GROMACS - Feature #1625

Gromacs Python API

10/13/2014 02:45 PM - Alexey Shvetsov

Status:	New	
Priority:	Normal	
Assignee:		
Category:	analysis tools	
Target version:	future	
Difficulty:	uncategorized	

Description

Hi all!

There is an prototype of Gromacs Python api[1]. This prototype is tightly coupled to C++ trajectory analysis code so it will be good if it will be merged to git master in near future.

[1] http://biod.pnpi.spb.ru/gitweb/?p=alexxy/gromacs.git;a=commit;h=07e8a2a25ab5a62f63af77fe0dd1405cd41ee5ce

Related issues:

Related to GROMACS - Task #2045: API design and language bindings	**New**

History

#1 - 10/13/2014 08:33 PM - Teemu Murtola

- Tracker changed from Bug to Feature

- Category set to analysis tools

Commenting here instead of in gmx-developers.

I think this is a good initiative, and to get it moving forward, I'd propose splitting this into two pieces. There will be some coupling between the two, but mostly these can be discussed independently:

Infrastructure (directory layout, integration into build system, technicalities etc.)

If/when people will be interested in developing Python bindings in the future, all of this should be fully reusable for that, and can be developed more or less independently of the API itself (as long as there are at least some binding that will benefit from such infra...). Things to consider:

- Is SIP the desired approach? I personally don't know about various alternatives, but choosing the tool that generates the wrappers is perhaps the biggest decision that will be difficult to change later.
- It is probably easy to get started by adding a GMX_PYTHON_BINDINGS CMake option and make it off by default. If later development makes it portable enough, with suitable detection in place, it may be an option to enable it by default if the detection says that it will work.
- Ideally, the build system should tell whether all prerequisites are satisfied, and fail gracefully if GMX_PYTHON_BINDINGS=ON, but not all the preconditions are in place.
- I would propose putting all the source code into src/python/, as there is no natural single library produced by all this, if there are no constraints from the issues below. This can be a home both for the generated bindings, as well as for pure Python code written on top of them to provide a better API where sensible.
- Jenkins should check that the bindings keep on working, ideally in a few different build configurations. Ideally, it would run at least some real unit tests that use the bindings.
- Is it possible to make the build such that it can be easily run without installing anything? The tests for the bindings should work this way, and also for development it would be convenient to be able to run everything from the build tree.
- How does the build system interact with other Python scripts currently used from CMake? Those other scripts may currently require Python 2.7; if this requires Python 3, it may be tricky to make things like find_package(PythonInterp) work sensibly.

Actual analysis API (how to proceed with the wrapping)

I would propose that instead of blindly wrapping the C++ API 1-to-1, the initial focus would be on supporting higher-level Python tools that, e.g., reuse results from the existing C++ tools. This has several benefits:

- Need to wrap less things for very nice functionality.
- Blind wrapping of C++ API down to the lowest level (e.g., rvec) into Python classes makes the Python code look more or less like equivalent C++ code. For implementing an analysis tool from scratch, one needs a lot of low-level functionality; for just combining results of existing tools, the API can operate on higher level and initially avoid some of these issues.
- In the long run, it would probably make sense to expose at least most data structures in numpy-compatible arrays. Ability to use numpy to do higher-level operations would give a lot more power to the Python API, compared to just exposing low-level lists or tuples.

I'm not saying that it is useless to have the ability to implement tools from scratch with Python, but I think the higher-level approach allows us to add more value with less effort (and less maintenance effort, as we can avoid wrapping a lot of low-level code that may still change as part of the ongoing reorganization).

As a concrete example, consider implementing a replacement for gmx sorient as a tool that

1. Runs gmx gangle and gmx pairdist with suitable selections, doing both analyses during a single pass over the trajectory.
2. Instead of writing all the individual angles and distances to files, captures them at the Python script using a customized gmx::AnalysisDataModuleSerial, and computes the desired output from the combined data.

Doing this requires some additional C++ code as well, but I think this is where the largest benefit from Python bindings would come from: the ability to combine results from multiple analyses without huge intermediate files, as well as just the ability to run multiple simple calculations in one pass over a large trajectory, would be quite nice boosts for productivity.

#2 - 10/13/2014 09:54 PM - Maxim Koltsov

Hi Teemu et al.
I'm the mentioned student working on pygromacs. Let me clarify some points of my work.

