Lecture `cond`: Condition number and limits of accuracy

September 14, 2022

Summary: Inputs and outputs of a computational problem. The concept of condition number for a problem explained in the context of a model problem. Particular focus on the problem of root-finding.

References: Chapter 1 of *Numerical Analysis* by T. Sauer, and also Lectures 12-14 in *Numerical Linear Algebra* by L. N. Trefethen and D. Bau.

Computational problems and condition number

Example problems and model problem

Here are some examples of problems encountered in the field of *Numerical Analysis*.

(i) Find the root r of a scalar function $f(x)$.

(ii) Compute the inner product $\mathbf{u}^T\mathbf{v}$, given two vectors $\mathbf{u}, \mathbf{v}$ of the same length.

(iii) Find the solution $\mathbf{x}$ of a linear system $A\mathbf{x} = \mathbf{b}$.

We develop *algorithms* to solve such problems computationally, but our focus now is on just the abstract problem and not (yet, at least) the solution process. Given a specific problem, want to identify the inputs ("independent variables") and the outputs ("dependent variables"). Moreover, we are often interested in the following question: *how sensitively does the output depend on the input*? If the answer is "very", then solving such a problem *computationally* (in an approximate way with a computer) will prove a difficult enterprise. Let us explore these ideas with a simple model.

We may *model* an abstract problem as a scalar function $h : \mathbb{R} \to \mathbb{R}$. Given an input t, we compute an output $y = h(t)$. Evaluation of h at a *fixed* input, for example evaluation of $h(0.2)$, is sometimes called a *problem instance*. Let's compare our model to the problems described in the last paragraph.

(i) The input is the function f, and the output is r. Therefore, $y = h(t)$ corresponds to "$r = h(f)$".

(ii) The input is the set $(\mathbf{u}, \mathbf{v})$, and the output is $\mathbf{u}^T\mathbf{v}$. Therefore, $y = h(t)$ corresponds to "$\mathbf{u}^T\mathbf{v} = h(\mathbf{u}, \mathbf{v})$".

(iii) The input is the set $(A, \mathbf{b})$ and the output is $\mathbf{x}$. Therefore, $y = h(t)$ corresponds to "$\mathbf{x} = h(A, \mathbf{b})$".

For most real problems h is clearly not a scalar function of scalar variable; nonetheless, the model proves useful conceptually.

Sensitivity and condition number

To address the issue of sensitivity, let's work with the model problem. Given a fixed input t, suppose the $t + \delta t$ is a nearby input. We ask how close is the output $h(t + \delta t)$, for input $t + \delta t$, to the output $h(t)$, for input t? For example, if $h(t) = c$ is a constant function, then the output is not sensitive at

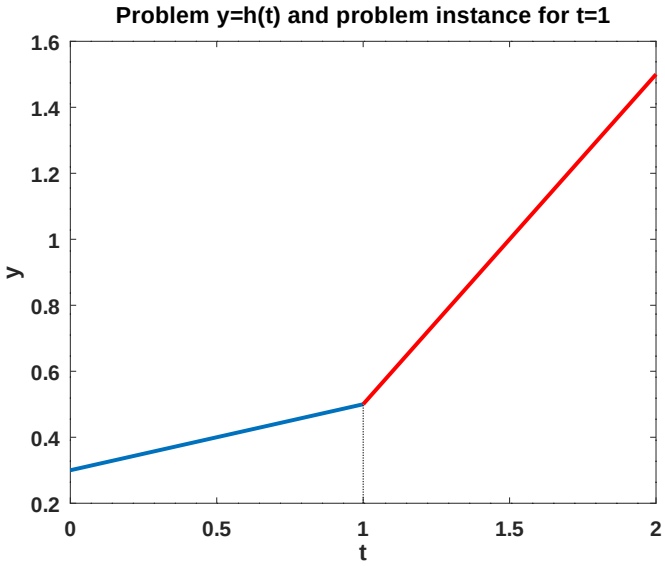

Figure 1: Computation of $h(1)$ as the problem instance.

