runtest Documentation

Release 1.0.0

Radovan Bast

Jun 11, 2017

Contents

Core library:

CHAPTER 1

About the runtest library

Scope

When testing numerical codes against functionality regression, you typically cannot use a plain diff against the reference outputs due to numerical noise in the digits and because there may be many numbers that change all the time and that you do not want to test (e.g. date and time of execution).

The aim of this library is to make the testing and maintenance of tests easy. The library allows to extract portions of the program output(s) which are automatically compared to reference outputs with a relative or absolute numerical tolerance to compensate for numerical noise due to machine precision.

Design decisions

The library is designed to play well with CTest, to be convenient when used interactively, and to work without trouble on Linux, Mac, and Windows. It offers test scripts a basic argument parsing. All host program independent code has been collected into the core library. The aim is to keep host program specifics minimal and to specify and keep them in adapter scripts outside the core library.

CHAPTER 2

Filter options

Relative tolerance

There is no default. You have to select either relative or absolute tolerance for each test when testing floats. You cannot select both.

In this example we set the relative tolerance to 1.0e-10:

```
f = Filter()
f.add(from_string   = 'Electronic energy',
      num_lines     = 8,
      rel_tolerance = 1.0e-10)
```

Absolute tolerance

There is no default. You have to select either relative or absolute tolerance for each test when testing floats. You cannot select both.

In this example we set the absolute tolerance to 1.0e-10:

```
f = Filter()
f.add(from_string   = 'Electronic energy',
      num_lines     = 8,
      abs_tolerance = 1.0e-10)
```

How to check entire file

By default all lines are tested so if you omit any string anchors and number of lines we will compare numbers from the entire file.

Example:

```
f = Filter()
f.add() # filter all lines
```

Filtering between two anchor strings

Example:

```
f = Filter()
f.add(from_string = '@   Elements of the electric dipole',
      to_string   = '@   anisotropy')
```

This will extract all floats between these strings including the lines of the strings.

The start/end strings can be regular expressions, for this use from_re or to_re. Any combination containing from_string/from_re and to_string/to_re is possible.

Filtering a number of lines starting with string/regex

Example:

```
f = Filter()
f.add(from_string  = 'Electronic energy',
      num_lines = 8,                          # here we compare 8 lines
      rel_tolerance = 1.0e-10)
```

The start string can be a string (from_string) or a regular expression (from_re). In the above example we extract and compare all lines that start with 'Electronic energy' including the following 7 lines.

Extracting single lines

This example will compare all lines which contain 'Electronic energy':

```
f = Filter()
f.add(string = 'Electronic energy',
      rel_tolerance = 1.0e-10)
```

Instead of single string we can give a single regular expression (re).

How to ignore sign

Sometimes the sign does is not predictable. For this set ignore_sign to True:

```
f = Filter()
f.add(string      = 'Something',
      ignore_sign = True)
```

How to ignore very small or very large numbers

You can ignore very small numbers with ignore_below. Default is 1.0e-40. Ignore all floats that are smaller than this number (this option ignores the sign).

As an example consider the following result tensor:

```
3716173.43448289          0.00000264         -0.00000346
     -0.00008183      75047.79698485          0.00000328
      0.00003493         -0.00000668      75047.79698251

      0.00023164    -153158.24017016         -0.00000493
  90142.70952070         -0.00000602          0.00000574
      0.00001946         -0.00000028          0.00000052

      0.00005844         -0.00000113    -153158.24017263
     -0.00005667          0.00000015         -0.00000022
  90142.70952022          0.00000056          0.00000696
```

The small numbers are clearly numerical noise and we do not want to test them at all. In this case it is useful to set ignore_below to 1.0e-4.

Alternatively one could use absolute tolerance to avoid checking the noisy zeros.

You can ignore very large numbers with ignore_above (also this option ignores the sign).

How to ignore certain numbers

The keyword mask is useful if you extract lines which contain both interesting and uninteresting numbers (like timings which change from run to run).

