

set col [string range {green world} 0 4]
string equal green $col

set x 5.4
string is integer $x
```

The append command is to faster than double quotes:

```
set x "$x years of work"    ;# is really the same as
append x " was waisted"
```

Lists

Lists are like vectors in R one dimensional arrays. Lists are created with the `list` command

```
set names [list Jeff Bob Lea Tiffany]
```

Here some basic list manipulations, read the online manual on lists, join and split for more information.

```
lindex $names 2                ;# access third element

lrange $names [list 1 3]       ;# access 2nd to 4th element

lset names 1 Bobby             ;# replace element

llength $names                 ;# length of the list

lappend names Paul             ;# append one element

concat $names [list Emilio Berta]     ;# combine two lists

lsearch -all -inline $names *ff*      ;# search for ff
```

where the asterisk ($\star$) in the last example stands for any string.

Arrays

A Tcl array is similar to the *list* data structure in R. Indices of arrays are keys (strings). See the tcl manual and wiki entry.

The basic syntax is

```
set array("key") "your string"    ;# define an array element
set $array("key")                 ;# access an element
```

To avoid possible errors, do not use space characters in the *key*, as parentheses do not group.

Arrays are one dimensional, but nothing stops you from using `1,1` and `1,2` etc. as the key string.

Control Flow Statements

Tcl provides

- looping commands `for`, `foreach` and `while`
- conditional commands `if` and `switch`

Follow the links to get the documentation and some examples for each of them.

To compare strings in a conditional statement, you may use `eq` and `ne` instead of the `string equal` command

```
if{[string equal $str1 $str2]} { ... }
```

is the same as

```
if{$str1 eq $str2} { ... }
```

Note: "You can control loop execution with the `break` and `continue` commands. The `break` command causes immediate exit from a loop, while the `continue` command causes the loop to continue with the next iteration." [Brent Welch]

Scoping and Namespaces

In Tcl, procedures do not see variables defined outside the procedure. You can, however, access them with the `global` and `upvar` commands.

Apart from calling procedures, it is possible to create namespaces with the `namespace` command. Like the `string` command, `namespace` provides a collection of operations, defined by its first argument.

Similar as in R, procedures can be accessed via the double colon `::`. From the online documentation:

```
namespace eval counter {
    namespace export bump      ;# export procedure
    variable num 0             ;# create and initialize num
        variable

    proc bump {} {
        variable num
        incr num               ;# num = num + 1
    }
}
```

So, you can increment `num` in the `counter` namespace with

```
counter::bump                  ;# increment num
```

Events

> *A single event loop permeates everything in Tcl/Tk programs that deal with I/O. When things happen on files, sockets, the GUI, timers or other input sources, events fire and Tcl callbacks are invoked, simple as that. [tcl.tk]*

Callbacks can be triggered by button clicks in GUIs, keyboard input or after a certain time delay, to name a few.

- Graphical user interfaces (GUIs) work with event binding. We will discuss them in the appropriate Tk section.
- The `after` command is used to delay the execution of a command to sometime in the future.
- The `vwait` command enters the *event loop* until a predefined variable gets modified.
- Idle events are processed when Tcl has nothing to do. They are registered with the `after idle` command.

More on the Tcl/Tk API for R

So far, we have used the `.Tcl` function in R to interact with the Tcl interpreter. There are some other low-level Tcl/Tk interface functions in R, see the R documentation with

```
> ?TclInterface
```

A more native feel to invoke Tcl procedures in R gives the `tcl` function. For example

```
> bar <- "x"
> tcl("set", bar, 1:5)
```

assigns the Tcl variable `x` the list (vector) `1 2 3 4 5`. Note how R converts its data structure –here a vector– to a Tcl list.

Tcl variables in R

You can of course access every Tcl variable in R via the `.Tcl` or `tcl` function. It is, however, possible to create an R variable which links to a Tcl variable

```
> x <- tclVar("3")
```

This creates the R variable `x` which links to a global Tcl variable called `RTcl1`, where the number gets for every new R/Tcl variable incremented

```
> .Tcl('set Rtcl1')
```

You can access the information from within R with

```
> tclvalue(x)
```

or change it with

```
> tclvalue(x) <- "3.26456"
```

Remember, every Tcl variable holds a string as its base type! Hence if you need to use a Tcl variable holding a integer within an R routine, use

```
> as.numeric(tclvalue(x))
```

Callbacks

Whenever you want to invoke an R function from Tcl, you need to get the hex-address of the R function. Say your function of interest is

```
foo <- function(a,b) {
  cat(paste("you entered: a=",a," and b=",b,'\n', sep=''))
}
```

You can get the Tcl command to invoke the function as follows

```
> .Tcl.callback(foo)
[1] "R_call 0x22b88a0 %a %b"
```

So

```
> .Tcl('R_call 0x22b88a0 3 4')
you entered: a=3 and b=4
<Tcl>
```

As to my knowledge, you can not use returned values in Tcl.

Tck/Tk Packages and Source files

Sometimes it is easier, and/or faster, to write your program completely in native Tcl code rather than using the wrapper functions provided by the `tcltk` package. If you want the Tcl program to work within R, you can write an interface with the commands presented in this chapter and load the Tcl procedures in R as follows:

```
> .Tcl('source myTclSourceFile.tcl')
```

You can load Tcl packages either with the `tclRequire` function form within R or with the Tcl command

```
% package require Img
```

if you were to load the `Img` package.

Emacs allows you to execute Tcl code interactively line by line. Another good tool which supports command completions with the *TAB* key is a program (tk extension) called tkcon.