Proposed New Integrator Formalism for GROMACS

Michael R. Shirts[1,*] and Can Pervane[2]

[1]*Department of Chemical and Biological Engineering, University of Colorado Boulder, CO 80309*

[2]*University of Colorado Boulder and University of Southampton*

(Dated: February 15, 2017)

Abstract

We propose a new integrator model for GROMACS based around symmetric Trotter decomposition of elementary integrator elements and Monte Carlo acceptance/rejection. The model is intended to be extremely modular in order to simplify the code and make it easy to modify the integrators.

INTRODUCTION

The main goals for the new integrator formalism are:

- Simplify the integrator loop.
- Support Monte Carlo, molecular dynamics, and combinations of the two.
- Negligible loss of performance for the main code paths.
- Make it easier to verify the correctness of the algorithms implemented.
- Eliminate support for integrator algorithms that have clear drawbacks (Nosé-Hoover chains)
- Make it easier to implement new algorithms that are based in the Trotter decomposition
- Support decomposition of the forces and energies into arbitrarily many fast and slow components for multistep algorithms
- Supports for all conditions (NVE, NVT, NPT, constrained versions of all of the above) with a well-validated algorithm with only, one communication step per iteration (though some algorithms might require separate communication steps).

We propose that the MD portion be based around symmetric Trotter integration. Virtually all standard MD algorithms can be written in terms of Trotter-splitting of the propagators, and it makes it very simple to perform splitting of force components. This will make a number of things possible, such as:

- harmonic bonds with short timesteps, with long timesteps for the nonbonded.
- more rigorous short/long cutoff integration
- Make it less computationally intensive to do PME/LJ, as the Fourier space component this only needs to be called relatively rarely (10-20 steps, likely).

Each integrator is composed of a number of elementary operations. We divide them into *propagators*, which change the state of the system (position, velocities, alchemical variables), and *inspectors*, which report data about the state of the system.

Code Structuring

The code in `do_md` can be broken down (after some rearranging) in the following way.

We focus mainly on what happens inside the loop, as we only anticipate changing `Before Loop` and `After Loop` where they need to be to adjusted to utilize new data structures. There are a number of function calls that are independent of the integrator that can be moved either to the beginning of the loop (setting booleans, domain decomposition load balancing), or the end of the loop.

The integrator `Integrator` itself is where the main core of what is currently seen as the integrators are. Integrator will be a class that has as a method `do_one_step` that calls as sequence of elements. We preserve the flexibility for this sequence of elements to be constructed at run time, as we want to preserve the ability of users to easily write new integrators (though the exact way this is done is deferred for another time). A pseudocode illustration of the class is given here:

