IGM OVERVIEW: INPUT, OUTPUT, STRUCTURE

The IGM (Integrative Genome Modeling) platform was designed in order to computationally generate a population of individual genome structures which overall recapitulate the features of genomic experimental data that are used as an input.
The current implementation (Jan 2021) handles diploid full genomes up to 200 kb resolution, and successfully integrates bulk Hi-C, laminB1 DamID, 3D FISH imaging and SPRITE data, within an idealized nuclear envelope. In addition, structures can be generated within a realistic irregularly-shaped nucleus with one or more nucleoli.
We refer to the pivotal paper by Boninsegna et al. for the technical details.

The main ingredients of the protocol are:

- **GENOME MODEL**: the genome is modeled as a simple polymer (string of beads), each bead is an effective particle which represents a given number of loci, depending on the segmentation/resolution chosen.

- **EXPERIMENTAL DATA**: each source of experimental data (e.g., Hi-C, lamina DamID, SPRITE, FISH) is translated into a spatial restraint between the beads making up the polymer (mostly spatial proximity constraints), and is modeled by a data-dependent force field. The information in the data is provided in separate files, according to specific format guidelines: in particular, the haploid loci annotation is used (distinguishing between different copies occurs inside the algorithm)

- **OPTIMIZATION**: restraints are gradually incorporated in the model using a step-wise iterative procedure, via s sequence of A/M iterations. Ideally, only a (usually the vast majority) fraction of all the restraints can be satisfied in a structure or across a population; IGM keeps track of this and uses that as a criterion to establish if the optimization went to completion or another run is in order.

Such a platform has two key implementation features:
- **PYTHON**: the full platform is hard-coded using Python 3.X
- **C++**: structure optimization requires running MD, and using the LAMMPS platform is versatile and well integrated within the python environment.

In order to install the software and the Lammps dependencies, please follow the step-by-step instructions provided in the README.md.

Expected Input Files:

- Pipeline configuration: **igm-config.json (**This json config file should include all input parameters, data files and parameters);
- **Data files** (.hcs, .txt, etc.) should also be available in the correct folder(s);
- *Lammps* executable file path also needs to be double checked.

All the parameters of the model (e.g., which kinds of experimental data are being incorporated, the optimization parameters, the number of structures, the name of relevant files) are listed in a **json** file (which is similar to a python dictionary, which can be viewed and edited using any text editor such as VIM). Such a file can be created manually (by editing an existing template) or by using the igm graphic interface, that is available.

Some parameters (such as the molecular dynamics optimization parameters) may require fine-tuning. Specifically, some entries require particular care: **genome** (i.e. ploidity) **segmentation** (i.e., degree of coarsening of the model), **population size** (i.e., number of structures to be generated), **optimization/kernel_opts/lammps_executable** (this path needs to point to the executable version of LAMMPS which was installed together with IGM in the first place, this you generally need to adjust only the first time running the code), **parameters** (change names to taste), **parallel/controller** (change this to ipyparallel (or serial) if parallel calculation (or simple test on a single core).

A comprehensive list of all the configuration file options (and their brief description) can be found on the GitHub @ igm/igm/core/defaults/config_schema.json.

Expected Output Files:

- Final output structures: **igm-model.hss** (default), or specified in **optimization/structure_output** in json configure file. This file can be read in using alabtools.HssFile("igm-model.hss",'r'). The coordinates are stored in 3D array of size [nbeads, nstruct, 3]. igm-model.hss.T is transposed, with dimensions: [nstruct, nbeads, 3]. Such a file also details the violations (satisfied and violated restraints, broken down into the different categories)

- If **optimization/keep_intermediate_structures** is set to true, there will be .hss files for each optimization step named like **igm-model.hss.sigma_#_iter_#.#######hash_value####.hss**

- Runtime log: **igm.log** file embedded with color code. Use *cat* to see colors in terminal.

- Evaluation folder: if **restraints/Hi-C/run_evaluation_step** is set to true, Hi-C evaluation step is executed to plot matrix comparison plots for each chromosome. If this step is run, it is recommended that multiple cores are requested on the same node (multi-thread),

- **Temp files** (regularly removed, at the end of each step): in folder **tmp**/

tmp/actdist/: default for Hi-C restraints temp files (activation distance file)
tmp/opt/: default for optimization lammps files ('lam', 'input', 'log', lammpstrj') and optimized structures ('hms)
tmp/damid_actdist/: default for damid distance files.
tmp/sprite_assign/: default for sprite cluster assignment files.
tmp/fish_assign/: default for fish distance distrib assignment files.

The tmp folder is where the assignment files are stored, the lammps optimization files are written and the .hms single structure files at the end of an M step are temporarily written, before they are concatenated into an hss file.

Running scheme

Running IGM is straightforward by typing:

`igm-run igm_config.json.`

The main script is `igm_run.sh` whose structure reads (igm_run.sh, which is in the bin folder, calls all the python scripts in the igm folder):

```
-------------------------------------------------------------------------
import igm (among others)

def run_pipeline(config):
      …….

# start the run as a subprocess
print('[CONTROL] Starting IGM:')

P = multiprocessing.Process(target = run_pipeline, args=(cfgfile,))
p.start()
```

```
p.join()

# exiting calculation smoothly
print('[CONTROL] Exiting')
-------------------------------------------------------------------------
```

The `run_pipeline` function consists of the following steps (see schematic in Appendix):

1. igm-run creates hidden file .igm-pid.txt to create the process id. This file indicates that the igm pipeline is running.

2. **igm.Config()**: Configuration is read from igm-config.json[1]

3. **igm.Preprocess()**: Preprocess genome, index and allocate disk space for genome structures (basically calls utilities from alabtools)

4. Generate starting coordinates with random coordinates or from another hss file (=this could be the population from a previous run whose optimization we would like to extend).

5. If start with random coordinates, **igm.RandomInit()** is run and simulation with chain and volume confinement restraints **igm.RelaxInit()** is run.

6. **A/M steps** (sweeping over decreasing values of threshold/harnesses values):

   # ----

   1. Assignment steps:

   This step reads in structures (.hss file) optimized in the previous step, and calculates the new assignment criteria for the current step: compute latent variables for next iteration. For example, Hi-C restraints are assigned using distance threshold called activation distance, which needs to be calculated. Following steps are run in sequence if corresponding data is provided.

      a. **igm.ActivationDistanceStep()** for HiC restraints
      b. **igm.FishAssignmentStep()** for FISH restraints
      c. **igm.SpriteAssignmentStep()** for Sprite restraints
      d. **igm.DamidActivationDistanceStep()** for DamID restraints

   2. Modeling/Optimization step:

[1] An additional entry in the .json dictionary is appended, “runtime”. This entry is required and allocates all generated parameters and those needed to run the different steps; it is updated for each list of threshold values (thetas and laminaDAMID thetas) in the preamble to “ActivationDistanceStep.py’ and ‘DamidActivationDistanceStep.py’ files. Once a value of sigma is run, it is systematically removed from all the lists in the “runtime” entry (.pop attribute), and once all those lists are empty, then the algorithm has gone to completion.

igm.ModelingStep(): translate latent variables into spatial restraints that are applied then to the genome polymer using Lammps. Runs the script generation for lammps. Build model and effectively apply the restraints computed from the assignment steps. Those restraints are then translated and applied to the polymer via LAMMPS. Run simulated annealing/conjugate gradient simulation and collect files and violation scores for individual restraints.