Teemu Murtola wrote:

> Commenting here instead of in gmx-developers.
>
> I think this is a good initiative, and to get it moving forward, I'd propose splitting this into two pieces. There will be some coupling between the two, but mostly these can be discussed independently:
>
> **Infrastructure** (directory layout, integration into build system, technicalities etc.)
>
> If/when people will be interested in developing Python bindings in the future, all of this should be fully reusable for that, and can be developed more or less independently of the API itself (as long as there are at least some binding that will benefit from such infra...). Things to consider:
>
> - Is SIP the desired approach? I personally don't know about various alternatives, but choosing the tool that generates the wrappers is perhaps the biggest decision that will be difficult to change later.

I've considered three main options:

- SWIG
- sip
- boost::python

First of all, SWIG vs. SIP: they are fairly similar, but swig is multi-targeted thing (it supports perl, java, etc...), while sip is aimed specifically at making wrappers for Python. I'm not sure who has the advantage in it, but I can say that sip is used (and was created) for big projects like PyQt and pykde, so it's likely to stay maintained. Moreover, I've seen claims that porting from swig to sip increases the performance. And sip files are rather easy to write, so I've chosen that way.
Now about boost::python: gromacs code guidelines advise developers to avoid using boost unless totally necessary, so that's a -1 for it. Additionally, sip declarative style seems more simple than boost's C++ code.

> - It is probably easy to get started by adding a GMX_PYTHON_BINDINGS CMake option and make it off by default. If later development makes it portable enough, with suitable detection in place, it may be an option to enable it by default if the detection says that it will work.
> - Ideally, the build system should tell whether all prerequisites are satisfied, and fail gracefully if GMX_PYTHON_BINDINGS=ON, but not all the preconditions are in place.
> - I would propose putting all the source code into src/python/, as there is no natural single library produced by all this, if there are no constraints from the issues below. This can be a home both for the generated bindings, as well as for pure Python code written on top of them to provide a better API where sensible.

Agreed on those three points.

> - Jenkins should check that the bindings keep on working, ideally in a few different build configurations. Ideally, it would run at least some real unit tests that use the bindings.
> - Is it possible to make the build such that it can be easily run without installing anything? The tests for the bindings should work this way, and also for development it would be convenient to be able to run everything from the build tree.

Well, python wants to see .so files along with __init__.py in directory called `gromacs` (the same as the package name), so making cmake build dir called `gromacs` and putting __init__.py there allowes one to use bindings without installing (that's how i test them now).

> - How does the build system interact with other Python scripts currently used from CMake? Those other scripts may currently require Python 2.7; if this requires Python 3, it may be tricky to make things like find_package(PythonInterp) work sensibly.

cmake modules and corresponding python files were borrowed from pykde4 project, which compiles on both pythons, so I think there will be no problem with this.

> **Actual analysis API** (how to proceed with the wrapping)
>
> I would propose that instead of blindly wrapping the C++ API 1-to-1, the initial focus would be on supporting higher-level Python tools that, e.g., reuse results from the existing C++ tools. This has several benefits:
>
> - Need to wrap less things for very nice functionality.
> - Blind wrapping of C++ API down to the lowest level (e.g., rvec) into Python classes makes the Python code look more or less like equivalent C++ code. For implementing an analysis tool from scratch, one needs a lot of low-level functionality; for just combining results of existing tools, the API can operate on higher level and initially avoid some of these issues.
> - In the long run, it would probably make sense to expose at least most data structures in numpy-compatible arrays. Ability to use numpy to do higher-level operations would give a lot more power to the Python API, compared to just exposing low-level lists or tuples.

Using numpy is a sensible suggestion, thanks. I will think on how can I implement it.

> I'm not saying that it is useless to have the ability to implement tools from scratch with Python, but I think the higher-level approach allows us to add more value with less effort (and less maintenance effort, as we can avoid wrapping a lot of low-level code that may still change as part of the ongoing reorganization).
>
> As a concrete example, consider implementing a replacement for gmx sorient as a tool that
>
> 1. Runs gmx gangle and gmx pairdist with suitable selections, doing both analyses during a single pass over the trajectory.
> 2. Instead of writing all the individual angles and distances to files, captures them at the Python script using a customized gmx::AnalysisDataModuleSerial, and computes the desired output from the combined data.
>
> Doing this requires some additional C++ code as well, but I think this is where the largest benefit from Python bindings would come from: the ability to combine results from multiple analyses without huge intermediate files, as well as just the ability to run multiple simple calculations in one pass over a large trajectory, would be quite nice boosts for productivity.

Actually, I was going to make it possible to implement sample module (template.cpp) in python to demonstrate use of bindings.

#3 - 10/14/2014 04:33 PM - Alexey Shvetsov

I added build target (disabled by default) for cmake build system. So now it can optionaly build python modules gromacs.Options and gromacs.TrajectoryAnalysis.
All code now lives in src/python

#4 - 10/14/2014 08:59 PM - Teemu Murtola

Maxim Koltsov wrote:

> Actually, I was going to make it possible to implement sample module (template.cpp) in python to demonstrate use of bindings.

