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

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

A collective variable (CV)

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

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

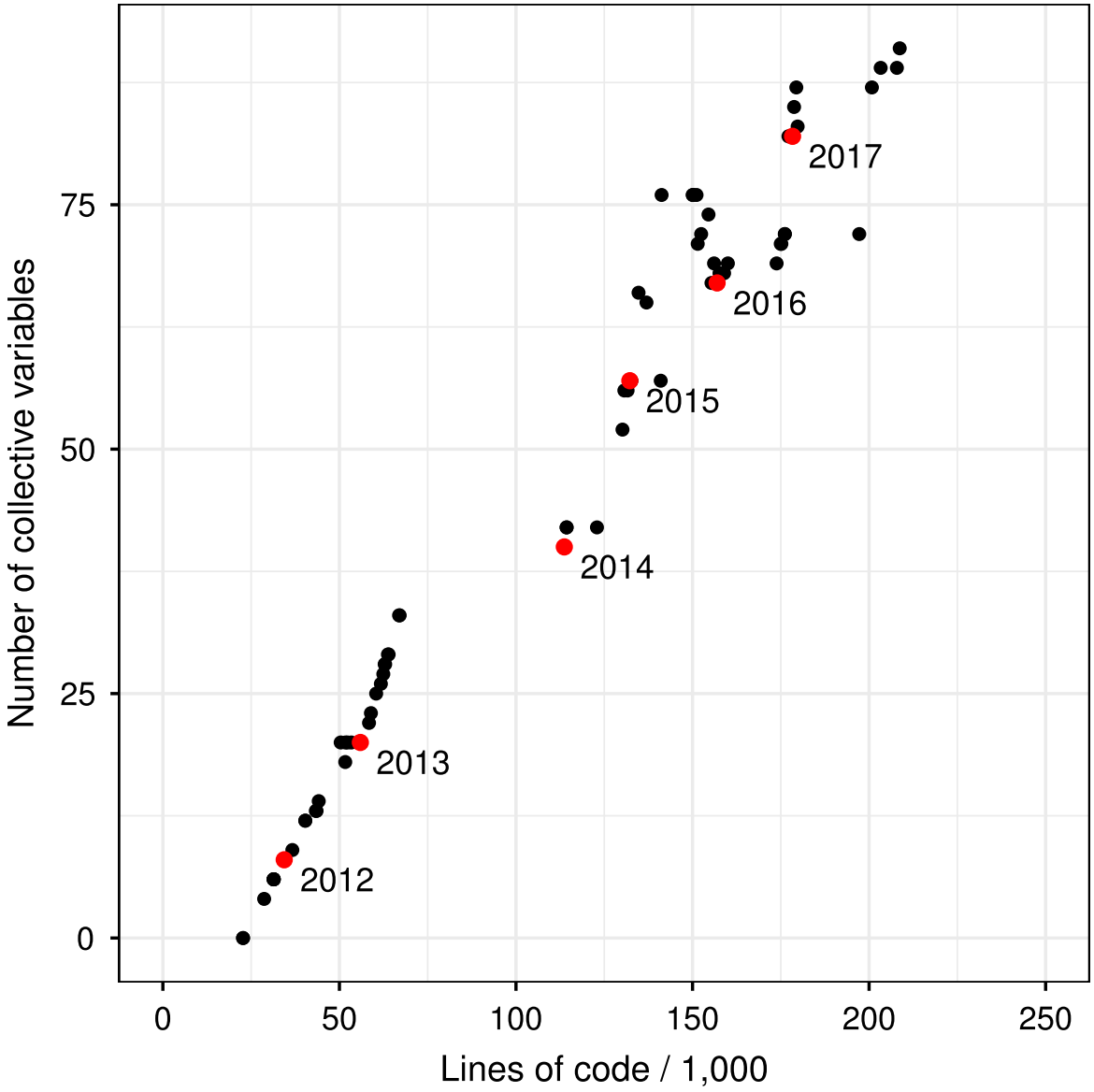
$$\mathbf{F}(\mathbf{x}) = -\nabla_{\mathbf{x}} V_1(f(\mathbf{x})) = -\frac{\partial V_1(f)}{\partial f} \nabla_{\mathbf{x}} f(\mathbf{x})$$