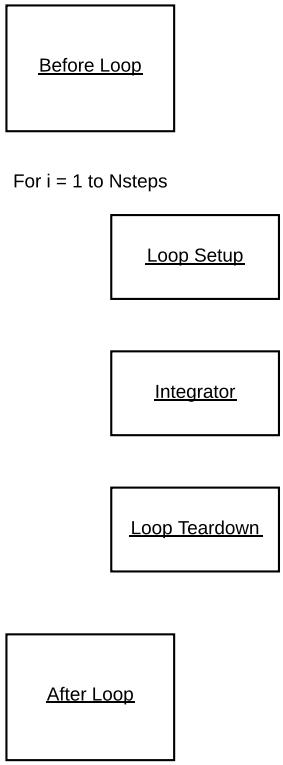

FIG. 1. Breaking down the loop in md.cpp. Blocks are not objects, but just containers for code that carry out certain functions.

```
Class Integrator
{
    public:
        //! the number of times this integrator is repeated
        int nrepeat;

        // the instruction to determine the sequence of elements, set at ru
        std::string element_sequence;

        // Constructor
        Integrator(sequence)
        {
```

```
  element_sequence = sequence
}

//! actually carry out the update
void do_one_step():
{
    for (int i=1;i<nrepeat;i++)
    {
        for (int j=1;j<nelements;j++)
        {
            elements_[j]->execute();
        }
    }
}

void initialize(elements, ndepth)
{
     set_depth(ndepth);
     set_nelements(elements.size());
     set_elements(elements);
}

void update()
{

  for (int j=1;j<nelements;j++)
  {
      elements_[j]->update();
  }

}
```

```
private:
    //! the elements of this integator
    //! set at runtime
    //! elements is a list of Element objects
    std::list<Element*> elements_;

    //! the number of elements in this integrator, set at creation
    int nelements_;

    //! Depth of this particular object in multistep recursion.
    void set_ndepth (int ndepth):
    {
        ndepth_ = ndepth;
    }

    //! set up the elements for this instantiation of the integrator
    void set_elements (std::map<char,Element*> elements):
    {
        for (int i=0; i<element_sequence.size(); i++)
        {
            switch(element_sequence[i]){
                case('A'):
                    elements_[i] = elements['A'];
                    break;
                case('B');
                    elements_[i] = elements['B'];
                    break;
                case('[');
                    // get subsequence from sequence
                    subSequence = getSubSeq(sequence);
                    // create integrator object with the subsequence
                    integrator = new Integrator(Subsequence)
```

```
                        // Initialize the integrator with the given element
                        integrator.intialize(elements);
                        element_[i] = integrator;
                        break;
                    .
                    .
                    default:
                        gmx_fatal("Not a valid integrator Element")
                }
            }
            nelements_ = elements_.size();
        }
}
```

The Element base class, propagators and inspectors will inherit from this class.

```
Class Element
{
    // sim is a pointer to Simulation class type
    public:
        virtual void execute()=0;
        virtual void initialize(Simulation& simulation)=0;

}
```

And an example of one of the propagators:

```
Class Position_Update : public Element
{
    private:
        //! Timestep used in this step; set at runtime, is not modified.
        real dt;
        //! carry out the updates
```

```
public:
    //! actually carrry out the step
    void do_update(state,start,nrend) {
         rvec * gmx_restrict x
         cosnt rvec * gmx_restrict v
         x = state->x;
         v = state->v;
         for (int i = start;i < nend; i++) {
             for (int d = 0; d < DIM; d++)
             {
                 x[i] += dt*v[i];
             }
         }
    }

    void initialize(Simulation& simulation){
        // retrieve the variables that are needed by this propagator.
    }

    void execute()
    {
        do_update(sim.state,start,nend);
    }

    void update()
    {
       // Called in loopSetup
       // Could be accessed via a getter
       // nrend=sim->get(nrend)
       nrend = sim.nrend
       start = sim.start
```

```
        }
}

Class Simulation
{
  // This class will hold all the information required by the elements
  // Thus Elements can access and change variables through this class

  std::map<char, Elements*> elements;

  Simulation(Integrator integrator)
  {
    // Create Objects of Element derived classes
    elements['A'] = new Position_Update();
    elements[nameOfElement] = new ...;

    // Loop over elements and initialize them
    elements[i]->initialize(*this);

    // initialize integrator with the elements
    // the order could be defined inside the integrator.initialize method.
    integrator.initialize(elements, ndepth);

  }

  void loopSetup(){
      // updates the element objects of integrator
      integrator.update()
  }
  void loopTeardown();
  ~Simulation(){
      //delete all elements
```

```
  }

}
```

A prototype of a workflow using the proposed classes

```
// create an integrator object with a given string sequence
integrator=Integrator(sequence)
// create a Simulation object which will create element objects
// and initialize them. Initialize integrator object
simulation=Simulation(integrator)

// Start md loop
for (int=i;i<nsteps;i++)
{
  simumaltion.loopSetup()
  // loop over the element objects and execute them
  integrator.do_one_step()
  simulation.loopTeardown()
}
```

Sequence diagram of the proposed integrator framework

There will be a function factory, running before the loop initiates, which generates objects of the `Integrator` class, each with different orderings of element objects (propagators and inspectors) depending on the algorithm. Eventually, we wish to make it possible to program integrators from the command line, but for now, all we need to do is have it created at runtime. Initially, we will simply construct them from a list of enums stored in a header file. Eventually, they can, if desired, be constructed in the MDP, and there will be several validation checks that can be run on these to check to make sure they are valid (are forces computed after position integration, but before velocity integration, are the elements called in a symmetric manner, etc.). This could in theory be done during the build process.

