CME 212: Final Project

Molecular Dynamics Simulation Module

Raunak Borker

Outline

- Porting from Python to C++
- Design and Architecture of Code
- Design Improvements
- Code Features
- Tests
- Usage Guide
- Performance Analysis and Optimization
- Summary

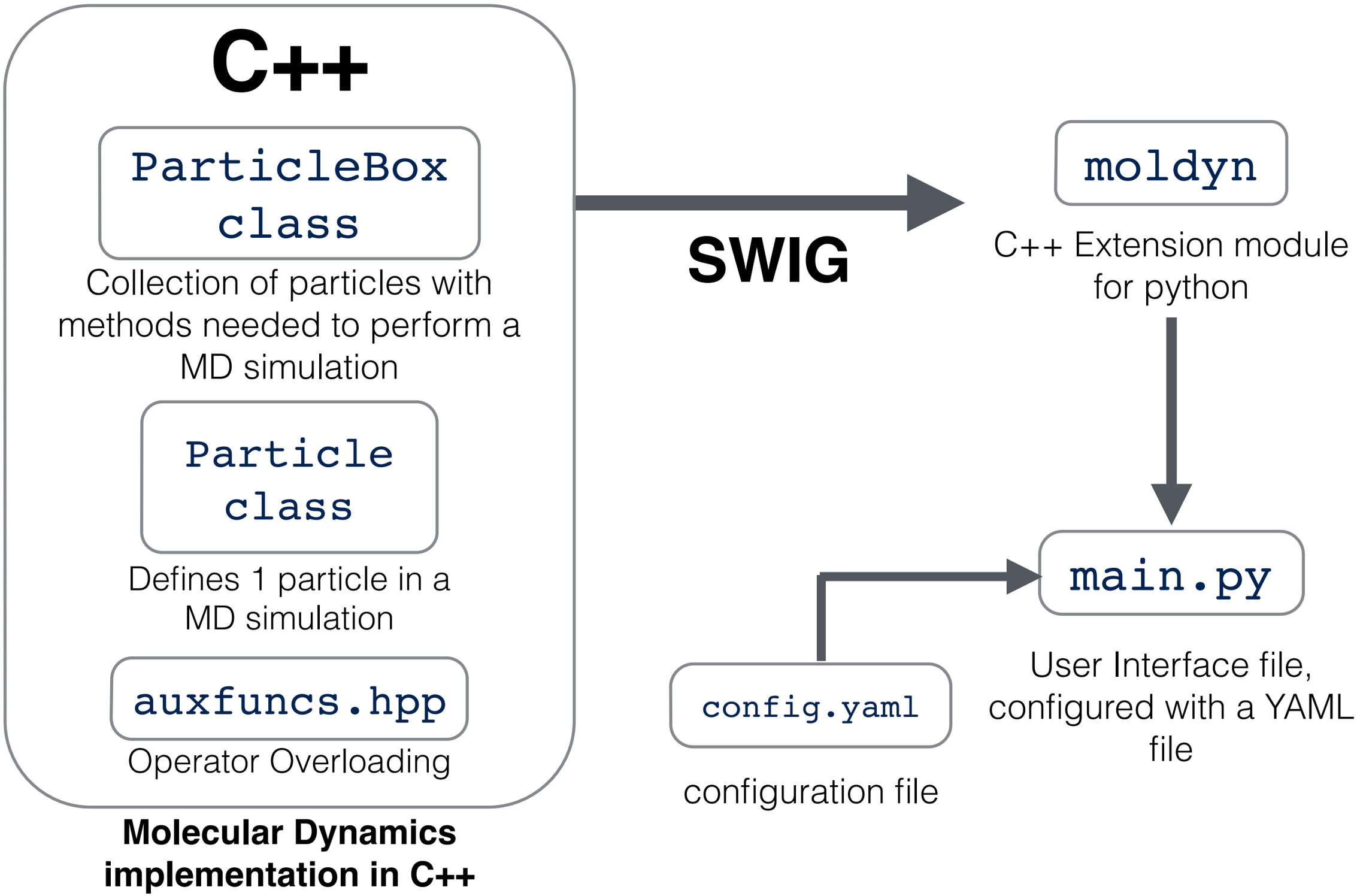
Python → C++

- C++ implementation uses a OOP design but has methods similar to the functions implemented in Python

- *Challenges*
  - Python implementation used a lot of cool numpy functions applied to arrays. In C++ these had to be explicitly implemented. e.g. sum, round, array arithmetic operations.

