testcode

Release 2

James Spencer

July 14, 2009

Contents

testcode is a python module for testing for regression errors in numerical (principally scientific) software. Essentially testcode runs a set of calculations, runs a data extraction program on the output and compares the data to that generated by a previous calculation (which is regarded to be "correct").

The idea behind testcode is that attempting to individually test each function in a large program is hard, especially if such a testing system was not implemented from the start of development. Instead, we compare only the final results and assume that the results of intermediate steps remain unchanged if the final results are also unchanged, up to some tolerance for numerical noise.

A script to extract the "important" data from the raw output is a useful tool in using a program. testcode harnesses the existence of such a script (which is straightforward to write if one does not exist) to know what data it should compare. The format of the output is the part which is program-specific and outsourcing this allows testcode to be a generic framework for testing for regression errors in scientic programs. Alternatively testcode contains a simple routine for extracting data to be checked from the output.

The main features of testcode are:

- Simple configuration files. Adding tests and specifying the local environment is very straightforward.
- Flexible job specification: a subset of tests can be selected, e.g. during development work on a specific part of the codebase.
- More than one program can be tested in a single test suite. This is useful for programs linked by, for instance, a shared library and should not be used for disparate codes.
- Comes with a build script for checking out, compiling and testing the code from a version control repository (currently subversion and git are supported).
- Can produce new benchmarks for new tests or after bug fixes.
- Can run programs either in serial or in parallel.
- Can run multiple tests at once, for rapid testing on multi-core platforms.
- Can run tests locally or via submission to a compute cluster using qsub.
- Can compare previously run tests to the benchmarks.

Much of the time testcode can be run from the command line with no options, but has many options (detailed below) that can change its behaviour to fit with the necessary workflow. Furthermore, testcode.py can be used as a module in custom python scripts; the source of testcode.py is structured and internally documented to aid this. checktest.py contains an example of using testcode as a module.

A test suite based on testcode contains the following:

- **testcode.py** the main test script.
- **userconfig** Configuration file containing user settings.
- **jobconfig** Configuration file containing the descriptions of the jobs and groupings of jobs into categories.
- **data extraction script (optional)** The (user supplied) script to extract data from the output of the program being tested. It is the extracted data which is compared to previous results. The data extraction script must output in a specified format (see below).
- **tests** The tests are simply input files. Ideally each test resides in its own directory along with the benchmark output file and any other required data files for running the test. (Multiple tests can, however, reside within one directory.)
- **checktest.py (optional)** The automated test script which can checkout the latest version of the code from a version control repository, compile it and run the test suite for different compilers. It is designed to be run on a regular basis via cron.

The two configuration files must be in the working directory.

1 Licence

testcode is released under the GNU Lesser General Public Licence, full details of which can be found in the files COPYING and COPYING.LESSER in the documentation directory. This provides the freedom to distribute it along side closed-source packages and to improve it. If testcode is distributed with a scientific program, then a citation is appreciated as it helps me justify spending time developing testcode.

2 Configuration

testcode uses two configuration files, both in an ini-style format, to obtain most of the settings.

- userconfig contains various user settings and preferences and so is best to keep a template of it under version control, so each user can set their own preferences easily.
- jobconfig contains the information about the tests and should not change substantially between users and is suitable for keeping under version control.

2.1 Format

The format used in both configuration files is the same as ini files:

```
[section_name]
variable_1 = value_1
variable_2 = value_2
```

This sets the variables *variable_1* and *variable_2* to be *value_1* and *value_2* respectively belonging to the sectiin called *section_name*.

2.2 userconfig

userconfig contains at least a section with the user option and a section providing information about the codebase to be tested.

The user options are in the [user] section. Valid options are:

benchfile [string] The benchfile is generated by testcode and added to userconfig automatically. Care should be taken if changing this by hand. The benchmarks are best generated via the –runbench option. It is best to set everything required bar the benchfile and then allow testcode to produce the benchmarks and add the necessary information to userconfig.

date_fmt [string] The test outputs are, by default, uniquely labelled using the date. The date format is set to be ddmmyy. This can be changed by setting date_fmt to be a valid date format string (see python documentation, http://docs.python.org/library/time.html).

default_job_type [string] Sets all jobs to have be of the default_job_type unless otherwise specified. If there is only one job type, then this option is irrelevant: all jobs are automatically set to belong to that job type.

diff [string] Set the diff program used to compare the test and benchmark output. The default program is diff.

test_files [string] Set the files (using the * wildcard is allowed) which are deleted from each test directory when the –tidy option is used. test_files is set by default to be 'test*out* submit*.sh* testcode_job*'.

testcode_vcs [string] Set the version control system under which the test suite is kept. The default value is svn. git is also supported.

tolerance [float] Set the default numerical tolerance used to determine if the test agrees with the benchmark. The default value is 1.0e-10.

