A short guide to the FHI-aims regression test tools

Arvid Ihrig [ihrig@fhi-berlin.mpg.de]

October 31, 2016

This document is intended to give you a short overview about the features provided by the regression test suite in FHI-aims. The core of the regression test is a python script package which allows you to easily run the regression tests in a highly automated way. You can either compare two arbitrary FHI-aims binaries directly or compare one to an archived set of results. The tests can be either run on your local machine or outsourced to a remote computational cluster. The result evaluation is always performed locally and therefore independent of the available software on the cluster you use.

This documentation will give you a short overview of the features this script offers and how to use it. Furthermore, the second part gives a short guideline for developers on how to add new test cases.

System Requirements

To run the automated regressiontests on your machine, you only need a compiled FHI-aims binary and a standard python installation, version 3.2 or newer. If you want to calculate the test cases on a remote system, there are no special software requirements besides those of FHI-aims itself. SSH-authentication is supported both with key-pairs and passwords.

However, it might be that the queuing system of your cluster is not yet supported by this script and an automated submission to the job scheduler is therefore not possible.

Part I.
User's Guide

1. Structure of the Regression Tests

The regression tests in FHI-aims can be divided into three different building blocks:

TestCase this is the basic block and contains an FHI-calculation to test (which can consist of several sub-calculations)

TestParameter these are subelements of the TestCases and define the quantities that should be checked for their consistency, e.g. total energy, number of SCF-steps etc

TestSuite these objects are used for logical grouping of the TestCases

A more detailed explanation of these objects and their usage can be found in Developer's Guide. For a normal user, the TestSuite is the most interesting object, because you can exclude individual TestSuites from your regression test.

2. Basic Usage of the regression test tools

Your main tool will be `regressiontools.py`, which has four different operational modes:

full run the regression test locally and then evaluate it

local run the regression test locally

remote send the regression test to a remote system and register it with the job scheduler

analysis evaluate the results from a previously executed regression test

To figure out the expected input for each mode, call the desired mode of the script with the -h option, e.g. as

```
./regressiontools.py full -h
```

to see a detailed help on all parameters known in the full mode. Most parameters are optional and have useful default values, the only thing you need to specify in all modes, except analysis, are the two revisions you want to compare. You can then run the regression test by calling

```
./regressiontools.py <mode> [arguments for chosen mode...]
```

The basic arguments are the two executables you want to compare, first specify the reference, then the test-binary. Instead of an executable, you can also specify a folder as reference. If this folder has the same structure as the script's working directory, the reference subfolder will be used. To compare to the values from the latest official FHI-aims release, you can use the folder `references_lastrelease`.

3. Storing often used commandline parameters

If you often use non-standard settings, you can save them to a file and tell the script to parse this file instead of retyping the whole series of flags each time. For example, if you do not have a FHI-aims version with Scalapack- support available, you likely want to skip the TestSuite for Scalapack-exclusive functionality. To do so, you could call the program with

```
./regressiontools.py full --exclude="ScalaPack only" <REF> <TEST>
```

Instead, you could also create a new file, let us call it `config.ini` with the content

```
full
--exclude="ScalaPack only"
--cpus=4
references_lastrelease
```

Here, we request the "full" mode with 4 cores while excluding the ScalaPack TestSuite and using the output files from the last FHI-aims release as a reference. Please note that each line in this file is interpreted as one parameter on the commandline. `--cpus 4` as one line will thus not work, you must use the assignment with the equal sign. This file you can now use in your program call by prefixing the file name with an @.

```
./regressiontools.py @config.ini <TEST>
```

Note that you can still specify more arguments, like the test executable in this example, on the command line. Also keep in mind that the order matters if you add positional arguments (operation mode, executables) to the configuration file.

After starting the regression test locally, the script will give you an update whenever it proceeds to calculate the next TestCase. After all calculations have been finished, the analysis will begin (if full mode was chosen) and you will see the results for the TestParameters that have been defined for the current TestCase in a compact table view. You can proceed to the next TestCase by pressing enter. In addition all results are also stored in a logfile, which defaults to `regressionlog.txt`.

