



MIDWEST INTEGRATED CENTER FOR COMPUTATIONAL MATERIALS

<http://miccom-center.org>

2017 Summer School

Topic: How to Add a New Collective Variable (CV)

Presenter: Michael A. Webb, University of Chicago

How to Add a new CV...



Collective variables (CVs) are *system descriptors with reduced complexity* compared to the full phase space. They are functions of many variables that are useful as biasing coordinates and macro-state analysis.

Many CVs already included in SSAGES

- *Angles*
- *Box Volume*
- *Gyration Tensor*
- *Particle Coordinate*
- *Particle Separation*
- *Torsions*
- *Secondary Structure*
- *Rouse Modes*

SSAGES makes it very easy to add new CVs to accomplish your research goals

- *modular (minimal tinkering with code)*
- *straightforward (organized structure)*
- *efficient (many existing tools)*

What you need:

- *literacy and generic programming*
- *basics of C++ syntax*
- *some familiarity with MPI*

...but current research often dictates specific use cases/analysis → more complex CVs not presently featured

Next 20-25 minutes:



Basic Overview



A new CV can be added by creating a new class that derives from the `CollectiveVariable` base class in SSAGES

This only requires the addition/modification of 3 files!

From within the
SSAGES directory:

```
[webbm@midway-login2 SSAGES]$ ls
build          doc           hooks         LICENSE.txt   schema        test
CMakeLists.txt Examples      include       README.md    src           Tools
```

`./src/CVs/CollectiveVariables.cpp`

Only need to add 2 lines here to handle “building” of your CV

`./src/CVs/CoolThingCV.h` **(new)**

The bulk of your effort and coding goes here to edit/add member functions (straightforward to just copy format of existing files)

| | | | | |
|------------------------------|-----------------------------------|---|--|-----|
| <code>CoolThingCV()</code> , | <code>Initialize()</code> , | <code>Evaluate()</code> , | <code>Build()</code> , | ... |
| <i>constructor</i> | <i>checks necessary variables</i> | <i>computes your CV value (and gradient!)</i> | <i>builds your CV object from JSON</i> | |

`./schema/CVs/coolthing.cv.json` **(new)**

Sets expectations for JSON fields that must be defined for the CV class

Basic Overview



A new CV can be added by creating a new class that derives from the `CollectiveVariable` base class in SSAGES

Step 1. *Formulate the CV*

Step 2. *Begin writing `CoolThingCV.h`*

Step 3. *Craft your JSON schema*

Step 4. *Finish writing `CoolThingCV.h`*

Step 5. *Make it buildable*

```
./src/CVs/CollectiveVariables.cpp
```

Only need to add 2 lines here to handle “building” of your CV

```
./src/CVs/CoolThingCV.h (new)
```

The bulk of your effort and coding goes here to edit/add member functions (straightforward to just copy format of existing files)

| | | | |
|------------------------------|-----------------------------------|---|--|
| <code>CoolThingCV()</code> , | <code>Initialize()</code> , | <code>Evaluate()</code> , | <code>Build()</code> , ... |
| <i>constructor</i> | <i>checks necessary variables</i> | <i>computes your CV value (and gradient!)</i> | <i>builds your CV object from JSON</i> |

```
./schema/CVs/coolthing.cv.json (new)
```

