“Computational Reproducibility: the Elephant in the Room”

Les Hatton and Michiel van Genuchten

Since 2010, we have been editing articles for this column from all over the world. The theme has essentially been the same – where does software appear, in what quantity and using what technologies ? We have had a vast range of responses all the way from the realms of the very large - space exploration, to the very small - the search for the Higgs boson. In between, we have touched on many practical systems such as air-traffic control, banking in emerging nations, navigation and general automobile systems.

One common factor repeatedly emerges, surprising even ourselves after careers working with software-based systems, and that is the sheer size or whatever other word you choose to use, of software which has been deployed. Thirty years ago, systems of more than a million lines of source code were comparatively rare. They were rare for a good reason as they present formidable challenges to development and later in their life, to their maintenance and in particular their reliability.We have repeatedly tried to present the economic consequences of such systems (Genuchten and Hatton, 2011). As the columns have shown however, million line systems written in whatever technology are now common-place. Indeed with the growth of software stacks combining rich functionality, systems of tens of millions of lines of code are appearing, (Mossinger 2010, Wester and Koster 2015).

This begs an interesting question. Have our testing procedures and methods of defect elimination kept up with the inexorable growth of around 20% per year (Hatton, Spinellis and van Genuchten 2017) ? We will discuss this in one particular manifestation as it appears to be exacerbating an existing crisis in the scientific world and that is the problem of reproducibility. Reproducibility is a very simple concept. In essence, all we ask in science is that scientific results can be independently repeated to some acceptable level of significance. This is how trust in a result grows. ~~It is causing~~ sufficient problems in science on its own even without including the effects of computation as we shall see, but we are now adding a deep and unquantifiable layer of uncertainty to this with the growing influence of computation in scientific discovery.

The scientific method

The scientific method has emerged in the last two hundred years of so. It led to a revolution in science but is widely misunderstood and is now in danger of being trampled underfoot by the growth of social media in the 21st century with it’s up and down voting systems and pseudo-numeric statistically illiterate conclusions, the relentless effects of which have had the consequence of elevating opinion to the same level as evidence. As social media is used increasingly to discuss science, there is a particular danger in this.

Perhaps the cleanest insight into the scientific method came from the philosophers of the 20th century such as Popper (1959) and Kuhn (1962). In particular Popper re-iterated the essence of the scientific method which was to promote *falsifiability*. A scientific experiment develops a theory to predict some aspect and then attempts to falsify it using experimental data. Failure to falsify it

means the theory lives to fight another day. If the results cannot be independently reproduced to some acceptable level of statistical significance, the theory is falsified. Over subsequent years and decades, good theories continue because additional experimental evidence fails to falsify them. Bad theories get broken and fall by the wayside. This does not of course mean the "good" theories are correct – it was centuries before Newtonian physics was found wanting by relativistic arguments - but it does mean that they accumulate trust. For Newtonian physics, it is a good enough approximation to the world we live in that nearly all our engineering structures depend on it in some way. It only fails when our world intrudes on the very large or very small as occurs in the GPS satellite navigation system for example, in which general relativistic effects and time dilation have to be taken into account.

Even without considering software, Science has a problem with regard to reproducibility. An estimated cost of irreproducible biomedical research is $28 billion/year (Freedman, Cockburn and Simcoe, 2015) and "*Currently, many published research findings are false or exaggerated, and an estimated 85% of research resources are wasted*" (Ioannidis 2014). Irreproducible results can have profound effects in the medical world where they may lead to questionable protocols and in some cases significant loss of life, (Huston 2014).

So what further complications does software introduce ?

Computational reproducibility

Like all the disparate products we have covered in these columns, software written to support scientific enterprise has been growing rapidly for exactly the same reasons – cheap and extremely powerful hardware, the explosive growth of open source software including compilers for languages such as C, C++, Ada and Fortran, and interpreters for the new(er) ones on the block, Python, Perl and Java including the very widespread availability of excellent libraries. The bottom line here is that to parse a protein sequence or analyse an email now is a matter of a few hundred lines of Python or whatever. Added to this we have higher level and very sophisticated open source systems for handling databases such as MySQL, (itself written in mountains of C), so we can store and apply computation to vast quantities of data. Perhaps even more significantly, we have open source statistical analysis programs of great sophistication such as R (also written in mountains of C), which include the ability to produce the kind of wonderfully elaborate 3-D plots which have become *de rigueur* in scientific publications. As a result of all this effort by volunteers around the world, writing such analysis and data capture software is precisely what scientists have been doing for the last decade or two and in huge quantities in fields as diverse as space research (Nagy et al 2016, Zwart and Bédorf 2016) and the search for the Higgs Boson, (Rousseau 2012). There is a problem however.

