Manual 08 for MD++

Tcl Language as MD++ Input

Keonwook Kang and Wei Cai

February 7, 2007

1 Tcl Basics

Tcl (pronounced like "tickle") stands for "Tool Command Language". It is a powerful and popular scripting language that has a very simple grammar. It was invented by Dr. John Ousterhout then employed by Sun Microsystems.[1] You can find more information and reference materials about Tcl at `http://www.tcl.tk/`. MD++ can take Tcl scripts as input files (suffix `.tcl`), in which we can use `if` statements, `for` loops, and define user procedures, to better control the flow of our simulations. MD++ understands all Tcl grammar and functions, plus three additional functions: `MD++`, `MD++_Get` and `MD++_Set`, which we will explain later. In this section, we give a brief introduction to Tcl through a few examples that illustrates the most common use of Tcl in MD++. You may skip this section if you are already familiar with Tcl.

1.1 Hello World !

Tcl is usually installed by default on any Linux system. To check whether you have Tcl on your computer, type

```
$ which tclsh
```

The shell command `which` prints out the location of the `tclsh`, the Tcl interpreter. The location is usually

```
/usr/bin/tclsh
```

As an example, let us create a simple Tcl script, `hello.tcl`, which reads,

```
#!/usr/bin/tclsh
puts "hello, world!!!"
puts "Print special characters: \$, \\, \[, \""
```

[1]Dr. Ousterhout is Professor of Computer Science at U. C. Berkeley and member of the National Academy of Engineering.

We can change it to an executable file and run it by typing

```
   $ chmod u+x hello.tcl
   $ ./hello.tcl
```

You should see the following printout on your screen.

```
   hello, world!!!
   Print special characters:  $, \, [, "
```

Usually every line in a Tcl script begins with a command, such as `puts`, followed by the arguments for the command, such as the string to be printed on the screen. The backslash \ needs to be used if we want to print special characters such as `$`. Otherwise, Tcl will interprete them differently. For example, Tcl would replace `$x` by the value of variable `x`.

1.2 Variables and numerical calculations

The following file, `yis.tcl`, illustrates how to set value to a variable and perform simple calculations.

```
#!/usr/bin/tclsh
set x 1
set y [expr 1.0/(99+$x)]
set z [expr 1/(99+$x)]
puts "y is $y and z is $z when x=$x."
puts "y is [format %20.13e $y]\
      and z is [format %5d $z] when x=$x."
```

In Tcl, we do not need to declare a variable before we first assign its value. Here we first set variable `x` to be 1. To be precise, the value of `x` is the character “1” but not the numerical value 1. To perform numerical calculations, we have to use the `expr` command, which will interprete `$x` as the numerical value 1. The square brackets, `[` and `]`, are special characters in Tcl. Tcl will execute the command within the square brackets and replace it with the return value of the command. In this example, `1.0/(99+$x)` is the argument for command `expr`. The return value of this command then becomes the second argument of the command `set` (the first argument of the command is a new variable `y`). Run this script by typing

```
   $ chmod +x yis.tcl
   $ ./yis.tcl
```

and you should see the following printout on your screen,

```
y is 0.01 and z is 0 when x=1.
y is  1.0000000000000e-02 and z is     0 when x=1.
```

The command `format` converts a variable to a string according to the format specified in its first argument. It is very similar to the `sprintf` function in ANSI-C. Notice that the values of `y` and `z` are different because the former is the result of floating point operations and the latter is the result of integer operations.