  - To match results between the 2 implementations the behavior of numpy.round needed to be replicated. The usage of nearbyint function in C++ provides same results although nearbyint varies slightly to numpy.round in a special case.

Design and Architecture



Design and Architecture

- 2 Classes: `Particle` and `ParticleBox` templated to handle single and double precision computations

    - `Particle`: defines a single particle in a Molecular Dynamics simulation, has properties of `mass`, `position`, `velocity` and `acceleration`. Vectorial properties are defined as `std::array`'s.

    - `ParticleBox`: defines the simulation box which holds a vector of particles.

        - The constructor of `ParticleBox` sets up a MD simulation and can be initialized by providing the *mass of a particle, initial temperature, box width, step size, number of particles or using a restart file as described later*.

        - `ParticleBox` has public methods to `Solve` (advance in time), `SaveOutput` (write data to disk) and `OutputEnergy` (return current energy state).

Design and Architecture

- `auxfuncs.hpp` defines a set of operator overloading functions for the `std::array`, used to simplify writing and reading of methods in the `ParticleBox` class.

- The `ParticleBox` class in then compiled as an extension module labeled `moldyn` to be used in Python. This is achieved using SWIG.

- `main.py` defines the user interface to run Molecular Dynamics simulations using the module `moldyn`.

- Simulation parameters are inputed using a YAML configuration file.

Design Improvements

- *Solve*: Internal force was being called twice at every time step, when involved repeated calculations since the position loop recomputed force from the previous steps velocity loop.

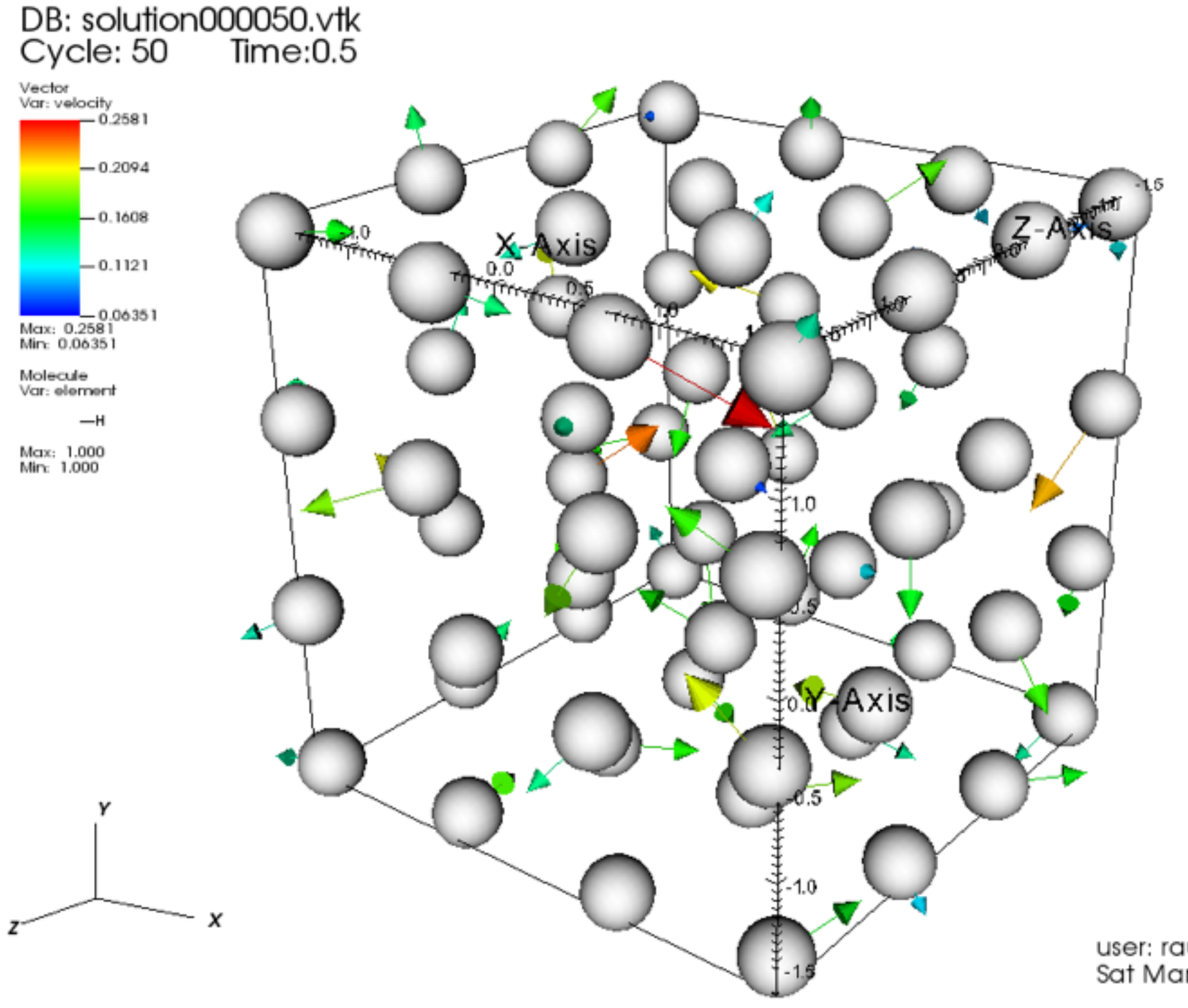
- *Force computation*: every particle contributes to every other particle. Python code performed n^2 computations. Reduced by a factor of 2, by computing the interaction between particle i and j only once.