Bias force

Generalized force

CV gradient

CV count & codebase growth



10.1016/j.cpc.2018.02.017 arXiv:1709.06780

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 Giorgino [1]

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

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

tonigi / **plumed2-automatic-gradients**

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`

1. Numerical differentiation: *don't*

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

$$\text{e.g.:}\ \ f(x) = 10^9 + x$$

- 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**

tonigi / **plumed2-automatic-gradients**

1 *Numerical* differentiation
`NUMERICAL_DERIVATIVES`

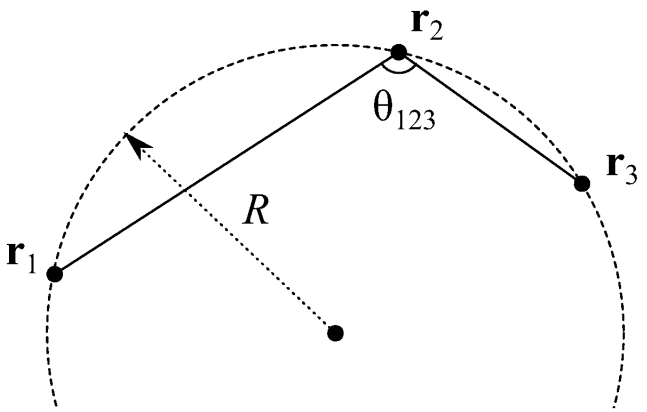
$$f(x) - f(x - 10^{-8})$$

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



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

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

```
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
```

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

* 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

```
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) {

   g1[0] = (1.0L/2.0L)*sqrt((pow(r_1x - r_3x, 2) + pow(r_1y - r_3y, 2) + pow(r_1z -
r_3z, 2))/(-pow((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)/((pow(r_1x - r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z -
r_2z, 2))*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))) +
1))*((1.0L/2.0L)*(2*r_1x - 2*r_3x)/(-pow((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)/((pow(r_1x - r_2x, 2) +
pow(r_1y - r_2y, 2) + pow(r_1z - r_2z, 2))*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y,
2) + pow(r_2z - r_3z, 2))) + 1) + (1.0L/2.0L)*((-2*r_1x + 2*r_2x)*pow((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)/(pow(pow(r_1x - r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z - r_2z, 2),
2)*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))) + (2*r_2x -
2*r_3x)*((r_1x - r_2x)*(r_2x - r_3x) + (r_1y - r_2y)*(r_2y - r_3y) + (r_1z -
r_2z)*(r_2z - r_3z))/((pow(r_1x - r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z - r_2z,
2))*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))))*(pow(r_1x -
r_3x, 2) + pow(r_1y - r_3y, 2) + pow(r_1z - r_3z, 2))/pow(-pow((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)/((pow(r_1x
- r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z - r_2z, 2))*(pow(r_2x - r_3x, 2) +
pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))) + 1, 2))*(-pow((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)/((pow(r_1x -
r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z - r_2z, 2))*(pow(r_2x - r_3x, 2) +
pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))) + 1)/(pow(r_1x - r_3x, 2) + pow(r_1y -
r_3y, 2) + pow(r_1z - r_3z, 2));
   g1[1] = (1.0L/2.0L)*sqrt((pow(r_1x - r_3x, 2) + pow(r_1y - r_3y, 2) + pow(r_1z -
r_3z, 2))/(-pow((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)/((pow(r_1x - r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z -
r_2z, 2))*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))) +
1))*((1.0L/2.0L)*(2*r_1y - 2*r_3y)/(-pow((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)/((pow(r_1x - r_2x, 2) +
pow(r_1y - r_2y, 2) + pow(r_1z - r_2z, 2))*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y,
2) + pow(r_2z - r_3z, 2))) + 1) + (1.0L/2.0L)*((-2*r_1y + 2*r_2y)*pow((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)/(pow(pow(r_1x - r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z - r_2z, 2),
2)*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))) + (2*r_2y -
2*r_3y)*((r_1x - r_2x)*(r_2x - r_3x) + (r_1y - r_2y)*(r_2y - r_3y) + (r_1z -
r_2z)*(r_2z - r_3z))/((pow(r_1x - r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z - r_2z,
2))*(pow(r_2x - r_3x, 2) + pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))))*(pow(r_1x -
r_3x, 2) + pow(r_1y - r_3y, 2) + pow(r_1z - r_3z, 2))/pow(-pow((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)/((pow(r_1x
- r_2x, 2) + pow(r_1y - r_2y, 2) + pow(r_1z - r_2z, 2))*(pow(r_2x - r_3x, 2) +
pow(r_2y - r_3y, 2) + pow(r_2z - r_3z, 2))) + 1, 2))*(-pow((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)/((pow(r_1x -
```

