Proposal for integrator framework (do_md) in future GROMACS

I'm going to put these all together at first, and then start breaking them up when we move from proposals and discussion to actual tasks.

I. Remove the iterative structure required for MTTK integration with constraints. MTTK integration will be left in (still a very nice formalism), but only for systems without constraints. This way, we get the benefits for some systems (like LJ) without the ugly and often-unstable-in-single precision iteration. This will help solve some of the issues with the difficulty to work with the md framework. I'll discuss how to get longer timesteps with MTTK below . . . One disadvantage: Will not be able to do do full, 'correct' pressure control dynamics with constraint systems, but pressure control is a bit artificial already; one will at least be able to constant pressure sampling using a MC barostat (see below). This is a todo in the next month or two (MRS). It will make it possible to make the integrator much simpler.

II. Convert the md integrator to a full Trotter decomposition framework.

This will have a number of advantages going forward. It will make it much easier to fit md and md-vv into the same framework. Note that this does not change either integrator; it just changes how it is written out and makes in more clear what steps go together in the code and where to put modifications. This is partly already in the code but not included. This formalism easily extends to the pressure control and temperature control algorithms; again, in most cases, this is the formalism that's implemented already, it's just a matter of rewriting it so it's easier to see where to add additional code when new methods are added.

III. Make it easier to do multistep integration using the Trotter decomposition framework.

It's already possible to do types of multistep integration using the Trotter decomposition framework; for example, doing pressure control or temperature control only every N steps. However, it would be useful to make it possible to only evaluate certain components of the potential energy / force every N steps as well, where N can varied independently for each component.

IV. Adding a Monte Carlo barostat

This consists in random volume changes, accept and reject based on Boltzmann weight exp(-(E+PV)). This has an advantage that there is no need for fancy integrators for NPT, which is giant pain, and the virial doesn't even need to be calculated - Virial only needs to be calculated when you want to see the pressure (every nstcalcenergy) Generalizable to anisotropic very easily. MTTK is especially a giant pain, and requires iteration to work with constraints, and current Parrinello-Rahman is incorrect because it uses mismatched virial and kinetic energies.

One disadvantage: this will require a second force/energy calculations per application (though only every nstpcouple steps). Computationally, this is not a big problem, but it will require making it easier to run do_force multiple times in a loop. Again, only the energy must be calculated additional times, not the forces.

V. Redo the main md loop to allow for either MC or MD.

Now that MD can be seen as rejectionless MD, so it is easiest to write the 'parent' loop as MC., with MD being a type of MC with 100% acceptance. Clearly point IV (MC barostat) fits under this as well. Expanded ensemble and replica exchange also fit into this.

Advantage: lots of potential advantages for calculating thermodynamic quantities and equilibrium ensembles. Lots of interest in improving MC in gromacs. Could be very useful with implicit solvent, especially, where large conformation changes could be made. Aggregation in implicit solvent can particularly be made faster. Also very useful for simple fluids.
Disadvantage: Makes it a little bit more complicated, but not much. Payoff should be worth it. As long as the general formalism is set up, extending MC by other developers or volunteers is very easy.

Thoughts? I can probably handle a lot of the gruntwork here, though there are some places help would be useful; for example, in adding the ability to call only parts of the force calculation each time.

Related issues:
Related to Gromacs - Task #1793: cleanup of integration loop

Associated revisions

Revision 30113ccc - 10/18/2013 12:11 AM - Mark Abraham
Move mdrun trajectory writing into wrapper function
Refs #1292, #1193, #1137
Change-id: I3f19e0995f7fab465184d5bab8bc2683260af853

Revision 8df8c14d - 10/28/2013 06:36 PM - Mark Abraham
Create fileio module

This patch contains only code motion. There are no functional code changes. Moves lots of I/O code into src/gromacs/fileio. This means lots of changes to avoid having everything as a compile-time dependency of everything else because everything is pulled in via typedefs.h, etc. Note that src/gromacs/legacyheaders/filenm.h and src/gromacs/legacyheaders/types/filenm.h have been consolidated into src/gromacs/fileio/filenm.h.

I/O code in files named stat*[ch] now lives in various new files in fileio.

Files outside of the module now #include its header files in a proper way, e.g. #include ".../fileio/filenm.h" or "gromacs/fileio/filenm.h" according to whether they are an installed header, or not. Files within the module are blessed and do not need that qualifier.

