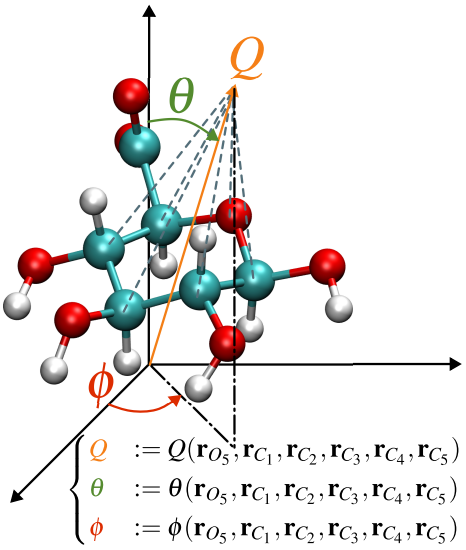
University of Illinois Urbana-Champaign
NIH Resource for Macromolecular Modeling and Visualization
Beckman Institute for Advanced Science and Technology
Computational Biophysics Workshop

Customizing Collective Variables in NAMD



NAMD Developers: **Haochuan Chen**
Emad Tajkhorshid

June 2024

A current version of this tutorial is available at
http://www.tcbg.illinois.edu/Training/Tutorials/

Contents

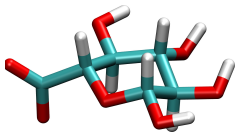


Figure 1: Structure of the anion of the glucuronic acid.

1 Introduction

A molecular dynamics (MD) simulation generates trajectories consisting of the evolution of atomic Cartesian coordinates and velocities over the simulation time. It can be imagined that not all degrees of freedom (DOFs) are relevant to our studies of the underlying biomolecular processes. These processes, such as conformational changes of biomolecules and membrane permeations, often involves rare events. The concept of collective variable (CV) [1;2] is useful for identifying relevant DOFs for analyzing and sampling the rare events. In general, a CV, ξ is a function of the atomic Cartesian coordinates, $\mathbf{r}$, namely

$$\xi = \xi(\mathbf{r}_1, \cdots, \mathbf{r}_N)$$

where N is the number of atoms. In addition, in many biased sampling methods, including steered MD, [3] metadynamics [4] and umbrella sampling, [5] an external force $F_{\text{bias}}(\xi)$, is attached to ξ, which is exerted on the underlying atoms by using the gradient of ξ with respect to atomic Cartesian coordinates,

$$\mathbf{F}_{\text{bias}}(\mathbf{r}_k) = F_{\text{bias}}(\xi)\ \nabla_{\mathbf{r}_k}\xi(\mathbf{r}_1, \cdots, \mathbf{r}_N)$$

There are various ways to compute CVs on-the-fly during a simulation in NAMD, [6] and using Colvars [7] is one of them. There are a plethora of predefined CV functions available in the Colvars module in NAMD, including dihedral angles, distances between atom groups, path CVs [8] and more. However, we may want to design a complex CV function that are not in the existing Colvars implementation. In such case, we have to (i) utilize the TCL scripting interface of NAMD and Colvars, or (ii) modify the C++ source code of Colvars to support our customization. In general, the approach (ii) could have better performance, but (i) is better for debugging and prototyping. In this tutorial, we will use the Cremer-Pople parameter [9] as an example to show how to implement a new CV in NAMD, and run a targeted MD simulation along the parameters to bias a glucuronic acid towards the $^1C_4$ conformation. It ought to be noted that in this tutorial, the targeted MD is performed along the new CV that we will implement, not the root-mean-square deviation (RMSD) with respect to a reference.

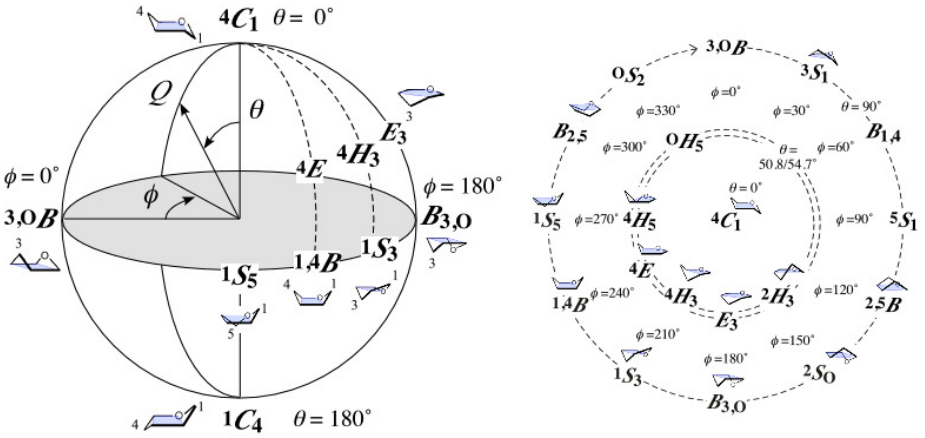


Figure 2: Schematic representation of describing the conformation of a pyranose ring using the Cremer-Pople parameter (Q, θ, ϕ). The figure was made by Shinya Fushinobu.

2 Cremer-Pople Parameters

2.1 Background

The Cremer-Pople parameter is a set of CVs describing the puckering of cyclic compounds, which uses $N - 3$ CVs to describe the N-member nonaromatic monocyclic rings. These CVs has been extensively used to characterize the conformations of carbohydrates in cyclic form. For example, the possible conformations of a six-membered pyranose ring can be represented in a polar coordinate system (Q, θ, ϕ), as shown in Figure 2 [1].

2.2 Calculation

In this tutorial, we focus on the case $N = 6$. The calculation of (Q, θ, ϕ) is summerized as follows:

Center the Cartesian coordinate. Assume the vectors from $\mathbf{r}_1$ to $\mathbf{r}_6$ are the positions of O1 and C1–C5 atoms, the centered position vectors are calculated as

$$\mathbf{R}_j = \mathbf{r}_j - \frac{1}{N}\sum_{k=1}^{N}\mathbf{r}_k \tag{1}$$

[1] URL: `http://enzyme13.bt.a.u-tokyo.ac.jp/CP/`

Determine the orientation of the mean plane. The normal of the mean plane, $\mathbf{n}$, can be determined by

$$\hat{\mathbf{n}} = \frac{\mathbf{R}' \times \mathbf{R}''}{|\mathbf{R}' \times \mathbf{R}''|} \tag{2}$$

where the two vectors $\mathbf{R}'$ and $\mathbf{R}''$ are from

$$\begin{cases} \mathbf{R}' &= \sum_{j=1}^{N} \mathbf{R}_j \sin\left[2\pi(j-1)/N\right] \\ \mathbf{R}'' &= \sum_{j=1}^{N} \mathbf{R}_j \cos\left[2\pi(j-1)/N\right] \end{cases} \tag{3}$$