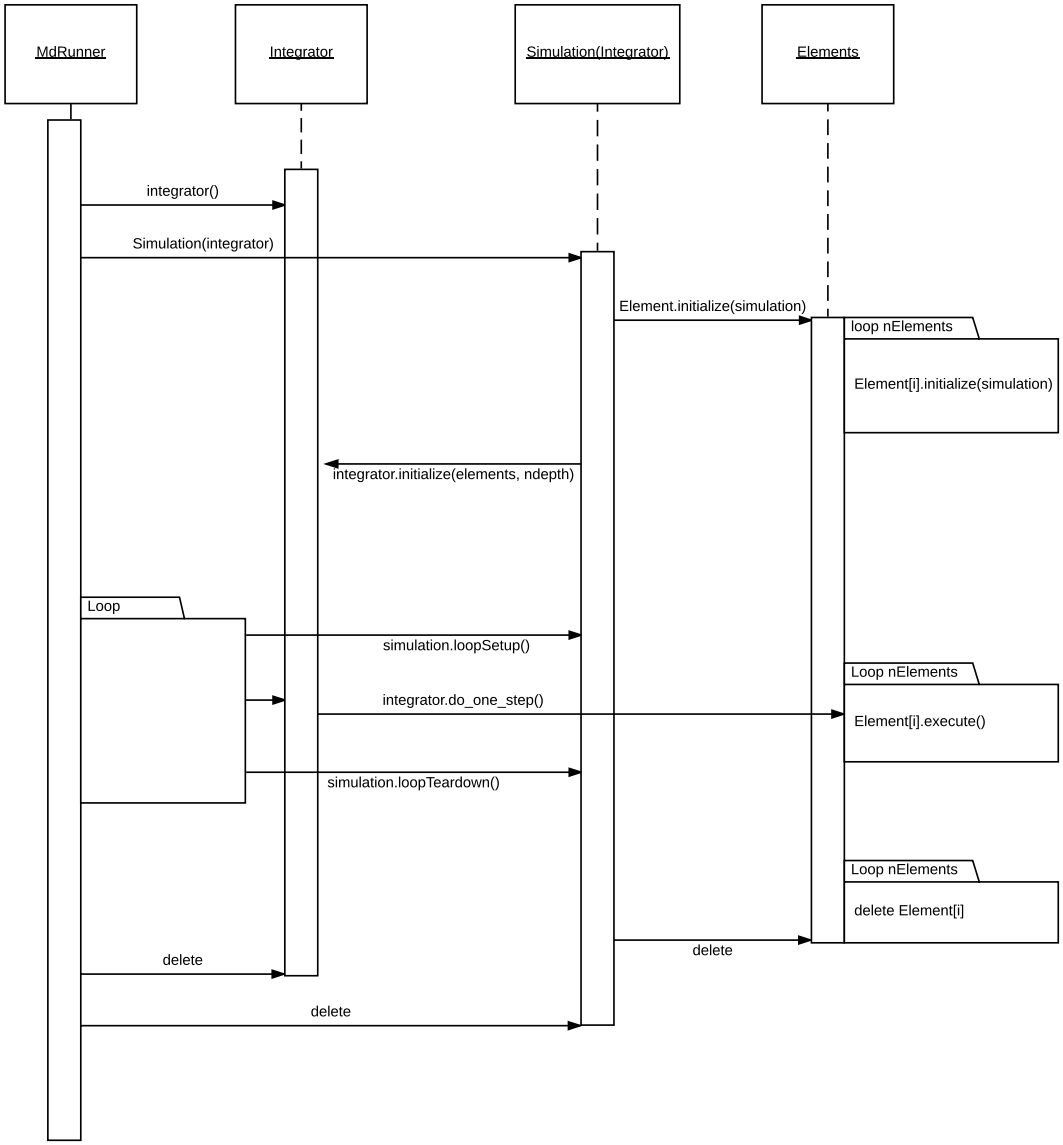

FIG. 2. Sequence Diagram of the integrator framework.

Data Flow. There are three types of data that need to flow into the elements.

1. The first is data that remains constant for the entirety of the simulation, such as Δt for propagators, and for inspectors, things like the number of DOF, the reference temperatures, the properties of the atoms, and so forth. These will be copied into the elements at the beginning – or for things that are shared, they will be placed in the `Simulation` class, which the elements will be able to access.
2. The second class are things that are set in the loop setup, but do not change during evaluation of the elements of the integrator. These will be updated using the `update` method of the `Element` class. This can include things like `start` and `nrend`, which can change from loop to loop depending on load balancing.
3. Finally, things that change during the evaluation of the list of elements of the integrator. By definition, these are either part of the dynamic state, which will be store in the `Simulation` class, or output of the inspector, like the kinetic energy, pressure, and energy, which will also be stored in the `Simulation` class.