Example:

```
f = Filter()
f.add(from_string = 'no.    eigenvalue (eV)   mean-res.',
      num_lines = 4,
      rel_tolerance = 1.0e-4,
      mask = [1, 2, 3])
```

Here we use only the first 3 floats in each line.

CHAPTER 3

Notes for contributors

Please communicate bugfixes and new features.

Classes, methods, and variables that start with underscore are private.

Please keep the output as silent as possible.

Please no writes to stderr or sys.exit inside the core library. Rather raise the error and let the caller deal with it.

Running a test script:

CHAPTER 4

Command-line arguments

-h, –help

Show help message and exit.

-b BINARY_DIR, –binary-dir=BINARY_DIR

Directory containing the binary/launcher. By default it is the directory of the test script which is executed.

-w WORK_DIR, –work-dir=WORK_DIR

Working directory where all generated files will be written to. By default it is the directory of the test script which is executed.

-v, –verbose

Give more verbose output upon test failure (by default False).

-s, –skip-run

Skip actual calculation(s), only compare numbers. This is useful to adjust the test script for long calculations.

CHAPTER 5

Generated files

The test script generates three files per run with the suffixes ".diff", ".filtered", and ".reference".

The ".filtered" file contains the extracted numbers from the present run.

The ".reference" file contains the extracted numbers from the reference file.

If the test passes, the ".diff" file is an empty file. If the test fails, it contains information about the difference between the present run and the reference file.

Documentation for people who add new tests:

CHAPTER 6

General tips

How to add a new test

Test scripts are python scripts which return zero (success) or non-zero (failure). You define what success or failure means. The runtest library helps you with basic tasks but you are free to go beyond and define own tests with arbitrary complexity.

Strive for portability

Avoid shell programming or symlinks in test scripts otherwise the tests are not portable to Windows. Therefore do not use os.system() or os.symlink(). Do not use explicit forward slashes for paths, instead use os.path.join().

Always test that the test really works

It is easy to make a mistake and create a test which is always "successful". Test that your test catches mistakes. Verify whether it extracts the right numbers.

Never commit functionality to the main development line without tests

If you commit functionality to the main development line without tests then this functionality will break sooner or later and we have no automatic mechanism to detect it. Committing new code without tests is bad karma.

Never add inputs to the test directories which are never run

We want all inputs and outputs to be accessile by the default test suite. Otherwise we have no automatic way to detect that some inputs or outputs have degraded. And degraded inputs and outputs are useless and confusing.

CHAPTER 7

Adding tests in DIRAC

The runtest_dirac.py is an extension of the runtest library. The runtest library is a low-level program-independent library that provides infrastructure for running calculations and extracting and comparing numbers against reference outputs.

Reference outputs

Reference outputs are placed in directory “result/”.

Easy example

Let us look at an easy example.

First we load some standard modules and import functionality from the library (highlighted lines). This part is generic to all DIRAC tests.

```
#!/usr/bin/env python

import os
import sys

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dirac import Filter, TestRun

test = TestRun(__file__, sys.argv)

f = Filter()
f.add(from_string   = '@   Elements of the electric dipole',
      to_string     = '@   anisotropy',
      rel_tolerance = 1.0e-5)
f.add(from_string   = '************ Expectation values',
      to_string     = 's0 = T : Expectation value',
```

```
      rel_tolerance = 1.0e-5)

test.run(['PBE0gracLB94.inp', 'GLLBsaopLBalpha.inp'], ['Ne.mol'], f)

sys.exit(test.return_code)
```

Then we construct the filter object. It consists of two filter tasks. We can construct as many filter objects as we like and each can consist of as many tasks as we like.

```
#!/usr/bin/env python

import os
import sys

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dirac import Filter, TestRun

test = TestRun(__file__, sys.argv)

f = Filter()
f.add(from_string   = '@   Elements of the electric dipole',
      to_string     = '@   anisotropy',
      rel_tolerance = 1.0e-5)
f.add(from_string   = '************ Expectation values',
      to_string     = 's0 = T : Expectation value',
      rel_tolerance = 1.0e-5)

test.run(['PBE0gracLB94.inp', 'GLLBsaopLBalpha.inp'], ['Ne.mol'], f)

sys.exit(test.return_code)
```

