runtest Documentation

Release 2.2.0

Radovan Bast

Sep 25, 2018

About

Numerically tolerant end-to-end test library for scientific codes.

This documents the latest code on the `master` branch. The `release-1.3.z` code is documented here: http://runtest.readthedocs.io/en/release-1.3.z/.

CHAPTER 1

Motivation

1.1 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.

1.2 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 a basic argument parsing for test scripts.

CHAPTER 2

Audience

2.1 Explain runtest in one sentence

Runtest will assist you in running an entire calculation/simulation, extracting portions for the simulation outputs, and comparing these portions with reference outputs and scream if the results have changed above a predefined numerical tolerance.

2.2 When should one use runtest?

- You compute numerical results.
- You want a library that understands that floating point precision is limited.
- You want to be able to update tests by updating reference outputs.
- You look for an end-to-end testing support.

2.3 When should one not use runtest?

- You look for a unit test library which tests single functions. Much better alternatives exist for this.

CHAPTER 3

Similar projects

- http://testcode.readthedocs.io: testcode is a python module for testing for regression errors in numerical (principally scientific) software.

CHAPTER 4

General tips

4.1 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.

4.2 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()`.

4.3 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.

4.4 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.

4.5 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. Degraded inputs and outputs are useless and confusing.

CHAPTER 5

How to hook up runtest with your code

The runtest library is a low-level program-independent library that provides infrastructure for running calculations and extracting and comparing numbers against reference outputs. The library does not know anything about your code.

In order to tell the library how to run your code, the library requires that you define a configure function which defines how to handle a list of input files and extra arguments. This configure function also defines the launcher script or binary for your code, the full launch command, the output prefix, and relative reference path where reference outputs are stored. The output prefix can also be `None`.

Here is an example module `runtest_config.py` which defines such a function:

```
def configure(options, input_files, extra_args):
    """
    This function is used by runtest to configure runtest
    at runtime for code specific launch command and file naming.
    """

    from os import path
    from sys import platform

    launcher = 'pam'
    launcher_full_path = path.normpath(path.join(options.binary_dir, launcher))

    (inp, mol) = input_files

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

    command = []
    command.append('python {0}'.format(launcher_full_path))
    command.append('--dirac={0}'.format(path.join(options.binary_dir, exe)))
    command.append('--noarch --nobackup')
    command.append('--inp={0} --mol={1}'.format(inp, mol))
    if extra_args is not None:
```

(continues on next page)

(continued from previous page)

```
        command.append(extra_args)

    full_command = ' '.join(command)

    inp_no_suffix = path.splitext(inp)[0]
    mol_no_suffix = path.splitext(mol)[0]

    output_prefix = '{0}_{1}'.format(inp_no_suffix, mol_no_suffix)

    relative_reference_path = 'result'

    return launcher, full_command, output_prefix, relative_reference_path
```

The function is expected to return `launcher`, `full_command`, `output_prefix`, and `relative_reference_path`.

CHAPTER 6

Example test script

Let us consider a relatively simple annotated example.

First we import modules that we need (highlighted lines):

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

# provides os.path.join
import os

# provides exit
import sys

# we make sure we can import runtest and runtest_config
sys.path.append(os.path.join(os.path.dirname(__file__), '..'))

# we import essential functions from the runtest library
from runtest import version_info, get_filter, cli, run

# this tells runtest how to run your code
from runtest_config import configure

# we stop the script if the major version is not compatible
assert version_info.major == 2

# construct a filter list which contains two filters
f = [
    get_filter(from_string='@   Elements of the electric dipole',
               to_string='@   anisotropy',
               rel_tolerance=1.0e-5),
    get_filter(from_string='************ Expectation values',
               to_string='s0 = T : Expectation value',
               rel_tolerance=1.0e-5),
]

# invoke the command line interface parser which returns options
```

(continues on next page)

(continued from previous page)

```
options = cli()

ierr = 0
for inp in ['PBE0gracLB94.inp', 'GLLBsaopLBalpha.inp']:
    for mol in ['Ne.mol']:
        # the run function runs the code and filters the outputs
        ierr += run(options,
                    configure,
                    input_files=[inp, mol],
                    filters={'out': f})

sys.exit(ierr)
```

Then we construct a list of filters. We can construct as many lists as we like and they can contain as many filters as we like. The list does not have to be called "f". Give it a name that is meaningful to you.

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

# provides os.path.join
import os

# provides exit
import sys

# we make sure we can import runtest and runtest_config
sys.path.append(os.path.join(os.path.dirname(__file__), '..'))

# we import essential functions from the runtest library
from runtest import version_info, get_filter, cli, run

# this tells runtest how to run your code
from runtest_config import configure

# we stop the script if the major version is not compatible
assert version_info.major == 2

# construct a filter list which contains two filters
f = [
    get_filter(from_string='@   Elements of the electric dipole',
               to_string='@   anisotropy',
               rel_tolerance=1.0e-5),
    get_filter(from_string='************ Expectation values',
               to_string='s0 = T : Expectation value',
               rel_tolerance=1.0e-5),
]

# invoke the command line interface parser which returns options
options = cli()

ierr = 0
for inp in ['PBE0gracLB94.inp', 'GLLBsaopLBalpha.inp']:
    for mol in ['Ne.mol']:
        # the run function runs the code and filters the outputs
        ierr += run(options,
                    configure,
                    input_files=[inp, mol],
```