All other sections correspond to a different job type, that is a different codebase. The name of the job type is given by the section name. Valid options are:

compile [string] Command or script which compiles the program. Only used by checktest.py.

data_tag [string] Set the string used to mark a line in the output which contains data to be tested. Used only if a data extraction script/command is not provided. Default is [QA]. See below.

exe [string] Set the executable location. Paths may be absolute or relative to the userconfig file.

extract [string] Set the command/script used to extract data (see below).

inp [string] Set the input file used in running tests. See below.

make_clean [string] Command or script which removes the compiled object files. Only used by checktest.py. Default: make clean.

out [string] Set the output file produced. If not specified, then assume output is sent to STDOUT. See below.

submit_template [string] Sets the template file to be used to submit jobs to a compute cluster. See the –queue option.

timing_output [string] Timing information is of interest during optimisation work, yet it is not meaningful to compare it to a benchmark value: there are too many variables to make this useful. timing_output sets the column names (as a space-separated list) from the data extraction output which contain the timing information and these columns are then not tested for agreement with the benchmark.

tolerance [float] Set non-default tolerances for specific items in the output for all jobs of this type. This allows items to be tested using to a lower or higher precision, as needed. The format is a space-separated list of key:tolerance entries. For instance, the entry Energy:1.e-12 would require all data items in the "Energy" column of the extracted test output to agree with those of the benchmark output to within 1.e-12 in order for the test to pass.

repository_location [string] Location of remote "master" repository. Only used by checktest.py.

vcs [string] Set the version control system under which the codebase is kept. The default value is svn. git is also supported.

Any other variables are assumed to be aliases for alternative executables of the program (e.g. pointing to an unmodified version or a development version).

A sample userconfig file is:

```
[user]
default_job_type = cpmd
benchfile = benchmark%(INPUTTAG)s-r15145.out
date_fmt  = %d%m%y
diff = vimdiff
testcode_vcs = git

[cpmd]
exe = /home/jss43/projects/CPMD/cpmd.x
cleanexe = /home/jss43/projects/CLEAN/CPMD/cpmd.x
inp = cpmd*.inp
extract = /home/jss43/projects/testcode/tools/extractdata.pl -sw --dp 11
timing_output = cpu elapsed
tolerance = KS:1.e-6 PT2:1.e-8
submit_template = /home/jss43/projects/testcode/tools/cpmd-submit.sh
compile = make new
repository_location = gitosis@server:cpmd.git/
vcs = git
```

2.3 jobconfig

jobconfig contains the desciptions of each test and groupings of tests into categories, which allows for subsets of all the tests to be selected quite easily. Each section apart from the categories section contains a description of a test, with the name of each test set to the section name.

The test groups are specified in the categories section. Each entry in the categories section consists of a category name being set to contain a list of tests, which can be specified using the test names and/or other categories. Each test is automatically placed in a category of its own.

There are two special categories: all_jobs contains all jobs unless set otherwise in jobconfig and default is set to be identical to all_jobs unless otherwise set. If no job options are specified on the command line then default job category is run unless new benchmarks are being generated when the all_jobs category is run.

The test descriptions can contain the following options:

dir [string] Set the directory containing the test(s), either as a absolute path or relative to the directory containing the userconfig and job config files. Defaults to a sub-directory of the directory containing the configuration files with the same name as the job name.

type [string] Set the job type. Defaults to default_job_type (if specified in userconfig) or the job type given in userconfig, if only one job type exists.

tolerance [float] Set non-default tolerances for specific items in the output this job. This allows items to be tested using to a lower or higher precision, as needed, and is useful for when iterative procedures cause a greater variance in the final result on different platforms for longer calculations. The format is a space-separated list of key:tolerance entries. For instance, the entry Energy:1.e-12 would require all data items in the "Energy" column of the extracted test output to agree with those of the benchmark output to within 1.e-12 in order for the test to pass.

override_tolerance_option [True/False] Use the specific tolerance values given in the job type and job descriptions rather than the value given on the command line if the –tol option is used. The default value is False.

processors [integer] Sets the number of processors to run the calculation on. If greater than 0, mpirun is used to launch the program with the number of processors specified; if not, then the program is launched in serial (default behaviour).

override_core_option [True/False] Override using the number of processors given via the –core option. The default value is False.