all to changes in the input. Indeed, in this case $h(t+\delta t) = c = h(t)$, and so $h(t+\delta t) - h(t) = 0$. However, for a more general function, define $\delta y = h(t+\delta t) - h(t)$ as the change in output, so that $h(t+\delta t) = h(t) + \delta y = y + \delta y$. Then we have

$$h(t+\delta t) - h(t) = \frac{h(t+\delta t) - h(t)}{\delta t}\delta t = \frac{\delta y}{\delta t}\delta t \simeq h'(t)\delta t,$$

provided $h(t)$ is differentiable at t and δt is small. The *absolute condition number* for the problem instance $y = h(t)$ is then defined as

$$\hat{\kappa}(t) = |h'(t)|. \tag{1}$$

If $\hat{\kappa}(t)$ (that is, $|h'(t)|$) is huge, then the change in output $\simeq h'(t)\delta t$ can be large even if δt is small. For such a problem, the output depends sensitively on the input.

Often a more meaningful measure of sensitivity involves relative changes. Indeed, if $h(t) = 1$ and $\delta t = 0.1$, then an output change $\delta y = 2$ would be large; indeed, $h(t+\delta t) = 3$. However, if $h(t) = 10,000$ and $\delta t = 0.1$, then an output change $\delta y = 2$ would not seem so large. Indeed, now $h(t+\delta t) = 10,002$ only. Similar remarks pertain to the size of δt relative to t: $\delta t = -0.1$ is a large t-change if $t = 0.2$, and a small one if $t = 50$. Based on such observations, we define the *relative condition number* for the (model) problem instance $y = h(t)$ as

$$\kappa(t) = \frac{|th'(t)|}{|h(t)|} \simeq \frac{|\delta y/y|}{|\delta t/t|} = \frac{\text{relative change in output}}{\text{relative change in input}}. \tag{2}$$

The formulas (1) and (2) hold when $h(t)$ is differentiable at t. When this condition does not hold, and also to have better analogy with more complicated problems, the definitions should be modified:

$$\hat{\kappa}(t) = \max_{\delta t}\left|\frac{\delta y}{\delta t}\right|, \qquad \kappa(t) = \max_{\delta t}\left|\frac{\delta y/y}{\delta t/t}\right|,$$

where for all small (infinitesimal) input change δt we try to find the maximum possible output change. For the model problem there are only two types of small δt, either positive or negative. Figure (1)

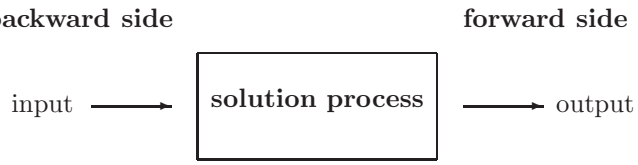

Figure 2: SCHEMATIC OF SOLVING A PROBLEM. This corresponds to $t \longrightarrow \boxed{\text{evaluate } h(t)} \longrightarrow y$ for the model problem.

depicts a situation involving the problem instance $y = h(1)$, where it is positive δt that corresponds to a larger output change. For a complicated problem there will be many ways to make a small change in the input. The definition of condition number (whether absolute or relative) involves finding the maximum output change over all possible ways to make small input changes.

Accuracy and backward stability

Figure 2 is a schematic for "solving a problem". Note the terminology **forward** (output side) and **backward** (input side) used here; it's a common one in numerical analysis. Typically, when solving a problem on a computer, we only do so approximately. For the model problem, this means we evaluate a *different function* $h_A(t)$, where, serendipitously, you may think of the subscript A here as standing for either *approximation* or *algorithm*. In the best of worlds, we have that $y_A = h_A(t) \simeq h(t) = y$, or, more precisely,

$$\Big|\frac{\overbrace{y_A - y}^{\delta y}}{y}\Big| = \Big|\frac{h_A(t) - h(t)}{h(t)}\Big| = O(\varepsilon_{\text{mach}}), \tag{3}$$

provided $y = h(t) \neq 0$, or else we would consider only the absolute error. If this condition holds, then we say the approximation (or algorithm) is *accurate*. In practice, this means

$$\Big|\frac{h_A(t) - h(t)}{h(t)}\Big| \simeq C\varepsilon_{\text{mach}} \approx 10^{-16},$$

where C is a constant (which may depend on t, the problem instance). However, the precise concept is a bit strange. For double precision, we have seen that $\varepsilon_{\text{mach}} = 2^{-52} \simeq 10^{-16}$, but the idea here is that if we computed $h_A(t)$ using a *sequence* of computers, each with a smaller machine epsilon than the previous one, then the approximation error would get closer and closer to zero. You might think of this sequence as solving the problem first using single precision ($\varepsilon_{\text{mach}} = 2^{-23}$), then double precision $\varepsilon_{\text{mach}} = 2^{-52}$), then quadruple precision ($\varepsilon_{\text{mach}} = 2^{-112}$), etc. Finally, in some case, the $\varepsilon_{\text{mach}}$ appearing in (3) might be replaced by another (presumably larger) tolerance of relevance to the problem, like 10^{-5} for example.