1.3 `while` and `for` loops

Consider the following script, `ysare.tcl`.

```
#!/usr/bin/tclsh
set x 1
while {$x < 100} {
  set y [expr 1.0/(99+$x)]
  puts "y = [format %20.13e $y] at x = [format %5d $x]"
  set x [expr $x + 1]
}
```

Run this script by

```
   $ chmod +x ysare.tcl
   $ ./ysare.tcl
```

The output should be

```
y =  1.0000000000000e-02 at x =     1
y =  9.9009900990099e-03 at x =     2
y =  9.8039215686275e-03 at x =     3
           :
           :
y =  5.1020408163265e-03 at x =    97
y =  5.0761421319797e-03 at x =    98
y =  5.0505050505051e-03 at x =    99
```

The `while` command takes two arguments, each enclosed by a pair of braces { and }, which are special characters of Tcl. The last line within the `while` loop can be replaced by `incr x 1`. Command `incr` can only be used to increment a variable by an integer. We can also replace the `while` loop by the following `for` loop.

```
#!/usr/bin/tclsh
for {set x 1} {$x < 100} {incr x} {
  set y [expr 1.0/(99+$x)]
  puts "y = [format %20.13e $y] at x = [format %5d $x]"
}
```

The `for` command has four arguments, each enclosed by a pair of braces. The white spaces between } and the following { are mandatory. If, for example, we write

```
for {set x 1}{$x < 100} {incr x} {
```

Then "`set x 1}{$x < 100`" will be treated as the first argument and the `for` loop will not run correctly.

1.4 `if` and `switch` flow control

The following script, `ysareif.tcl`, illustrates how to pass arguments to the script from the command line, and how to use conditional statements such as `if` and `which` to direct the flow of the program.

```
#!/usr/bin/tclsh
puts "The name of this script is $argv0"
if {$argc > 0} {
  puts "There are $argc arguments to this script."
  puts "The argument is : $argv"
  set flag $argv
} else {
  puts "There are no argument to this script."
  puts "Default argument = \"G(radual)\" is used."
  set flag "G"
}

switch $flag {
  G {
    puts "In case of argument = \"G(radual)\","
    for {set x 1} {$x < 100} {incr x} {
      set y [expr 1.0/(99+$x)]
      puts "y = [format %20.13e $y] at x = [format %5d $x]"
    }
  }
  I {
    puts "In case of argument = \"I(nstantaneous)\","
    for {set x 1} {$x < 100} {incr x} {
      set y [expr $x/100.0]
      puts "y = [format %20.13e $y] at x = [format %5d $x]"
    }
  }
  default {
    puts "No action specified for the given argument"
  }
}
```

You can run the script with argument "G"

```
   $ ./ysareif.tcl G
```

or with argument “I”

```
$ ./ysareif.tcl I
```

or other (more than one) arguments to see the effects. More tutorial materials can be found at `http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html`.

2 Using Tcl in MD++

MD++ will interprete the input file according the Tcl syntax if it has `.tcl` as the extension name. In comparison, MD++ will treat the input file as a regular script file (syntax described in Manuals 01-04) if the extension name is `.script`. In a Tcl input file, we can execute an MD++ command by putting `MD++` at the beginning of the line. Tcl will treat the remaining part of the line as the arguments of the command `MD++`, which will call MD++ to interprete its arguments. We can also pass multiple lines of MD++ script as the argument of the `MD++` command, by enclosing them with braces { and }. For example, the `si.script` file discussed in Manual 02 can be translated to the following Tcl file, `si.tcl`, and MD++ will perform identical operations, i.e. creating a perfect crystal and visualizing it, when receiving both files as inputs.

```
# -*-shell-script-*-
MD++ setnolog
MD++ setoverwrite
MD++ dirname = runs/si-example

#------------------------------------------------------------
#Create Perfect Lattice Configuration
#
MD++  element0 = Si
MD++  crystalstructure = diamond-cubic
MD++  latticeconst = 5.4309529817532409 #(A) for Si
MD++  {
  latticesize = [ 1 0 0 2
                  0 1 0 2
                  0 0 1 3 ]
}
MD++  makecrystal  writecn

#------------------------------------------------------------
#Plot Configuration
#
MD++  atomradius = 0.67 bondradius = 0.3 bondlength = 2.8285
MD++  atomcolor = orange highlightcolor = purple
MD++  bondcolor = red backgroundcolor = white
MD++  plotfreq = 10  rotateangles = \[ 0 0 0 1.25 \]
```

```
MD++  openwin alloccolors rotate saverot eval plot
MD++  sleep quit
```