- *PutInBox*: python code used the round function every time the PutInBox was called. In the current implementation it is called only if the absolute value of distance between particles is more than half a box width.

Code Features

- *VTK binary output:*
  - The code is equipped with the capability to save output data as a VTK binary file
    - This can be used for visualization purposes as has been shown on the next slide
- *Checkpoint/Restart:*
  - The same VTK file can also be used to restart a simulation from a checkpoint

Code Features



- Mass = 48.0
- T0 = 0.728
- 64 particles
- L =4.2323

*The arrows indicate the velocity vectors which are color coded by their magnitude

Tests

- Test1: Uses a predetermined initial state (same as python code) and advances the solution by 10 steps. The energies computed after 10 steps are compared against those obtained from the python implementation.

- Test2: Reference solution file after 10 steps is used to advance another 10 steps and energies compared against energies from python after 20 steps. (this tests restart/ checkpoint functionality from vtk file)

- Test3: Similar to test1 but now compares after 1000 steps to make sure that round off errors don’t hinder accuracy till these many steps.

Tests Drawbacks

- The tests only compare energies and don't really compare position, velocity and acceleration. Which may not be a full proof way of comparing results, since a derived quantity being equal doesn't necessarily mean the base quantities are as well.

- The comparison is being made against an existing code. So all that is guaranteed is that it produces same results as the python code, which doesn't necessarily mean it produces correct molecular dynamics results, unless the python code is rigorously tested. For that another test case needs to be written.

Building the code

All paths in this usage guide will be relative to the project source directory, which could be something like

```
cd ~/CME212/FinalProject/
```

➡ `mkdir build`

➡ `cd build/`

➡ `cmake ..`

➡ `make`

➡ `make test` (will run the test suite, should indicate no failure)

Running the code

Navigate to the `src` directory in `build`

➡ `cd build/src/` if already in `build` type `cd src/`

This directory will have the `config.yaml` file, which is used to setup the simulation parameters.

- Available Parameters:
  - `input`, `output`, `nparticles`, `mass`, `temp0`, `savefreq`, `nsteps`, `boxsize`, `stepsize`, `precision`

Configuration file

- `input -` name of the *input file in vtk format with it's path if not in* `build/src/` and could be a previously saved output file. (optional parameter) If not specified or left empty the simulation will default to running using a *random initial state*.