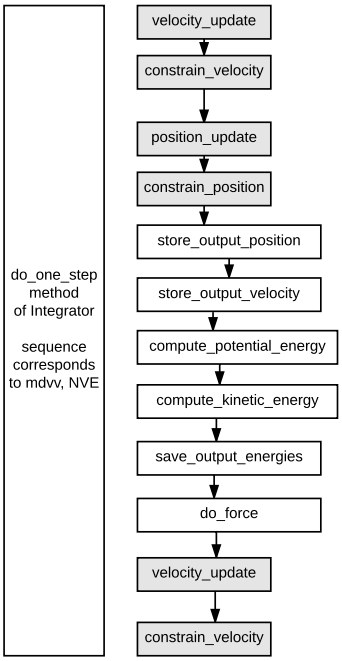

FIG. 3. Diagram of an example of `do_one_step` method of the Integrator class.

Multistep integrators Multistep integrators are encoded by making one of the integrator elements be a loop over a separate Integrator object calling `do_one_step`. We will have a counter that keeps track of the depth of the recursive call, so that we can define at runtime the element sequence for each level of depth, and the portion of the force that is evaluated (Fig. 4)

Monte Carlo Monte Carlo integrators will al-

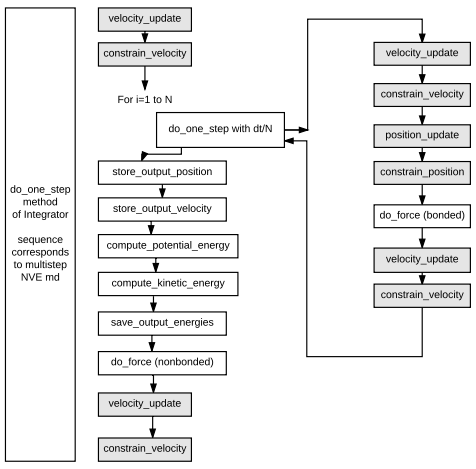

FIG. 4. Diagram of a multistep integrator with one level of depth

ways start with a `save_state` function, and end with an `accept_reject` function; if accepted, then the program simply continues; if rejected, it returns to saved state (Fig 5

Automated Validation Eventually, we will want to build into the physical validation of the integrators themselves. This would be a algorithmic level validation; Some of the things that could be checked for include patterns that 1) weren't trotter decompositions (v(dt/2) p (dt) v(dt/3)) for example and 2) required things that weren't computed (for example, requires a pressure that is never computed). This would be done on whatever data structures are used to construct the integrator by the user, and would be independent of the actual implementation in the code.

Additional Code Details Propagators operate on the state of the system, which is defined as all the dynamic elements of the simulation. The state consists of:

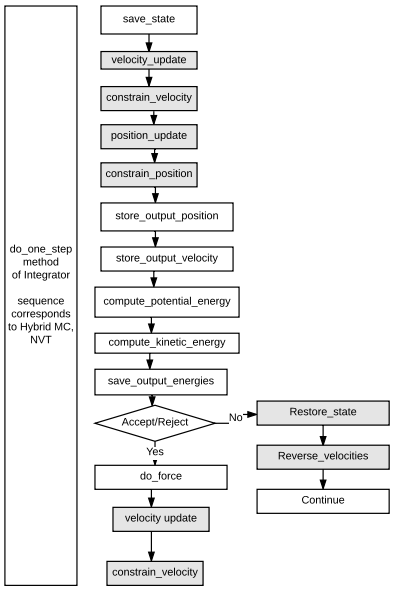

FIG. 5. Diagram of hybrid Monte Carlo integrator

- coordinates x
- velocities p
- box vector b
- box vector velocities (η) (in Parrinello-Rahman)
- alchemical variables λ
- velocity of alchemical variables λ
- auxiliary variables such as heat baths, barostats, etc. required for algorithm (ξ for Nosé-Hoover, etc.)
- velocities of auxiliary variables
- histories for history dependent algorithms (since these are functions of the state)
  - Free energy histories

- essential dynamics histories

Possibly we might consider as propagators that affect the functions:

- external fields and thermodynamic parameters s (such as T, P, μ, electric field E)