Let us consider some specific examples. First, let $h(t) = at$ where a is a *fixed* number. Then $h_A(t)$ would be given by

$$\begin{aligned} h_A(t) &= \text{fl}(a) \odot \text{fl}(t) \\ &= \text{fl}(\text{fl}(a) \cdot \text{fl}(t)) \\ &= \text{fl}(a(1+\alpha)t(1+\beta)) \\ &= at(1+\alpha)(1+\beta)(1+\gamma), \end{aligned}$$

where $\odot$ represents floating point multiplication and $|\alpha|, |\beta|, |\gamma| \le \varepsilon_{\text{mach}}$. Here, for example $\text{fl}(t) = t(1+\beta) \in \mathbb{F}$ is the floating point number closest to $t \in \mathbb{R}$, obeying (provided $t \neq 0$)

$$\Big|\frac{\text{fl}(t) - t}{t}\Big| = |\beta| \le \tfrac{1}{2}\varepsilon_{\text{mach}},$$

as we saw in the PDF notes `float`. The α likewise stems from the floating point representation of a, and the γ accounts for the fact that $\odot$ is not exact multiplication. For this example, if $a, t \neq 0$, then

$$\Big|\frac{h_A(t) - h(t)}{h(t)}\Big| = |(1+\alpha)(1+\beta)(1+\gamma) - 1| = |\alpha + \beta + \gamma + \alpha\beta + \alpha\gamma + \beta\gamma + \alpha\beta\gamma| = O(\varepsilon_{\text{mach}}).$$

So the approximate $\text{fl}(a) \odot \text{fl}(b)$ is accurate.

The previous example is also what's called a ***backward stable*** algorithm. Namely $h_A(t) = h(t+\delta t)$ with $|\delta t/t| = O(\varepsilon_{\text{mach}})$:

> The approximate output (of the algorithm) happens to be the *exact* output of h at a *slightly different* input.

Indeed,

$$h_A(t) = at(1+\alpha)(1+\beta)(1+\gamma) = a\big[t(1+\alpha)(1+\beta)(1+\gamma)\big] = h(t(1+\alpha)(1+\beta)(1+\gamma)) = h(t+\delta t),$$

where $\delta t = t\big[(1+\alpha)(1+\beta)(1+\gamma) - 1\big]$. For this δt, we have

$$\Big|\frac{\delta t}{t}\Big| = |(1+\alpha)(1+\beta)(1+\gamma) - 1| = O(\varepsilon_{\text{mach}}).$$

Backward stability is a nice property, and many algorithms are backwards stable. However, some are not. For example, let $\mathbf{u} = (u_1, u_2)^T$ and $\mathbf{v} = (v_1, v_2)^T$, and suppose the problem is to compute the outer product

$$\mathbf{u}\mathbf{v}^T = \begin{pmatrix} u_1v_1 & u_1v_2 \\ u_2v_1 & u_2v_2 \end{pmatrix},$$

which is a rank-1 matrix. For simplicity, assume that $u_1, u_2, v_1, v_2 \in \mathbb{F}$ are already floating point numbers and so represented exactly on the computer. Even still, the computer will compute the approximate answer

$$\text{fl}(\mathbf{u}\mathbf{v}^T) = \begin{pmatrix} u_1v_1(1+\gamma_{11}) & u_1v_2(1+\gamma_{12}) \\ u_2v_1(1+\gamma_{21}) & u_2v_2(1+\gamma_{22}) \end{pmatrix}.$$

Here the γ corrections are $O(\varepsilon_{\text{mach}})$, and they take into account the inexact floating point multiplications. Since these γ terms are independent (not correlated), this matrix need not be rank-1. In general, it is not the *exact* outer product of different input vectors. Although this computation is "stable" (the meaning of which we will not define), it is not "backward stable" which is a stronger condition.

