

How to differentiate collective variables in free energy codes

10.1016/j.cpc.2018.02.017



Toni Giorgino

National Research Council of Italy

www.giorginolab.it



@giorginolab



[giorginolab / PlumedAutoDiff-Lugano-2019](#)

plumID

19.011

Open source software for enhanced-sampling simulations
Lugano, 26th of July 2019

Generalized forces in biased sampling

$$f(\mathbf{x}) = f(\mathbf{x}_1, \dots, \mathbf{x}_n)$$

A collective variable (CV)

$$V(\mathbf{x}) = V_1(f(\mathbf{x}))$$

Bias potential depending
on \mathbf{x} only through CV

$$\underline{\mathbf{F}(\mathbf{x}) = -\nabla_{\mathbf{x}} V_1(f(\mathbf{x}))} = -\frac{\partial V_1(f)}{\partial f} \nabla_{\mathbf{x}} f(\mathbf{x})$$

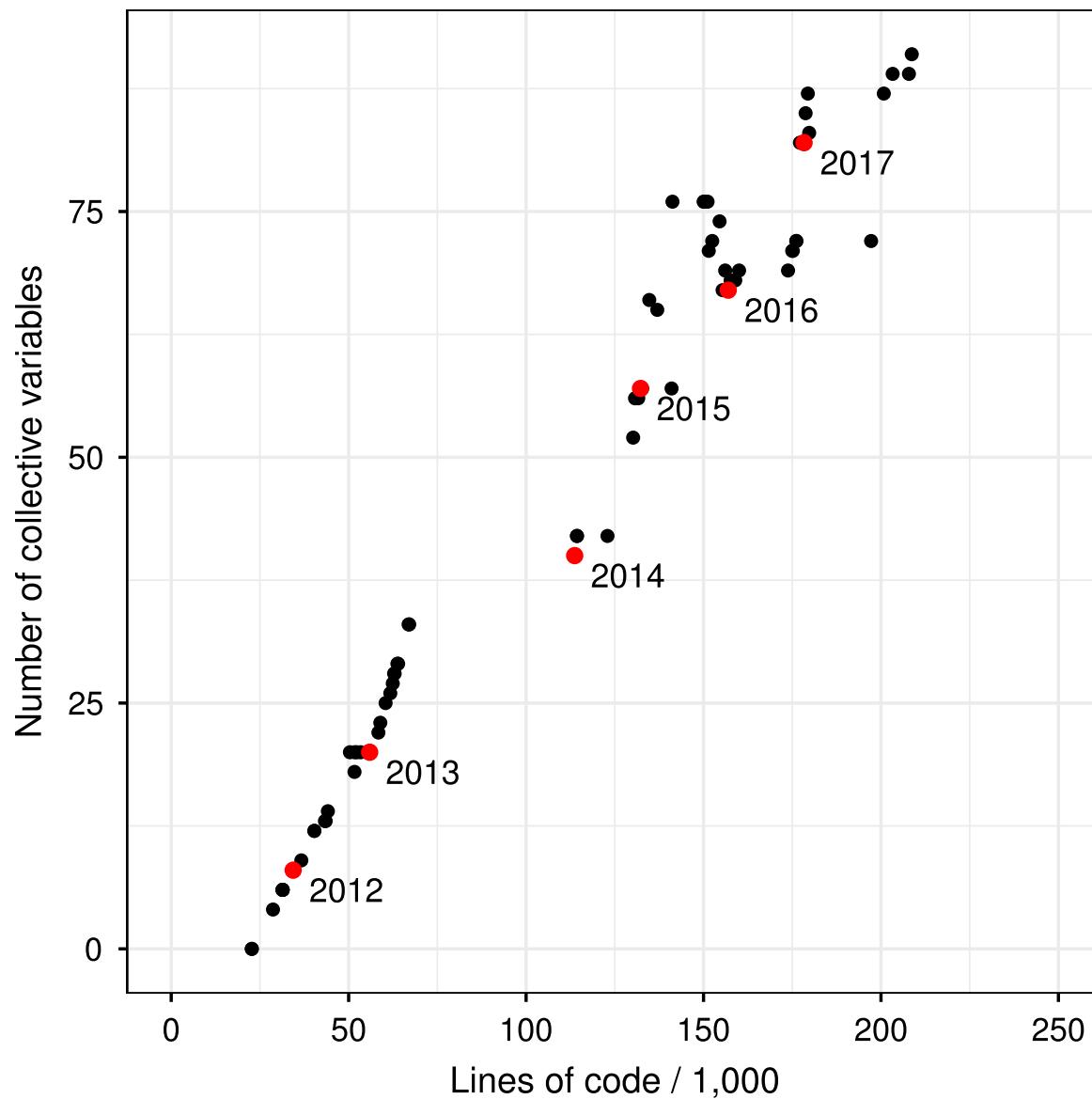
Bias force

Generalized
force

CV
gradient



CV count & codebase growth





Computer Physics Communications 228 (2018) 258–263

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

How to differentiate collective variables in free energy codes: Computer-algebra code generation and automatic differentiation[☆]

Toni Giorgino¹

Institute of Neurosciences, National Research Council (CNR-IN), Corso Stati Uniti 4, I-35127, Padua, Italy



plumID

19.011

ARTICLE INFO

Article history:

Received 2 September 2017

Received in revised form 12 February 2018

Accepted 18 February 2018

Available online 26 February 2018

Keywords:

Molecular dynamics

Free energy

Biased sampling

Metadynamics

Symbolic

C++

ABSTRACT

The proper choice of collective variables (CVs) is central to biased-sampling free energy reconstruction methods in molecular dynamics simulations. The PLUMED 2 library, for instance, provides several sophisticated CV choices, implemented in a C++ framework; however, developing new CVs is still time consuming due to the need to provide code for the analytical derivatives of all functions with respect to atomic coordinates. We present two solutions to this problem, namely (a) symbolic differentiation and code generation, and (b) automatic code differentiation, in both cases leveraging open-source libraries (SymPy and Stan Math, respectively). The two approaches are demonstrated and discussed in detail implementing a realistic example CV, the local radius of curvature of a polymer. Users may use the code as a template to streamline the implementation of their own CVs using high-level constructs and automatic gradient computation.