This module installs most of its headers (because they're almost all inter-dependent; gmxfio_int.h is not installed because it is only useful internally, vmdio.h is not installed because it relies on a header from src/external)

Files in new module

- conform to preferred include-guard format.
- have up-to-date copyright headers thanks to Teemu's automatic script
- that are installed headers refer to other GROMACS include files via relative paths

Moves mdrun trajectory writing into wrapper function.

Removes small pieces of I/O code that was hiding behind "#if 0".

Some pieces of I/O code specific to the gmxpreprocess module have remained there.

Moved a cppcheck suppression to follow matio.cpp to its new home.

Minor fix to xdrf.h logic, since it is now the subject of a CMake test.

Refs #1292, #1193, #1137
Change-id: I820036298d574966d596ab9e258ed8676e359184
Revision 488464e7 - 12/15/2014 09:40 PM - Mark Abraham

Removing iteration + constraints framework

Getting rid of iteration + constraints required by the use of MTTK + constraints, in order to simplify the main loop.

Eliminated related variables and arguments that are now unused.

Left some otherwise useless brace pairs in do_md(), so that uncrustify-friendly formatting was preserved, so we can more easily review this for correctness. Left TODOs to remove those braces later.

Implemented mdrun check so that an old .tpr with MTTK + any form of constraints cannot be run.

Refs #1137

Change-Id: i22816de7db4420a66341fa8b35d967a71ad6568

---

Revision 6ead809b - 11/16/2015 03:22 PM - Mark Abraham

Remove "support" for twin-range with VV integrators

Group-scheme twin-range non-bonded interactions never worked with VV+constraints, and removing it was a to-do item. There are no plans to make it work with VV, and there are plans to remove the twin-range scheme entirely, as well as rework leap-frog more closely into the Trotter scheme.

Refs #1137, #1793

Change-Id: ibd70b5397568bfcd328cd6dd1c5c99384d7aaca8

---

History

#1 - 02/05/2013 09:04 PM - Michael Shirts
- Description updated

Yeah, formatting looks awful . . . trying to figure out how to make it look better. Comments can be updated, but I can't figure out how to do the same with the original post. [Update] OK, I figured this out now. For later users, go to update, and then at the top of the updating section, click on the 'more' next to 'Change properties', and you will be able to edit the original post.

#2 - 02/05/2013 09:04 PM - Michael Shirts
As a rule of thumb it would be great to separate the math (integrator/constraints) from the physics (forces). Maybe that is not feasible though.

Apropos PME/LJ, at least for relatively simple membrane systems one can relatively simply implement an analytical dispersion correction that is Z-dependent. It is useful for surface tension calculations as well.

Could you be more specific? I'm not sure what you mean here.

Apropos PME/LJ, at least for relatively simple membrane systems one can relatively simply implement an analytical dispersion correction that is Z-dependent. It is useful for surface tension calculations as well.

Right, but what if the membrane starts to buckle? Then the Z-dependence fails. I think it's best to have something that is more general, and involves far fewer assumptions!

What I meant is that the integrator should be ignorant of what it is integrating. It gets x,v,a and does it's thing. It should therefore not need any
information about atoms. Again, this may be merely academic, but if this is feasible one can also test the integrator without actually doing an MD simulation.

In C++ this could be implemented - in principle - by having an integrator class with a virtual method "calc_gradient" which is then overload by the descendant class, be it mdrun or a test program.

#11 - 02/06/2013 12:06 PM - Berk Hess
Well, you do need masses, which are properties of atoms. But that's about it.
It is indeed useful to be able to test the integrator without doing a simulation, but I don't how having atoms or not comes into play here.

#12 - 02/06/2013 02:57 PM - Michael Shirts
Berk Hess wrote:

Well, you do need masses, which are properties of atoms. But that's about it.
It is indeed useful to be able to test the integrator without doing a simulation, but I don't how having atoms or not comes into play here.

If the inputs are accelerations, then you don't need masses, since f/m = a. For a number of reasons, that might not be the best way to actually pass the data in, but in theory, it could be done.

#13 - 02/06/2013 03:10 PM - Michael Shirts
David van der Spoel wrote:

What I meant is that the integrator should be ignorant of what it is integrating. It gets x,v,a and does it's thing. It should therefore not need any information about atoms. Again, this may be merely academic, but if this is feasible one can also test the integrator without actually doing an MD simulation.

In C++ this could be implemented - in principle - by having an integrator class with a virtual method "calc_gradient" which is then overload by the descendant class, be it mdrun or a test program.