A sample jobconfig file is:

```
[t1]

[test_2]
dir = t2

[test_3]
dir = t3

[categories]
default = t1 test_2
```

The tests in this example reside in the sub-directories t1, t2 and t3. The categories are t1, test_2, test_3, default (containing jobs t1 and test_2) and all_jobs (containing jobs t1, test_2 and test_3).

3 Input and output files

If the output is sent to STDOUT (i.e. no out variable is set in the job type configuration), then testcode assumes the program is run using the command:

```
executable input_file > output_file 2> error_file
```

and substitutes in the appropriate values for the executable and files. Parallel calculations are similarly run using:

```
mpirun -np n_procs executable input_file > output_file 2> error_file
```

where the appropriate value for the number of processors is also provided.

If the output file is set in the job type configuration, then testcode runs the program using the command:

```
executable input_file 2> error_file
```

and similarly for paralllel calculations. The output file is then moved to the test output file names (described below).

The command used to run the tests can be easily changed if required in the run_cmd function.

testcode requires that all tests have the same name for the input file (or rather, the same pattern) so that it is straightforward to find unique output and benchmark filenames. If the input file contains a wildcard pattern (e.g. *.inp), then this is expanded and all files matching the pattern are used as input files to run tests. The tests are labelled according to the matched pattern, e.g. if the input file is specified as test*.inp and a directory contains the input files testshort.inp and testlong.inp, then all output files will be labelled with _short and _long as appropriate. If the output filename is specified using the out variable in the job type configuration and contains a *, then the * is replaced with the label obtained from the input filename, e.g. if out is set to be *.out, then the output in the above example would be written to short.out and long.out respectively.

The test output files are called testJOBTAG.out-ID and the benchmark files are called benchmarkJOBTAG-ID.out, where JOBTAG (possibly a null string) is the tag from the input file (e.g. short and long from the example given) and ID is a unique identifier for the test output (which is related to the date by default) and benchmark output (which defaults to the VCS identifier of the code used to generate the benchmark).

The test outputs and benchmark outputs are both referred to via command line options using the ID string. The –benchmark option takes an ID string and forms the appropriate benchmark filenames and the –revid takes an ID string and forms the appropriate test output filenames. These can be overridden. If –benchmark is given the option “t:ID”, then the testJOBTAG.out-ID files are used as the benchmark. This is useful for using test outputs as the benchmarks temporarily. If –revid is given the option “b:ID”, then the benchmarkJOBTAG-ID.out files are used as the test output filenames. This is useful for comparing different benchmarks.

4 Data comparison

testcode does not analyse directly the output of calculations, but rather compares selected data from the benchmark and test output. The data can be obtained either from labelling lines in the output or from a user-provided data extraction script.

If no data extraction script is provided then testcode contains a simple utility for finding the relevant data (if more flexibility is required then a data extraction script should be written). testcode searches the output for lines with the specified data_tag (default: [QA]) as the first non-space character. The first numerical entry on that line is taken to be the data entry to be tested. For instance, if data_tag is [QA] and the output contains the following lines:

```
Energy converged to 0.001 a.u.
[QA] Energy = 1.20987 a.u.
Calculation complete.
```

then the value 1.20987 will be tested unless the string 'Energy' matches a timing_output entry. The string between the data tag and the value is stripped of additional spaces, colons and equal signs and printed as a column header in the testcode output.

Alternatively an external script can be called to analyse the output from the calculations. The output from the extraction script is assumed to be printed to STDOUT in a (space-separated) tabular format, where the first row and any subsequent rows containing no numbers are assumed to form headers of a sub-table, and so form the keys for the subsequent sub-table. Blank lines are ignored.

Output from the data extraction script is parsed in the following manner:

```
a  b  c
1  2  3
```

refers to a=1, b=2, c=3 (simplest and most common format);

```
a  b  c
1  2  3
4  5  6
```

refers to a=[1,4], b=[2,5], c=[3,6]; and

```
a  b  c
1  2  3
4  5  6

a  b  d
7  8  9
```

refers to a=[1,4], b=[2,5], c=[3,6] and (tested separately) a=7,b=8,d=9 (two sub-tables).

In the case of multiple lines of data output, then the rows of the test output are required to be in the same order as in the benchmark output.

testcode then compares the values from the test and benchmark for each data item to within the specified tolerance. If the values are non-numeric, then a strict equality (as far as python regards equality) is required for a successful test.