With test.run we really run the job. Note how we pass the filter object. If we omit to pass it, then the calculations will be run but not verified. This is useful for multi-step jobs. Also observe how we give a list of input files and molecule files (in this case two input files and one molecule file). The test library will run and test all input/molecule file combinations (in this case two). We could have executed them separately in two lines. This would make no difference, just more typing.

```
#!/usr/bin/env python

import os
import sys

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dirac import Filter, TestRun

test = TestRun(__file__, sys.argv)

f = Filter()
f.add(from_string   = '@   Elements of the electric dipole',
      to_string     = '@   anisotropy',
      rel_tolerance = 1.0e-5)
f.add(from_string   = '************ Expectation values',
      to_string     = 's0 = T : Expectation value',
      rel_tolerance = 1.0e-5)

test.run(['PBE0gracLB94.inp', 'GLLBsaopLBalpha.inp'], ['Ne.mol'], f)
```

```
sys.exit(test.return_code)
```

Finally on the last line we exit with test.return_code. This is important. The integer test.return_code equals the number of failed test runs. It is zero if the test is successful.

Multi-step tests

Here is an example for a multi-step test. Note how only every second run is actually verified by passing the filter object.

```
#!/usr/bin/env python

import os
import sys
import shutil

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dirac import Filter, TestRun

test = TestRun(__file__, sys.argv)

f = Filter()
f.add(from_string = 'Energy at final geometry is',
      num_lines = 3,
      rel_tolerance= 1.0e-4)

test.run(['O.inp'], ['O.mol'], args='--get=DFCOEF')
shutil.copy('DFCOEF', 'DFPROJ')
test.run(['H2O.inp'], ['H2O.mol'], f, args='--copy=DFPROJ')

test.run(['O.2c_iotc.inp'], ['O.mol'], args='--get=DFCOEF')
shutil.copy('DFCOEF', 'DFPROJ')
test.run(['H2O.2c_iotc.inp'], ['H2O.mol'], f, args='--copy=DFPROJ')

test.run(['O.2c_iotc_noamfi.inp'], ['O.mol'], args='--get=DFCOEF')
shutil.copy('DFCOEF', 'DFPROJ')
test.run(['H2O.2c_iotc_noamfi.inp'], ['H2O.mol'], f, args='--copy=DFPROJ')

# cleanup
os.unlink('DFCOEF')
os.unlink('DFPROJ')

sys.exit(test.return_code)
```

The other runs only serve to prepare files and are not checked (no filter passed as argument).

```
#!/usr/bin/env python

import os
import sys
import shutil

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dirac import Filter, TestRun

test = TestRun(__file__, sys.argv)
```

```
f = Filter()
f.add(from_string = 'Energy at final geometry is',
      num_lines = 3,
      rel_tolerance = 1.0e-4)

test.run(['O.inp'], ['O.mol'], args='--get=DFCOEF')
shutil.copy('DFCOEF', 'DFPROJ')
test.run(['H2O.inp'], ['H2O.mol'], f, args='--copy=DFPROJ')

test.run(['O.2c_iotc.inp'], ['O.mol'], args='--get=DFCOEF')
shutil.copy('DFCOEF', 'DFPROJ')
test.run(['H2O.2c_iotc.inp'], ['H2O.mol'], f, args='--copy=DFPROJ')

test.run(['O.2c_iotc_noamfi.inp'], ['O.mol'], args='--get=DFCOEF')
shutil.copy('DFCOEF', 'DFPROJ')
test.run(['H2O.2c_iotc_noamfi.inp'], ['H2O.mol'], f, args='--copy=DFPROJ')

# cleanup
os.unlink('DFCOEF')
os.unlink('DFPROJ')

sys.exit(test.return_code)
```

Passing arguments to pam

This can be done with args. Example:

```
test.run(['O.inp'], ['O.mol'], args='--get=DFCOEF')
```

Catching expected errors

We have tests which fail with MPI or integer(4) compilation in a predictable and controlled way. In this case we don't want to see the test failing, but we want it to pass.