Because square brackets, `[` and `]`, are special characters in Tcl syntax, we need to precede them by backslash, i.e. \\`[` and \\`]`, or enclose them by braces, { and }, before passing them as arguments to the `MD++` command. The latest version of MD++ parses Tcl syntax by default. When the input file has extension name `.script`, it automatically creates a temporary `.tcl` file, which starts with `MD++` { and ends with }, and includes the original `.script` file in between. MD++ then proceeds to parse the temporary `.tcl` file.

2.1 Grouping

Tcl allows us to define our own functions (subroutines, or procedures). We can take advantage of this feature to better organize the MD++ input file. The syntax for procedure definition is

`proc` *name* { *args* } { *body* }

Consider the following Tcl input file `si2.tcl`.

```
# -*-shell-script-*-
#******************************************
# Definition of procedures
#******************************************
proc initmd { n } { MD++ {
setnolog
setoverwrite
dirname = runs/si-example-$n
zipfiles = 1              # zip output files
NIC = 200 NNM = 200
#-------------------------------------------
# Create Perfect Lattice Configuration
#
element0 = Si
crystalstructure = diamond-cubic
latticeconst = 5.430949821 #(A) for Si
latticesize = [ 1 0 0 2
                0 1 0 2
                0 0 1 3 ]
} }

#-------------------------------------------
proc openwindow { } { MD++ {
# Plot Configuration
#
atomradius = 0.67 bondradius = 0.3 bondlength = 2.8285 #for Si
atomcolor = orange highlightcolor = purple
bondcolor = red backgroundcolor = gray70
plotfreq = 10  rotateangles = [ 0 0 0 1.25 ]
openwin alloccolors rotate saverot eval plot
} }

#*****************************************
# Main program starts here
#*****************************************
initmd 1
MD++ makecrystal writecn
openwindow
MD++ sleep quit
```

This input file also asks MD++ to create a perfect Si crystal and visualize it. The main program is contained in the last 4 lines of this file. The first line will lead to the creation of the directory `runs/si-example-1` because the argument for command `initmd` is `1`. Using `proc` allows us to give more structure to the input file and make it easy to read and manage.

2.2 `MD++_Get` and `MD++_Set`

`MD++_Get` retrieves the value of an MD++ variable and pass it to Tcl, and `MD++_Set` allows Tcl to assign the value of an MD++ variable. For example,

```
puts "totalsteps = [ MD++_Get totalsteps ]
```

prints the value of MD++ variable `totalsteps` on the screen (which is not possible to do within the original script file). To assign value to the variable `totalsteps`, we can write, e.g.

```
MD++_Set totalsteps 1000
```

We can also do the same thing by writting

```
MD++ totalsteps = 1000
```

Hence there is some redundancy here. We can also specify the individual value of an array by, e.g.,

```
MD++_Set EPOT_IND(10) 0.1
```

