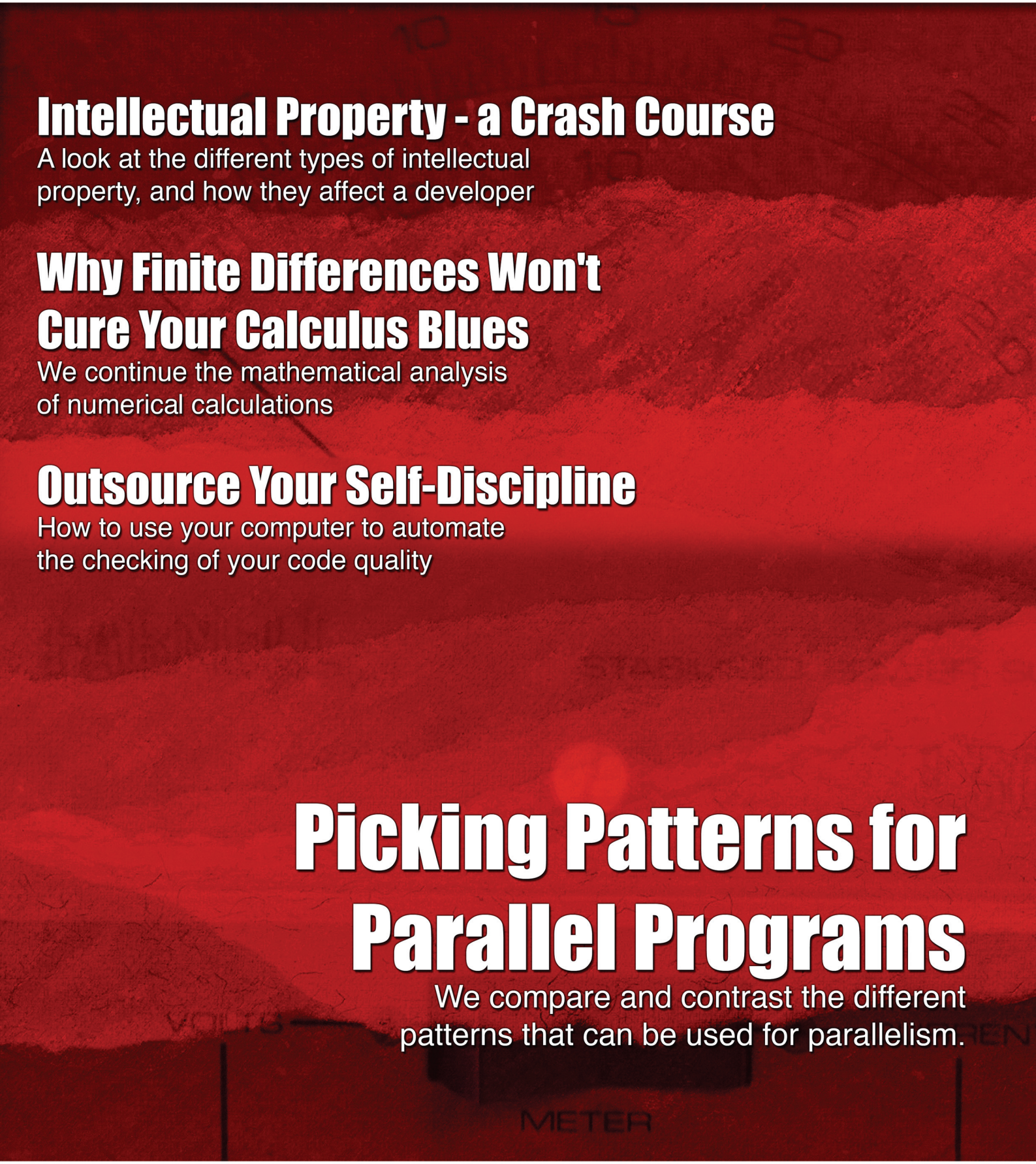
overload 105
OCTOBER 2011 £3



Intellectual Property - a Crash Course
A look at the different types of intellectual property, and how they affect a developer

Why Finite Differences Won't Cure Your Calculus Blues
We continue the mathematical analysis of numerical calculations

Outsource Your Self-Discipline
How to use your computer to automate the checking of your code quality

Picking Patterns for Parallel Programs
We compare and contrast the different patterns that can be used for parallelism.

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 105

October 2011

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 106 should be submitted by 1st November 2011 and for Overload 107 by 1st January 2012.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of ACCU For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

A Journey Through History

Despite early pioneers, the computer revolution is relatively young. Ric Parkin takes a personal tour.

A few news stories recently reminded me of just how far the world of computing has progressed, often in a surprisingly short space of time, so I thought I'd look at a potted, and at times personal, history of computing.

In a literal sense, the first computers were just mechanical devices that helped humans make calculations, whether the humble (yet powerful in the right hands) abacus, the impressive Antikythera mechanism [Antikythera], via clocks and orrories, up to powerful mechanical devices such as automated programmable looms [Jacquard]. While not radically different from these forerunners, the Babbage Difference Engine is still historically interesting. It was basically a big automated calculator that could calculate polynomial approximations using finite differences (those reading Richard Harris' articles will note their long history). Their advantages were they were faster than a human and avoided the inevitable errors caused bythe tedious task of calculating tables of logs and trigonometry for various uses such as in navigation, ballistics, and engineering.

But this just set the scene for his Analytical Engine. This was much more powerful and flexible, using various types of punch cards – as seen in the programmable looms – not only as input and output, but also as a way of changing the behaviour of the engine. In terms of its architecture it was revolutionary and way ahead of its time – with separate arithmetical unit, a central processing unit that supported conditional branching and looping, and seperate storage, input and output units. It is recognisably a modern computer. However, compared to current (or even old!) hardware it is a lumbering beast – estimates would be of a few KB of memory at most, with the processor only executing a handful of instructions per second, which may have been able to multiply two numbers in a couple of minutes. All in a package the size of a large room! Sadly only small prototypes were built, although there is a project that is trying to reconstruct a working model [Analytical]. In many ways its design was way ahead of anything built for another century, which has led to plenty of 'Alternative History' fiction wondering what the world would have been like it it had succeeded [Gibson Sterling]. It did bring about another first though – while translating a paper describing the machine in 1842, Ada Lovelace added many notes and thoughts, including the instructions needed to calculate Bernoulli numbers. Subsequent analysis has shown this would have worked correctly had the engine ever been built, and so is credited as the world's first computer program. [Lovelace]

Things went quiet after that, until the pressure leading up to the second world war led to many of these ideas being reinvented (sadly Babbage's work itself was largely forgotten and was never really influential at that time). Some are relatively obscure nowadays – I found that a German, Konrad Zuse, had invented a series of increasingly powerful computers in the late 1930s, culminating in the Z3 [Zuse], one of the first Turing-Complete. However the authorities thought it was 'strategically unimportant' and didn't fund his work. Post-war, he continued to develop his ideas and invented the first high-level language 'Plankalkül', although it wasn't actually implemented until 2000.

The more famous pioneers at the time were of course the code breakers at Bletchley Park, who not only came up with ingenious ways of automating the tedious work of decrypting masses of intercepted messages, but improved on Polish Bombes that checked for possible decryption keys, and Colossus which was semi-programmable and used to crack the hardest codes. Kept secret for many decades, it's only relatively recently that the importance of this period has been recognised, and efforts made to restore and rebuild both the park and its treasures. Some news in this area is that Astrid Byro has completed her trek to Everest Base Camp, raising money for the continuing work at Bletchley, and generating many stunning photographs [Byro]. And sadly, Tony Sale, who led the mamoth task of rebuilding Colossus and will be recalled by those who have attaended the ACCU autumn security conferences, died recently aged 80 [Sale]. Apart from Colossus, he achieved many amazing engineering feats including building an early android, and is a true inspiration.

Post war saw a flurry of new computers which are more well-known, such as the USA's Eniac [Eniac], and Manchester's SSEM [Baby]. These were now fully programmable, with even more of the many features that we now take for granted. The revolution had started, with many key technologies helping boost the power of computers, such as the invention of the transitor at Bell Labs in 1948 and the integrated circuit at Texas instruments in 1958. Programming languages were also being improved, with still extant ones such as FORTRAN in 1954 and LISP in 1958.

The subsequent history I can couch in a more personal way. As the 60s ended, my father was a computer engineer, maintaining computers for Sperry, the UNIVAC being one I remember. Work was sometimes brought home in the form of used punch cards and folded teletype paper, which held little interest to myself beyond their use as drawing paper. By the end of the 70s he'd moved on to being a trainer of other engineers for DEC in Manchester. For various reasons we'd pick him up on the way back from school, and as it was usually too early to leave I'd be allowed to keep myself occupied on some of the computers there, mainly PDPs and VAXes accessed via various devices. Some of these were quite memorable, such as playing Lunarlander [Lunarlander] on a vector graphics screen using a light pen, Adventure [Adventure] on a teletype (which is where the phrase 'You are in a maze of twisty little passages, all alike' comes from), or Star Trek on a VDU. Of course after a while you get bored, and so I was given a small blue book which told me about something called BASIC.

This was the time of the home computer revolution, and the chemistry teacher at school set up a computer club with one of the early ZX80s, followed by ZX81s and an Apple II. After saving up, I finally (after a notoriously long wait) got a 48K ZX Spectrum. I learnt an awful lot from this humble machine, not only programming in BASIC, but trying assembler, C and Pascal. I even used it in my Control Technology 'O'

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

Level project, where it drove a simple 'Multiple Choice Quiz Marker'. (Of course, disaster struck on the morning of it being marked – one of the light detectors used to find marks on the paper broke. Fortunately I realised that the sample paper I was using just happened to have some redundancy in it, and I could use the other three to infer the result of the broken one – my first experience of error correction!)

The home computer craze had run out of steam, and I'd sold my computer before going to 6th Form and University. I still occasionally came into contact with computers though, using Pascal and LaTEX on VAXes during summer jobs at the Royal Radar And Signals Establishment, and some university projects on numerical approximation when solving Schrodinger's Equation.

By the time of my first job in early 1991, the PC had come to dominate. So my first work machine was running an Intel 286, running at about 8MHz. I forget how much memory it had, but it was pitiful by today's standards, but it, along with a few 386 machines and a rather expensive 486, were used to do some rather remarkable jobs. We were a small company that did map digitising for the likes of the Ordinance Survey, but also lots of location based applications such as hydrolographic surveying. This was quite fun as we'd go out to interesting locations, such as when we developed the software for a major survey ship [Protea]. At the time the use of land based transponders and triangulation was the main technique to determine location, but we did have one of the new GPS receivers, which was the size of a small suitcase. My last job there was to write a new version of our graphical survey editing suite for an upstart operating system called Windows 3.0.

At my next job, I came across a language called C, and even invented some simple ones myself. We quickly developed an application for a book database, with various search and results screens. Knowing that the database format and screen layout was bound to need to be tweaked close to release, I came up with a simple script format that would describe the database, define ways of searching it, give the screen layouts and how the navigation between them fitted together. Without realising it I had basically (poorly) reinvented SQL and HTML!

After that I moved to a job in the computing hotspot known as 'Silicon Fen', where we were porting from C to the new trendy language C++. We also had a new tool – something called email that worked across a thing called 'the Internet'. This had also got something new called 'The World Wide Web', accessed by a browser such as Lynx or Mosaic. We weren't sure what it would be useful for though, although a very popular early use was to see if there was any coffee, even though it was someone else's [Trojan].

By the end of the 90s I'd got a bit cocky, thinking I was a bit of a C++ wizz, so discovering Usenet and in particular comp.lang.c++ moderated came as a bit of a shock: in reality I knew so little, and I hadn't even known it! Apparently this is quite common – if you graph perceived ability against actual ability you end up with a graph with an early peak followed by a valley and slow rise, where you overrate yourself to start with, but when you discover just how much you don't know it falls back and you tend to underrate yourself. I think this is similar to the Dunning-Kruger effect [D-K], but can be seen in a single individual learning over time. But via the newsgroups, I did find out about ACCU, which helped me realise my ignorance as well a provide a means of improving!

