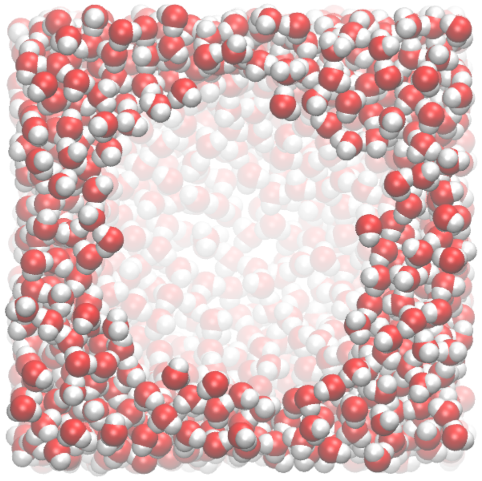
University of Illinois at Urbana-Champaign
Beckman Institute for Advanced Science and Technology
Theoretical and Computational Biophysics Group
Computational Biophysics Workshop

User-Defined Forces in NAMD



Alek Aksimentiev

David Wells

Greg Sigalov

November 2006

Contents

Introduction

This tutorial is designed to guide users of VMD and NAMD in the use of `tclForces` and `tclBC`. The tutorial assumes that you already have a working knowledge of VMD and NAMD, as well as the Tcl language. For the accompanying VMD and NAMD tutorials go to:
`http://www.ks.uiuc.edu/Training/Tutorials/`
This tutorial has been designed specifically for VMD 1.8.5, and should take about 5 hours to complete in its entirety.

The tutorial is subdivided into two separate units, one for `tclForces` and one for `tclBC`. These script-based facilities make possible virtually any perturbation of your system, and after completing this tutorial you will have the basic skills to use them.

Throughout the text, some material will be presented in separate "boxes". These boxes include complementary information to the tutorial and tips or technical details that can be further explored by the advanced user.

Tcl code is written in `this font`. When a line ends with a backslash (\) this means that there should be **no line break** . . . don't press enter, just keep typing!

If you have any questions or comments on this tutorial, please email the TCB Tutorial mailing list at tutorial-l@ks.uiuc.edu. The mailing list is archived at http://www.ks.uiuc.edu/Training/Tutorials/mailing_list/tutorial-l/.

Required programs

The following programs are required for this tutorial:

- **VMD:** Available at http://www.ks.uiuc.edu/Research/vmd/ (for all platforms)
- **NAMD:** Available at http://www.ks.uiuc.edu/Research/namd/ (for all platforms)

Getting Started

You can find the files for this tutorial in the `forces-tutorial-files` directory. Below you can see in Fig. 1 the files and directories of `forces-tutorial-files`.

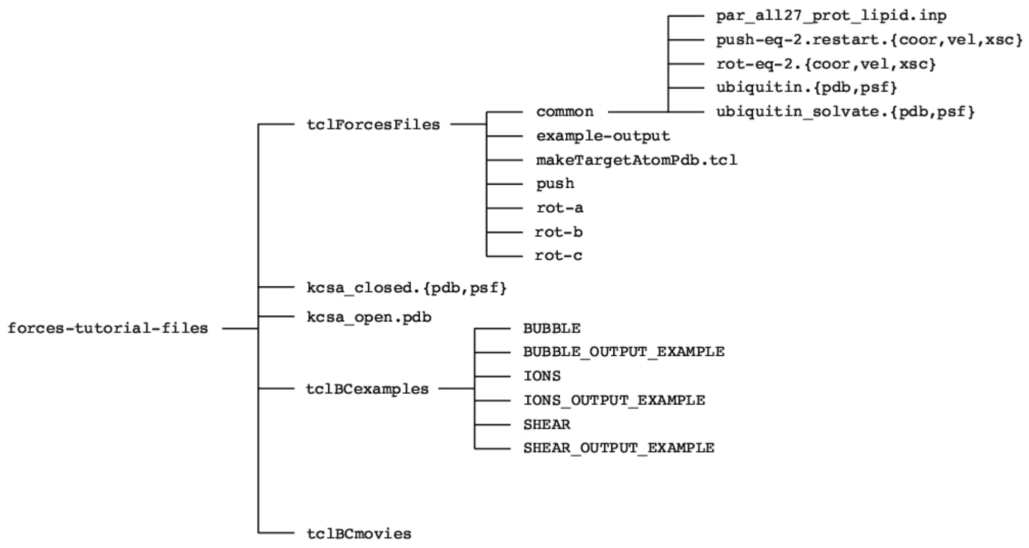


Figure 1: Directory structure of forces-tutorial-files

To start VMD type `vmd` in a Unix terminal window, double-click on the VMD application icon in the `Applications` folder in Mac OS X, or click on the Start → Programs → VMD menu item in Windows.

1 TclForces

In this unit, you will learn about NAMD's TclForces functionality. Although many user needs were anticipated in the development of NAMD, to anticipate them all specifically would be impossible. TclForces allows users to easily supplement the built-in constraint and force functionality with script-based forces.

1.1 Introduction

The basic idea of `tclForces` is very simple: we provide NAMD with a Tcl script which tells NAMD to apply certain forces to certain atoms. The only special requirement of this script is that it define a command named `calcforces`. This command is called by NAMD each timestep. Within the script, we can access atomic positions and masses, as well as parameters set in the NAMD configuration file. In this unit, we will see how to use `tclForces` through four examples of increasing complexity.

1.2 Example 1: Constant Forces

For our first exposure to `tclForces`, we will do one of the simplest thing one could imagine: apply a constant net force to a system.

1 First, let's look at the NAMD configuration file. Go to the directory `forces-tutorial-files/tclForcesFiles/push` and open the file `push.namd`. Everything is standard about it until the end, where we see some unfamiliar lines:

```
tclforces        on
set linaccel     "30 0 0"
tclforcesscript  push.tcl
```

The first line simply turns `tclForces` on. The second sets a parameter for the acceleration that we'll use in the script, as we'll soon see. The last line is the name of the script which actually does the work.

In-line scripts. It is also possible to include the contents of your `tclForces` script directly in your NAMD configuration file. Instead of providing the name of a script, we could instead write the script contents within curly braces:

`tclforcesscript` {
... *contents of script* ...
}

This is most useful for particularly short scripts.

2 Now let's look at the script. Open the file `push.tcl`, in the same directory.

3 First, we select the atoms we're interested in:

```
set numatoms 1231

set atoms {}
for { set i 1 } { $i <= $numatoms } { incr i } {
  lappend atoms $i
}

foreach atom $atoms {
  addatom $atom
}
```

The structure we're using is ubiquitin, the same protein as that used in the VMD and NAMD Tutorials. Here, we're using the protein alone, in vacuum, and so the structure contains just 1231 atoms. We first build a list of the atomic indices (NAMD atom indices, unlike VMD atom indices, start at 1.) For each atom, we then call the `addatom` command. This tells NAMD that we want access to this atom's coordinates. Forces may be applied to an atom without this call.

4 We next convert from the natural acceleration units Å·ps^{-2} to the NAMD units kcal/mol·Å·amu, and print some information:

```
set linaccel_namd [vecscale [expr 1.0/418.68] $linaccel]
print "Linear acceleration imparted:  ($linaccel) Ang*ps^-2"
```

Note that to print messages, we must use the `print` command, instead of the normal Tcl command `puts`.

NAMD Units. The basic NAMD units are kcal/mol for energy, Angstroms for length, atomic mass units (a.k.a. Daltons) for mass, and Bar for pressure. Other units are derived from these, e.g. the NAMD unit of force is 1 kcal/mol·Å ≈ 69.48 pN.

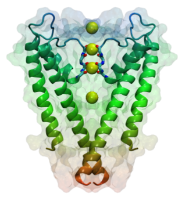
5 The only thing left is the `calcforces` command:

```
proc calcforces { } {
  global atoms numatoms linaccel_namd
  loadcoords coords
  loadmasses masses
  set comsum "0 0 0"
  set totalmass 0
  foreach atom $atoms {
    set force [vecscale $masses($atom) $linaccel_namd]
    addforce $atom $force
    set comsum [vecadd $comsum \
      [vecscale $masses($atom) $coords($atom)]]
    set totalmass [expr $totalmass + $masses($atom)]
  }
  print "Center of mass = [vecscale [expr 1.0/$totalmass] \
    $comsum]"
}
```

The first line of the command is very important. It basically imports the variables named, so that they can be accessed in the `calcforces` command.