- `output -` naming format to be used for output file. The convention is a *prefix* followed by *?* followed by a dot and the file extension.
  For eg: solution?.vtk, where *prefix*-solution, *extension*-vtk. Anything between ? and . and including ? will be replaced by a 6 digit number indicating the step after which the solution was saved. (optional parameter) If not specified solution?.vtk will be used. Example of this will be seen on the next slide.

- `savefreq -` frequency after which output should be saved to disk (**necessary parameter**).

- `nsteps -` number of steps to be advanced in time (**necessary parameter**).

- `precision -` Can be specified as 'double' or 'float'. If not specified will default to 'double' (optional parameter). Precision specified has to match restart/input file if one has been specified.

The following parameters are **necessary** if an input file is not specified, but will be ignored if an input file is provided.

- `nparticles -` number of particles to be used in the simulation, `mass -` mass of a particle, `temp0 -` Initial temperature, `boxsize -` Box width, `stepsize -` time step.

Running the code

Navigate to the `src` directory in `build` if not already in it and run the `main.py` using `python` with a necessary input argument of the configuration file

➡ `cd build/src/`

➡ `python main.py config.yaml`
eg: usage syntax is
`python main.py <yaml file name>`

➡ `ls`
`solution000010.vtk solution000020.vtk solution000030.vtk …`

Output will be saved in the same directory based on the output name convention specified in the configuration file. For the example shown above the naming convention is the default one with a save frequency of 10.

Performance Analysis and Optimization

- The Design improvement shown earlier are mainly the algorithmic improvements which end up providing more than **5x** improvement.

- It was observed using cachegrind that for the tested cases of particles < 512, the cache performance was very good, giving L1 miss rates < 0.1%.

- Hence the remaining effort was directed to optimize the compute performance. gProfile/OProfile were employed for this purpose. One of the choices made previously for this purpose was the use of std::array for all vectorial quantities as the spatial dimension is fixed to 3 and hence array are more suitable than std::vector.

Performance Analysis and Optimization

- In addition all methods are passed arguments by reference to avoid the creation of copies which can hinder performance.

- The analysis of the previous implementation gave the following gProfile:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.08      0.05     0.05    12800     3.91     3.91 ParticleBox<double>::InternalForce(unsigned int)
  0.00      0.05     0.00     1400     0.00     0.00  void std::vector<std::array<double, 3ul>,
std::allocator<std::array<double, 3ul> > >::_M_emplace_back_aux<std::array<double, 3ul>
>(std::array<double, 3ul>&&)
  0.00      0.05     0.00      101     0.00     0.00  ParticleBox<double>::ComputeEnergy(std::array<double,
3ul>&)
  0.00      0.05     0.00        1     0.00     0.00  ParticleBox<double>::ParticleBox(std::string)
  0.00      0.05     0.00        1     0.00     0.00  main
```

- The algorithmic improvements on internal force reduced the 100% to 15% and compute energy showed up alongside (Result not attached due to slide constraint)

Performance Analysis and Optimization

- Further optimization was performed in computing the distance. *Distance*: only even powers are used in the algorithm, hence distance square is stored, avoiding the computation of a square root.

- The usage of power function was avoided totally as powers are always small integer values which be be hand written.

- Performance was compared on the `corn04` machine

Number of Particles	**Number of steps**	**Time in seconds**		**Speedup**
		Python	**C++**	
64	5000	1404	0.497	**2824x**
512	100	1829	0.530	**3450x**

- 'Time' here refers to the time required to advance the required steps and is measured around the solve function. It doesn't involve the reading and writing of vtk files. Involves the writing to screen of computed energies in both python and C++. All comparisons have been performed using double precision arithmetic.

Summary

- We now have a C++ extension module that can be used through python to do Molecular Dynamics simulations ~3000 times faster in double precision.

- The speed up is not a fair comparison of C++ and python (the languages themselves), since some of the contribution to the speed up was algorithmic.

- The code needs to be tested more comprehensively if indeed it is to be used for simulation purposes.