The 2000s will be more familiar to most people – the internet became ubiquitous, chip clock speeds stalled, but Moore's Law continued to hold with the extra transistors going towards more on-chip caches, multi-cores, and dedicated graphics and audio functions. Social media facilitated by computers and mobile communications have put people in touch like never before (I've literally just heard from an old girlfriend who has lived near Edinburgh for 17 years – in a previous era we'd have never met again). Politics and technology are still ill-at-ease – in the wake of the urban disturbances in the UK over the summer there were calls for the government to be able to shut down the likes of Blackberry and Twitter on the basis that they could be used to organise trouble. Thankfully people have realised that they in themselves aren't the problem, they were also used to respond positively, and obvious workarounds exist. The existing laws for incitement dealt with the issue quickly (and easily as such communications could be tracked, providing evidence).

So what of the future? Cloud computing has been The Next Big Thing for a while, but may well go mainstream if someone can make it as reliable, usable, and seamless as local computing. Multicore and parallel processing are here now and will grow in importance – learning how to design software that safely takes advantage of it is the big problem for the next few years.

And the big news for many will be the ratification of the new C++ Standard. Some compilers already implement parts, and the next couple of years will see better compliance – I hope the major vendors commit to full implementations and not just pick and choose parts. For commercial vendors, pressure from their customers (ie you) will help, and for the open source implementations, input from their developers (ie you) will as well. But be quick, the next C++ standard is already being considered!

On a personal note

I came up with the idea for this historical overview a while ago because of some stories in the news. But the personal aspect now seems most apt: while writing it my father, the person who got me started with computers all those years ago, died suddenly aged 71. I'd like to dedicate this to him.

References

[Adventure] http://en.wikipedia.org/wiki/Colossal_Cave_Adventure

[Analytical] http://www.bbc.co.uk/news/technology-15001514

[Antikythera] http://www.antikythera-mechanism.gr/

[Baby] http://en.wikipedia.org/wiki/Manchester_Small-Scale_Experimental_Machine

[Byro] http://abc-ebc.blogspot.com/

[D-K] http://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

[Eniac] http://en.wikipedia.org/wiki/ENIAC

[Gibson Sterling] http://en.wikipedia.org/wiki/The_Difference_Engine

[Jacquard] http://en.wikipedia.org/wiki/Jacquard_loom

[Lovelace] http://en.wikipedia.org/wiki/Ada_Lovelace

[LunarLander] http://en.wikipedia.org/wiki/Lunar_Lander_%28video_game%29

[Protea] http://www.navy.mil.za/vtour/protea/index.htm

[Sale] http://www.bbc.co.uk/news/technology-14720180

[Trojan] http://www.cl.cam.ac.uk/coffee/qsf/coffee.html

[Zuse] http://en.wikipedia.org/wiki/Z3_%28computer%29

Why Finite Differences Won't Cure Your Calculus Blues

Now we know our problem in depth. Richard Harris analyses if a common technique will work adequately.

In the previous article we discussed the foundations of the differential calculus. Initially defined in the 17th century in terms of the rather vaguely defined infinitesimals, it was not until the 19th century that Cauchy gave it a rigorous definition with his formalisation of the concept of a limit. Fortunately for us, the infinitesimals were given a solid footing in the 20th century with Conway's surreal numbers and Robinson's non-standard numbers, saving us from the annoyingly complex reasoning that Cauchy's approach requires.

Finally, we discussed the mathematical power tool of numerical computing; Taylor's theorem. This states that for any sufficiently differentiable function f

$$f(x+\delta) = f(x) + \delta \times f'(x) + \tfrac{1}{2}\delta^2 \times f''(x) + \ldots + \tfrac{1}{n!}\delta^n \times f^{(n)}(x) + R_n$$

$$\min\left(\tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(x+\theta\delta)\right) \leq R_n \leq \max\left(\tfrac{1}{(n+1)!}\delta^{n+1} \times f^{(n+1)}(x+\theta\delta)\right) \quad \text{for } 0 \leq \theta \leq 1$$

If we do not place a limit on the number of terms, we have

$$f(x+\delta) = f(x) + \delta \times f'(x) + \tfrac{1}{2}\delta^2 \times f''(x) + \ldots + \tfrac{1}{n!}\delta^n \times f^{(n)}(x) + \ldots = \sum_{i>=0} \tfrac{1}{i!}\delta^i f^{(i)}(x)$$

Note that here $f'(x)$ stands for the first derivative of f at x, $f''(x)$ for the second and $f^{(n)}(x)$ for the n^{th} with the convention that the 0^{th} derivative of a function is the function itself. The capital sigma stands for the sum of the expression to its right for every unique value of i that satisfies the inequality beneath it and $i!$ stands for the factorial of i, being the product of every integer between 1 and i, with convention that the factorial of 0 is 1.

You may recall that we used forward finite differencing as an example of cancellation error in the first article of this series [Harris10]. This technique replaces the infinitesimal δ in the definition of the derivative with a finite, but small, quantity.

We found that the optimal choice of this finite δ was driven by a trade off between approximation error and cancellation error. With some fairly vigorous hand waving, we concluded that it was the square root of ε, the difference between 1 and the smallest floating point number greater than 1.

This time, and I fancy I can hear your collective groans of dismay, we shall dispense with the hand waving.

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

Forward finite difference

Given some small finite positive δ, the forward finite difference is given by

$$\frac{f(x+\delta) - f(x)}{\delta}$$

Using Taylor's theorem the difference between this value and the derivative is equal to

$$\tfrac{1}{2}\delta \times f''(y)$$

for some y between x and $x+\delta$.

Assuming f introduces a relative rounding error of some non-negative integer n_f multiples of ½ε and that x has already accumulated a relative rounding error of some non-negative integer n_x multiples of ½ε then, if we wish to approximate the derivative as accurately as possible we should choose δ to minimise

$$\frac{n_f\varepsilon}{\delta}\left|f(x)\right| + \left(\tfrac{1}{2}n_f + 1\right)\varepsilon\left|f'(x)\right| + \tfrac{1}{2}\left(n_x\varepsilon\left|x\right| + \delta\right)\left|f''(y_1)\right|$$

as shown in derivation 1.

Now this is a function of δ of the form

$$f(x) = \frac{a}{x} + bx + c$$

Such functions, for positive a, b and x, have a minimum value of $2\sqrt{ab} + c$ at $\sqrt{a/b}$ (taking the positive square roots) as shown in derivation 2.

To leading order in ε and δ the worst case absolute error in the forward finite difference approximation to the derivative is therefore

$$\sqrt{2n_f\varepsilon\left|f(x)f''(y_1)\right|} + \left(\tfrac{1}{2}n_x\left|xf''(y_1)\right| + \left(\tfrac{1}{2}n_f + 1\right)\left|f'(x)\right|\right)\varepsilon$$

when δ is equal to

$$\sqrt{2n_f\varepsilon\left|\frac{f(x)}{f''(y_1)}\right|}$$

taking the positive square roots in both expressions.

Now these expressions provide a *very* accurate estimate of the optimal choice of δ and the potential error in the approximation of the derivative that results from that choice. There are, however, a few teensy-weensy little problems.

The first is that these expressions depend on the relative rounding errors of x and f. We can dismiss these problems out of hand since if we have no inkling as to how accurate x or f are then we clearly cannot possibly have any expectation whatsoever that we can accurately approximate the derivative.

The second, and slightly more worrying, is that the error depends upon the very value we are trying to approximate; the derivative of f at x.

Given the circumstances, the best thing we can really do is to guess how the second derivative behaves

Approximation error of the forward finite difference

From Taylor's theorem we have

$$f(x+\delta) = f(x) + \delta f'(x) + \tfrac{1}{2}\delta^2 f''(y)$$

for some y between x and $x+\delta$.

We shall assume that f introduces a proportional rounding error of some non-negative integer n_f multiples of $\tfrac{1}{2}\varepsilon$ and that x has a proportional rounding error of some non-negative n_x multiples of $\tfrac{1}{2}\varepsilon$.

We shall further assume that we can represent δ exactly and that the sum of it and x introduces no further rounding error.

Under these assumptions the floating point result of the forward finite difference calculation is bounded by

$$\frac{\left(f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)\times\left(1\pm\tfrac{1}{2}n_f\varepsilon\right)-f\left(x\pm\tfrac{1}{2}n_x\varepsilon x\right)\times\left(1\pm\tfrac{1}{2}n_f\varepsilon\right)\right)\times\left(1\pm\tfrac{1}{2}\varepsilon\right)}{\delta}\times\left(1\pm\tfrac{1}{2}\varepsilon\right)$$

where the error in x is the same in both cases.

This is in turn bounded by

$$\frac{f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)\times\left(1\pm\tfrac{1}{2}n_f\varepsilon\right)-f\left(x\pm\tfrac{1}{2}n_x\varepsilon x\right)\times\left(1\pm\tfrac{1}{2}n_f\varepsilon\right)}{\delta}\times\left(1\pm\tfrac{1}{2}\varepsilon\right)^2$$

$$\in\frac{f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)-f\left(x\pm\tfrac{1}{2}n_x\varepsilon x\right)\pm\tfrac{1}{2}n_f\varepsilon\times\left(\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)\right|+\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x\right)\right|\right)}{\delta}\times\left(1\pm\tfrac{1}{2}\varepsilon\right)^2$$

Noting that

$$\begin{aligned}
&f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)-f\left(x\pm\tfrac{1}{2}n_x\varepsilon x\right)\\
&\quad=f(x)+\left(\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)f'(x)+\tfrac{1}{2}\left(\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)^2 f''(y_1)-f(x)-\left(\pm\tfrac{1}{2}n_x\varepsilon x\right)f'(x)-\tfrac{1}{2}\left(\pm\tfrac{1}{2}n_x\varepsilon x\right)^2 f''(y_0)\\
&\quad=\delta f'(x)+\tfrac{1}{8}\left(n_x\varepsilon x\right)^2\left(f''(y_1)-f''(y_0)\right)+\tfrac{1}{2}\left(\pm n_x\varepsilon\delta x+\delta^2\right)f''(y_1)
\end{aligned}$$

$$\begin{aligned}
&\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)\right|+\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x\right)\right|\\
&\quad=\left|f(x)+\left(\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)f'(x)+\tfrac{1}{2}\left(\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)^2 f''(y_1)\right|+\left|f(x)+\left(\pm\tfrac{1}{2}n_x\varepsilon x\right)f'(x)+\tfrac{1}{2}\left(\pm\tfrac{1}{2}n_x\varepsilon x\right)^2 f''(y_0)\right|\\
&\quad\le 2\left|f(x)\right|+\left(n_x\varepsilon|x|+\delta\right)\left|f'(x)\right|+\tfrac{1}{8}\left(n_x\varepsilon x\right)^2\left(\left|f''(y_1)\right|+\left|f''(y_0)\right|\right)+\tfrac{1}{2}\left(n_x\varepsilon\delta|x|+\delta^2\right)\left|f''(y_1)\right|
\end{aligned}$$

the result is hence bounded by

$$\left[\begin{array}{l}
f'(x)+\tfrac{1}{8}\dfrac{\left(n_x\varepsilon x\right)^2}{\delta}\left(f''(y_1)-f''(y_0)\right)+\tfrac{1}{2}\left(\pm n_x\varepsilon x+\delta\right)f''(y_1)\\
\pm\tfrac{1}{2}n_f\varepsilon\times\left(2\dfrac{\left|f(x)\right|}{\delta}+\left(\dfrac{n_x\varepsilon|x|}{\delta}+1\right)\left|f'(x)\right|+\tfrac{1}{8}\dfrac{\left(n_x\varepsilon x\right)^2}{\delta}\left(\left|f''(y_1)\right|+\left|f''(y_0)\right|\right)+\tfrac{1}{2}\left(n_x\varepsilon|x|+\delta\right)\left|f''(y_1)\right|\right)
\end{array}\right]\times\left(1\pm\tfrac{1}{2}\varepsilon\right)^2$$

giving a worst-case absolute error of

$$\begin{aligned}
&\tfrac{1}{2}\left(n_x\varepsilon|x|+\delta\right)\left|f''(y_1)\right|+\tfrac{1}{2}n_f\varepsilon\times\left(2\frac{\left|f(x)\right|}{\delta}+\left|f'(x)\right|\right)+\varepsilon\left|f'(x)\right|+O\left(\frac{\varepsilon^2}{\delta}\right)+O(\varepsilon\delta)\\
&\quad=\frac{n_f\varepsilon}{\delta}\left|f(x)\right|+\left(\tfrac{1}{2}n_f+1\right)\varepsilon\left|f'(x)\right|+\tfrac{1}{2}\left(n_x\varepsilon|x|+\delta\right)\left|f''(y_1)\right|+O\left(\frac{\varepsilon^2}{\delta}\right)+O(\varepsilon\delta)
\end{aligned}$$

If our guess is significantly smaller, however, we're in trouble since we shall in this case underestimate the error

Fortunately, we can recover the error to leading order in ε by replacing it with the finite difference itself.

