Using the ProtoMol/MDL 3.0 TestSuite

Trevor Cickovski, Joseph Coffland, Chris Sweet and Jesús A. Izaguirre
Laboratory for Computational Life Sciences, University of Notre Dame

March 8, 2010

1 Obtaining the TestSuite

The testsuite for ProtoMol/MDL 3.0 is obtainable by checking out the most recent version of the software from the `sourceforge.net` server, using Subversion. You may checkout the software anonymously using the following command:

```
svn co https://protomol.svn.sourceforge.net/svnroot/protomol protomol
```

Upon executing this command, a `protomol/` subdirectory will be produced in your working space. Change to this directory and you will find a subdirectory `protomol-test` which contains all regression tests for both PROTOMOL and MDL , along with Python scripts to execute the tests. Thus before using the testsuite, you must minimally have version 2.5 of Python installed (MDL requires this version anyway).

2 Running Tests

The testsuite expects two things to be contained in your executable `PATH`: the Python interpreter and the ProtoMol executable. Thus you must compile ProtoMol before running the testsuite, and must provide its containing directory in the `PATH` environment variable. Then to run the entire testsuite, from the `protomol-test` directory you can just run:

```
python testHarness
```

which will produce output similar to the following:

```
********** 2008-25-07 17:46:14 UTC-4 ************
Test Name: [file...]                        Result
************** Test Suite: Config ***************
LeapfrogTest:                               [PASS]
NMLTest:                                    [PASS]
BornForceTest:                              [PASS]
LeapfrogTruncatedShadowTest:                [PASS]
CoulombForceCutoffTest:                     [PASS]
VelacTest:                                  [PASS]
....
```

```
*************** Test Suite: MDL *****************
LeapfrogTest:                                [PASS]
NMLTest:                                     [PASS]
BornForceTest:                               [PASS]
....
```

The testsuite contains tests for both PROTOMOL (the `Config` testsuite) and MDL (the `MDL` testsuite). Each of their listings contains a set of tests which are each designed individually to test one feature of the software. For example, `LeapfrogTest` tests the Leapfrog propagator, `NML` tests Normal Mode Langevin dynamics, `BornForceTest` tests computation of the Born radii for implicit solvent simulations. Assuming you checked out a fresh version of the software and did not make any source code modifications, all PROTOMOL and MDL tests should initially pass.

2.1 Options

Running the `testHarness` script with the `-h` option will provide help:

```
> python testHarness -h
Usage: testHarness [options] [command]

Command:
  enable <test>      enable a previously disabled a test
  disable <test>     disable a test
  init <test>        initialize a test
  reset <test>       reset a test by removing its .expect files
  run <test>         run a specific test
  try <test>         run a test and view its results (implies -I)
  view <test>        view a test's files

  If no command is given all tests will be run.

Options:
  -h, --help           show this help message and exit
  -I, --interactive    run tests in interactive mode
  -V, --valgrind       run tests under valgrind
  -v                   increase verbosity
  -C, --disable-color  disable color output
```

Thus the `testHarness` script accepts both flags or special actions to take for certain tests. As the help output states, if no command is given then all tests will be run, which we have already seen. We can enable or disable certain tests, i.e.:

```
> python testHarness disable ConfigTests/LeapfrogTest
```

It is important to note that disabling or enabling a test has effect until rerunning `enable` or `disable`, even after the test harness stops executing. Initializing a test must be performed when adding a new test to the testsuite. Alternatively, assuming correct file placement the new test will run automatically but an

UNINITIALIZED message will be displayed when the test harness is executed, prompting for initialization. We outline this in Chapter 2. Resetting a test is appropriate if at some point the results produced by a test in the suite are discovered to be incorrect, this will remove all *benchmark* output which is being compared, and require the test to be reinitialized. By using the run command, we can just run one test, i.e.:

```
> python testHarness run ConfigTests/NMLTest
```

which is useful if one test is being debugged, to avoid rerunning all tests upon every modification. The try command is similar, but provides interactivity if the test fails, which we outline below. Benchmark files for a specific test can be viewed using the view command.