Program summary

Program Title: Practical approaches to the differentiation of collective variables in free energy codes:

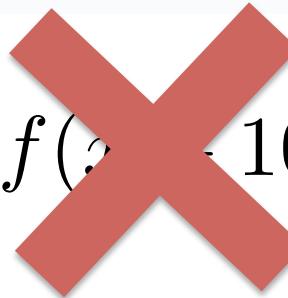
tonigi / plumed2-automatic-gradients
 giorginolab / PlumedAutoDiff-Lugano-2019

← Code for this talk



1

Numerical differentiation NUMERICAL_DERIVATIVES

$$f(x) - f(x + 10^{-8})$$


2

Symbolic + code generation

- src/curvature_codegen
- CURVATURE_CODEGEN



3

Automatic differentiation, i.e. propagation of derivatives through operands (in code)

- src/curvature_autodiff
- CURVATURE_AUTODIFF



I. Numerical differentiation: *don't*

$$f'(x) \asymp [f(x + \Delta) - f(x)] / \Delta$$

e.g.: $f(x) = 10^9 + x$

- Small differences between large floating-point numbers get lost in finite precision:

$$10^9 + \varepsilon - 10^9 \asymp 0$$

- Inefficient (repeat \forall coordinates)
- Can differentiate arbitrary functions

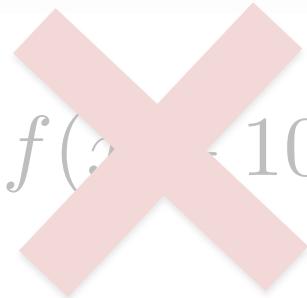
Already in Plumed, **NUMERICAL_DERIVATIVES**



1

Numerical differentiation
NUMERICAL_DERIVATIVES

$$f(x) - f(x + 10^{-8})$$



2

Symbolic + code generation

- src/curvature_codegen
- CURVATURE_CODEGEN



3

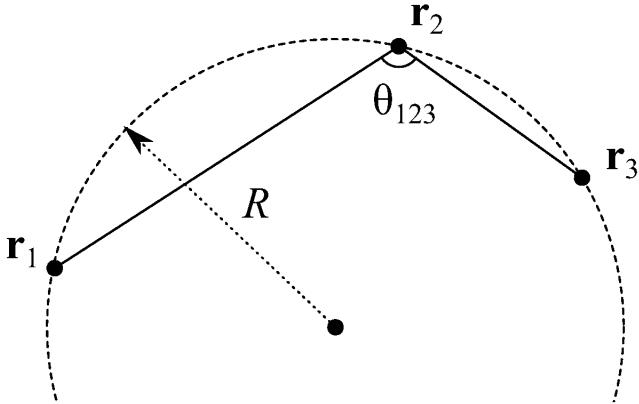
Automatic differentiation,
i.e. propagation of derivatives
through operands (in code)

- src/curvature_autodiff
- CURVATURE_AUTODIFF



2. Symbolic differentiation: *sometimes*

1. Symbolic CAS software* to differentiate...
2. ...then auto-generate C code



$$R : \mathbb{R}^9 \rightarrow \mathbb{R}^+$$

$$2R = \frac{|\mathbf{r}_{13}|}{\sin \theta_{123}} \quad \cos \theta_{123} = \frac{\mathbf{r}_{12} \cdot \mathbf{r}_{23}}{|\mathbf{r}_{12}| |\mathbf{r}_{23}|}$$

```
In [4]: side2 = r_13.dot(r_13)
cos2a = r_12.dot(r_23)**2 /
         (r_12.dot(r_12) * r_23.dot(r_23))
sin2a = 1-cos2a

R2 = side2/sin2a/4
R = sqrt(R2)
R
```

$$\frac{1}{2} \sqrt{\frac{(r_{1x} - r_{3x})^2 + (r_{1y} - r_{3y})^2 + (r_{1z} - r_{3z})^2}{-\frac{((r_{1x}-r_{2x})(r_{2x}-r_{3x})+(r_{1y}-r_{2y})(r_{2y}-r_{3y})+(r_{1z}-r_{2z})(r_{2z}-r_{3z}))^2}{((r_{1x}-r_{2x})^2+(r_{1y}-r_{2y})^2+(r_{1z}-r_{2z})^2)((r_{2x}-r_{3x})^2+(r_{2y}-r_{3y})^2+(r_{2z}-r_{3z})^2)}+1}}$$

* SymPy, also online

src/curvature_codegen

CURVATURE_CODEGEN

plumed2-automatic-gradients / src / curvature_codegen / sympy_codegen / CurvatureCodegen.ipynb



```
In [53]: from sympy.tensor.array import derive_by_array
mgrad_1=derive_by_array(R,r_1).tomatrix()
mgrad_2=derive_by_array(R,r_2).tomatrix()
mgrad_3=derive_by_array(R,r_3).tomatrix()

# Use e.g. mgrad_1 to print the expressions. They are several lines long.
```



Code generation is performed with the `codegen()` function, which will write `R` and the components of `mgrad_*` in the files `curvature_codegen.*`.