The problem with software

Perhaps 30 years ago, we had a wide understanding of Popperian falsifiability in the field of *software testing*. Indeed the whole point of testing was not to affirm that software behaved correctly but *to find those circumstances and conditions under which it failed to perform correctly;* to falsify its premise of working correctly. As Myers (1979) states "Testing is the process of executing a program with the intention of finding errors". In short, the whole idea was to *break* it, and hand the pieces back to the developer, (trying very hard not to be smug about it).

We seem to have taken a step backwards from this. Perhaps overwhelmed by the massive and continuing growth of software, we have invented agile and other methodologies whose intention is primarily to converge on an acceptable solution with the end-user which may have little to do with the end-user's original aspirations, either in its delivered or its future behaviour. This may be

acceptable in the case that the software supports functions such as publishing a picture on a timeline or sending a message from one user to another. It is not good enough if the function is to control a robot operating on a patient or an autonomous car in city traffic. As is the way with software technologies, these have unfortunately been burdened with an arcane vocabulary all of their own obscuring the clinical essence of falsifiability. As a result, many still believe erroneously that testing in so far as it exists at all, is there to prove that the software "works".

We need to be quite blunt about this. Computer science has no technology to guarantee the absence of defect; we never have had such a technology and in all likelihood we never will. We don't even have any easily applicable technologies to quantify the impact of residual defects which we inevitably introduce and fail to remove before delivery, and yet the amount of software source code generally continues to grow by about 20% per year as we saw above. We can therefore assert that scientific experiments which rely heavily on computation **cannot** follow the scientific method unless the complete means to reproduce the computational results independently, is provided as an essential part of the whole. Paraphrasing the wise words of the geophysicist Jon Claerbout at Stanford, without the means to reproduce them, the results are merely an advertisement of scholarship and not the scholarship itself (Claerbout 1994). To be really blunt about it, they are not science in the sense of the scientific method.

What does the complete means to reproduce actually mean? It seems to us that unless a scientific paper allows the reader to reproduce every table, diagram and statistical result using code for which the source and means to run it are provided, then the results are unquantifiably in error. Nobody of course expects every line to be pored over for every scientific paper, but for really critical results, we might have to do exactly that. The whole stack needs to be open – the knock-on effects of incorrect results can be hugely expensive.

This is no mere quibble. Even with an open stack, it is inconceivable for one reader to verify a scientific result. Instead the reader must confine themselves to inspecting some part of the analysis or statistics code leaving everything deeper to others. Figure 1 gives some idea of this but is not to scale. The bit that most scientists have to write is typically a few thousands up to a million or so lines of code and is shown in yellow. The next bit down is the applications we use such as the language interpreters or compilers, the database software, the graphics or the statistical analysis software. This is coloured in green and will typically be around 1-5 million lines of code. The purple section at the bottom is tens of millions of lines of code corresponding to the operating system and all the C libraries for example. although larger, software lower down the stack tends to be more reliable because its usage is proportionately much higher. The most we can reasonably ask for is that enough people read the yellow parts. This is the science.