OK, I see. The one issue that I see with this is that all integrators have certain ingredients -- position incrementor, velocity incrementor, box volume incrementor, box velocity incrementor, bath variable integrator -- and there are certain other operations that need to be done, such as computing kinetic energies and pressures, collecting statistics, performing parallel operations, writing checkpoints, etc. The biggest difference between integrators is the order that these are done.

So it's not clear one can completely modularize each integrator in the way you propose. It's very possible to modularize each of the components, but the integrator itself will be calling lots of different things that require atomic information inside it -- so testing independent of atoms may not be possible.
I'm not enough of a C++ expert to say the virtual method approach is wrong, but overall the force calculation, load balancing and all communication in the main MD loop is very complicated, and we will likely soon move to a data-driven scheme where different tasks can be carried out asynchronously whenever possible, and in particular separate operations on local-only vs. remote data. Obviously, this should include the integration too.

While it might be clean from an integrator testing point-of-view to abstract everything else away in a function, the integration is still a relatively straightforward operation compared to lots of other parallel algorithms, so I still think it is going to be easier to plug integration into the data-driven task framework rather than plug everything else into integration, but maybe that remains to be seen.

While it might be clean from an integrator testing point-of-view to abstract everything else away in a function, the integration is still a relatively straightforward operation compared to lots of other parallel algorithms, so I still think it is going to be easier to plug integration into the data-driven task framework rather than plug everything else into integration, but maybe that remains to be seen.

An advantages of Trotter factorization is that even NPT and NVT integrators become relatively straightforward operations, in that you write out the decomposition, and just make sure that all the quantities are computed in the proper order in the code (or, if required for efficiency, you swap steps that commute, and one can compile a list of those commuting steps).

I'm not quite visualizing what "data-driven task framework" means in the context of integrators, but I'm happy to address any issues there once I figure that out :)
In my proposal the integrator becomes the do_md routine, calling functions as needed in order to move forward in time.

Obviously everything needs to be very tightly coupled for large scale simulations.

#18 - 02/06/2013 04:29 PM - Erik Lindahl

It is still just plans, and whatever we do will be much more specific than Charm++, but we simply don't see much alternatives in a future where you are running simulations over hundreds of nodes each with multiple accelerators and maybe 128 cores each. Amdahl, imperfect load balancing and synchronization will kill us otherwise.

In my proposal the integrator becomes the do_md routine, calling functions as needed in order to move forward in time.

Yes, that sounds nice on the high level, but it relies on the concept of a black box routine that magically does the right thing with the gradient returned as a single vector. do_md() isn't that - even without a data-driven approach such a new integrator module will now have to be aware of and handle MPI, thread, and accelerator setup, parallelization, direct-space vs. PME decomposition and load balancing (home atoms will change), and probably do some communication too.

#19 - 02/06/2013 07:41 PM - David van der Spoel

Maybe this is just semantics but the integration algorithm is what determines which operations have to be be done in which order. Sure, we have to move data around and there are different possibilities on how to parallellize the sub-tasks. I'm not saying that it will be less complicated in the end (though one can hope). But the integrator does not have to decide (or even know) how the force computations are divided over cores.

#20 - 02/18/2013 08:54 PM - Teemu Murtola

Responding to some of Mark's thoughts from gmx-dev:

- In C++, being able to construct an MDLoop object that contains (lots of) objects that already have their own "constant" data will mean we only need to pass to methods of those objects any remaining control values for the current operation
  - passing of state information managed by letting the MDLoop own that data and have the object implementing the strategy ask for what it needs?

If it is feasible, I think a better design would be such that the MDLoop pushes any data that is required into the object. So that, e.g., the force evaluation object does not have any dependency on the MDLoop object itself. This of course requires that the MDLoop knows what data the object needs, but it would anyways be better to have control over this (in particular when it comes to modifying the data). If this is too complex, it is also possible to just pass the entire "state" into the object and let it deal with it. A design where there are no cyclic dependencies is much easier to unit test.

- Those objects will have a lot of inter-relationships, so probably need a common interface for (say) thermostat algorithms so that (say) the MDLoop update method knows it can just call (say) the thermostat object’s method and the result will be correct, whether there’s a barostat involved, or not
  - easily done with an (abstract?) base class and overriding virtual functions
    - however, that kind of dynamic-binding run-time polymorphism is overkill - likely any simulation knows before it gets into the main loop that it’s only ever going to call (say) AndersenThermostat’s methods
    - the overhead from such function calls is probably not a big deal - this loop is always going to be heavily dominated by CalculateForces()
    - inheritance can maximise code re-use

I don't understand what you mean by this. What is the base class, and what would the subclasses correspond to? In particular, I don't understand how
this differs from the second alternative below (any reasonable implementation that I can think of is some flavor of the second approach).