Next we call the command `loadcoords`. This sets the variable `coords` to a Tcl array of coordinates of the atoms that have been added using `addatom`. In this case, of course, that is all the atoms of the system. Array elements in Tcl are accessed using parentheses. Similarly, `loadmasses` retrieves the atomic masses, putting them in an array named `masses`.

Then we loop through the atoms, applying the specified force to each using the `addforce` command. Along the way, we also calculate the center of mass of the protein. We see two more Tcl commands in actions here: `vecadd` and `vecscale`. These add two vectors and multiply a vector by a scalar, respectively. There is a third vector command, `vecsub`, which we will see a bit later.



Linear acceleration. Linear acceleration is governed by Newton's Second Law:

$$\mathbf{F} = m\mathbf{a}$$

Here, $\mathbf{F}$ is a force vector, m is the mass, and $\mathbf{a}$ is the acceleration vector. Thus, this equation says that to achieve a specified acceleration, we must apply a force equal to that acceleration times the mass, in the same direction we want to accelerate. In our case, we apply a force to each individual atom, scaled by its mass so that each experiences the same acceleration, and thus the protein moves together as a whole.

6 Because the system is so small, you can run it on your desktop or laptop without trouble, and see the protein move. Run the file `push.namd`. With a normal installation, you would do this with the command

```
namd2 push.namd > push.log
```

Coordinates. Coordinates in `tclForces` are not wrapped around the unit cell. If you need wrapped coordinates for whatever reason, you must wrap them yourself using the periodic cell dimensions and origin.

7 After this finishes, we want to see how the protein was affected. We will now use VMD's MultiPlot plugin to view the results. Open the file `plot.tcl` in your favorite text editor, and change the first line to be the location of the log file you just created, e.g.

```
set log push.log
```

if you named the log file `push.log`. To run the script, either type

```
vmd -e plot.tcl
```

from the command line, or open VMD and source the script file in the Tk Console:

```
source plot.tcl
```

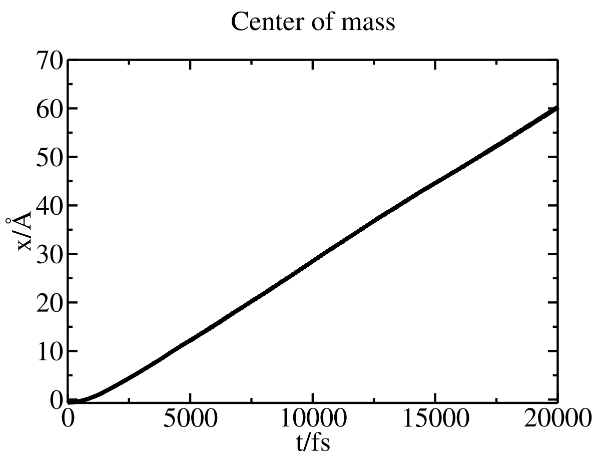


Figure 2: Center of mass position plotted versus time.

8 You should see something like Fig. 2. The beginning of the graph is quadratic, as we expect. However, the velocity saturates at some value because of the Langevin dynamics used in the simulation. Langevin dynamics involve a damping factor, which creates a terminal velocity.

Langevin dynamics. Langevin dynamics are commonly used in NAMD to simulate at constant temperature, in this case in a so-called NVT ensemble. The keywords beginning with `langevin` in the NAMD configuration file control its settings. Without Langevin dynamics activated, we would be simulating in the NVE ensemble, in which case energy rather than temperature is constant. NAMD also includes constant-pressure options for simulation in the NPT ensemble.

9 Now open the trajectory in VMD. First launch VMD, then open the Tk Console. Make sure you're in the `tclForcesFiles` directory, then type the command

```
mol load psf common/ubiquitin.psf dcd push/push.dcd
```

If you could not produce the trajectory, there is a sample one in the `example-output` directory. Notice that the Langevin dynamics prevent the protein from moving too quickly.

1.3 Example 2: Rotation

We will now add a layer of complexity: we will force atoms differently based on their coordinates. In this example, we will force atoms in such a way that, in addition to a linear force, the protein also rotates about an axis parallel to the z-axis and through the protein's center of mass. Note that this represents a perturbation which cannot be achieved using NAMD's built-in constant force and rotating constraint functionalities.

1 Change to the directory `forces-tutorial-files/tclForcesFiles/rot-a`.

2 Look at the NAMD configuration file. Open the file `rot-a.namd`. At the end, we see the following:

```
tclforces        on
set linaccel     "30 0 0"
set angaccel     1
tclforcesscript  rot-a.tcl
```

This is very similar to the previous example, but here we provide an additional scalar to describe the angular acceleration.

3 Now let's look at the script itself. Open the file `rot-a.tcl`. The atom selection part at the beginning is identical to the last case, so we will not repeat the discussion. The next part of the file is again similar, but also processes the angular acceleration, and sets a constant for π.

```
set linaccel_namd [vecscale [expr 1.0/418.68] $linaccel]
set angaccel_namd [expr double($angaccel)/418.68]
set PI 3.1415926535898
```

```
print "Linear acceleration imparted:  ($linaccel) Ang*ps^-2"
print "Angular acceleration imparted:  (0 0 $angaccel) Rad*ps^-2"
```

We convert from natural units to NAMD units, just like before. Then we print some informational text using the `print` command.

4 Now we'll look at the `calcforces` command. The first part is nearly the same as in the previous example. The differences are the change of one of the `global` variables, and the fact that we no longer apply forces in the first loop.

```
proc calcforces { } {
  global atoms numatoms linaccel_namd angaccel_namd
  global PI

  loadcoords coords
  loadmasses masses

  set comsum "0 0 0"
  set totalmass 0
  foreach atom $atoms {
    set comsum [vecadd $comsum [vecscale $masses($atom) \
      $coords($atom)]]
    set totalmass [expr $totalmass + $masses($atom)]
  }
  set com [vecscale [expr 1.0/$totalmass] $comsum]
  print "Center of mass = $com"
```

5 The last part of the script is where the real differences lie:

```
  foreach atom $atoms {
    set linforce [vecscale $masses($atom) $linaccel_namd]
    set r [vecsub $coords($atom) $com]
    set x [lindex $r 0]
    set y [lindex $r 1]
    set rho [expr sqrt(pow($x, 2) + pow($y, 2))]
    set phi [expr atan2($y, $x) + $PI/2]

    if { $atom == 1 } {
      print "atom $atom:  phi = $phi"
    }

    set angdir "[expr cos($phi)] [expr sin($phi)] 0.0"
    set angforce [vecscale [expr $masses($atom) * \
      $angaccel_namd * $rho] $angdir]
    set force [vecadd $linforce $angforce]
    addforce $atom $force
  }
}
```

The first section above calculates the distance from the rotation axis, `rho`, and the angle from the x-axis at which the force should be applied, `phi` (90° from the angle of the position vector, hence the addition of $\pi/2$ radians.) Fig. 3 shows how these variables are related.

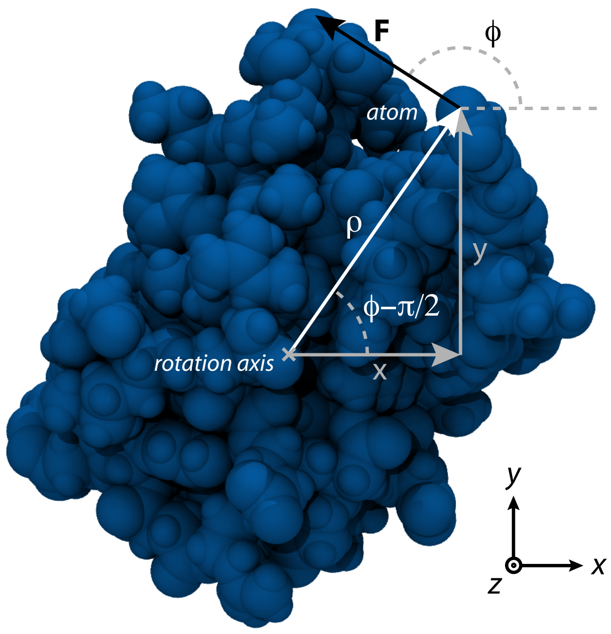
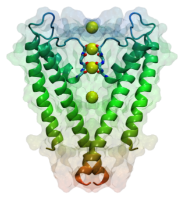


Figure 3: Relationships among variables in the script `rot-a.tcl`

After printing the force angle associated with one atom so we can monitor the rotation, we find a unit vector (i.e. a vector with length one) pointing in the `phi` direction. We then multiply this by the mass of the atom and its distance from our rotation axis, giving us the correct angular force for the angular acceleration requested in the NAMD configuration file. Finally, we add the linear and angular forces to get the total force.