3. Evaluation step (optional):

Contact matrices are generated (using multiple cores). Plot matrix comparison plots for each chromosome (**igm.HicEvaluationStep()**).

4. Assess restraint violations: each restraint type is associated with a violation score, which equals 0 is the restraint is satisfied, and is more and more positive the more violated the restraint is. At the end of an optimization run, all the violation scores are computed for each category of restraints, and the fraction of violations automatically determines if the loop shall continue or if the calculation is over. At the end of each A/M step, the violations are detailed, broken down into the different restraint categories that are applicable.

Done

Remark: checkpointing and restarting

A production run also creates a database file with extension **.sqlite** (filename can be edited in the configuration file) on-the-fly, which keeps track of the progress status of the calculation (checkpointing). If such a file from a previous run is in the main folder, a new calculation looks into it (first thing first) and picks up where the previous calculation left off. For instance, if the previous run exceeded walltime during a mapping step, this is recorded and written to the .sqlite file, and the new run starts from that point, writing to the log file that “step XXX is already completed, moving on”. The .sqlite file should be removed ONLY if a new fresh calculation is to be submitted. The job tracking is carried out in the **igm/igm/core/job_tracking.py.**

It is recommended that a job that crashed during the mapping step be restarted only after clearing the (‘.hms) temporary files from the attempted optimization.

Post-IGM Conformational Analysis

A standard zero-th order analysis procedure (achronym QC) has been coded and added to the IGM software in such a way that the output structure population (.hss) can be processed and the essential information is extracted and saved to a folder named **QC_igm-model**.
The analysis code can be found in **igm/bin/igm-report**[2] and takes as an input the hss filename. This code uses all the routines that are defined in **img/igm/report**, and heavily relies on the alabtools python module.

A number of subfolders are created (**violations, radius_of_gyration, shells, radials, damid, matrix_comparison, images**), each populated by the corresponding files. A report.html file is also output. As a comparison term, 5000 configurations generate approximately 200MB of analysis report. The bin/igm-report.sh file uses the analysis scripts in igm/report.

[2] It is recommended that the script be submitted as a regular job to Hoffman2, since memory requirements can be quite intensive.

DETAILS OF THE INPUT DATA

The particles modeling the domains at a given resolution are subjected to a variety of constitutive restraints, which ensure chain connectivity and chain existence.
Particles have a radius, so an excluded volume term is added to the system Hamiltonian; neighboring particles within the same chain (i.e., chromosome) are subjected to a connectivity restraint, particles are required to move only inside the volume encompassed by the nuclear envelope. Up to this point, such restraints are introduced to ensure that the model makes physical sense.
When the actual modeling part starts, then each kind of experimental data needs to be translated into an appropriate spatial energy term and figuring out which particles in whcih structures in the population have to be restrained is crucial for the modeling step to be successful; this whole step is called Assignment step, and the output is a set of individual Hamiltonians, each describing one single structure in the population; overall, they recapitulate the effects of the experimental data.
Each kind of experimental data to be incorporated in the population modeling needs to be preprocessed and written to a file, whose format and content is compatible with the way IGM was designed. Here we cover a few details. Just keep in mind that each restraint amounts to a specific spatial restraint.

-Hi-C ----------

The input data (input_matrix) consists of a (haploid) contact probability matrix, which is saved into a .hcs file, and is read in by IGM using the alabtools python package (see ActivationDistanceStep.py). The A(i,j) entry indicates the probability that a contact is formed between i and j locii; clearly, this depends on how a contact cutoff between locii is defined.

A (i,j) contact is enforced in a structure, by adding a proper harmonic restraint between i and j. An iterative procedure is employed, where multiple decreasing threshold probability values are swept, as detailed in the pioneering papers. Hi-C contacts with higher occurrence probabilities are enforced first, in order to allow harmonious and non-disruptive data integration.

The relevant part in the configuration file looks as follows:

```
==========
"Hi-C": {
        "input_matrix": "path_to_input/hff_200kb_probability.hcs",
        "intra_sigma_list": [
           1.0, 0.2,0.1,0.05, 0.02, 0.01, 0.008
```

```
        ],      list of threshold values for intra-chromosome contacts (to be iteratively enforced)
        "inter_sigma_list": [
            1.0,0.2,0.1, 0.05,0.02,0.01,0.008,0.005,0.002,0.001
        ],      list of threshold values for inter-chromosome contacts (to be iteratively enforced)
        "contact_range": 2.0,
        "contact_kspring": 1.0,
        "actdist_file": "actdist.h5",        file generated from ActivationStep.py
        "tmp_dir": "actdist",
        "keep_temporary_files": false,
        "run_evaluation_step": false
    },
================
```

The output file actdist_file is then used (and iteratively updated during each iteration) in order to effectively assign more and more restraints to the structures. Such a file has a compressed format and essentially looks like a list of lines such as:

```
i    j      d      p
```

E.g.; for (diploid) contact between locii *i* and *j*, the activation distance d_ij is given and the probability *p* for that contact to be formed.

------ *Damid* ----------

The input (input_profile) consists of a (haploid) lamina contact probability array, which is saved into a .txt file, and is read in by IGM in the DamidActivationStep.py part. The E[i] entry gives the probability that haploid locus i is close to the nuclear envelope.

A lamina contact is enforced in a structure by adding an energy term which is either spherical or ellipsoidal, depending on the envelope shape. Such an energy term equals one (in arbitrary units) when the bead is in contact with the envelope, and gets closer to zero, the innerer the position of the bead.

The relevant part in the configuration file looks as follows:

```
================
"DamID": {
        "Input_profile": "path_to_input/damid_contact_prob_input.txt",
        "sigma_list": [
            0.7, 0.5, 0.4, 0.2        list of threshold values of lamina contacts (to be iteratively
enforced)
```

```
        ],
        "contact_range": 0.05,
        "contact_kspring": 1.0,
        "damid_actdist_file": "actdist.h5",        file generated from DamidActivationStep.py
        "tmp_dir": "damid",
        "keep_temporary_files": false
    },
===============
```

The output file damid_actdist_file is then used in order to effectively assign the restraints to the structures. Such a file has a compressed format and essentially looks like:

i *d* *p*

E.g.; for (diploid) nuclear envelope contact for particle *i*, the activation distance *d_i* is given and the probability *p* for that contact to be formed.

- *Sprite ----------*

Sprite experimental data provides information about multi-body contacts between locii, which is expressed in terms of “clusters”. Clearly, more information is needed as to how clusters are defined and which distance thresholds are selected to define locus proximity.

The input data consists of a (haploid) cluster characterization, which is saved into a .h5 file (clusters), and is read in by IGM in the SpriteAssignment.py step.

- h5[‘data’] the indexes of all (haploid) locii making up the clusters are listed for each clusters, and all clusters are concatenated.
- h5[‘indptr’] contains the information about cleaving points identifying each cluster in ‘data’. An example may be beneficial to understanding this. Assume a chunk reads [0,3,13,21]: then, such a sequence of locus indexes indicating there are 3 clusters, [0,2], [3, 12] and [13,21][3]. Please note that the length of the list equals the number of clusters (plus one), and the sum of all the cluster sizes equals the length of the ‘data’ list. This is a good consistency check to make sure the input file is not broken.

