Molecular (and Brownian) Dynamics using MMTK
An Introduction to Python and OOP for Scientists

P Sunthar

Department of Chemical Engineering
Indian Institute of Technology, Bombay
Mumbai 400076, India
P.Sunthar@iitb.ac.in
Visiting Academic
Department of Chemical Engineering
Monash University, Melbourne, Australia

26 Oct 2009
Outline of MMTK Introduction Modules

1. Introduction to MMTK and Python
2. MD/BD 101: Simple Scripts
3. MMTK Internals: MD Example
4. MMTK Extension: BD Example
Outline of this Module

1. Introduction to MMTK and Python
   - Molecular Modelling Tool Kit
   - Python
Overview of Molecular Modelling Tool Kit

- A software library to perform Molecular Dynamics
- Written in Python and C Programming Languages ($\approx 1 \times 10^5$ lines)
- Simple python interface to setup problem, analyse, and display results (about 20 lines).
- Written by Dr Konrad Hinsen, Center for Molecular Bio-physics, CRNS, Orléans, France.
Capabilities of MMTK

- Molecular Dynamics (Repository of chemical interaction energies)
- Biomolecular dynamics (Proteins, Nucleic acids)
- Infinite or Periodic domains
- Standard Force Fields (LJ, ES, Amber)
- Ewalds Sum and Fast Multipole (DPMTA)
- Normal mode analysis (Macromolecules)
- Visualisation
Why use MMTK?

- Simple scripting (like CHARMM) Open Source (like GROMACS)
- Extension to other Forcefields and Algorithms
- Python based (Object Oriented):
  - Rapid Development and Testing
  - Several related python libraries available
- Interface with C, provides comparable speed as C/Fortran
- Large Monolithic programs difficult to extend
Overview of Python

- Interpreted (Matlab)
- Object Oriented (C++, Java)

Modules:
- Numerics
  - Numerical Analysis
  - Vectors, Tensors
  - Linear Algebra
- Hi Performance
  - C/Fortran Calls
  - MPI Threads
- Visualisation
  - Plotting
  - 3D Models
  - Animation
- Text Processing
  - Splitting
  - Searching
Interpreted Language

- Interpreted, as against Compiled
- Rapid code development (Easy to debug, and test)
- Slow execution of compute intensive operations
- Byte-compiled: shortens module loading (not execution) time
- Interpreter started by `python` command and `>>>` prompt
- Portable (across platforms)

```
$ python
Python 2.6.2 (release26-maint, Apr 19 2009, 01:56:41)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>> foo = 1.0
>>> a = foo + 2
```
Interactive Interface: ipython

- Provides useful information on variables (objects), help
- Useful for learning, developing
- Similar to Mathematica interface

IPython 0.9.1 -- An enhanced Interactive Python.

help -> Python’s own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

In [1]: foo = 1.0
In [2]: foo
Out[2]: 1.0
In [3]: ?foo
Type: float
Base Class: <type 'float'>
String Form: 1.0
Namespace: Interactive
Docstring:
   float(x) -> floating point number
   Convert a string or number to a floating point number, if possible.
Dynamic Types

- No pre-declaration required (as in C, Fortran)
- python *introspects* to find the type from RHS
- only RHS must be pre-defined

```python
>>> foo = 1.0    # foo initiated as floating point type
>>> bar = 2      # bar initiated as integer type
>>> foo = 'string'  # foo becomes a character string
>>> from MMTK import *  # import some definitions
>>> bar = Atom('C')  # object of 'Atom' class, element is Carbon
```

- *Atom* class, is defined by MMTK
- object (LHS): integer, float, string, Atom (type or class)
- object $\leftarrow$ class
Lists, Tuples, Dict I

- Sequence of objects of any (mixed) type; generic array
- Tuple as in: single, double, triple, quadruple, …, $n$-tuple

**Assignment**

**List:**
```
mylist = [1, 'two', 3.0]
```

**Tuple:**
```
mytupl = (1, 'two', 3.0)
```

**Dict:**
```
mydict = {'first':1,
          'second': 'two', 'third': 3.0}
```

**Retreival**

**List:**
```
mylist[0]
```

**Tuple:**
```
mytupl[0]
```

**Dict:**
```
mydict['first']
```
Lists, Tuples, Dict II

- Colon operation \((\text{begin}:\text{end})\), begins at zero (C convention)!
  