Sets expectations for JSON fields that must be defined for the CV class

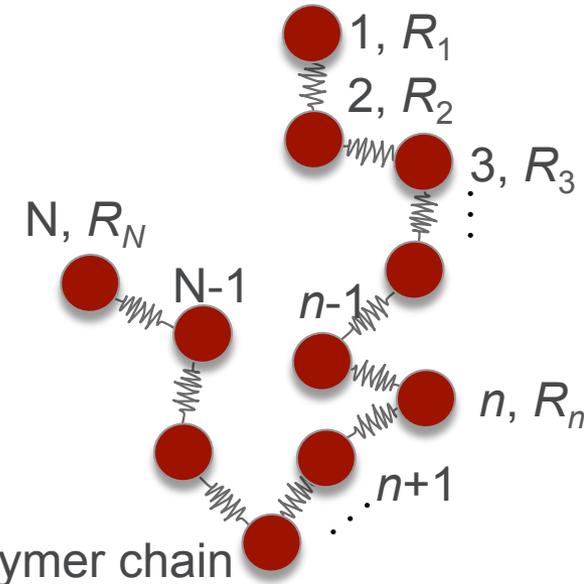
Example: Polymer Rouse modes



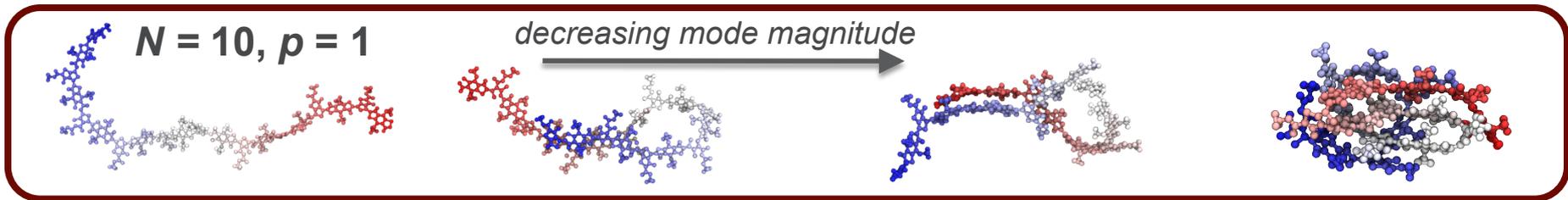
The Rouse modes of a polymer are CVs that involve the coordinates of all “beads” in the polymer; they ...

- represent the normal-mode coordinates for a Gaussian chain
- describe dynamics/relaxation over different lengthscales
 - $p=0$ describes the chain center-of-mass
 - $p>0$ describes sub-chains of $(N-1)/p$ beads
- characterize chain conformations at those lengthscales

$$X_p = \sqrt{\frac{c_p}{N}} \sum_{i=1}^N \mathbf{R}_i \cos\left[\frac{p\pi}{N}\left(i - \frac{1}{2}\right)\right] \quad c_p = \begin{cases} 1, & \text{if } p = 0 \\ 2, & \text{otherwise} \end{cases}$$



- \mathbf{R}_i Cartesian vector coordinate of the i th bead in the polymer chain
- N Number of beads comprising the polymer chain
- p Rouse mode index
- X_p Vector coordinate of the p th Rouse mode



Step 1. Formulate the CV



What are the probable use cases?

Atomistic or coarse-grained polymer/macromolecule simulations
Different modes and discretization levels

How will it be calculated? (must be a scalar)

We will compute the CV as the Euclidean norm of the Rouse mode coordinate:

$$CV = \sqrt{X_p \cdot X_p}$$

Bead coordinates will be the center-of-mass of a group of particles:

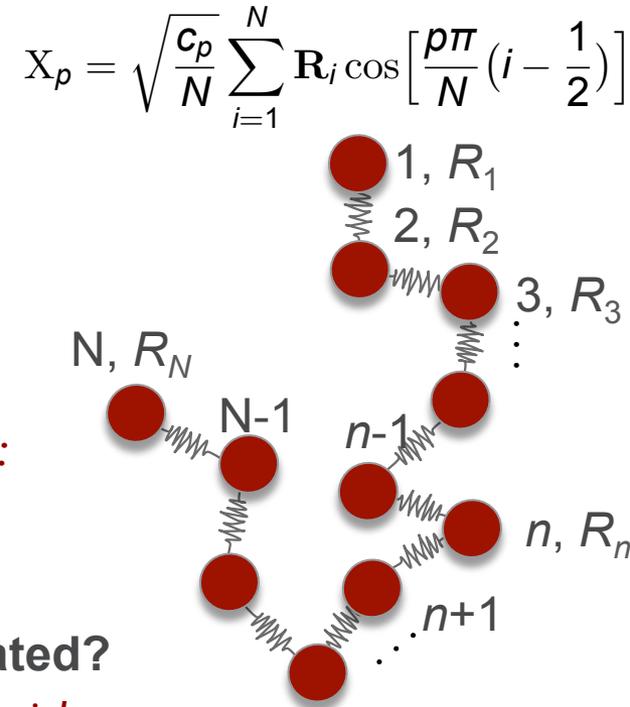
$$\mathbf{R}_i = \frac{1}{M_i} \sum_{j \in \mathbb{G}_i} m_j \mathbf{r}_j \quad \mathbb{G}_i = \{\text{id}_1, \text{id}_2, \dots, \text{id}_{N_i}\}$$

How will the gradient (for particle positions) be calculated?

By chain rule, the gradient with respect to the position of the j^{th} particle:

$$\nabla_{\mathbf{r}_j} CV = \frac{X_p}{CV} \sqrt{\frac{c_p}{N}} \sum_{i=1}^N \cos \left[\frac{\rho\pi}{N} \left(i - \frac{1}{2} \right) \right] \frac{m_j}{M_i} \underline{\delta_j(\mathbb{G}_i)}$$

1 if j is in the group
 0 otherwise



What information will be needed?