$\frac{\partial}{\partial}$

```
In [6]: from sympy.utilities.codegen import codegen
codegen([ ("curvature_radius",R),
          ("curvature_radius_grad", [
              Eq(MatrixSymbol("g1",3,1),mgrad_1),
              Eq(MatrixSymbol("g2",3,1),mgrad_2),
              Eq(MatrixSymbol("g3",3,1),mgrad_3) ]),
          to_files=True,
          prefix="curvature_codegen",
          project="plumed_curvature",
          language="C" )
```

$d_{12}^2 d_{23}^2 - ((r$

...
...

This concludes the code generation necessary for implementation. Code is now in `curvature_codegen.[ch]`. Include it and call the `curvature_radius` and `curvature_radius_grad` functions from the collective variable C++ files. See the code in `src/curvature-codegen/Curvature.cpp` and the other source files in the same directory.

codegen

Note 1. The generated code is somewhat redundant and therefore relies on the *common subexpression elimination* optimization pass of the compiler. The later section [Common Subexpression Elimination](#) demonstrates how to leverage SymPy's high-level CSE instead.

Note 2. Here we took an hands-off approach and differentiate with respect to the coordinates of each atom. Differentiating with respect to pairwise distances is also an option - see the section [Gradients with respect to distances](#) at the end of the notebook.

Common subexpression elimination

...
...

Code generation

Naïve: ~2,000 f. p. ops

Symbolic CSE: ~150 f. p. ops

```
void curvature_radius_grad(double r_1x, double r_1y, double r_1z,
    double r_2x, double r_2y, double r_2z,
    double r_3x, double r_3y, double r_3z,
    double *g1, double *g2, double *g3) {
    double p0 = -r_3x;
    double p1 = p0 + r_1x;
    double p2 = -r_3y;
    double p3 = p2 + r_1y;
    double p4 = -r_3z;
    double p5 = p4 + r_1z;
    double p6 = pow(p1, 2) + pow(p3, 2) + pow(p5, 2);
    double p7 = r_1x - r_2x;
    double p8 = r_1y - r_2y;
    double p9 = r_1z - r_2z;
    double p10 = 1.0/(pow(p7, 2) + pow(p8, 2) + pow(p9, 2));
    double p11 = p0 + r_2x;
    double p12 = p2 + r_2y;
    double p13 = p4 + r_2z;
    double p14 = 1.0/(pow(p11, 2) + pow(p12, 2) + pow(p13, 2));
    double p15 = p11*p7 + p12*p8 + p13*p9;
    double p16 = 1.0/(p10*p14*pow(p15, 2) - 1);
    double p17 = sqrt(-p16*p6);
    double p18 = (1.0/2.0)*p17*p6;
    double p19 = p10*p15;
    double p20 = p19*p7;
    double p21 = p10*p14*p15*p16*p6;
    double p22 = p19*p8;
```

Common Subexpression Elimination

But: can not differentiate arbitrary functions

$$2)*(\text{pow}(\text{r}_2\text{x} - \text{r}_3\text{x}, 2) + \text{pow}(\text{r}_2\text{y} - \text{r}_3\text{y}, 2) + \text{pow}(\text{r}_2\text{z} - \text{r}_3\text{z}, 2)) + (2*\text{r}_{22} - 2*\text{r}_{32})*(\text{r}_{1x} - \text{r}_{2x})*(\text{r}_{2x} - \text{r}_{3x}) + (\text{r}_{1y} - \text{r}_{2y})*(\text{r}_{2y} - \text{r}_{3y}) + (\text{r}_{1z} - \text{r}_{2z})*(\text{r}_{2z} - \text{r}_{3z})]/((\text{pow}(\text{r}_{1x} - \text{r}_{2x}, 2) + \text{pow}(\text{r}_{1y} - \text{r}_{2y}, 2) + \text{pow}(\text{r}_{1z} - \text{r}_{2z}, 2), 2)*(\text{pow}(\text{r}_{2x} - \text{r}_{3x}, 2) + \text{pow}(\text{r}_{2y} - \text{r}_{3y}, 2) + \text{pow}(\text{r}_{2z} - \text{r}_{3z}, 2)))*(\text{pow}(\text{r}_{1x} - \text{r}_{3x}, 2) + \text{pow}(\text{r}_{1y} - \text{r}_{3y}, 2) + \text{pow}(\text{r}_{1z} - \text{r}_{3z}, 2))/(\text{pow}(\text{r}_{1x} - \text{r}_{2x} - \text{r}_{3x}), (\text{r}_{1y} - \text{r}_{2y})*(\text{r}_{2y} - \text{r}_{3y}) + (\text{r}_{1z} - \text{r}_{2z} - \text{r}_{3z})*(\text{r}_{2z} - \text{r}_{3z}), 2)/((\text{pow}(\text{r}_{1x} - \text{r}_{2x}, 2) + \text{pow}(\text{r}_{1y} - \text{r}_{2y}, 2) + \text{pow}(\text{r}_{1z} - \text{r}_{2z}, 2))*(\text{pow}(\text{r}_{2x} - \text{r}_{3x}, 2) + \text{pow}(\text{r}_{2y} - \text{r}_{3y}, 2) + \text{pow}(\text{r}_{2z} - \text{r}_{3z}, 2)) + 1, 2)*(\text{pow}(\text{r}_{1x} - \text{r}_{3x}) + (\text{r}_{1y} - \text{r}_{2y})*(\text{r}_{2y} - \text{r}_{3y}) + (\text{r}_{1z} - \text{r}_{2z} - \text{r}_{3z})*(\text{r}_{2z} - \text{r}_{3z}), 2)/(\text{pow}(\text{r}_{1x} - \text{r}_{2x}, 2) + \text{pow}(\text{r}_{1y} - \text{r}_{2y}, 2) + \text{pow}(\text{r}_{1z} - \text{r}_{2z}, 2))*(\text{pow}(\text{r}_{2x} - \text{r}_{3y}, 2) + \text{pow}(\text{r}_{2z} - \text{r}_{3z}, 2)) + 1)/(\text{pow}(\text{r}_{1x} - \text{r}_{3y}, 2) + \text{pow}(\text{r}_{1z} - \text{r}_{3z}, 2));$$