It should be similar to the current `state` structure, except we try to kill

- ekinstate (not an algorithm portable data structure)
- nhchainlength (not needed after getting rid of NH chains)
- nnhpres (not needed after getting rid of NH chains)
- ngtc (not entirely clear it should be in the state, since it's a global quantity?)
- pres_prev, svir_prev, fvir_prev quantities, if we can eliminate them using better

There are four inspectors that are collective functions:

- do_force (depends on x,b,λ). These require communication, and may be partitioned into arbitrarily many multiple components of different timescales (such as bonded/nonbonded, bonds and angles/torsions/short range electrostatics/long range electrostatics) and so forth. A partition is not spatial, so forces are defined for all particles for each partition.
- Calculation of potential energy (depends on x, b,λ). Like forces, require communication, and may be partitioned into arbitrarily many components of different timescales.
- Calculation of virial (depends on x, b,λ). Require communication, can be partitioned (since it is defined as $\sum \vec{F}_i \cdot r_i$), though it's not clear that it needs to be or easily can be done this way in the code.
- Calculation of kinetic energy (depends on p, λ (if masses change with lambda)). Cannot be partitioned, requires communication

Generally, potential energies can be computed at the same time as forces with relatively little extra effort, as can virials. Often, for different MD methods, it's possible to move forward using only the forces, so it would be preferable (though not necessary) to have routines that only compute one or the other if it doesn't make the coding too difficult.

Kinetic energies are calculate separately from energies and forces; the computational effort is almost trivial, except for the need to perform communication to sum up the total KE, which must be global.

We assume in most cases that the communications happens immediately after the corresponding collective routine is called; there may be exceptions.

Things not in `integrator` We will assume the following can be pulled out of the `Integrator` class into either setup or teardown at each loops step.

- signaling
- domain decomposition rebalancing
- file I/O (within the integrator, data is SAVED for output, but not actually written to file until the `Integrator` object returns.
- Replica exchange (maybe?)
- others?

Speed considerations of the design and workaround There are a number of possible ways that this procedure will slow down the current code down at least somewhat:

- the integrator framework means that a relatively large number of function calls happen, potentially slowing execution as these are resolved. The limiting factor
- One challenge with the Trotter formulation will be minimizing communication cycles required for computing constraints, summing kinetic energies, and summing potential energies and virials. Ideally, there is only one global communication step per cycle, which can sometimes be difficult to obtain with scaling thermostats because of the need to calculate the kinetic energy relatively frequently, and not always at the same time as the force. Particular attention must be paid to constrained dynamics, as they may require additional communication if constraint systems extend too far; the communication for constraints could potentially prove a roadblock for many Trotter formulations. HOWEVER, it will be possible to at least maintain the current procedure using fused elements, as described below.
- With the current framework, instead of having one loop over all atoms as in the current code (or, mostly one loop), there will be multiple loops over atoms, each of which carries out a simpler procedure, and there is some overhead to calling multiple loops.

If there are particular code paths we want to have maximum speed, we can *fuse* elements together, creating a larger element. These elements can then be automatically verified versus an integrator loop with the elements that they are supposed to be equivalent to.

Note that with multistep framework, it will be relatively easy to come up with a highly energy conserving framework that only requires energies being printed out whenever they are written to file. For example, if Langevin dynamics are used with a multistep integrator with 0.5 fs steps for bonded and 2 fs steps for nonbonded (without constraints), then the only communication required will be when domain decomposition is energies are printed out, which may be only every 1000 steps or so.

The propagators are defined as (labeled with letters that will be used below):

A. MD position step: $x_i(t + \delta t) = x_i(t) + v_i(t)\delta t$
B. MC position step: $x_{i+1} = \text{MC}_{move}(x_i)$
C. MD velocity step: $v_i(t + \delta t) = v_i(t) + M_i^{-1}F_i(t)\delta t$, where $M_i(t)$ are particle masses.
D. Velocity reassignment: $v_{i+1} = F(T)$.
E. MD position constraint: Constraints bonds $x(t+\delta t)$ to bond lengths at $x(t)$ (requires communication if constraints cross boundaries):
F. MD velocity constraint: Constrains $v(t)$ to have no perpendicular components to $x(t)$
G. Stochastic thermalization (Orstein-Uehlenbeck operator) of variable $\vec{s}$:
   If $a = e^{-\gamma\delta t}$, and $N(0, 1)$ is a random normal(0,1), then the process is $s_i(t + \delta t) = \sqrt{a}s_i(t) + \sqrt{\frac{1-a^2}{\beta M_i}}R$, where M_i is the mass associated with s_i, and γ is the friction, and R is a random normal(0,1)
H. Brownian dynamics (math does not allow us to compose this with other integrators) $x_i(t + \delta t) = x_i(t) + \gamma^{-1}F_i(t)\delta t + \sqrt{\frac{2k_BT\delta t}{\gamma}}R$
I. Heat bath velocity increment $\xi(t + \delta t) = \xi(t) + \delta t\mu^{-1}(KE - N_{DOF}k_BT)$ (requires KE to be calculated)
J. Velocity scaling by some variable a: $p(t) =$a$p(t)$
K. MC box vector change: Randomly choose new box vectors. Will need need to take into account the change in Jacobian $(V_{old}/V_{new})^N$ in the MC accept/reject.
L. Hamiltonian update