or

```
MD++ EPOT_IND(10) = 0.1
```

We can also reference individual elements of an array in `MD++_Get` in the same way. With `MD++_Get` we can retrieve internal variables of MD++, such as potential energy `EPOT`, without having to grep the `A.log` file. The following Tcl file (`si-bulk.tcl`) performs the calculations needed to compute the bulk modulus (see Manual 03) and prints out the potential energy as a function of the lattice constant. The results are printed out to the screen by a user-defined function `MD++_PrintVar`, which uses `MD++_Get` to fetch the value of the variable of interest.

```
# -*-shell-script-*-
#******************************************
# Definition of procedures
#******************************************
proc initmd { } { MD++ {
setnolog
setoverwrite
dirname = runs/si-example
zipfiles = 1   # zip output files
#
NIC = 200 NNM = 200
#------------------------------------------
# Create Perfect Lattice Configuration
#
element0 = Si
crystalstructure = diamond-cubic
latticeconst = 5.430949821 #(A) for Si
latticesize = [ 1 0 0 4
                0 1 0 4
                0 0 1 4 ]
} }

proc MD++_PrintVar { name {unit \?} {fmt %11.4e} } {
    puts "$name\t= [format $fmt [MD++_Get  $name]] (in $unit)"
}

#******************************************
# Main program starts here
#******************************************
initmd
MD++ makecrystal writecn

MD++ saveH

for {set x 997} {$x <= 1003} {incr x} {
  set y [expr $x/1000.0]
  MD++ restoreH input = \[ $y \] scaleH
```

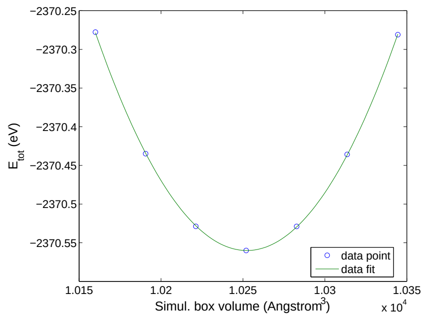

Figure 1: Potential energy E_{pot} vs. simulation box volume Ω. Data points are fitted to a parabola to compute bulk modulus B.

```
  MD++ eval
  MD++_PrintVar OMEGA "A^3" "%21.14e"
  MD++_PrintVar EPOT "eV" "%21.14e"
}

MD++ quit
```

Every MD++ variable that was bound to a string (by `bindvar` in md.cpp) can be fetched by `MD++_Get`. Some widely used variables are: number of atoms `NP`, simulation box volume `OMEGA`, potential energy `EPOT`, kinetic energy `KATOM`, instantaneous temperature `Tinst`, pressure `PRESSURE`, current iteration step `curstep`, total internal stress (Virial + kinetic term) `TSTRESS_xx`, `TSTRESS_yy`, `TSTRESS_zz`, `TSTRESS_xy`, `TSTRESS_xz`, `TSTRESS_yz`, *etc.*. The Tcl file above can be run with SW potential by typing

```
$ bin/sw_gpp scripts/Examples/Tcl/si-bulk.tcl
```

Results like the following will be printed on the screen.

```
OMEGA    =  1.03445152348275e+04 (in A^3)
EPOT     = -2.37028099055573e+03 (in eV)
```

The `for` loop creates 7 different configurations and asks MD++ to evaluate the potential energy. The data points, (`OMEGA`, `EPOT`), can be fitted to a parabola as shown in figure 1. The bulk modulus of $B = 108.1$ GPa can be obtained from the second derivative of the curve. MD++ command `saveH` saves the current box matrix H to storage, i.e. `H0 := H`. `restoreH` performs `H := H0`. `scaleH` multiplies the constant specified by `input` to every components of `H`, which is equivalent to give a uniform strain to the entire simulation box.

2.3 Writing data file

Instead of printing the results to the screen, it is usually more convenient to save the results in a data file in your favorite format. This can be easily done with the `open` and `puts` command in Tcl. The final part of the previous Tcl file can be modified to the following to save the results in to file `EvsVol.dat`.

```
MD++ saveH
set fileID [open "EvsVol.dat" w]  # open file to write
for {set x 997} {$x <= 1003} {incr x} {
  set y [expr $x/1000.0]
  MD++ restoreH input = \[ $y \] scaleH
  MD++ eval
  puts $fileID "[format %21.14e [MD++_Get OMEGA]]\t \
                [format %21.14e [MD++_Get EPOT]]"
}
close $fileID                     # close data file
```