And as a nitpick, in modern object-oriented programming, re-using code is not a good justification for inheritance. ;)

- can be done by having function pointers that get set up correctly in the MDLoop constructor (i.e. "static" run-time polymorphism, as dictated by the .tpr)
  - this might lead to code duplication?
  - might lead to the current kind of conditional-heavy code, because it is now the coder's job to choose the right code path, but hopefully only in construction

There is no need to use "function pointers" for this, but instead interfaces and classes implementing those interfaces. Those classes can still share some common implementation to avoid unnecessary code duplication. Sure, the conditionals need to exist somewhere, but in the initialization there is much more freedom to organize the code, since performance is not a concern.

- could be done with compile-time polymorphism (i.e. templates)
  - lots of duplicated object code because of the explosion of templated possibilities

There are simply too many combinations to enumerate them all for templates, and the benefit of avoiding indirect function calls is very small compared to the explosion in the binary size.

- Perhaps a good way to start to get a handle on what kinds of objects and relationships we needs is to make an ideal flowchart for a plausible subset of mdrun functionality, and see what data has to be known where. Perhaps Michael can sketch something for us that illustrates what the algorithmic requirements of a "full Trotter decomposition framework" would be. (But probably not in time for this week!)

I'm strongly in favor of this approach. I don't think there is a need for any elaborate proof-of-concepts that drill into every detail at this point, but instead it is probably better to try to understand the big picture. And this kind of flow chart (and in particular the data dependencies) is essential no matter which approach we choose in the end to implement it.

---

#21 - 02/18/2013 09:35 PM - Michael Shirts

The first step is to strip out the iterative algorithm required for constraints + MTTK. I am debugging a draft of this right now, and can hopefully post it
by this weekend.

I'll start working on a draft of the Trotter integration framework, at least the algorithmic part. And yes, it will not be done by Wednesday! I can probably get a draft done by the next meeting.

---

**#22 - 02/21/2013 12:45 PM - Mark Abraham**

Sounds great, Michael. We really need something upon which we can start visualizing the way data has to be accessed/transferred and the way parallelism has to be incorporated into the design.

At the teleconference 20 Feb we also discussed:

That the use of virtual functions for managing the different behaviours of related kinds of objects is acceptable - if later there's a performance cost to them, then we should do optimization once we know that.

The idea of trying to construct a container of work task objects at MDLoop construction time might help limit the explosion of conditionality. That's still going to mean some conditional code in constructing the order of the objects in the container. These objects might do their work by knowing what virtual function to call. Inside the work triggered from the ComputeForces object there's likely going to be some similar stuff going on for the task-based parallelism we have in mind.

---

**#23 - 02/21/2013 11:08 PM - David van der Spoel**

It would be good to have a flowchart in a format that we can edit and have under version control. Any suggestions? Or is this not necessary? Such a flowchart could be part of the developer documentation. Or can it be made by software like doxygen?

---

**#24 - 02/22/2013 12:04 AM - Michael Shirts**

David van der Spoel wrote:

> It would be good to have a flowchart in a format that we can edit and have under version control. Any suggestions? Or is this not necessary? Such a flowchart could be part of the developer documentation. Or can it be made by software like doxygen?

I agree it would be nice. I also do not know the right format to put one together. I'll focus on getting the latex documentation of the iterator algorithms done (estimated 2-3 weeks).

---

**#25 - 02/22/2013 06:09 AM - Teemu Murtola**

David van der Spoel wrote:

> It would be good to have a flowchart in a format that we can edit and have under version control. Any suggestions? Or is this not necessary? Such a flowchart could be part of the developer documentation. Or can it be made by software like doxygen?

This is a good idea. It can't be generated directly by Doxygen (this is a higher-level representation than what can be automatically generated). Depending a bit on what information we want to have in the graph, and how we want to represent it, I can think of the following formats (of the top of my head, did not do any searching):
All are plain text (the last is XML), so can be readily version-controlled. msc and dot may have some limitations in the amount of text that can be easily put into nodes/edges and on the layout (both are designed for automatic layout), while graphml (depending on the editor) can be used to represent anything.