- *mode index*
- *number of “beads”*
- *Particle positions/masses → bead position/mass*
- *atom indices comprising the particle groups*

Note: You can assume that all typical simulation info (particle positions, velocities, masses, etc.) is exposed within SSAGES through “Snapshot” (more on this later)

Step 2. Begin writing CoolThingCV.h



CoolThingCV.h defines your CV class and it is derived from
CollectiveVariable

Base Class Snippet:

```
./src/CVs/CollectiveVariable.h
#pragma once
#include "../JSON/Serializable.h"
#include "types.h"
#include <vector>

// Forward declare.
namespace Json {
    class Value;
}

namespace SSAGES
{
    /** Abstract class for a collective variable.
     * \ingroup CVs
     */
    class CollectiveVariable
    {
    protected:
        /** Gradient vector dCv/dxi.
         * \code std::vector<Vector3> grad_;
        */
        std::vector<Vector3> grad_;

        /** Gradient w.r.t box vectors dCv/dHij.
         * \code Matrix3 boxgrad_;
        */
        Matrix3 boxgrad_;

        /** Current value of CV.
         * \code double val_;
        */
        double val_;

        /** Bounds on CV.
         * \code std::array<double, 2> bounds_;
        */
        std::array<double, 2> bounds_;
    public:
        /** Constructor.
         * \code CollectiveVariable() :
         * \code grad_(0), boxgrad_(Matrix3::Zero()), val_(0), bounds_{{0,0}}
         * \code {}
         */
        CollectiveVariable() :
            grad_(0), boxgrad_(Matrix3::Zero()), val_(0), bounds_{{0,0}}
        {}

        /** Destructor.
         * \code virtual ~CollectiveVariable(){}
         */
        virtual ~CollectiveVariable(){}
    };
};
```

some variable types are pre-defined for convenience, i.e., Vector3 in place of Eigen::Vector3d or Label in place of std::vector<int>

private class variable names indicated by a trailing '_'

The calculation of these protected variables in YOUR CoolThingCV.h makes the CV unique

Step 2. Begin writing CoolThingCV.h



To write CoolThingCV.h, we need to craft *four* main member functions; we will start with the first *three*

In Evaluate(), we need to compute val_ and grad_; everything else is mostly bookkeeping

```
#pragma once
#include "CollectiveVariable.h" ./src/CVs/MockCV.h
#include <array>
#include <cmath>

namespace SSAGES
{
    ///! Mock collective variable for testing purposes.
    class MockCV : public CollectiveVariable
    {
    private:
        ///! User defined gradient vector.
        Vector3 usergrad_;

    public:
        ///! Constructor.
        /*!
         * \param value Value the Mock CV will return.
         * \param grad Gradient vector the Mock CV will return.
         * \param upper Value for the upper bound of the CV.
         * \param lower Value for the lower bound of the CV.
         */
        Construct a mock CV
        /*!
         * \param value Value the Mock CV will return.
         * \param grad Gradient vector the Mock CV will return.
         * \param upper Value for the upper bound of the CV.
         * \param lower Value for the lower bound of the CV.
         */
        MockCV(double value, const Vector3& grad, double upper, double lower) :
            usergrad_(grad)
        {
            val_ = value;
            bounds_ = {{upper, lower}};
        }

        ///! Initialize necessary variables.
        /*!
         * \param snapshot Current simulation snapshot.
         */
        void Initialize(const Snapshot& snapshot) override
        {
            // Initialize gradient
            auto n = snapshot.GetPositions().size();
            grad_.resize(n, usergrad_);
        }

        ///! Evaluate the CV.
        /*!
         * \param snapshot Current simulation snapshot.
         */
        void Evaluate(const Snapshot& snapshot) override
        {
        }
    };
}
```

private variable
& function space

public variable &
function space

1. the constructor—called
when the CV is setup

2. Initialize() — called at
beginning of simulation

3. Evaluate() — called at
every simulation timestep

My suggested workflow:

- Shamelessly copy the code for an existing CV with similar features
- Make all your private variable declarations (that you can initially think of)
- Write the constructor
- Write Initialize()
- Write Evaluate()
- Patch up needed variables/functions
- Write Build()