A ‘cluster’ contact is enforced by introducing a dummy particle located at the geometric center of the cluster, and then by applying harmonic restraints between it and each locus assigned to that cluster (think about this as many springs originating from the cluster centroid and reaching out to each locus).

The relevant part in the configuration file looks as follows:

[3] The locii assigned to the first cluster are then data[0], data[1], data[2]; locii assigned to second cluster are then [data[3], …, data[12]], and so on and so forth.

==========

```
"sprite" : {
        "clusters": "/u/scratch/b/bonimba/SPRITE/HFF_SPRITE_input_inter_triplets.h5f",
        "volume_fraction_list": [0.001, 0.001, 0.005, 0.005, 0.005, 0.01, 0.01, 0.05],
        "radius_kt": 50.0,
        "assignment_file": "sprite_assignment.h5",
        "tmp_dir": "sprite_assign",
        "keep_temporary_files": false,
        "batch_size": 10,
        "kspring": 1.0
    },
```

==========

The output file assignment_file (produced by the assignment step) is then used to effectively assign the restraints to the structures in the population. Such a file is still h5 compressed, and the dictionary breaks down as follows:

- 'Assignment': array of length N (n_cluster), entry i-th given the index of structure in which cluster i-th is to activated
- 'Selected': same as 'data' in input file, but locus indexing is now diploid
- 'Indptr': same as 'indptr' in input file

Sprite assignment loops over volume fraction list (from smaller to larger, which correspoinds to larger to smaller cluster radius). The assignment is performed multiple times, and the specific assignment to structure(s) changes because the structures vary over the calculation. There is no probability here, the only criterion to establish if optimization went to completion is by looking at the number of violations.

Sprite assignment is implemented using a Gibbs distribution concept which uses the radius of gyration and a penalty term proportional to how many clusters have been assigned to a given structure already. Ideally, the more clusters are there already, the less likely it is that a new cluster be assigned to the same structure; that would be extremely frustrated.

----- 3D FISH ----------

The input (input_profile) consists of a set of cumulative distance distributions for radial distances (distance of a given probe to the nuclear center) or pairwise distances (between given haploid pairs making up the genome). The list of probes and pairs, and their associated probability distributions for (either maximal or minimal or both) distances are given in a h5py format file, that is read in by IGM in the FishAssignmentStep.py part.

T = h5py.File(input_fish)

Dict = {‘probes’ (nprobes), ‘pairs’ (npairs x 2), ‘rad_min, max’ (nprobes * nstruct), ‘pair_min,max’ (n_pairs x nstruct)}.

It is crucial that the number of sampled distances for each distribution equals the number of structures in the population; this may require preprocessing, especially if experimental FISH data is used: the number of distances making up the distribution will almost never equal the number of structures. Distances need to be expressed in nanometers. FISH restraints are then enforced by using upper and lower bound harmonic potentials acting between two particles (if binary FISH) or the one probe and the nuclear center (if radial FISH).

Each radial FISH data is coded in 3 restraints, each pairwise FISH is 5 restraints.

FISH restraints are incorporated in the configuration file as follows:

==========

```
"FISH" : {
        "input_fish" : "path_to_input/fish_distribution_input.hf5",
        "rtype" : "p",        type of FISH restraint (radial (r) , pair (p), min or max)
        "kspring" : 1.0,
        "tmp_dir" : "fish_actdist",
        "keep_temporary_files" : "true",
        "tol_list" : [100.0, 100.0, 100.0, 100.0, 50.0, 50.0, 50.0, 50.0, 25.0],    # tolerance values, errobars
        "fish_assignment_file": "fish_assignment.h5"    # assignment file
    },
```

==========

The output file assignment_file (produced by the assignment step) is then used to effectively assign the restraints to the structures in the population. Such a file is still h5 compressed, and the dictionary is analogous to the input file format. Here, probes and pairs are given in a diploid format, and the struct_id for assignment is specified, as a result of the assignment step.

DETAILS OF THE DIFFERENT ALGORITHMIC STEPS

A controller is a more general, higher hierarchical layer, which maps a serial function (task) to the workers: the actual way this is performed is conditional on the specific choice of controllers which is made in the JSON config file. Syntax will be different (check IGM/parallel)

Random Initialization Step

Configurations are randomly initialized within a sphere during the mapping step (which is managed by the controller onto the engines), the coordinates of each structure are stored in a

.hms file (an h5py file) in $TMP_DIR. A master file .hss (n_beads x n_struct x 3) is then created during the reduce step and is then “transposed” in the “repacking” step.

Relaxation Step

Structures that were randomly initialized in the previous steps are here relaxed using a SA/CG procedure using LAMMPS. While progress status bar progresses, a bunch of different files are created.
In TMP_DIR/OPT files ‘FILENAME_%d’ + str(struct_id) + **.log, .lam, .lammpstrj, .data.** When the calculation is over, the corresponding **relax_#.hms** and **relax_#.hms.ready**[4] files are created one level ahead, in TMP_DIR, and the temporary files in OPT are removed.

The .hms files contain the last frame from the relaxation step for a given configuration which are later also merged (reduce step) into a master file .hss and redirected to the log file to keep track. That information the software needs to figure out if an extra iteration is needed, for the same set of parameters.

Reduce step: the master file .hss is opened, and the “relaxed” coordinates from each .hms files are copied into the corresponding structure slice in the .hss file. The master file is closed and flipped (repacking).

Multiprocesses: parallel to igm_run there is a “poller process” going on which constantly checks if the optimization is over and the ‘.ready’ is created, so that the .hss file can be updated on the fly, without having to wait for ALL the relaxation processes to be over.
Please note that this occurs at the controller level, the engines are sleeping as of now...each of them communicates back the information by creating the corresponding .hms file.

It also seems like the tqmd progress bar stalls if configuration A has not been optimized, even though A+1, A+2, …. have been. This may need to be adapted. As of now, the POLL_INTERVAL parameter was increased to 10, this way the actual calculations are not constantly interrupted by the polling function continuously checking .ready files.
One thing to be particularly careful about is when the modeling step is interrupted, for whatever reason. If temporary files .log, .lam, etc are in the TMP_DIR/OPT folder, but no calculation went to completion (such that the corresponding .hms files are not in TMP_DIR), then restarting would automatically skip those configurations and move on to the next ones, so it is best to remove those files and restart from the modeling step.

Activation Distance Step

In TMP_DIR, a ACTDIST folder is created and inside it files of type **#.in.npy** are created. Each file contains a numpy array (N x 4), N = number of structures. A bunch of **#.out.npy** files are

[4] .ready files are empty and are used as signals: if those are there, then coordinates are available for updating the .hss master file.

then created and then merged together into a **actdist.hdf5** file. There is a mapping-reducing sequence here too, but it seems very fast (look into igm.log).

Damid Activation Distance Step

In TMP_DIR a DAMIDACTDIST folder is created and almost instantaneously a file **damid_actdist.hdf5** is created. There is a mapping-reducing sequence here too, but it seems very fast (look into igm.log for details).

Now, the actual optimization step starts, and things are equivalent to the mapping step detailed in the Relaxation Step paragraph. The files created in TMP_DIR are now called **mstep_#.hms** and **mstep_#.hms.ready**, and contain the optimized coordinates upon relaxing the hss configurations with restrained computed in the (Damid)Activationstep. What is interesting here is that the file **igm-model.hss.T** is being edited on the fly while the mapping step proceeds. That is probably due to mapping reading in initial configurations from the master hss file.