  **List:** `mylist[0:1]`
  
  **Tuple:** `mytupl[0:1]`
  
  **Dict:** Not possible

- Adding more objects
  
  **List:** `mylist.extend([4])`
  
  **Tuple:** Not possible
  
  **Dict:** `mydict.update({"fourth":4})`
  
  `mydict["fourth"] = 4`

- Dicts are associative arrays: `{key: value}`

- Items in dict are obtained by `mydict.items()`, which is a list of tuples.
  
  `[("second", "two"), ("third", 3.0), ("first", 1)]`
Lists and Tuples are different

- Lists are *mutable* (as in mutation); Tuples are *immutable*, constants once created
- Tuple objects are fixed length, no extension possible
- To change, overwrite Tuple object (Dynamic typing)
- Tuples are faster (for passing around fixed objects)
- Eg: Constituents of water cannot be changed when passed around
- Eg: Components of additive can be modified.

```python
water = (Atom('O'), Atom('H'), Atom('O'))
from MMTK.Proteins import Protein
additive = [water, Protein('insulin')]
additive.extend([Atom('C')])
thawed_water = list(water)
frozen_additive = tuple(additive)
```

# cut and paste? make sure ' is keyboard single quote character
• `mylist[i]` gives the \((i-1)\)-th element: can be used generate a list through a for loop

• **In line** `for` **loop**

  ```python
  >>> range(10)
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  >>> even = [2*x for x in range(10)]
  >>> even
  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
  >>> [ob for ob in water]  # list of objects in water
  [Atom oxygen, Atom hydrogen, Atom oxygen]
Multiline Loop, Blocks, and Indents

>>> for i in range(2):
    ...
    print i
    ...
0
1

- >>> and ... are prompts from python (Don’t type!)
- A Multiline block has : (colon) on the first line
- **Amount of left indent** (space) differentiates a block (there are no { } or BEGIN or END)
- Unusual for a programming language. But enforces readability!
Executing a script

- In a script, multi level blocks look like:

```python
for j in range(4):
    for k in range(5):
        print '    inner block', k
    print '    outer block', j
print 'end of loops'

>>> execfile('blockindents.py')
inner block 0
inner block 1
inner block 2
inner block 3
inner block 4
outer block 0
inner block 0
inner block 1
inner block 2
inner block 3
inner block 4
outer block 1
```
Object Oriented Program (OOPs!): Terminology

- Paradigms: Procedural vs Object Oriented Programming
- Types: class, object, instance
- Contents: attributes (data), methods (functions)
- Inheritance: base class, derived class
- Privacy: Data Encapsulation
- Life: Constructors, Destructors

Procedural Programming

- Separate Data and Methods (functions, procedures)
- Sequence of function calls
- Limited type (float, int, char) checking $\Rightarrow$ Bugs
**Object Oriented Programming**

- **class**: data and methods
- **object**: instance of a class
- **Everything is an Object**
- **Privacy of data**
- **Objects passed, operated**
- **Strong type checking**
class Complex(object):
    def __init__(self, real=None, imag=None):
        self.r = real
        self.i = imag

    def conjugate(self):
        return Complex(self.r, -self.i)

    def __call__(self):
        print('My pretty look: (', self.r, ',', self.i, 'i)')

__init__() method: a = Complex(2,3) ⇒ a.r = 2 & a.i = 3
self object: the object being operated upon
other def methods: b = a.conjugate()
__call__() method: b(), object called as a function results:
My pretty look: ( 2 , -3 i )
**Inheritance**

```python
class AtomCluster (object):
    def molecularWeight():
    def interAtomicDistanceTensor():

class Molecule (AtomCluster):
    def interAtomicBondTensor():

- **Molecule** is a derived (child) class, from base (parent) class **AtomCluster**
- Shares the methods of base:

  ```python```
  water = Molecule()
  water.molecularWeight()
  ```

These code snippets are not complete.
- **Base classes are abstraction of generic features**
Packages, Modules, or Extensions