In this class it will be useful to think of algorithms as backward stable. Therefore, when we compute $h(t)$ as $h_A(t)$ on a computer, what we get is $h(t+\delta t)$ for a small δt. In other words:

> *Perturbations* of the input are inevitable.

At the simplest level, this is true because of rounding. The number $\pi \notin \mathbb{F}$, and so we can't compute $h(\pi)$ on a computer, rather what we compute is $h(\pi + \delta t)$, where the perturbation δt is close to 10^{-16} (in double precision). Backward stability is intimately related to accuracy. Indeed, for a backward stable algorithm, we have

$$\Big|\frac{h_A(t) - h(t)}{h(t)}\Big| = \Big|\frac{h(t+\delta t) - h(t)}{h(t)}\Big| = \Big|\frac{h(t+\delta t) - h(t)}{\delta t}\frac{t}{h(t)}\frac{\delta t}{t}\Big| \simeq \kappa(t)\Big|\frac{\delta t}{t}\Big| = \kappa(t)O(\varepsilon_{\text{mach}}).$$

For a backward stable algorithm, the relative condition number $\kappa(t)$ of the problem therefore determines how well $h_A(t)$ approximates $h(t)$.

Root-finding and error magnification

The model problem is that of evaluating a function $h(t)$. But often we are *solving an equation*. For example, given $f(x)$ find an r such that $f(r) = 0$ (root condition) or, relatedly, such that $r = f(r)$ (fixed point condition). As mentioned above, we might view this problem formally as a function evaluation "$r = h(f)$". However, whereas the input $t \in \mathbb{R}$ is a real number for the model problem of evaluating $y = h(t)$, now the input f is a function, a rather different animal! Nonetheless, we explore some of the concepts raised in the context of the model problem.

First, let's assume that what we compute numerically is $r_A = h_A(f)$, where $h_A(f)$ corresponds to finding the root with a computer with inexact arithmetic, and using a particular algorithm like the bisection algorithm or fixed point iteration. Here r_A is an approximate root. Motivated by the idea of backward stability introduced above, we adopt the viewpoint that $r_A = h_A(f) = h(f + \delta f)$ is the *exact root of a perturbed function* $f + \delta f$. The simple choice $\delta f(x) = -f(r_A)$ (constant function) does the trick, for $f(x) + \delta f(x) = f(x) - f(r_A)$ clearly vanishes at r_A. Nonetheless, let's consider a more realistic choice. Indeed, even if the influence of the adopted algorithm (bisection, fixed point iteration, choice of tolerance, etc) is ignored, due to round off errors, we can't expect to evaluate the function $f(x)$ exactly. Therefore, in practice we are working with a *different function*

$$f(x) + \delta f(x) = f(x) + \varepsilon_{\text{mach}} g(x), \tag{4}$$

where the perturbation $\delta f(x) = \varepsilon_{\text{mach}} g(x)$ stems from round off errors. For simplicity, we assume that $|g(x)| \lesssim 1$ and that $g(x)$ is continuously differentiable. In reality, $g(x)$ would be discontinuous and fluctuating, even stochastic. A final assumption is that $g(r) \neq 0$, so that the perturbation $\delta f(x)$ actually does change the location of the root from r to r_A.

Even without worrying about the influence of the algorithm, we will be finding the root not of $f(x)$ but of $f(x) + \varepsilon_{\text{mach}} g(x)$, and so it's reasonable to assume that r_A is an exact root of this function. Namely that

$$f(r_A) + \varepsilon_{\text{mach}} g(r_A) = 0, \tag{5}$$

so that $|f(r_A)| = \varepsilon_{\text{mach}} |g(r_A)| \lesssim \varepsilon_{\text{mach}}$.

Forward and backward error, error magnification

Since the root r is the output, based on Fig. 2 we call $|r_A - r|$ the *forward error*, and (provided $r \neq 0$) define the

$$\text{relative forward error} = \left| \frac{r_A - r}{r} \right|.$$

Also based on Fig. 2, we introduce the

$$\text{backward error} = |f(r_A)|.$$

At least for now, we only motivate this terminology by noting that f is the input, and so appears on the backward side in Fig. 2. Based on our assumptions above,

$$\text{backward error} = O(\varepsilon_{\text{mach}}). \tag{6}$$

Indeed, in computations backward errors are often (although not always) of machine precision size. We further define the

$$\text{error magnification} = \frac{\text{relative forward error}}{\text{backward error}} = \frac{|r_A - r|/|r|}{|f(r_A)|}. \tag{7}$$