A temporary hss file summarizing the results for this setup (in terms of activation and damid activation thresholds) is then produced after reducing and repacking. Then the next iteration starts. Before repacking step, the code prints a summary of the thermodynamic properties and violations during the optimization.

The next iteration sees the actdist.hdf5 file touched and then the new #.in.npy(s) files are produced in the TMP_DIR/ACT_DIST directory. The number of .npy files changes from iteration to iteration, since the number of restraints also increases by lowering the probability thresholds. The the files #.out.npy(s) are generated, everything is purged and a file ‘actdist.hdf5.sigma_0.2000.iter_0’ is created. Now, only this file and the original ‘actdist.hdf5’ are in the folder.

Then the file in DAMIDACTDIST is updated/touched. Then, the minimization of all the structures can start.
The logger info (‘Read 860 probabilities from last step‘) appears in the log file, right after DamidActivationDistanceStep (cut=50.00%, iter=0) - starting

And immediately before
DamidActivationDistanceStep (cut=50.00%, iter=0) - mapping

NB: the igm/igm/steps/ModelingStep.py poller was adapted in such a way that attribute hss.set_coordinates() is only called once (the code works by replacing the i-th slice in the master matrix referring to the coordinates of the i-th configuration). Now the reducing step is lightning fast :) Look into ‘Relaxation’ for the details of the mapping step.

Modeling Step

The chromatin model is defined by introducing degrees of freedom (particles) and interactions (mainly structural integrity restraints, such as connectivity and excluded volume, and IGM restraints from Hi-C or DAMID). The model is them optimized the same way as in RelaxInit step,

The main difference is that the output .hms file also contains the optimization statistics, especially the number of restraint violations, classified by restraint kind. All the statistics are collectively summarized and saved into the hss population file, recapitulated in the logger file and are processed by the igm-report script to produce a preliminary analysis.

Codes and Wrappers

---------- igm/igm/tasks/modeling_step.py

Serial function to be mapped in parallel, the controller is general and can be specified in the configuration file. This file supersedes everything else, defines the most general structure for managing task distributions between controller and workers, which can be ipyparallel or dask or serial (see Controller_class dictionary opening the file, each entry is defined, especially the map attribute, in the corresponding files...see below).

Main structure:

```
Controller_class = {
      u'serial' : SerialController,
      u'ipyparallel' : AdvancedIppController,
      u'ipyparallel_basic' : BasicIppController,}

cfg = Config(cfg_file)
def modeling_task(struct_id, cfg_file)
      ….

def modeling_step(model, cfg):

      serial_function = partial(modeling_task, cfg_file)

      pctype = cfg['parallel_controller']
      popts = cfg['parallel_controller_options']
      controller = controller_class[pctype](**pcopts)
      argument_list = list(range(n_struct))

      controller.map(serial_function, argument_list)
```

---------- **igm/igm/parallel/ipyparallel_controller.py**

This is the part of the code where the main code “connects” to the ipyparallel environment. The key is the map function (lines 66 and following), where the `load_balanced_view()` and `map_async(IppFunctionWrapper(parallel_task, ...))` are called, where the `parallel_task` (usually the lammps relaxation) is submitted to the different workers. If the ipyparallel environment is not run, then a “remote failure” message appears.

---------- **igm/igm/parallel/async_file_operations.py**

This is the file where the polling function is defined: when one calculation is completed in any engine (the polling checks that actively on the fly), then one operation is carried out, usually it is updating the coordinates of the corresponding structure index in the hss matrix. The key ingredient is the callback parameter, which indicates the operation to be performed within the polling function. The enumerate function is then used together with the progress bar module to estimate the remaining time to completion.

---------- **igm/igm/core/step.py**

This is the unified template for any step included in the overall IGM pipeline. Any computation is a IGM pipeline should be a subclass of this class. The .run() attribute executes, in order:

```
setup()
before_map()
map()
before_reduce()
reduce()
cleanup()
```

If the full step already went to completion, then the `skip()` member function is called and the code moves on to the next step.

The name() function returns the name of the class. It is mainly used for logging and visualization purposes. Overloading it to add information helps detail the output.

---------- **setup()**:

The role of setup is to prepare folders, files, data for the step. In particular, it needs to set the `self.argument_list` special variable, which contains the argument to be mapped in parallel. It may also modify the self.tmp_dir or modify the "runtime" section of the configuration object, stored in `self.cfg`. Also, automatic temporary files deletion can be specified, see the `Special member variables` section below. Note that this function will be called also on restart runs, if the whole step was not completed.

---------- **before_map()**:

This function is executed just before mapping. It is intended o setup resources which are needed ONLY in case of mapping. in a restart run, it is skipped if mapping has already been completed.

---------- **task(arg, cfg, tmp_dir)**:

The static `task` method is executed on parallel workers. Each run of the task method gets as first parameter one of the arguments in `argument_list`. The second and third parameters are the same for each run and are the configuration object and the temporary directory. Note that any value returned by task is ignored because of a design choice. Any data processed and further needed should be stored on an accessible location, like a shared file system or server.

[In general, large amounts of data or complex objects can be difficult to save properly. This means additional complexity in the restart design if something fails, which is better handled by delegating the data transport/storage to each specific case.]

---------- **before_reduce()**:

This function is executed just before reducing. It is intended to setup resources which are needed only for the reduce step, for example initializing in-memory resources if the mapping step is skipped. In a restart run, it is skipped if reduce() has already been completed.

---------- **reduce()**:

This is executed only on the master node after mapping, and it is intended for either serial steps which do not require a mapping, or to collect, reduce, and atomically write the results.

---------- **cleanup()**:

This is the last step, intended to cleanup temporary files and resources, and update the runtime environment. In general, overriding this method is unnecessary and not suggested.

---------- **skip()**:

When the step is skipped in a restart run, this function is called. It is intended for possibly setting runtime variables

Special Member Variables

---

cfg : Config
    configuration object
db : StepDB
    database object
argument_list : list
    arguments to be mapped in parallel
tmp_dir : str
    path of the directory where temporary files will be saved
tmp_extensions : list
    list of extensions of temporary files. All files with the
    listed extensions in tmp_dir are removed at the cleanup()
    call if `keep temporary files` is True
keep_temporary_files: bool
    if True, it deletes temporary files during cleanup
uid : str
    a unique string identifying the current step

The **run()** attribute actually runs the calculation and contains the part where the database file is checked (line 240) and the completed parts are skipped.

Also, this file contains the class StructGenStep(step) which is called in the Initialization Step.