If the program already outputs in this format, then the data extraction program can simply be *cat*.

The tolerance used to compare the test data to the benchmark data can varied according to the test and the following order of precedence is used:

1. Tolerance specified in the job description if the override_tolerance_option is set to true.
2. Tolerance specified in the job type description if the override_tolerance_option for the specific job is set to true.
3. Tolerance given via the –tol command line option.
4. Tolerance specified in the job description if the override_tolerance_option is set to false.
5. Tolerance specified in the job type description.
6. Tolerance specified as the default tolerance in the user configuation section in userconfig.
7. Default tolerance value (1.e-10).

5 Options

testcode's behaviour can be controlled via many command line options. It also contains a few utilities which do not run any tests; these can also be selected via the command line.

```
Usage: testcode.py [options]
Test code against previous results.
Default behaviour: test all jobs in the default job category.
```

The options are detailed in the following sections.

5.1 Utilities

These options do not run any tests, but instead offer either information or provide a useful function.

-h, –help Show help message and exit.

-p, –printcats Print available job categories and the test jobs in each category and exit. No other options are useful in conjunction with –printcats.

–tidy=<number of days> Clean up the test directories and exit. Regular testing leads to a large number of output files in each test directory. This option removes all test*.out*, submit*.sh* and testcode_job* files older than the specified number of days from each test directory. No other options are useful in conjunction with –tidy.

5.2 Comparisons to previous tests

These options do not run tests, but instead compare test output from previous tests to the benchmark.

-c, –compare Compare previous tests to their respective benchmarks.

–diff Launch a diff program (specified in userconfig) to compare previous tests with their respective benchmarks.

The tests to be compared can be specified via the –revid option (see below). If the test id is not given, then the last set of test outputs generated are used.

5.3 Main options

-b <revision id>, –benchmark=<revision id> Specify a different benchmark file by the revision id, as described above.

–config=CONFIG Set path to config files. Default: current directory.

–cores=<number of cores> Launch program using mpirun and the specified number of cores if the number provided is positive. The default behaviour is to run in serial mode, unless a default number of cores is specified for a given job in jobconfig.

-e <executable>, –exe=<executable> Set executable location(s). The executable location can be specified in four different ways:

1. Executable location is specified as an alias as set in userconfig. This sets all job types to use this alias.

2. Executable location is specified as a path to an executable. Warning: this sets all job types to use this executable.
3. job_type:alias is used to set the executable location for the given job type to be that corresponding to the alias as given in userconfig.
4. job_type:executable is used to set the executable location for the given job type to be the specified executable path.

Either an absolute path or a path relative to the current directory can be used in the last two cases.

-j <job category>, –jobcat=<job category> Run tests only on the jobs in the specified job category. Can be specified multiple times to run tests jobs from multiple categories. A test is only run once, even if it is specified multiple times through the –jobcat and –jobregex options.

–jr=<regular expression>, –jobregex=<regular expression> Run tests on all jobs with names that match the specified regular expression. Can be specified multiple times. A test is only run once, even if it is specified multiple times through the –jobcat and –jobregex options.

-n <number of concurrent threads>, –nthreads=<number of concurrent threads> Set the number of tests to run concurrently. This should be used only on platforms with multiple cores/processors and care should be taken if it is used in conjunction with running parallel tests. The default behaviour is to run one test at a time.

–queue Run the tests by submitting them to a cluster queue using qsub. A template submit script must be specified in the relevant job_type section(s) of userconfig. testcode replaces the following strings in the submit file:

- testcode.jobdir with the directory that contains the test.
- testcode.jobinput with the input filename of the test.
- testcode.joboutput with the output filename of the test.
- testcode.runcmd with the command used to run the test.

All other necessary infrastructure (e.g. setting environment variables, copying files to/from a local workspace etc.) must be provided in the template submit file.

-q, –quiet Minimalist output. The default is for verbose output, which contains information about the executable used, tolerances, the data extracted from the output and which (if any) tests failed. In quiet mode, a full-stop is printed if a test passes and a F is printed if a test fails. After all the tests are run, the number of passed tests and the total number of tests are printed. This allows for the testcode output to take up only a few lines at most.

-r <test id>, –revid=<test id> Choose a suffix that is appended to the test output filename to identify the set of tests used. The default is to generate a unique date-based filename.

–runbench Run jobs and use the test outputs to produce a new set of benchmarks.

-t <tolerance>, –tol=<tolerance> Change the default tolerance for testing agreement between test jobs and benchmarks. The default behaviour is to use the tolerance values specified in the configuration files.