Project the centered position vectors to the normal. The projections, z_j, is computed as

$$z_j = \mathbf{R}_j \cdot \hat{\mathbf{n}} \tag{4}$$

Determine the spherical coordinates. The spherical coordinates can be solved by the following linear system when $N = 6$,

$$\begin{cases} Q\sin\theta\sin\phi &= -\sqrt{\frac{2}{N}}\sum_{j=1}^{N} z_j \sin\left[\frac{2\pi}{N}2(j-1)\right] = -\sqrt{\frac{2}{N}}A \\ Q\sin\theta\cos\phi &= \sqrt{\frac{2}{N}}\sum_{j=1}^{N} z_j \cos\left[\frac{2\pi}{N}2(j-1)\right] = \sqrt{\frac{2}{N}}B \\ Q\cos\theta &= \sqrt{\frac{1}{N}}\sum_{j=1}^{N}(-1)^{j-1} z_j = \sqrt{\frac{1}{N}}C \end{cases} \tag{5}$$

and the solution should be

$$\begin{cases} Q &= \sqrt{\frac{1}{N}\left(2A^2 + 2B^2 + C^2\right)} \\ \phi &= \arctan 2\left(-A, B\right) \\ \theta &= \arccos\frac{C}{\sqrt{2A^2 + 2B^2 + C^2}} \end{cases} \tag{6}$$

3 NAMD Implementation

The NAMD simulation workflow is shown in Figure 3. The Colvars module accepts the total forces and positions of specified atoms, and computes the biasing forces acting on these atoms for the integration of next step. The following two subsections will describe how to implement the calculations of Cremer-Pople parameters from Eq. (1) to Eq. (6). The first subsection focuses on the TCL scripting interface when using NAMD with Colvars, and the second one tries to improve the performance by porting the TCL code to C++ in Colvars. Both subsections assume that the readers have known basic NAMD simulation techniques including equilibrium simulations and how to enable Colvars.

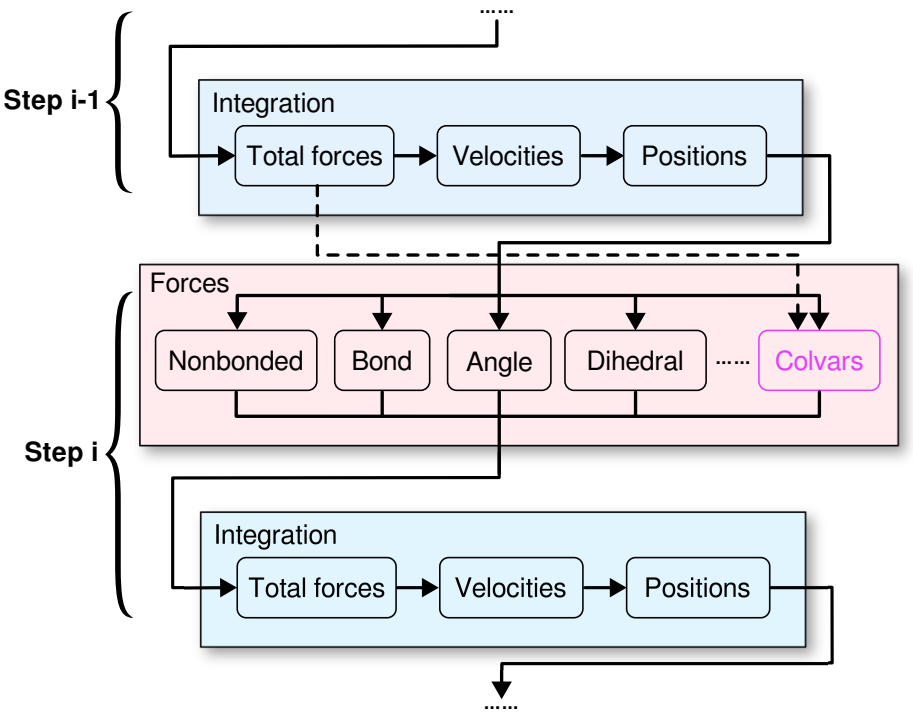


Figure 3: Schematic representation of the workflow of NAMD when enabling Colvars.

3.1 TCL Scripting

3.1.1 Load TCL procedures from a script.

Under the hood, to compute the CVs using TCL, NAMD loads a TCL script, and then Colvars runs specific TCL procedures (also known as functions in other programming languages) from the script every time step. The cruxes here are (i) how to load the TCL script file and (ii) what the signatures of the TCL

procedures should be. Assuming that the TCL script is `pucker.tcl` and locates in the same directory of the NAMD configuration file, the answer to (i) is simply adding the following lines of code before any of the `run` command in the NAMD configuration file (lines starting with # are code comments):

```
# Other NAMD simulation options

# Colvars configuration
colvars       on
colvarsConfig pucker_tcl.colvars

load          ./pucker.tcl
# run 100000
```

The configuration above also assumes that the Colvars configuration file is `pucker_tcl.colvars`. As for (ii), we are going to used the `scriptedFunction`[2] feature in Colvars, and it depends on the names of the Colvars components. In our implementation of Cremer-Pople parameters, there should be three Colvars components corresponding to Q, θ and ϕ, respectively. A minimal code example is shown below

Listing 1: pucker_tcl.colvars

```
# pucker.colvars
colvar {
  name                  Q
  scriptedFunction      cpQ
  cartesian {
    atoms {
      atomnumbers { 7  1  8  12  16  5 }
    }
  }
}

colvar {
  name                  theta
  scriptedFunction      cptheta
  cartesian {
    atoms {
      atomnumbers { 7  1  8  12  16  5 }
    }
  }
}

colvar {
```

[2]URL: `https://colvars.github.io/colvars-refman-namd/colvars-refman-namd.html#colvar|scriptedFunction`

```
  name                  phi
  scriptedFunction      cpphi
  cartesian {
    atoms {
      atomnumbers { 7  1  8  12  16  5 }
    }
  }
}
```