This is the part of code where the computing environment is mentioned for the first time.

```
from ..parallel import Controller

self.controller = Controller(cfg)

serial_function = partial(self.__class__.task,
                                        cfg=self.cfg,
                                        tmp_dir=self.tmp_dir)
self.controller.map(serial_function, self.argument_list)
```

Each parallel controller comes with a “map” attribute with enables to distribute the “serial function” task to the workers, each of which produces one or more files. Controllers that involve parallel computing are “parallel, ipyparallel, slurm”. “Serial” greatly simplifies everything,

The controller files are in /igm/parallel folder. In particular, the __init__.py file reads:

```
from __future__ import division, print_function

from .parallel_controller import SerialController
from .ipyparallel_controller import BasicIppController
from .slurm_controller import SlurmController
controller_class = {
```

```
    "serial" : SerialController,
    "slurm" : SlurmController,
    "ipyparallel" : BasicIppController,
    "ipyparallel_basic" : BasicIppController,
}

def Controller(cfg):
    parallel_cfg = cfg.get("parallel", dict())
    pctype = parallel_cfg.get("controller", "ipyparallel")
    pcopts = parallel_cfg.get("controller_options", dict()).get(pctype,
dict())
    return controller_class[pctype](**pcopts)
```

---------- **igm/igm/model/kernel/lammps.py**

This is the reference script where data is collected, scripts are prepared and submitted, and the actual optimization of the structures is performed (Modeling Step).

```
def create_lammps_data(model, user_args)

def create_lammps_script(model, user_args)

def optimize(model, cfg):

      m = LammpsModel(model)

      # prepare data and scripts)
      create_lammps_data(m, run_opts)
      create_lammps_script(m , run_opts)

      # run the lammps minimization
      with open(script_fname, 'r') as lamfile:

            proc = Popen([lammps_executable, '-log', log_fname],
                         stdin = lamfile,
                         stdout = PIPE, stderr = PIPE)
            output, error = proc.communicate()

      ...
```

The key quantity is the `igm.model` object, which is the abstract model object for simulation optimization (define particels, define interactions, define restraints…)

It is crucial to stop and realize that the modeling step is carried out on a bunch of different hierarchical levels. First, the igm “Model” class is created, which was hard-coded, together with model Particles and Forces. All the necessary extra fictional particles and interactions (according to the Assignment Step) are here listed and one can check exactly how they are applied and to which sets of particles.
The information in “Model” need to be translated into a LAMMPS compatible fashion, and here is where the “kernel” folder comes into play. First, an ad hoc “Lammps_model” class is created, which is a fresher reformulation of the igm Model class. Then, “Lammps_model” is used as an input to create the actual and proper LAMMPS input data (.data) and LAMMPs run script (.lam). Then, LAMMPS can be successfully run.

Add new restraints to the pipeline

Adding a new type of restraint from experimental data requires particular care, since many files making up the code require appropriate editing accordingly. In particular:

- Configuration file, since new parameters need to be added (this requires editing of the GUI to allow implementation);
- A new assignment (A) step might be needed. A new class inherited from Step needs to be created, overloading setup, task, reduce, clean-up functions accordingly;
- A new **restraint** class (inherited from Restraint) needs to be created, to add restraint to the structures. Please keep in mind that if novel types of interaction forces are needed, once has to create a new Force class and the code for the kernel to create it (aka, Lammps fix); an appropriate violation score shall also be introduced.
- The modeling step (M) needs to be updated to include the new restraint (if specified in the configuration JSON file)
- Finally, the igm-run script needs to include the assignment step, update checks on satisfactory completion, eventually update the runtime data.

To test an Assignment step, one can run the following:

```
from igm import Config
from igm.steps import NewAssignmentStep
cfg = Config(x)
t   = NewAssignmentStep(jsonfile.json)
t.run()
```

And make sure that the controller is turned into ‘serial’ in the JSON file.

RUN IGM ON UCLA HOFFMAN2 HCP CLUSTER

UCLA Hoffman2 cluster uses a Univa Sun Grid Engine job scheduler, whose syntax is very close in spirit to that of PBS schedulers, albeit with crucial differences. A good starting point to submit jobs onto the cluster would be the following .sh templates.

Running IGM is a multi-step submission procedure:

- (If applicable) Remove temporary or old files (from previous productions) from folder(s)
- Double check configuration .json file

- Create ipyparallel environment using **ipcluster**, controller + N engines: (it is recommended that the controller and the engines be started separately, this way the status of the connection/environment can be checked on-the-fly by looking into the output/error files)

- Submit serial **igm_run.sh** job (actual calculation)

It is crucial that the jobs start serially:

Controller ---> Engines ---> Igm

In the following, template SGE scripts are provided to submit the different steps to the cluster.

If the HCP resources available uses a different job scheduler, the run script syntax needs to be adapted adequately.

---------- TEMPLATE: INITIALIZE ipyparallel CONTROLLER ----------

```
#!/bin/bash
```

```
#--- memory of the controller -----#
SMEM = 4G
#--- walltime (not to exceed 24hrs) ------- #
WTIME = 20:00:00

# --- email and flags for communication ----#
#$ -M XXXX@g.ucla.edu
#$ -m ea
#$ -N ipycontroller

#--- allocate resources for controller ---#
#$ -l h_data=${SMEM}
#$ -l h_rt=${WTIME}
#$ -l highp
#$ -cwd

# --- output files ----#
#$ -o out_controller
#$ -e err_controller

#--- copy full environment (modules) onto computing node --- #
#$ -V
export PATH="$PATH"
ulimit -s 8192

cd $SGE_O_WORKDIR

#--- extract ip address to start controller ---#
myip=\$(getent hosts \$(hostname) | awk '{print \$1}')

# --- launch ipcontroller which is being monitored ----#
MONITOR=$(command -v monitor_process)
if [[ ! -z "$MONITOR" ]]; then
    monitor_process --wtype S ipcontroller --nodb --ip=\$myip
else
    ipcontroller --nodb --ip=\$myip
fi

Echo “Controller submitted!”
```

---------- TEMPLATE: ALLOCATE ipyparallel ENGINES ----------

```
#!/bin/bash

#$ -M XXXX@g.ucla.edu
#$ -m ea
#$ -N ipycluster
#$ -l h_data=1G
#$ -l h_rt=20:00:00
```

```
#$ -l highp
#$ -cwd
#$ -o out_engines
#$ -e err_engines
#$ -V

#--- create parallel environment (pe): allow for processors to be allocated on different physical nodes (dc*)...this is mandatory, since no node physically contain more than ~30 cores
#$ -pe dc* ${NTASKS}

export PATH="$PATH"
ulimit -s 8192
cd $SGE_O_WORKDIR

# --- start engines (aka SRUN --n=$TASKS, in SLURM syntax) simultaneously ---- #
MONITOR=$(command -v monitor_process)
if [[ ! -z "$MONITOR" ]]; then
    mpirun --n=${NTASKS} monitor_process --wtype W ipengine
else
    mpirun --n=${NTASKS} ipengine
fi

echo "should have submitted $NTASKS engines on the top of the controller!"
```

---------- TEMPLATE: SUBMIT IGM COMPUTATION ----------

```
#!/bin/bash

#$ -M XXX@g.ucla.edu
#$ -m ea
#$ -N run_igm
#$ -o out_igm
#$ -e err_igm
#$ -V
#$ -cwd

# --- resources  ---- #
#$ -l h_data=20G
#$ -l h_rt=23:59:00
#$ -pe shared 2

# --- define parallel processes  ---- #
Export NUM_OMP_THREADS = 2

Echo “submit actual IGM calculation.”
```

```
# --- execute job and redirect output  ---- #
igm-run igm-config.json > igm_output.txt
```

NB: igm_run is strictly serial, it is controller responsibility to efficiently parallelize the calculation (the main idea is that each structure in the population can be optimized separately from all the others, which is a trivially parallel trask). However, there is multiprocess step (polling step), so using 2 processors which share memory (open_omp style) may improve communication and not have competing processes on the same CPU.

Please note that file '**err_igm**' is updated on the fly and details the steps that are eventually summarized in the igm.log file. For example, the progress status bar for the mapping and reducing steps with the completion percentage, can be found there.