Variable `$fileID` contains the file handler for the data file that was just opened for writing. When `$fileID` is given as the first argument for `puts`, the output is redirected to the file instead of to the screen. Don't forget to close the file by the `close` command before exiting the simulation. After running MD++ with this input file, you will obtain a `EvsVol.dat` file with the following content.

```
 1.01599792136425e+04     -2.37027775032590e+03
 1.01905815401367e+04     -2.37043479747376e+03
 1.02212452554299e+04     -2.37052875901589e+03
 1.02519704210336e+04     -2.37055999874412e+03
 1.02827570984600e+04     -2.37052887814698e+03
 1.03136053492207e+04     -2.37043575643357e+03
 1.03445152348275e+04     -2.37028099055573e+03
```

This can be easily loaded into your favorite software to be plotted, such as `Matlab` or `Octave`, or `Gnuplot`.

2.4 Command line arguments

With Tcl we can pass extra arguments from the command line when we run MD++. In this way, one input file can perform a set of different but related tasks, depending on the command line arguments. In the following example (`si-bulk3.tcl`), MD++ will decide whether or not to relax the atomic positions before printing out the potential energy, depending on the value of the first command line argument (following the input file name).

```
# -*-shell-script-*-
#*******************************************
# Definition of procedures
#*******************************************
```

```
proc initmd { } { MD++ {
setnolog
setoverwrite
dirname = runs/si-example
zipfiles = 1   # zip output files
NIC = 200 NNM = 200
#---------------------------------------------
# Create Perfect Lattice Configuration
#
element0 = Si
crystalstructure = diamond-cubic
latticeconst = 5.430949821 #(A) for Si
latticesize = [ 1 0 0 4
                0 1 0 4
                0 0 1 4 ]
} }

#---------------------------------------------
proc relax_fixbox { } { MD++ {
# Conjugate-Gradient relaxation
conj_ftol = 1e-6 conj_itmax = 1000 conj_fevalmax = 1000
conj_fixbox = 1  relax
} }

#*******************************************
# Main program starts here
#*******************************************
initmd
MD++ makecrystal
if {$argc > 0} {
  set do_relax [ lindex $argv 0 ]
  puts "do_relax = $do_relax"
}

MD++ saveH
set fileID [open "EvsVol.dat" w]
for {set x 997} {$x <= 1003} {incr x} {
  set y [expr $x/1000.0]
  MD++ restoreH input = \[ $y \] scaleH
  if { $do_relax == 1 } { relax_fixbox }
  MD++ eval
  puts $fileID "[format %21.14e [MD++_Get OMEGA]]\t \
                [format %21.14e [MD++_Get EPOT]]"
}
close $fileID
MD++ quit
```

If we run this script by typing

```
$ bin/sw_gpp scripts/Examples/Tcl/si-bulk3.tcl 1
```

then MD++ relaxed the atomic structure to an energy minimum before printing out the potential energy to the data file, because the first argument is 1. `$argv` is an array and `[ lindex $argv 0 ]` retrieves the first element of the array. On the other hand, if we run MD++ by

```
$ bin/sw_gpp scripts/Examples/Tcl/si-bulk3.tcl
```

then there is no relaxation. In this particular case, there is no difference in the data file created by these two runs. But if we compute elastic constants other than the bulk modulus (such as C_{11}, C_{12}, C_{44}), allowing the atoms to relax will generally give a different value for a compound crystal structure such as diamond cubic (with more than one basis atom). The results for relaxed configurations correspond to experimental measurements.