The three Colvars components feature three different parameters of `scriptedFunction`, namely `cpQ`, `cptheta` and `cpphi`. The `cartesian` blocks specifies the atomic coordinates necessary for the calculation of the Cremer-Pople parameters, and the `atomnumbers` blocks list the atom serial numbers of O1, C1, C2, C3, C4, and C5, which can be found in the second column of the input PDB file for NAMD simulations, as shown in Figure 4. As described in

```
ATOM      1  C1  BGLC    1      -1.041   1.300   0.283  1.00  0.00      CARB C
ATOM      2  H1  BGLC    1      -1.108   1.189   1.392  1.00  0.00      CARB H
ATOM      3  O1  BGLC    1      -1.929   2.326  -0.085  1.00  0.00      CARB O
ATOM      4  HO1 BGLC    1      -1.775   2.496  -1.020  1.00  0.00      CARB H
ATOM      5  C5  BGLC    1       1.229   0.586   0.412  1.00  0.00      CARB C
ATOM      6  H5  BGLC    1       1.179   0.498   1.519  1.00  0.00      CARB H
ATOM      7  O5  BGLC    1       0.320   1.589  -0.061  1.00  0.00      CARB O
ATOM      8  C2  BGLC    1      -1.445  -0.007  -0.379  1.00  0.00      CARB C
ATOM      9  H2  BGLC    1      -1.326   0.108  -1.484  1.00  0.00      CARB H
ATOM     10  O2  BGLC    1      -2.807  -0.331  -0.126  1.00  0.00      CARB O
ATOM     11  HO2 BGLC    1      -3.056  -0.059   0.768  1.00  0.00      CARB H
ATOM     12  C3  BGLC    1      -0.523  -1.119   0.078  1.00  0.00      CARB C
ATOM     13  H3  BGLC    1      -0.609  -1.224   1.186  1.00  0.00      CARB H
ATOM     14  O3  BGLC    1      -0.881  -2.369  -0.498  1.00  0.00      CARB O
ATOM     15  HO3 BGLC    1      -1.771  -2.619  -0.224  1.00  0.00      CARB H
ATOM     16  C4  BGLC    1       0.904  -0.748  -0.251  1.00  0.00      CARB C
ATOM     17  H4  BGLC    1       0.994  -0.639  -1.359  1.00  0.00      CARB H
ATOM     18  O4  BGLC    1       1.809  -1.757   0.175  1.00  0.00      CARB O
ATOM     19  HO4 BGLC    1       2.618  -1.648  -0.359  1.00  0.00      CARB H
ATOM     20  C6  BGLC    1       2.610   1.101   0.013  1.00  0.00      CARB C
ATOM     21  O61 BGLC    1       3.457   1.114   0.943  1.00  0.00      CARB O
ATOM     22  O62 BGLC    1       2.744   1.433  -1.194  1.00  0.00      CARB O
```

Figure 4: The glucuronic acid molecule section in the PDB file. The numbers marked in red are the atom serial numbers used for defining the Cremer-Pople parameters in Colvars.

the Colvars manual [3], to calculate the CV values, Colvars would call `calc_cpQ`, `calc_cptheta` and `calc_cpphi` for the three components, respectively. In other words, we should have the following TCL procedure signatures in `pucker.tcl`,

Listing 2: pucker.tcl

[3] URL: `https://colvars.github.io/colvars-refman-namd/colvars-refman-namd.html#colvar|scriptedFunction`

```tcl
# pucker.tcl
proc calc_cpQ {args} {
  # Implementation will be detailed below
}

proc calc_cptheta {args} {
  # Implementation will be detailed below
}

proc calc_cpphi {args} {
  # Implementation will be detailed below
}
```

The three TCL procedures are expected to accept a list of atomic coordinates from `args` and then return floating point values representing Q, θ and ϕ. The `args` is 2D TCL list (or list of lists), and the first element in the list corresponds to a list of atomic coordinates in `xyz...xyz` order. The following TCL commands exemplifies how to retrieve the atomic coordinates (it ought to be noted that the array starts with index 0):

Listing 3: pucker.tcl

```tcl
proc calc_cpQ {args} {
  # Get the z coordinate of 4th atom
  set z4 [lindex [lindex $args 0] [expr (4-1)*3+2]]
  # Get the x coordinate of 6th atom
  set x6 [lindex [lindex $args 0] [expr (6-1)*3+0]]
}
```

A schematic representation of the TCL procedure requirements above is shown in Figure 5.

3.1.2 Implement the TCL procedures.

The implementation of the TCL procedures can be broken down into small functions to do each of the tasks in Section 2.2:

Center the Cartesian coordinate. The calculation of R_j could be implemented by as follows:

Listing 4: pucker.tcl

```tcl
# Shift the atom coordinates to the origin
proc calc_Rj {args} {
  # Number of atoms
  set num_atoms [expr [llength $args] / 3]
  # Center of geometry
```

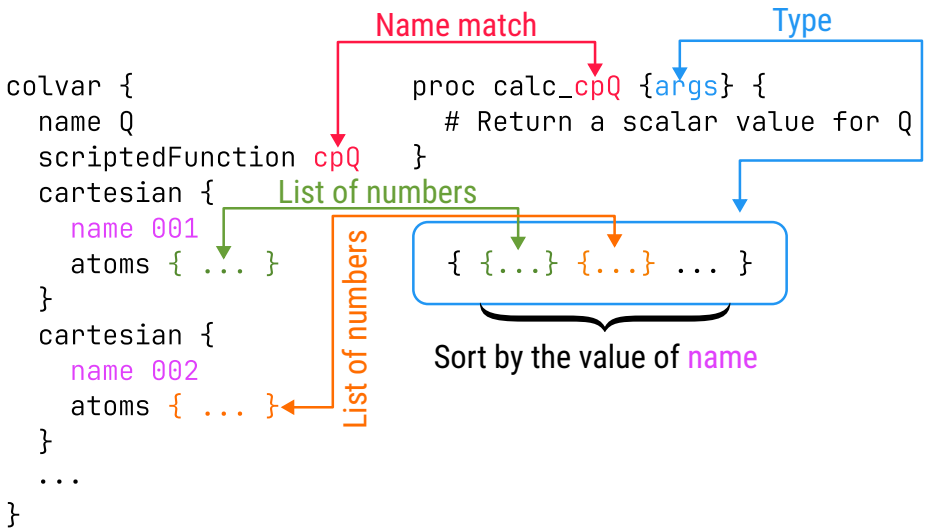


Figure 5: Schematic representation of the TCL procedure requirements used by `scriptedFunction` in Colvars.

```
6     set cog_x 0.0
7     set cog_y 0.0
8     set cog_z 0.0
9     for {set i 0} {$i < $num_atoms} {incr i} {
10      set i_atom [expr $i * 3]
11      set cog_x [expr $cog_x + [lindex $args [expr $i_atom + 0]]]
12      set cog_y [expr $cog_y + [lindex $args [expr $i_atom + 1]]]
13      set cog_z [expr $cog_z + [lindex $args [expr $i_atom + 2]]]
14    }
15    set cog_x [expr $cog_x / $num_atoms]
16    set cog_y [expr $cog_y / $num_atoms]
17    set cog_z [expr $cog_z / $num_atoms]
18    # R_j = r_j - R_cog
19    set result [list]
20    for {set i 0} {$i < $num_atoms} {incr i} {
21      set i_atom [expr $i * 3]
22      set R_jx [expr [lindex $args [expr $i_atom + 0]] - $cog_x]
23      set R_jy [expr [lindex $args [expr $i_atom + 1]] - $cog_y]
24      set R_jz [expr [lindex $args [expr $i_atom + 2]] - $cog_z]
25      lappend result [list $R_jx $R_jy $R_jz]
26    }
27    return $result
28  }
```

`args` here is a 1D list consisting of atomic Cartesian coordinates. `cog_x`, `cog_y`

and `cog_z` are the center of geometry. To facilitate the further calculations, the code also "reshape" the array from the 1D list into a $N \times 3$ 2D list where the i-th element corresponds to the i-th atom coordinates.