Example:

```
#!/usr/bin/env python

import os
import sys

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dirac import Filter, TestRun

test = TestRun(__file__, sys.argv)

f = Filter()
f.add(string = 'Number of determinants/combinations')
f.add(string = ' Final energy')

test.run(['He.inp'],
         ['He.mol'],
```

```
          f,
          accepted_errors=['memory off-set too big for INTEGER*4',
                           'FATAL ERROR for LUCITA runs: memory offset (dynamic␣
↪memory - static memory) is too big for i*4'])

 sys.exit(test.return_code)
```

When we run this test as separate script, we see:

```
$ ./test -b ~/dirac/build/

running test: He He
found error which is expected/accepted: FATAL ERROR for LUCITA runs: memory offset␣
↪(dynamic memory - static memory) is too big for i*4
```

CHAPTER 8

Adding tests in Dalton

Reference outputs

Reference outputs are placed in directory "result/".

Creating test scripts

First follow the documentation for DIRAC, see *Adding tests in DIRAC*.

A difference with respect to DIRAC is that in Dalton we do pass a suffix-filter dictionary in order to be able to test different output files.

Let us look at an example. Observe how we create a filter for "out", and another filter for "stdout", which we pass as a dictionary (curly brackets):

```
#!/usr/bin/env python

import os
import sys

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dalton import Filter, TestRun

test = TestRun(__file__, sys.argv)

f_out = Filter()
f_out.add(from_string = 'Final results from SIRIUS',
          num_lines = 11,
          rel_tolerance = 1.0e-7)

f_stdout = Filter()
f_stdout.add(from_string  = 'beta = -Efff',
             num_lines = 10,
```

```
              rel_tolerance = 1.0e-7,
              ignore_below = 1.0e-7)

test.run(['hf.dal', 'hf_2np1.dal'], ['h2o2.mol'], {'out': f_out, 'stdout': f_stdout})

sys.exit(test.return_code)
```

Updating tests

In most cases it is sufficient to update/replace the reference output.

Documentation for people who implement host-code specific adapters:

CHAPTER 9

Adapter for DIRAC

In runtest_dirac.py we have subclassed TestRun to override the execute method for convenience:

```
import os
import sys
import runtest


class Filter(runtest.Filter):

    def __init__(self):
        runtest.Filter.__init__(self)

    def add(self, *args, **kwargs):
        try:
            runtest.Filter.add(self, *args, **kwargs)
        except runtest.FilterKeywordError, e:
            sys.stderr.write(str(e))
            sys.exit(-1)


class TestRun(runtest.TestRun):

    def __init__(self, _file, argv):
        runtest.TestRun.__init__(self, _file, argv)
        self.return_code = 0

    def run(self, inp_files, mol_files, f=None, args='', accepted_errors=[]):

        launch_script = os.path.normpath(os.path.join(self.binary_dir, 'pam'))
        if self.skip_run:
            sys.stdout.write('\nskipping actual run\n')
        else:
            if not os.path.exists(launch_script):
                sys.stderr.write('ERROR: launch script %s not found\n' % launch_
↪script)
```

```
                sys.stderr.write('       have you set the correct --binary-dir (or -
↪b)?\n')
                sys.stderr.write('       try also --help\n')
                sys.exit(-1)

        if sys.platform == "win32":
            dirac_exe='dirac.x.exe'
        else:
            dirac_exe='dirac.x'

        launcher = 'python "%s" --dirac=%s --noarch --nobackup %s' % (launch_script,␣
↪os.path.join(self.binary_dir, dirac_exe), args)

        for inp in inp_files:
            inp_no_suffix = os.path.splitext(inp)[0]
            for mol in mol_files:
                mol_no_suffix = os.path.splitext(mol)[0]
                out = '%s_%s.out' % (inp_no_suffix, mol_no_suffix)
                sys.stdout.write('\nrunning test: %s %s\n' % (inp_no_suffix, mol_no_
↪suffix))

                command = launcher + ' --inp=%s --mol=%s' % (inp, mol)
                try:
                    runtest.TestRun.execute(self,
                                            command=command,
                                            accepted_errors=accepted_errors)
                    if f is None:
                        sys.stdout.write('finished (no reference)\n')
                    else:
                        try:
                            f.check(self.work_dir, '%s' % out, 'result/%s' % out,␣
↪self.verbose)
                            sys.stdout.write('passed\n')
                        except IOError, e:
                            sys.stderr.write('ERROR: could not open file %s\n' % e.
↪filename)
                            sys.exit(-1)
                        except runtest.TestFailedError, e:
                            sys.stderr.write(str(e))
                            self.return_code += 1
                        except runtest.BadFilterError, e:
                            sys.stderr.write(str(e))
                            sys.exit(-1)
                        except runtest.FilterKeywordError, e:
                            sys.stderr.write(str(e))
                            sys.exit(-1)
                except runtest.AcceptedError, e:
                    sys.stdout.write(str(e))
                except runtest.SubprocessError, e:
                    sys.stderr.write(str(e))
                    sys.exit(-1)
```

This has the advantage that we can run and check a series of input file tuples with the same filter set as demonstrated here:

```
#!/usr/bin/env python

import os
```

```
import sys

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from runtest_dirac import Filter, TestRun

test = TestRun(__file__, sys.argv)

f = Filter()
f.add(from_string   = '@   Elements of the electric dipole',
      to_string     = '@   anisotropy',
      rel_tolerance = 1.0e-5)
f.add(from_string   = '************ Expectation values',
      to_string     = 's0 = T : Expectation value',
      rel_tolerance = 1.0e-5)

test.run(['PBE0gracLB94.inp', 'GLLBsaopLBalpha.inp'], ['Ne.mol'], f)

sys.exit(test.return_code)
```

Note that the filter argument is optional which allows us to run calculations which are not tested (we need this in multi-step jobs where not all the steps are tested).