x 3

Common Subexpression Elimination

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 = (1.0L/2.0L)*p17/p6;
        double p19 = p10*p15;
        double p20 = p19*p7;
        double p21 = p10*p14*p15*p16*p6;
        double p22 = p19*p8;
        g1[1] = p18*(-p21*(p12 - p22) + p5);
        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);
    }
```

But: can not differentiate arbitrary functions

Algorithms ⊃ functions

- Differentiate this…

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

Algorithms ⊃ functions

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

… and generalized forces in biased sampling

tonigi / **plumed2-automatic-gradients**

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…

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

- 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

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

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

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

…etc

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

An intuitive idea…

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

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

…*many* coordinates…

$$\vdots \qquad \vdots$$

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

Hello, I am a program!

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

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

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

- Do it *reverse mode*

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

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 $\times$ ∂
- At end: whole gradient at once
  - Trades off memory for speed

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

Pass 1 *(in execution order)*

$$v_1 = x_1 \qquad -\,(!)$$

$$v_2 = x_2 \qquad -\,(!)$$

$$v_3 = v_1 v_2 \qquad \partial v_3/\partial v_1 = v_2; \quad \partial v_3/\partial v_2 = v_1$$

$$v_4 = \sin v_1 \qquad \partial v_4/\partial v_1 = \cos v_1$$

$$v_5 = v_3 + v_4 \qquad \partial v_5/\partial v_3 = +1; \quad \partial v_5/\partial v_4 = +1$$

Pass 2 *(in reference order)*

$$a_5 \leftarrow 1; \quad a_1 = \cdots = a_4 \leftarrow 0$$

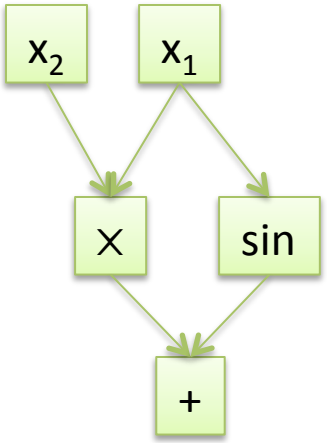
$$a_3 \leftarrow a_3 + a_5 \cdot \partial v_5/\partial v_3 \qquad (= a_5)$$

$$a_4 \leftarrow a_4 + a_5 \cdot \partial v_5/\partial v_4 \qquad (= a_5)$$

$$a_1 \leftarrow a_1 + a_4 \cdot \cos v_1$$

$$a_1 \leftarrow a_1 + a_3 \cdot v_2$$

$$a_2 \leftarrow a_2 + a_3 \cdot v_1$$



In practice,
for PLUMED...

mc-stan.org

Stan

stan-dev / math

arXiv:1509.07164

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

- We only use its **C++ library** (stan::math) providing automatic differentiation and math functions [BSD-3]

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

eigen.tuxfamily.org

- 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
```