- Basic python only provides limited base classes and methods
- Derived classes and methods are defined in Modules
- A collection of modules are Packaged
- Packaged Modules ⇒ Extension (Libraries)
- Name of a module: Folder.Script, where the script resides in Folder/Script.py
- Classes from modules need to be imported

```python
import ModuleName  # a = ModuleName.Molecule()
from ModuleName import *  # a = Molecule()
from ModuleName import Molecule
from ModuleName import Molecule as MyMolecule
alias_a = MyMolecule()  # object of ModuleName.Molecule class
```

- `__init__.py` in a Module folder imports some defaults
numpy: Numerical Analysis Package

- Defines class array, $n$-dimensional array of same type
- Most matrix operations of Matlab can be carried out
- Matlab ⇔ Numpy Syntax Cross reference
  - http://www.scipy.org/NumPy_for_Matlab_Users
  - http://mathesaurus.sf.net/

**Example:**

```python
>>> from numpy import *
>>> from numpy.random import random

>>> a = random((3,3))  # shape has to be a tuple
>>> unity = eye(3)    # only two-D possible

>>> b = random((3,1))

>>> dot(a,b)
array([[ 0.84233898],
       [ 0.22930088],
       [ 0.51795472]])
```
Other Python Resources

- Python for Physics Computing
  http://pentangle.net/python/handbook

- Python for Scientists, from MMTK site (long url here)

- Python for high performance computing
  http://scipy.org/PerformancePython

- Common Python Mistakes
  http://onlamp.com/pub/a/python/2004/02/05/learn_python.html
Discussion Time
2 MD/BD 101: Simple Scripts

- MD Script
- Important Objects in MMTK
- BD Script
MD: Sequence of Events

Aim

Simulate dynamics of Argon molecules at a given temperature and pressure

- Import Modules
- Assign Initial Configuration
  - Simulation Box = universe, and Atoms
  - Positions, Temperature, Pressure
- Define Output trajectory variables
- Equilibration at given temperature and pressure
- Production: MD time step integration

Learn more on MMTK Objects  Skip to Simple BDS
Importing Modules

```python
from MMTK import *
from MMTK.ForceFields import LennardJonesForceField
from MMTK.Environment import NoseThermostat, AndersenBarostat
from MMTK.Trajectory import Trajectory, TrajectoryOutput, LogOutput
from MMTK.Dynamics import VelocityVerletIntegrator, VelocityScaler, TranslationRemover, BarostatReset
import string
from Scientific.IO.TextFile import TextFile
```

- Imports standard definitions in `MMTK/__init__.py`
- Specific ForceField class (for LJ potential)
- Trajectory class stores trajectory data
- Integrator class does the actual integration
conf_file = TextFile('argon.conf.gz')

lx, ly, lz = map(float, string.split(conf_file.readline()))

- map function repeatedly applies string.atof() to the list in the second argument
- A given config file has the box dimensions, followed by x, y, z for each of the atoms

34.7786 34.7786 34.7786
3.37312  2.329858  -2.826008
2.18435  -0.690223  -0.462450
universe = OrthorhombicPeriodicUniverse((lx*Units.Ang, ly*Units.Ang, lz*Units.Ang), LennardJonesForceField(15.*Units.Ang))

while 1:
    line = conf_file.readline()
    if not line: break
    x, y, z = map(string.atof, string.split(line))
    universe.addObject(Atom('Ar', position=Vector(x*Units.Ang, y*Units.Ang, z*Units.Ang)))

- The simulation box object in MMTK is typically called as universe, world, system, etc (any name is fine)
- Periodic in three directions
- All constituents interact through LJ potential
- Atom objects are added to the universe
```
temperature = 94.4*Units.K
pressure = 1.*Units.atm
universe.thermostat = NoseThermostat(temperature)
universe.barostat = AndersenBarostat(pressure)
universe.initializeVelocitiesToTemperature(temperature)
```

- Temperature and Pressure values are set using different algorithms (which are objects!)
- The initial velocities are taken from a Gaussian
Trajectory Variables

```python
trajectory = Trajectory(universe, "argon_npt.nc", "w", "Argon_NPT_test")
output_actions = [TrajectoryOutput(trajectory,
    ('configuration', 'energy', 'thermodynamic', 'time', 'auxiliary'), 0, None, 20),
    LogOutput("argon.log",
    ('time', 'energy'), 0, None, 100)]
```

