Molecular Dynamics

Alexandra Pappas
March 2015

Contents

Introduction

Starting from a Python implementation of a simple molecular dynamics code, I incrementally turned it into a well-engineered high-performance implementation with features that include:

- **C++ implementation**
- **Python interface for scripting and control**
- **Checkpoint / restart capability**
- **Build system**
- **Regression tests**
- **Visualization integration**
- **Performance profiling and optimization**

Using object-oriented programming, I simulated the interaction of a fixed number of particles in a box with periodic boundary conditions and recorded the properties of these interactions.

The Porting Effort

Description

I used SWIG (Simplified Wrapper and Interface Generator) to implement a hybrid Python/C++ architecture. A hybrid architecture has the advantage of being able to write portions of the code that are not critical for performance in a high level language like Python thereby enhancing programmer productivity. It also allowed me to use each language to my advantage in other ways. For example, the processing of strings is much easier to complete in Python as opposed to C++, so those operations were moved accordingly.

Considerations and Challenges

C++ code needed to be implemented for different data types, as C++ specifies both single and double precision data types.

The original Python implementation was not particularly well written, so it was necessary to improve the algorithm efficiency of certain functions (e.g. InternalForce)

Some Python features are implemented differently in C++. For example, the rounding function has a different behavior between the two languages.