Calculate the projection of $\mathbf{R}_j$ onto the mean plane. The code here just fuses the calculation of the normal vector of the mean plane $\hat{\mathbf{n}}$ and the projection of $\mathbf{R}_j$ on it into a single procedure, with $\mathbf{R}$ as the input arguments. The code is listed as follows:

Listing 5: pucker.tcl

```
# Calculate z_j, args are R_j
set pi 3.14159265358979323846
proc calc_zj {args} {
  # Calculate R' and R''
  set N [llength $args]
  set Rpx 0.0
  set Rpy 0.0
  set Rpz 0.0
  set Rppx 0.0
  set Rppy 0.0
  set Rppz 0.0
  global pi
  # Calculate R' and R''
  for {set i 0} {$i < $N} {incr i} {
    set factor [expr (2.0 * $pi * $i) / $N]
    set sin_f [expr sin($factor)]
    set cos_f [expr cos($factor)]
    set R_j [lindex $args $i]
    set R_jx [lindex $R_j 0]
    set R_jy [lindex $R_j 1]
    set R_jz [lindex $R_j 2]
    set Rpx [expr $Rpx + $sin_f * $R_jx]
    set Rpy [expr $Rpy + $sin_f * $R_jy]
    set Rpz [expr $Rpz + $sin_f * $R_jz]
    set Rppx [expr $Rppx + $cos_f * $R_jx]
    set Rppy [expr $Rppy + $cos_f * $R_jy]
    set Rppz [expr $Rppz + $cos_f * $R_jz]
  }
  # Calculate the normal vector (step 1): R' cross R''
  set cross_x [expr $Rpy * $Rppz - $Rpz * $Rppy]
  set cross_y [expr $Rpz * $Rppx - $Rpx * $Rppz]
  set cross_z [expr $Rpx * $Rppy - $Rpy * $Rppx]
  # Calculate the normal vector (step 2): |R' cross R''|
  set n_norm_factor [expr sqrt(\
    $cross_x * $cross_x + $cross_y * $cross_y + \
```

```
36      $cross_z * $cross_z)]
37    set n_x [expr $cross_x / $n_norm_factor]
38    set n_y [expr $cross_y / $n_norm_factor]
39    set n_z [expr $cross_z / $n_norm_factor]
40    set z [list]
41    set n [list]
42    # Project R_j onto the normal vector
43    for {set i 0} {$i < $N} {incr i} {
44      set R_j [lindex $args $i]
45      set R_jx [lindex $R_j 0]
46      set R_jy [lindex $R_j 1]
47      set R_jz [lindex $R_j 2]
48      set z_j [expr $R_jx * $n_x + $R_jy * $n_y + $R_jz * $n_z]
49      lappend z $z_j
50    }
51    return $z
52  }
```

Determine the spherical coordinates. We break the procedure into two steps for better code reusability. The first step is the calculation of A, B and C from all z:

Listing 6: pucker.tcl

```
1   # Calculate A, B and C from z_j
2   proc calc_ABC {args} {
3     set N [llength $args]
4     set A 0.0
5     set B 0.0
6     set C 0.0
7     global pi
8     for {set i 0} {$i < $N} {incr i} {
9       set factor [expr 2.0 * $pi / $N * 2 * $i]
10      set sin_f [expr sin($factor)]
11      set cos_f [expr cos($factor)]
12      set A [expr $A + [lindex $args $i] * $sin_f]
13      set B [expr $B + [lindex $args $i] * $cos_f]
14      set C [expr $C + [lindex $args $i] * ((-2.0) * ($i % 2) + 1.0)]
15    }
16    return [list $A $B $C]
17  }
```

The second step is then the calculation of (Q, θ, ϕ) by calling all the procedures above:

Listing 7: pucker.tcl

```
proc calc_cptheta {args} {
  # args is a list of atom coordinates of length 3*N
  set R_j [calc_Rj {*}[lindex $args 0]]
  set z_j [calc_zj {*}$R_j]
  lassign [calc_ABC {*}$z_j] A B C
  global pi
  set cptheta [expr 180.0 / $pi * \
    acos($C / sqrt(2.0 * $A * $A + 2.0 * $B * $B + $C * $C))]
  return $cptheta
}

proc calc_cpphi {args} {
  # args is a list of atom coordinates of length 3*N
  set R_j [calc_Rj {*}[lindex $args 0]]
  set z_j [calc_zj {*}$R_j]
  lassign [calc_ABC {*}$z_j] A B C
  global pi
  set cpphi [expr 180.0 / $pi * (atan2(-$A, $B))]
  if {$cpphi < 0} {
    set $cpphi [expr $cpphi + 360.0]
  }
  return $cpphi
}

proc calc_cpQ {args} {
  # args is a list of atom coordinates of length 3*N
  set R_j [calc_Rj {*}[lindex $args 0]]
  set z_j [calc_zj {*}$R_j]
  lassign [calc_ABC {*}$z_j] A B C
  set q [expr sqrt((2.0 * $A * $A + 2.0 * $B * $B + $C * $C) / 6.0)]
  return $q
}
```

The full source code can be found in the `pucker.tcl` in the tutorial files.

3.1.3 Simulation and Performance

With the `pucker.tcl` above, we could try running our simulation printing the values of (Q, θ, ϕ) every 1000 step by adding the following lines to the beginning of `pucker.colvars`:

```
Colvarstrajfrequency    1000
Colvarsrestartfrequency 10000
```

If we run the simulation, then we can get the (Q, θ, ϕ) in the `<outputname>.colvars.traj` file (the exact filename depends on the `outputname` setting in the NAMD configuration file), which may contains

```
# step         theta                  phi                    Q
           0    1.63816558478516e+01   8.53793052984263e+01   6.12507333001755e-01
        1000    1.21118794619448e+01   5.05819956466481e+01   5.59147282106055e-01
        2000    1.36943489028639e+01   5.52729603443354e+01   6.25136065248863e-01
        3000    2.85165637199508e+00  -1.67644190362664e+02   6.02365951802746e-01
        4000    1.10522913896010e+01   1.12666490322298e+02   6.37176507034846e-01
        5000    1.43026502793403e+01   3.04050362556335e+01   5.86074768967032e-01
```

You are encouraged to check if the code computes the (Q, θ, ϕ) correctly by comparing the Colvars TCL implementation with other existing implementations, such as the online calculator on http://enzyme13.bt.a.u-tokyo.ac.jp/CP/. It ought to be noted that there could be marginal numerical difference between the TCL scripting implementation and the online calculator, as the latter uses the PDB file content that has less precision as the input.

So far, we could satisfy with the TCL scripting solution. However, the execution of the TCL script hurts the overall performance. From the NAMD log file, we may see

Listing 8: Performance of the TCL scripting implementation

```
Info: Benchmark time: 4 CPUs 0.00163979 s/step 105.379 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.00163778 s/step 105.509 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.00163713 s/step 105.551 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.00164208 s/step 105.233 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.00164251 s/step 105.205 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.001661 s/step 104.034 ns/day 0 MB memory
```

which indicates that the simulation is about 105 ns/day. If we disable the Colvars and do not perform the calculation of (Q, θ, ϕ), we may get

Listing 9: Performance of simulation without Colvars

```
Info: Benchmark time: 4 CPUs 0.000444843 s/step 388.452 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000447507 s/step 386.14 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000449203 s/step 384.681 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000457953 s/step 377.332 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000450557 s/step 383.526 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000456553 s/step 378.488 ns/day 0 MB memory
```

which approximates to 380 ns/day. It seems by simply calculating (Q, θ, ϕ) on-the-fly the simulation speed drops more than two third. The slowdown is caused mainly by the fact that TCL is a scripting language and not optimized ahead-of-time like compiled languages such as C++. In the next subsection, we will illustrate how to modify the C++ source code of Colvars to add the calculation of Cremer-Pople parameters there to improve the performance.