The third, and by far the most significant, is that both expressions depend upon the behaviour of the unknown second derivative of f.

Unfortunately this is ever so slightly more difficult to weasel our way out of. By which I of course mean that in general it is entirely impossible to do so.

Given the circumstances, the best thing we can really do is to guess how the second derivative behaves. For purely pragmatic reasons we might assume that

$$f''(y_1) = \frac{f(x)}{(|x|+1)^2}$$

since this yields

$$\delta = \sqrt{2n_f\varepsilon} \times (|x|+1)$$

The advantage of having a δ of this form is that it helps alleviate both rounding and cancellation errors; it is everywhere larger than $(2n_f\varepsilon)^{1/2}|x|$ and hence mitigates against rounding error for large $|x|$ and it is nowhere smaller than $(2n_f\varepsilon)^{1/2}$ and hence mitigates against cancellation error for small $|x|$.

The minimum of *a*/*x*+*bx*+*c*

Recall that the turning points of a function f, that is to say the minima, maxima and inflections, occur where the derivative is zero.

$$f(x) = \frac{a}{x} + bx + c$$

$$f'(x) = -\frac{a}{x^2} + b$$

$$f'\left(\sqrt{a/b}\right) = -\frac{a}{a/b} + b = -b + b = 0$$

$$f\left(\sqrt{a/b}\right) = \frac{a}{\sqrt{a/b}} + b\sqrt{a/b} + c = 2\sqrt{ab} + c$$

Note that this is only a real number if a and b have the same sign.

We can use the second derivative to find out what kind of turning point this is; positive implies a minimum, negative a maximum and zero an inflection.

$$f''(x) = 2\frac{a}{x^3}$$

$$f''\left(\sqrt{a/b}\right) = 2\frac{a}{(a/b)\times\sqrt{a/b}} = 2\frac{b}{\sqrt{a/b}}$$

If both a and b are positive and we choose the positive square root of their ratio then this value is positive and we have a minimum.

Derivation 2

Substituting this into our error expression yields

$$\frac{\sqrt{2n_f\varepsilon}}{|x|+1}|f(x)| + \tfrac{1}{2}n_x\varepsilon\frac{|xf(x)|}{(|x|+1)^2} + \left(\tfrac{1}{2}n_f+1\right)\varepsilon|f'(x)|$$

Of course this estimate will be inaccurate if the second derivative differs significantly from our guess.

This is little more than an irritation if our guess is significantly larger than the second derivative since the true error will be smaller than our estimate and we will still have a valid, if pessimistic, bound.

If our guess is significantly smaller, however, we're in trouble since we shall in this case underestimate the error. Unfortunately there is little we can do about it.

One thing worth noting is that if the second derivative is very large at x then the derivative will be rapidly changing in its vicinity. The limiting case as the second derivative grows in magnitude is a discontinuity at which the derivative doesn't exist.

If we have a very large second derivative, we can argue that the derivative is in some sense approaching non-existence and that we should need to be aware of this and plan our calculations accordingly.

We have one final issue to address before implementing our algorithm; we have assumed that we can exactly represent both δ and $x+\delta$. Given that our expression for the optimal choice of δ involves a floating point square root operation this is, in general, unlikely to be the case.

Fortunately we can easily find the closest floating point value to δ for which our assumptions hold with

```
template<class F>
class forward_difference
{
public:
  typedef F function_type;
  typedef typename F::argument_type argument_type;
  typedef typename F::result_type result_type;

  explicit forward_difference(
     const function_type &f);
  forward_difference(const function_type &f,
                     unsigned long nf);

  result_type operator()(
     const argument_type &x) const;

private:
  function_type f_;
  result_type   ef_;
};
```

Listing 1

To compare our approximation of the error with the exact error we shall count the number of leading zeros after the decimal point of each

```
template<class F>
forward_difference<F>::forward_difference(
  const function_type &f) : f_(f),
  ef_(sqrt(result_type(2UL)*eps<result_type>()))
{
  if(ef_<result_type(eps<argument_type>()))
  {
    ef_ = result_type(eps<argument_type>());
  }
}

template<class F>
forward_difference<F>::forward_difference(
  const function_type &f,
  const unsigned long nf) : f_(f),
  ef_(
     sqrt(result_type(2UL*nf)*eps<result_type>()))
{
  if(ef_<result_type(eps<argument_type>()))
  {
    ef_ = result_type(eps<argument_type>());
  }
}
```

Listing 2

$$\left(x+\sqrt{2n_f\varepsilon}\times\left(|x|+1\right)\right)-x$$

Naturally this will have some effect on the error bounds, but since it will only be $O(\varepsilon\delta)$ it will not impact our approximation of them.

Listing 1 provides the definition of a forward finite difference function object.

Note that we have two constructors; one with an argument to represent n_f and one without. The latter assumes a rounding error of a single $\frac{1}{2}\varepsilon$, as shown in listing 2, and is intended for built in functions for which such an assumption is reasonable.

```
template<class F>
typename forward_difference<F>::result_type
forward_difference<F>::operator()(
   const argument_type &x) const
{
  const argument_type abs_x =
     (x>argument_type(0UL)) ? x : -x;
  const argument_type d =
     ef_*(abs_x+argument_type(1UL));
  const argument_type u = x+d;

  return (f_(u)-f_(x))/result_type(u-x);
}
```

Listing 3

Note also that we are assuming that the result type of the function object has a **numeric_limits** specialisation (whose **epsilon** function is represented here by the typesetter-friendly abbreviation ***eps*<T>**), can be conversion constructed from an unsigned long and has a global namespace overload for the **sqrt** function. To all intents and purposes we are assuming it is an inbuilt floating point type.

We should rather hope that the argument type of the function object is the same as its result type, and for that matter that this is a floating point type, but must provide for a minimum δ just in case the user decides otherwise, which we do by setting a lower bound for **ef_**. We must be content in such cases with the fact that they have made a rod for their own back when it comes time to perform their error analysis!

Listing 3 gives the implementation of the function call operator based upon the results of our analysis.

As an example, let us apply our forward difference to the exponential function with arguments from -10 to 10. We can therefore expect that n_f is equal to one and n_x to zero and hence that our approximation of the error is

$$\frac{\sqrt{2\varepsilon}}{|x|+1}\left|f\left(x\right)\right|+1\tfrac{1}{2}\varepsilon\left|f'\left(x\right)\right|$$

Since the derivative of the exponential function is the exponential function itself, we can accurately calculate the true error by taking the absolute difference between it and the finite difference approximation.

To compare our approximation of the error with the exact error we shall count the number of leading zeros after the decimal point of each, which we can do by negating the base 10 logarithm of the values.

Figure 1 plots the leading zeros in the decimal fraction of our approximate error as a dashed line and in that of the true error as a solid line, with larger values on the y axis thus implying smaller values for the errors.

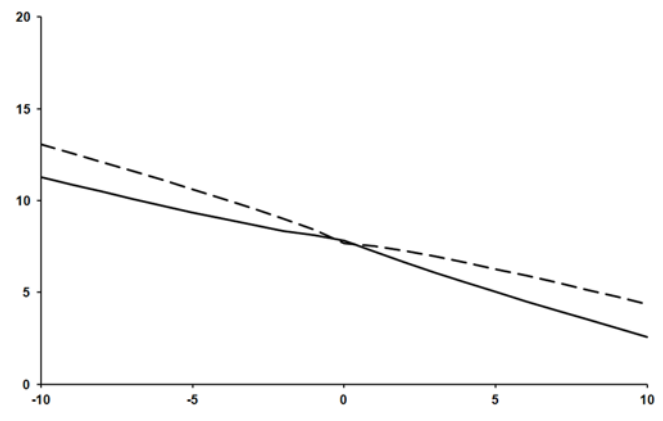


Figure 1

Our approximation clearly increasingly underestimates the error as the absolute value of x increases. This shouldn't come as too much of a surprise since our assumption about the behaviour of the second derivative grows increasingly inaccurate as the magnitude of x increases for the exponential function.

Nevertheless, our approximation displays the correct overall trend and is nowhere catastrophically inaccurate, at least to my eye.

The question remains as to whether we can do any better.

Symmetric finite difference

Returning to Taylor's theorem we can see that the term whose coefficient is a multiple of the second derivative is that of δ^2. This has the very useful property that it takes the same value for both $+\delta$ and $-\delta$.

If we approximate the derivative with the finite difference between a small step above and a small step below x we can arrange for this term to cancel out. Specifically, the expression

$$\frac{f(x+\delta)-f(x-\delta)}{2\delta}$$

differs from the derivative at x by

$$\tfrac{1}{6}\delta^2 \times f'''(y)$$

for some y between $x-\delta$ and $x+\delta$ as shown in derivation 3.

Now this is a rather impressive order of magnitude better in δ than the forward finite difference considering that it involves no additional evaluations of f.

That said, it is not at all uncommon that both the value of the function and its derivative are required, in which case the finite forward difference can get one of its function evaluations for free.

With a similar analysis to that we made for the forward finite difference, given in derivation 4, we find that the optimal choice of δ must minimise

$$\frac{n_f\varepsilon}{2\delta}|f(x)|+\left(\tfrac{1}{2}n_f+1\right)\varepsilon|f'(x)|+\tfrac{1}{2}n_x\varepsilon|xf''(x)|+\tfrac{1}{6}\delta^2|f'''(y)|$$

This time the quantity we wish to minimise is a function in δ of the form

$$f(x)=\frac{a}{x}+bx^2+c$$

which, as shown in derivation 5, given positive b is minimised by

$$f\left(\left(\frac{a}{2b}\right)^{\frac{1}{3}}\right)=\left(\frac{27}{4}\right)^{\frac{1}{3}}a^{\frac{2}{3}}b^{\frac{1}{3}}+c$$

To leading order in ε and δ the minimum relative error in the symmetric finite difference approximation to the derivative is therefore

$$\left(\frac{9}{32}\right)^{\frac{1}{3}}\left(n_f\varepsilon|f(x)|\right)^{\frac{2}{3}}\left(|f'''(y)|\right)^{\frac{1}{3}}+\left(\tfrac{1}{2}n_f+1\right)\varepsilon|f'(x)|+\tfrac{1}{2}n_x\varepsilon|xf''(x)|$$

The symmetric finite difference

From Taylor's theorem we have

$$f(x-\delta)=f(x)-\delta f'(x)+\tfrac{1}{2}\delta^2 f''(x)-\tfrac{1}{6}\delta^3 f'''(y_0)$$
$$f(x+\delta)=f(x)+\delta f'(x)+\tfrac{1}{2}\delta^2 f''(x)+\tfrac{1}{6}\delta^3 f'''(y_1)$$

for some y_0 between $x-\delta$ and x and y_1 between x and $x+\delta$

The symmetric finite difference is therefore

$$\frac{f(x+\delta)-f(x-\delta)}{2\delta}=\frac{2\delta f'(x)+\tfrac{1}{6}\delta^3 f'''(y_0)+\tfrac{1}{6}\delta^3 f'''(y_1)}{2\delta}$$
$$=f'(x)+\tfrac{1}{12}\delta^2\left(f'''(y_0)+f'''(y_1)\right)$$

The intermediate value theorem states that for a continuous function, there must be a point x between points x_0 and x_1 such that

$$f(x)=\frac{f(x_0)+f(x_1)}{2}$$

If the second derivative of our function is continuous this means that

$$\frac{f(x+\delta)-f(x-\delta)}{2\delta}=f'(x)+\tfrac{1}{6}\delta^2 f'''(y)$$

for some y between $x-\delta$ and $x+\delta$.