Note there are no chains listed above. Instead of using Nosé-Hoover chains, we will use the Nosé-Hoover-Langevin integrator, which uses a heat bath, but thermostats the heat bath with a Langevin thermalization process rather than a series of chains. This both makes the coding easier and removes some theoretical problems about ergodicity when using deterministic algorithms. We will also thermostat bath variables the same way.

Possible extensions in the future One algorithm that might be useful to have would be Grand Canonical Monte Carlo, in which the system can change the number of particles, and semigrand Canonical Monte Carlo, in which components can swap identity. This will require substantial changes by making all of the atom lists dynamic. One alternative is to have a Gibbs ensemble framework, where there number of particles remains the same, but there are two boxes, and the particles are exchanged between volumes. This would also involve a number of additional changes, because there would need to be two simulation "frames" maintained, with separate domain decomposition, parallelization over different sets of processors, barostats, two force calls, and so forth. It is not clear if either of these makes the most sense to do currently, as the proposed changes are already fairly substantial.

We assume there is an initial force call when we start, so we have $E(t)$ and $F_i(t)$, and that initial coordinates are constrained (if running constraint dynamics). We will use | to indicate calculation and collection of F (and optionally PE and V, though we indicate just by F), and ! to indicate calculation of KE. () indicate that it takes an argument. All letters indicate a $\delta t/2$ update: repeated letters indicate either two consecutive updates, or a doubled update with timestep δt. (I think) two smaller steps will usually be better, but might require additional computation.

Calculation of a collective implies that it is immediately followed by the communication required to calculate the quantity.

: Below isn't quite filled yet

APPENDIX A

List of the elements and the data required for each element (being updated)

- momentum step

  - Requires: f, a scalar function of box vector variables (for NPT), and a scalar function of heat bath variables, TC variables, Acceleration variables.
  - Alters: v

- position constraint (requires global communication)
  - Requires: x, masses, atom numbers, equilibrium lengths, box vector, dt, information variables (constr structure)
  - Alters: x, virial, dvdlambda, Langevin multipliers

- velocity constraint (requires global communication)
  - Requires: x, v, dt, atom numbers, masses, equilibrium lengths, Langevin multipliers (from the position constraint step)
  - Alters: v, virial, dvdlambda (mass)

- Stochastic thermalization (Orstein-Uehlenbeck operator)
  - Requires: random variable state, T, friction, dt, masses
  - Alters: p or box vector momentum or lambda momentum all (equivalent math), random variable state.

- Brownian dynamics (may be somewhat separate integrator - investigate)
  - Requires: T, x, mass, forces, random variable state
  - Alters: x, random variable state. Does not affect velocity

- Heat bath velocity increment
  - Requires: KE, T
  - Alters: xi

- Velocity scaling by some variable
  - Requires: T, P, b (any other variables that affect the box vector scaling?).
  - Alters: v

- Box vector scaling
  - Requires:
  - Alters: b

- Box vector velocity scaling

  - Requires:
  - Alters: $\dot{b}$

- MC coordinate moves:
  - Requires: U
  - Alters: x

- MC state moves:
  - Requires: U
  - Alters: x, lambda

- MC box vector move:
  - Requires: U
  - Alters: b, x (possibly v's)

APPENDIX B

Some examples of different algorithms implemented by the integrator framework are:

- BA2|!B: NVE MD with velocity Verlet
  1. MD velocity update
  2. MD position update (2×)
  3. Calculate collectives F, KE
  4. MD velocity update
- |!B2A2: NVE MD with leapfrog (actually same as velocity Verlet except for calculation of observables, we just offset the "start")
  1. Calculate collectives F, KE (KE calculated as average KE at $t+\delta t/2$ and $t-\delta t/2$).
  2. MD velocity update (2×)
  3. MD position update (2×)
- Rerun (special case - no dynamics required)
  1. Load new coordinates
  2. Calculate collectives F, PE, V, KE (may only be able to get KE in some circumstances)

- Two different Langevin dynamics expansions (additional terminology from Leimkuhler)

    1. A | BE2(p)BA (or "ABOBA", Symmetric Langevin Position-Verlet)
        (a) MD position update
        (b) Calculate collectives F (KE not required for integration)
        (c) MD velocity update
        (d) Stochastic thermalization of momentum (2×)
        (e) MD velocity update
        (f) MD position update
    2. BAE2(p)A | B (or "BAOAB", Symmetric Langevin Velocity-Verlet)
        (a) MD velocity update
        (b) MD position update
        (c) Stochastic thermalization of momentum (2×)
        (d) MD position update
        (e) Calculate collective F (KE not required for integration)
        (f) MD velocity update

- A | B!GHE2(ξ)HGBA: Nosé-Hoover-Langevin dynamics

    1. MD velocity update
    2. MD position update
    3. Calculate collective F, KE
    4. thermal bath update (requires KE)
    5. velocity scaling (by $e^{-\gamma\xi\delta t/2}$)
    6. stochastic randomization of thermal bath variable ξ (2×)
    7. velocity scaling (by $e^{-\gamma\xi\delta t/2}$)
    8. thermal bath update (no additional KE calculation required, as long as scaling constants are saved)
    9. MD position update
    10. MD velocity update

- Bussi-Parrinello dynamics with leapfrog (only requires one calculation)

    1. MD velocity update (2×)

2. MD position update (2×)
3. Calculate collectives F, KE (KE calculated as average KE at $t+\delta t/2$ and $t-\delta t/2$.)
4. Thermalize target KE_{target}
5. Scale velocity by KE_{target}/KE

- Bussi-Parrinello dynamics with velocity Verlet

1. MD velocity update
2. MD position update (2×)
3. Calculate collective F
4. MD velocity update
5. Calculate collective KE
6. Thermalize target KE_{target}
7. Scale velocity by KE_{target}/KE

- Twin-step (bonded, nonbonded) velocity Verlet Langevin dynamics

1. MD velocity update (time step = δt)
2. main_propagator (nsteps = n, inner time step = $\delta t/n$)
   (a) MD velocity update
   (b) MD position update
   (c) Stochastic thermalization of v
   (d) Calculate collective F (bonded)
   (e) MD velocity update
3. Calculate collective F(nonbonded)
4. MD velocity update (time step δt)

- Infrequent updates of the thermal bath of the Nose-Hoover-Langevin operator to reduce the global communication needed for KE calculation.

1. thermal bath update (requires KE from previous step)
2. velocity scaling (by $e^{-\gamma\xi\delta t/2}$)
3. stochastic randomization of thermal bath variable ξ
4. main_propagator(nsteps=n, timestep = $\delta t/n$)
   (a) MD velocity update
   (b) MD position update (2×)

(c) Calculate F

(d) MD velocity update

5. stochastic randomization of thermal bath variable ξ
6. velocity scaling (by $e^{-\gamma\xi\delta t/2}$)
7. Calculate KE
8. thermal bath update (requires KE)

- Langevin velocity-verlet with MC barostat

1. MD velocity update
2. MD position update
3. Stochastic thermalization of momentum (2$\times$)
4. main_propagator (nsteps = 1)

   (a) MC Box vector change

   (b) Calculate collective PE
5. MD position update
6. Calculate collectives F (KE not required for integration)
7. MD velocity update

- Langevin velocity-verlet with MC barostat every n steps. Note that in this case, we are alternating MC and MD, which will obey balance, but not detailed balance (need to double check on this), since MD always follows MC, and MC always follows MD. The two options could be selected randomly, this would throw off a lot of bookkeeping.

1. main_propagator (nsteps = 1)

   (a) MC Box vector change

   (b) Calculate collective PE
2. main_propagator (nsteps = n)

   (a) MD velocity update

   (b) MD position update

   (c) Stochastic thermalization of momentum (2$\times$)

   (d) MD position update

   (e) Calculate collectives F

(f) MD velocity update