2.5 User defined Tcl subroutines for MD++

As mentioned earlier, Tcl allows us to define subroutines to describe commonly used operations and facilitate our use of MD++. For example, the following subroutines can help us print out MD++ matrices, arrays, and vector arrays conveniently within Tcl.

```
proc index3 { ID coord } {
    set ind.x 1; set ind.y 2; set ind.z 3
    expr { $ID * 3 + [set ind.$coord] - 1 }
}

proc MD++_GetVector { name ID coord } {
    MD++_Get $name [index3 $ID $coord]
}

proc MD++_PrintVar { name {unit \?} {fmt %11.4e} } {
    puts "$name\t= [format $fmt [MD++_Get  $name]] (in $unit)"
}

proc MD++_PrintMatrix { name {unit \?} {fmt %11.4e} } {
    puts "$name\t= [format $fmt [MD++_Get $name 0]]\
                   [format $fmt [MD++_Get $name 1]]\
                   [format $fmt [MD++_Get $name 2]]  (in $unit)"
    puts      "\t  [format $fmt [MD++_Get $name 3]]\
                   [format $fmt [MD++_Get $name 4]]\
                   [format $fmt [MD++_Get $name 5]]"
    puts      "\t  [format $fmt [MD++_Get $name 6]]\
                   [format $fmt [MD++_Get $name 7]]\
                   [format $fmt [MD++_Get $name 8]]"

proc MD++_PrintArray {name {unit \?} {i0 0} {i1 3} {fmt %11.4e}} {
    for {set ID $i0} {$ID < $i1} {incr ID 1} {
        puts "$name\($ID\)= [format $fmt [MD++_Get  $name $ID]]\
                            [expr {$ID==$i0?"(in $unit)":" "}]"
    }
}

proc MD++_PrintVectorArray { name {unit \}?} {i0 0} {i1 3}
                                                     {fmt %11.4e} } {
    for {set ID $i0} {$ID < $i1} {incr ID 1} {
        puts "$name\($ID\)=([format $fmt [
                                    MD++_GetVector $name $ID x]]\
                            [format $fmt [
                                    MD++_GetVector $name $ID y]]\
                            [format $fmt [
                                    MD++_GetVector $name $ID z]])\
                            [expr {$ID==$i0?"(in $unit)":" "}]"
  }
}
```

If these definitions are included at the beginning of your Tcl input file, then you can use the following commands.

```
MD++_PrintMatrix TSTRESS "eV/A^3"
MD++_PrintMatrix H       "A"
MD++_PrintVectorArray SR "no unit" 0 5
MD++_PrintArray EPOT_IND "eV"      0 5
```

They will lead to printouts that look like the following.

```
TSTRESS = -5.9944e-03  5.4507e-21 -1.0648e-20  (in eV/A^3)
           5.4507e-21 -5.9944e-03  5.4507e-21
          -1.0648e-20  5.4507e-21 -5.9944e-03
H       =  2.1789e+01  0.0000e+00  0.0000e+00  (in A)
           0.0000e+00  2.1789e+01  0.0000e+00
           0.0000e+00  0.0000e+00  2.1789e+01
SR(0)= (-5.0000e-01 -5.0000e-01 -5.0000e-01) (in no unit)
SR(1)= (-3.7500e-01 -3.7500e-01 -5.0000e-01)
SR(2)= (-5.0000e-01 -3.7500e-01 -3.7500e-01)
SR(3)= (-3.7500e-01 -5.0000e-01 -3.7500e-01)
SR(4)= (-4.3750e-01 -4.3750e-01 -4.3750e-01)
EPOT_IND(0)= -4.6295e+00 (in eV)
EPOT_IND(1)= -4.6295e+00
EPOT_IND(2)= -4.6295e+00
EPOT_IND(3)= -4.6295e+00
EPOT_IND(4)= -4.6295e+00
```

The command `MD++_PrintMatrix` prints a matrix, such as the total stress tensor (`TSTRESS`). The command `MD++_PrintVectorArray` prints a segment of an array of vectors. In the example we printed the scaled coordinates (`SR`) of atoms 0 to 4. The command `MD++_PrintArray` prints an array of scalar, such as the individual potential energy contribution of atoms 0 to 4. It is likely that more Tcl subroutines will be developed by MD++ users as the use of Tcl in MD++ continues.