3.2 C++ within Colvars

The Colvars developer documentation can be found in http://colvars.github.io/doxygen/html/index.html. The steps of implementing a new CV can be

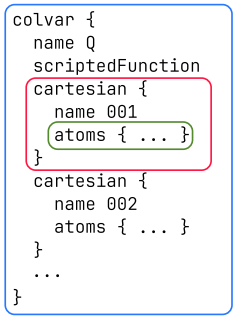


Figure 6: Colvars configuration hierarchy. The C++ classes `colvar`, `cvc` (or its derived classes) and `colvaratoms` correspond to the blue, red and green blocks of the configurations.

found in https://colvars.github.io/doxygen/html/classcolvar_1_1cvc.html. Here, we try to go through all the steps that are necessary to implement the new CVs. The source code of Colvars can be downloaded from the Github:

```
git clone https://github.com/Colvars/colvars.git
```

3.2.1 Modify the source code.

Before modifying the source code, we need to take a look at the structure of the Colvars configuration file. An example of the Colvars configuration block is shown in Figure 6. The outermost "`colvar`" block (the blue box in Figure 6) merely serves as a "container", which contains the inner "`cartesian`" blocks (the red box in Figure 6) and may modify the results from them. In other words, the inner "`cartesian`", also known as *components*[4], do the actual CV calculations. The "`colvar`" block can further modify the results from the component blocks. In Figure 6 the "`scriptedFunction`" is used for passing the results from the "`cartesian`" components to TCL procedures. There are many useful options[5] for the "`colvar`" blocks. This tutorial will only cover those that are relevant to our new CVs. A component can accepts the atomic coordinates (shown in the green box in Figure 6) as its inputs. The blue, red and green Colvars configuration blocks in Figure 6 correspond to the C++ classes `colvar`, `cvc` and `colvaratoms`, respectively. In summary, we need to implement new derived classes of `cvc` to calculate (Q, θ, ϕ). After the implementation, the new Colvars configuration blocks should look like

Listing 10: pucker_cpp.colvars

```
1  colvar {
2    name Q
```

[4]URL: `https://colvars.github.io/master/colvars-refman-namd.html#sec:cvc_list`
[5]URL: `https://colvars.github.io/master/colvars-refman-namd.html#sec:colvar`

```
  cpQ {
    atoms {
      atomnumbers { 7  1  8  12  16  5 }
    }
  }
}

colvar {
  name theta
  cptheta {
    atoms {
      atomnumbers { 7  1  8  12  16  5 }
    }
  }
}

colvar {
  name phi
  cpphi {
    atoms {
      atomnumbers { 7  1  8  12  16  5 }
    }
  }
}
```

Derive classes from `cvc`. The class definition of `cvc` can be found in `colvarcomp.h`. Our derived classes need to (i) read the atomic coordinates of the six atoms, and (ii) perform the calculations of Cremer-Pople parameters. For (i), the derived classes need to have their own "`init`" functions to know which atoms are specified. For (ii), the derived classes must have the "`calc_value`" functions implemented. We declare the three new derived classes in `colvarcomp_pucker.h`, namely `cpQ`, `cptheta` and `cpphi` as

Listing 11: colvarcomp_pucker.h

```
class colvar::cpQ : public colvar::cvc
{
protected:
  /// Atom group
  cvm::atom_group  *atoms = nullptr;
public:
  cpQ();
  virtual int init(std::string const &conf);
  virtual void calc_value();
};

```

```
12 class colvar::cptheta : public colvar::cvc
13 {
14 protected:
15   /// Atom group
16   cvm::atom_group  *atoms = nullptr;
17 public:
18   cptheta();
19   virtual int init(std::string const &conf);
20   virtual void calc_value();
21 };
22
23 class colvar::cpphi : public colvar::cvc
24 {
25 protected:
26   /// Atom group
27   cvm::atom_group  *atoms = nullptr;
28 public:
29   cpphi();
30   virtual int init(std::string const &conf);
31   virtual void calc_value();
32 };
```

The `atoms` pointers are used for tracking the positions of specified atoms inside the component blocks.

Implement the constructors of the derived classes. The class constructors, need to (i) set the name of the components, (ii) specify the type of the CV (either scalar or vector), and (iii) set the periodic boundary if necessary. The example code is shown below

Listing 12: colvarcomp_pucker.cpp

```
1  colvar::cpQ::cpQ() {
2    set_function_type("cpQ");
3    x.type(colvarvalue::type_scalar);
4  }
5
6  colvar::cptheta::cptheta() {
7    set_function_type("cptheta");
8    init_as_angle();
9  }
10
11 colvar::cpphi::cpphi() {
12   set_function_type("cpphi");
13   x.type(colvarvalue::type_scalar);
14   provide(f_cvc_periodic);
```

```
15    enable(f_cvc_periodic);
16    period = 360.0;
17    init_scalar_boundaries(0, 360.0);
18  }
```

The variable `x` is a protected member of the `cvc` class to store the result of the CV. The variable θ should always be in the range $[0°, 180°]$, so we call `init_as_angles()` to set it. The variable ϕ is periodic and has a period of 360°, so we need to setup `period` and the boundaries to make sure it is wrapped.

Implement the `init` functions. The `init` function is responsible for reading the atoms defined in the "`atoms {...}`" block. The example code below of `init` (i) calls the initialization function of the parent class, and (ii) read the definition of atoms:

Listing 13: colvarcomp_pucker.cpp

```
1  int colvar::cpQ::init(const std::string& conf) {
2    int error_code = cvc::init(conf);
3    atoms = parse_group(conf, "atoms");
4    if (!atoms || atoms->size() != 6) {
5      return error_code | COLVARS_INPUT_ERROR;
6    }
7    return error_code;
8  }
```

The implementation also checks the number of atoms specified, and returns an error if it is not exactly six, as we only implement the Cremer-Pople parameters for $N = 6$. The `init` functions of the `cptheta` and `cpphi` can be implemented in the same way.

