Science & Technology Facilities Council

www.scitech.ac.uk

# TolDiff: A regression testing tool for validating results of scientific codes

Huub J. J. van Dam
STFC Daresbury Laboratory, Computational Chemistry Group

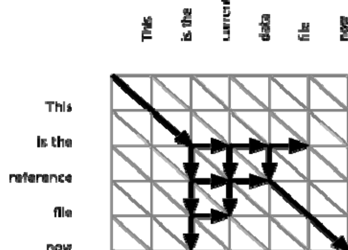
## Introduction

Testing software is crucially important to maintain the quality of a code while it is being ported or further developed. Although there are many aspects to testing ultimately a number of test cases have to be run and the results verified.

The simplest way to verify a result is to compare the output against a reference result that is known to be correct by running the Diff program [1]. In practice this approach does not work well for scientific codes. The reason is that there are a number of data elements that may change without invalidating the result. Typical examples are:

- Floating point numbers that change in the last few printed decimals.
- Meta-data such as the name of the machine the job ran on, the user name, and the date and time the job started.
- Timing information.

*Figure 1: Schematic diff procedure*




To circumvent the file comparison problem often a "filter" approach is used. In such an approach the important bits of the output are extracted and fed to a checker. The down sides to this approach are:

- Code specific filters need to be set up and maintained.
- Errors that express themselves in "un-important" parts are missed.

The way forward proposed here is based on the realisation that Diff is ideal apart from the fact that it reports all differences, even the known un-important ones. Therefore I propose a Tolerant Diff program named ToIDiff [2] that can suppress known un-important differences. The idea itself is not new and programs based on it date back to 1988 when Nachbar proposed Spiff [3], and further examples are NumDiff and NDiff.

ToIDiff is similar to alternative approaches in that compares text as text and numbers as numbers. As numbers it recognises integers, floating point and complex numbers.

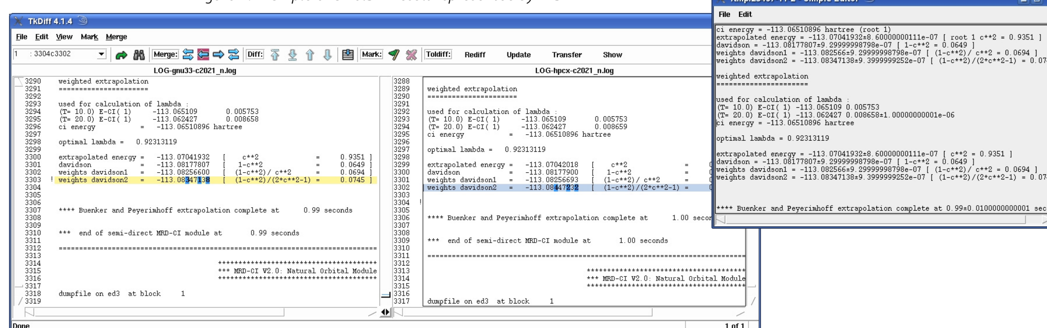
ToIDiff is significantly different from other solutions in that it:

- Can tolerate any kind of changes that Diff can detect, including changes to text and numbers but also the appearance of extra text or numbers, or the disappearance of text and numbers.
- Can record tolerable differences automatically avoiding manually maintaining tolerance data.
- Produces output that closely matches that of Diff so that it can be used with tools designed to work with the latter such as TkDiff [4].

## The ToIDiff approach

The general way ToIDiff is used is to first create a reference output file that is correct. Subsequent outputs of the same calculations can then be compared against the reference file. The results of this comparison are fed back to the tester who decides whether the differences are significant or un-important. If the differences are un-important the tolerances are updated automatically running ToIDiff with the corresponding command line flag. Repeating this procedure all acceptable changes are discovered and recorded over time.

*Figure 2: Example of a ToIDiff result represented by TkDiff*



Apart from the usage above there are two more important use cases leading to the following four main use cases.

**Diff**

Compares an output file against the reference file exploiting any known tolerances and reporting all potentially important differences.

**Update**

Compares an output file against the reference file and adds any differences to the tolerance file so that they are ignored in future.

**Transfer**

If code developments change the reference output significantly ToIDiff can compare the new reference file against the original one. Where possible it can transfer known tolerances to a new tolerance file associated with the new reference file.

**Show**

To check what ToIDiff does it can show the tolerances in the context of the reference file for inspection.

## Implementation

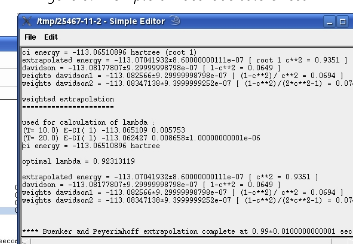
Underlying the above use cases is the Diff algorithm by Meyers and Ukkonen [5]. The algorithm is illustrated in Figure 1 where the reference file is displayed vertically and the output or data file is displayed horizontally. The aim of the algorithm is to find a path from the top-left-hand corner to the bottom-right-hand corner with as many diagonal steps as possible. However, a diagonal step is allowed only if the two items connected by the end point match. In its implementation the algorithm will progress a list of searches extending each search by as many diagonal steps as possible and then adding a horizontal or vertical step. The first search to hit the end point is the desired solution.