4. Regression tests with custom test parameters

If the default setup of the regression tests does not match your need, you can create copies of the `defaultparameters.xml` and `testsuite.xml` files and modify[1] these according to your needs. You can then use the `--configuration` and `--defaulttests` commandline flags to specify which version to use. This is particular useful because you can have two variants with the same TestCases, but different TestParameters (e.g. tighter criterions than the official regression test, or your custom profiling statements) and then use the analysis mode to evaluate both of them without having to rerun the calculations first.

[1] see the Developer's Guide for details about them

Part II.
Developer's Guide

5. Adding a new case for the Regression test

New TestCases can be defined in the `testsuite.xml`. This xml-document contains the hierarchical structure of the regression test. Let's start by having a closer look at the the declaration of the testcases in the `testsuite.xml`:

```
<testsuite name="DFT, isolated molecules">
    <testcase name="N2, PW-LDA"
        folder="N2.pw-lda.superpos_rho"
        skip="Initialization Energy (eV)"
    />
    [...]
    <testcase name="H2O, ESP-charges, PBE" folder="H2O_esp_cluster">
        <testparameter name="fitted ESP charges"
            importance="mandatory"
            comparison="cmp_float.abs_diff(1E-6, 3)"
            regex="Atom[ ]+[0-9]+:[ ]+[A-z]+, ESP charge:[ ]+([-0-9.]+)"
            occurence="1:3"
        />
        <testparameter name="RRMS"
            importance="mandatory"
            comparison="cmp_float.abs_diff(1E-6, 3)"
            regex="RRMS:[ ]+([-0-9.]+)"
            occurence="1"
        />
    </testcase>
</testsuite>
```

excerpt from testsuite.xml

The toplevel-nodes are the `testsuite`, which contain one or more `testcase`, which in turn may contain `testparameter` nodes. To add your test first find the TestSuite that fits best[2] and add a new `testcase` node to it. You can insert it at any place within the `testsuite` node, because the script internally orders the testcases by name[3].

As you can see, all nodes contain additional attributes which are denoted by a `key="value"` pair. Each type has its own specific keywords that will be explained below. As you can see from the `N2, PW-LDA` testcase, it has not defined any testparameters. This is possible, because a second file, `defaultparameters.xml` contains the definitions for testparameters which are universal enough to apply to all testcases:

[2] If necessary you can also create a new one, but there should not be a testsuite for each testcase.
[3] The same applies to the order of the `testsuite` nodes within the root node "regressionsuite".

```
<defaults>
    [...]
    <testparameter name = "Total calculation time (sec)"
        comparison = "cmp\_float.rel\_diff(5,2)"
        importance = "optional"
        regex = "Total time[ ]+:[ ]+([-0-9.]+) s"
    />
    <testparameter name = "SCF HOMO energy (eV)"
        comparison = "cmp\_float.abs\_diff(1E-6)"
        importance = "mandatory"
        regex="Highest occupied state \(VBM\) at[ ]+([-0-9.]+) eV"
        occurence="-1"
    />
    [...]
</defaults>
```
excerpt from defaultparameters.xml

As you can see these parameters here have exactly the same attributes as those in the `testsuite.xml` and are added implicitly to all testcases, unless they are explictly skipped by the `skip` attribute on a testcase. The only difference is that they are now direct children of the root node defaults.

It is possible to define a default folder and a default output file for all default parameters. This can be useful if the relevant calculation is in a subfolder. To define a default folder and file, use the `defaultfolder` and `defaultfile` attributes:

```
<testsuite name="Example with subfolders">
    [...]
   <testcase name="3-Step DFT example" folder="dft_3step"
        subfolders="step1,step2,step3"
        defaultfolder="step3"
        defaultfile="aims.custom.out"
   />
</testsuite>
```
example for default output folder

In this example, the outputfile for all default testparameters is taken from `step3/aims.custom.out`. If you need different testparameters from different output folders, you need to define custom testparameters even for the default parameters and use the appropriate file for each of the parameters.

5.1. Defining a new testcase

To define a new testcase, just add a new `testcase`-node into the chosen `testsuite`. The allowed attributes for this node are:

name (mandatory) The name of this testcase, must be unique within the current test-suite.

folder (mandatory) The folder with the input data, relative to the source directory specified when calling the program, which is by default `./testcases`.

subfolders (optional) If the test cases is composed of several subcalculations, you can specify a comma- separated list of subfolders (relative to the folder argument) to compute.

skip (optional) If you want to skip any of the default parameters for this testcase, you can specify them here as a comma-separated list of their names.