Step 2. Begin writing CoolThingCV.h



```
namespace SSAGES
{
    ParticleSeparationCV.h
    /** Collective variable on the distance between two particles' centers of mass.
    /**
    * Collective variable on two particle positions. This CV will return the
    * distance between two specific atom groups of the simulation.
    *
    * \ingroup CVs
    */
    class ParticleSeparationCV : public CollectiveVariable
    {
    private:
        Label group1_; /**< Atom ID's of group 1.
        Label group2_; /**< Atom ID's of group 2.

        /** Each dimension determines if it is applied by the CV.
        Bool3 dim_;

    public:
        /** Constructor
        ParticleSeparationCV(const Label& group1, const Label& group2) :
        group1_(group1), group2_(group2), dim_{true, true, true}
        {}

        /** Constructor
        ParticleSeparationCV(const Label& group1, const Label& group2, Bool3 dim) :
        group1_(group1), group2_(group2), dim_(dim)
        {}
    }
```

a. Shamelessly copy the code for an existing CV with similar features

```
namespace SSAGES
{
    RouseModeCV.h
    /** Collective variable is a Rouse mode for a polymer chain comprised of N particle groups
    /**
    * This CV returns the value for the p'th Rouse mode, computed as
    *  $X_p(t) \sim (1/N) \sum_{i=1}^N r_i(t) \cos[\pi(n-0.5)p/N]$ ,
    * where N is the number of particle groups, p is the mode index, r_i is the center-of-mass position of
    * a collection of atoms comprising the i'th bead in the N-bead polymer chain
    *
    * \ingroup CVs
    */
    class RouseModeCV : public CollectiveVariable
    {
    private:
        std::vector<Label> groups_; // vector of groups of indices to define the particle groups
        std::vector<double> massg_; // vector of the total mass for each particle group
        int N; // number of particle groups
        int p; // index of relevant Rouse mode
        Vector3 xp_; // 3d vector for containing vectorial rouse amplitude
        std::vector<Vector3> r_; // vector of coordinate positions for each bead

    public:
        /** Basic Constructor for Rouse Mode CV
        /**
        * \param groups - vector of vector of atom IDs, each group comprising a bead in the Rouse chain
        * \param p - index for relevant Rouse mode
        */
        RouseModeCV(const std::vector<Label>& groups, int p) :
        groups_(groups), p_(p), N_( groups_.size())
        { massg_.resize( groups_.size(),0.0 ); }
    }
```

b. Make all your private variable declarations

What do we need to compute val_ and grad_?

c. Write the constructor

Should assume what information gets passed to the constructor here

Step 2. Begin writing CoolThingCV.h



```
#!/ Initialize necessary variables.
void Initialize(const Snapshot& snapshot) override
{
    using std::to_string;

    auto n1 = group1_.size(), n2 = group2_.size();

    // Make sure atom ID's are on at least one processor.
    std::vector<int> found1(n1, 0), found2(n2, 0);
    for(size_t i = 0; i < n1; ++i)
    {
        if(snapshot.GetLocalIndex(group1_[i]) != -1)
            found1[i] = 1;
    }

    for(size_t i = 0; i < n2; ++i)
    {
        if(snapshot.GetLocalIndex(group2_[i]) != -1)
            found2[i] = 1;
    }

    MPI_Allreduce(MPI_IN_PLACE, found1.data(), n1, MPI_INT, MPI_SUM, snapshot.GetCommunicator());
    MPI_Allreduce(MPI_IN_PLACE, found2.data(), n2, MPI_INT, MPI_SUM, snapshot.GetCommunicator());

    unsigned ntot1 = std::accumulate(found1.begin(), found1.end(), 0, std::plus<int>());
    unsigned ntot2 = std::accumulate(found2.begin(), found2.end(), 0, std::plus<int>());
    if(ntot1 != n1)
        throw BuildException({
            "ParticleSeparationCV: Expected to find " +
            to_string(n1) +
            " atoms in particle 1, but only found " +
            to_string(ntot1) + "."
        });

    if(ntot2 != n2)
        throw BuildException({
            "ParticleSeparationCV: Expected to find " +
            to_string(n2) +
            " atoms in particle 2, but only found " +
            to_string(ntot2) + "."
        });
};
};
```

ParticleSeparationCV.h

This looks good

We should add checks on the mode index

This checks groups 1 and 2; we should change to check the vector of groups

We should set up the same, tailored for the RouseModeCV

d. Write Initialize()

Initialization appears to mostly be checking for consistency between variables from simulation and user-supplied parameters

```
#!/ Initialize necessary variables.
RouseModeCV.h
/*!
 * \param snapshot Current simulation snapshot.
 */
void Initialize(const Snapshot& snapshot) override
{
    using std::to_string;

    // Check for valid p_
    if (p_ > groups_.size())
        throw std::invalid_argument(
            "RouseModeCV: Expected to find p to be less than " +
            to_string(groups_.size()) + " but found p = " +
            to_string(p_)
        );

    // Check for valid groups
    for (size_t j = 0; j < groups_.size(); ++j) {
        auto nj = groups_[j].size();
        // Make sure atom IDs in the group are somewhere
        std::vector<int> found(nj,0);
        for (size_t i = 0; i < nj; ++i) {
            if(snapshot.GetLocalIndex(groups_[j][i]) != -1)
                found[i] = 1;
        }

        MPI_Allreduce(MPI_IN_PLACE, found.data(), nj, MPI_INT, MPI_SUM, snapshot.GetCommunicator());
        unsigned njtot = std::accumulate(found.begin(), found.end(), 0, std::plus<int>());

        if(njtot != nj)
            throw std::invalid_argument(
                "RouseModeCV: Expected to find " +
                to_string(nj) +
                " atoms in group " + to_string(j) + ", but only found " +
                to_string(njtot) + "."
            );
    }

    // Set the masses of each particle group in massq_
    this->setMasses( groups_, snapshot);
};
```

