The OpenMM API

Peter Eastman
OpenMM Workshop, Sept. 6, 2012
The OpenMM Architecture

- High Level Python Code: Application Layer
- Public Interface: OpenMM Public API
- Platform Independent Code: Implementation Layer
- Platform Abstraction Layer: OpenMM Low Level API
- Computational Kernels: CUDA/OpenCL/MPI/etc.
Why Use the API?

• Application Developers
  – This is how you use OpenMM within your programs

• Chemists/Biologists
  – It’s available to your control scripts… if you want it
  – There’s a lot you can do with it!
Goals of the API

• Simple
  – Easy to learn
  – Easy to use

• Extensible
  – Add new force fields, integration methods, etc.
  – Support new hardware platforms

• Can be implemented efficiently on a variety of hardware platforms

• Easy to incorporate into existing codebases
Choice of Language

• The OpenMM API is written in C++
• API wrappers are provided for Python, C, and Fortran
  – Provide access to most features
  – Plugins can only be written in C++
Public API Classes (1 of 3)

• System
  – A collection of interacting particles
  – Defines the mass of each particle
  – Specifies distance constraints
  – Contains a list of Force objects that define the interactions

• Context
  – Contains all state information
    • Positions, velocities, other parameters
Public API Classes (2 of 3)

• Force
  – Anything which affects the system’s behavior
  – Forces, thermostats, barostats, etc.
  – A Force may:
    • Apply forces to particles
    • Contribute to the potential energy
    • Define adjustable parameters
    • Modify positions, velocities, and parameters at the start of each time step
Public API Classes (3 of 3)

• Integrator
  – Advances the system through time
  – Both fixed and variable step size integrators are supported

• State
  – A snapshot of the state of the system
  – Immutable (used only for reporting)
  – Creating a State is the only way to access positions and velocities
  – Can optionally include forces and energies
Example

• Create a System

```python
system = System()
for particle in particles:
    system.addParticle(particle.mass[i])
for constraint in constraints:
    system.addConstraint(constraint.atom1, constraint.atom2, constraint.distance)
```

• Add Forces to it

```python
bondForce = HarmonicBondForce()
for bond in bonds:
    bondForce.addBond(bond.atom1, bond.atom2, bond.length, bond.k)
system.addForce(bondForce)
# ... add Forces for other force field terms.
```
Example (continued)

• Simulate it

integrator = LangevinIntegrator(297.0, 1.0, 0.002)  # Temperature, friction, step size
context = Context(system, integrator)
context.setPositions(positions)
context.setVelocities(velocities)
integrator.step(500)  # Take 500 steps

• Retrieve state information

state = context.getState(getPositions=True, getVelocities=True)
for position in state.getPositions():
    print position
Platforms

• The API defines the *interface*
• A Platform provides the *implementation*
• Available Platforms:
  – Reference
  – OpenCL
  – CUDA
The Platform API

• Select a Platform to use

```python
platform = Platform.getPlatformByName("OpenCL")
context = Context(system, integrator, platform)
```

• Check what Platform is being used

```python
print context.getPlatform().getName()
```

• List available Platforms

```python
for i in range(Platform.getNumPlatforms()):
    print Platform.getPlatform(i).getName()
```
Documentation
C++/Python Differences

- Context.getState() uses boolean arguments instead of flags

C++:
```cpp
context.getState(State::Positions | State::Velocities);
```

Python:
```python
context.getState(getPositions=True, getVelocities=True)
```
C++/Python Differences, cont.

• Multiple return values are returned directly, not as arguments

C++:
```cpp
int particle1, particle2;
double length, k;
f.getBondParameters(i, particle1, particle2, length, k);
```

Python:
```python
(particle1, particle2, length, k) = f.getBondParameters(i)
```

• Quantities have explicit units
Why Units?

• Many different units are common in MD
  – Time in ps or fs?
  – Distance in nm or Angstroms?
  – Energy in kcal/mol or kJ/mol?
• You *will* make mistakes!
• This *will* produce bugs!
...and you’re not alone!

- Mars Climate Orbiter
  - Crashed into Mars in 1999
  - Cost $125 million

- Air Canada Flight 143
  - Ran out of fuel in midair in 1983

Both caused by errors in unit conversions
Units in OpenMM

• Just multiply each value by its units

```python
>>> size = 5*nanometers
>>> print size
5 nm
>>> accel = 9.8*meters/second**2
>>> print accel
9.8 m/(s**2)
```

• Can convert to any compatible unit

```python
>>> print size.in_units_of(angstroms)
50.0 Å
>>> print size.value_in_unit(angstroms)
50.0
```
Units in OpenMM, continued

• Conversions happen automatically when doing math

```python
>>> print 5*nanometers+25*angstroms
7.5 nm
>>> print 5*nanometers+25*picoseconds
Traceback (most recent call last):
...
TypeError: Cannot add two quantities with incompatible units "nanometer" and "picosecond".
```

• Can apply units to lists, tuples, and arrays

```python
>>> x = (1.0, 1.5, 0.0)*nanometers
>>> positions = state.getPositions().in_units_of(angstrom)
```
Units in OpenMM, continued

• Units are optional (but recommended!) on input values
• Output values always have units
• Default OpenMM units:
  – nm, ps, K, amu (g/mole), kJ/mole, e
  – These form a consistent unit system!
Which Language to Use?

- **Python**
  - Faster development
  - Interactive mode for experimenting
  - Explicit units

- **C++**
  - Faster execution
  - Easier to call from other languages
  - Can write plugins