Derivation 3

Approximation error of the symmetric finite difference

From Taylor's theorem we have

$$f(x+\delta)=f(x)+\delta f'(x)+\tfrac{1}{2}\delta^2 f''(x)+\tfrac{1}{6}\delta^3 f'''(y)$$

for some y between x and $x+\delta$.

Making the same assumptions as before about rounding errors in both f and x, the floating point result of the symmetric finite difference calculation is bounded by

$$\frac{\begin{pmatrix}f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)\times\left(1\pm\tfrac{1}{2}n_f\varepsilon\right)\\-f\left(x\pm\tfrac{1}{2}n_x\varepsilon x-\delta\right)\times\left(1\pm\tfrac{1}{2}n_f\varepsilon\right)\end{pmatrix}\times\left(1\pm\tfrac{1}{2}\varepsilon\right)}{2\delta}\times\left(1\pm\tfrac{1}{2}\varepsilon\right)$$

which is in turn bounded by

$$\frac{\begin{bmatrix}f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)-f\left(x\pm\tfrac{1}{2}n_x\varepsilon x-\delta\right)\\\pm\tfrac{1}{2}n_f\varepsilon\times\left(\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)\right|+\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x-\delta\right)\right|\right)\end{bmatrix}}{2\delta}\times\left(1\pm\tfrac{1}{2}\varepsilon\right)^2$$

The error in x is again the same in all cases giving us

$$f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)-f\left(x\pm\tfrac{1}{2}n_x\varepsilon x-\delta\right)$$
$$=2\delta f'(x)\pm n_x\varepsilon\delta x f''(x)$$
$$\pm\left(\tfrac{1}{48}\left(n_x\varepsilon x\right)^3+\tfrac{1}{12}n_x\varepsilon\delta^2 x\right)\left(f'''(y_1)-f'''(y_0)\right)$$
$$+\left(\tfrac{1}{24}\left(n_x\varepsilon x\right)^2\delta+\tfrac{1}{6}\delta^3\right)\left(f'''(y_1)+f'''(y_0)\right)$$

$$\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x+\delta\right)\right|+\left|f\left(x\pm\tfrac{1}{2}n_x\varepsilon x-\delta\right)\right|$$
$$\le 2|f(x)|+\left(n_x\varepsilon|x|+2\delta\right)|f'(x)|$$
$$+\left(\tfrac{1}{4}\left(n_x\varepsilon|x|\right)^2+n_x\varepsilon\delta|x|+\delta^2\right)|f''(x)|$$
$$+\left(\tfrac{1}{48}\left(n_x\varepsilon|x|\right)^3+\tfrac{1}{24}\left(n_x\varepsilon|x|\right)^2\delta+\tfrac{1}{12}n_x\varepsilon\delta^2|x|+\tfrac{1}{6}\delta^3\right)$$
$$\times\left(|f'''(y_1)|+|f'''(y_0)|\right)$$

and hence a worst case absolute error of

$$\tfrac{1}{2}n_x\varepsilon|xf''(x)|+\tfrac{1}{12}\delta^2|f'''(y_1)+f'''(y_0)|+\frac{n_f\varepsilon}{2\delta}|f(x)|$$
$$+\left(\tfrac{1}{2}n_f+1\right)\varepsilon|f'(x)|+O\left(\frac{\varepsilon^2}{\delta}\right)+O(\varepsilon\delta)$$

or, by the mean value theorem again

$$\tfrac{1}{2}n_x\varepsilon|xf''(x)|+\tfrac{1}{6}\delta^2|f'''(y)|+\frac{n_f\varepsilon}{2\delta}|f(x)|$$
$$+\left(\tfrac{1}{2}n_f+1\right)\varepsilon|f'(x)|+O\left(\frac{\varepsilon^2}{\delta}\right)+O(\varepsilon\delta)$$

Derivation 4

The minimum of *a*/*x*+*bx*2+*c*

We find a turning point of *f* with

$$f(x) = \frac{a}{x} + bx^2 + c$$

$$f'(x) = -\frac{a}{x^2} + 2bx$$

$$f'\left(\left(\frac{a}{2b}\right)^{\frac{1}{3}}\right) = -a\left(\frac{2b}{a}\right)^{\frac{2}{3}} + 2b\left(\frac{a}{2b}\right)^{\frac{1}{3}} = -(2b)^{\frac{2}{3}}a^{\frac{1}{3}} + (2b)^{\frac{2}{3}}a^{\frac{1}{3}} = 0$$

$$f\left(\left(\frac{a}{2b}\right)^{\frac{1}{3}}\right) = a\left(\frac{2b}{a}\right)^{\frac{1}{3}} + b\left(\frac{a}{2b}\right)^{\frac{2}{3}} + c = \left(\frac{27}{4}\right)^{\frac{1}{3}}a^{\frac{2}{3}}b^{\frac{1}{3}} + c$$

We have a second derivative of

$$f''(x) = 2\frac{a}{x^3} + 2b$$

$$f''\left(\left(\frac{a}{2b}\right)^{\frac{1}{3}}\right) = 2\frac{a}{a/2b} + 2b = 6b$$

so if *b* is positive we have a minimum.

Derivation 5

when δ is equal to

$$\left(\tfrac{3}{2}n_f\varepsilon\left|\frac{f(x)}{f'''(y)}\right|\right)^{\frac{1}{3}}$$

Now this error is $O(\varepsilon^{2/3})$ and we should therefore expect this to be significantly better than the forward finite difference with its $O(\varepsilon^{1/2})$ error.

Unfortunately in order to achieve this we have compounded the problem of unknown quantities in the error and the choice of δ.

The optimal choice of the δ is now dependent on the properties of the third rather than the second derivative of *f* so we cannot use our previous argument that it may in some sense be reasonable to ignore it.

Furthermore, the resulting approximation error is dependant on *both* the second and the third derivatives of *f*.

We can deal with the first problem in the same way as we did before. In the name of pragmatism we assume that

$$f'''(y) = \frac{f(x)}{(|x|+1)^3}$$

giving a δ of

$$\left(\tfrac{3}{2}n_f\varepsilon\right)^{\frac{1}{3}} \times (|x|+1)$$

It's a little more difficult to justify a guess about the form of the second derivative since it plays no part in the choice of δ.

We could arbitrarily decide that it has a similar form to that we chose for it during our analysis of the forward finite difference. Specifically

$$f''(x) = \frac{f(x)}{(|x|+1)^2}$$

This strikes me a vaguely unsatisfying however, since it is not consistent with our assumed behaviour of the third derivative.

Instead, I should prefer something that satisfies

$$\frac{d}{dx}f''(x) = f'''(x) = \frac{f(x)}{(|x|+1)^3}$$

since this is approximately consistent with our guess.

A consistent second derivative

Consider first

$$f''(x) = a\frac{f(x)}{(|x|+1)^2}$$

$$\frac{d}{dx}f''(x) = a\frac{f'(x)}{(|x|+1)^2} - 2\operatorname{sgn}(x)a\frac{f(x)}{(|x|+1)^3}$$

where *a* is a constant and sgn(*x*) is the sign of *x*. For simplicity's sake, we shall declare the derivative of the absolute value of *x* at 0 to be 0 rather than undefined.

The second term has the required form so if we can find a way to cancel out the first we shall have succeeded. Adding a second term whose derivative includes a term of the same form might just do the trick.

$$f''(x) = a\frac{f(x)}{(|x|+1)^2} + b\frac{f'(x)}{|x|+1}$$

$$\begin{aligned}\frac{d}{dx}f''(x) &= a\frac{f'(x)}{(|x|+1)^2} - 2\operatorname{sgn}(x)a\frac{f(x)}{(|x|+1)^3} \\ &\quad + b\frac{f''(x)}{|x|+1} - \operatorname{sgn}(x)b\frac{f'(x)}{(|x|+1)^2} \\ &= (a - \operatorname{sgn}(x)b)\frac{f'(x)}{(|x|+1)^2} - 2\operatorname{sgn}(x)a\frac{f(x)}{(|x|+1)^3} \\ &\quad + \frac{b}{|x|+1}\left(a\frac{f(x)}{(|x|+1)^2} + b\frac{f'(x)}{(|x|+1)}\right) \\ &= (a - \operatorname{sgn}(x)b + b^2)\frac{f'(x)}{(|x|+1)^2} \\ &\quad + (ab - 2\operatorname{sgn}(x)a)\frac{f(x)}{(|x|+1)^3}\end{aligned}$$

We therefore require

$$a - \operatorname{sgn}(x)b + b^2 = 0$$

$$ab - 2\operatorname{sgn}(x)a = 1$$

Solving for *a* and *b* yields a serendipitously unique result.

Derivation 6

Derivation 6 shows that we should choose

$$f''(x) = -0.43016\frac{f(x)}{(|x|+1)^2} - 0.32472\frac{f'(x)}{|x|+1}$$

for positive *x*,

$$f''(x) = -f(x) - f'(x)$$

for *x* equal to zero and

$$f''(x) = -3.07960\frac{f(x)}{(|x|+1)^2} - 2.32472\frac{f'(x)}{|x|+1}$$

for negative *x*, with terms given to 5 decimal places.

Substituting these guessed derivatives back into our error formula yields an estimated error of

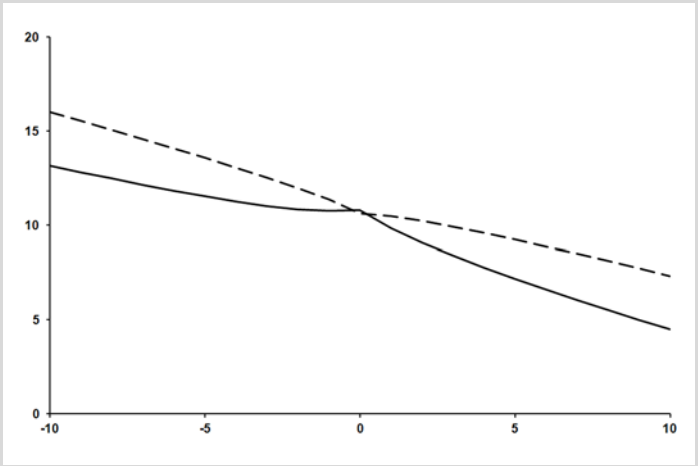


Figure 2

$$\left[\left(\frac{9}{32}\right)^{\frac{1}{3}}\frac{\left(n_f\varepsilon\right)^{\frac{2}{3}}}{|x|+1}+0.21508\frac{n_x\varepsilon|x|}{\left(|x|+1\right)^2}\right]\times|f(x)|$$
$$+\left[\tfrac{1}{2}n_f\varepsilon+0.16236\frac{n_x\varepsilon|x|}{|x|+1}+\varepsilon\right]\times|f'(x)|$$

for positive x,

$$\left[\left(\frac{9}{32}\right)^{\frac{1}{3}}\frac{\left(n_f\varepsilon\right)^{\frac{2}{3}}}{|x|+1}+\tfrac{1}{2}n_x\varepsilon|x|\right]\times|f(x)|$$
$$+\left[\tfrac{1}{2}n_f\varepsilon+\tfrac{1}{2}n_x\varepsilon|x|+\varepsilon\right]\times|f'(x)|$$

for x equal to zero and

$$\left[\left(\frac{9}{32}\right)^{\frac{1}{3}}\frac{\left(n_f\varepsilon\right)^{\frac{2}{3}}}{|x|+1}+1.53980\frac{n_x\varepsilon|x|}{\left(|x|+1\right)^2}\right]\times|f(x)|$$
$$+\left[\tfrac{1}{2}n_f\varepsilon+1.16236\frac{n_x\varepsilon|x|}{|x|+1}+\varepsilon\right]\times|f'(x)|$$

for negative x.