We'll add a placeholder for initializing the masses of the groups

Step 2. Begin writing CoolThingCV.h



e. Write Evaluate()

Available information in Evaluate() is determined during construction/initialization or provided through snapshot

```
void Evaluate(const Snapshot& snapshot) override
{
    // Get local atom indices.
    std::vector<int> idx1, idx2;
    snapshot.GetLocalIndices(group1_, &idx1);
    snapshot.GetLocalIndices(group2_, &idx2);
    // Get data from snapshot.
    auto n = snapshot.GetNumAtoms();
    const auto& masses = snapshot.GetMasses();
    // Initialize gradient.
    std::fill(grad_.begin(), grad_.end(), Vector3{0,0,0});
    grad_.resize(n, Vector3{0,0,0});
    boxgrad_ = Matrix3::Zero();
    // Get centers of mass.
    auto mtot1 = snapshot.TotalMass(idx1);
    auto mtot2 = snapshot.TotalMass(idx2);
    Vector3 com1 = snapshot.CenterOfMass(idx1, mtot1);
    Vector3 com2 = snapshot.CenterOfMass(idx2, mtot2);
    // Account for pbc.
    Vector3 rij = snapshot.ApplyMinimumImage(com1 - com2).cwseProduct(dim_.c);
    // Compute gradient.
    val_ = rij.norm();
    for(auto& id : idx1)
    {
        grad_[id] = rij/val_*masses[id]/mtot1;
        boxgrad_ += grad_[id]*rij.transpose();
    }
    for(auto& id : idx2)
    {
        grad_[id] = -rij/val_*masses[id]/mtot2;
    }
}
```

✓indices for a group of particles

✓masses for particles

✓mass of a group

✓center-of-mass

✓MIC

✓vector norm

ParticleSeparationCV.h

```
void Evaluate(const Snapshot& snapshot) override
{
    // Get data from snapshot.
    auto ntot = snapshot.GetNumAtoms(); // total number of atoms
    const auto& masses = snapshot.GetMasses(); // mass of each atom
    // Initialize working variables
    double ppi_n = p_ * M_PI / N_; // constant
    xp_.fill(0.0); // vectorial Rouse mode
    r_.resize(N_); // position vector for beads in Rouse chain (unwrapped)
    grad_.resize(ntot, Vector3{0,0,0}); // gradient set to 0.0 for all atoms
    std::fill(grad_.begin(), grad_.end(), Vector3{0,0,0});
    // Iterate over all N_ atom groups and compute the center of mass for each
    std::vector<Vector3> rcom; // vector of COM positions
    for (size_t i = 0; i < N_; ++i) {
        Label idi; // list of indices
        snapshot.GetLocalIndices(groups_[i], &idi);
        rcom.push_back(snapshot.CenterOfMass(idi, massg_[i])); // center of mass for group
    }
    // Now compute differences vectors between the neighboring beads
    // accumulate displacements to reconstruct unwrapped polymer chain
    // for simplicity, we consider the first bead to be the reference position
    // in all snapshots
    r_[0] = rcom[0];
    for (size_t i = 1; i < N_; ++i) {
        Vector3 dri = snapshot.ApplyMinimumImage(rcom[i] - rcom[i-1]);
        r_[i] = r_[i-1] + dri; // r_i = r_{i-1} + (r_i - r_{i-1})
    }
    // Determine the value of the Rouse coordinate
    // Xp(t) = 1/sqrt(N) * sum_{i=1}^N ri, p = 0
    // Xp(t) = sqrt(2/N) * sum_{i=1}^N ri * cos[p*pi*(i-0.5)], p = 1,...,N-1
    // Note: this solution is valid for homogeneous friction
    xp_.fill(0.0);
    for (size_t i = 0; i < N_; ++i) {
        xp_ += r_[i]*cos ( ppi_n * (i+0.5) );
    }
    xp_ /= sqrt(N_);
    if ( p_ != 0 ) xp_ *= sqrt(2.0) ;
    // Compute Rouse vector norm as the CV
    // CV = sqrt(Xp*Xp), Xp = (Xp1,Xp2,Xp3)
    val_ = xp_.norm();
    // Now perform gradient operation
    // dCV/dxjd = (Xpd/CV)*(c/N)**0.5*sum_i=1^N cos[p*pi*(i-0.5)/N] mj/Mi *delta_j({i})
    // delta_j({i}) = 1, if j in {i}, 0 otherwise
    Vector3 gradpcon = xp_ / sqrt(N_) / val_;
    if ( p_ != 0 ) gradpcon *= sqrt(2.0); // (Xpd/CV)*(c/N)**0.5
    for (size_t i = 0; i < N_; ++i) {
        Label idi; // list of indices
        snapshot.GetLocalIndices(groups_[i], &idi);
        // go over each atom in the group and add to its gradient
        // Note: performance tradeoff here. All gradient elements have a common factor of
        // Xpd/CV*sqrt(c/N) = prefactor, with c = 2 if p != 0
        // this could be factored out, but if ntot >> number of atoms in groups
        // then it won't be worth it to post multiply all gradient terms...
        // Could also go over loop again after to do the multiplication, but
        // that is troublesome if atom ids appear in multiple groups for some reason
        double cosval = cos(ppi_n*(i+0.5)) / massg_[i]; // cos[p*pi*(i-0.5)/N] / Mi
        for (auto& id : idi) {
            grad_[id] += gradpcon* cosval * masses[id];
        }
    }
}
```