Angular acceleration. Angular acceleration is governed by an equation analogous to Newton's Second Law:

$$\boldsymbol{\tau} = I\boldsymbol{\omega} \tag{1}$$

$\boldsymbol{\tau}$ is torque (analogous to force), $\boldsymbol{\omega}$ is the angular acceleration, and I is the moment of inertia (analogous to mass). Torque is given by

$$\boldsymbol{\tau} = \boldsymbol{\rho} \times \mathbf{F}$$

where $\mathbf{F}$ is the force and $\boldsymbol{\rho}$ is a position vector relative to the rotation axis (see Fig. 3). In our case, we apply the force perpendicular to the $\boldsymbol{\rho}$ vector, so the cross product simplifies and we get

$$\tau = \rho F \tag{2}$$

for the magnitude. For point particles, the moment of inertia is simply $I = m\rho^2$, where m is the particle's mass. Combining this with Eqns. 1 and 2 and solving for F, we get

$$F = m\omega\rho$$

for the force per particle to achieve a specified angular acceleration.

6 Now run the simulation:

```
namd2 rot-a.namd > rot-a.log
```

This should again take just a few minutes.

7 Plot the angle `phi` by using the script `plot.tcl` just as you did before (there is a different copy in the current directory.)

You should see something like Fig. 4. Plot the center of mass as well, by opening `plot.tcl` and changing the `var` variable to `"Center of mass"`. This can be accomplished by simply commenting the first `set var` line, and uncommenting the other:

```
#set var "atom 1:  phi"
set var "Center of mass"
```

8 We again see a roughly quadratic time dependence at the beginning of the trajectory give way to a linear one, again a consequence of the Langevin damping.

9 Open the trajectory in VMD. With VMD open, go to the main `tclForcesFiles` directory, then type the following in the Tk Console:

```
mol load psf common/ubiquitin.psf dcd rot-a/rot-a.dcd
```

Challenge: Find the dependence of the terminal linear and angular velocities on the `langevinDamping` parameter of the simulation, set in the configuration file, as well as the dependence on the applied force.

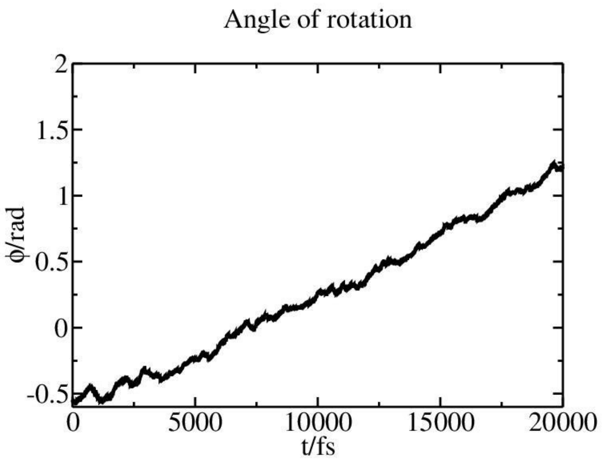


Figure 4: Rotation angle plotted versus time.

1.4 Example 3: Forcing a Subset of Atoms

This example deals with the important ability to force a subset of your atoms. In practice, you almost never want to apply force to all atoms in the system. In this section, we will apply forces only to the backbone atoms of the protein. We accomplish this by making a special “target atom” PDB file, using the beta column to mark the atoms we want to apply force to. In addition, we will use the occupancy column to tell our script the atomic masses, in order to demonstrate how to provide the script with additional information.

1 Change to the directory `forces-tutorial-files/tclForcesFiles/rot-b`.

2 As usual, we will first examine the NAMD configuration file. Open the file `rot-b-short.namd`. First, note that we are now using a different system, one in which the ubiquitin has been solvated. This is a more natural environment for the protein, and we can use it now that we are not applying forces to all atoms.

Look at the end of the file, where we set up `tclForces`. We see a new line:

```
set targetAtomPdb  ../common/ubiquitin_solvate_backbone.pdb
```

With this line, we specify the aforementioned target atom PDB file. Thus, our first task is to produce that file. We will use a VMD script to accomplish this.

3 Open the file `../makeTargetAtomPdb.tcl`. The first section sets a few variables. First, the relevant file names:

```
set pdb common/ubiquitin_solvate.pdb
set psf common/ubiquitin_solvate.psf
set targetPdb common/ubiquitin_solvate_backbone.pdb
```

Next, we set the selection text we will use to select our atoms, and the beta value that these atoms will receive:

```
set selection "protein and backbone"
set targetMark "1.00"
```

4 Next, we load the structure into VMD, and set both the beta and occupancy columns to zero:

```
mol load psf $psf pdb $pdb
set all [atomselect top all]
$all set beta 0
$all set occupancy 0
```

5 Now we make a selection of the target atoms, set their beta and occupancy columns to the appropriate values, and write the output PDB. Note the ease with which the list of masses are used to set the occupancy:

```
set target [atomselect top $selection]
set masses [$target get mass]
$target set beta $targetMark
$target set occupancy $masses

$all writepdb $targetPdb
exit
```

6 Now run the script:

```
vmd -dispdev text -e ../makeTargetAtomPdb.tcl
```

(Note: For Windows users, open the VMD GUI and source the script file in the Tk Console.)

If you were unable to run the script, copy the file `example-output/ubiquitin_solvate_backbone.pdb` into the `common` directory.

7 Next, take a look at the `tclForces` script. Open the file `rot-b.tcl`. Right from the beginning, we notice many differences. It first sets a `targetMark` variable so that the script can recognize the marks we've set in the target PDB:

```
set targetMark  "1.00"
```

8 Now we have to process the PDB. There are no built-in routines for accomplishing this, so the following code is quite low-level, but it is conceptually simple. The columns of a PDB file have a fixed character width, so we must simply read each line, and break it into fixed-sized pieces:

```
set targets {}
set masses {}
set inStream [open $targetAtomPdb r]
foreach line [split [read $inStream] \n] {
  set string1 [string range $line 0 3]
  set string2 [string range $line 6 10]
  set string3 [string range $line 17 20]
  set string4 [string range $line 12 15]
  set string5 [string range $line 46 53]
  set string6 [string range $line 72 75]
  set string673 [string range $line 73 75]
  set string7 [string range $line 22 25]
  set string8 [string range $line 62 65]
  set string9 [string range $line 55 60]
```

9 These pieces are then parsed according to the PDB format. We first make sure that this line corresponds to an atom record and not a comment line or some other entry. Then, if the atom has a beta value matching the target value, we form a triple consisting of the atom's `segname`, `resid`, and name. This is necessary for finding the index of the atom:

```
  if { ([string equal $string1 {ATOM}] || \
       [string equal $string1 {HETA}] ) &&
       [string equal $targetMark $string8] } {
    lappend targets "[string trim $string6] \
              [string trim $string7] [string trim $string4]"
    lappend masses "[string trim $string9]"
  }
}
close $inStream
```

10 The next step is to use the atom triples to find its index, and form a list of these, using the `atomindex` command. `addatom` is then called for each of these atoms:

```
set atoms {}
foreach target $targets {
  foreach {segname resid atom} $target { break }
  set atomindex [atomid $segname $resid $atom]
  lappend atoms $atomindex
  addatom $atomindex
}
```

Notice the third line. This is a common trick in Tcl for succinctly extracting the elements of a list into individual scalar variables.