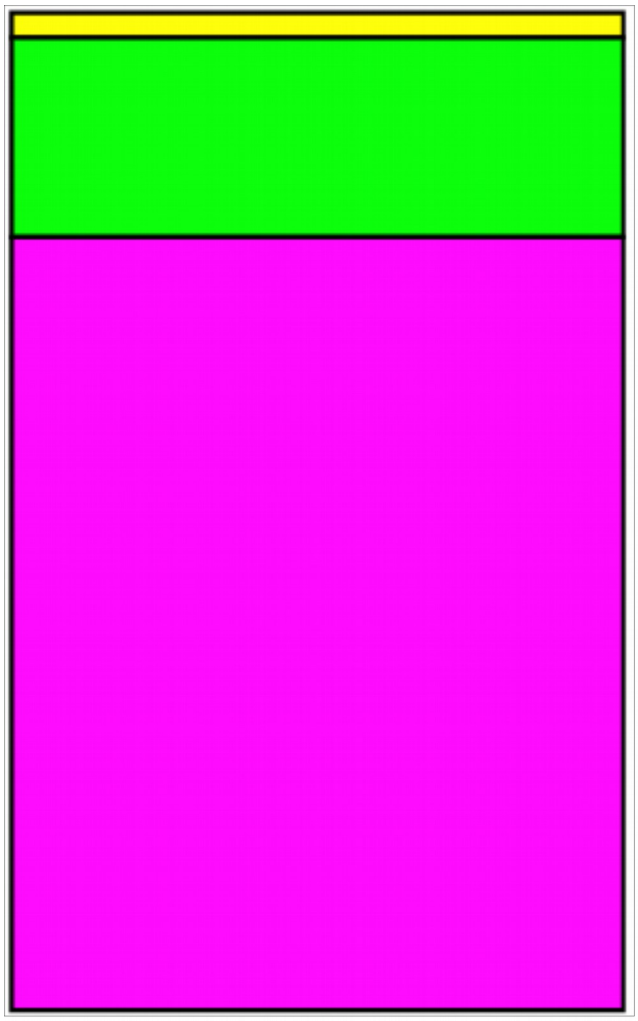

Figure 1. An illustration of a typical software stack in computational science.

We conclude by saying that Science has a serious and growing problem. Science is beginning to appreciate the size of the problem (Perkel 2018), but it is shared by all consumers of software as the numbers of failures in established systems will testify, (the reader might like to type “software failures” into their browser of choice - be prepared for a shock.) The humble and amusing bug of 30 years ago has grown into a system and career-wrecking monster. It is within our powers to reign back on this inexorable growth, but not until we remember what falsifiable **really** means.

A call for columns

At the start of the 10th year of the Impact series, we would like to invite you to discuss the impact of your own software. We have published over 40 Impact columns so far and you will be in a good company with authors from large and small institutions who went before you. There are only three requirements: an interesting read for the IEEE Software audience and some metrics about size, volume of shipment if relevant and how you address the maintenance. We have published columns from all continents except Antarctica but would welcome more contributions from software powerhouses in the East. Also, financial applications have eluded us so far, so don’t be shy. Your software does not have to be millions of lines of code or be associated with a high-profile phenomenon such as the Mars Rover, the Higgs Boson discovery or Dieselgate. We have a short

and painless review process (no endless cycles; we are too busy for that), and while we can't promise fame,we can promise that your experiences will help enrich the practical knowledge of how software systems perform in the myriad ways we ask of them.

References

Claerbout J.F. (1994) "Seventeen years of super computing and other problems in seismology", National Research Council meeting on High Performance Computing in Seismology, Oct 2 1994. http://sepwww.stanford.edu/sep/jon/nrc.html, accessed 25-Sep-2018.Freedman L.P., Cockburn I.M. and Simcoe T.S. (2015) "The Economics of Reproducibility in Preclinical Research", PLOS, https://doi.org/10.1371/journal.pbio.1002165

van Genuchten M., Hatton L. (2011) "Software mileage", IEEE Software, 28(5), September/October.

Hatton L., Spinellis D. and van Genuchten M. (2017) "The long-term growth rate of evolving software: Empirical results and implications" Journal of Software: Evolution and Process, 29(5), https://doi.org/10.1002/smr.1847.

Huston L. (2014) https://www.forbes.com/sites/larryhusten/2014/09/26/new-england-journal-of-medicine-declines-to-retract-papers-from-disgraced-research-group/#7f8405a17228, accessed 27-Sep-2018.

Ioannidis JPA (2014) How to Make More Published Research True. PLoS Med 11(10): e1001747. https://doi.org/10.1371/journal.pmed.1001747

Kuhn T.S. (1962) "The Structure of Scientific Revolutions", University of Chicago Press, ISBN 978-0-22-645811-3.

Mossinger J.M. (2010) "Software in Automotive Systems", IEEE Software, 27(2), March/April 2010.

Myers G.J. (1979) "The Art of Software Testing", John Wiley, ISBN 0-471-4328-1.

János Nagy, Kálmán Balajthy, Sándor Szalai, Bálint Sódor, István Horváth, and Csaba Lipusz (2016) "Obstanovka: Exploring Nearby Space", IEEE Software, 33, (4), July/August

Simon Portegies Zwart and Jeroen Bédorf (2016), "Creating the Virtual Universe", IEEE Software, 33, (5), September/October

Perkel J., (2018) "A toolkit for data transparency takes shape", https://www.nature.com/articles/d41586-018-05990-5

Rousseau, D. (2012), "The Software behind the Higgs Boson Discovery", IEEE Software, 29, (5), September/October

Popper K. (1959) "The Logic of Scientific Discovery", Routledge, ISBN 1-1344-7002-9.

Rogier Wester and John Koster (2015), "The Software behind Moore's Law", IEEE Software, 32, (2), March/April.