x 3

```

g1[1] = p18*(-p24*(p12 - p22) + p3);
g1[2] = p18*(-p21*(p13 - p23) + p5);
g2[0] = -p26*(p20 + p25 + r_1x - 2*r_2x + r_3x);
g2[1] = -p26*(p22 + p27 + r_1y - 2*r_2y + r_3y);
g2[2] = -p26*(p23 + p28 + r_1z - 2*r_2z + r_3z);
g3[0] = -p18*(p1 - p21*(p25 + p7));
g3[1] = -p18*(-p21*(p27 + p8) + p3);
g3[2] = -p18*(-p21*(p28 + p9) + p5);
}

```

Algorithms ⊢ functions

- Differentiate this...

```
def pow(x,n):  
    y=1.0  
    for i in range(n):  
        y *= x  
    return y
```

Algorithms \supset functions

- Implicit solvers \longrightarrow Root finding
- ODEs \longrightarrow Sensitivity analysis
 - Study $y(t)$ as a function of parameters and y_0
- Optimization \longrightarrow Neural network training

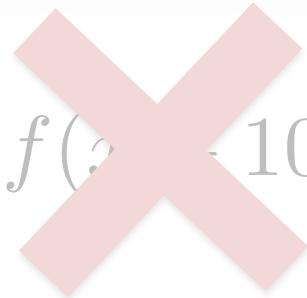
... and generalized forces in biased sampling



1

Numerical differentiation
NUMERICAL_DERIVATIVES

$$f(x) - f(x + 10^{-8})$$



2

Symbolic + code generation

- src/curvature_codegen
- CURVATURE_CODEGEN



3

Automatic differentiation,
i.e. propagation of derivatives
through operands (in code)

- src/curvature_autodiff
- CURVATURE_AUTODIFF



Automatic reverse-mode differentiation

(Seppo Linnainmaa, 1970)

A generic code does...

Hello, I am a program!

$$y(\underline{w_n}(w_{n-1}(\cdots w_1(x))))$$

- Let's ∇ *forward*... (Deepest parenthesis executed first)

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_n} \frac{\partial w_n}{\partial x} = \frac{\partial y}{\partial w_n} \left(\frac{\partial w_n}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} \right) = \dots$$

Forward mode differentiation

- Idea: much like “error propagation”
- Easy with OOP: operator overload

$$X \doteq (x, d)$$

$$X^2 \doteq (x, d) \rightarrow (x^2, 2xd)$$

$$X \times Y \doteq (x, d), (y, e) \rightarrow (xy, xe + yd)$$

...etc

- Set one “error” to 1 (gradient direction),
the others to 0, compute, *repeat. Inefficient.*

An intuitive idea...

$$f(X(x_1, x_2, \dots, x_n), Y(x_1, x_2, \dots, x_n))$$

$$\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial X} \frac{\partial X}{\partial x_1} + \frac{\partial f}{\partial Y} \frac{\partial Y}{\partial x_1}$$

...many coordinates...

⋮

⋮

$$\frac{\partial f}{\partial x_n} = \frac{\partial f}{\partial X} \frac{\partial X}{\partial x_n} + \frac{\partial f}{\partial Y} \frac{\partial Y}{\partial x_n}$$

Hello, I am a
program!

$$\underline{y(w_n(w_{n-1}(\cdots w_1(x))))}$$

- Rather than ∇ **forward**... (Deepest parenthesis executed first)

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_n} \frac{\partial w_n}{\partial x} = \frac{\partial y}{\partial w_n} \left(\frac{\partial w_n}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} \right) = \dots$$

- Do it **reverse mode**

$$= \frac{\partial y}{\partial w_1} \frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = ((\) \) \frac{\partial w_1}{\partial x} = \dots$$

Reverse mode* implementation

- When accessing an operand, put it in a stack with its partial derivatives
 - All ∂ 's are numbers
 - C++ by OOP operator overload
- Initialize one *adjoint* per step encountered
- In the second pass, pop the stack
 - Increase the adjoints by result $\times \partial$
- At end: whole gradient at once
 - Trades off memory for speed