As an example, consider finding the root of $\sin x$ on $[3\pi/4, 3\pi/2]$. The input is the function $f(x) = \sin(x)$ and the output is the root $r = \pi$. Further, suppose that the solution process is the bisection algorithm (with a default 10^{-8} tolerance). From the OCTAVE command window, we find the following.

```
octave:1>  format long g; format compact
octave:2>  [rA k]= bisection(@sin,3*pi/4,3*pi/2)
Converged in 27 iterations with tol 1.0e-08
rA = 3.141592650663956
k = 27
octave:3> forerr = abs(rA-pi)
forerr = 2.925837350176153e-09
octave:4> forerr_relative = abs(rA-pi)/abs(pi)
forerr_relative = 9.313229539268548e-10
octave:5> bwderr = abs(sin(rA))
bwderr = 2.925837472640832e-09
octave:6> error_mag = forerr_relative/bwderr
error_mag = 0.3183098728605221
```

This is a small error magnification (indeed, it's really a demagnification). You might wonder why the backward error is about 10^{-9} here, whereas based on our analysis we found (6). The reason is that our analysis leading to (6) ignored the algorithm, and in particular the choice of tolerance. If we choose a tighter tolerance, we get closer to the result promised in (6).

```
octave:7>  [rA k]= bisection(@sin,3*pi/4,3*pi/2,1e-15)
Converged in 50 iterations with tol 1.7e-15
rA = 3.141592653589794
k = 50
octave:8> forerr_relative = abs(rA-pi)/abs(pi)
forerr_relative = 2.827159716856459e-16
octave:9> bwderr = abs(sin(rA))
bwderr = 7.657137397853899e-16
octave:10> error_mag = forerr_relative/bwderr
error_mag = 0.3692188829795375
```

In any case, the error magnification is comparable.

Explicit formula for error magnification

Assume that r is a *simple root* of $f(x)$; that is, $f(r) = 0$, but $f'(r) \neq 0$. Subject to this assumption (and those above) we now work out an explicit formula for the error magnification (7). Since $r_A = r + \delta r$, by first-order Taylor expansion

$$0 = f(r+\delta r) + \varepsilon_{\text{mach}} g(r+\delta r) = f(r) + f'(r)\delta r + \varepsilon_{\text{mach}} g(r) + \varepsilon_{\text{mach}} g'(r)\delta r + O((\delta r)^2). \tag{8}$$

But $f(r) = 0$; therefore,

$$\delta r \simeq \frac{-\varepsilon_{\text{mach}} g(r)}{f'(r) + \varepsilon_{\text{mach}} g'(r)} \simeq -\varepsilon_{\text{mach}} \frac{g(r)}{f'(r)},$$

where the second approximation assumes $|\varepsilon_{\text{mach}} g(r)| \ll |f'(r)| \neq 0$. Therefore, we have what T. Sauer calls the *sensitivity formula*

$$\delta r \simeq -\varepsilon_{\text{mach}} \frac{g(r)}{f'(r)} \implies |\delta r|/|r| \simeq \varepsilon_{\text{mach}} \Big| \frac{g(r)}{r f'(r)} \Big|.$$

The backward error is $|f(r_A)|$; from (8) which says $0 = f(r_A) + \varepsilon_{\text{mach}} g(r_A)$ this is equivalent to $\varepsilon_{\text{mach}} |g(r_A)|$. Therefore, from (7) we arrive at the formula

$$\text{error magnification} = \Big| \frac{g(r)}{g(r_A)} \frac{1}{r f'(r)} \Big| \simeq \Big| \frac{1}{r f'(r)} \Big|. \tag{9}$$

Let's consider the example $f(x) = (x-1)(x-2)(x-3)$ perturbed by $\delta f(x) = \epsilon x^4$, where $\epsilon = 10^{-7}$ rather than $\varepsilon_{\text{mach}}$ for the purpose of illustration. Then the peturbation of the root $r = 2$ is

$$\delta r \simeq -\epsilon \frac{g(r)}{f'(r)} = -10^{-7} 2^4/(-1) = 1.6 \times 10^{-6}.$$