Once again we shall not use δ directly, but shall instead use the difference between the floating point representations of $x+\delta$ and $x-\delta$.

Listing 4 provides the definition of a symmetric finite difference function object.

We again have two constructors; one for built in functions and one for user defined function, as shown in listing 5. Listing 6 gives the definition of the function call operator based upon our analysis.

Figure 2 plots the negation of the base 10 logarithm of our approximation of the error in this numerical approximation of the derivative of the exponential function as a dashed line and the true error as a solid line.

Clearly the error in the symmetric finite difference is smaller than that in the forward finite difference, although it appears that the accuracy of our approximation of that error isn't quite so good. That said, the average ratios between the number of decimal places in the true error and the approximate error of the two algorithms are not so very different; 1.21 for the forward finite difference and 1.24 for the symmetric finite difference.

Still, not too shabby if you ask me.

But the question *still* remains as to whether we can do any better.

Higher order finite differences

As it happens we can, although I doubt that this comes as much of a surprise. We do so by recognising that the advantage of the symmetric finite difference stems from the fact that terms dependant upon the second derivative largely cancel out. If we can arrange for higher derivatives to cancel out we should be able to improve accuracy still further.

```
template<class F>
class symmetric_difference
{
public:
  typedef F function_type;
  typedef typename F::argument_type argument_type;
  typedef typename F::result_type result_type;

  explicit symmetric_difference(
     const function_type &f);
  symmetric_difference(const function_type &f,
                       unsigned long nf);

  result_type operator()
     (const argument_type &x) const;

private:
  function_type f_;
  result_type   ef_;
};
```

Listing 4

Unfortunately, doing so makes a full error analysis even more tedious than those we have already suffered through. I therefore propose, and I suspect that this will be to your very great relief, that we revert to our original hand-waving analysis.

In doing so our choice of δ shall not be significantly impacted, but we shall have to content ourselves with a less satisfactory estimate of the error in the approximation.

We shall start by returning to Taylor's theorem again.

$$f(x+\delta)=f(x)+\delta\times f'(x)+\tfrac{1}{2}\delta^2\times f''(x)$$
$$+\tfrac{1}{6}\delta^3\times f'''(x)+\tfrac{1}{24}\delta^4\times f^{(4)}(x)+O\left(\delta^5\right)$$

```
template<class F>
symmetric_difference<F>::symmetric_difference(
  const function_type &f) : f_(f),
  ef_(pow(result_type(3UL)/result_type(2UL) *
      eps<result_type>(),
          result_type(1UL)/result_type(3UL)))
{
  if(ef_<result_type(eps<argument_type>()))
  {
    ef_ = result_type(eps<argument_type>())
  }
}

template<class F>
symmetric_difference<F>::symmetric_difference(
  const function_type &f,
  const unsigned long nf) : f_(f),
  ef_(pow(result_type(3UL*nf)/result_type(2UL) *
      eps<result_type>(),
          result_type(1UL)/result_type(3UL)))
{
  if(ef_<eps<argument_type>())
  {
    ef_ = eps<argument_type>();
  }
}
```

Listing 5

```
template<class F>
typename symmetric_difference<F>::result_type
symmetric_difference<F>::operator()(
   const argument_type &x) const
{
  const argument_type abs_x =
     (x>argument_type(0UL)) ? x : -x;
  const argument_type d =
     ef_*(abs_x+argument_type(1UL));
  const argument_type l = x-d;
  const argument_type u = x+d;

  return (f_(u)-f_(l))/result_type(u-l);
}
```

Listing 6

From this we find that the numerator of the symmetric finite difference is

$$f(x+\delta)-f(x-\delta)=2\delta\times f'(x)+\tfrac{1}{3}\delta^3\times f'''(x)+O(\delta^5)$$

Performing the same calculation with 2δ yields

$$f(x+2\delta)-f(x-2\delta)=4\delta\times f'(x)+\tfrac{8}{3}\delta^3\times f'''(x)+O(\delta^5)$$

With a little algebra it is simple to show that

$$f'(x)=\frac{\begin{bmatrix}8\left(f(x+\delta)-f(x-\delta)\right)\\-\left(f(x+2\delta)-f(x-2\delta)\right)\end{bmatrix}}{12\delta}+O(\delta^4)$$

Assuming that each evaluation of f introduces a single rounding error and that the arguments are in all cases exact this mean that the optimal choice of δ is of order $\varepsilon^{1/5}$ as shown in derivation 7.

By the same argument from pragmatism that we have so far used we should therefore choose

$$\delta=\left(|x|+1\right)\times\varepsilon^{1/5}$$

to yield an $O(\varepsilon^{4/5})$ error in our approximation.

With sufficient patience, we might continue in this manner, creating ever more accurate approximations at the expense of increased calls to f.

Unfortunately, not only is this extremely tiresome, but we cannot escape the fact that the error in such approximations shall always depend upon the behaviour of unknown higher order derivatives.

For these reasons I have no qualms in declaring finite difference algorithms to be a flock of lame ducks.

tutti: Quack! ■

Reference

[Harris10] Harris, R., 'You're Going to Have to Think; Why [Insert Technique Here] Won't Cure Your Floating Point Blues', *Overload* 99, ACCU, 2010

The optimal choice of δ

Noting that multiplying by a power of 2 never introduces a rounding error, the floating point approximation to our latest finite difference is bounded by

$$\frac{\begin{bmatrix}8\begin{pmatrix}f(x+\delta)\times(1\pm\frac{1}{2}\varepsilon)\\-f(x-\delta)\times(1\pm\frac{1}{2}\varepsilon)\end{pmatrix}\times(1\pm\frac{1}{2}\varepsilon)\\-\begin{pmatrix}f(x+2\delta)\times(1\pm\frac{1}{2}\varepsilon)\\-f(x-2\delta)\times(1\pm\frac{1}{2}\varepsilon)\end{pmatrix}\times(1\pm\frac{1}{2}\varepsilon)\end{bmatrix}\times(1\pm\frac{1}{2}\varepsilon)}{12\delta}\times(1\pm\tfrac{1}{2}\varepsilon)^2$$

which simplifies to

$$\frac{8\left(f(x+\delta)-f(x-\delta)\right)-\left(f(x+2\delta)-f(x-2\delta)\right)+O(\varepsilon)}{12\delta}$$
$$=f'(x)+O(\delta^4)+O\left(\frac{\varepsilon}{\delta}\right)$$

The order of the error is consequently minimised when

$$O(\delta^4)=O\left(\frac{\varepsilon}{\delta}\right)$$
$$O(\delta)=O(\varepsilon^{1/5})$$

Derivation 7

Outsource Your Self-Discipline

It's all too easy to skip those tedious but vital steps towards code quality. Filip van Laenen suggests getting someone else to do them.

Huge progress has been made in the past decade in the field of code quality assurance. Test-Driven Development (TDD) and Pair Programming are two key practices that have made a big difference and improved code quality substantially. Many readers will agree however, that there's still plenty of room for improvement. One important factor is that the IT systems that we build today are among the most complicated systems mankind has ever made. On the other hand, many of the tasks related to code quality are simple yet very repetitive, and therefore also extremely boring. As a consequence, doing the simple tasks like good and consistent code formatting, or proper function and variable naming, are usually more a question of conscience and self-discipline than knowledge and training. Is there an alternative to all the things we let slip due to bad conscience on a daily basis in our industry?

Why would you need assistance?

Even though there are many good arguments for pair programming, one of them is in my opinion wrong even though it is a very popular one: the idea that the second person will act to keep you on the straight-and-narrow. However, using a colleague to watch over your shoulder so that you give your functions appropriate names, that you comment your code where it's necessary, or that you don't write deeply nested functions, is a waste of resources we really can't afford in our industry. If you don't bother doing it when you're on your own then why do you punish your colleague with this boring task? Besides, you have a friend who's much better at it. In fact, he's much faster, does the job better and in a more consistent and precise manner than any of your colleagues, and he never gets tired of boring, repetitive, trivial jobs. That's right, I'm talking about your computer. Just think of how many lines of code it can analyse in a millisecond and compare it to how much time your colleague will need just to check that a single line of code is formatted correctly.

But even if you're a disciplined programmer who always formats his code consistently and practises TDD according to the book, your computer can be of great help. Remember the principles of TDD? The flowchart in Figure 1 gives a quick overview: first you write a failing unit test, then you write only as much source code as needed to make the test pass. Next you refactor the code if needed, and unless you're done, go back to the start and write a new failing unit test. But how do you know whether you should refactor your code or not, and how do you know when you've refactored it enough? Everybody has an idea about what 'good code' looks like but it's often a very subjective thing. What looks 'good enough' today may look 'just below the bar' on another day. And once you've started refactoring it may be hard to stop when it's just 'good enough' and instead continue to polish the code that little bit more.

Filip van Laenen is a chief engineer at the Norwegian software company Computas, which he joined in 1997. As the company's competency lead for Software Engineering, it's his job to inspire his colleagues and help them improve their software quality skills. He can be contacted at f.a.vanlaenen@ieee.org.

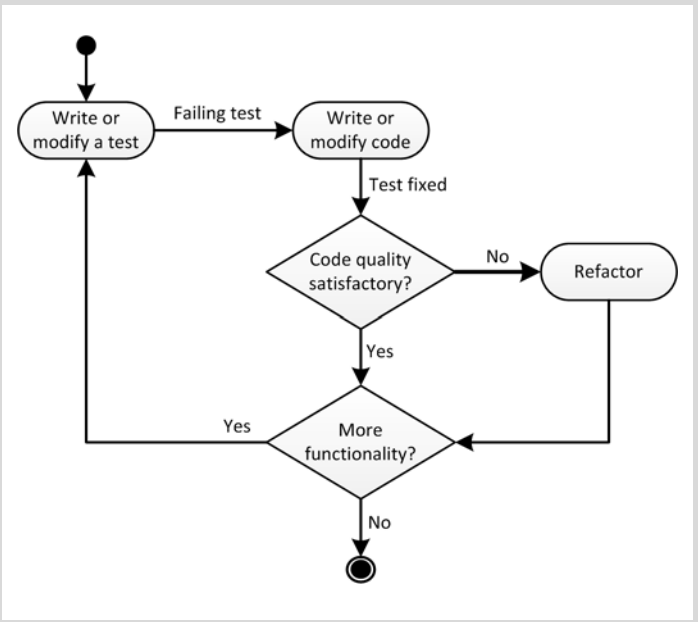


Figure 1

Just to give an example: unless you're using a programming language in which calls to subroutines are very expensive, having fifteen levels of nesting in a single method is definitely a sign of bad code quality. But if you start refactoring the method how do you know when you should stop? Reducing the nesting to six or seven levels clearly isn't 'good enough', but what about three or four levels? And is three levels 'good enough' so that you can move on to the next problem, or should you continue until all functions have no more than one level of nesting?

What's even more difficult is getting four, five or even more programmers to agree on what will be the minimal code quality requirements for a project. And that's before you want them to apply those minimal requirements to every piece of code in the project in a consistent way and without long (or even endless) discussions, throughout the lifetime of that project. Wouldn't it be great to get help from an impartial judge, who can give his opinion about the quality of a certain piece of code based on a set of objective criteria? Again, this is where the computer can help you, e.g. applying some automated rules against your code in order to check that it meets the minimal code quality requirements you all agreed upon.

How your computer can help

There are many tools available that you can use to check the quality of your code. Some of them can be plugged into the build process so that all developers in your project team can follow the exact same coding standard, and that all code adheres to the same minimum code quality requirements. You'll get the best results if you let the build process fail whenever

nobody will read the reports, let alone act on them to clean up the code

violations are found, but that's not always possible, or it may not fit with your organization or the project you're working on right now. But in my experience when you use the tools to only generate reports, nobody will read the reports, let alone act on them to clean up the code.