6 Exit status

testcode exits with a return code of 1 if one or more tests are not passed and 0 otherwise.

7 Automated testing

checktest.py can be used to obtain the latest version of the codebase(s) and test suite from a repository, compile the program(s) and run the test suite with a variety of compilers. This is very useful when run from cron to provide regular testing.

checktest assumes the test suite and the source code are under source code control.

The steps checktest performs for each platform/compiler are:

1. Revert and update the test suite. This is performed after the testcode module has been loaded and so changes to the testcode source are not taken into account on this run of checktest.
2. If necessary, check the source code out of a repository for each job type.
3. Revert and update the source code for each job type to ensure the source code is a clean copy of the repository version.
4. Run make clean (using the command given in userconfig) on each job type to remove any previously compiled objects. The command is run in the directory of the executable of each job type.
5. Compile the binary for each job type using the command specified in userconfig. The command is run in the directory of the executable of each job type.
6. Checks the timestamp on the executable(s) to ensure that new executable(s) have been produced.
7. Run the test suite.

The output from running checktest is stored in a log file in logs/month-yyyy/dd-mm-yyyy.log, where logs is a subdirectory of the directory containing the test suite. If checktest is run multiple times in a day, then the log filename is appended with -i, where i is an integer so that the log filename is unique.

An email report can also be sent out.

checktest settings are given in the checktest_config file, which is of the same format as jobconfig and userconfig.

7.1 checktest configuration file

The checktest configuration file, checktest_config, contains at least one section ('user') containing the user settings and at least one section containing the information about the compiler/platform used to compile the program(s). There can also be a section ('email') which contains the settings required to send out email reports.

Valid checktest_config settings are:

[user] Start section containing user options.

testcode_path [string] Set the path to the directory containing testcode.py. Required.

environment_modules_path [string] checktest knows how to use the environment modules system (see http://modules.sourceforge.net/). environment_modules_path sets the path to the directory containing the python module (assumed to be called environment_modules.py) which initialises the modules system. If environment_modules_path is not specified then the modules system is not used.

testcode_config_path [string] Set the path containing the configuration files for testcode. Default: same path as testcode_path.

exe_alias [string] Set which executable alias (as given in userconfig) should be tested. Note that any uncommitted changes to the source code will be reverted. Default: exe alias.

[email] Start section containing email options. If not given then no emails are sent.

sender [string] Email address of user "sending" the emails. Required.

to [string] Email addresses of recipient(s) as a space-separated list. Required.

cc [string] Email addresses of cc recipient(s) as a space-separated list. Default: none.

email_server [string] SMTP server address. Required.

format [string] Set the format of the email. The only acceptable options are short and long. The full output of checktest is sent if the format is long. The full output of checktest is only sent for platforms on which tests fail if the format is short. Default: short.

mail [string] Set when the email is sent. Acceptable options are failure, mail and nomail which cause an email to be sent if any tests fail, cause an email to always be sent and cause emails to never be sent respectively. Default: failure.

All other sections are assumed to contain a platform on which the test suite is to be tested.

[platform_name] Set the name of the platform. The platform name replaces the PLATFORM keyword in the compile command, e.g. the compile command:

```
make ARCH=PLATFORM
```

is run as:

```
make ARCH=platform_name
```

modules [string] Set the modules (as a space-separated list) to be used if environment modules are being used.

job_types [string] Set the job types to be tested on this platform. Default: test all job types.

job_categories [string] Set the job categories of tests to be run on this platform. Default: all_jobs.

options [string] Set any additional testcode options. Default: none.

7.2 checktest options

In a similar fashion to testcode, checktest's behaviour can be altered by the command-line options.

```
Usage: checktest.py [options]
Compile latest version in repository and run test suite.  Email if there's a problem or the tests fa
```

The options are:

-h, –help show this help message and exit

-c COMMENT, –comment=COMMENT Add a comment to the output (e.g. explaining multiple tests in a day).

–config=CONFIG Set the location of the configuration file. Default: checktest_config in the same directory as checktest.py.

-f, –failure Send email only if tests fail. Default behaviour unless otherwise specified in config file.

-l, –long Set email format to be long (complete test output).

-m, –mail Send email even if all tests pass.

-n, –nomail Do not send email even if tests fail.

-p PLATFORM, –platform=PLATFORM Run on platform specifed. Can be specified multiple times.

-s, –short Set email format to be short: do not include test output for platforms which pass. Default format unless otherwise specified in the config file.

-v, –verbose Write actions to screen. [Default: False]