The first two can be directly embedded into Doxygen documentation (using `\dot`, `\dotfile`, `\msc` and `\mscfile` tags). Have not used msc in practise, and have only used dot as an intermediate format for automatically generated graphs, so don't know whether there are good editors for them. But the syntax is not that hard to edit by hand. Have used `yEd` for creating graphml graphs; as graphml is quite "flexible", we probably need to agree on an editor to use to have it understand and write the same extensions.

---

**#26 - 02/22/2013 08:16 AM - David van der Spoel**

I played a bit with a tiny script called cpp2dia [http://cpp2dia.sourceforge.net/](http://cpp2dia.sourceforge.net/) which generates dia or dot output. Unfortunately the output is not very useful. However for the purpose of designing a high level flow chart we do not need every detail. `yEd` maybe a decent tool indeed, although it may take some learning.

Including `\dot` or `\msc` into the code seems like a lot of work judging from the documentation at [http://www.stack.nl/~dimitri/doxygen/manual/commands.html#cmddot](http://www.stack.nl/~dimitri/doxygen/manual/commands.html#cmddot).

---

**#27 - 02/22/2013 09:11 PM - Teemu Murtola**

David van der Spoel wrote:

> I played a bit with a tiny script called cpp2dia [http://cpp2dia.sourceforge.net/](http://cpp2dia.sourceforge.net/) which generates dia or dot output. Unfortunately the output is not very useful. However for the purpose of designing a high level flow chart we do not need every detail.

As I said, I don't think that this kind of diagram can ever be extracted from the source code directly. It requires too much human input in what is relevant and what not to produce a clear and concise representation of, e.g., the data and control flow within mdrun. For class diagrams and such, Doxygen already creates quite a few diagrams like inheritance and collaboration diagrams. Not all of them are currently enabled in our config, and they would require installing graphviz on Jenkins to get them produced, but you can take a look at them locally if you are interested.

Including `\dot` or `\msc` into the code seems like a lot of work judging from the documentation at [http://www.stack.nl/~dimitri/doxygen/manual/commands.html#cmddot](http://www.stack.nl/~dimitri/doxygen/manual/commands.html#cmddot).

I don't understand your point. If the dot/msc diagram already exists, it takes literally 30 seconds to include it into the documentation produced by Doxygen (well, a bit more if you want to add cross-references from the diagram and they are not yet there). If the dot/msc diagram does not exist, then the effort required to create it doesn't depend at all on whether it will be put into the Doxygen documentation or somewhere else. And I'm not proposing that we start maintaining, e.g., class diagrams manually; as I said above, Doxygen already produces at least some types of diagrams quite nicely fully automatically.

As an addition to my list, `Dia` is also one option (haven't tried it myself). We could also have the diagram as simple SVG, but a diagram created in a dedicated graph-drawing tool may be easier to both create initially and maintain in the long run.
OK, I've committed in the "remove constraints + MTK self-consistent iterations" each step. Next task is to put together the trotter factorization outline with suggestions for code organization. Will probably take 2-3 weeks to find time to finish, I'm estimating.

Mark Abraham 05/22/2013 06:08 AM

Going to need something on paper soon, Michael, if we will have time to design and implement on the proposed time scale.

Michael Shirts 08/07/2013 04:32 AM

I've been working off and on the document for this, and it's not coming quickly. I'll try to post something this weekend before I head off on travel.

However, I've come to the conclusion that probably the only way to make everything very clear will be to work on the code directly. Even to myself, it's hard to know exactly where everything fits without making the changes myself! In conjunction with this document, then I will work on the following steps:

1) Validate the removal of iteration, including catching up with master.
2) Explicitly "Trotterize" the code so that both leapfrog and velocity Verlet are explicitly written in terms of full position and half step velocities. This should not change any output, as the same steps will be done in the same order -- they will just be broken up into routines differently. SD will probably be removed in this draft version, as I'm proposing an alternative to that I think will work better in the end (see next post).
3) See if I can implement a very basic Monte Carlo wrapping code in the integrator, of which MD is a subset.

I think MC barostat will come after, as that will be pretty easy to do as a stand alone. More straightforward multistep will come after as well. But the parts above are the ones that really need to be organized to see how it all goes together.

We'll see how this goes. A draft of the document will be posted this weekend, but getting it all into code will take a bit longer. Plus, I'm still working out all of the free energy issues + organizing the workshop in Sept. But if I get the parts listed above in, even roughly, then it should start making it much easier for other people to join in to get more of the work done.