The flags are fairly self explanatory; we have already seen -h for the help screen, -v provides more helpful output upon a test failure, -C removes color and -V will also profile execution using ValGrind (valgrind.org) assuming it is installed. A particularly helpful flag is the -I flag, which runs tests interactively. Individual tests can be run interactively with the try command as outlined above. Interactivity provides the ability for a user to choose the action to take upon a test failure. When a test fails, output similar to the following is produced:

```
***************** 2008-31-07 11:42:39 UTC-4 ************************
Test Name: [file...]                                          Result
***************** Test Suite: Config ******************************
LeapfrogTest: std                                             [FAIL]
(a)bort (c)ontinue (D)isable (d)iff (i)nitialize (l)og (r)eset
(t)est (q)uit (v)iew?
```

There are several actions that can subsequently be taken. The <a> key will abort the test harness script, and <c> and <q> take identical action by continuing the harness without taking any action. The failed test can be disabled (<D>), reset (<r>) to use the output currently generated as the new benchmark, or initialized (<i>). Benchmark output can be viewed by pressing the <v> key. Pressing the lowercase <d> will run the diff command to generate a description of differences between currently generated and benchmark output. The <l> key displays any logged output from the testsuite, and <t> will rerun the test (useful if the test has been initialized or reset). Note the current test remains active until <a>, <c> or <q> is pressed.

3 Creating New Tests

3.1 Test Architecture

In order to define new PROTOMOL and MDL tests, you must understand the testsuite structure and components. Within protomol-test, you will find two subdirectories ConfigTests and MDLTests. Each of these subdirectories contains a set of small Python scripts, one per test. A requirement enforced by the testsuite is that each of these scripts end in Test. To show an example, let's look at AngleForceTest:

```
#!/usr/bin/python
import os

os.system('python ProtoMolCheck AngleForce xyz')
```

Note that the last line of the script is a system call to the Python interpreter, passing as arguments a script `ProtoMolCheck`, the current test name (without the `Test` suffix), and a set of one or more file suffixes. In general, the tests for PROTOMOL will run the `ProtoMolCheck` Python script and the MDL tests will run the `MDLCheck` script. These are also contained within their respective testsuite subdirectories.

`ProtoMolCheck` takes as arguments the test name and file suffixes. All data files are contained within the `data/` subdirectory of `ConfigTests`. `ProtoMolCheck` will execute PROTOMOL on a configuration file with the same name as the test name, concatenated with a `.conf` suffix. By default, `ProtoMolCheck` checks three types of output: standard output, standard error, and the return code (which should always be zero upon a successful execution). These outputs are compared to files in the `data` directory with the `.expect` suffix – for example, `AngleForceTest.return.expect`, `AngleForceTest.stdout.ex` etc. If file suffixes are provided as arguments to `ProtoMolCheck`, the `DataComparator` script is run to compare numerical values between a `.orig` file and an output file generated at runtime consisting of the test name concatenated with the appropriate suffix (for example, the above case would compare `AngleForceTest.xyz` and `AngleForceTest.xyz.orig`). Thus it is mandatory that in each test configuration file output file names which should be compared are named properly. Once compared, the generated output files are removed from the `data/` directory.

The `DataComparator` source code is contained within a subdirectory of the `protomol-test` root, called `src/`. When the `protomol-test` scons script is run, the `DataComparator` executable is produced in the `protomol-test` root directory. For any test which check file output as well as defaults, the `DataComparator` is run. The `DataComparator` accepts a command line argument for tolerance when comparing values, which in the `ProtoMolCheck` script is set to 0.0000001. When a PROTOMOL-generated output file is compared to a `.orig`, each numerical value is compared to ensure the absolute value of the difference falls within this tolerance. If it does not, a `Files do not match` message is sent to standard output and the test fails. The `DataComparator` can recognize XYZ trajectory and snapshot files, as well as PDB files and DCD trajectories, and energy files.

MDL tests work very similar to those for PROTOMOL, with the differences being (1) that they are contained within a subdirectory `MDLTests` instead of `ConfigTests`, (2) that scripts with the test name concatenated with the `.py` extension are executed as opposed to running the PROTOMOL executable on a `.conf` file, and (3) the driver script is named `MDLCheck` instead of `ProtoMolCheck`. Everything else works the same.