So what are the tools that we can use to improve code quality? Automatic code formatting, static code analysis, test coverage reports and mutation testing are four examples of tools that can take care of some of our self-discipline. Let's have closer look at each of these tools, to see how they work and what they can do for us.

Automatic code formatting makes sure that all code conforms to the same coding standard all the time. Sure enough, these tools aren't able to format all code exactly the way you wish, but on the other hand they never forget to format it, they're never sloppy, and they never change their mind. That is unless you change the code formatting rules, but if you ever do that the tool will reformat all the code according to those new rules instantly. Many people are sceptical about automated code formatting tools, but I still think it's better to have 100% of the code formatted in a consistent way all the time than to have 10% inconsistantly formatted because a tool would not be able to format the last 0.1% of the lines exactly to your taste.

Most modern IDEs have some sort of code formatting tool included, but it's often possible to do better than that. Indeed, one of the problems is that if you want to use an IDE's code formatting tool in your project you're still relying on your developers manually running the formatting tool against every code file they've modified before they check in. Furthermore, making sure every developer in your project uses the same code formatting rules can be a challenge too if it can't be distributed easily and used by the IDE automatically.

In my experience you get the best results if you can run the code formatting tool completely automatically, virtually outside the control of the developers. Maybe it can be part of the building process, e.g. using the Jalopy-plugin to Maven in a Java project [Jalopy]. Alternatively you may be able to fire the code formatting tool from a hook of your version control system, e.g. upon check-in of code files. But if neither of them is possible in your project, you may resort to using tools like Checkstyle [Checkstyle] to check whether the code conforms to all (or at least some) of your code formatting rules. And if even that's not possible you can always run `grep`, e.g. using a regular expression like `\{\s*\}` to find empty blocks in curly-braced languages.

Static code analysis can be used to find undesired patterns in your code. These patterns range from simple things like empty blocks and writing to the console over deeply nested functions, confusing or overly complex code to known 'anti-patterns', and potential bugs. Notice that many compilers have options to enable some static code analysis, usually in the form of warnings, but dedicated tools have a wider range of rules and patterns they can check. If you want to apply static code analysis on an existing project, start with a small rule set and pick from time to time a new rule that looks useful for you. Remove all violations of the rule from your project, one at a time. Then, when you're done, consolidate the rule by including it in your rule set so that you (or your colleagues) won't create new violations of it. When you get the hang of it and you've implemented most of the rules that looked useful to you, maybe you should consider creating your own rules to get rid of some of those particular bad habits you or your colleagues have. And if you're using an open source tool, maybe you even want to donate them back to the project so that others can benefit from them too.

There's a large variety of tools that can be used to analyse your code statically. In fact, compilers often have some options you can switch on to do some very basic static code analysis. The already mentioned Checkstyle focuses mostly on coding style but it also does some static code analysis. FindBugs [FindBugs] and PMD [PMD] are two other tools for the Java language, the former being a bit more oriented towards finding bugs per se whereas the latter casts its net more broadly. If you're a .Net programmer you should probably check out FxCop [FxCop]. Lint is the original static code analysis tool for C, and Cppcheck [Cppcheck] is probably the de facto standard static code analysis tool for C/C++. If you program in another language, or you want to check out even more static code analysis tools, be sure to check out Wikipedia's overview [Wikipedia].

It should be noted that static code analysis on dynamically typed languages is a difficult task. In fact, one could almost say that's so by definition if you notice the 'static' on the one side and the 'dynamic' on the other. Nevertheless, even for a language like Ruby there are some tools available, e.g. Roodi [Roodi]. It's even possible to create your own static code analysis tools using, amongst others, regular expressions and string functions. A few years ago, I was on a project where we had a simple tool making sure all SQL scripts followed some basic rules.

Test coverage tools will reveal which parts of your code aren't tested by your automatic tests, or maybe even not in use at all. Sometimes low test coverage isn't an issue, e.g. when it's difficult to set up automatic tests against a simple and stable interface that's easy to test manually. But the core of your system, the part where most of your business logic resides, should have a test coverage rate as close as possible to 100%. Aim for the high numbers in those parts of your system, not just 60% or 70%. If you can't reach your goal, try again before you lower your goal or make a local exception. And don't forget to consider deleting some code – you'll be surprised how often that's the right decision.

In this category too there are many tools available to help you in your project. I have used both EMMA [EMMA] and Cobertura [Cobertura] in Java projects with great success, and Rcov [Rcov] in some Ruby projects. If you're a .Net programmer, NCover [NCover] is probably the tool for you, but there are many others. Just as for static code analysis tools, Wikipedia has a page [Wikipedia2] with a good overview of tools available in a number of programming languages.

Mutation testing is a very powerful tool, but sometimes can be a bit annoying and irritating. It can be described as a sort of automated code critique, and in the beginning it can be hard to accept the results it produces. What it does is that it makes simple changes ('mutations') to the source code that are guaranteed to change the behaviour of the system. Switching a condition or removing a line of code are good examples of changes that should be noticed somewhere. When it has made a mutation, the tool checks that at least one unit test starts to fail. If no test fails clearly

old problems and bad habits that have plagued your project over a long time, will disappear, and never come back

something is wrong. Maybe the tool found a condition that isn't covered by a unit test, and you should add one? Or maybe it found a branch that can't be reached and you can delete some code? I have to confess that, even though I have used it for years, there are still occasions where I have to manually apply the mutation to the code and run the unit tests again, just to accept that what it says is correct. On the other hand I've learned a lot from it, even though it can sometimes be very irritating that it so meticulously points out the mistakes I make in unexpected places.

Personally, I haven't had much success yet running mutation testing in any language other than Ruby. There may be something particular about Ruby that makes it well suited to mutation testing, or it could be that the programmers behind Heckle [Heckle] found a clever way to make the set-up of the mutation testing easy. I would like to mention Jester [Jester], Jumble [Jumble] and PIT [PIT] as three mutation testing tools for the Java language that look promising, but so far don't seem to be quite mature yet. I hope to see some evolution here, because mutation testing is one of the things I really miss when programming in Java.

Figure 2 explains how the code quality tools discussed fit in with TDD. Automatic code formatting doesn't appear in the figure because it happens behind the scenes and is therefore totally transparent to whichever development method you use otherwise. Static code analysis, code coverage and mutation testing are part of the decision box in the middle and help to find an answer to the question of whether the code quality is good enough. Note that code coverage reports and mutation testing often will give you the inspiration, even if they don't actively force you, to write new unit tests, and therefore in a sense can send you straight back to the 'Write or modify a test' box.

Getting started

If you want to use any of these tools, introduce them slowly in your project. It's always a good idea to start with the generation of some reports just to see how you're doing. Then try to fix the simple things, and start automating your code quality requirements. As you continue to use the tools and add more and more requirements, you'll learn how the tools work, and you can start to create your own rules or extensions. But don't add requirements you don't understand, and maybe even more important, how violations should be fixed, because that will bring you problems. Over time, you'll see that old problems and bad habits that have plagued your project over a long time, will disappear, and never come back.

It's important, however, to understand that these tools won't solve all your code quality issues. You and your colleagues will still have to use your brains while you're programming, because not all software quality requirements can be expressed in rules that can be automated. But these tools can automate some of the most boring tasks, so that you can concentrate more on the creative, fun part of programming. And isn't that why we all started programming in the first place? ■

References

This article was based on a lightning talk I held at the ROOTS 2011 conference in Bergen, Norway earlier this year.

[Checkstyle] http://checkstyle.sourceforge.net/
[Cobertura] http://cobertura.sourceforge.net/
[Cppcheck] http://cppcheck.sourceforge.net/
[EMMA] http://emma.sourceforge.net/
[FindBugs] http://findbugs.sourceforge.net/
[FxCop] http://msdn2.microsoft.com/en-us/library/bb429476.aspx
[Heckle] http://docs.seattlerb.org/heckle/Heckle.html
[Jalopy] http://jalopy.sourceforge.net/
[Jester] http://jester.sourceforge.net/
[Jumble] http://jumble.sourceforge.net/
[NCover] http://ncover.sourceforge.net/
[PIT] http://pitest.org/
[PMD] http://pmd.sourceforge.net/
[Rcov] https://github.com/relevance/rcov/
[Roodi] http://roodi.rubyforge.org/
[Wikipedia] http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
[Wikipedia2] http://en.wikipedia.org/wiki/Code_coverage

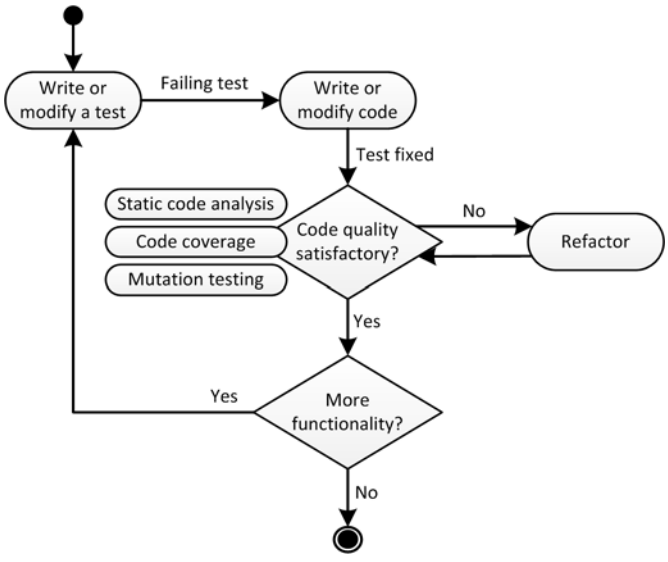


Figure 2

Picking Patterns for Parallel Programs (Part 1)

Designing programs for multi-core systems can be extremely complex. Anthony Williams suggests some patterns to keep things under control.

There are many ways of structuring parallel programs, and you'd be forgiven for finding it difficult to identify the best solution for a given problem. In this series of articles I'm going to describe some common patterns for structuring parallel code, and for communicating between the different parts of your program that are running in parallel. I'm also going to provide some basic guidelines for choosing which patterns to use in a given scenario.

This article will describe some simple structural patterns; further structural patterns and communication patterns will be covered in later articles.

Structural patterns

Structural Patterns are about the general 'shape' of a solution: how the data and tasks are divided between threads, how many threads are used, and so forth. Each structural pattern has a different set of trade-offs with regards to performance, scalability, and so forth. Which pattern to choose depends strongly on the characteristics of the particular problem being solved.

We'll start by looking at LOOP PARALELLISM.

Loop parallelism

The most basic of structural patterns is LOOP PARALLELISM. The basic premise is that you have something like a for loop that applies the same operation to many independent data items.

This is your classic embarrasingly parallel scenario, and scales nicely across as many processors as you've got, up to the number of data items – if you've only got 5 data items to process you cannot make use of more than 5 processors with a single layer of loop parallelism.

This is such a common and simple scenario that frameworks for concurrency and parallelism frequently provide a `parallel_for_each` operation or equivalent. e.g.

```
std::vector<some_data> data;
parallel_for_each(data.begin,
                  data.end(),
                  process_data);
```

or, for a compiler that supports OpenMP:

```
#pragma omp parallel for
for(unsigned i=0;i<data.size();++i) {
  process_data(data[i]);
}
```

The key thing to remember about Loop Parallelism is that the operation in the loop must depend solely on the loop counter value, and the execution for one loop iteration must not interact with the data used by any other loop iteration. This is absolutely crucial since the order of execution of iterations cannot be guaranteed, and may vary from run to run or from machine to machine. Consequently you cannot guarantee that iterations would be run in the correct order for any loop-carried dependencies and concurrent access to the same variables can lead to a data race and undefined behaviour.

Though some frameworks provide mechanisms for handling such loop-carried dependencies, the presence of such dependencies typically means that your problem is not ideally suited to loop parallelism, and an alternative pattern may be more appropriate.