CHAPTER 10

Adapter for Dalton

```
import os
import sys
import runtest


class Filter(runtest.Filter):

    def __init__(self):
        runtest.Filter.__init__(self)

    def add(self, *args, **kwargs):
        try:
            runtest.Filter.add(self, *args, **kwargs)
        except runtest.FilterKeywordError, e:
            sys.stderr.write(str(e))
            sys.exit(-1)


class TestRun(runtest.TestRun):

    def __init__(self, _file, argv):
        runtest.TestRun.__init__(self, _file, argv)
        self.return_code = 0

    def run(self, inp_files, mol_files, f=None, args='', accepted_errors=[]):

        launch_script = os.path.normpath(os.path.join(self.binary_dir, 'dalton'))
        if self.skip_run:
            sys.stdout.write('\nskipping actual run\n')
        else:
            if not os.path.exists(launch_script):
                sys.stderr.write('ERROR: launch script %s not found\n' % launch_
↪script)
                sys.stderr.write('       have you set the correct --binary-dir (or -
↪b)?\n')
```

```
                sys.stderr.write('       try also --help\n')
                sys.exit(-1)

        launcher = '%s -noarch -nobackup %s' % (launch_script, args)

        for inp in inp_files:
            inp_no_suffix = os.path.splitext(inp)[0]
            for mol in mol_files:
                mol_no_suffix = os.path.splitext(mol)[0]
                output_no_suffix = '%s_%s' % (inp_no_suffix, mol_no_suffix)
                sys.stdout.write('\nrunning test: %s %s\n' % (inp_no_suffix, mol_no_
↪suffix))

                command = launcher + ' %s %s' % (inp_no_suffix, mol_no_suffix)
                try:
                    runtest.TestRun.execute(self,
                                            command=command,
                                            stdout_file_name = '%s.stdout' % output_
↪no_suffix,
                                            accepted_errors=accepted_errors)
                    if f is None:
                        sys.stdout.write('finished (no reference)\n')
                    else:
                        try:
                            # f is a suffix-filter dictionary
                            for suffix in f:
                                out = '%s.%s' % (output_no_suffix, suffix)
                                f[suffix].check(self.work_dir, '%s' % out, 'result/%s
↪' % out, self.verbose)
                            sys.stdout.write('passed\n')
                        except IOError, e:
                            sys.stderr.write('ERROR: could not open file %s\n' % e.
↪filename)
                            sys.exit(-1)
                        except runtest.TestFailedError, e:
                            sys.stderr.write(str(e))
                            self.return_code += 1
                        except runtest.BadFilterError, e:
                            sys.stderr.write(str(e))
                            sys.exit(-1)
                        except runtest.FilterKeywordError, e:
                            sys.stderr.write(str(e))
                            sys.exit(-1)
                except runtest.AcceptedError, e:
                    sys.stdout.write(str(e))
                except runtest.SubprocessError, e:
                    sys.stderr.write(str(e))
                    sys.exit(-1)
```

CHAPTER 11

Adapter for GPUnCH

```
import os
import sys
import runtest


class Filter(runtest.Filter):

    def __init__(self):
        runtest.Filter.__init__(self)

    def add(self, *args, **kwargs):
        try:
            runtest.Filter.add(self, *args, **kwargs)
        except runtest.FilterKeywordError, e:
            sys.stderr.write(str(e))
            sys.exit(-1)


class TestRun(runtest.TestRun):

    def __init__(self, _file, argv):
        runtest.TestRun.__init__(self, _file, argv)
        self.return_code = 0

    def run(self, inp_files, f=None, accepted_errors=[]):

        if sys.platform == "win32":
            executable = 'gpunch.exe'
        else:
            executable = 'gpunch'

        executable = os.path.normpath(os.path.join(self.binary_dir, executable))
        if self.skip_run:
            sys.stdout.write('\nskipping actual run\n')
```

```
        else:
            if not os.path.exists(executable):
                sys.stderr.write('ERROR: executable not found in %s\n' % executable)
                sys.stderr.write('       have you set the correct --binary-dir (or -
↪b)?\n')
                sys.stderr.write('       try also --help\n')
                sys.exit(-1)

        for inp in inp_files:
            out = '%s.out' % os.path.splitext(inp)[0]
            sys.stdout.write('\nrunning test: %s\n' % inp)

            command = executable + ' %s %s' % (inp, out)
            try:
                runtest.TestRun.execute(self,
                                        command=command,
                                        accepted_errors=accepted_errors)
                if f is None:
                    sys.stdout.write('finished (no reference)\n')
                else:
                    try:
                        f.check(self.work_dir, '%s' % out, 'reference/%s' % out, self.
↪verbose)
                        sys.stdout.write('passed\n')
                    except IOError, e:
                        sys.stderr.write('ERROR: could not open file %s\n' % e.
↪filename)
                        sys.exit(-1)
                    except runtest.TestFailedError, e:
                        sys.stderr.write(str(e))
                        self.return_code += 1
                    except runtest.BadFilterError, e:
                        sys.stderr.write(str(e))
                        sys.exit(-1)
                    except runtest.FilterKeywordError, e:
                        sys.stderr.write(str(e))
                        sys.exit(-1)
            except runtest.AcceptedError, e:
                sys.stdout.write(str(e))
            except runtest.SubprocessError, e:
                sys.stderr.write(str(e))
                sys.exit(-1)
```