5.2. Adding custom test parameters to it

Now that you have your `testcase`-node, you can start adding `testparameter`-nodes into it to include the quantities of interest in the regression test. Each test parameter first must define how to acquire the data to compare. This can either happen by extracting values from the target file using one of three different methods:

single values specify a regular expression pattern to extract single values of interest from the output file, e.g. the DFT total energy [specify `regex` argument]

tables specify a regular expression pattern for a table header and then compare the following lines, e.g. DFPT IR frequencies [specify `regex` and `tablerows` arguments]

files compare the whole file, e.g. cube files [do not specify `regex` argument]

In all cases you can specify a custom comparison function and for single values and table you can also specify which occurence(s) in the sourcefile should be compared. The allowed attributes are:

name (mandatory) The name of this testparameter, must be unique within the current testcase.

file (optional) The path to the file that contains the data of interest, relative to the folder of the current testcase, optionally you can also specify a shell-pattern to repeat this test for all matching files. (defaults to `aims.out`)

comparison (mandatory) The comparison function you want to import, the syntax will be explained below

importance (mandatory) The importance of this test, can be either "mandatory" (must succeed), "optional" (failing has no consequences except for a warning) or "consistent" (as mandatory, but counts as successful if reference and test yield the same error, e.g. a missing file)

regex (optional) the regex pattern to search for, if none is specified, the file comparison mode is used (see the section below for some remarks on the regex pattern)

tablerows (optional) the number of rows in the table that should be extracted. If specified, the given regex will be interpreted as the table header pattern and the table comparison mode is used.

occurence (optional) The set of regex-matches to extract, given as `start:stop:step` tuple, where stop and step are optional. If the elements are positive, the first occurences in the file will be fetched, negative elements will fetch the reverse-counted occurences. Mixed positive and negative elements are not allowed.

Input Syntax for Comparison functions

All comparison functions are located inside the `comparisons` folder of this toolset. In there you will find several files with comparison functions for floats, integers, strings and files. These can either be simple python functions (as in the string case) or *function closures*. A function closure is a function that when called returns another function which was defined inside the called function. The advantage of this approach is that the inner function has access to the variables of the outer function and thus all specific data can be stored on the outer function, giving the inner functions a unified call signature. Let's have a look at the absolute value difference integer comparison as an example:

```
def abs_diff(tolerance):
    """
    returns true, if the two objects can be converted to ints and their
    absolute difference is smaller than the given tolerance.
    """
    tolerance = int(tolerance)
    msg = r"abs diff =  " + "(tol = %i)"%tolerance
    msg_file = r"abs diff =  " + "(tol = %i) [file=]"%tolerance
    def inner_int_absdiff(file, obj1, obj2):
        """inner function of closure"""
        diff = abs(int(obj1)-int(obj2))
        if file:
            return diff <= tolerance, msg_file.format(diff, file)
        else:
            return diff <= tolerance, msg.format(diff)
    return inner_int_absdiff
```

cmp_int.abs_diff

As you can see the function inner_int_absdiff(file, obj1, obj2) has been defined inside the outer function def abs_diff(tolerance) and serves as the return value. Note that the variable tolerance is only defined in the outer function, but can be referenced without previous definition in the inner function. By calling the outer function with a specified

tolerance, we will now obtain a comparison function that takes three strings (file name for info messages, and the two values to compare) and tries to convert them into integers. If their absolute difference is smaller than the tolerance, the function returns True. The practical advantage of this approach is that you can easily specify those comparison-specific parameters in the input file. For example the value `cmp_float.abs_diff(1E-6)` as a comparison function tells the program to use the function `abs_diff` from the file `cmp_float` inside the `comparisons` folder and call it with the argument 1E-6 to obtain the comparison function.

Input Syntax for Regular Expression Extraction Pattern

The regex pattern used in this program use the `re` module from the python standard library. Please note that any parentheses, brackets and pipes that occur in the line you are searching for, must be escaped with a backslash. For an introduction about regular expressions, please refer to the python documentation of the `re` module.