11 Next we find the number of target atoms. We also set a boolean flag, `applyforces`, telling the rest of the script whether any atoms have been selected to force, allowing us to skip the calculation code if it's unnecessary:

```
set numatoms [llength $atoms]
if { $numatoms > 0 } {
  set applyforce 1
} else {
  print "WARNING: no target atoms have been detected"
  set applyforce 0
}

set linaccel_namd [vecscale [expr 1.0/418.68] $linaccel]
set angaccel_namd [expr double($angaccel)/418.68]
set PI 3.1415926535898
print "Linear acceleration imparted:  ($linaccel) Ang*ps^-2"
print "Angular acceleration imparted:  (0 0 $angaccel) Rad*ps^-2"
```

12 Now we get to the main part of the script, the `calcforces` definition. It is essentially identical to the last example, except for the test of `applyforce`:

```
proc calcforces { } {
  global atoms numatoms masses linaccel_namd angaccel_namd
  global applyforce
  global PI

  if { $applyforce } {
    loadcoords coords
    set comsum "0 0 0"
    set totalmass 0
    foreach atom $atoms mass $masses {
      set comsum [vecadd $comsum [vecscale $mass $coords($atom)]]
      set totalmass [expr $totalmass + $mass]
    }
    set com [vecscale [expr 1.0/$totalmass] $coordsum]
    print "Center = $com"

    foreach atom $atoms mass $masses {
      set linforce [vecscale $mass $linaccel_namd]
      set r [vecsub $coords($atom) $com]
      set x [lindex $r 0]
      set y [lindex $r 1]
      set rho [expr sqrt(pow($x, 2) + pow($y, 2))]
      set phi [expr atan2($y, $x) + $PI/2]

      if { $atom == 1 } {
        print "atom $atom:  phi = $phi"
      }
```

```
        set angdir "[expr cos($phi)] [expr sin($phi)] 0.0"
        set angforce [vecscale [expr $angaccel_namd * $rho * \
          $mass] $angdir]
        set force [vecadd $linforce $angforce]
        addforce $atom $force
      }
    }
  }
```

13 Now run the simulation, just as you have in previous examples:

```
namd2 rot-b-short.namd > rot-b-short.log
```

If you have access to a cluster, run it on the cluster—refer to local instructions for how this is done. This simulation will run for just 100 steps, so that the benchmark times are printed. In the next section, we will see how to improve this performance.

If you are interested, examine the files `rot-b-long.log` and `rot-b-long.dcd` in the `example-output` directory, which were produced with a slightly longer simulation. When loading the trajectory in VMD, use the PSF `common/ubiquitin_solvate.psf`

1.5 Example 4: Improving Efficiency

In most cases, it's unnecessary to recalculate the force values applied at each time step. Depending on the situation, one may be able to recalculate every 1000 steps, saving a lot of computational effort and therefore increasing the speed of the simulation. In this section, we will see how to do that.

1 Change to the directory `forces-tutorial-files/tclForcesFiles/rot-c`.

2 Open the NAMD configuration file, `rot-c-short.namd`. The new line this time is

```
set forcesRecalcFreq  10
```

As its name implies, this parameter sets how often our forces will be recalculated.

3 Now open the script `rot-c.tcl`. The first differences start at line 58:

```
set forces {}
foreach index $atoms {
  lappend forces "0.0 0.0 0.0"
}

set forcecount $forcesRecalcFreq
set printcount 0
```

We first set up a `forces` list. This is the variable which holds the values of the forces we will apply to the atoms. We also set up some counter variables: `forcecount` and `printcount` are both incremented each timestep. When `forcecount` equals `forcesRecalcFreq`, the forces are recalculated, and `forcecount` is reset to zero; otherwise, the forces are not recalculated.

4 Now we begin the `calcforces` command. First, as usual, we declare our global variables. We then apply the forces that are saved in the `forces` list.

```
proc calcforces { } {
  global atoms numatoms forcemult masses avgmass forces
  global applyforce forcesRecalcFreq
  global forcecount printcount
  global PI

  if { $applyforce } {
    foreach atom $atoms force $forces {
      addforce $atom $force
    }
```

5 Next, we test whether the forces will be recalculated *next* timestep, and if so, tell NAMD that we will want atomic coordinates for the target atoms by calling `addatom`:

```
    if { $forcecount == [expr $forcesRecalcFreq - 1] } {
      print "Adding atoms prior to reconfiguring forces at \
        $printcount"
      foreach atom $atoms {
        addatom $atom
      }
    }
```

As we will see shortly, we call `clearconfig` after the forces are recalculated. This call erases all `addatom` records; without it, the coordinates of the atoms added will be available *every timestep*, independently of whether we call `loadcoords`, and therefore much of the potential speed gain will be lost. However, this means that `addatom` must be called again each time we want to recalculate forces, and because of technical details it has to be done at least one step before the coordinates will be used.

6 Next we have the code that recalculates the force. As alluded to above, the recalculation happens when `forcecount` equals `forcesRecalcFreq`. The rest of the force calculation code is identical to the code we had in the last example, except that instead of calling `addforce` with each force vector, we now put the vector in the `forces` list variable:

```
    if { $forcecount == $forcesRecalcFreq } {
      print "Recalculating forces at $printcount"
```

```
        loadcoords coords
        set comsum "0 0 0"
        set totalmass 0
        foreach atom $atoms mass $masses {
          set comsum [vecadd $comsum [vecscale $mass $coords($atom)]]
          set totalmass [expr $totalmass + $mass]
        }
        set com [vecscale [expr 1.0/$totalmass] $comsum]
        print "Center of mass = $com"

        set forces {}
        foreach atom $atoms mass $masses {
          set linforce [vecscale $mass $linaccel_namd]
          set r [vecsub $coords($atom) $com]
          set x [lindex $r 0]
          set y [lindex $r 1]
          set rho [expr sqrt(pow($x, 2) + pow($y, 2))]
          set phi [expr atan2($y, $x) + $PI/2]

          if { $atom == 1 } {
            print "atom $atom:  phi = $phi"
          }

          set angdir "[expr cos($phi)] [expr sin($phi)] 0.0"
          set angforce [vecscale [expr $angaccel_namd * $rho * \
            $mass] $angdir]
          set force [vecadd $linforce $angforce]
          lappend forces $force
        }
```

7 Finally, we print some information, call `clearconfig`, and reset `forcecount`.

```
        print "Step ${printcount}:  Recalculated \
          [llength $forces] forces"
        set forcecount 0
        clearconfig
      }
      incr forcecount
    }
    incr printcount
    return
  }
```

8 Now run the simulation:

```
namd2 rot-c-short.namd > rot-c-short.log
```

Again, this is a short simulation, running 100 steps, just long enough for benchmark timing information to be printed. Compare the speed of this simulation to the last. There should be a significant difference—comparing the example files `rot-b-short.log` and `rot-c-short.log`, which were run on just a single CPU, there is nearly a 15% speed increase. As the size of the system and the number of CPUs increase, applying `tclForces` efficiently becomes even more important: comparing `rot-c-long.log` with `rot-b-long.log`, both run on 20 processors, `rot-c` is almost 60% faster!

Once again, there are sample output files, `rot-c-long.dcd` and `rot-c-long.log` in the `example-output` directory, and should again be loaded together with the PSF file `common/ubiquitin_solvate.psf`.

Challenge: Take the provided file `kcsa_closed.pdb`, and use the partial structure `kcsa_open.pdb` as a target to open the channel using `tclForces`.

You now know the basics of `tclForces`. In the next section, you will learn about a similar feature of NAMD called `tclBC`, which is suitable for different circumstances.

2 TclBC

In a NAMD configuration file, one sets up the simulation parameters, such as the temperature, pressure, or a uniform electric field, for the molecular system as a whole. In many cases, though, you will need to impose boundary conditions, apply a non-uniform electric field, or selectively apply forces to a group of atoms of the given type or located in a particular area. For this purpose, you can use a `tclBC`(which stands for Tcl Boundary Conditions) script. A `tclBC`script is written in the Tcl programming language (see http://tmml.sourceforge.net/doc/tcl/), a rudimentary knowledges of which would be enough for most tasks. The script will be interpreted line-by-line, therefore slowing down the simulation, sometimes considerably. For this reason you will want to keep your scripts simple and efficient.

`tclBC` **versus** `tclForces`. `tclBC` and `tclForces` are both scripting interfaces that allow to get information about the atoms' positions and apply forces to selected atoms during a NAMD simulation. In both scripts, the force can be calculated individually for each atom, depending on its unique ID number and its current position. The main difference between `tclBC` and `tclForces` is that an independent instance of `tclBC` is running on each processor, and only atoms forming the patch treated by that processor are visible to the given instance of `tclBC`. This feature makes `tclBC` more efficient than `tclForces`, which is run on just one CPU, but also limits its capabilities. If you need to apply forces to each atom irrespective of the position of other atoms, use `tclBC`. On the other hand, if you have to consider mutual positions of two or more atoms or gather information about the whole system, you will need `tclForces`.