Michael Shirts 08/07/2013 04:35 AM

In terms of Langevin integrators: The framework here by Sivak, Chodera, and Crooks looks very appealing.


It's very general, and reduces to standard integrators in a number of cases (Brownian, Bussi thermostat, velocity verlet).

I believe the Nose-Hoover approach can be plugged in trivially instead of the stochastic Ornstein-Uhlenbeck step.

Michael Shirts 08/07/2013 04:48 AM

One general note:

I think a lot of the organization issue comes down to leapfrog vs. velocity Verlet. For simple cases, they are pretty much interchangeable. I would
argue there are two main perceived advantages of leapfrog.

- 1/2 step velocities are less biased better than full step velocities as a function of time step. But this isn't actually a property of leapfrog. It's a property of where you collect statistics for the kinetic energies, and where in the integral you supply your temperature exchange. The right solution might be to implement AS velocity verlet, but with 1/2 step averaged kinetic energies.

- You condense the integration steps into fewer steps. However, as long as you are not collecting global statistics at at every half step, the speed cost is negligible. 2 extra order N routines are cheap. There are perhaps marginal roundoff error improvements, but given the errors inherent with 1-2 fs timesteps, these should be negligible.

- Fewer constraint steps In this case as well -- if you are not collecting full time step statistics, then you don't need to constraint the full time steps. So you can write a "leapfrog" version that only constraints every other step.

I'm leaving out a few details here. But the point is, I think that one can get most of the advantages that leapfrog provides while explicitly casting the integrator in a symmetric Trotter form that is organized more like velocity Verlet.

So I'd argue that to make it most general, we write the code in terms of a Trotter formalism, with explicit half velocity and full velocity steps, and place the data collection / constraints in a way that can get the "leapfrogish" advantages.

Please bring up anything I'm missing here before I jump in and start coding :)

#33 - 08/09/2013 02:18 AM - Michael Shirts

Being a bit more explicit on the last comment:

As I go through the code I'm making some notes. This is my basic understanding so far. I want to make clear the plans before I post them.

- Leapfrog and velocity Verlet are alternative ways to write a symmetric Trotter decomposition.

\[ \text{coup}(1/2) + \text{vel}(1/2) + \text{pos}(full) + \text{vel}(1/2) + \text{coup}(1/2) \]

Where coup performs all the coupling steps (pressure, temperature, etc.)

- There are two main reasons leapfrog as traditionally handled may have advantages.
    1. It explicitly calculated the kinetic energy from the half-steps, which is a lower bias estimate of the kinetic energy
    2. It allows constraint and coupling half step terms to be "collapsed," which can save communication.

These are the prime reasons. For NVT, when one collapses the terms, then one would like to do this:

\[ \text{vel}(1/2) + \text{coup}(1/2) + \text{coup}(1/2) + \text{vel}(1/2) + \text{pos}(full) \]

However, for NVT methods, you are actually doing this:

\[ \text{coup}'(full) + \text{vel}(full) + \text{pos}(full) \]

in which some algebra is used to do the split step reversibly, so that the coupling step inside is equivalent to the coupling-prime step outside.

With pressure control, the algebra to move the coupling term outside is not quite exact, and there can be slight errors.

If this is correct, then the plan will be:

To write the code in terms of velocity Verlet, with the leapfrog essentially being velocity Verlet with 1) a 'doubled' coupling operator to save constraint and communication costs and 2) a half-step averaged KE.
When using the doubled coupling operators, velocities would be output as half steps; otherwise, they would be full step velocities.

#34 - 08/09/2013 02:57 AM - Mark Abraham
That all sounds great Michael, inasmuch as I still do not completely understand the whole of it!

My mental grand scheme is to move to a family of objects implementing integration strategy elements via Runner owning objects whose virtual methods resolve at run time to code that executes what the .tpr said should be happening (rather than horrible n-part conditionals and thousand-line functions of conditional nests that have to pass a dozen parameters if they dare call a function). That is somewhat orthogonal to this development, but it is much easier to do one before the other. So go right ahead. Anything that clarifies, comments, consolidates or corrects is fine by me!

#35 - 08/09/2013 03:13 AM - Peter Kasson
Agree here--I haven't gone through the integrator math but am following with interest. Comments from Berk etc. on how all this works with constraints would be most helpful I think.

#36 - 08/09/2013 06:08 AM - Michael Shirts
Peter Kasson wrote:

Agree here--I haven't gone through the integrator math but am following with interest. Comments from Berk etc. on how all this works with constraints would be most helpful I think.