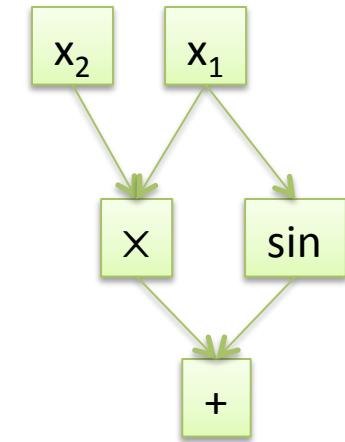
Reverse $f(x_1, x_2) = x_1x_2 + \sin(x_1)$

Pass 1
(in execution order)

$v_1 = x_1$	$- (!)$
$v_2 = x_2$	$- (!)$
$v_3 = v_1v_2$	$\partial v_3 / \partial v_1 = v_2; \quad \partial v_3 / \partial v_2 = v_1$
$v_4 = \sin v_1$	$\partial v_4 / \partial v_1 = \cos v_1$
$v_5 = \underline{v_3} + \underline{v_4}$	$\partial v_5 / \partial v_3 = +1; \quad \partial v_5 / \partial v_4 = +1$

Pass 2
(in reference order)

$a_5 \leftarrow 1; \quad a_1 = \dots = a_4 \leftarrow 0$	
$a_3 \leftarrow a_3 + \underline{a_5 \cdot \partial v_5 / \partial v_3}$	$(= a_5)$
$a_4 \leftarrow a_4 + \underline{a_5 \cdot \partial v_5 / \partial v_4}$	$(= a_5)$
$a_1 \leftarrow a_1 + \underline{a_4 \cdot \cos v_1}$	
$a_1 \leftarrow a_1 + a_3 \cdot v_2$	
$a_2 \leftarrow a_2 + a_3 \cdot v_1$	



**In practice,
for PLUMED...**



stan-dev / math

arXiv:1509.07164

- Platform for MC-based inference and estimation.
 - Has its own modeling language + interfaces.
- }



```
stan::math::gradient(f, x, f_x, grad_f_x);
```





- A **high-level C++ library** of template headers for linear algebra, matrix and vector operations, geometrical transformations, numerical solvers and related algorithms. [MPL]
- Stan provides high-level functions which **assume Eigen types** in- and out- types

```
Eigen::Matrix<T, #rows, #cols> m = ...
```

#rows and **#cols** integer or Eigen::Dynamic

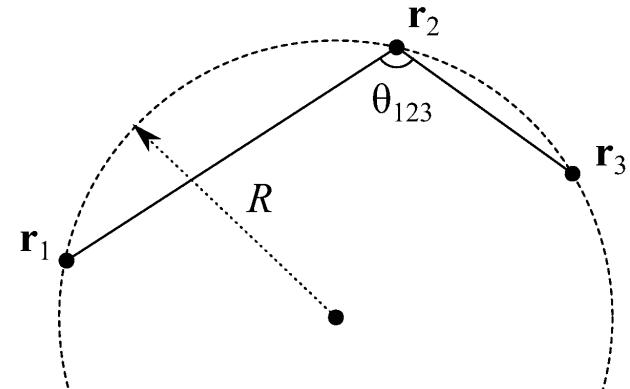
```

1 struct curvature_fun {
2 private:
3     bool inverse;           // List of parameters
4 public:
5     curvature_fun(bool inverse): inverse(inverse) {}
6
7     template <typename T>
8     T operator()(const Matrix<T, Dynamic, 1>& x)
9     const {
10         // Split into 3D vectors for convenience
11         Matrix<T, 3, 1> r1, r2, r3;
12         r1 = x.segment(0,3);
13         r2 = x.segment(3,3);
14         r3 = x.segment(6,3);
15
16         Matrix<T, 3, 1> r12, r32, r13;
17         r12 = r1-r2;           // Triangle sides
18         r32 = r3-r2;
19         r13 = r1-r3;
20
21         T cos2_a = pow(r12.dot(r32),2.0)
22             / r12.dot(r12) / r32.dot(r32);
23         T sin2_a = 1.0 - cos2_a;
24
25         T radius2 = r13.dot(r13) / sin2_a / 4.0;
26         T radius = sqrt(radius2);    // Eq. (1)
27
28         if(inverse)
29             radius = 1.0/radius;
30
31         return(radius);
32     }
33 };

```



Templated $\langle T \rangle$ function (functor).
In: Eigen::Matrix<T, *, 1>
Out: scalar of type T
Params: arbitrary (here bool)



$$\cos^2 \theta = \frac{|\mathbf{r}_{12} \cdot \mathbf{r}_{23}|^2}{|\mathbf{r}_{12}|^2 |\mathbf{r}_{23}|^2}$$

$$4R^2 = |\mathbf{r}_{13}|^2 / \sin^2 \theta$$



```
// Prepare the four arguments.  
  
// 1. The function to differentiate, declared as above.  
curvature_fun f(inverse);  
  
// 2. The point at which the function is to be computed, given as a  
// 1D Eigen Matrix.  
Eigen::Matrix<double,3*natoms,1> x;  
getAtomPositionsAsEigenMatrix(x);  
  
// 3. The variable to receive the return value of the function.  
double f_x;  
  
// 4. The variable to receive the gradient, as an Eigen Matrix.  
Eigen::Matrix<double,Eigen::Dynamic,1> grad_f_x;  
  
// 5. The evaluation happens here. The derivative formula have been  
// generated at compile time!  
stan::math::gradient(f, x, f_x, grad_f_x);
```



CURVATURE_CODEGEN**CURVATURE_MULTICOLVAR_CODEGEN**

Home (v2.5)