So we expect

$$r_A = r + \delta r = 2.0000016, \qquad \text{error magnification} = \Big| \frac{1}{2 f'(2)} \Big| = \tfrac{1}{2}.$$

Using OCTAVE's roots function, we may compute the roots of the perturbed polynomial $10^{-7}x^4 + (x-1)(x-2)(x-3) = 10^{-7}x^4 + x^3 - 6x^2 + 11x - 6$ (a quartic, rather than cubic one, so we will have one more root):

```
octave:20> roots([1 -6 11 -6])
ans =
  3.000000000000002
  1.999999999999998
  1
octave:21> roots([1e-7 1 -6 11 -6])
ans =
  -10000005.9999975
  2.999995949998947
  2.000001600002346
  0.9999999500010948
octave:22> rA = 2.000001600002346
rA = 2.000001600002346
octave:23> error_mag = abs((rA-2)/2)/abs((rA-1)*(rA-2)*(rA-3))
error_mag = 0.50000000000128
```

Clearly, the third root here is very close to the one we predicted.

Relation to condition number

Since we have taken the viewpoint that $r_A = h_A(f) = h(f + \delta f)$ is the exact root of a perturbed function $f + \delta f$, the relative forward error

$$\Big|\frac{r_A - r}{r}\Big| = \Big|\frac{h(f+\delta f) - h(f)}{h(f)}\Big|$$

is the same as the relative change in the output. How would we measure the change in the input, i.e. somehow the size of δf.

Again here the input is not a scalar number t, rather the function f, a rather different animal. Whereas $|t|$ measures the size of t, to measure that size of f we need a *function norm*, for example

$$\|f\|_1 = \int_a^b |f(x)|dx \quad \text{or} \quad \|f\|_2 = \sqrt{\int_a^b |f(x)|^2 dx} \quad \text{or} \quad \|f\|_\infty = \max_{x\in[a,b]} |f(x)|,$$

where $[a, b]$ is an appropriate interval. The second of these is a generalization of the Euclidean norm used for vectors. Let $\|\cdot\|$ be some function norm (perhaps one of the three just given), then $t + \delta$ corresponds to $f(x) + \delta f(x)$, whereas $y + \delta y$ corresponds to $r + \delta r$. So by analogy the relative condition number would be

$$\kappa(f) = \text{maximum possible}\Big|\frac{\text{relative change in output}}{\text{relative change in input}}\Big| = \max_{\delta f} \frac{|\delta r|/|r|}{\|\delta f\|/\|f\|}.$$

If we assume $\|\delta f\|/\|f\| = \varepsilon_{\text{mach}}$ and that $g(r) = 1$ corresponding to the maximum, then

$$\kappa(f) = \Big|\frac{1}{r f'(r)}\Big|,$$

the same as the error magnification.

Ill-conditioned problems

A problem with a large condition number, one that is very sensitive to small changes in input, is called poorly conditioned or *ill-conditioned*. As seen above and in Fig. 3, a small $f'(r)$ corresponds to poor conditioning. As an extreme example, consider the function $f(x) = (x - \frac{4}{3})^3$, with $r = 4/3$ its only root. This root has multiplicity 3, and in particular $f'(4/3) = 0$, so the error magnification

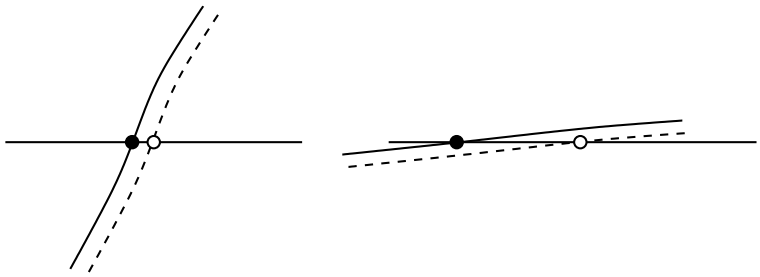

Figure 3: Perturbations of functions. The dotted line is the perturbed function. Clearly if $f'(r) \simeq 0$ the perturbation more drastically changes the location of the root.