Constraints are pretty much transparent to NVT scaling methods, regardless of how you do the bookkeeping. Since all temperature control methods scale velocities isotropically, whether you scale before or after constraints doesn't matter. Randomization methods (langevin or andersen) are a bit different, since you need to remove the component of the randomization that is parallel to the bonds). But this constraint step is cheaper, since the Lagrange multipliers for each bond can be used, i.e. noniterative. Sometimes you need to do some bookkeeping so you have the right components in the right place, and it will take a bit of work to make it clean, but not a basic problem.

Barostats are all horrible with constraints. There is no way to do barostats exactly right with constraints without iteration, which is nightmarish. You can get sort of close with some tricky math. Current PR is a case in point -- close to being right, but not quite.

A MC barostat (next task after rationalizing the basic integrator step) can address problem. MC barostats are not super efficient (can't take large steps) but they don't care about constraints if set up correctly.

#37 - 08/09/2013 07:42 AM - Erik Lindahl
Not having to do a second communication step on the velocities is indeed one of the main reasons why we have liked leapfrog, and communication will likely be a worse bottleneck in the future, so that is an important feature to keep.

If we can get both that and the half-step kinetic energies (at least optionally) I think it would be great. The latter aren't discussed as much in the field (I still think Gromos has it wrong, for instance), but it can lead to strange issues where some properties are quite sensitive to your timestep, although the timestep being small enough for accurate integration!
and the half-step kinetic energies (at least optionally) I think it would be great. The latter aren't discussed as much in the field (I still think Gromos has it wrong, for instance), but it can lead to strange issues where some properties are quite sensitive to your timestep, although the timestep being small enough for accurate integration!

Erik, can you restate this? I think a word might have gone missing. Do you mean using the half-step kinetic energies results in some properties that are sensitive to timestep, or that not using them results in some properties that are sensitive to timestep? And you mean in NVT, correct? In NVE, the kinetic energy is the only property that will change - the trajectory will be exactly the same for leapfrog and vv (mod rounding error)

IIRC, our current approach of calculating velocity-derived properties at the velocity steps \((n+0.5)\cdot dt\) and then averaging them around each whole step provides more accurate estimates than first interpolating velocities corresponding to the position steps \((n\cdot t)\) and then using these to calculate e.g. temperature. I even think there might be something about this in a (very) old commit message from Berk...

For kinetic energies, then averaging the half step properties is definitely lower bias (though it's noisier). A few citations on this that I'll post eventually. However, were there other velocity dependent properties that this has been explored with?

Also note that there are thermostats that require NO communication. For example, Langevin thermostats and Andersen thermostat require zero calculation of total kinetic energies -- temperature control is entirely local, not global. Bussi applied at a per-atom or per DOF level is also zero communication (though it would need to be tweaked to be efficient when doing per atom communication).

On the Trotter decompositions, everybody I've read makes the arbitrary choice of splitting the operator that combines A and B in the following way:

\[ \exp(dt \, [A+B]) = \exp(0.5 \, dt \, A) \exp(dt \, B) \exp(0.5 \, dt \, A) + O(dt^3). \]

Is the choice of one half just about minimizing the size of the \(O(dt^3)\) term? Put more concretely, that the integration phase of either \(x\) or \(v\) is more
accurate if each is done with the half-step update of the other? So that using (say) 1/3 and 2/3 would be less effective?

#43 - 08/11/2013 11:36 PM - Michael Shirts

Mark Abraham wrote:

On the Trotter decompositions, everybody I've read makes the arbitrary choice of splitting the operator that combines A and B in the following way:

\[ \exp(dt \ [A+B]) = \exp(0.5 \ dt \ A) \ \exp(dt \ B) \ \exp(0.5 \ dt \ A) + O(dt^3). \]

Is the choice of one half just about minimizing the size of the \(O(dt^3)\) term? Put more concretely, that the integration phase of either x or v is more accurate if each is done with the half-step update of the other? So that using (say) 1/3 and 2/3 would be less effective?

My understanding is that the 1/2 + 1/2 is necessary to make it time symmetric, which is good for a number of reasons (physics are time-symmetric, minimizes error, etc).

Doing the position operator as the inner operation vs. velocity operator is arbitrary, but would require two force evaluations in the inner loop, which isn't so good.