Code Design and Architecture

TOP DIRECTORY

build
CMakeLists.txt
include
src
tests

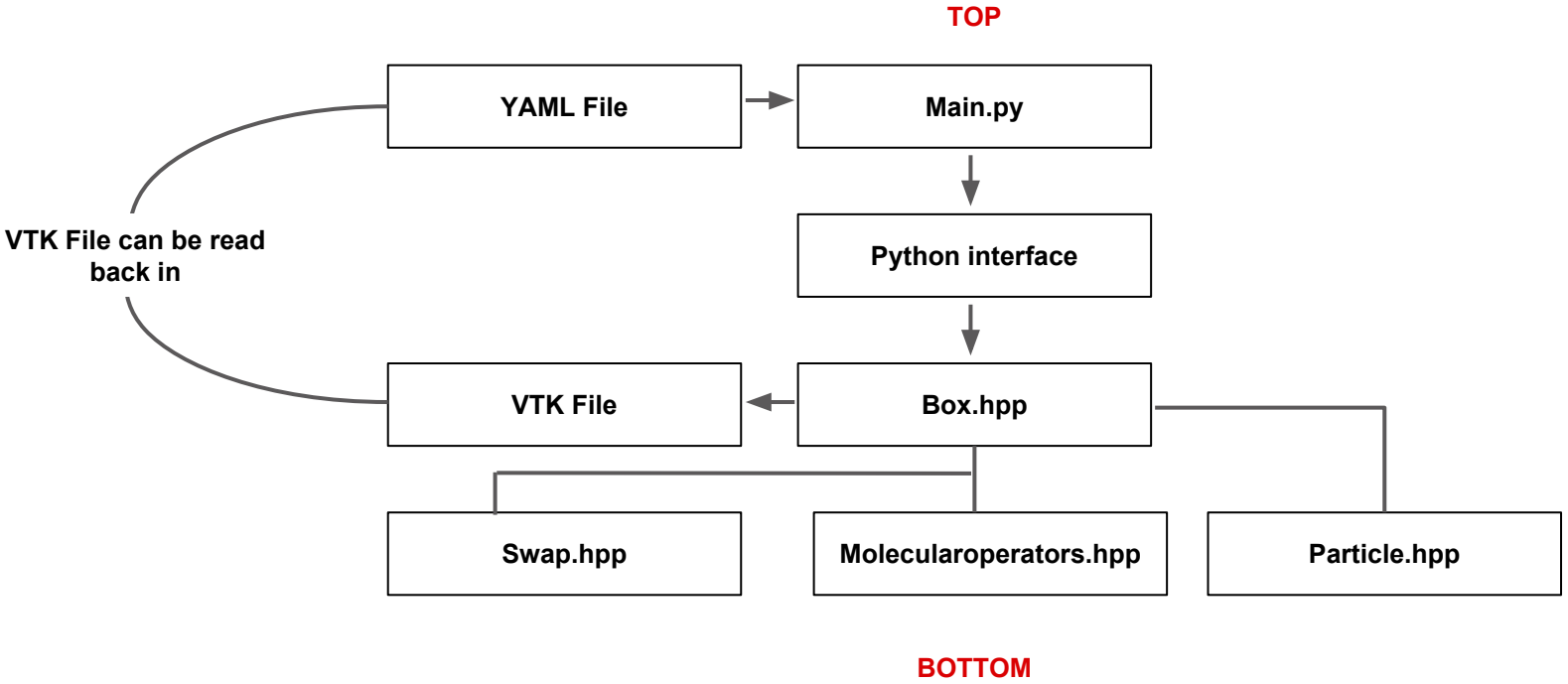
SUBDIRECTORIES

./include/box.hpp
./include/molecularoperators.hpp
./include/particle.hpp
./include/swap.hpp

./src/exec/CMakeLists.txt
./src/exec/box.i
./src/exec/main.py
./src/exec/solution0.dat
./src/exec/solution0_512.vtk
./src/exec/solution0_64.vtk
./src/exec/solution0_a.yaml
./src/exec/solution0_b.yaml
./src/exec/solution0_c.yaml
./src/exec/solution0_d.yaml
./src/exec/solution0_e.yaml
./src/exec/solution0_f.yaml
./src/exec/solution10_64.vtk
./src/exec/swap.cpp

./tests/CMakeLists.txt
./tests/FindCPPUnit.cmake
./tests/tests.cpp

Code Design and Architecture



Checkpoint/Restart

The original C++ code with a corresponding main.cpp ran 25 time steps, printed the energies at each time step, and exited. I then implemented checkpoint and restart capabilities (see right) using command line flags. For the final implementation with both Python and C++ code, the interface for specifying input and solution files moved from the command line to a YAML file.

CHECKPOINT

adds the ability to specify the number of time steps and write the solution file given in a file in YAML format

RESTART

adds the ability to provide an input (checkpoint) file and run additional time steps

VTK Binary Format

Output files are written in VTK "legacy" format, which allows for particle visualization with the VisIt program. The VTK format for the output files is shown and described at right.

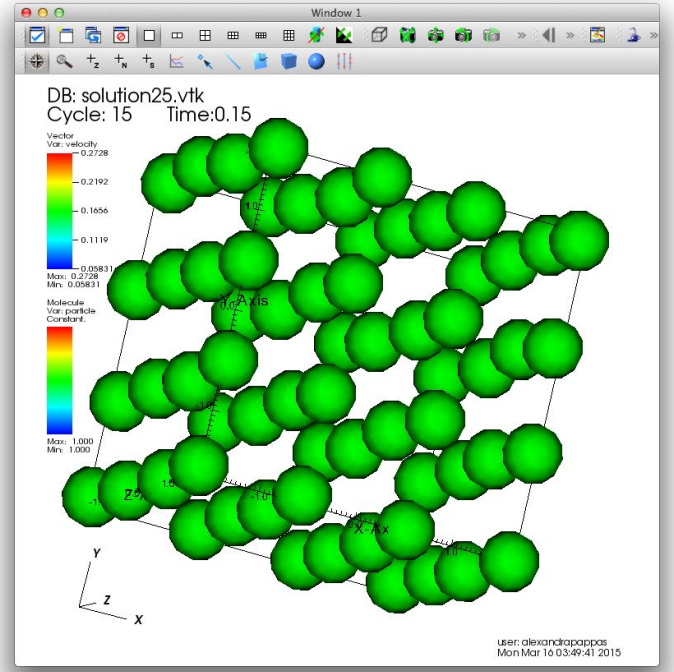
```
# vtk DataFile Version 3.0
vtk output
BINARY
DATASET POLYDATA
FIELD FieldData 2
TIME 1 1 double
0
CYCLE 1 1 int
0
POINTS 3 double
01.58  -1.58   -1.58
POINT_DATA 3
SCALARS particle double
LOOKUP_TABLE default
1  1  1
VECTORS velocity double
-0.14  -0.14  0.12
```