RouseModeCV.h

Note: Most of Evaluate() changes, BUT you can learn a lot about the SSAGES snapshot functionality by looking at the various .h files. Otherwise, take a look at ./src/Snapshot.h or the API reference to see all the member variable/functions.

Step 2. Begin writing CoolThingCV.h



RouseModeCV.h

f. Patch up needed variables/functions

Earlier, we assumed a function that would determine the masses of the “beads.” Now is the time to actually make sure these functions exist!

```

//! Initialize necessary variables.
//!
//! * \param snapshot Current simulation snapshot.
//! */
void Initialize(const Snapshot& snapshot) override
{
    using std::to_string;

    // Check for valid p_
    if (p_ > groups_.size())
        throw std::invalid_argument(
            "RouseModeCV: Expected to find p to be less than " +
            to_string(groups_.size()) + " but found p = " +
            to_string(p_)
        );

    // Check for valid groups
    for (size_t j = 0; j < groups_.size(); ++j) {
        auto nj = groups_[j].size();
        // Make sure atom IDs in the group are somewhere
        std::vector<int> found(nj,0);
        for (size_t i = 0; i < nj; ++i) {
            if(snapshot.GetLocalIndex(groups_[j][i]) != -1)
                found[i] = 1;
        }

        MPI_Allreduce(MPI_IN_PLACE, found.data(), nj, MPI_INT, MPI_SUM, snapshot.GetCommunicator());
        unsigned njtot = std::accumulate(found.begin(), found.end(), 0, std::plus<int>());

        if(njtot != nj)
            throw std::invalid_argument(
                "RouseModeCV: Expected to find " +
                to_string(nj) +
                " atoms in group " + to_string(j) + ", but only found " +
                to_string(njtot) + "."
            );
    }
}

```

```

public:
//! Basic Constructor for Rouse Mode CV
//!
//! * \param groups - vector of vector of atom IDs, each group comprising a bead
//! * \param p - index for relevant Rouse mode
//! */
RouseModeCV(const std::vector<Label>& groups, int p) :
groups_(groups), p_(p), N_( groups_.size())
{ massg_.resize( groups_.size(),0.0 ); }

//! Helper function to determine masses of each group
//! Note: this here assumes that the masses of each group are not changing during
//! the simulation, which is likely typical...
//! * \param groups - vector of vector of atom IDs, each group comprising a bead
//! */
void setMasses(const std::vector<Label>& groups, const Snapshot& snapshot ) {
// Compute mass of each group and store to massg_
double massi; Label listi;
for (size_t i = 0; i < N_; ++i) {
    listi = groups[i]; // MW: can this be used directly in TotalMass
    Label idi;
    snapshot.GetLocalIndices(listi, &idi);
    massg_[i] = snapshot.TotalMass(idi);
}
}

```

```

// Compute mass of each particle group in massg_
// * \param groups - vector of vector of atom IDs, each group comprising a bead
// * \param snapshot - current simulation snapshot
// */
void setMasses(const std::vector<Label>& groups, const Snapshot& snapshot) {
// Compute mass of each particle group and store to massg_
double massi; Label listi;
for (size_t i = 0; i < N_; ++i) {
    listi = groups[i]; // MW: can this be used directly in TotalMass
    Label idi;
    snapshot.GetLocalIndices(listi, &idi);
    massg_[i] = snapshot.TotalMass(idi);
}
}