#44 - 08/12/2013 09:15 AM - Mark Abraham

Michael Shirts wrote:

Mark Abraham wrote:

On the Trotter decompositions, everybody I've read makes the arbitrary choice of splitting the operator that combines A and B in the following way:

\[ \exp(dt \ [A+B]) = \exp(0.5 \ dt \ A) \ \exp(dt \ B) \ \exp(0.5 \ dt \ A) + O(dt^3). \]

Is the choice of one half just about minimizing the size of the \(O(dt^3)\) term? Put more concretely, that the integration phase of either x or v is more accurate if each is done with the half-step update of the other? So that using (say) 1/3 and 2/3 would be less effective?

My understanding is that the 1/2 + 1/2 is necessary to make it time symmetric, which is good for a number of reasons (physics are time-symmetric, minimizes error, etc).

Time symmetry does not seem to me to be the right explanation; the overall step should be symmetric in time (and the non-commutativity of the position and velocity updates makes that non-trivial), but one can do an \(x(1/3)v(1)x(2/3)\) in both time directions as readily as any other x splitting.

Having now read further, the issue of the choice of 1/2 was dealt with very early by Newmark (1959)

Interpolation of (say) accelerations to update a velocity is a fundamental operation, and that is most accurately done when done evenly. It seems that doing anything else introduces some artificial damping. So using 1/2 is about minimizing error, but it is not the error in the splitting of the update operator that is being minimized.

Also, nobody needs to make the attempt to read the actual Trotter paper. It's on Banach algebras and not for the faint of heart. Fortunately, I had a friend visiting from Australia who is a Banach algebraist... The Trotter result only confirms that in the limit of an infinitesimal time step, the split
propagation operator reduces to the unsplit operator. That's the least of our concerns!

Doing the position operator as the inner operation vs. velocity operator is arbitrary, but would require two force evaluations in the inner loop, which isn't so good.

Sure

#45 - 08/19/2013 06:48 AM - Michael Shirts
I've posted a .tex file 'integrators5.tex' in the git manual repository (master branch) which outlines the integrator, thermostat, barostat and MC plans for 5.0. It is still VERY preliminary, and not entirely complete, but I wanted to get something up there.

Current todos

- The first step is to remove all iterative steps. A draft has already been completed in the master branch.
- The next step is to make integrators explicitly Trotter factorization, making leapfrog code branch "doubling" of Trotter steps, starting with the temperature control, which is generally straightforward. This process is underway.
- The third step is to combine and rationalize the pressure control integrators.
- The next step is to Monte Carlo-ize the integrator routine.

Ideally the second step will be completed by the GROMACS workshop, but since I've been running it, that may not happen.

#46 - 09/16/2013 02:19 AM - Michael Shirts
Just had a good discussion with Mark A. at the conference. We discussed particularly:

- the organization necessary to reorganize the loop so that we can better support multistep integrators and MC/MD. The basic format is then:

  do from n=0 to nstep
do_force(x(t))
save the energy for next loop.

(accept/reject) (MD is a 100% acceptance, others are and based on new energy and previous energy)
  If reject, return to backup state

save trajectory and energy information

copy the state into a state_old

11/19/2015
propose a new configuration (in MD, is N steps of MD integrator)

This ordering will support wide variety of functionality MD and MC functionality, including hybrid MC and
an MC barostat.

2) hide a lot of the signaling and trajectory processing inside functions so md.c is clearer, and reorder it so that
the integrator can be isolated to the same self-contained parts of the code.

#47 - 09/18/2013 12:53 AM - Mark Abraham
Agreed. Various clean-up patches have hit gerrit. I plan some wrapping of signalling stuff, too.

#48 - 05/13/2014 09:34 AM - Mark Abraham
- Subject changed from Proposal for integrator framework (do_md) in 5.0 to Proposal for integrator framework (do_md) in 5.1
- Target version changed from 5.0 to 5.x

#49 - 08/03/2015 10:13 PM - Mark Abraham
- Subject changed from Proposal for integrator framework (do_md) in 5.1 to Proposal for integrator framework (do_md) in future GROMACS
- Target version changed from 5.x to future

#50 - 08/03/2015 11:08 PM - Mark Abraham
- Related to Task #1793: cleanup of integration loop added

#51 - 10/07/2015 09:33 PM - Gerrit Code Review Bot
Gerrit received a related patchset '1' for Issue #1137.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: Ibd70b5397568bfcd328cd6dd1c5c99384d7aaca8
Gerrit URL: https://gerrit.gromacs.org/5187