cecam
Centre Européen de Calcul Atomique et Moléculaire

SimTech
Cluster of Excellence

Universität Stuttgart

Institute for Computational Physics

ESPResSo Summer School 2012

Introduction to Tcl

Pedro A. Sánchez

Institute for **C**omputational **P**hysics

Allmandring 3
D-70569 Stuttgart
Germany

Outline

- History, Characteristics, Online resources, Getting things running
- Variables, grouping and nested commands
- Math expressions
- Control structures

Hands on!

- User defined commands

Hands on!

- Lists and Arrays

Hands on!

- Working with files, command line arguments, modularization

Hands on!

History

- “Tool command language”, pronounced “tickle” or “tee-see-ell”
- John Ousterhout, Berkley, 1988
- Originally invented for GUI programming (Tcl/Tk)
- Very successful language in the 1990s, adopted by many companies
- Not very active and popular anymore
- Some scientific programs still use Tcl/Tk, e.g. VMD and NAMD
- … but most are slowly switching to Python...

Characteristics

- Interpreted scripting language, cross-platform (available almost everywhere), originally (and mainly used as) procedural
- Motto: “Radically simple”. Simple syntax
- No data types: all data treated as strings
- All operations are commands (=functions), including control structures
- Dynamic: everything can be (re-)defined easily, including source code
- Simple C-API, easy to extend and embed
- Free, open-source (BSD license)
- Current version 8.5.12 (July 27, 2012)

Online resources

- Huge documentation and resources at the official website: http://www.tcl.tk
    - http://wiki.tcl.tk/
    - Built-in commands quick reference: http://www.tcl.tk/man/tcl8.5/TclCmd/contents.htm
    - Complete tutorial: http://www.tcl.tk/man/tcl/tutorial/tcltutorial.html
- Nice interactive offline tutorial for self-learning, written in Tcl/Tk: http://www.msen.com/~clif/TclTutor.html

Getting things running...

- Interactive consoles:
  - Standard interpreter: tclsh
  - Improved console: tkcon
    - http://tkcon.sourceforge.net/
- Script files:
  - Usual extension: *.tcl
  - Run from command line:

```
$>tclsh myNiceScript.tcl
```

  - Executable scripts: prepend script with

```
#!/usr/bin/tclsh
```

Hello world!

- General syntax:

```
command argument1 argument2 ...
```

- Commands end with newline or semicolon `;`
- `" "` or `{}` used to group arguments
- Arguments are represented as **strings**
- Comments start with `#`

```
# This is a comment
puts "Hello World!"
puts "This is line 1"; puts "this is line 2"
puts "Hello, World - In quotes" ;# This is a comment
puts "Hello, World; - semicolon inside the quotes"
puts {Hello, World – in Braces}
puts HelloWorld
puts {Bad syntax example} # *Error* no semicolon!
```

Variables

- Assignement command: `set`

```
set variableName value
```

- Variable substitution: before a command is executed all variables, referenced as `$variableName`, are substituted for its value
- Backslash `\` prevents subtitution of the next character. Usual backslashed codes (“backslash-sequences”) exist `\n`, `\t`, ...
- Unset variables are reported

```
set myMessage "Hello World!"
puts $myMessage
set a 1.0
puts $a+$a
puts $a\n$a
puts \$a
puts $unknownVar
```

Variable substitution and argument grouping

- Argument grouping via `""`:
  - Variable substitution and backslash-sequences work
  - Use for strings
- Argument grouping via `{}`:
  - No substitution nor backslash-sequences
  - Use for code blocks

```
set myMessage "Hello World!"
puts "Say $myMessage\nNext line”
puts {Say $myMessage\nNext line}
set myFullMessage "Say $myMessage\nNext line”
puts $myFullMessage
```

Nested commands

- Command substitution: strings within square brackets [ ] are evaluated as commands
- Variable substitution works within command substitution
- Command substitution works within quotes, not within braces

```
set y [set x "def"] ;# command set returns the assigned value

set x "def"
set z [set y $x]

set z "[set x {This is a string within braces within quotes}]"

set z {[set x "This is a string within quotes within braces"]}
```