```

Step 3. Craft the JSON Schema



A file specifying the JSON schema must be created in `./schema/CVs/`

This file acts as pre-filter that sets the conditions for variables in your programming environment

```
{
  "type" : "object",
  "varname" : "ParticleSeparationCV",
  "properties" : {
    "type" : {
      "type" : "string",
      "enum" : ["ParticleSeparation"]
    },
    "group1" : {
      "type" : "array",
      "minItems" : 1,
      "items" : {
        "type" : "integer",
        "minimum" : 0
      }
    },
    "group2" : {
      "type" : "array",
      "minItems" : 1,
      "items" : {
        "type" : "integer",
        "minimum" : 0
      }
    },
    "bounds" : {
      "type" : "array",
      "minItems" : 2,
      "maxItems" : 2,
      "items" : {
        "type" : "number"
      }
    },
    "dimension" : {
      "type" : "array",
      "minItems" : 3,
      "maxItems" : 3,
      "items" : {
        "type" : "boolean"
      }
    }
  },
  "required" : ["type", "group1", "group2"],
  "additionalProperties" : false
}
particleseparation.CV.json
```

Copy and modify
existing schema
according to desired
functionality

```
{
  "type" : "object",
  "varname" : "RouseModeCV",
  "properties" : {
    "type" : {
      "type" : "string",
      "enum" : ["RouseMode"]
    },
    "groups" : {
      "type" : "array",
      "minItems" : 1,
      "items" : {
        "type" : "array",
        "minItems" : 1,
        "items" : {
          "type" : "integer",
          "minimum" : 0
        }
      }
    },
    "mode" : {
      "type" : "integer",
      "minItems" : 1,
      "maxItems" : 1,
      "minimum" : 0
    }
  },
  "required" : ["type", "groups", "mode"],
  "additionalProperties" : false
}
rousemode.cv.json
```

Step 4. Finish up CoolThingCV.h



Knowing the set up of the JSON, we need to finish up CoolThingCV.h

```
{
  "type" : "object",
  "varname" : "ParticleSeparationCV",
  "properties" : {
    "type" : {
      "type" : "string",
      "enum" : ["ParticleSeparation"]
    },
    "group1" : {
      "type" : "array",
      "minItems" : 1,
      "items" : {
        "type" : "integer",
        "minimum" : 0
      }
    },
    "group2" : {
      "type" : "array",
      "minItems" : 1,
      "items" : {
        "type" : "integer",
        "minimum" : 0
      }
    },
    "bounds" : {
      "type" : "array",
      "minItems" : 2,
      "maxItems" : 2,
      "items" : {
        "type" : "number"
      }
    },
    "dimension" : {
      "type" : "array",
      "minItems" : 3,
      "maxItems" : 3,
      "items" : {
        "type" : "boolean"
      }
    }
  },
  "required": ["type", "group1", "group2"],
  "additionalProperties": false
}
particleseparation.cv.json
```

g. Write Build()

```
static ParticleSeparationCV* Build(const Json::Value& json, const std::string& path)
{
  Json::ObjectRequirement validator;
  Json::Value schema;
  Json::Reader reader;

  reader.parse(JsonSchema::ParticleSeparationCV, schema);
  validator.Parse(schema, path);

  // Validate inputs.
  validator.Validate(json, path);
  if(validator.HasErrors())
    throw BuildException(validator.GetErrors());

  std::vector<int> group1, group2;

  for(auto& s : json["group1"])
    group1.push_back(s.asInt());

  for(auto& s : json["group2"])
    group2.push_back(s.asInt());

  ParticleSeparationCV* c;
  if(json.isMember("dimension"))
  {
    auto fixx = json["dimension"][0].asBool();
    auto fixy = json["dimension"][1].asBool();
    auto fixz = json["dimension"][2].asBool();
    c = new ParticleSeparationCV(group1, group2, fixx, fixy, fixz);
  }
  else
    c = new ParticleSeparationCV(group1, group2);

  return c;
}
```

This looks fairly generic—will mostly copy

This is specific to the JSON, so we can match it to our own needs

ParticleSeparationCV.h

Step 4. Finish up CoolThingCV.h



Knowing the set up of the JSON, we need to finish up CoolThingCV.h

```
{
  "type" : "object",
  "varname" : "RouseModeCV",
  "properties" : {
    "type" : {
      "type" : "string",
      "enum" : ["RouseMode"]
    }
    "groups" : {
      "type" : "array",
      "minItems" : 1,
      "items" : {
        "type" : "array",
        "minItems" : 1,
        "items" : {
          "type" : "integer",
          "minimum" : 0
        }
      }
    }
    "mode" : {
      "type" : "integer",
      "minItems" : 1,
      "maxItems" : 1,
      "minimum" : 0
    }
  },
  "required": ["type", "groups", "mode"],
  "additionalProperties": false
}
rousemode.cv.json
```