tonigi / plumed2-automatic-gradients

… / CurvatureAutoDiff.cpp

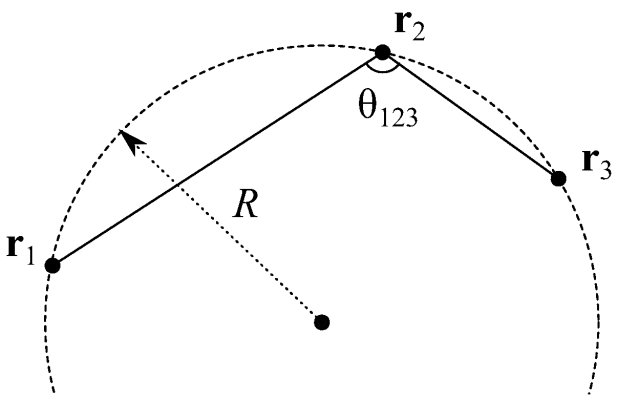
```cpp
struct curvature_fun {
private:
  bool inverse;           // List of parameters
public:
  curvature_fun(bool inverse): inverse(inverse) {}

  template <typename T>
  T operator()(const Matrix<T, Dynamic, 1>& x)
  const {
    // Split into 3D vectors for convenience
    Matrix<T, 3, 1> r1, r2, r3;
    r1 = x.segment(0,3);
    r2 = x.segment(3,3);
    r3 = x.segment(6,3);

    Matrix<T, 3, 1> r12, r32, r13;
    r12 = r1-r2;                  // Triangle sides
    r32 = r3-r2;
    r13 = r1-r3;

    T cos2_a = pow(r12.dot(r32),2.0)
               / r12.dot(r12) / r32.dot(r32);
    T sin2_a = 1.0 - cos2_a;

    T radius2 = r13.dot(r13) / sin2_a / 4.0;
    T radius = sqrt(radius2);     // Eq. (1)

    if(inverse)
      radius = 1.0/radius;

    return(radius);
  }
};
```