3.2 Creating a PROTOMOL Test

Now suppose we wanted to create our own test for PROTOMOL, for a new integrator `FooBar`. For the sake of simplicity, we'll assume that the integrator is single timestepping (STS) and its only necessary parameter is a timestep. Suppose we wanted to test the integrator on alanine dipeptide in vacuum. An appropriate name for our new test would be `FooBarTest`, so our PROTOMOL configuration file must be named `FooBarTest.conf` and might look something like this:

```
firststep 0
numsteps 1

# Constraints
angularMomentum 0
comMotion 0
exclude scaled1-4
```

```
posfile alanine.pdb # from solvated simulation with same name
psffile alanine.psf
parfile alanine.par
temperature 310

boundaryConditions vacuum

cellManager Cubic

Integrator {
 level 0 FooBar {
        timestep 0.5
    force Bond
    force Angle
    force Dihedral
    force Improper
    force LennardJones Coulomb
       -algorithm NonbondedSimpleFull
  }
}

finpdbposfile FooBarTest.pdb
allenergiesfile FooBarTest.energies
```

Note that we generated two types of output: positions in a PDB snapshot (most appropriate to test when dealing with integrators) and an energies file, and that we named them using the test name concatenated with the appropriate suffix. We place this file in the directory `ConfigTests/data`. Now, we must create a `FooBarTest` script within the `ConfigTests/` directory one level up in the hierarchy. We must call `ProtoMolCheck` through a system call to the Python interpreter, passing the test name and appropriate suffixes which are in this case `pdb` and `energies`:

```
import os

os.system('python ProtoMolCheck FooBar pdb energies')
```

The test has now been created, because the `testHarness` automatically executes all Python test scripts within the `ConfigTests` and `MDLTests` subdirectories of the root. When the `testHarness` script is executed, you'll see the following output:

```
FooBarTest:                                          [UNINITIALZED]
```

If a test is `UNINITIALIZED`, that means that the `.expect` and `.orig` files have not been created yet. To initialize and create these files, you'll want to run the `testHarness` with the `-I` option, which pauses on any test that does not pass and provides you with choices:

```
python testHarness -I
```

which will stop after detecting an uninitialized test:

```
FooBarTest:                                        [UNINITIALZED]
(a)bort (c)ontinue (D)isable (d)iff (i)nitialize (l)og (r)eset
(t)est (q)uit (v)iew?
```

Now you will want to press the `<I>` key to initialize this test. Once initialized, the `testHarness` script will continue. If you run the script a second time, the test should pass:

```
FooBarTest:                                                [PASS]
```

3.3 Creating an MDL Test

Once the new `FooBar` integrator has been wrapped for MDL , we can also develop an analogous MDL test for the integrator. The Python script `FooBarTest.py` for running the same simulation protocol as the configuration file for the PROTOMOL test, would look like this:

```
from MDL import *

phys = Physical()
io = IO()
io.readPDBPos(phys, "data/alanine.pdb")
io.readPSF(phys, "data/alanine.psf")
io.readPAR(phys, "data/alanine.par")
phys.bc = "Vacuum"
phys.temperature = 310

forces = Forces()
ff = forces.makeForceField(phys, "charmm")

io.files = {'energies':('data/FooBarTest.energies', 1)}

prop = Propagator(phys, forces, io)
prop.propagate("FooBar", steps=1, dt=0.5, forcefield=ff)
io.writePDBPos(phys, 'data/FooBarTest.pdb')
```

This file is placed in the `MDLTests/data` directory, and we next must create a `FooBarTest` script one level higher in the directory hierarchy, which runs `MDLCheck`:

```
import os
os.system('python MDLCheck FooBar pdb energies')
```

That's it! Rules with respect to initialization also apply to MDL tests.

4 Regression Testing Rules

As a rule of thumb, the regression testsuite should be executed upon any change of PROTOMOL or MDL source code viewed as a permanent change. This of course has different meanings for developers and users; for developers the testsuite should be executed before committing updated source code to the centralized repository, for users the testsuite should be executed to ensure the software still runs properly after any local source code modifications.

Note that changing PROTOMOL source code can affect the precompiled binaries of MDL , and so both sets of regression tests should be run upon changes to PROTOMOL . Changes to pure Python libraries of MDL only require the MDL tests to be run to ensure validity. This is an implication of the multi-tiered design of the problem solving environment: higher level layers (i.e. a domain-specific language like MDL) only depend on levels lower in the hierarchy (i.e. the PROTOMOL computational back end).

5 Contact Info

If you have questions or comments, please direct them to:

Laboratory For Computational Life Sciences
University of Notre Dame
325 Cushing Hall
Notre Dame, IN 46556

Or e-mail:

1. LCLS: *lcls@nd.edu*
2. Trevor Cickovski: *cickovtm@eckerd.edu*
3. Jesús A. Izaguirre: *izaguirr@nd.edu*