Getting Started

Tutorials

Index of Actions

Search

- ▶ CV Documentation
- ▶ Distances from reference configurations
- ▶ Functions
- ▼ MultiColvar
 - MultiColvar functions
 - MultiColvar bias
 - Extracting all the base quantities

ANGLES

BOND_DIRECTIONS

BRIDGE

COORDINATIONNUMBER

CURVATURE_MULTICOLVAR_CODEGEN

DENSITY

DISTANCES

FCCUBIC

HBPAMM_SH

INPLANEDISTANCES

MOLECULES

PLANES

Q3

Q4

Q6

MOMENTS

calculate the moments of the distribution of collective variables. The first moment of a distribution is calculated using $\frac{1}{N} \sum_{i=1}^N (s_i - \bar{s})^m$, where \bar{s} is the average for the distribution. The moments keyword takes a lists of integers as input or a range. Each integer is a value of m . The final calculated values can be referenced using moment- m . You can use the COMPONENT keyword in this action but the syntax is slightly different. If you would like the second and third moments of the third component you would use MOMENTS={COMPONENT=3 MOMENTS=2-3}. The moments would then be referred to using the labels moment-3-2 and moment-3-3. This syntax is also required if you are using numbered MOMENT keywords i.e. MOMENTS1, MOMENTS2...

Examples

The following input tells plumed to calculate the curvature radiiuses of the three circles passing through atoms 1,2 and 3; 2, 3 and 4; and 3, 4 and 5. The minimum of the three curvatures is used as a colvar and printed.

```
CURVATURE_MULTICOLVAR_CODEGEN ATOMS1=1,2,3 ATOMS2=2,3,4 ATOMS3=3,4,5 MIN={BETA=0.1} LABEL=d1
```

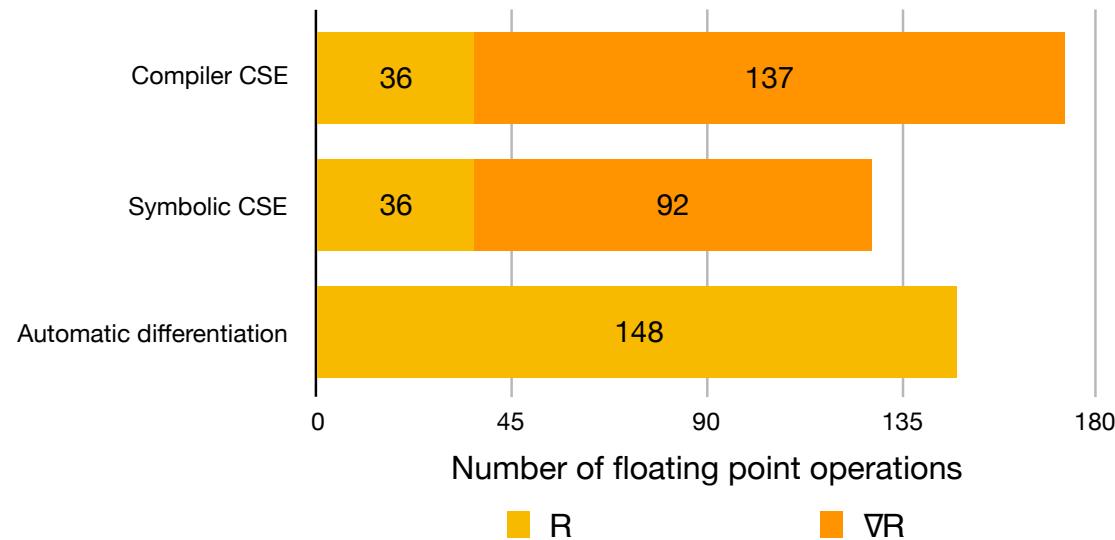
The following input uses the POLYMER keyword and it is equivalent to the previous one.

```
CURVATURE_MULTICOLVAR_CODEGEN POLYMER=1-5 MIN={BETA=0.1} LABEL=d1
```

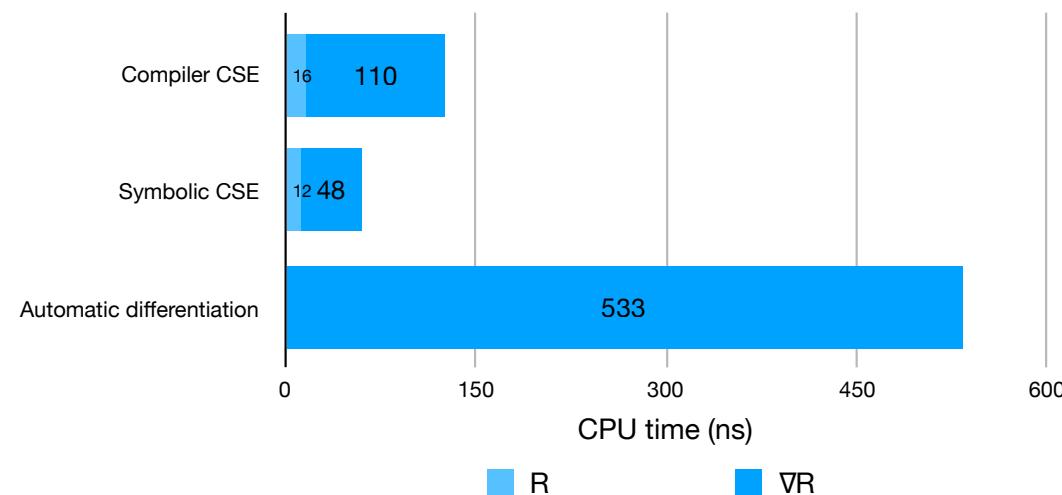
Count the number of beads in a polymer having a local radius of curvature less than 3.5:

```
d1: CURVATURE_MULTICOLVAR_CODEGEN POLYMER=1-100 LESS_THAN={RATIONAL R_0=3.5} LOWMEM
PRINT ARG=d1.lessthan
```

Performance



Measured through
hardware FP
counters,
**FP_COMP_OPS_EXE:
SSE_SCALAR_DOUB
LE &c**



Quiz

How do gradients look like?