Fork/Join

Also called 'divide and conquer', the essential idea of the FORK/JOIN pattern is that a task is divided into two or more parts, tasks are run in parallel (forked off) to process these parts, and then the driver code waits for these parallel tasks to finish (joins with them).

The Fork/Join pattern is often used recursively with each task being subdivided into its own set of parallel tasks, until the task cannot usefully be divided any further. Listing 1 shows how such a recursive technique could be used to implement a parallel Fast Fourier Transform algorithm for a power-of-2 FFT.

```
template<typename Iter>
void do_fft_step(Iter first,Iter last) {
  unsigned long const
     length=std::distance(first,last);
  if(length<minimum_fft_length) {
      do_serial_fft_step(first,last);
  } else {
      Iter const mid_point=first+length/2;
      auto top=std::async([=]{
          do_fft_step(first,mid_point);});
      do_fft_step(mid_point,last);
      top.wait();
      merge_fft_halves(first,mid_point,last);
  }
}

template<typename Iter>
void parallel_fft(Iter first,Iter last) {
  prepare_fft(first,last);
  do_fft_step(first,last);
  finalize_fft(first,last);
}
```

Listing 1

Anthony Williams is the author of *C++ Concurrency in Action*, published by Manning. He lives at the far end of Cornwall, where he can gaze longingly at the sea from his office window when trying to solve knotty problems.

it is important to ensure that the pipeline stages are all of similar duration

In this case the merging steps mean that you can't readily process each section independently with loop parallelism, but the recursive division allows for parallel execution of the smaller steps.

This uses `std::async` with the default launch policy, so the C++ runtime can decide how many threads to spawn for the `std::async` tasks, and switch to using synchronous tasks which run in the waiting thread rather than asynchronous tasks when there are too many threads running. Also, rather than submitting a second `async` task for the 'bottom half' we execute this directly. This avoids the overhead of creating the `std::async` data structures, and potentially creating a new thread for the task when the current thread is just going to wait anyway. Put together, this therefore allows the task to scale with the number of processors.

FORK/JOIN works best at the top level of the application, where you are in control of how many tasks are running in parallel – if it is used deep within the implementation of an already-parallel algorithm then you may well find that all the available hardware parallelism is already being used by other parts of the application, so there is no benefit. You also need to be able to see how many existing tasks are running in parallel, so that you can avoid excessive oversubscription of the processors.

By its very nature, the Fork/Join pattern produces 'bursty' concurrency – initially there is no concurrency, then there is a 'burst' of parallel tasks, then they are joined and there is again no concurrency. If your application is done after one such cycle then this is fine, but if your application requires a number of Fork/Join cycles then there is spare processing power that could potentially be used during each join/re-fork period.

Finally, you need to watch for uneven workloads between the tasks – if one task finishes much later than the others then you are wasting any hardware parallelism that could potentially be used whilst the master task is waiting for the long-running task to finish. e.g. if you are searching for prime numbers then you don't want to divide your number-line into equally-sized ranges – it is much quicker to check the lower numbers for primality than the higher ones, so if each task deals with 1 million numbers, the task starting at 1 will finish much quicker than the task starting at 100 trillion.

Pipelines

The PIPELINE pattern handles the scenario where you have a set of tasks that must be applied in sequence, the output of one being the input to the next, and where this sequence of tasks must be applied to every item in a large data set.

As you would expect for a pipeline, the order of the data that goes through the pipeline is preserved – the data that comes out the end first is the result of applying the operations in the pipeline to the data that was put in to the pipeline first, and so forth.

Another characteristic of pipelines is that there is a startup period, during which the pipeline is being filled, and thus the parallelism is reduced. Initially there is only one data item in the pipeline, being processed by the first task; once that is done then a new item can be processed by the first task, whilst the first item is processed by the second task. Once the first item has made it all the way through the pipeline, the pipeline will remain at maximum parallelism until the last item is being processed. There will then be a draining of the pipeline, as the last item makes its way through each task, the parallelism reducing with each step. (See Figure 1.)

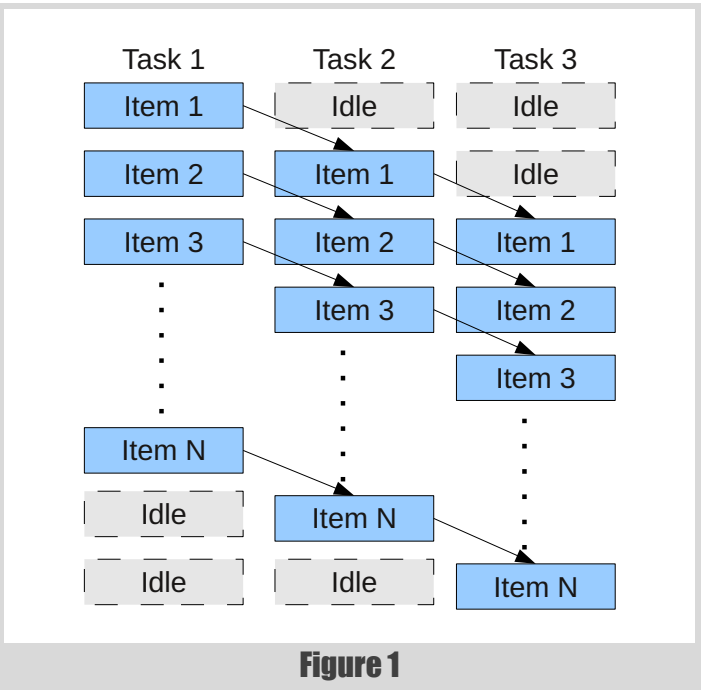


Figure 1

As should be obvious, the maximum parallelism that can come from the pipeline itself is the number of tasks in the pipeline. If you have more hardware parallelism available then this will not be utilised without further thought. One possibility is to use parallelism within each pipeline stage (which may well lead to 'bursty' parallelism as we saw with fork/join), and another is to run multiple pipelines in parallel (in which case you need to be careful that the order is preserved if it is important).

Either way, it is important to ensure that the pipeline stages are all of similar duration – if one stage takes much longer than the others then it will limit the rate at which data can be processed, and thus processors running other stages will potentially be running idle as they wait for the long running stage to complete.

One potential downside of pipelines is the way they interact with caches. If each task is fixed to a single processor, then as stage *N* finishes processing a data item, the output of stage *N* has to be transferred from the cache of the processor running stage *N* to the cache of the processor running stage *N*+1. Depending on the complexity of the task and the size of the data, this may take a significant amount of time.

The alternative is to have the whole pipeline run on each processor, with some additional logic to ensure that any required ordering between data items is preserved. This has the benefit that the data no longer has to be transferred between caches, as it is right there waiting for the next task in

actors are not good for short-lived tasks, as the overhead of setting up an actor and managing the message queue can outweigh the benefits

the pipeline. However, in this case it is the code for the task that must be loaded into the instruction cache – by running the whole pipeline on each processor we increase the chance that the code for each stage has been dropped from the cache, and will thus have to be reloaded. Again, this can take a noticable amount of time.

As with everything, if performance is important, then time various options and choose the best for your particular application.

Actor

The ACTOR model is basically message passing Object Orientation with concurrency. Message sending is asynchronous, so the code that sends a message does not wait for the receiving object to handle it, and each object (actor) responds to incoming message asynchronously on its own thread. This is the model used by Erlang, where each Erlang process is an Actor. It is also similar to the model used by MPI, and essentially synonymous with Hoare's Communicating Sequential Processes. (See Figure 2.)

In its purest form, there is no shared state in the Actor model, and all communication is done via the message queues. Some languages (such as Erlang) enforce this; in C++ it is your responsibility to follow the rules.

The prime benefit is that each actor can be analysed independently of the others – incoming messages are queued until the actor is ready to receive a message so it is only the order of messages that matters. You can therefore test that each actor sends out the appropriate sequence of messages given a particular input sequence. If you stick to the rule that the only communication between actors is via the message queues (no shared mutable state) then such basic testing is sufficient, and it is certainly much easier than testing multiple interacting threads.

The lack of shared mutable state has another benefit – data races are impossible. You can still potentially get race conditions, where two or more actors send a message to the same recipient, and the order the messages arrive affects the outcome, but this is easier to handle as you can just test with all possible order combinations and verify that the recipient does something sensible in each case.

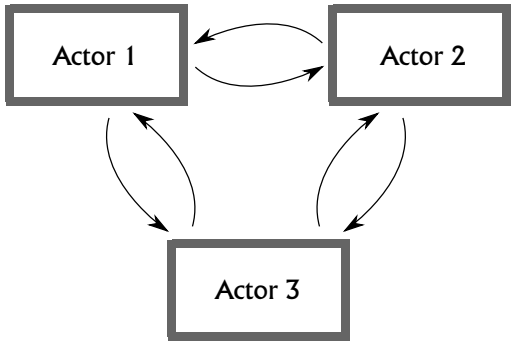


Figure 2

```
int main() {
  struct pingpong { jss::actor_ref sender; };
  jss::actor pp1([] {
    for(;;) {
      jss::actor::receive().match<pingpong>(
         [](pingpong p) {
        std::cout<<"ping\n";
        p.sender.send(
           pingpong{jss::actor::self()});
      });
    } });
  jss::actor pp2([] {
    for(;;) {
      jss::actor::receive().match<pingpong>(
         [](pingpong p) {
        std::cout<<"pong\n";
        p.sender.send(
           pingpong{jss::actor::self()});
      });
    } });

  pp1.send(pingpong{pp2});
}
```

Listing 2

Another benefit is that the independence makes actors easy to reason about, as each can be considered on its own. You can, for example, make each actor a state machine with well-defined transitions and behaviours.

One downside is that actors are not good for short-lived tasks, as the overhead of setting up an actor and managing the message queue can outweigh the benefits. Also, message passing isn't always the ideal communication mechanism; sometimes it's just more efficient to carefully synchronize access to shared state.

Finally, the scalability is limited to the number of actors – if you've only got 3 actors then the actor model won't scale to more than 3 processing cores unless you can make use of additional concurrency within each actor. Of course, many problems can be divided into very fine-grained tasks and thus be constructed out of lots of actors, but this then relates back to the task size – there is no point generating an actor just to add two integers, as even the message passing overhead will far exceed the actual cost of the operation.

Listing 2 shows a simple use of actors in C++, with each actor sending a message to the other, and then waiting for the next message in return.

Next time

In part 2 of this article I'll cover more structural patterns, starting with SPECULATIVE EXECUTION. ■

Intellectual Property – a Crash Course for Developers

Interpreting law is a tricky business. Sergey Ignatchenko introduces someone who can help you avoid expensive mistakes.

Being on vacation, I wasn't able to write my usual article for *Overload* myself, so I have asked my friend Samuel 'Lawyer' Bunny, Professor of Law at Efrafa University, to write a brief description of intellectual property law aimed at software developers. If some of you consider it as lacking in controversy, don't worry – I will be back from my vacation in time to write the next one.

'No Bugs' Bunny

Disclaimer: opinions within this article are those of Samuel Bunny, and do not necessarily coincide with the opinions of the translator and *Overload* editors; please also keep in mind that translation difficulties from Lapine (like those described in [LoganBerry2004]) might have prevented an exact translation. In addition, both the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

In addition, as always with legal matters, it should be mentioned that neither the author nor translator are practising lawyers and that nothing in this article should be interpreted as legal advice. If you are in need of legal advice, you should ask your attorney.

There is nothing more powerful than an idea whose time has come.

Victor Hugo (one of the authors behind the 'Berne Convention' on copyright)

Yet Another Disclaimer: in this article I will not address questions about the usefulness or morality of the existing laws (I'll leave that to 'No Bugs' when he's back). My goal here is to explain the basic concepts as they stand now, and not to discuss whether they should or should not be changed. Also it should be noted that intellectual property laws differ from one jurisdiction to another, and while an effort has been made to write only about issues which are widely acceptable, there is no guarantee that any statement will be valid where you are.