I gathered as much by looking at your initial code. And this is why I tried to make a point that I see very few advantages in doing that instead of what I propose, but I see a lot of potential disadvantages. You are of course free to do as you will, but one could hope that you can at least provide a rationale for what you are doing that justifies why you did not consider the advantages of the alternative approach that was presented to you.

#5 - 10/14/2014 09:33 PM - Alexey Shvetsov

I think python api can be devided into two parts:
1. Low level wrapped c++ functions and classes
2. High level Python api around low level functions and classes

#6 - 10/15/2014 06:10 AM - Teemu Murtola

Alexey Shvetsov wrote:

> I think python api can be devided into two parts:
> 1. Low level wrapped c++ functions and classes
> 2. High level Python api around low level functions and classes

No one probably disagrees with that (I, for example, wrote *"This can be a home both for the generated bindings, as well as for pure Python code written on top of them to provide a better API where sensible"*). But this is not the discussion here. My point is whether the selection of wrapped C++ functions/classes should be:

1. Blind wrapping of everything that is potentially usable for writing analysis tools from scratch, such that people can convert line-to-line between C++ and Python code, or
2. A more clever selection of only high-level classes, with some extra code added so that the Python bindings would allow for genuine added value that is more complex to achieve with C++ code alone.

#7 - 10/15/2014 08:11 AM - Roland Schulz

I looked at multiple of the possible tools and I think of those you listed sip is best. I think the biggest disadvantages of sip are that it doesn't use a full c++ parser, that it isn't documented well, and that it doesn't produce informative error messages. Alternatives you didn't list are pybindgen, pyste, xdress, and Shiboken. Further are listed here: https://github.com/cython/cython/wiki/AutoPxd. I think the design and documentation of xdress are particular good. But it is still relative new and doesn't have support for subclassing c++ classes in python (https://github.com/xdress/xdress/issues/176)

I think there would be value in being able to write python analysis tools using the same API. If for nothing else than allowing people not familiar with C++ to write analysis tools. Having an interface to Numpy is essential but shouldn't be hard.

#8 - 10/15/2014 01:04 PM - Alexey Shvetsov

SIP is used by large projects such as kde and qt (both are in C++). So SIP is well suited for C++ projects. Also are there some decisions related to vector and matrix storage classes in C++? Or C++ api will still use rvec arrays?

#9 - 10/15/2014 07:07 PM - Roland Schulz

When I said "full c++ parser" I didn't mean to say it only has to support for C or a subset of C++. What I meant was that it does not understand all of the contents of C/C++ headers. Because C/C++ are notoriously difficult to parse. Thus only those wrapper generator which either use gcc (gccxml) or clang (libclang) can be mostly automatic. For sip one of the main reasons why one needs the sip files is that the parser isn't complete. Thus it works for large C++ projects but is less automatic than one of the solutions which build on gcc/clang.

#10 - 11/23/2014 12:58 PM - Alexey Shvetsov

Maks converted py api to use numpy arrays

https://biod.pnpi.spb.ru/gitweb/?p=alexxy/gromacs.git;a=commitdiff;h=1623bb80202254a83e125ce2ea51209e9f2b97bb

#11 - 12/06/2014 05:14 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1625.
Uploader: Alexey Shvetsov (alexxy@omrb.pnpi.spb.ru)
Change-Id: I27da34dc5e04a253cb88f0e4fc0dd6f48ad9c2ca
Gerrit URL: https://gerrit.gromacs.org/4269

#12 - 12/12/2014 01:03 PM - Maxim Koltsov

Well, I'm ready to have a wider discussion on the future of these bindings. I must confess, I know little of the internal structure of gromacs, so if you gave me some pointers on how to implement your suggestions (gmx sorient and so on), that'd be great.

I was given a generic task 'make gromacs python bindings', so I didn't truly know what is needed. I'm willing to make this useful and comfortable for gromacs users :)

