Catastrophic Cancellation; differentiating a program

Figure 1.16 very nicely illustrates the devastating effect of catastrophic cancellation. Here are the facts.

Subtraction of two nearly equal numbers is one of the few floating-point operations which is carried out error-free. Paradoxically, it is the one operation one should be most worried about, for the following reason: The numbers `x` and `y` whose difference $\mathtt{x} - \mathtt{y}$ we compute usually are, themselves, somewhat in error, even if all we did was compute them as $\mathtt{x} \leftarrow \mathtt{fl}(x)$, $\mathtt{y} \leftarrow \mathtt{fl}(y)$. With luck, these errors are not much bigger than those expected from rounding, hence are *relatively* small. However, if `x` and `y` are nearly equal, then their difference $\mathtt{x} - \mathtt{y}$ will be much smaller than either `x` or `y`, hence the *relative* error in the difference $\mathtt{x} - \mathtt{y}$ as an approximation to $x - y$ will be much larger.

In short: while the floating-point difference of nearly equal numbers can be computed without error, such differencing magnifies the relative importance of errors already present in those two numbers.

Here is a simple example, the function $f(x) = x - \sin(x)$ near $x = 0$ in which the error in the computed value of $\sin(x)$ inherited by the calculated $x - \sin(x)$ is relatively huge. The following table lists the function values obtained by evaluating `x - sin(x)` in `matlab` as well as the first 6 significant digits of the correct value of $x - \sin(x)$, along with absolute and relative error.

x	x - sin(x) (exact)	x - sin(x) (computed)	Absolute Error	Relative Error
1e-01	1.66583e-04	1.66583e-04	2.75241e-15	1.65e-11
1e-02	1.66666e-07	1.66666e-07	4.26534e-19	2.56e-12
1e-03	1.66667e-10	1.66667e-10	5.67063e-21	3.40e-11
1e-04	1.66667e-13	1.66667e-13	5.10014e-21	3.06e-08
1e-05	1.66667e-16	1.66667e-16	6.18234e-22	3.71e-06
1e-06	1.66667e-19	1.66654e-19	1.29343e-23	7.76e-05
1e-07	1.66667e-22	1.72054e-22	5.38690e-24	3.23e-02
1e-08	1.66667e-25	0.00000e+00	1.66667e-25	1.00e+00
1e-09	1.66667e-28	0.00000e+00	1.66667e-28	1.00e+00
1e-10	1.66667e-31	0.00000e+00	1.66667e-31	1.00e+00

The unit round-off in `Matlab` is about 10^{-17} and, correspondingly, for $x = 10^{-7}$, the computed value for $f(x)$ has only its first two significant digits accurate. For smaller values of x, the computed answer is nonsense.

The only defense against such **catastrophic cancellation** is to reformulate the expression to be calculated in such a way that this subtraction can be carried out *symbolically*.

In the above example, this can be done by observing that

$$\sin(x) = x - x^3/3! + x^5/5! - \cdots,$$

hence

$$x - \sin(x) = x^3/3! - x^5/5! + \cdots,$$

and, depending on how close x is to 0 and how small a relative error I would like in the computed value, I would use one or two or three terms of this series. E.g.,

$$x - \sin(x) = x^3/3! + error(x),$$

with $|error(x)| \le |x^5/5!|$ (since the series is alternating), hence, for $|x| < 10^{-5}$, we get a relative error of about $|x^5/5!|/|x^3/3!| = |x^2/20| < 10^{-11}$, or more than 11 significant decimal digits of accuracy.

As another example, there is catastrophic cancellation in the calculation of $f(x) = \sqrt{1+x} - 1$ for x near zero. In this case, multiplying and dividing by $\sqrt{1+x}+1$ gives the mathematically equivalent formula $f(x) = x/(\sqrt{1+x}+1)$ which, for x near zero, can be evaluated safely.

differentiation, numerical and otherwise

A completely different example is provided by numerical differentiation, discussed in Section 1.5.2. In numerical differentiation, one positively invites catastrophic cancellation since one approximates the derivative by a difference quotient, as in

$$f'(a) = \frac{f(a+h) - f(a)}{h} - hf''(\eta)/2,$$

(with η between a and $a+h$) and, for the sake of accuracy, wants to choose the *step size* h quite small, and that produces catastrophic cancellation in the numerator of the difference quotient. The table on page 55 of the textbook shows this effect very nicely. Here, too, the answer is to reformulate the problem so as to carry out the differencing symbolically, in the following way.

If the function f in question is one of the basic functions, then you have learned how to differentiate it exactly in Calculus and don't need numerical differentiation. Usually, numerical differentiation is used when all you have is a program for evaluating f at a given point. However, here, too, the answer is simple: simply **differentiate the program**, using the rules you've learned in Calculus for differentiating expressions, like sums and products and ratios and elementary functions (sin, cos, arctan, etc).

This is an opportunity to admire the **power of Calculus** which teaches us how to compute derivatives exactly, namely by symbolic means.

The idea is very simple: *treat every (relevant) variable in the program as a function of the independent variable at which a function value is being sought*, computing not just the value of the variable but also the value of its first derivative with respect to the independent variable. The following example will make this clearer.

This example concerns the standard procedure for evaluating a polynomial by **nested multiplication** or **Horner's scheme**, on page 79 of the textbook, *except* that, in contrast to the book, I follow `Matlab`'s handling of polynomials, in `polyfit`, `polyval`, `root`, etc, of listing the polynomial coefficients $\mathtt{a}(1), \ldots, \mathtt{a}(n)$ of a polynomial of degree $< n$ 'from highest to lowest', hence the script below evaluates the polynomial

$$p(x) = \mathtt{a}(1)x^{n-1} + \mathtt{a}(2)x^{n-2} + \cdots + \mathtt{a}(n-1)x + \mathtt{a}(n).$$

(Note how, in each term, the index of the coefficient and the exponent of x add up to n, the **order** of this polynomial.)

```
n = length(a);
pval = a(1);
for i=2:n
   pval = x*pval + a(i);
end
```

From this, we get a script for also computing the first derivative of this polynomial by preceding each relevant line of code with its derivative with respect to `x`, getting the following enlarged script:

```
n = length(a);
dpval = 0;
pval = a(1);
for i=2:n
   dpval = pval + x*dpval;
   pval = x*pval + a(i);
end
```

At the end, `pval` contains the value of $p(x)$ while `dpval` contains the value of $p'(x)$. Both scripts will, of course, work for a *vector* `x` provided the multiplications are done pointwise.

Of course, for larger scripts, we would use some program to generate the differentiated program. Also, it is possible to differentiate a program repeatedly.