Single Value Extraction

In this mode, each regex pattern must contain exactly one matching group, which is enclosed in parentheses. If the current line matches the regex, the string inside the matching group will be given to the comparison function for further evaluation.

Table Extraction

In this mode, the regex must contain no matching group. If the current line matches the pattern, it will be considered as the header of the table and the following lines, specified by the `tablerows` argument, will be extracted. The header and all extracted rows will then be compared.

Examples

This parameter fetches the total computational time from the aims.out file and makes a relative float comparison with an accepted deviation of 5 percent. However, the whole test is optional, so failing this criterion only causes a warning.

```
<testparameter name = "Total calculation time (sec)"
    comparison = "cmp_float.rel_diff(5,2)"
    importance = "optional"
    regex = "Total time[ ]+:[ ]+([-0-9.]+) s"
/>
```

definition of the total time testparameter

This parameter fetches the last occurence of the HOMO level energy and the compares it by means of the absolute difference. Note the escaped parentheses around the VBM part of the search pattern.

```
<testparameter name = "SCF HOMO energy (eV)"
    comparison = "cmp_float.abs_diff(1E-6)"
    importance = "mandatory"
    regex="Highest occupied state \(VBM\) at[ ]+([-0-9.]+) eV"
    occurence="-1"
/>
```
definition of the SCF HOMO energy testparameter

This parameter takes a bunch of bandstructure files and compares each reference/test file pair by means of a table comparison file diff function (with 50 columns and a tolerance of 1E-4).

```
<testparameter name="Band Structure"
    importance="mandatory"
    comparison="cmp_file.table_file_absdiff(50, 1E-4)"
    file="band*.out"
/>
```
definition of the band structure testparameter

And finally, this parameter extracts a table with 6 rows after it found the header for the vibrational frequencies.

```
<testparameter name="Vibrational Frequencies"
    importance="mandatory"
    comparison="cmp_table.table_float_absdiff(3, 1E-4)"
    regex="Mode number      Frequency \[cm\^\(-1\)\]   IR-intensity"
    tablerows="6"
    occurence="-1"
/>
```
definition of the DFPT vibrational frequencies testparameter

6. Defining a new testsuite

This is the simplest part. Because the testsuite is essentially only a wrapper object for ordering testcases, it only has one allowed attribute, it's name:

name (mandatory) The name of this testsuite, must be unique within the document.

7. Defining a custom comparison function

As described in Input Syntax for Comparison functions, most comparison functions in this program are function closures. If the set of provided functions does not cover your needs, you can easily add your own functions. Simply define your function closure and add it to any of the existing files, or create a new one if the provided categories do not

match your application. Afterwards you can directly use your function, there is no need to add extra import statements anywhere.

Please note that you can use the `nonlocal` statement to explicitly declare in the inner function that you want to overwrite the variables listed after the statement in the outer scope, instead of overshadowing the outer variable with a local one after assigning a value to it. This functionality allows you to create very complex functions, e.g. by storing previous lines in the closure of a file-diff function.

The signatures for new functions (in case of closures this means the inner functions) should be:

```
def regex_diff(string filename, string obj1, string obj2)
def table_diff(string filename, string[] table1, string[] table2)
def file_diff(string filename, int line, string line1, string line2)
```
required signatures for comparison functions

Note that the "file" argument for `regex_diff` and `table_diff` will be None, unless a shell-pattern was specified for the input file and matched more than one file. In the table comparison mode, the two string-lists will include the header of the table.

The return value should be a 2-tuple `(boolean equal, string info)` in both cases. The boolean indicates whether the values have compared equal and the info string contains any extra information you want to show in the last column of the output table. Note that long info strings will be cropped to the size of the terminal, but the uncropped output will be written to the logfile.

8. Defining a custom remote communication protocol

The first version of this script only supports remote clusters that run the Sun Grid Engine as job scheduler and allow SSH connections. If you want to add support for another queuing system/protocol, you can have a look at `core/remotes/abstractremote.py` and `core/remotes/gridengineremote.py`. Once you have familiarized yourself with the `AbstractRemote` API, you can implement a new subclass of this abstract class that has the required functionality for your environment. Any subclass of `AbstractRemote` will automatically be recognized as such, if you add an import statement to submodule, i.e. to `core/remotes/__init__.py`.