Header
Title
Format: (BINARY)
DATASET <type>

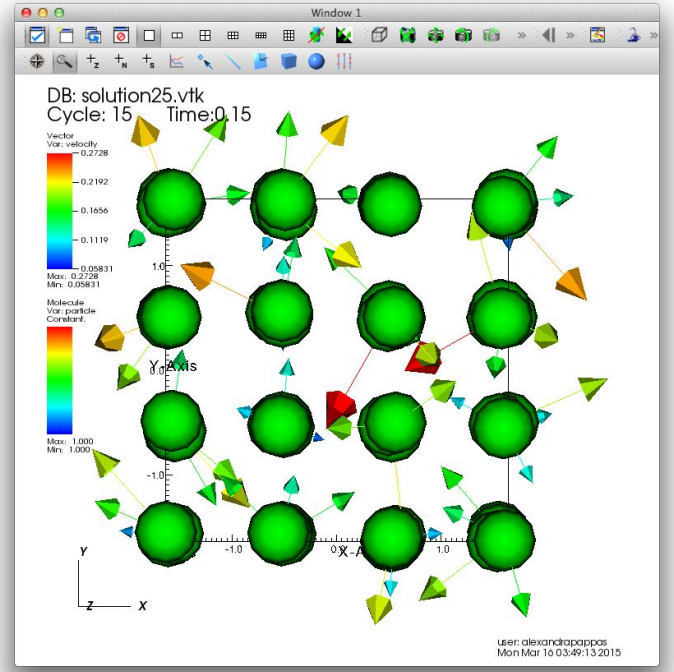
Time Information
(needs to appear directly after DATASET)

POINTS <npoints> <data type (e.g. float or double)>
<first point> <second point> …
Scalar data with name 'element' and data type 'float'
Use default lookup table for converting numbers into colors
Three scalar values
VECTORS <description (e.g. velocity)> <data type>
<first velocity> <second velocity> ….

Visualization



Molecule plot



Molecule plot with vector overlay

Code Testing

I implemented a basic regression test in a CppUnit framework to check the test the checkpoint/restart capability of the program. Regression testing is the comprehensive testing of the application to make sure it has no defects.

TEST DESCRIPTION

> Read in a vtk file corresponding to the correct solution for time step 0
> Run 10 steps
> Read in a vtk file corresponding to the correct solution for time step 10
> Run 10 steps

LIMITATIONS

As currently implemented, the test does not check for all the file input formats allowed in the program, or for a random initialization of the system. It also doesn't perform any unit tests, perhaps to see that positions and velocities are being correctly initialized. Lastly, the tests do not check the Python/C++ interface. Ideally, this is something that would be automated.