g. Write Build()

```
static RouseModeCV* Build(const Json::Value& json, const std::string& path)
{
  Json::ObjectRequirement validator;
  Json::Value schema;
  Json::Reader reader;

  reader.parse(JsonSchema::RouseModeCV, schema);
  validator.Parse(schema, path);

  // Validate inputs.
  validator.Validate(json, path);
  if(validator.HasErrors())
    throw BuildException(validator.GetErrors());

  std::vector<Label> groups;
  for (auto& group : json["groups"])
  {
    groups.push_back({});
    for(auto& id : group)
      groups.back().push_back(id.asInt());
  }

  auto mode = json.get("mode",0).asInt();
  return new RouseModeCV( groups, mode);
}
RouseModeCV.h
```

Step 5. Make it buildable



Finally, we must ensure that the CV can be built in `CollectiveVariables.cpp`!

```
#include "CollectiveVariable.h"
#include "AngleCV.h"
#include "BoxVolumeCV.h"
#include "CoordinationNumberCV.h"
#include "GyrationTensorCV.h"
#include "ParticleCoordinateCV.h"
#include "ParticlePositionCV.h"
#include "ParticleSeparationCV.h"
#include "RouseModeCV.h"
#include "TorsionalCV.h"
#include "json/json.h"
#include <stdexcept>

namespace SSAGES
{
    CollectiveVariable* CollectiveVariable::BuildCV(const Json::Value &json, const std::string& path)
    {
        // Get move type.
        auto type = json.get("type", "none").asString();

        if(type == "Angle")
            return AngleCV::Build(json, path);
        else if(type == "BoxVolume")
            return BoxVolumeCV::Build(json, path);
        else if(type == "CoordinationNumber")
            return CoordinationNumberCV::Build(json, path);
        else if(type == "GyrationTensor")
            return GyrationTensorCV::Build(json, path);
        else if(type == "ParticleCoordinate")
            return ParticleCoordinateCV::Build(json, path);
        else if(type == "ParticlePosition")
            return ParticlePositionCV::Build(json, path);
        else if(type == "ParticleSeparation")
            return ParticleSeparationCV::Build(json, path);
        else if(type == "Torsional")
            return TorsionalCV::Build(json, path);
        else
            throw std::invalid_argument(path + ": Unknown CV type specified.");
    }
}
```

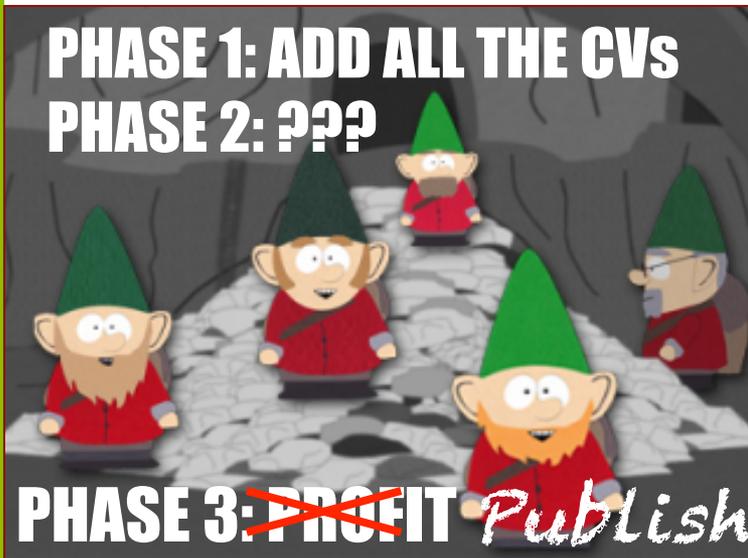
CollectiveVariable.cpp

These conditional statements handle the building of each CV; just add one!



Summary

A new CV can be added by creating a new class that derives from the `CollectiveVariable` base class in SSAGES



Step 6. Add a unit test

- good practice & self-contained check that all is well
- See `./test/unit_tests` for examples

Expanding on Phase I

Step 1. Formulate the CV

- pen and paper portion
- how will it be used?/what is needed?

Step 2. Begin writing `CoolThingCV.h`

- must add a file for this!
- bulk of the effort goes here
- easiest to start with an existing CV

Step 3. Craft your JSON schema

- must add `coolthing.cv.json`
- easy based on previous steps

Step 4. Finish writing `CoolThingCV.h`

- easy now based on Step 3

Step 5. Make it buildable

- just edit `CollectiveVariable.cpp`
- very trivial modifications