VMD File Edit Templates

Structure Help

- Build reference structure...
- Insert native contacts CV...
- Insert backbone torsion $\phi/\psi/\omega$ CVs...
- Insert group for secondary structure RMSD...
- Display gradients and forces...

← VMD-Plumed Plugin
(under Extensions)

Plumed-GUI collective variable analysis tool
unnamed.plumed

Enter collective variable definitions below, in your engine's syntax.
Click 'Plot' to evaluate them on the 'top' trajectory.
VMD atom selections in square brackets expand automatically. For example:

```
protein: COM ATOMS=[chain A and name CA]
ligand: COM ATOMS=[chain B and noh]

DISTANCE ATOMS=protein,ligand
```

Default UNITS are nm, ps and kJ/mol unless changed.
Right mouse button provides help on keywords.

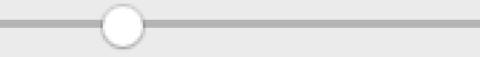
UNITS LENGTH=A ENERGY=kcal/mol TIME=ps

d1: DISTANCE ATOMS=1,200 # Just an example

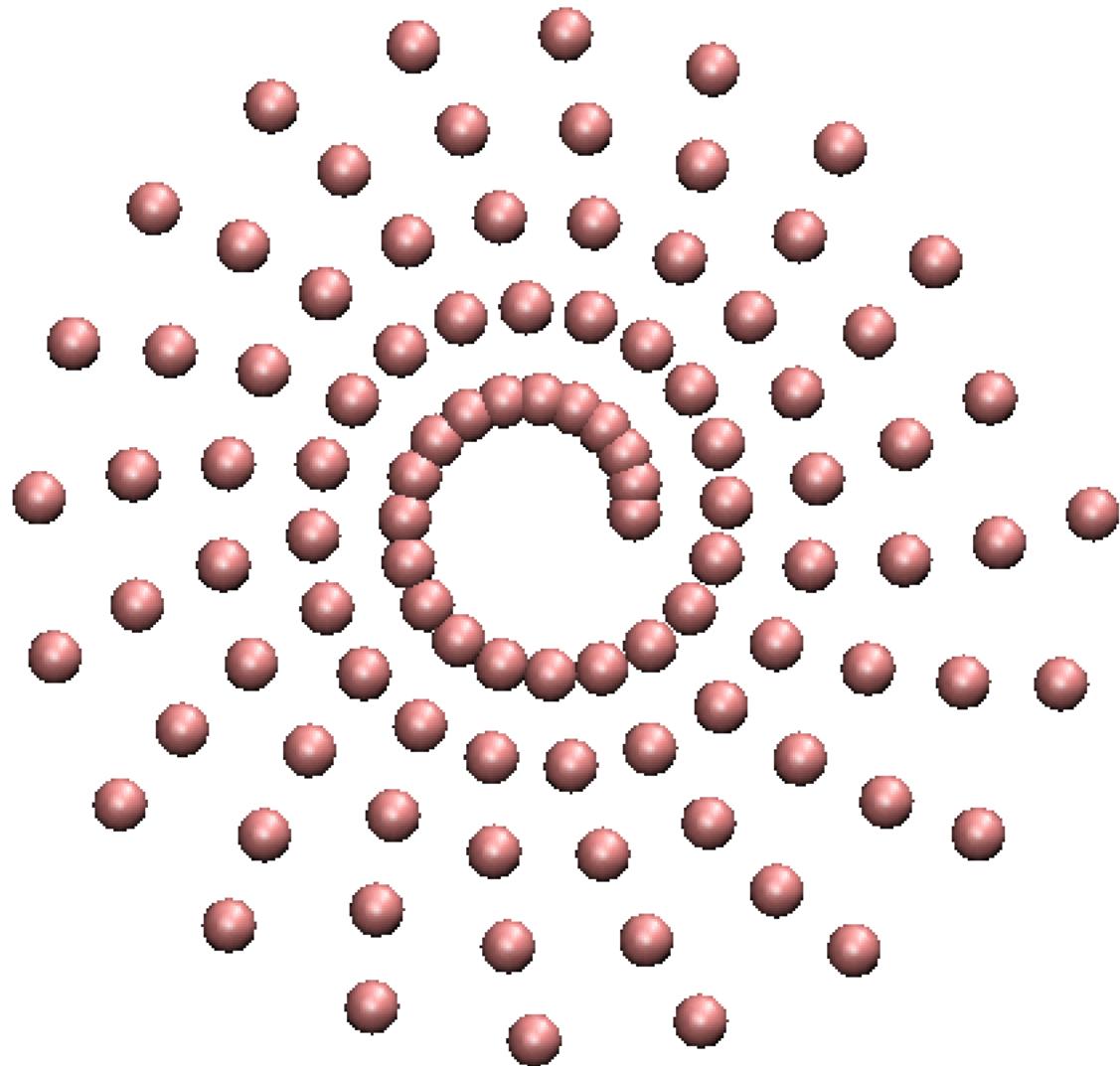
Display gradients and forces

Display the force vector that would be applied to each atom.
To visualize the effect of a bias on a CV
you may want to apply a constant unitary force to it, e.g.:

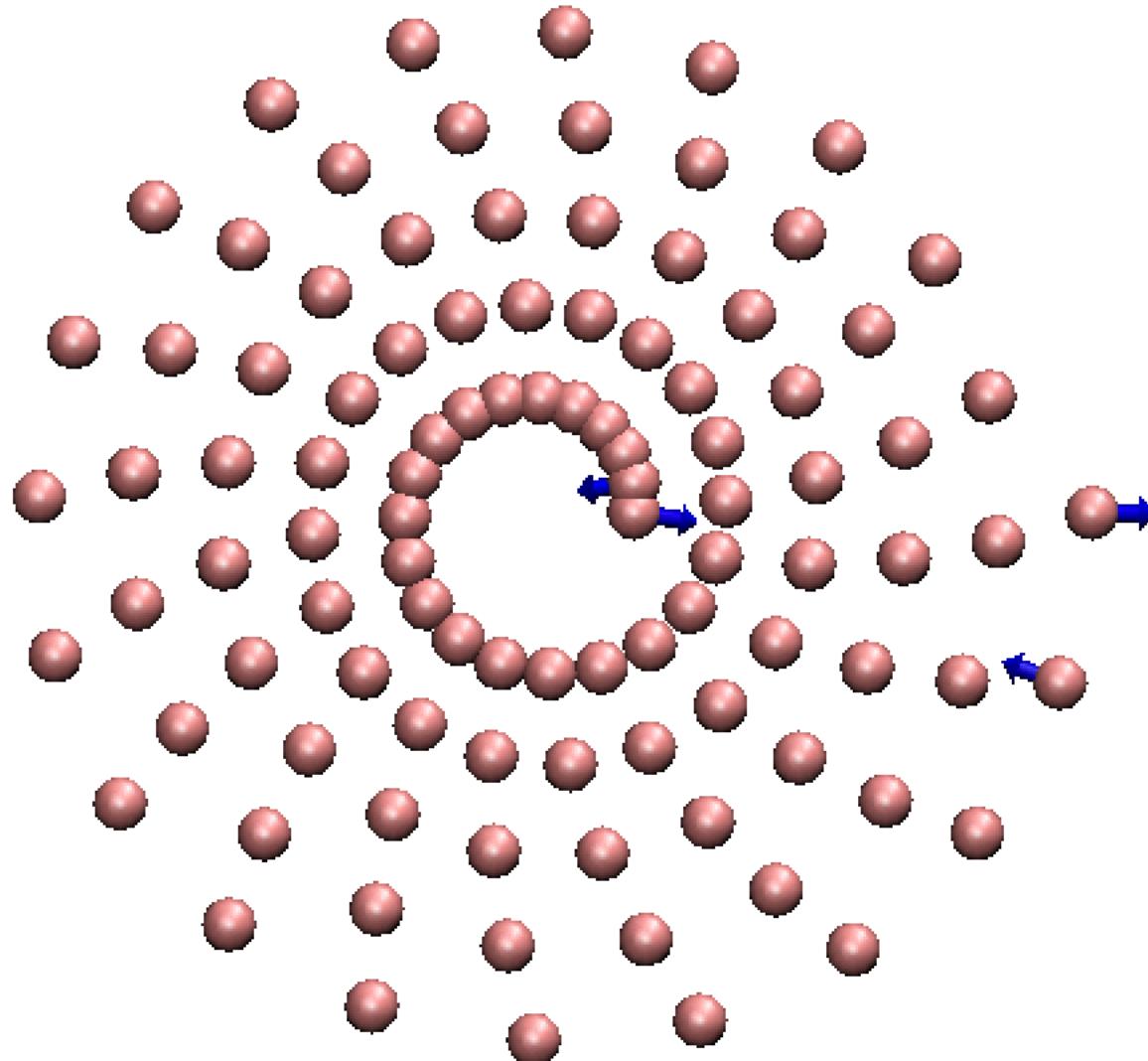
```
RESTRAINT ARG=mycv AT=0 SLOPE=-1
```

Arrow scale:  0.12 Å per kJ/mol/nm

Gradient of the *mean* radius of curvature?



r: CURVATURE_MULTICOLVAR_CODEGEN POLYMER=1-100 MEAN
RESTRAINT AT=0 SLOPE=-1 ARG=r.mean



In summary

Approach	Stability	Generality	High D?	DIY°
Numeric	-	+	-	+
Symbolic	+	-	+	-
Auto forward	+	+	-	+
Auto reverse	+	+	+	-

- Differentiation of code is a solved problem
- Don't do finite differences
- Consider code-gen for static expressions
- Reverse mode for ∇ of many variables
 - Many existing libraries, e.g. Stan Math*

° Yet no real need to

* Also provides linear algebra, ODEs, special functions, etc. etc.

Acknowledgements

The workshop organizers and CECAM

Prof. G. Bussi, SISSA

Prof. C. Camilloni, Uni Milano

Prof. De Fabritiis & group,
Universitat Pompeu Fabra, Barcelona

Acellera Ltd. (Funding)

Univ. of Insubria + AIRC (Funding)