The way the coordinates of atoms are wrapped around periodic boundaries in a `tclBC` script is controlled using the command `wrapmode`, which that may have one of the following arguments:

- `patch` is the default mode: the atom's coordinates are considered as the position in NAMD's internal patch data structure. This is in general different from an atom's coordinates relative to the global origin, so `patch` mode should not be used unless you know what you are doing.
- In mode `input`, the atom's coordinates correspond to its position in the input files of the simulation.
- In mode `cell`, the atom's equivalent position in the unit cell centered on the `cellOrigin` is used.
- In mode `nearest`, the atom's equivalent position nearest to the `cellOrigin` is used.

For most purposes, mode `cell` is a good choice.

2.1 Example 1: Making a Bubble

Imagine that you have a water box, and you want to create a spherical bubble of vacuum. You can do that by applying forces to every atom found inside this sphere to push it out. To avoid instability, you would want to start with a small bubble and then increase it until the desired size is reached. The rate of increasing the size of the bubble is up to you. If this rate is too large, some atoms will be pushed too hard and therefore move too fast, causing the simulation to crush. Fast-moving atoms may also break the structure of a double-stranded DNA or another molecular complex held together by relatively weak non-bonded interactions (van der Waals and electrostatic forces). Even if the simulation remains stable, you should watch the changes in temperature and pressure to make sure that the energy influx, which is equal to the work done by the force you are applying, has enough time to dissipate.

Choosing the loading rate. Generally you would want to keep the perturbation of the system as low as possible by imposing any external forces as slowly as you can afford given your available computational resources. It is a good idea, though, to run a trial simulation at a fast pace to see what is happening to the system, and thereby get an idea of how fast you can move things without losing the stability of the simulation or increasing the temperature or pressure beyond reason.

1 Open the file `tclBCexamples/BUBBLE/eq04.bubble.namd` in a text editor. Scroll down to the end of the file to the line `tclBC on`.

You should find the following code:

```
tclBC on
tclBCScript {
  set bubbleCenter    "0.0  0.0  0.0"
  set tclBCScript     < your working directory >/bubble.tcl
  source $tclBCScript
}
tclBCArgs {0. 15. 0.01 5.}
```

Since a `tclBC` script is called from NAMD, it is referenced NAMD configuration file, which is also a good place to set the script parameters. The NAMD command `tclBC on` turns the TclBC interface on. In our example, `tclBCScript { ... }` contains the initialization of a key variable and a reference to the file that contains the script itself: `source $tclBCScript`. If the script is short, it can be placed entirely within the body of the command `tclBCScript` in the NAMD configuration file. Finally, the command `tclBCArgs` is used to pass a list of variables to the main TclBC procedure `calcforces` (we will talk about it very soon). In this case, the four arguments found in curly brackets have the meaning: "make a bubble starting from radius 0 and increase it to 15 at a rate of 0.01 per simulation step, by applying forces (proportional to) 5."

2 Look through the script `tclBCexamples/BUBBLE/bubble.tcl`.

`bubble.tcl` contains the following code:

```
# Two first agruments of calcforces are automatically forwarded
# to it by NAMD. The other 4 arguments match the list of 4 values
# from command tclBCArgs.

proc calcforces {step unique Rstart Rtarget Rrate K} {

  global bubbleCenter ;# defined in tclBCScript{ ... }

  # increase R, starting from $Rstart, by $Rrate at each step,
  # until it reaches $Rtarget; then keep it constant

  set R [expr $Rstart + $Rrate * $step]
  if { $R > $Rtarget } { set R $Rtarget }

  # let only the main processor print the output

  if { $unique } {
    print "step $step, bubble radius = $R"
  }

  # get the components of the bubble center vector

  foreach { x0 y0 z0 } $bubbleCenter { break }

  # pick atoms of the given patch one by one

  while {[nextatom]} {

    set rvec [getcoord] ;# get the atom's coordinates

    # get the components of the vector
    foreach { x y z } $rvec { break }

    # find the distance between the atom and the bubble center
    # (long lines can be broken by a backslash and continued
    # on the next line)

    set rho [expr sqrt(($x-$x0)*($x-$x0) + ($y-$y0)*($y-$y0) + \
      ($z-$z0)*($z-$z0))]

    # if the atom is inside the sphere, push it away radially
    if { $rho < $R } {
      set forceX [expr $K * ($x-$x0) / $rho]
      set forceY [expr $K * ($y-$y0) / $rho]
      set forceZ [expr $K * ($z-$z0) / $rho]
      addforce "$forceX $forceY $forceZ"
    }
  }
}
```

The structure of a TclBC script. Just as in the case of `tclForces`, every `tclBC` script must define a `calcforces` command. Unlike in the case of `tclForces`, however, `calcforces` now takes two or more arguments. The first two arguments are forwarded to it automatically: `step` is simply the simulation step, and `unique` is nonzero for only one instance of `calcforces` corresponding to the atom patch 0. The `tclBCArgs` line in the NAMD configuration file specifies the rest.

In the code `bubble.tcl`, we use the `unique` flag to print something from within `calcforces` once at each step. Without `if { $unique }` clause, the print command would be executed by every processor.

The list of variables `Rstart Rtarget Rrate K` forwarded to `calcforces` matches the list of values `tclBCArgs {0. 10. 0.01 5.}` of the NAMD configuration file. You can use more or fewer variables, or none at all, as long as the lengths of the two lists are the same. The automatic variable `step` and `unique` don't count, so `calcforces` will always have two more arguments than to `tclBCArgs`.

The key command in `calcforces` is `nextatom`, which tells NAMD to select the next atom of the given patch. `nextatom` returns a value which is nonzero if the next atom has been successfully selected, or zero if no more atoms are left to process. Thus, `while {[nextatom]} { ... }` is a loop that will be repeated until all atoms of the current patch are processed.

In principle, you never know which exactly atom will be selected next. However, once an atom is selected by `nextatom`, a number of commands can be used to get its properties:

```
set atomid [getid]     ;# 1-based atom ID number
set coords [getcoord]  ;# 3-component coordinate vector (A)
set mass   [getmass]   ;# mass (atomic units)
set charge [getcharge] ;# charge (atomic units)
```

Using these parameters, you can decide what to do to that atom.

In the script `bubble.tcl`, we find the coordinates of the atom using the command `foreach { x y z } $rvec { break }`; this is a standard trick to get the components of a vector or a list in Tcl. Using the same method, we find the coordinates of the bubble center. Then we check whether the atom is inside the bubble of radius `$R` and center `$bubbleCenter`. If so (`$rho < $R`), we calculate the components of the force vector such that it is directed radially from the bubble's center. In this simple example, the absolute value of the force is always `$K` whenever the force is applied. Finally, `addforce` tells NAMD to actually push the given atom by adding the force vector "`$forceX $forceY $forceZ`" to the force that would act on that atom otherwise.

3 Change (`cd`) to the directory `tclBCexamples/BUBBLE`.

4 Run a NAMD simulation by typing

```
namd2 eq04.bubble.namd > eq04.bubble.log
```

(for standard installations of NAMD.)

5 Open file `eq04.bubble.log`, and find a line that reports the speed of computation, e.g. `TIMING: 3000 CPU: 1340.82, 0.4363/step`. The speed of computation is given in seconds per step, one step being 1 fs in this case. Write these numbers down, as you will need to compare them to the speed of an optimized script.

6 If you are unable to run the simulation, use the sample output file `../BUBBLE_OUTPUT_EXAMPLE/eq04.bubble.log`.

7 Open the simulation trajectory in VMD. Use the sample trajectory file `../BUBBLE_OUTPUT_EXAMPLE/eq04.bubble.dcd` if necessary.

8 In main VMD window, click Graphics, then Representations. In Graphical Representations window, type `x>0` in Selected Atoms text box, press Enter. Below this box, click Trajectory, select Update Selections Every Frame. Rotate the image if necessary using mouse.

9 In the main VMD window, click Display, then check Depth Cueing box. Click Display again, click Display Settings. Set Cue Mode to Linear. Set Cue Start 2.00, Cue End 3.00, vary them to see the effect of this parameters on the system's view.