Templated <T> 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);
```

src/curvature_codegen

CURVATURE_CODEGEN
CURVATURE_MULTICOLVAR_CODEGEN

calculate the moments of the distribution of collective variables. The mth 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...

MOMENTS

Examples

The following input tells plumed to calculate the curvature radiuses 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.

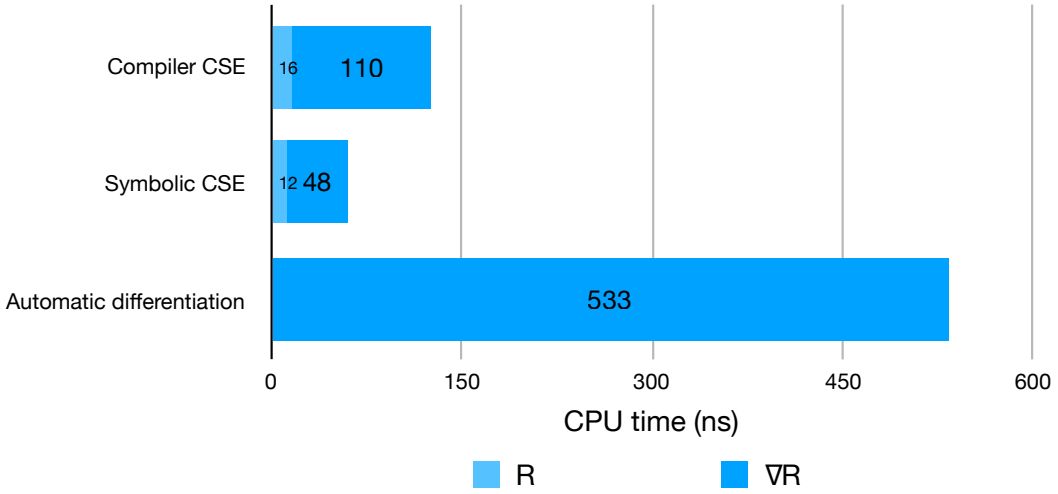
```
CURVATURE_MULTICOLVAR_CODEGEN POLYMER=1-5 MIN={BETA=0.1} LABEL=d1
```

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

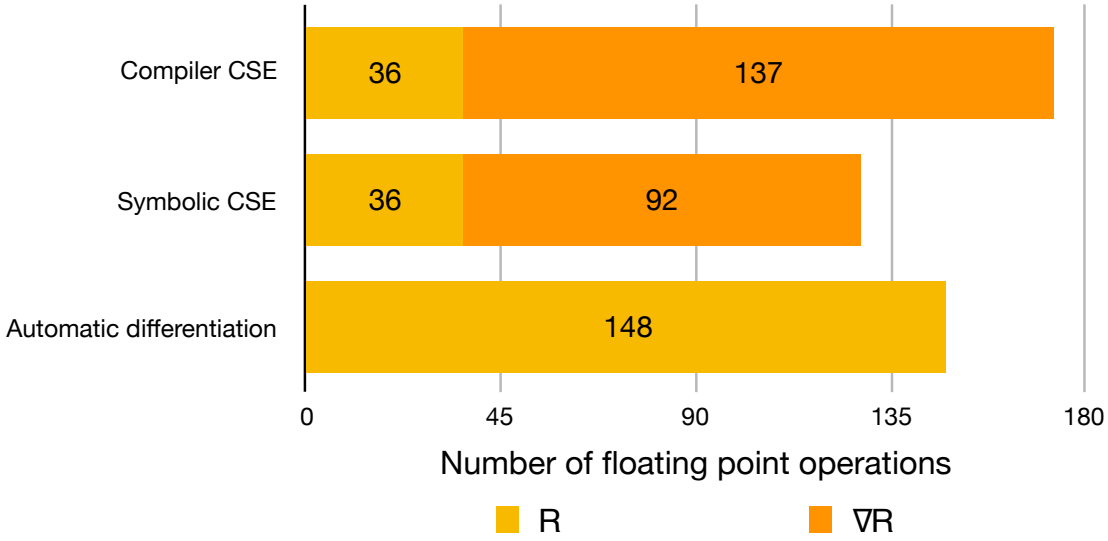