Remark: the ipyparallel environment

1. The controller and engines can be started at once using the integrated command

   ```
   ipcluster  start --n=${TASKS} --ip=$(hostname -i | awk '{print $0}')
   ```

   However, keeping track of the status of the controller/engines is more involved.

2. Make sure the different output/error files are named differently, in order to keep track of the different steps.

3. Upon creating the ipcluster environment, one can check the engine status from terminal by opening iPython and interactively typing:

```
# ---- import ipyparallel
from ipyparallel import Client

# ----- call the Client
r = Client()
```

```
# ---- extract indexes for different engines (this is a good point to double check the number of engines activated)
r.ids

# ----- import socket on ALL engines
with r[:].sync_imports(): import socket

# ----- print hostname of ALL engines (this is a good point to check the physical allocation of the cores on the supercomputer)
%px print(socket.gethostname())
```

---

1. **lgm-run** uses a similar protocol to connect to the ipcluster environment and distribute the tasks. This happens in the task file in parallel folder.

```
import ipyparallel as ip
Rc = ip.Client( -- profile --)          Use the local ip address

#--- Let the controller manage the distribution of tasks among the different engines
V = rc.load_balanced_view()

#--- The same function is run on ALL engines
Res = v.map_sync(f, *args)

#--- Check status of the different tasks on each core of the network
rcl.queue_status()
```

Freely experimenting with IGM

Assume that a user wants to experiment with the IGM code, without having to push/pull changes from the main repository. Please follow:

1. Fork the IGM repository onto one's own GitHub account
2. Enter target folder from the terminal
3. Type **pip install -e Forked-address** ./
4. (Just for the sake of doing that, maybe reinstall the serial version of lammps)

The "editable" version will be compiled automatically upon each edit, locally. If the ALBERLAB version was previously installed, it will also be purged. This way, import igm will automatically call the editable version.

Also, if you want something to be printed into the log file, please follow the syntax

```
from ..utils.log import logger
logger.info('------stuff to be printed ----')
```

Missing how to merge/pus/pull from/to forked repository

IGM INTERFACING WITH LAMMPS

Let us assume the Assignment step has been completed and that a detailed list of interactions is available which we would like to add to the force field and, eventually, simulate. Classic interactions may include excluded volume restraints, nuclear envelope restraints and chain connectivity (nearest neighbors) restraints; also, we should have pairwise interactions between beads that are far apart along the primary sequence (e.g., from Hi-C or SPRITE data) and envelope affinity interactions (e.g.; from lamina DAMID data). This section details how the input files are created to feed to the LAMMPS kernel for Conjugate Gradient (CG)/Simulated Annealing energy minimization (Modeling Step).

Upon the Assignment Step, we have a `igm.model` object: `particle` and `forces` attributes can be used to check the number of particles and interactions the model has been endowed with. As an example, let us assume we set out to optimize a Hi-C + DamID + SPRITE model; we then expect a variety of particles and interactions.

```
In [13]: for i in range(29837,29865):
             print(i, '     ', model.particles[i])
```

```
29837      (1308.2152099609375 -869.3126831054688 -3437.713623046875, 118.54745483398438):NORMAL
29838      (0.0 0.0 0.0, 0.0):DUMMY_STATIC
29839      (0.0 0.0 0.0, 0.0):DUMMY_STATIC
29840      (-191.1259765625 768.7637939453125 165.761474609375, 0.0):DUMMY_DYNAMIC
29841      (-19.581884384155273 1754.0491943359375 1513.7618408203125, 0.0):DUMMY_DYNAMIC
29842      (1889.830322265625 -2132.17626953125 1007.5888061523438, 0.0):DUMMY_DYNAMIC
29843      (311.8243713378906 3498.357421875 367.25762939453125, 0.0):DUMMY_DYNAMIC
```

The model started out with 29838 beads (‘NORMAL’ attribute), then we have two extra DUMMY_STATIC particles that are fixed in the geometric center of the nucleus and are used to apply both envelope restraints (all beads are forced to occupy the nuclear volume, some of them are forced to occupy a peripheral position due to DamID). Also, we have a bunch of extra DUMMY_DYNAMIC particles representing the centroids for the SPRITE clusters (those can move!).

```
some of the forces added to the model...
0      FORCE: EXCLUDED_VOLUME (NATOMS: 29838)
1      FORCE: ENVELOPE
2      FORCE: HARMONIC_UPPER_BOUND 0 1 0
3      FORCE: HARMONIC_UPPER_BOUND 1 2 0
4      FORCE: HARMONIC_UPPER_BOUND 2 3 0


29790      FORCE: HARMONIC_UPPER_BOUND 29833 29834 0 sep =  1
29791      FORCE: HARMONIC_UPPER_BOUND 29834 29835 0 sep =  1
29792      FORCE: HARMONIC_UPPER_BOUND 29835 29836 0 sep =  1
29793      FORCE: HARMONIC_UPPER_BOUND 29836 29837 0 sep =  1
29794      FORCE: HARMONIC_UPPER_BOUND 15453 15455 1 sep =  2
29795      FORCE: HARMONIC_UPPER_BOUND 15457 15459 1 sep =  2
```

Accordingly, the list of model interactions is given in the order they were added to the model (excluded volume first, envelope restraints then, ‘harmonic_upper_bound’ for nearest neighbors and the Hi-C contacts, and so on and so forth).
This model does contain all the information needed by using an IGM compatible syntax. However, it is crucial to move on and make sure that such interactions are coded correctly within the force field that we would like LAMMPS to optimize.

This is a multi-step procedure, and the relevant pieces of code can be found in igm/igm/model/kernel. In particular:

- Lammps_model.py: this code bridges IGM and LAMMPS syntax, it is an auxiliary step. All the information contained in img.model is processed and a lammps_model object is created, which is going to be crucial to produce the input scripts to feed to LAMMPS to run the minimization;
- Lammps_io.py: self-explanatory, used to extract simulation relevant information from a LAMMPS production;
- Lammps.py: this is the key step; the .data and .lam files are generated, from the lammps_model object. “.data” file contains the list of atoms, of bonds and the interaction coefficients, and will be read as an input by the “.lam “ script, which contains information about the integrator, about the steps the simulated annealing needs to be

broken down into, etc. Eventually, LAMMPS is run using the '.lam' script as a template and the '.data' file as actual input. Output of the simulation is given in the `.log` and `.lammpstrj` extension files. The .lammpstrj file in particular contains the trajectory of the structure during the simulated annealing/conjugate gradient pipeline. The coordinates from last frame are then stored and indicate the "optimized structure".

Please note that, up to this point, all the information contained in the `restraints` files (violation score, etc) have not been used yet. It'll come in handy later on.

Lammps simulation script XXX.lam

The syntax is quite involved, but is (more or less) well documented online. The key point that has to be clear in mind is the concept of **fixes**. In LAMMPS, any operation computed during time stepping that alters some property of the system is called a fix. Essentially everything that happens during a simulation besides force computation, neighbor list construction and output, is a "fix". This includes time integration itself and diagnostics (say, compute the diffusion coefficient). A "fix" is also the natural way to modify LAMMPS, by adding additional features such as fancy interactions.