10 Run the trajectory animation, and watch the formation of the bubble.

Sample snapshots of the simulation trajectory are shown in Fig. 5. The directory `tclBCmovies` contains MPEG movies `bubble*.mpg` that show the bubble formation trajectory animation at 0.1 ps per frame.

2.2 Optimizing a TclBC script

The above script `bubble.tcl` is far from being perfect. The same job can be done much more efficiently by decreasing the number of atoms processed at each step, as well as the number of arithmetic operations. Note that Tcl arithmetic is much slower than hard-coded `C++` arithmetic of the NAMD. A more efficient script is given in file `BUBBLE/bubbleFast.tcl`.

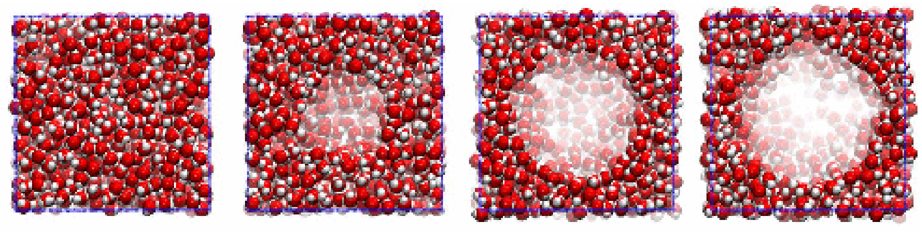


Figure 5: Snapshots of the bubble formation. The initial simulation cell boundaries are shown by a line.

Dropping atoms. The command `dropatom` tells the processor to no longer select the given atom by the command `nextatom` in a TclBC script. This increases the speed of computation because fewer atoms are considered. The command `cleardrops` cancels all the `dropatom` commands called earlier on the given processor. Note that a dropped atom can drift to another processor where it has not been dropped. Since the list of atoms in a patch changes with time, it is often a good idea to call `cleardrops` occasionally and then drop unnecessary atoms again.

1 Take a look at the script `BUBBLE/bubbleFast.tcl`, also presented below:

```
wrapmode cell

# this line is moved above calcforces to avoid doing this
# transformation more than once

foreach { x0 y0 z0 } $bubbleCenter { break }

set TOL 3. ;# distance tolerance parameter

proc calcforces {step unique Rstart Rtarget Rrate K} {

  global x0 y0 z0 TOL

  set R [expr {$Rstart + $Rrate * $step}]
  if { $R > $Rtarget } { set R $Rtarget }

  if { $unique } {
    print "step $step, bubble radius = $R"
  }

  # Atoms found at a distance larger than $TOL from the surface
  # of the bubble, or $RTOL from the center of the bubble, will
  # be ignored (dropped) for the rest of a 100-step cycle.

  set RTOL [expr {$R + $TOL}]

  # restore the complete list of atoms to consider

  if { $step % 100 == 0 } { cleardrops }

  while {[nextatom]} {

    set rvec [getcoord]
    foreach { x y z } $rvec { break }
    set rho [expr {sqrt(($x-$x0)*($x-$x0) + ($y-$y0)*($y-$y0) + \
      ($z-$z0)*($z-$z0))}]

    # Atoms at distances 0 to $R from the bubble center are pushed,
    # atoms father than $RTOL are dropped. Atoms between $R to $RTOL,
    # that is with a layer of thickness $TOL, are neither pushed nor
    # dropped at this step, so that they will be considered again at
    # the next step(s). They may come closer to the bubble and then
    # they will have to be pushed.

    if { $rho < $R } {

      set forceX [expr {$K * ($x-$x0) / $rho}]
      set forceY [expr {$K * ($y-$y0) / $rho}]
      set forceZ [expr {$K * ($z-$z0) / $rho}]
      addforce "$forceX $forceY $forceZ"

    } elseif { $rho > $RTOL } {

      dropatom ;# no longer consider this atom until "cleardrop"

    }
  }
}
```

Since most water molecules are far from the bubble, we don't need to check their coordinates every step. It's enough to do that once in a while (every 100 steps in this example). At all other times, only a small number of atoms that are still inside the bubble or within distance `$TOL` from it, and only these are selected by `nextatom` each step.

2 Open configuration file `eq04.bubbleFast.namd`, and make sure that it is calling the `tclBC` script `bubbleFast.tcl`. Run a NAMD simulation by typing `namd2 eq04.bubbleFast.namd > eq04.bubbleFast.log`.

3 If you are unable to run the simulation, use the sample log file `../BUBBLE_OUTPUT_EXAMPLE/eq04.bubbleFast.log`.

4 Look through file `eq04.bubbleFast.log`, find the word `TIMING`, and compare the speed to that found when using the script `bubble.tcl`. In our samples, the speeds obtained on a single CPU are 0.4363 s/step (8.0 ps/hr) and 0.2084 s/step (18.8 ps/hr) for the original and optimized script, respectively. Your numbers may differ.

5 Run a NAMD simulation using file `eq04.bubbleFast.namd` on a cluster. Make sure to direct the output to a new file. Repeat the above steps and find the computation speed. In our example (`../BUBBLE_OUTPUT_EXAMPLE/eq04.bubbleFast.4CPU.log`), the speed on 4 CPUs increased to 0.0619 s/step (40.3 ps/hr).

Scaling of the speed of computation. The NAMD performance on a multi-processor cluster or supercomputer is roughly proportional to the number of processors, when the number of CPUs is up to ≈ 1 CPU per 1000 atoms. Since a `tclBC` script distributes its computations over all CPUs used, the performance of NAMD with `tclBC` also scales well, though some loss in speed is inevitable. Optimization of `tclBC` scripts, in particular by the use of the `dropatom` command, is essential to keeping up high simulation performance.

2.3 Example 2: Concentrating the ions in a solution

A more complex initialization of the TclBC script is necessary when it is supposed to process a selected group of atoms. Imagine that, for whatever reason, you want to gather all K^+ and Cl^- ions within a sphere with the same center and radius as in the above example. For this purpose, we will push all ions toward the center of the sphere. The force applied will be constant for all ions beyond the sphere, and proportional to the distance between the ion and the sphere center for atoms inside the sphere.

1 Change to the directory `tclBCexamples/IONS`.

2 Open file `eq04.concentrateIons.namd`, scroll to the end, and look at the `tclBC` block.

As before, the `tclBC` script is set up in the NAMD configuration file:

```
tclBC on
tclBCScript {
  set sphereCenter    "0.0  0.0  0.0"
  set sphereRadius    10.0
  set maxForce         5.0
  set pdbSource       waterbox40-0.2M.pdb
  set tclBCScript     concentrateIons.tcl
  source $tclBCScript
}
tclBCArgs { }
```

In this case, we don't explicitly pass any extra arguments to `calcforces` (the variables `step` and `unique` will still be passed implicitly). Instead, we set up global variables `sphereCenter`, `sphereRadius`, `maxForce`, and `pdbSource`. They will be made visible in `calcforces` using command `global`.

To identify the ions, we will read the system's PDB file, locate the ions' entries, and make a list of their IDs.

3 Open the file `waterbox40-0.2M.pdb` in a text editor and find the lines containing text `CLA` (for chloride) or `POT` (for potassium).

The ion entries in the PDB file for a large system may look like

```
ATOM  *****  CLA CLA   170     -36.100 -48.734   7.045  1.00  0.00      ION CL
...
ATOM  *****  POT POT     1      -0.574 -49.643  34.378  1.00  0.00      POT  K
```

In the above example, atom IDs for the ions are above 99999, and therefore not shown in the PDB file by a decimal. Thus, to find out the atom IDs of the ions, we will need to read the PDB file line by line while counting the atom entries. An entry for each atom must begin with four letters `ATOM` or `HETA`. In the PDB file above, each ion is identified by the residue name `CLA` or `POT`, which occupies positions 17-19 of the line. Note that a 4-letter residue name would occupy positions 17-20 of the line. To make the script more general, we will read symbols 17-20 and then trim them (remove spaces).