For `CURVATURE` – few ops., so not representative of complex CVs



Naive code generation, then optimization by compiler

Code generation+CSE: as fast as it gets

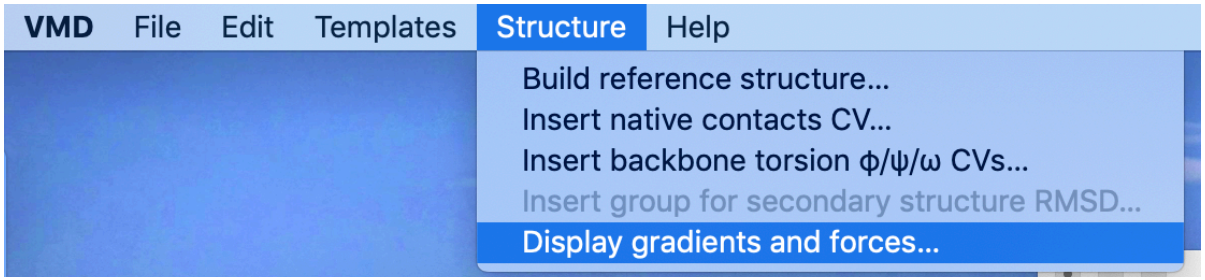
Auto-diff: small (in absolute terms) price for full generality



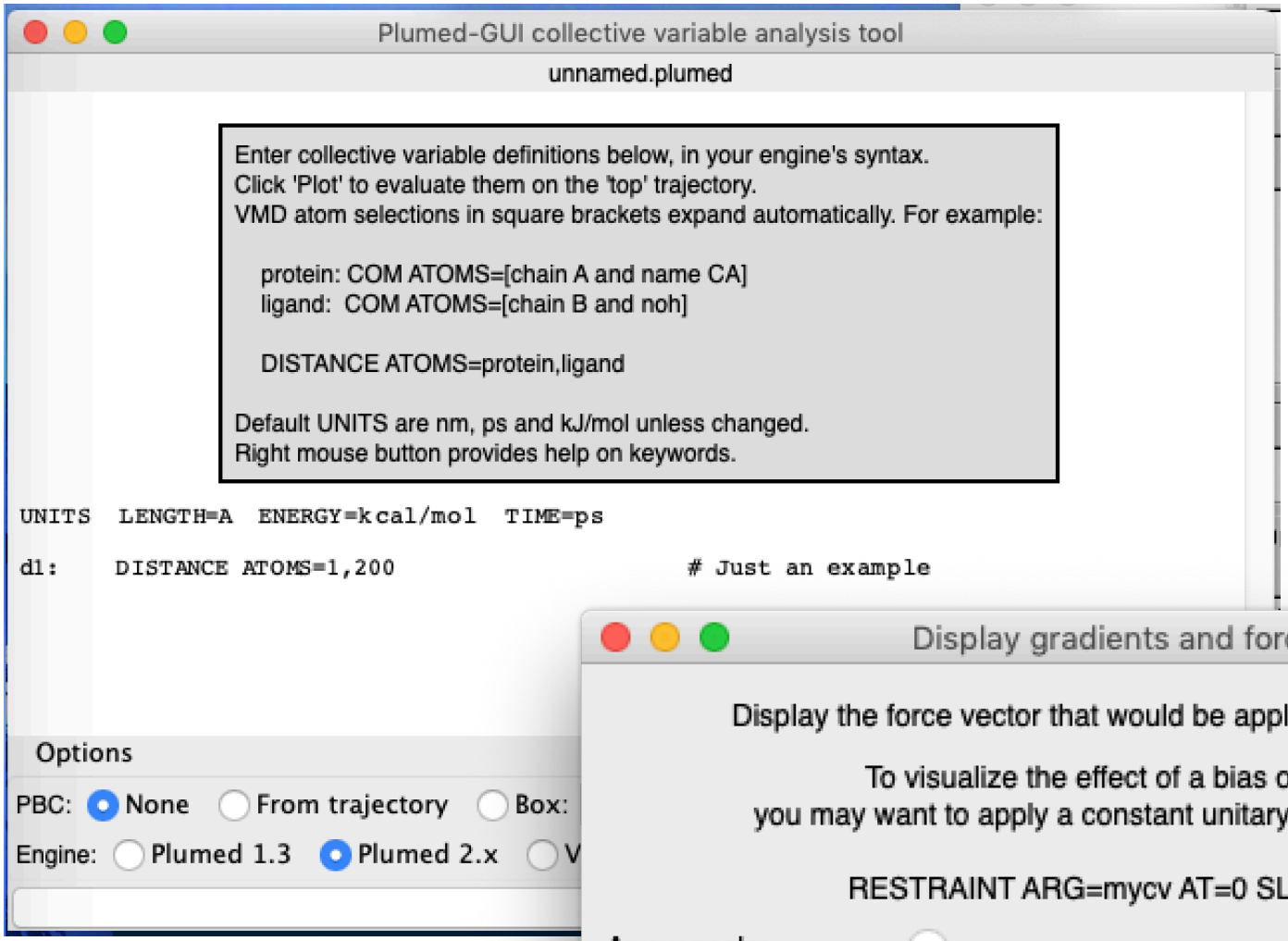