Building and Running

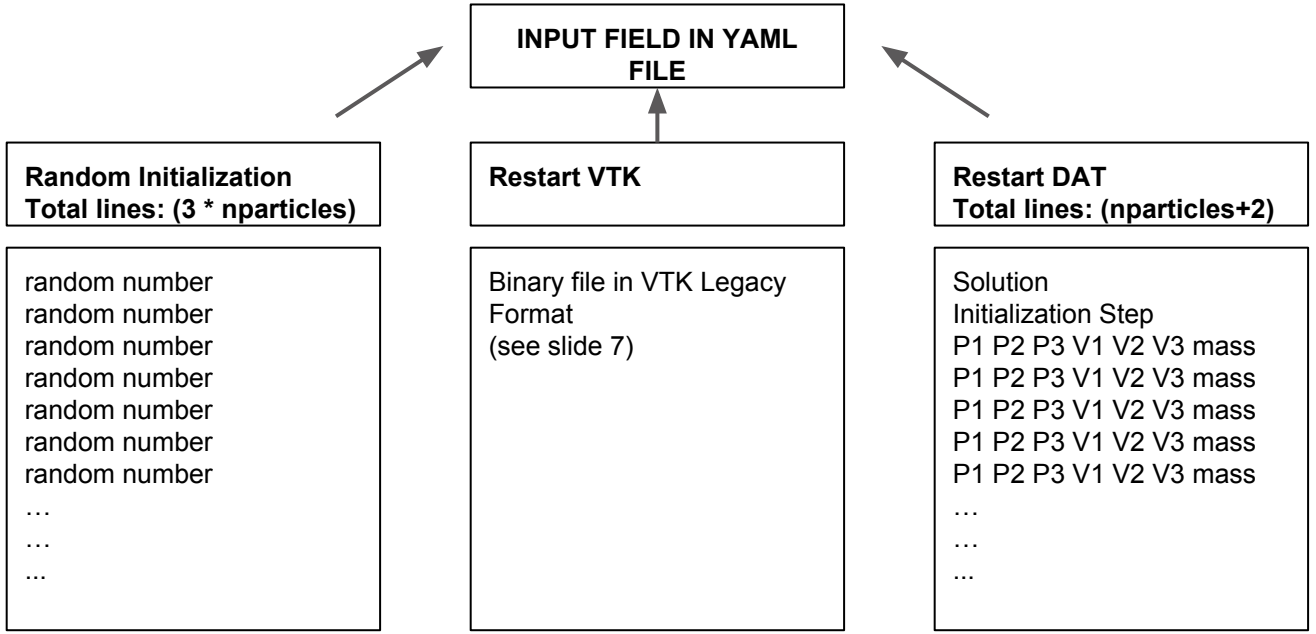
Main.py accepts a single command line argument that specifies that name of an input configuration file in the YAML format. To run from an initial state (not a random initialization file) do not include a field for input. Other fields that can be excluded include output file and mass.

```
precision: double
mass: 48.
nparticles: 64
boxsize: 4.2323167
temp0: 0.728
stepsize: 0.01
nsteps: 500
input: solution0_64.vtk
savefreq: 10
output: solution?????.vtk
```

```
precision: double
nparticles: 64
boxsize: 4.2323167
temp0: 0.728
stepsize: 0.01
nsteps: 500
savefreq: 10
```

Building and Running

Acceptable input file formats for YAML file



Building and Running

I used CMake for an out of place build, and enabled compiler warnings and release mode by default. To build and run, type the following commands from the top of the directory

$ mkdir build

$ cd build

$ cmake ..

$ make

$ cd src/exec

$ Python main.py <yaml file>

I also provided a number of YAML files that test certain program conditions. Again, ideally this would be automated in some way.

solution0_a.yaml (double data type and vtk input file)

solution0_b.yaml (dat input file)

solution0_c.yaml (float data type)

solution0_d.yaml (512 particles and no input or output specified)

solution0_e.yaml (removes necessary arguments)

Performance Optimization

I profiled my program for two tests, one where the box contains 64 particles and a certain box side length and another where the box contains 512 particles and a different box side length. Each case was run for 25 steps. Using cProfile allowed me to compare performance between the two tests cases and see where time is most spent in the program. For the two tests cases, I observed a 100x speedup and 450x speedup, respectively.

	Time for Python implementation (in seconds)	Time for Python/C++ implementation (in seconds)
Test case 1 Particles: 64 Box side length: 4.2323167	17.380	.165
Test case 2 Particles: 512 Box side length: 8.4646344	910.313	2.005

Each test is run for 25 steps

Performance Optimization

Expensive Methods/Computations	Implemented optimizations
PutInBox GetForce Distance InternalForce Use of rounding	My first C++ implementation used vectors, not arrays, which was suboptimal because of the layout of vectors in linear memory, which is not contiguous. Using arrays for properties that had fixed size (e.g. positions, velocities) was the solution. I eliminated/minimized pass by copy where appropriate. I checked the algorithm efficiency of expensive methods. One implementation that I changed was the implementation of InternalForce, since it was doubly computing the force for particles. I removed redundant computations often by making sure that computations were not being executed in loops unnecessarily. If a number was always constant throughout the loop, I brought it outside. I replaced expensive computations (raising a number to a power) A further optimization that could be implemented would be to improve or remove the need for rounding.

Summary

Transforming the architecture of a program can seem rather complex at first. However, with a structured approach it can help you achieve significant design and performance gains.