- **Trajectory** object is responsible saving and retrieving data to disk, and for processing data later.
- **actions** are periodic actions to be taken while integrating
- **TrajectoryOutput** is passed to the integrator to gather data and add it to the trajectory object. (every 20 steps)
- **LogOutput** is similar but writes to a file a few key parameters (every 100 steps)
integrator = VelocityVerletIntegrator(universe, 
delta_t=10*Units.fs)

integrator(steps = 100, 
    actions = [TranslationRemover(0, None, 100), 
               VelocityScaler(temperature, 
                             0.1*temperature, 0, None, 100), 
               BarostatReset(100)] + output_actions)

integrator(steps = 2000, 
    actions = [TranslationRemover(0, None, 100)] 
               + output_actions)

trajectory.close()

- integrator is an object of Integrator class, and defines the time integration algorithm for MD.
- integrator() is a executes the __call__() of the Integrator class, with specified periodic actions.
- First: Equilibration, Second: Production
- Closing Trajectory object ⇒ file to disk
**Atoms, Molecules, Proteins ... Objects**

- **Basic unit** is `ChemicalObject` **which can be a**
  - Atom, Molecule, Complex, Protein, Group

- Details available in **module reference**

- Elementary properties of Atoms, Proteins, and Peptides predefined in package database

- PDB data can be imported

- Provides `MoleculeFactory` module to construct and define arbitrary molecule chemistry

- Methods(): constituents list, centre of mass, translation, rotation

```python
>>> w = Molecule('water')
>>> w.atomList()
[Atom water.H1, Atom water.H2, Atom water.O]
>>> w.O  #is an Atom Object
Atom water.O
```
Non-bonded Forcefields
  - Lennard Jones
  - Electrostatic
  - Macromolecule Elastic Network Models

(Chemical) Bonded Forcefields for Molecular mechanics (AMBER)

Multiple forcefields (denoted by simple ‘addition’)

```python
compoundFF = LennardJonesForceField() +
            ElectrostaticForceField(name='es', cutoff=10.)
```

Electrostatics with Cutoff, Ewald Sum, or Fast Multipole
Universe Object

- The complete simulation box
  - Atoms, Interaction Force fields, External fields
  - Thermo- and baro-stats
  - Periodicity
- Particle properties (fetch and set)
- List of atoms `universe.atomList()`
- Current configuration of universe
  `universe.configuration()`
**Trajectory (and TrajectoryVariable) Object**

- Performs basic file I/O for trajectory data
- Mainly used to store configuration \( x_\mu(t) \) data,
  \[ \mu = \{0, N - 1\}, t = \{0, N_{\text{steps}} - 1\} \]
- Also force \( F_\mu(t) \), velocity \( v_\mu(t) \), and total energy \( E(t) \)

```python
>>> traj = Trajectory(universe,'file.nc','r')  # Read config
>>> len(traj)  # Number of sampled points
10
>>> t = 0
>>> init_config_array = traj.configuration[t].array
>>> # N x 3 array of initial position
>>> array(traj.configuration).shape
(10, 5, 3)
>>> # N_{\text{steps}} = 10, N = 5, d = 3
```

- **file.nc is a netCDF file (Common Data Form)**
Integrator Object

Molecular dynamics integration

- Can handle NVE, NVT, NPT ensembles
- All possible forces on each atom calculated and summed

\[ \mathbf{F}_\mu = \sum_{f}^{N_{ff}} \sum_{\nu}^{N} \mathbf{F}_{\mu f}(\mathbf{x}_\nu) \]

- Time integration Velocity Verlet algorithm, given \( \Delta t \) and \( N_{\text{steps}} \)
- Periodic writing of output: Trajectory variables, Log, Snapshots
Aim

Simulate equilibrium free draining dynamics of a collection of an assorted polymer chains with Hookean springs. Estimate $\langle Q \rangle^2$ of the springs.