While classic MD interactions such as bonds, LJ are already built in the LAMMPS interface, novel forces such as envelope, volumetric and other restraints need to be hard-coded, and fixes help accomplish exactly that. Fixes are implemented according to an agreed upon format: one needs to create a .h and a .cpp file; the .cpp file imports the .h file and actually performs the computation. A good reference for designing one-body term fixes can be the `fix_gravity.h and fix_gravity.cpp` files available with the LAMMPS distribution.

The .lam script contains the syntax for performing a calculation, standard and user-defined. The main commands are indicated as follows:

```
#---------- EXAMPLE OF A WORKING LAMMPS SIMULATION SCRIPT -------#

#specify physical units to be used in simulation
Units        lj
# what style of atoms are used; here, BEAD spring polymer
Atom_style         bond
# set format LAMMPS uses to compute bond interactions (nearest neighbors)
Bond_style      hybrid      harmonic_upper_bond     harmonic_lower_bond
# define the simulation box feature
Boundary           s s s
# we are not re-scaling any of the LJ/coulombic interaction coeff
Special_bonds    lj/coul 1.0 1.0 1.0

????
```

```
Fix userprop all property_atom i_chainid d_radius

# pairs are the non-adjacent interactions.237 is the cutoff, this is used as excluded volume and the cutoff is defined as twice the particle radius (approx 2 * 118... ). The amplitude is defined in the simulation protocol part below
Pair_style       soft     237

# give input file (listing atoms and bonds), enforce “User” entries to be read
Read_data    filename     fix userprop NULL User

# define groups of atoms, ‘nonfixed’ is the set of actual physical beads
Group dummy type 2
Fix 1 dummy setforce 0.0 0.0 0.0     # initialize forces for dummies
Group nonfixed type 1

# turn off nearest neighbor interactions for all DUMMY particles
Neigh modify exclude group dummy all
Neighbor 118.5   bin                # algorithm used to build neighbor list,  ‘118.5’ is the skin parameter,  which is a distance beyond force cutoff to consider to run the neighbor search
Neigh_modify every 1 check yes     # compute neighbors at each time step
Neigh_modify one 6000 page 12000    # max number of neighbors

# use NVE integration on the nonfixed group particles
Fix integrator nonfixed nve/limit 1000.0

# define groups of beads to which the envelope force needs to be applied
Group envgrp0 id 1 2 3 … 29838
Group envgrp1 id 4 8 1000 1200 ….

# define simulation timestep
Timestep 0.25

# output production every 40k steps (multiple lines reading “x,y,z,fx,fy,fz”)
Dump crd_dump all custom 40000  out_file id type x y z fx fy fz

#--- SIMULATION: simulated annealing stuff (see below)

Min_style cg  # use conjugate gradient

# minimization until one of the conditions is satisfied (etol, ftol, maxiter, maxeval)
Minimize  0.0001 1e-06 500 500
Info time
```

Now, the different iterations of conjugate gradient are setup by using the runtime information available in the .JSON file `cfg['optimizer_options']['simulated_annealing_options']`. Each iteration consists in:

1. Properly rescale the “soft interaction” amplitude, the envelope “semiaxes” and “fixing” those into the simulation protocol;
2. Run a 200 steps NVE relaxation at a reference unitary temperature (in LJ units);
3. Run an N steps NVE relaxation at a temperature T_start, and then rescale temperature to T_end by rescaling particle velocities.
4. Print thermodynamic information using an appropriate style

Specifically, one iteration reads:

```
# Introduce the evprefactor variable
Variable evprefactor equal 0.5

# Change soft potential amplitude A to (A * ev_prefactor), making it harder and harder
Fix exVolAdapt0 all adapt 0 pair soft a * * v_ev_prefactor scale yes reset yes

# add fixes where the ellipsoidal force is introduced, and the energy (please note that in each iteration of CG, the semiaxes a, b, c are rescaled)
Fix envelope0 envgrp0 ellipsoidalenvelope a b c k
Fix_modify envelope0 energy yes
Fix envelope1 envgrp1 ellipsoidalenvelope a b c -k
Fix_modify envelope1 energy yes

#--- RUN SHORT EQUILIBRATION AT REF TEMPERATURE ----#
# generate consistent velocity distributions using random seed
Velocity nonfixed create 1.0 832827

# reset temperature of groups of atoms by explictely rescaling their velocities
Fix thermostat nonfixed temp/rescale 1 1.0 1.0 0.1 1

# Perform constant NVE updates of position and velocity for atoms in the group each timestep. A limit is imposed on the maximum distance an atom can move in one timestep
Fix integrator nonfixed nve/limit 10.0
Run 200

#--- RUN ACTUAL CALCULATION AND RESCALE THE TEMPERATURE ---#

Fix integrator nonfixed nve/limit 1000.0
Velocity nonfixed create 5000.0 832827
Fix thermostat nonfixed temp/rescale 1 5000.0 500.0 0.1 1

# define style and print thermodynamic information every x steps
Themo_style custom step temp epair ebond f_envelope0 f_envelope1
Thermo_modify norm no
Thermo 10000

# run for a number of steps
Run 5000
```

```
# turnoff fix, restore amplitude of soft interaction
Unfix exVolAdapt0
```

Lammps simulation script XXX.data

Here the syntax is sufficiently clear. A few things to details:

- `NUMBER OF ATOMS`: this includes both physical and fictional particles (if SPRITE data is used, then an additional dummy particle, e.g. centroid, is added for each cluster, if DAMID data is used, an addiitonal particle in the geometric center is added)

- `ATOM TYPES`: it is usually 1 (only phsycial particles), 2 (either dummy static or dynamic), or 3 (all particle types)

- `BOND TYPES`: this is tricky. There is a standard type for bonding between physical particles. One additional type is introduced for each SPRITE cluster type (two clusters are of different kinds if they differ in the number of atoms they comprise); i.e., the number of bond types is not directly related to the number of centroids added.

- `BOND_COEFF`: syntax is
  `bond_ID harmonic_upper_bond k_spring activation distance`

- `BONDS`: list of all the bonds (each SPRITE cluster is broken down as a sequence of pairwise bonds of each bead in the cluster with the cluster representative centroid), using the following syntax
  `Bond_number bond_ID particle_idx particle_idx`

- `PAIRIJ COEFF`: specify pair interaction coefficients (those which require the computation of a neighbor list) for all pair types. Ideally, if the number of atom types is 3, we would expect N(N+1)/2 lines here, one for each combination (N over 2) (combinatorial coefficient). Usually, only the physical-physical interaction is assigned coefficients which are different from 0.0, since other atom types are fictional (in the current formulation). Remember that PAIR_STYLE interactions represent the way restraints are added to the system, and are therefore harmonic. The coefficients are the elastic costant and the equilibrium position, respectively

- `USER`: adds a list of all the particles, the chain they belong to and the radius. Index 0 is used for fictional particles.

LAMMPS fixes

There are dozens of fix options already in LAMMPS: each class comes with a bunch of different methods which specify different options, for instance at what point during the timestep the fix is called. A fix is called within a LAMMPS input script .lam in the following way:

```
Fix   fix_id   group   fix_name   args
```

Where “fix_id” is the name given to the fix in the script, “group” is the atom/particle group name defined above, “fix_name” is the official fix name as from the .h file, “args” are possible arguments. For instance, the envelope restraining force is called using:

```
Fix   envelope0   envgrp1   ellipsoidalenvelope   a b c k
```

Now, the crucial feature is to calibrate the .h and .ccp files coding the fix you have in mind. As a general rule, the .cpp file imports the .h file, together with other fixes.