4 Open file `concentrateIons.tcl`, and look through it.

The file `concentrateIons.tcl` begins with the following code:

```
set ionList {} ;# a list of atom IDs of the ions
set atomID 1   ;# in NAMD, atom IDs are 1-based numbers

set inStream [open $pdbSource r] ;# opening the PDB file

# reading the PDB file line after line:
foreach line [split [read $inStream] \n] {

  # symbols 0-3 should be ATOM or HETA
  set string1 [string range $line 0 3]

  # symbols 17-20 contain the residue name
  set string2 [string range $line 17 20]
  set string2 [string trim $string2] ;# trimming the residue name

  if { [string equal $string1 {ATOM}] || \
   [string equal $string1 {HETA}] } {

    # so it's a valid atom entry; let's see if this is an ion

    if { [string equal $string2 {CLA}] || \
     [string equal $string2 {POT}] } {

      # yes it's an ion; append its index to the list
      lappend ionList $atomID
    }

    # increase the atomID counter by 1 (default increment)
    incr atomID
  }
}
close $inStream
```

This part of the code, preceding `proc calcforces`, will be executed once by each processor. Identical instances of the list `ionList` of atom IDs for the ions will also be kept in memory at each processor.

Now we are ready to write the main part of the code:

```
wrapmode cell

# in this case the command tclBCArgs {} located in the NAMD config
# file doesn't pass any arguments to calcforces, therefore only
# step and unique will be listed below:

proc calcforces {step unique} {

  # list all global variables set outside this procedure
  # (except those we won't need, like pdbSource)

  global sphereCenter sphereRadius maxForce ionList

  # find the components of the sphere's center

  foreach { x0 y0 z0 } $sphereCenter { break }

  while {[nextatom]} {

    set atomid [getid] ;# get the ID of the current atom
    # check if this ID is listed in ionList; if it’s found,
    # lsearch will return the position of the search pattern
    # in the list (a number >= 0), otherwise -1 is returned

    if { [lsearch $ionList $atomid] >= 0 } {

      set rvec [getcoord] ;# get the ion’s coordinates
      foreach { x y z } $rvec { break }

      # find the distance between the ion and the sphere’s center

      set rho [expr sqrt(($x-$x0)*($x-$x0) + ($y-$y0)*($y-$y0) + \
        ($z-$z0)*($z-$z0))]

      # Apply same force $maxForce to each ion if it’s outside the
      # sphere. The components of the force vector are chosen so
      # that the vector is directed toward the sphere’s center,
      # and the vector’s norm is $maxForce.

      if { $rho > $sphereRadius } {

        set forceX [expr -$maxForce * ($x-$x0) / $rho]
        set forceY [expr -$maxForce * ($y-$y0) / $rho]
        set forceZ [expr -$maxForce * ($z-$z0) / $rho]

      } else {

      # If the ion in already inside the sphere, scale the force
      # by a factor of $rho/$sphereRadius, so that the force
      # decreases from $maxForce to 0 as the ion approaches the
      # sphere’s center:

        set forceX [expr -$maxForce * ($x-$x0) / $sphereRadius]
        set forceY [expr -$maxForce * ($y-$y0) / $sphereRadius]
        set forceZ [expr -$maxForce * ($z-$z0) / $sphereRadius]

      }

      # Finally, applying the force calculated above
      # to the current atom

      addforce "$forceX $forceY $forceZ"
    }
  }
}
```

5 Run a NAMD simulation with file `eq04.concentrateIons.namd`.

6 If you are unable to do that, find the example output files `eq04.concentrateIons.log` and `eq04.concentrateIons.dcd` in the directory `../IONS_OUTPUT_EXAMPLE`.

7 Open the log file `eq04.concentrateIons.log`, and find the speed of computation marked by keyword `TIMING`. In our example, it is 0.2038 s/step (17.2 ps/hr).

8 Load the simulation trajectory in VMD. Create two separate VDW representations for potassium (`resname POT`) and chloride (`resname CLA`) ions. For each of them, select Coloring Method: ColorID 4 (yellow) for Cl^- and 12 (lime) for K^+ ions.

9 Use the Molecule File Browser to load the same structure (`waterbox40-0.2M.pdb` and `waterbox40-0.2M.psf` files) again as a new molecule. Select this system in the Graphical Representations window, type `"water"` in the Selected Atoms box, and select coloring method ColorID 15 (iceblue) and drawing method MSMS with transparent material.

10 In Main window, double-click in column T of the molecule for which you created the ions' representations to make this molecule the `top` one. Run the animation and watch the ions gather in the center of the water box.

Sample snapshots of the ion concentration trajectory are shown in Fig. 6. The directory `tclBCmovies` contains MPEG movie `ionConcentration.mpg` that shows the process at 0.1 ps per frame.

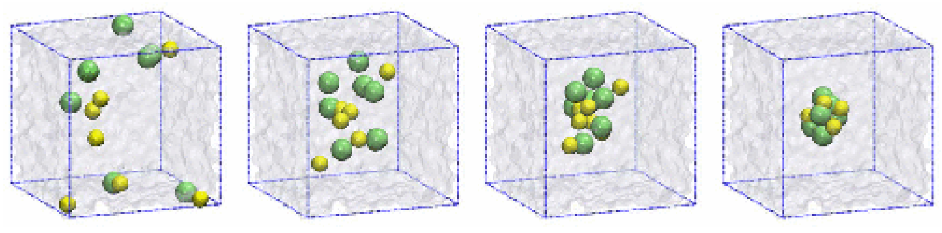


Figure 6: Snapshots of the ion concentration trajectory. The simulation cell boundaries are shown for reference.

2.4 Example 3: Improving efficiency

The above code can be made both more computationally efficient, as well as more concise and elegant. In our case, we are pushing only the ions, therefore all other atoms can be safely dropped using the statement

```
if { [lsearch $ionList $atomid] == -1 } {

  dropatom
  continue

}
```

This way all non-ion atoms will be dropped at the first step and never selected again by `nextatom` until `cleardrops is called`, thus making the code more efficient. Some extra time could be saved if we didn't need to continue pushing the atoms that are already inside the sphere:

```
    if { $rho < $sphereRadius } {
      dropatom  ;# don't process this atom in the future
      continue  ;# don't process this atom now, either
    }

    # time to time (every 100 steps in this example),
    # let's restore the complete list of atoms
    # to catch ions escaping from the sphere

    if { $step % 100 == 0 } { cleardrops }
```

Using the vector routines, instead of robust but lengthy (and therefore error-prone) code

```
  foreach { x0 y0 z0 } $sphereCenter { break }
  foreach { x y z } $rvec { break }

  set rho [expr sqrt(($x-$x0)*($x-$x0) + ($y-$y0)*($y-$y0) + \
    ($z-$z0)*($z-$z0))] ;# find the distance between two points

  # calculate the force components

  set forceX [expr -$maxForce * ($x-$x0) / $rho]
  set forceY [expr -$maxForce * ($y-$y0) / $rho]
  set forceZ [expr -$maxForce * ($z-$z0) / $rho]

  addforce "$forceX $forceY $forceZ" ;# apply the force
```

we could write briefly and clearly:

```
  set relativePosition [vecsub $rvec $sphereCenter]
  set rho              [veclength $relativePosition]
  set forceVector      [vecscale $relativePosition \
    [expr -$maxForce / $rho]]
  addforce $forceVector
```

or even more briefly (though maybe less clearly):

```
  set relativePosition [vecsub $rvec $sphereCenter]
  addforce [vecscale [vecscale $relativePosition \
    [expr -$maxForce / [veclength $relativePosition]]]]
```

Another trick can simplify a `tclBC` script if you need to process all atoms of a certain type, as K^+ and Cl^- in the above example. Since such atoms have unique masses, you could avoid reading the PDB file, and make the list of atoms by writing, e.g.:

```
  while { [nextatom] } {
    if { [getmass] > 35 && [getmass] < 40 } { ... }
  }
```

Likewise, you could use `if { [getcharge] }` as a selection criterion. Just make sure that no other atoms could match the conditions of the selection.