- Import Modules
- Assign Initial Configuration
  - Simulation Box = universe, and Atoms
- Equilibration, Production: MD time step integration
- Analysis of trajectory results
  - Slow method
  - Fast method

Skip to Next Section
from MMTK import *
from MMTK.Trajectory import Trajectory, TrajectoryOutput, StandardLogOutput
from PolymerModel import *
from dumpObjects import *

- Polymer Models and BD integrator is defined in PolymerModel.py
- dumpObjects is a utility (free download) to inspect objects’ data and methods. (more than the default help)
Universe: Dilute System

```python
universe = InfiniteUniverse()
for i in range(3):
    Nbeads = 2*i + 2  # Chains of several sizes
    lc = LinearChain(Nbeads, name='lc' + str(Nbeads),
                     sptype='FENE', radius=0.50, b_parameter=50.)
    universe.addObject(lc)

universe.setForceField(EntropicSpringForceField())
```

- **InfiniteUniverse** is an unbounded simulation box
- **LinearChain** is a new class of ChemicalObject
- **name** can be anything, **sptype** is one of
  - Hookean, FENE, ILC, WLC
- **radius**: dimensionless radius of the bead
- **b_parameter**, **b**
- The only force acting on the beads is the Entropic spring forces of connectors
**Free Draining Integration**

```python
integrator = BrownianIntegrator(universe)

# Equilibration
integrator(steps=1000, delta_t=0.05)
trajectory = Trajectory(universe,
     "brownian-mchain.nc", "w", "Brownian_CHAIN_{Test}")

# Production
integrator(steps=1e5, delta_t=0.05,
actions = [ TrajectoryOutput(trajectory,
     ("time", "energy", "configuration"),
     0, None, 20),
     StandardLogOutput(1000)])
trajectory.close()
```

- **BrownianIntegrator** is the new Integrator class
- Trajectory saves time, configuration, energy in file
- Periodic logs to the screen
Trajectory Analysis: Slow

tsav = Trajectory(universe, "brownian-mchain.nc", "r")
universe.configuration()  # sets indices
i = []; j = []
for bu in universe.bondedUnits():
    if hasattr(bu, 'bonds'):
        for b in bu.bonds:
            i.append(b.a1.index);
            j.append(b.a2.index)
i = array(i); j = array(j)
X = array(tsav.configuration)
Ns = X.shape[0]; N = X.shape[1]
Qvec = X[:,j] - X[:,i]
Q2 = (Qvec * Qvec).sum(2)
q2avg = mean(Q2);
q2err = std(Q2)/sqrt(N+Ns)
print '<Q2>=',q2avg,'+/-', q2err

- Analysis part can be separate file (but universe has to be defined)
- i, j denote a set of atom indices for bonded atom
- Qvec(t) can be directly computed from trajectory configuration.
- Qvec.shape = \( N_{\text{samples}} \times N_c \times 3 \)
- sum(2) is along the third dimension of the array.
from Scientific.IO.NetCDF import *
from numpy import *
dfile = NetCDFFile('brownian-mchain.nc', 'r')
t = dfile.variables['time'][:]
X = dfile.variables['configuration'][:]
Ns = X.shape[0]; N = X.shape[1]
Qvec = X[:,j] - X[:,i]
Q2 = (Qvec * Qvec).sum(2)
q2avg = mean(Q2)
q2err = std(Q2) / sqrt(N+Ns)
print 'N=', N, '; Ns=', Ns
print '<Q2>=', q2avg, '+/-', q2err

- Direct reading of netCDF file
- NetCDFFile object has variables dict object
- The names used to save the trajectory can be used to retrieve data
- The Trajectory object processing of configuration is slow
- Direct is more than 30 times faster
Discussion Time
Outline of this Module

3 MMTK Internals: MD Example
   - Forcefields and Energy Terms
   - Integrator and Trajectories
Evaluator Object
Energy Terms
Passing data to C-functions
Two eval_funcs
Sequence of Calls for Force evaluation
Input Data to Integrator
Choice of Trajectory Variables
How Trajectory Output works?
Outline of this Module

4 MMTK Extension: BD Example

- New Assemblies
- New Forcefields
- New Dynamics
Spring connector
Linear Chains
Entropic Spring ForceField
Bonded Atom Indices
Evaluator Term (C-function)