Math: expression evaluation

- Mathematical operations computed with the command `expr`
- Expressions mostly like C operators and mathematical functions: `+`, `-`, `*`, `/`, `%`, `pow`, `sin`, `cos`, ...

```
puts "1+1”
puts 1+1
puts [expr 1+1]
puts [expr "1+1”]

puts [expr 1/2]
puts [expr 1./2]

set x 2
puts "$x plus $x is [expr $x+$x]"
puts "The square root of $x is [expr sqrt($x)]"

puts [expr pow($x,2)]
puts [expr ($x+1) % 2]
```

Math: type conversion and random numbers

- Since all data is treated as a string, numbers should be transformed to and from strings → slow numerics in Tcl!!!!
- Explicit type conversions: `abs`, `int`, `double`, `round`
- Tcl provides a pseudo-random number generator: `rand()`, `srand()`

```
puts [expr double(1)]

puts [expr rand()] ;# pseudo-random number (0., 1.)

expr srand(1) ;# set seed for a reproducible sequence

expr rand()
```

Control structures: conditionals

- The `if` command:

```
if expr1 then body1 elseif expr2 then body2 ... else bodyN
```

- The words *then* and *else* are optional
- The test expressions following the word `if` are evaluated as in the `expr` command

```
set x 1
if {$x == 1} {puts "x is 1"} else {puts "x is not 1"}

# mind the spaces between arguments!!!
if {$x == 1}{puts "x is 1"}

if {$x == 1} { ;# this is more readable in scripts
    puts "x is 1"
} else {
    puts "x is not 1"
}
```

Control structures: loops

- The `while` command:

```
while test body
```

- The `for` command:

```
for start test next body
```

- The command `break` breaks a loop. The command `incr` increments the integer value of a variable

```
set i 0
while {$i < 3} {puts $i; incr i}

for {set i 0} {$i<3} {incr i} {puts $i}

for {set i 0} {$i<3} {incr i} {
   puts $i
}
```

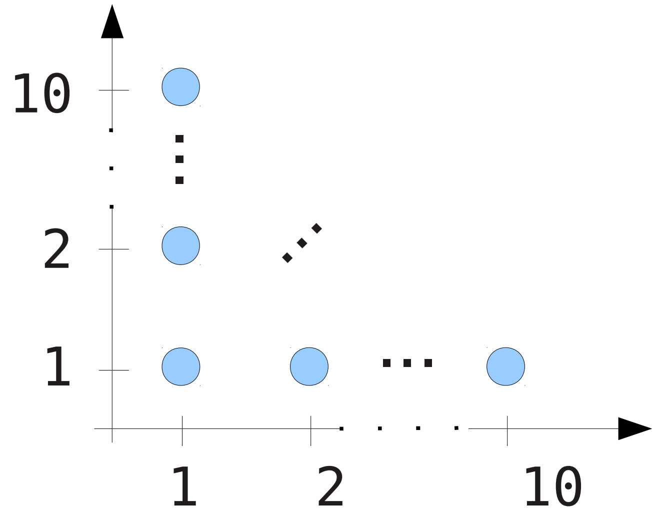
Hands on!

- Write a tcl script to calculate the center of mass of this system:
  - 100 point particles

    in a square x-y lattice:

    $\{(X, Y)\} = \{(1, 1); (1, 2); \ldots$

    $\ldots ; (10, 9); (10, 10)\}$



  - Mass depends on the product of the coordinates:
    - Particles with even product $X*Y$ have mass 2.0 (“even mass”)
    - Particles with odd product $X*Y$ have mass 1.0 (“odd mass”)

Adding new commands

- Command `proc` creates a new command

```
proc commandName arguments body
```

- All variables in body are local (including arguments) except those explicitly declared global with `global` or `upvar`
- The new command returns to the caller a value optionally specified with `return` or the output of the last command found within body by default

```
set myglobal "global"; set othervar "other"
proc myProc {arg1 {arg2 "default"}} {
   global myglobal;
   puts "arg1 is $arg1"; puts "arg2 is $arg2"
   puts "Global var is $myglobal"; return "returned"
}

set result [myProc "first"]
```

Pass-by-reference to procs

- Pass-by-reference of variables to commands is emulated with `upvar`:

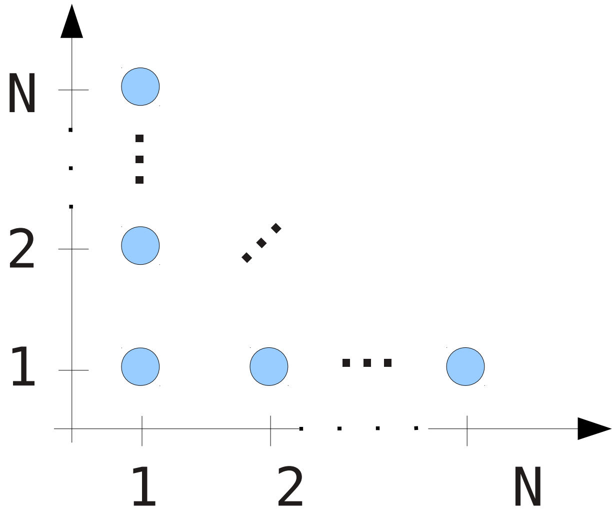
```
proc myIncr {arg1 {arg2 1}} {
   upvar $arg1 res
   set res [expr res + $arg2]
   return $res
}

set a 1
puts [myIncr a]
```

Hands on!

- Rewrite your script using a command definition:
  - Write a command to calculate the COM of the x-y square lattice system for NxN particles and arbitrary “even” and “odd masses”
    - Particles with even product

      X*Y have “even mass”
    - Particles with odd product

      X*Y have “odd mass”



Lists

- A list is just an ordered collection of data, is the basic data structure in Tcl. Lists are strings, can be defined in many ways
- Data items can be accessed and extended with list commands: `lindex`, `foreach`, `lappend`, `llength`
- Lists can be nested

```
set myList "1 2 3", set myList {1 2 3}
puts [lindex $myList 2]; puts [llength $myList]
foreach j $myList {puts $j}
lappend myList 4 5 6; puts $myList

set myEmptyList {} ; lappend myEmptyList {Not empty anymore!!}

set Nested {{1 2 3} {4 5 6}};set Nested [list "1 2 3" "4 5 6"]
puts [lindex $myNestedList 0 1]
```

Arrays

- Associative arrays (lists of key-value pairs) can be defined either by putting the key within parentheses `()`:

```
set myArray(1) One
puts $myArray(1)
```

- or from a list of key-value pairs using the `array` command:

```
array set myArray [list 1 One 2 Two 3 Three]
puts $myArray(1)
```

- Multidimensional arrays can be emulated using smart strings as keys:

```
set myArray(1,1) {One One}
puts $myArray(1,1)
```

Hands on!

- Rewrite your script using lists and/or arrays
  - Write two separated commands, one to generate the positions and masses of the x-y lattice system and the other one to calculate the COM of a collection of arbitrary positions and masses passed as arguments

Working with files

- Get a I/O channel to access a file:

```
open fileName access
```

where `access` sets the channel for reading (default), `"r"`, writing, `"w"` or append `"a"`. Read/write data with commands `gets`/`puts`. Close channel with `close`

- Parse lines of data read from files with command `split`

```
set fp [open "myfile.dat" "w"]
puts $fp "1,2\n3,4\n"
close $fp

set fp [open "| cat /proc/cpuinfo"]; #open a pipe
puts [gets $fp]; #read a line of data

set fp [open "myfile.dat" "r"]
set data [split [gets $fp] ","]; #split using “,” as delimiter
```

Command line arguments

- Number of command line arguments in global variable `$argc`
- Name of the script in global variable `$argv0`
- List of command line arguments in global variable `$argv`

```
puts "There are $argc arguments to this script"
puts "The name of this script is $argv0"
if {$argc > 0} {
   puts "Arguments are $argv"
}
```

Modularization

- The `source` command loads and executes a Tcl script:

```
source scriptName
```

- This allows to split a program in different files, useful for code reutilization and maintenance

Hands on!

- Write a more flexible and modularized version of the COM calculation program:
  - Split the commands for the generation of the x-y lattice system and the calculation of the COM into separate scripts
  - Write a main script which loads the splitted command scripts, generates the system according to command line arguments and makes the calculation
  - Alternatively, make the lattice generator script an independent program that works with command line arguments and writes the system data into a file. Make the main COM script to load and parse the data file for the calculation

Thank you!