(continues on next page)

(continued from previous page)

```
                    filters={'out': f})

sys.exit(ierr)
```

After we use the command line interface to generate options, we really run the test. Note how we pass the configure function to the run function. Also note how we pass the filter list as a dictionary. If we omit to pass it, then the calculations will be run but not verified. This is useful for multi-step jobs. From the dictionary, the library knows that it should execute the filter list "f" on output files with the suffix "out". It is no problem to apply different filters to different output files, for this add entries to the *filters* dictionary.

CHAPTER 7

Run function arguments

The `run` function has the following signature:

```
def run(options,
        configure,
        input_files,
        extra_args=None,
        filters=None,
        accepted_errors=None):
    ...
```

`options` is set by the command line interface (by the user executing runtest).

`configure` is specific to the code at hand (see the *Example test script*).

`input_files` contains the input files passed to the code launcher. The data structure of `input_files` is set by the `configure` function (in other words by the code using runtest).

There are three more optional arguments to the `run` function which by default are set to `None`:

`extra_args` contains extra arguments. Again, its data structure of is set by the `configure` function (in other words by the code using runtest).

`filters` is a dictionary of suffix and filter list pairs and contains filters to apply to the results. If we omit to pass it, then the calculations will be run but not verified. This is useful for multi-step jobs. See also the *Example test script*. If the `output_prefix` in the `configure` function is set to None, then the filters are applied to the file names literally.

CHAPTER 8

Filter options

8.1 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 at the same time.

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

```
get_filter(from_string='Electronic energy',
           num_lines=8,
           rel_tolerance=1.0e-10)
```

8.2 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 at the same time.

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

```
get_filter(from_string='Electronic energy',
           num_lines=8,
           abs_tolerance=1.0e-10)
```

8.3 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:

```
get_filter(rel_tolerance=1.0e-10)
```

8.4 Filtering between two anchor strings

Example:

```
get_filter(from_string='@   Elements of the electric dipole',
           to_string='@   anisotropy',
           rel_tolerance=1.0e-10)
```

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.

8.5 Filtering a number of lines starting with string/regex

Example:

```
get_filter(from_string='Electronic energy',
           num_lines=8,  # here we compare 8 lines
           abs_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.

8.6 Extracting single lines

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

```
get_filter(string='Electronic energy',
           abs_tolerance=1.0e-10)
```

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

8.7 How to ignore sign

Sometimes the sign is not predictable. For this set `ignore_sign=True`.

8.8 How to ignore the order of numbers

Setting `ignore_order=True` will sort the numbers (as they appear consecutively between anchors, one after another) before comparing them. This is useful for tests where some numbers can change place.

8.9 How to ignore very small or very large numbers

You can ignore very small numbers with skip_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 actually numerical noise and we do not want to test them at all. In this case it is useful to set `skip_below=1.0e-4`.

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

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

8.10 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:

```
get_filter(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. Counting starts with 1.

CHAPTER 9

Command-line arguments

9.1 -h, –help

Show help message and exit.

9.2 -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.

9.3 -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.

9.4 -l LAUNCH_AGENT, –launch-agent=LAUNCH_AGENT

Prepend a launch agent command (e.g. “mpirun -np 8” or “valgrind –leak-check=yes”). By default no launch agent is prepended.

9.5 -v, –verbose

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

9.6 -s, –skip-run

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

9.7 -n, –no-verification

Run calculation(s) but do not verify results. This is useful to generate outputs for the first time.

CHAPTER 10

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.

CHAPTER 11

Contributing

Yes please! Please follow this excellent guide: http://www.contribution-guide.org. We do not require any formal copyright assignment or contributor license agreement. Any contributions intentionally sent upstream are presumed to be offered under terms of the Mozilla Public License Version 2.0.

Methods, and variables that start with underscore are private.

Please keep the default output as silent as possible.

11.1 Where to contribute

Here are some ideas:

- Improve documentation
- Fix typos
- Make it possible to install this package using pip
- Make this package distributable via PyPI

CHAPTER 12

Branching model

We follow the semantic branching model: https://dev-cafe.github.io/branching-model/