Implement the `calc_value` functions. To simplify and reuse the code, we can port the `calc_ABC` TCL procedure to C++, and then call it from `cpQ::calc_value`, `cptheta::calc_value` and `cpphi::calc_value`:

Listing 14: colvarcomp_pucker.cpp

```
1  void colvar::cpQ::calc_value() {
2    cpABC result = calc_cpABC(*atoms);
3    cvm::real A = result.A;
4    cvm::real B = result.B;
5    cvm::real C = result.C;
6    x.real_value = std::sqrt((
7      2.0 * A * A + 2.0 * B * B + C * C) / 6.0);
8  }
9
10 void colvar::cptheta::calc_value() {
```

```
  cpABC result = calc_cpABC(*atoms);
  cvm::real A = result.A;
  cvm::real B = result.B;
  cvm::real C = result.C;
  x.real_value = 180.0 / M_PI *
    std::acos(C / std::sqrt(2.0 * A * A +
                            2.0 * B * B + C * C));
}

void colvar::cpphi::calc_value() {
  cpABC result = calc_cpABC(*atoms);
  cvm::real A = result.A;
  cvm::real B = result.B;
  cvm::real C = result.C;
  x.real_value = 180.0 / M_PI * std::atan2(-A, B);
  if (x.real_value < 0) {
    x.real_value += 360.0;
  }
}
```

The full implementation of `calc_cpABC` and relevant helper code can be found from `colvarcomp_pucker.h` and `colvarcomp_pucker.cpp` in the Github repository [6]. The new `colvarcomp_pucker.h` and `colvarcomp_pucker.cpp` are assumed to be in `colvars/src`.

Make the new `cvc` classes accessible from the `colvar` class. The `colvar` block ought to recognize our new `cvc`-derived classes, which can be done by (i) forward declaring the classes `cpQ`, `cptheta` and `cpphi` in `colvar.h`, and then (ii) modifying the `colvar::define_component_types()` function in `colvar.cpp` with `add_component_type`.

3.2.2 Compile NAMD with the updated Colvars.

Update the build configuration file. The Colvars build configuration file relevant to NAMD is `colvars/namd/colvars/src/Makefile.namd`. Since we have added `colvarcomp_pucker.{h,cpp}`, we need to modify the `Makefile.namd` to ensure the new code is compiled, which can be done by appending `\$(DSTDIR)/colvarcomp_pucker.o` to the `COLVARSLIB` variable.

Update the NAMD source code. The NAMD source code is bundled with the official version of Colvars, and we are going to replace that one with our modified version with the new CVs. Assuming that the NAMD source code is in `<namd_src>`, we can run the following command to update the Colvars code,

```
<colvars_src_dir>/update-colvars-code.sh ./
```

[6] URL: `https://github.com/HanatoK/colvars/tree/pucker/src`

where the `<colvars_src_dir>` refers to the Colvars source code location.

Build NAMD. Now we can follow the same instruction as `<namd_src>/notes.txt` to build NAMD (see the "Compiling NAMD" section). The only difference is that before running "`make`" we need to run "`make depends`" at first.

3.3 Simulation and Performance

Now we can try the newly built NAMD with the same system and the Colvars config as shown in Listing 10. The NAMD configuration file is modified as follows,

```
# Colvars configuration
colvars       on
colvarsConfig pucker_cpp.colvars
```

The performance of the C++ implementation of the Cremer-Pople parameters should look like

Listing 15: Performance of the TCL scripting implementation

```
Info: Benchmark time: 4 CPUs 0.000496346 s/step 348.144 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000497668 s/step 347.219 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000500817 s/step 345.036 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000498759 s/step 346.46 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000502659 s/step 343.772 ns/day 0 MB memory
Info: Benchmark time: 4 CPUs 0.000505699 s/step 341.705 ns/day 0 MB memory
```

which is almost three times faster than the TCL scripting implementation!

4 Targeted MD along the CV

If our initial conformation for simulating the glucuronic acid molecule is not the chair conformation, and we want to perform a targeted MD biasing glucuronic acid towards the ${}^1C_4$, what should we do? In other words, we want to apply external forces on the six atoms of the ring, and make θ close to 180° as much as possible. The force acting on the six atoms can be derived from the generalized force acting on θ as follows,

$$\begin{cases} V_{\mathrm{TMD}}(\theta) & = \dfrac{1}{2}k\left(\theta - \theta_c(t)\right)^2 \\ F_{\mathrm{TMD}}(\theta) & = -\dfrac{\mathrm{d}V_{\mathrm{TMD}}(\theta)}{\mathrm{d}\theta} \\ \mathbf{F}_{\mathrm{TMD}}(\mathbf{r}_j) & = F_{\mathrm{TMD}}(\theta)\nabla_{\mathbf{r}_j}\theta \end{cases} \tag{7}$$

where $V_{\mathrm{TMD}}(\theta)$ is the targeted MD biasing potential, $F_{\mathrm{TMD}}(\theta)$ is the biasing force along θ, k is a spring force constant, $\theta_c(t)$ is a moving reference value that gradually evolves towards 180° during simulation, and $\nabla_{\mathbf{r}_j}$ is the gradient of θ with respect to the Cartesian coordinates of the atoms. At first glance, it seems that we have to implement $V_{\mathrm{TMD}}(\theta)$, $F_{\mathrm{TMD}}(\theta)$ and all gradients of θ with respect to all six atom positions. However, Colvars has already supported targeted MD along CVs, and the biasing forces to the atoms are obtained by the chain rule,[10] namely,

$$\mathbf{F}(\mathbf{r}_i) = \sum_{j=1}^{M} F(\xi_j)\frac{\partial \xi_j(\mathbf{r})}{\partial \mathbf{r}_i} \tag{8}$$

where $\mathbf{F}(\mathbf{r}_i)$ is the three-dimensional force acting on atom i, M is the number of CVs, and $F(\xi_j)$ is the biasing force acting on CV ξ_j. In other words, all we need is to tell Colvars how to calculate the gradients of θ, and Colvars shall handle the rest.

4.1 Gradient of the CV

As you can imagine, the calculation of gradients could be complicated. Technically there are several ways that you could try to make such task easier, including (i) using a math library supporting automatic differentiation in Colvars, (ii) using a computer algebra system (CAS) like Maxima or Mathematica to do the calculation, and (iii) asking a multimodal language model, for example, ChatGPT to do it for you. Unfortunately, the author does not have good knowledge of (i) and (ii), and is not sure if a language model can give the correct answer, so let us derive the gradient of one component, and then complete the remaining by induction.

We start from the derivative of z_j with respect to r_{ix}, where r_{ix} is the x

coordinate of atom i, as follows,

$$\begin{aligned}\frac{\partial z_j}{\partial r_{ix}} &= \frac{\partial}{\partial r_{ix}} \frac{\mathbf{R}_j \cdot (\mathbf{R}' \times \mathbf{R}'')}{|\mathbf{R}' \times \mathbf{R}''|} \\ &= \frac{1}{|\mathbf{R}' \times \mathbf{R}''|^2} \Bigg\{ |\mathbf{R}' \times \mathbf{R}''| \frac{\partial}{\partial r_{ix}} \left[\mathbf{R}_j \cdot (\mathbf{R}' \times \mathbf{R}'')\right] + \\ &\quad \left[\mathbf{R}_j \cdot (\mathbf{R}' \times \mathbf{R}'')\right] \frac{\partial}{\partial r_{ix}} |\mathbf{R}' \times \mathbf{R}''| \Bigg\}\end{aligned} \tag{9}$$

where the two partial derivatives with respect to r_{ix} are computed as follows (using the rule of derivative of scalar triple product of vectors),

$$\begin{aligned}\frac{\partial}{\partial r_{ix}} \left[\mathbf{R}_j \cdot (\mathbf{R}' \times \mathbf{R}'')\right] = &\frac{\partial \mathbf{R}_j}{\partial r_{ix}} \cdot (\mathbf{R}' \times \mathbf{R}'') + \\ &\mathbf{R}_j \cdot \left(\frac{\partial \mathbf{R}'}{\partial r_{ix}} \times \mathbf{R}''\right) + \\ &\mathbf{R}_j \cdot \left(\mathbf{R}' \times \frac{\partial \mathbf{R}''}{\partial r_{ix}}\right)\end{aligned} \tag{10}$$

$$\frac{\partial}{\partial r_{ix}} |\mathbf{R}' \times \mathbf{R}''| = \frac{\mathbf{R}' \times \mathbf{R}''}{|\mathbf{R}' \times \mathbf{R}''|} \cdot \left(\frac{\partial \mathbf{R}'}{\partial r_{ix}} \times \mathbf{R}'' + \mathbf{R}' \times \frac{\partial \mathbf{R}''}{\partial r_{ix}}\right). \tag{11}$$