All fixes are derived from the Class fix and must have constructor with signature

FixMine(class LAMMPS *, int, char **)

Every fix must be registerd in LAMMPS by writing the following lines of code in the header before including guards.

```
#ifdef FIX_CLASS
FixStyle(name_of_your_fix_in_script, name_of_fix_class)
#else
```

This code allows LAMMPS to find your fix when it parses the input script. In additionally, the fix header must be included in the style “style_fix.h”.

Let’s take a look at the .h file for the envelope restraint:

```
/* -*- c++ -*- ----------------------------------------------------------
   LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
   http://lammps.sandia.gov, Sandia National Laboratories
   Steve Plimpton, sjplimp@sandia.gov

   Copyright (2003) Sandia Corporation.  Under the terms of Contract
   DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
   certain rights in this software.  This software is distributed under
   the GNU General Public License.

   See the README file in the top-level LAMMPS directory.
------------------------------------------------------------------------- */

#ifdef FIX_CLASS

FixStyle(ellipsoidalenvelope,FixEllipsoidalEnvelope)

#else

#ifndef LMP_FIX_ELLIPSOIDALENVELOPE_H
#define LMP_FIX_ELLIPSOIDALENVELOPE_H

#include "fix.h"

namespace LAMMPS_NS {

class FixEllipsoidalEnvelope : public Fix {
 public:
  FixEllipsoidalEnvelope(class LAMMPS *, int, char **);
  virtual ~FixEllipsoidalEnvelope();
  virtual double memory_usage();
  virtual void grow_arrays(int);
  virtual void copy_arrays(int, int, int);
  virtual void set_arrays(int);
  virtual int pack_exchange(int, double*);
  virtual int unpack_exchange(int, double*);
  int setmask();
  void setup(int);
  void min_setup(int);
  void post_force(int);
  void post_force_respa(int, int, int);
  void min_post_force(int);
  double compute_scalar();
  double compute_vector(int);

 private:
  double a, b, c, a2, b2, c2, kspring;
  double* radius;
  double** v2r;
  double etotal;
  double* ftotal;
};

}

#endif
#endif
```

A key method for a fix is the `setmask()` command, which the user employs to specify which other methods should be called during executions. The methods are called in predefined order

during the execution of the verlet algorithm, so keep this in mind. Fixes that perform time integration (nve, npt, nvt) implement initial_integrate() and final_integrate(); fixes that constrain forces only implement post_forces().

If a fix needs to store information for each atom that persists from timestep to timestep, it can manage that memory and migrate the info with the atoms as the move from processors to processors by implementing the `grow_arrys`, `copy_arrays`, `pack_exhange`, `unpack_exchange`. The lamina DamID restraint is to be applied only to a group of atoms, and indices need to be transferred, since they are messed up by computing the neighbor list. In contrast, Guido's prototype for a volumetric fix does not include such classes for information exchange since ALL particles are subjected to that restraint. The thermo method enables a fix to contribute values to thermodynamic output, as printed quantities and/or be summed to the potential energy of the system.

To touch base, let us look into the .cpp file.
Crucial here is the use of the atom.h class, which automatically points to the coordinates (x), velocities (v) and forces (f) of the simulation particles. These quantities are automatically contributed to the fix, there is no need to define them. Here we are computing the envelope restraining force, so we pass the semiaxes and the elastic constant as parameters. Scalars are handles trivially, arrays are allocated using the "atom" class, as shown in lines 55-56. In line 66, an option is introduced to make sure that the storing in memory of information is possible.

The key point where the calculation happens is the "post_force(int vflag)" method, where the different atom class quantities are imported, and regular local variables are introduced. Please note again that "x" and "f" are global, whereas "vr2, etotal, ftotal" were declared and initialized in the function header.

Then the "f" array is updated with the ellipsoidal force contribution (lines 202-204), so are the global parameters "etotal" and "ftotal". Again, "f" is not initialized anywhere here, just summed over: f already contains the bond/pair forces, we are just summing on the top the local restraint.

The other methods are quite standard, and are necessary to make sure the fix smoothly interfaces with LAMPPS.

- post_force(int vflag): after forces have been calculated in Verlet, add fix contribution
- compute_scalar(): return the energy
- compute_vector(in n): return the force for particle n-th.

The remaining sytax can be copied and adapted, the purpose is the same.

```
# allocate memory for vr2 array, atom class, size (nmax, 3)
memory->create(this->v2r, atom->nmax, 3, "FixEllipsoidalEnvelope:v2r")
```

define parameters and make sure the correct number of those is given as an input

```
if (narg != 7) error->all(FLERR,"Illegal fix ellipsodal envelope command [args: a, b, c, k]");
a = force->numeric(FLERR,arg[3]);
b = force->numeric(FLERR,arg[4]);
c = force->numeric(FLERR,arg[5]);
kspring = force->numeric(FLERR,arg[6]);
```

unregister fix so class does not invoke that anymore, destroy locally stored arrays

```
FixEllipsoidalEnvelope::~FixEllipsoidalEnvelope() {
  memory->destroy(v2r);
  memory->destroy(ftotal);
  atom->delete_callback(id, 0);
}
```

define methods to be used

```
int FixEllipsoidalEnvelope::setmask() {
  int mask = 0;
  mask |= POST_FORCE;
  mask |= THERMO_ENERGY;
  mask |= POST_FORCE_RESPA;
  mask |= MIN_POST_FORCE;
  return mask;
}
```

memory usage of locally stored atom-based arrays

```
double FixEllipsoidalEnvelope::memory_usage()
{
  int nmax = atom->nmax;
  double bytes = 0.0;
  bytes += nmax * 3 * sizeof(double);
  return bytes;
}
```

stuff to ensure consistent copy of data (allocate, copy, pack, unpack)

```
void FixEllipsoidalEnvelope::grow_arrays(int nmax)
{
  memory->grow(this->v2r, nmax, 3, "FixEllipsoidalEnvelope:v2r");
}

void FixEllipsoidalEnvelope::copy_arrays(int i, int j, int delflag)
{
  memcpy(this->v2r[j], this->v2r[i], sizeof(double) * 3);
}

void FixEllipsoidalEnvelope::set_arrays(int i)
{
  memset(this->v2r[i], 0, sizeof(double) * 3);
}

int FixEllipsoidalEnvelope::pack_exchange(int i, double *buf)
{
  int m = 0;
  buf[m++] = v2r[i][0];
  buf[m++] = v2r[i][1];
  buf[m++] = v2r[i][2];
  return m;
}

int FixEllipsoidalEnvelope::unpack_exchange(int nlocal, double *buf)
{
  int m = 0;
  v2r[nlocal][0] = buf[m++];
  v2r[nlocal][1] = buf[m++];
  v2r[nlocal][2] = buf[m++];
  return m;
}
```

Standard setup to interface this with the verlet

```
void FixEllipsoidalEnvelope::setup(int vflag)
{
  if (strstr(update->integrate_style,"verlet"))
    post_force(vflag);
  else {
    int nlevels_respa = ((Respa *) update->integrate)->nlevels;
    for (int ilevel = 0; ilevel < nlevels_respa; ilevel++) {
      ((Respa *) update->integrate)->copy_flevel_f(ilevel);
      post_force_respa(vflag,ilevel,0);
      ((Respa *) update->integrate)->copy_f_flevel(ilevel);
    }
  }
}
```