1 Open the file `concentrateIonsBrief.tcl`, look through the code, partly explained above:

```
# we will use the following variable to calculate and print the
# average number of ions found outside the sphere at each step

set avgNumIons 0

set forceCoef [expr -$maxForce/$sphereRadius]

wrapmode cell

##################################################

proc calcforces {step unique} {

  global sphereCenter sphereRadius maxForce avgNumIons

  if { $step > 0 && $step % 100 == 0 } {

    # Calculate and print the average number ions found outside
    # the sphere

    set avgNumIons [expr $avgNumIons / 100.]
    print "Step $step, average number of ions outside \
      the sphere: $avgNumIons"
    set avgNumIons 0

    cleardrops
  }

  while {[nextatom]} {

    if { [getmass] < 35 || [getmass] > 40 } {
      dropatom ;# not a K+ or Cl-, forget about it
      continue
    }

    # vector between the ion and the sphere's center

    set relativePosition [vecsub [getcoord] $sphereCenter]
    set rho              [veclength $relativePosition]

    if { $rho > $sphereRadius } {
      addforce [vecscale $relativePosition [expr -$maxForce/$rho]]
      incr avgNumIons
    } else {
      dropatom ;# this ion is already inside the sphere
    }
  }
}
```

2 Note that in this code, the ions already inside the sphere are no longer pushed and are dropped. Can they escape from the sphere? [1]

3 Run a NAMD simulation with file `eq04.concentrateIonsBrief.namd` that calls the `tclBC` script `concentrateIonsBrief.tcl`. If unable to do so, find the exemplary output `concentrateIonsBrief.log` and `concentrateIonsBrief.dcd` in directory `../IONS_OUTPUT_EXAMPLE`.

4 Open file `concentrateIonsBrief.log`, find the computation speed of the optimized script, compare it to the speed you found when using the original script. In our examples, the speeds are 0.1330 s/step (26.8 ps/hr) versus 0.2038 s/step (17.2 ps/hr).

5 In the file `concentrateIonsBrief.log`, find the lines that look like this: `TCL: Step 100, average number of ions outside the sphere:  13.0`. Plot the number of ions against simulation step. Hint: under a UNIX/Linux system, you can type `grep outside eq04.concentrateIonsBrief.log | awk '{print $3 " " $11}' > numIons.dat`, and import the data from the file `numIons.dat` into `xmgrace` or another technical graphics program. Open the `xmgrace` file `IONS_OUTPUT_EXAMPLE/numIons.agr` to see a sample plot.

[1]Answer: yes, ions can escape, but they will be pushed back once the command `cleardrops` is executed.

2.5 Example 4: Imposing a Shear Flow

As previously mentioned, the main feature of TclBC scripts is their ability to apply geometry-based forces. Using a TclBC script, it is possible to impose arbitrary fields of forces, thus imitating complex geometries and interactions while keeping the molecular system relatively small and simple. For example, a solution containing DNA in a microfluidic channel undergoes a shear flow, so that parallel layers of the liquid move with different velocities. We can simulate the shear flow by applying force to oxygen atoms of water molecules found within thin layers in XY planes at and around $z =$ `zLo` and $z =$ `zHi`, which variables will be set in the NAMD configuration file.

1 Change to the directory `tclBCexamples/SHEAR`, open file `eq04.shear.namd`, scroll to its end.

The following initialization is made in the NAMD configuration file:

```
tclBC on
tclBCScript {
  set zLo          -15. ;# lower plane where forces are applied
  set zHi           15. ;# top plane where forces are applied
  set dz             3. ;# half-width of the layer
  set TOL            6. ;# drop atoms that far from either layer
  set force          5.
  set pdbSource      dsDNA6_solv.pdb
  set tclBCScript    shear.tcl
  source $tclBCScript
}
tclBCArgs { }
```

2 Open the `tclBC` script `shear.tcl`, read it while paying attention to the comments (the same script is shown below).

```
set atomList {} ;# a list of atom IDs of water oxygens
set atomID 1   ;# in NAMD, atom IDs are 1-based numbers

set inStream [open $pdbSource r] ;# opening the PDB file

# reading the PDB file line after line:
foreach line [split [read $inStream] \n] {

  # symbols 0-3 should be ATOM or HETA
  set string1 [string range $line 0 3]

  # symbols 13-16 contain the atom's name
  set string2 [string range $line 13 16]
  # trimming the atom's name
  set string2 [string trim $string2]

  if { [string equal $string1 {ATOM}] || \
   [string equal $string1 {HETA}] } {

    # so it's a valid atom entry; let's see if
    # this is a water's oxygen

    if { [string equal $string2 {OH2}] } {

      # yes it is; append its index to the list

      lappend atomList $atomID
    }
    incr atomID
  }
}
close $inStream

puts "[llength $atomList] water oxygens atoms found"

set totalTorque 0.

wrapmode cell

####################################################

proc calcforces {step unique} {

  global atomList shearStress zLo zHi dz TOL force

  # the value of a global variable will be stored
  # between calls to this procedure, while a local
  # variable would be lost

  global totalTorque

  if { $step % 100 == 0 } {

    if { $totalTorque != 0. } {
      print "Step $step, total torque applied: $totalTorque"
      set totalTorque 0.
    }
    cleardrops
  }
  while {[nextatom]} {

    # check if this atom should be considered at all

    if { [lsearch $atomList [getid]] == -1 } {

      dropatom
      continue
    }

    # now check if it's within bounds zLo to zHi

    foreach { x y z } [getcoord] { break } ;# get coordinates

      if { $z >= [expr {$zLo-$dz}] && $z <= [expr {$zLo+$dz}] } {

      addforce "[expr {-$force}] 0.0 0.0"
      set totalTorque [expr {$totalTorque - $force * $z}]

    } elseif { $z >= [expr {$zHi-$dz}] && \
               $z <= [expr {$zHi+$dz}] } {

      addforce "$force 0.0 0.0"
      set totalTorque [expr {$totalTorque + $force * $z}]

    } elseif { ( $z >= [expr {$zLo-$TOL}] && \
                 $z <= [expr {$zLo+$TOL}] ) || \
               ( $z >= [expr {$zHi-$TOL}] && \
                 $z <= [expr {$zHi+$TOL}] ) } {

      continue ;# keep an eye on it, it may come closer next time

    } else {

      dropatom ;# this atom is too far, forget about it for now

    }
  }
}
```

3 Run a NAMD simulation with configuration file `eq04.shear.namd` that calls the `tclBC` script `shear.tcl`. Use the sample output from directory `../SHEAR_OUTPUT_EXAMPLE` if necessary.

4 View the simulation trajectory in VMD. Use representation style Licorice for both DNA strands, coloring method ColorID 11 (purple) for `segid`

`ADNA` and ColorID 7 (green) for `segid BDNA`. For selection `water`, use style Lines, ColorID 15 (iceblue).

5 Why were `zLo` and `zHi` not set to be the lower and upper boundaries of the system?[2]

6 Open the output file `eq04.shear.log`, find a line with the total torque output. Plot the total torque versus simulation step. Use the hint from the previous section to extract the data. Open `xmgrace` file `SHEAR_OUTPUT_EXAMPLE/torque.agr` to see a sample plot.

7 Run the same simulation on a cluster. How many times is the "total torque" printed at each checkpoint? Can you modify the TclBC script to print a system-wide total just once each time?[3]

Sample snapshots of the simulation trajectory are shown in Fig. 7. MPEG movies (found in directory `tclBCmovies`) `shearSlow.small.mpg` and `shearSlow.big.mpg` show the trajectory animation at 0.1 ps per frame. At this slow framerate, the movement of water layers is clearly visible. In movies `shearFast.small.mpg` and `shearFast.big.mpg`, the frame rate is 1 ps per frame, which allows one to see the slow rotation and dissociation of DNA in the shear flow.

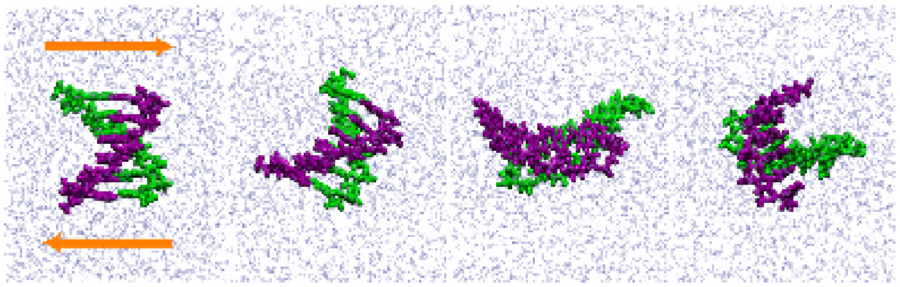


Figure 7: Snapshots of the shear flow. Arrows in the leftmost frame show the direction and place where the shear forces are applied.

[2]Answer: The system has periodic boundary conditions, therefore its apparent top and lower boundaries are actually the same plane.

[3]Answer: The torque is printed as many times as there are patches containing atoms to which the force is applied. No, it's impossible to collect this information from all patches in a `tclBC` script. Each instance of the script can only access information about the atoms handled by one processor.