Measured through hardware FP counters, **`FP_COMP_OPS_EXE: SSE_SCALAR_DOUBLE`** etc: all nearly optimal

Quiz

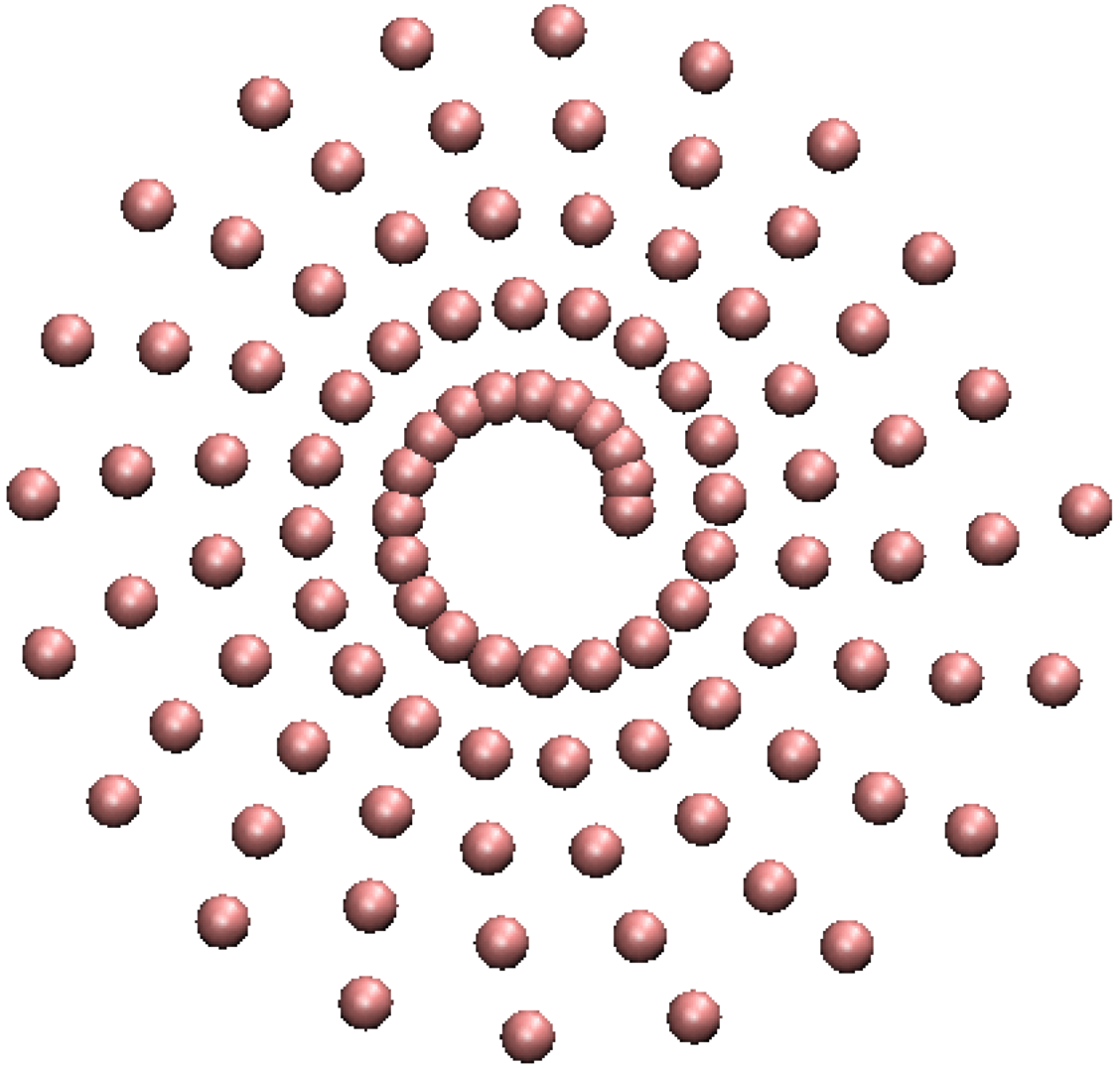
How do gradients look like?



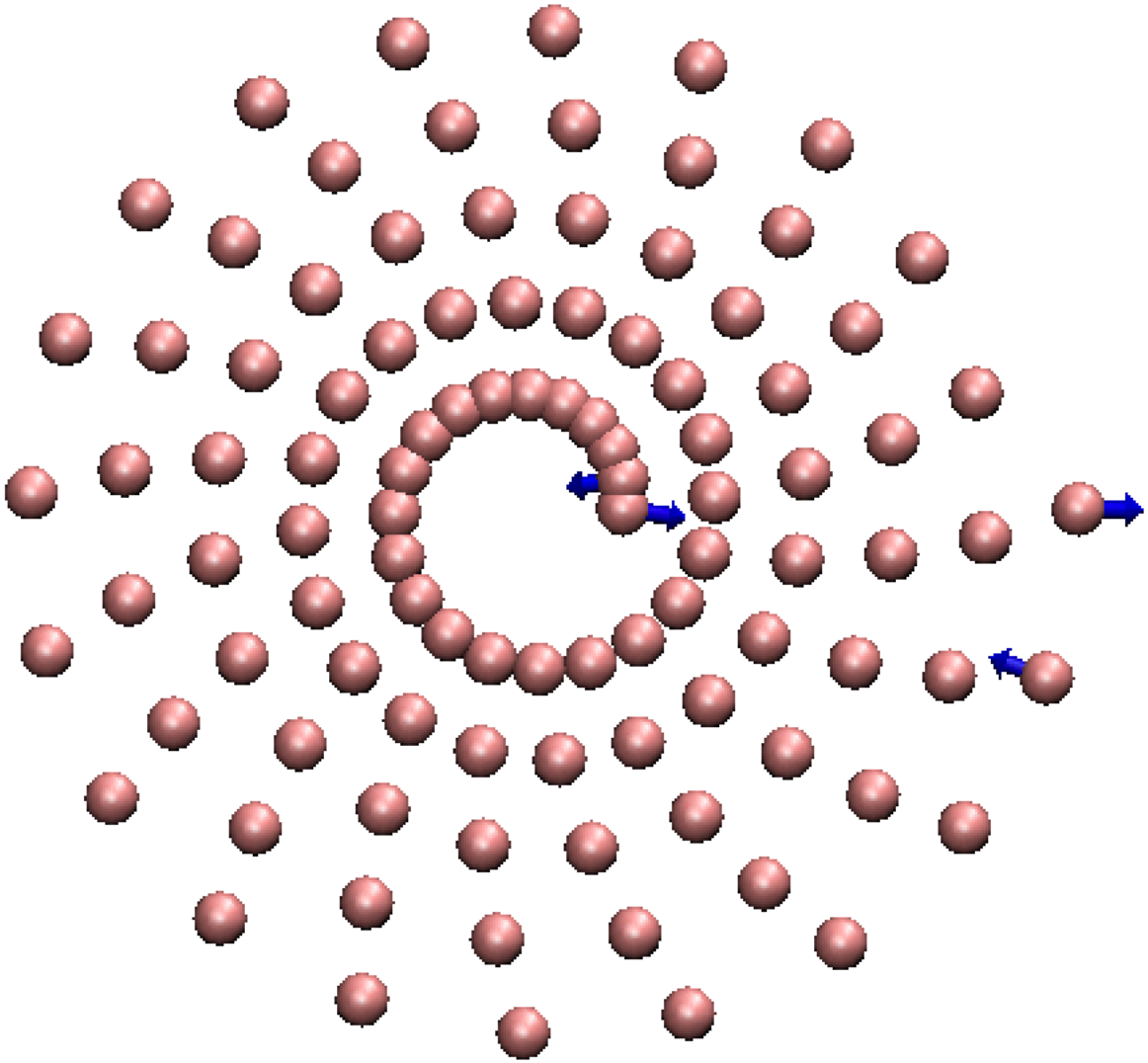
← VMD-Plumed Plugin
(under Extensions)



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)