The partial derivatives of $\mathbf{R}'$ and $\mathbf{R}''$ can be expanded into

$$\begin{cases} \dfrac{\partial \mathbf{R}'}{\partial r_{ix}} &= \displaystyle\sum_{j=1}^{N} \sin \frac{2\pi(j-1)}{N} \frac{\partial \mathbf{R}_j}{\partial r_{ix}} \\ \dfrac{\partial \mathbf{R}''}{\partial r_{ix}} &= \displaystyle\sum_{j=1}^{N} \cos \frac{2\pi(j-1)}{N} \frac{\partial \mathbf{R}_j}{\partial r_{ix}} \end{cases} \tag{12}$$

where only $\partial \mathbf{R}_j / \partial r_{ix}$ remains unknown, and considering Eq. 1, we have

$$\begin{cases} \dfrac{\partial \mathbf{R}_j}{\partial r_{ix}} = \left(1 - \dfrac{1}{N}, 0, 0\right), & i = j \\ \dfrac{\partial \mathbf{R}_j}{\partial r_{ix}} = \left(-\dfrac{1}{N}, 0, 0\right), & i \neq j \end{cases}. \tag{13}$$

Eq. 13 indicates that we could also easily derive the derivatives of $\mathbf{R}_j$ with respect to the r_{iy} and r_{iz}. Our final goal is $\partial\theta / \partial r_{ix}$, which can be obtained by the derivative rules,

$$\begin{aligned}\frac{\partial \theta}{\partial r_{ix}} = &- \frac{1}{\sqrt{1 - \left(\frac{C}{\sqrt{2A^2+2B^2+C^2}}\right)^2}} \cdot \frac{1}{2A^2+2B^2+C^2} \cdot \\ &\left(\frac{\partial C}{\partial r_{ix}} \sqrt{2A^2+2B^2+C^2} - C \frac{\partial}{\partial r_{ix}} \sqrt{2A^2+2B^2+C^2}\right)\end{aligned}, \tag{14}$$

$$\frac{\partial}{\partial r_{ix}}\sqrt{2A^2+2B^2+C^2} = \frac{1}{\sqrt{2A^2+2B^2+C^2}}\cdot \left(2A\frac{\partial A}{\partial r_{ix}} + 2B\frac{\partial B}{\partial r_{ix}} + C\frac{\partial C}{\partial r_{ix}}\right) \tag{15}$$

The partial derivatives of A, B and C with respect to r_{ix} are obtained by the chain rule,

$$\begin{cases} \dfrac{\partial A}{\partial r_{ix}} &= \displaystyle\sum_{j=1}^{N}\frac{\partial A}{\partial z_j}\frac{\partial z_j}{\partial r_{ix}} = \sum_{j=1}^{N}\sin\left[\frac{2\pi}{N}2(j-1)\right]\frac{\partial z_j}{\partial r_{ix}} \\ \dfrac{\partial B}{\partial r_{ix}} &= \displaystyle\sum_{j=1}^{N}\frac{\partial B}{\partial z_j}\frac{\partial z_j}{\partial r_{ix}} = \sum_{j=1}^{N}\cos\left[\frac{2\pi}{N}2(j-1)\right]\frac{\partial z_j}{\partial r_{ix}} \\ \dfrac{\partial C}{\partial z_j} &= \displaystyle\sum_{j=1}^{N}\frac{\partial C}{\partial z_j}\frac{\partial z_j}{\partial r_{ix}} = \sum_{j=1}^{N}(-1)^{j-1}\frac{\partial z_j}{\partial r_{ix}} \end{cases} \tag{16}$$

where $\partial z_j/\partial r_{ix}$ has been derived from Eq. 9 to Eq. 13.

4.2 Implementation

If the TCL scripting implementation is used and the gradient is required, Colvars will try to find the "`calc_<scripted_name>_gradient`" function for calculating the gradient, where "`<scripted_name>`" refers to the value of the corresponding "`scriptedFunction`". In our example, we want the gradient of θ, and we have already defined the `calc_cptheta` procedure in `pucker.tcl` to compute the value of θ, then we need a procedure `calc_cptheta_gradient` to compute its gradient. This procedure is expected to return an array that has the same shape of `args`. In other words, if `args` is a 2D list, then `calc_cptheta_gradient` should also be a 2D list that has the same number of elements of `args`. The code below (listing 16) serves an example (see the attached `pucker.tcl` for the full source code listing).

Listing 16: pucker.tcl

```
proc calc_cptheta_gradient {args} {
  global pi
  set N [llength [lindex $args 0]]
  # args is a list of atom coordinates of length 3*N
  set R_j [calc_Rj {*}[lindex $args 0]]
  set z_j [calc_zj {*}$R_j]
  lassign [calc_ABC {*}$z_j] A B C
  lassign [calc_ABC_gradient {*}[lindex $args 0]] dA dB dC
  set tmp1 [expr 2.0 * ($A * $A + $B * $B) + $C * $C]
  set factor [expr -180.0 / $pi / sqrt(1.0 -\
   ($C * $C / $tmp1)) * (1.0 / $tmp1)]
  set tmp2 [expr sqrt($tmp1)]
```

```
13    set tmp3 [expr 1.0 / $tmp2]
14    set grad [list]
15    for {set i 0} {$i < $N} {incr i} {
16      lappend grad [expr $factor * ($tmp2 * [lindex $dC $i] - \
17        $C * $tmp3 * (2.0 * $A * [lindex $dA $i] + \
18          2.0 * $B * [lindex $dB $i] + $C * [lindex $dC $i]))]
19    }
20    # Expect to return an array with the same shape of args
21    return [list $grad]
22  }
```

If the C++ implementation is preferred for its better performance, we can try implementing the "`calc_gradients`" virtual function for the "`cptheta`" class. The updated declaration of the class is,

Listing 17: colvarcomp_pucker.h

```
1  class colvar::cptheta : public colvar::cvc
2  {
3  protected:
4    /// Atom group
5    cvm::atom_group  *atoms = nullptr;
6    cvm::real A;
7    cvm::real B;
8    cvm::real C;
9    std::vector<cvm::rvector> dA_dr;
10   std::vector<cvm::rvector> dB_dr;
11   std::vector<cvm::rvector> dC_dr;
12 public:
13   cptheta();
14   virtual int init(std::string const &conf);
15   virtual void calc_value();
16   virtual void calc_gradients();
17 };
```