In ToIDiff the above standard algorithm as well as two variations on it needed for the tolerant Diff and the Update procedure have all been implemented in Python [6]. Obviously for performance reasons uses of the standard algorithm are handed to native Diff implementations through a system call when ever possible.

**Diff**

In ToIDiff the Diff procedure is derived from the standard Diff in a simple way. First the reference file and output files are broken up into tokens.

*Figure 3: Example of recorded tolerances*



The two token lists are then compared using the standard Diff algorithm. The sections that differ are then analysed with a Diff algorithm that is modified to accept matches for tokens that differ no more than the tolerance set for the token in the reference file. Remaining differences, if any, are then filtered based on tolerable additions or deletions of tokens. The resulting path is passed to the Diff report generator. Figure 2 shows an example of a Diff result.

**Update**

The Update procedure follows the same steps as the Diff procedure. However instead of producing a Diff report the final differences are analysed to find best matching tokens. This involves a modified Diff algorithm in which from a given point all possible paths are tried, horizontal, diagonal and vertical. With a diagonal step a score is computed based on how well the tokens at the end point match, allowing for any pre-existing tolerances. For a pair of string tokens the score is based on the number of matching characters, for numbers the score is based on the number of matching leading digits. In the end the path with the highest overall score is selected. This path is subsequently used to compute new tolerances for tokens that were matched, set a delete tolerance for tokens that were not matched in the output file, and set the number of tolerable additional tokens if tokens were identified in the output file that did not match any tokens in the reference file. Figure 3 shows examples of tolerances recorded in this way.

**Transfer**

This procedure is essentially the same as the Diff procedure applied to the reference file and a new reference file. The only difference is that the resulting path is fed to a routine that uses this as a mapping between tokens in the reference file and the new reference file to copy tolerances across. Any tolerances on tokens that weren't mapped onto tokens in the new reference file are lost.

## Performance

With respect to the performance of ToIDiff there are two obvious questions:

1. As it is written in an interpreted language is it fast enough to be usable?
2. As the approach relies on repeatedly updating tolerances as more un-important differences are discovered does it converge fast enough?

To answer these questions the subset of the GAMESS-UK [7] validation suite known as chap2 (short for "chapter 2") has been run on 5 different platforms. Of the 5 sets of results one was selected as the reference set, and tolerances with respect to the other sets were computed. Each sets of results contained 176 output files totalling half a million lines of data. All 120 permutations were considered and the results averaged. The results are shown in Figures 4 and 5. From Figure 4 it is clear that Diffing or Updating all files takes just over one minute or on average less than 0.5 seconds per file.

Figure 5 shows with every update the number of differences detected in the next round is reduced by a third. This means that starting from almost 35000 differences after 29 updates only half a difference should be found in the whole set of results. In practice most of the variation stems from the large amount of timing data present in the outputs. Most of the variations in the timings can be discovered automatically simply by rerunning the calculations a number of times. A scale factor can be applied when the tolerances are updated to speed the convergence up.

*Figure 4: Time required for ToIDiff operations as function of the update number*

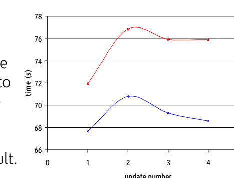
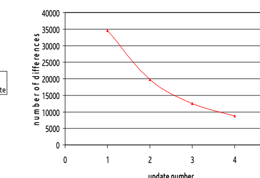



*Figure 5: The of differences reported as function of the update number*




## Conclusion

Toldiff is suggested as a code independent and practical tool for comparing outputs in regression testing of scientific applications. Its strengths are rigorous checking and minimal tester effort required to get started.

## References

[1] W. Miller, E.W. Meyers, "A file comparison program", Software – Practice & Experience **15** (1985) 1025-1040, doi:10.1002/spe.4380151102.

[2] ToIDiff, http://sourceforge.net/projects/toldiff/ [16 June 2008].

[3] D. Nachbar, "Spiff--A Program for Making Controlled Approximate Comparisons of Files". In Proceedings of the Summer 1988 USENIX Conference (San Francisco, CA, June 21-24). USENIX Association, Berkeley, CA, 1988, pp. 73- 84.

[4] TkDiff, http://sourceforge.net/projects/tkdiff/ [16 June 2008].

[5] E.W. Meyers, "An O(ND) difference algorithm and its variations", Algorithmica **1** (1986) 251-266, doi:10.1007/BF01840446;

E. Ukkonen, "Algorithms for approximate string matching", Information and Control **64** (1985) 100-118, doi:10.1016/S0019-9958(85)80046-2.

[6] Python, http://www.python.org/ [16 June 2008].

[7] M.F. Guest, I.J. Bush, H.J.J. van Dam, P. Sherwood, J.M.H. Thomas, J.H. van Lenthe, R.W.A. Havenith, J. Kendrick, "The GAMESS-UK electronic structure package: algorithms, developments and applications", Molecular Physics 103 (2005) 719-747, doi:10.1080/00268970512331340592;

GAMESS-UK, http://www.cfs.dl.ac.uk/gamess-uk/ [16 June 2008].