(9) is formally $+\infty$. Figure 4 illustrates the trouble: the relative forward error $|r_A - \frac{4}{3}|/|\frac{4}{3}|$ can be large, despite the backward error $|(r_A - \frac{4}{3})^3|$ being small. Indeed, assume $r_A = 4/3 \pm 10^{-p}$, so that $|r_A - \frac{4}{3}|/|\frac{4}{3}| = \frac{3}{4}10^{-p}$, while $|f(r_A)| = \pm 10^{-3p}$. The block dotted line in Fig. 4 corresponds to $p = 1$ and $r_A = \frac{4}{3} - 10^{-1}$, with a backward error $|(r_A - \frac{4}{3})^3| = 10^{-3}$.

Let us consider the problem of finding the root $r = \frac{4}{3}$. To make the problem seem more real-world, let's pretend we don't know the function in it's factored form. After all, root-finding techniques are important precisely when we don't know the root. Therefore, assume that we know the function in its expanded form

$$f(x) = x^3 - 4x^2 + \tfrac{16}{3}x - \tfrac{64}{27}.$$

We often encounter the problem of finding the root of a polynomial without being able to factor it explicitly (although cubics can always be factorized). Near the root $f(x) \simeq 0$, so the sum of these terms is small and prone to cancellation errors. Figure 5 shows a zoom-in of this function near the root, and we see that round-off errors result in a perturbed function. If we use the bisection algorithm to find the root, then we are finding the root of this perturbed function. Here's what we get

```
octave:28> [rA k] = bisection(@(x)(x^3-4*x^2 + 16*x/3-64/27),1,2,1e-11)
rA = 1.333328247070312
k = 16
octave:29> forwarderr_relative = abs((rA-4/3)/(4/3))
forwarderr_relative = 3.814697265569489e-06
octave:30> backerr = rA^3-4*rA^2 + 16*rA/3-64/27
backerr = 0
```

Despite computing a zero (!) backward error, we have a large forward error, one orders of magnitude worse than the chosen tolerance of 10^{-11}.

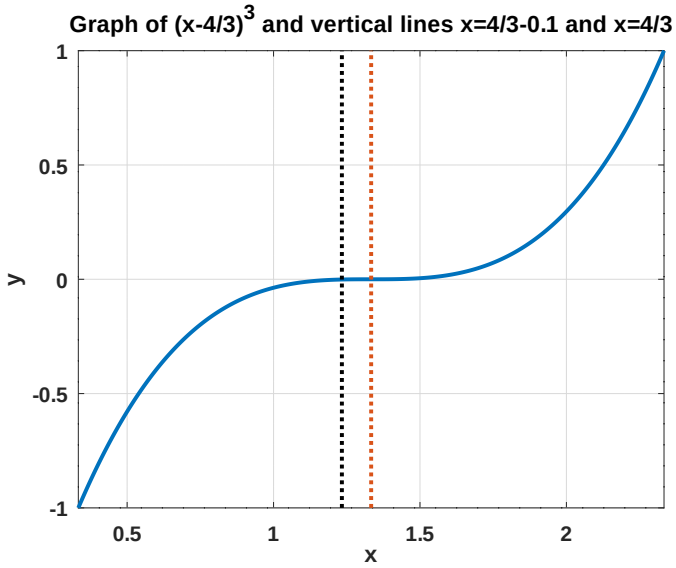

Figure 4: Simple cubic polynomial.

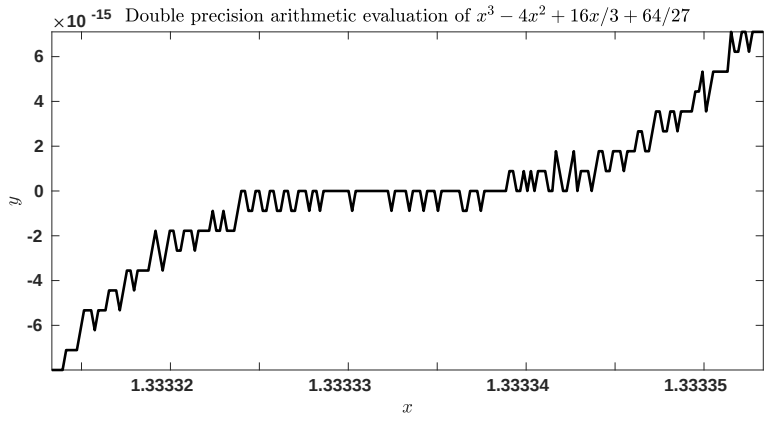

Figure 5: The choppy appearance stems from round-off cancellation errors.