Here we also add a few member variables to save the intermediate A, B and C to optimize the calculation, since Colvars always runs `calc_value` before `calc_gradients`, and A, B and C with their derivatives could be determined in `calc_value` at the same time when computing θ. For the same reason, `calc_value` and `calc_cpABC` are also modified to compute $\partial A/\partial r_j$, $\partial B/\partial r_j$ and $\partial C/\partial r_j$ at the same time and save the results to `dA_dr`, `dB_dr` and `dC_dr`. In the definition of `calc_gradients`, the gradient of each atom position is supposed to save to `(*atoms)[ia].grad` in the parsed atom group as follows (the full code listing can be found at Github),

Listing 18: colvarcomp_pucker.cpp

```
void colvar::cptheta::calc_gradients() {
  const cvm::real tmp1 = 2.0 * (A * A + B * B) + C * C;
  const cvm::real factor = -180.0 / M_PI /
    std::sqrt(1.0 - (C * C / tmp1)) * (1.0 / tmp1);
  const cvm::real tmp2 = std::sqrt(tmp1);
  const cvm::real tmp3 = 1.0 / tmp2;
  for (size_t ia = 0; ia < atoms->size(); ia++) {
    (*atoms)[ia].grad = factor * (dC_dr[ia] * tmp2 -
      C * tmp3 * (2.0 * A * dA_dr[ia] +
                  2.0 * B * dB_dr[ia] + C * dC_dr[ia]));
  }
}
```

4.3 Simulation Setup

We are going to run a targeted MD simulation along θ to generate an ${}^1C_4$ structure. The first step of simulation is to determine the value of θ of our initial structure, which can be done by running a "zero"-step equilibrium simulation (using `run 0` in the NAMD configuation file). In our example, we have determined that θ is 16.38°, and we want to gradually "pull" it to 180° in 50,000 simulation steps, given that we have already implemented the gradients of θ with respect to the atomic coordinates, we can add the following section in the Colvars configuration file,

Listing 19: Targeted MD along θ

```
harmonic {
  colvars theta
  centers 16.38
  targetCenters 180.0
  targetNumSteps 50000
  forceConstant 0.5
  outputEnergy on
  outputCenters on
}
```

We can then combine the Listing 19 and Listing 10 into a new file `pucker_cpp_tmd.colvars` and update the `colvarsConfig` field in the NAMD configuration as well (see `tmd_cpp.namd` in the tutorial files). Then we start a simulation for 100,000 steps, the results of which are shown in Figure 7.

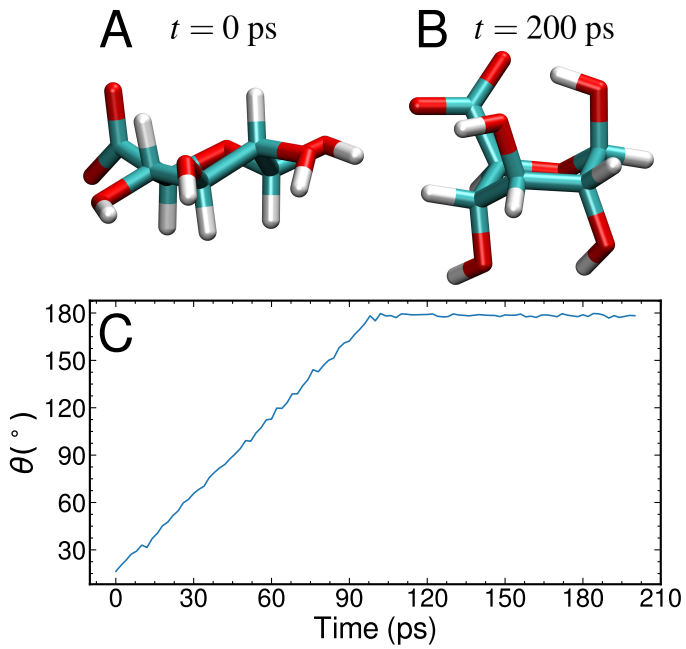


Figure 7: (A) Initial conformation of the glucuronic acid. (B) Resultant conformation of the glucuronic acid after 100-ps targeted MD simulation and 100-ps simulation with θ being restrained to 180°. (C) Time evolution of θ.

Acknowledgements

Development of this tutorial was supported by the National Institutes of Health (R24-GM145965 - Resource for Macromolecular Modeling and Visualization).

References

[1] B. Peters, "Reaction Coordinates and Mechanistic Hypothesis Tests," *Annu. Rev. Phys. Chem.*, vol. 67, pp. 669–690, May 2016.

[2] J. Rogal, "Reaction coordinates in complex systems-a perspective," *Eur. Phys. J. B*, vol. 94, p. 223, Nov. 2021.

[3] B. Isralewitz, M. Gao, and K. Schulten, "Steered molecular dynamics and mechanical functions of proteins," *Curr. Opin. Struct. Biol.*, vol. 11, pp. 224–230, Apr. 2001.

[4] A. Laio and M. Parrinello, "Escaping free-energy minima," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 99, no. 20, pp. 12562–12566, 2002.

[5] G. M. Torrie and J. P. Valleau, "Nonphysical sampling distributions in Monte Carlo free-energy estimation: Umbrella sampling," *J. Chem. Phys.*, vol. 23, no. 2, pp. 187–199, 1977.

[6] J. C. Phillips, D. J. Hardy, J. D. C. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. Hénin, W. Jiang, R. McGreevy, M. C. R. Melo, B. K. Radak, R. D. Skeel, A. Singharoy, Y. Wang, B. Roux, A. Aksimentiev, Z. Luthey-Schulten, L. V. Kalé, K. Schulten, C. Chipot, and E. Tajkhorshid, "Scalable molecular dynamics on CPU and GPU architectures with NAMD," *J. Chem. Phys.*, vol. 153, p. 044130, July 2020.

[7] G. Fiorin, M. L. Klein, and J. Hénin, "Using collective variables to drive molecular dynamics simulations," *Mol. Phys.*, vol. 111, pp. 3345–3362, Dec. 2013.

[8] D. Branduardi, F. L. Gervasio, and M. Parrinello, "From A to B in free energy space," *J. Chem. Phys.*, vol. 126, p. 054103, Feb. 2007.

[9] D. Cremer and J. A. Pople, "General definition of ring puckering coordinates," *J. Am. Chem. Soc.*, vol. 97, pp. 1354–1358, Mar. 1975.

[10] M. Bonomi, D. Branduardi, G. Bussi, C. Camilloni, D. Provasi, P. Raiteri, D. Donadio, F. Marinelli, F. Pietrucci, R. A. Broglia, and M. Parrinello, "PLUMED: A portable plugin for free-energy calculations with molecular dynamics," *Comput. Phys. Commun.*, vol. 180, pp. 1961–1972, Oct. 2009.