Basically, all Intellectual Property rights can be divided into three distinct categories: copyright, patents and trademarks (there are also other related rights, like industrial design rights, which we won't discuss here). These three categories of rights, as a rule of thumb, provide separate and independent protection. It means, in particular, that the very same software program can be protected by copyright, by patent(s), and by trademark(s).

Copyright

Out of the three protection mechanisms I'll look at, copyright seems to be the simplest concept. If you have written a poem you are entitled to a certain set of rights concerning your work, including the right to prevent others from using it, the right to prevent others from modifying it, and so on. In most places you are not required to register your copyright, and are not even required to put copyright notice (like 'Copyright 2011 Samuel Bunny') for your rights to be enforceable.

So far so simple. The complications begin when we start to consider situations when the copyrighted work is allowed to be used without the permission of the copyright holder. This concept is known as 'fair dealing' in the UK, and 'fair use' in the US. It allows some limited use of copyrighted works. For example, you can usually use some quotes from the original book, even when you're writing a negative review about it and don't have permission from the author. It is an effective defence in many practical scenarios, for example, Wikipedia claims 'fair use' for certain types of images in limited circumstances.

It should be noted that there is a rather common misconception that copying for non-commercial purposes automatically provides a 'fair use' defence. This is dead wrong: while non-commercial can be one of the factors when 'fair use' is considered, courts in various jurisdictions have held that copying for non-commercial purposes can constitute copyright infringement, including criminal copyright infringement. You need to keep in mind that non-commercial purposes does not guarantee 'fair use' to be a valid defence. In general, the determination of 'fair use' tends to be very complicated, and the safest thing is not to rely on it until you've got advice from your attorney who specializes in IP law (unfortunately I'm already retired, so I won't be able to help you – sorry).

Impact of copyright on software developers

For software developers, the basics of copyright can be summarized in a very simple way: don't copy-and-paste third-party code into your own code. While this is not an exact definition (for example, replacing copy-and-paste with re-typing the source code won't help to avoid liability), it is a good starting point.

Furthermore, we need to mention that even if you're using third-party code in a modified form, you can still be found liable for copyright infringement. The safest thing (and the most detrimental for solicitor salaries) is to avoid any kind of copying of source code which is not yours.

One other thing to remember (to be discussed in more detail below) is that even copying open-source code can be dangerous and detrimental for your company, so check with your legal department (or with the project steering committee) what you are allowed to use in your project, and what you are not. And while we are on the subject: the absence of a copyright notice does not mean that the material is not copyrighted – most text, including source code, is copyrighted by default without any notice – so it is better not to copy anything before clarifying it.

Patents

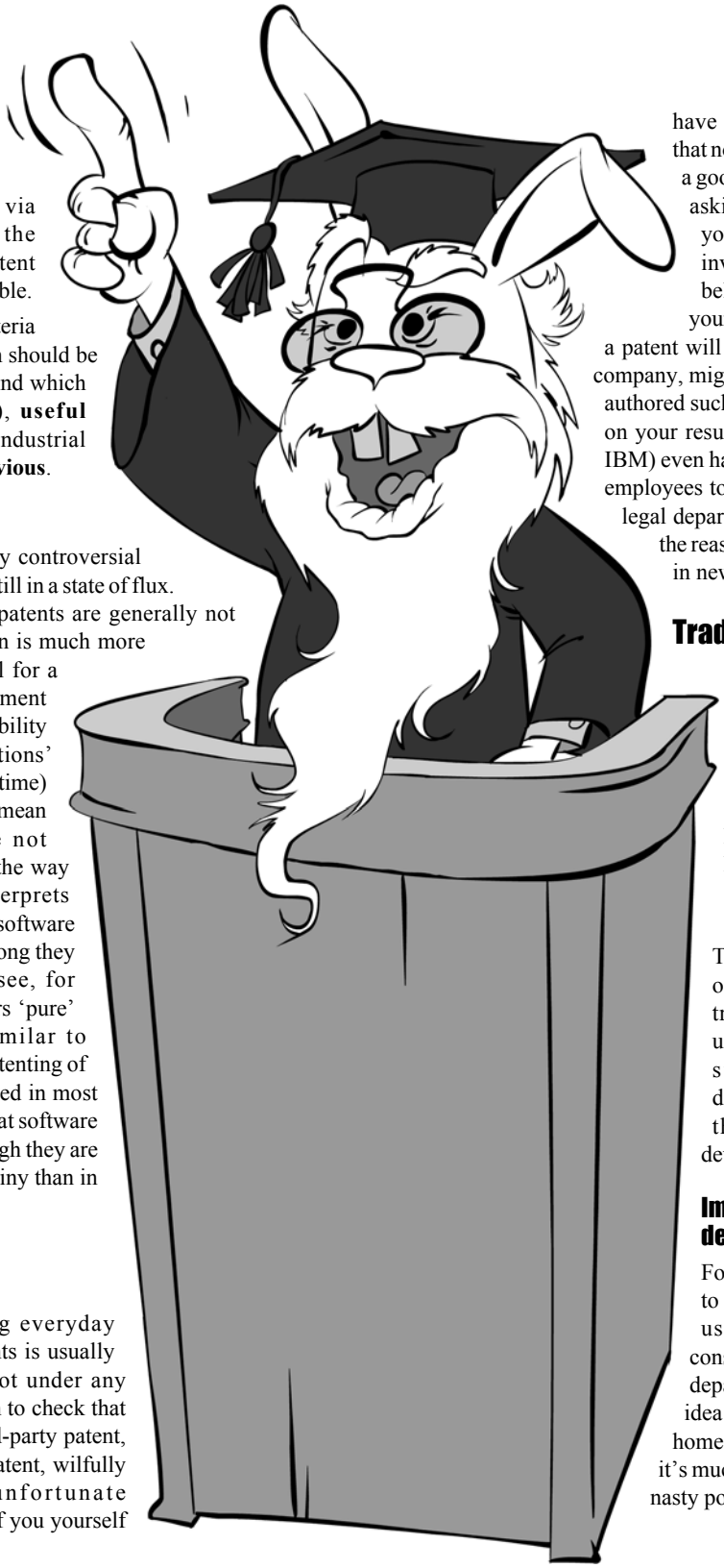
As we have seen, normally copyright arises at the very moment that an author has written something, and no registration is necessary. In contrast, a patent arises only when the inventor goes to a competent patent office and says 'I have a new invention to patent' and submits a patent application which describes the invention in detail. From this point on inventors are allowed to use the words 'patent pending' when they describe their

Samuel 'Lawyer' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams [Adams].

Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

product. Still, this is merely the start of a lengthy process known as 'patent prosecution', when the task of the inventors – usually via patent attorneys representing the inventor – is to persuade the patent office that their invention is patentable.

In general, there are three main criteria for being patentable: the invention should be **novel** (so no 'prior art' can be found which describes the same invention), **useful** (known as 'utility' in US and 'industrial applicability' in EU), and **non-obvious**.



Software patents

Software patents are an extremely controversial subject and legislation on them is still in a state of flux. As of 2011, in the US software patents are generally not prohibited. In the EU the situation is much more complicated: while the 'Proposal for a Directive of the European Parliament and of the Council on the patentability of computer-implemented inventions' (which caused much debate at the time) was rejected in 2005, this doesn't mean that software inventions are not patentable in the EU. Currently, the way the European Patent Office interprets existing law is that (very roughly) software inventions are only patentable as long they have a 'technical character' (see, for example, [EPO2007]). It considers 'pure' software algorithms to be similar to mathematical formulae, and the patenting of mathematical formulae is prohibited in most countries, and in practice means that software patents are possible in the EU though they are subject to substantially more scrutiny than in the US.

Impact of patents on software developers

For software developers doing everyday development, the impact of patents is usually not large. Generally, you are not under any obligation to make a patent search to check that your code is not infringing a third-party patent, though if you do know about a patent, wilfully infringing it might have unfortunate implications. On the other hand, if you yourself have made something which you're sure that nobody else has done before it might be a good idea to approach your management asking if they're interested in patenting your invention (as a rule of thumb, any invention made during working hours belongs to the company). Assuming that your invention is viable, going ahead with a patent will certainly improve your status in the company, might result in a bonus, plus the line 'co-authored such and such patent' will look very nice on your resume. Some companies (most notably IBM) even have special programmes to encourage employees to submit inventions to a dedicated IP legal department (and this programme is one of the reasons why IBM has been a world leader in new patents for many years).

Trademarks

The first de-facto trademarks appeared when a blacksmith in a small town decided to put a special sign on his swords, to make sure that whenever somebody is impressed with the sword quality, he can identify the smith and come to him to buy another sword. The very same idea is now known as 'trademark'.

Trademarks can be either 'registered' or 'unregistered'. For registered trademarks, usually the symbol ® is used, for unregistered trademarks the symbol ™. There are further differences between the two, but they're of no interest to software developers, so we'll leave them aside.

Impact of trademarks on software developers

For software developers, the only thing to remember about trademarks is not to use third-party trademarks without consulting management and/or the legal department. While it might seem a good idea to add a huge Apple logo onto your home page to promote your Mac programs, it's much safer to ask before doing it to avoid nasty potential problems.

Clearance from management should be obtained. Otherwise, it can lead to nasty legal problems

Licences

We have discussed the three main forms of intellectual property, but many software developers will ask: how do licences fit into this picture?

Technically, licences do not belong to intellectual property law – instead they're usually considered as a part of contract law. For example, when you're buying a software product you're buying not only the physical CD containing the binary executable, you're also entering into a contract which allows you to use the product (within certain limitations specified in the licence).

With open-source licences the situation is usually the following: when you get the source code you are not automatically allowed to use it (because it is copyrighted). The only reason that you are able to use it is because you have also entered into a contract with the copyright holder; this contract will allow you to use copyrighted code in exchange for agreeing to observe some limitations stipulated in the licence. In case of common open-source licences, acceptance of licence terms is implicit when you start to use the code (this means that if you're using the code you must agree to licence terms).

Impact of licences on software developers

For software developers, the impact of licences is very similar to the impact of copyright, as described above: basically, you should not copy-and-paste third-party code into your own code unless you've got clearance from the management (legal department, project steering committee, etc.). In addition, don't use third-party libraries without such clearance.

It should be noted that even if the library is open-source, clearance from management should still be obtained. Otherwise, it can lead to nasty legal problems.

An example: you're working for a big company. One day you find a good library which you're craving to use in your project. You've checked that it is an open-source licence so you've decided to go ahead without letting anybody know. Two years down the road, an auditor has found this code which appears to be licensed under the GNU General Public License [GPL]. Now you're in serious trouble because of the 'viral' nature of the GPL: it requires that all the code compiled together with a single piece of GPL code must itself be licensed under the GPL. Your company now faces a dilemma: either to licence the whole application under the GPL, which is an unlikely choice for the company, or to scrap your code (and probably yourself). If the third-party code is not GPL, but the GNU Lesser General Public License [LGPL] it might be more acceptable for your company though clearance is still required. In particular because figuring out who really owns the code might be tricky, and because certain paperwork might be necessary to ensure compliance with the LGPL.

A similar situation can easily arise even within open-source projects: for example, using GPL'd code in Apache products is prohibited [Apache], so if you use GPL'd code within an Apache project it will be a waste of time not only for you putting it there but also for somebody else to remove it later. And BTW, despite it seeming counter-intuitive – while LGPL has a chance of being allowed for commercial projects, it is still prohibited for an Apache project.

As you can see all this licensing business is extremely convoluted, so I will summarize the bottom line: don't rely on your own judgement: don't use anything from third-parties unless you have very clear permission from management (legal department, project steering committee, etc.). Asking in advance can save you and your company/project lots and lots of time. ■

References

[Adams] http://en.wikipedia.org/wiki/Lapine_language

[Apache] http://www.apache.org/legal/resolved.html

[EPO2007] EPO Board of Appeal Case Law Special edition 6 l Official Journal 2007

[GPL] www.gnu.org/copyleft/gpl.html

[LGPL] www.gnu.org/licenses/lgpl.html

[Loganberry2004] David 'Loganberry', Frithaes! - an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html