#13 - 12/13/2014 01:21 PM - Teemu Murtola

Instead of blind duplication of the C++ API, I'd focus on being able to write Python code along these lines (exact API can of course adapt to what is easiest to implement and easiest to understand in various contexts):

```
class MyCustomDataCombiner(object):
    def __init__(self, distances, angles):
        self._distances = distances
        self._angles = angles

    def process_frame(self):
        curr_dist = self._distances.get_current_frame()
        curr_angles = self._angles.get_current_frame()
        # do whatever custom analysis on the combined angles and distances

# Parameters can specify the trajectory, begin and end times etc
runner = PythonTrajectoryAnalysisRunner(...)
# These make the runner run the specified predetermined analysis modules, with the given parameters used to in
itialize them.
distances = runner.add_module('distance', select=['<selection1>', '<selection2>'], ...)
angles = runner.add_module('gangle', g1='vector', g2='vector', group1=[...], group2=[...])
# The custom module gets called each frame, and can combine the data from the various
# intermediate data structures that the predefined modules provide to compute, e.g.,
# cross-correlation.
custom = MyCustomDataCombiner(distances.data['dist'], angles.data['angle'])
runner.add_frame_callback(custom.process_frame)

# This reads in the trajectory (possibly, the trajectory could also be specified here for additional flexibili
ty),
# and does all the specified analyses with a single pass.
runner.run()
```

On top of this, we can then add helper functions to, e.g., make it easy to provide some of the input parameters from argparse etc.

This is of course more difficult to implement than blind wrapping of every single C++ class, but this also hides all C++-specific things that the Python code does not really need. For example, there is no point for the Python code to use the Options module for providing command-line parameters when Python already has argparse. And a large fraction of the methods in TrajectoryAnalysisModule are irrelevant for serial processing (which anyways will be the desired mode for Python). It will also be trivial to add a 'gmx trajectory' tool that just writes out the coordinates for selections, which allows many types of custom analyses without exposing anything more than those classes shown above.

This basic structure can then be extended further by, e.g., exposing raw selection support or various AnalysisDataModuleInterface subclasses to the custom modules, but none of this is requires for a wide variety of analyses. There is quite a limited set of C++ classes that actually need to be wrapped in order to implement the above basic syntax (and some additional C++ classes written).

#14 - 12/14/2014 10:34 AM - Alexey Shvetsov

You propose just wrapping of existing tools with ability to combine their output, while my idea was to allow to write new tools in python using gromacs api. This will even allow to call some specific stuff from mdrun (e.g. energy minimization) if it will be wrapped, so it is a little bit longer but much more interesting for users. They will be able to write their own data processing stuff.

#15 - 12/14/2014 12:40 PM - Teemu Murtola

Alexey Shvetsov wrote:

> You propose just wrapping of existing tools with ability to combine their output, while my idea was to allow to write new tools in python using gromacs api.

You should read what I write, and put some thought into it before replying... I said that implementing gmx trajectory and/or extending the API to support, e.g., selections, allows the user to do (nearly) arbitrary analysis. But unlike in your proposal, such complexity and additional maintenance is not necessary to get significant added value.

> This will even allow to call some specific stuff from mdrun (e.g. energy minimization) if it will be wrapped, so it is a little bit longer but much more interesting for users. They will be able to write their own data processing stuff.

Let's keep the discussion focused. There is nothing in your proposal that directly supports this, and there is nothing in my proposal that would make it more difficult to implement such support. This is simply irrelevant for discussing how to structure the basic analysis API.

Also, let's be realistic. There has been discussion on having a wider API to cover, e.g., mdrun functionality, for at least as long as I've been involved with Gromacs (which is quite long), and nothing has happened. Throwing in a student with only little knowledge of Gromacs is not going to make the code reorganize itself to produce a stable API with a design that is maintainable in the long term. The only part of the Gromacs C/C++ API where a significant goal for the design has been external exposure/stability is some parts of the C++ analysis API, and even there the lack of resources means that this may not really stay that stable all the time.

#16 - 07/16/2015 12:11 AM - Alexey Shvetsov

We added simple pipeline with PyAPI

https://biod.pnpi.spb.ru/gitweb/?p=alexxy/gromacs.git;a=commit;h=1241cd15da38bf7afd65a924100730b04e430475

So now its quite simple to process trajectory with multiple existing c++ and py modules in one read =)

#17 - 09/04/2016 02:00 AM - Roland Schulz

- Related to Task #2045: API design and language bindings added

#18 - 03/08/2017 05:49 PM - Gerrit Code Review Bot

Gerrit received a related patchset '38' for Issue #1625.
Uploader: Alexey Shvetsov (alexxy@omrb.pnpi.spb.ru)
Change-Id: gromacs~master~I27da34dc5e04a253cb88f0e4fc0dd6f48ad9c2ca
Gerrit URL: https://gerrit.gromacs.org/4269

#19 - 03/02/2019 01:44 AM - Eric Irrgang

This issue seems to be related to running Python code from GROMACS, whereas issues descended from #2045 include provisions focused on running GROMACS from Python. However, the API for such language bindings includes the ability to pass executable function objects, which can refer to objects owned by the parent Python interpreter.