Chapter 1

Errors and Arithmetic

What better place to start a book than with error! We need to know how errors arise, how they are propagated in calculations, and how to measure and bound errors.

1.1 Sources of Error

Suppose an engineer wants to study the stresses in a bridge. The study would begin by gathering some data, including the lengths and angles for the girders and wires and the material properties for each component. There is some **measurement error**, though, since no measuring device gives full precision. Therefore, the measurements would typically be recorded as a value plus or minus an uncertainty.

The engineer would then need to model the stresses on the bridge. The bridge might be approximated by a "finite element model", for example, and this is an additional source of error. Simplifying assumptions might be made; for example, we might assume that the material in each girder is homogeneous. **Modeling error** is the result of the difference between the true bridge and our computable model.

Now we have a mathematical model and we need to compute the stresses. If the model is large or nonlinear, then a numerical analyst might develop an algorithm that computes the solution as

$$\lim_{n\to\infty} G(n)$$

where, for example, $G(n)$ might be the result of n iterations. In general, we can't take this limit on a computer, so we might decide that $G(150)$ is good enough. This introduces **truncation error**.

Finally, we implement the algorithm and run it on our favorite computer. This introduces additional error, since we don't compute with real numbers but with **finite-precision** numbers: a fixed number of digits are carried in the computation. The effect of this is **roundoff error**.

Therefore, the results obtained for the stresses on the bridge are contaminated by these four types of error: measurement error, modeling error, truncation error,

and roundoff error. It is important to note that no **mistakes** were made:

- The engineer did not misread the measurement device.
- The model was a good approximation to the true bridge.
- The programmer did not type the value of π incorrectly.
- The computer worked flawlessly.

But at the end of the process, the engineer needs to ask what the computed solution has to do with the stresses on the bridge!

1.2 Computational Science and Scientific Computing

In order to answer the question posed at the end of the previous section, we require several types of expertise. We use **science** and **engineering** to formulate the problem and determine what data is needed. We use **mathematics** and **statistics** to design the model. We use **numerical analysis** to design and analyze the algorithms, develop mathematical software, and answer questions about how accurate the final answer is. Therefore, our project could easily involve an interdisciplinary team of four or more experts; see Figure 1.1. Often, though, if the model is more or less routine, one person might fill all roles.

A **computational scientist** is a team member whose focus is on **scientific computing**: intelligently using mathematical software to analyze mathematical models. To do this requires a basic understanding of how computers do arithmetic.

1.3 Computer Arithmetic

Computers use **binary arithmetic**, representing each number as a **binary number**, a finite sum of integer powers of 2. Some numbers can be represented exactly, but others, such as 1/10, 1/100, 1/1000, ..., cannot. For example, in binary,

$$2.125 = 2^1 + 2^{-3}$$

has an exact representation, but

$$3.1 \approx 2^1 + 2^0 + 2^{-4} + 2^{-5} + 2^{-8} + \cdots$$

does not. And, of course, there are numbers like π that have no finite representation in either our usual decimal number system or in binary.

Computers use two formats for numbers. **Fixed-point** numbers are used to store integers. Typically, each number is stored in a **computer word** consisting of 32 binary digits (**bits**) with values 0 and 1. Therefore, at most 2^{32} different numbers can be stored. If we allow for negative numbers, then we can, for example, represent integers in the range $-2^{31} \leq x \leq 2^{31} - 1$. Since $2^{31} \approx 2.1 \times 10^9$, the range for fixed-point numbers is too limited for scientific computing. Therefore, they are used mostly for indices and counters.

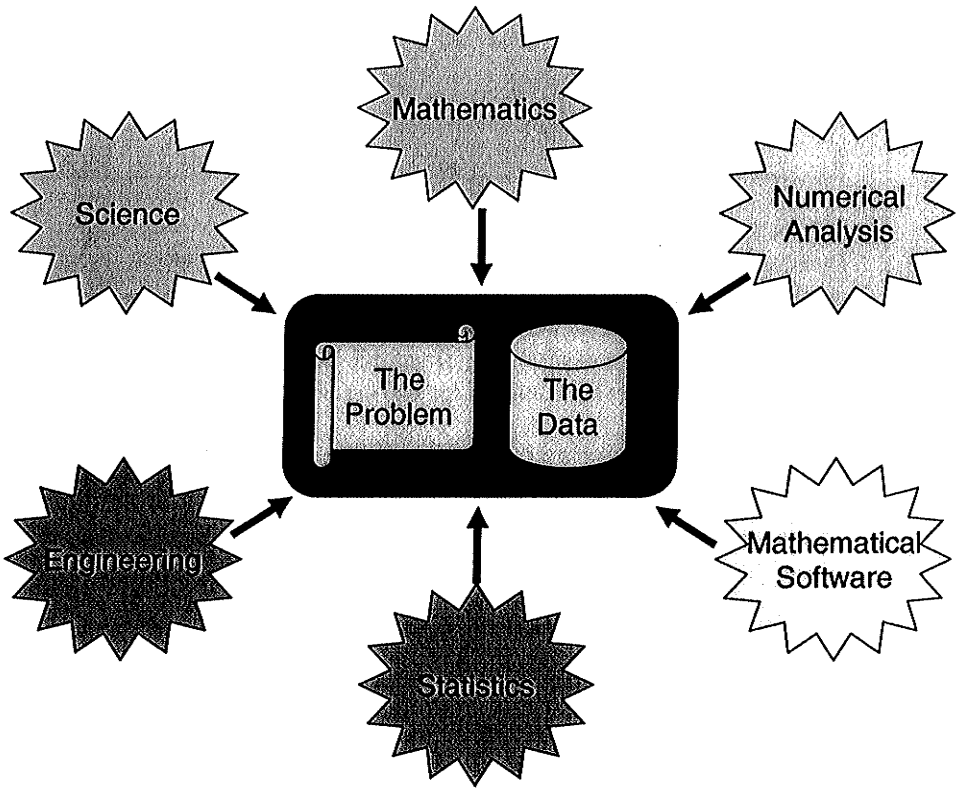


Figure 1.1. *Computational science involves knowledge from many disciplines.*

An alternative to fixed-point numbers is **floating-point** numbers, which approximate real numbers. We'll discuss features of the most common floating point number system, the IEEE Standard Floating Point Arithmetic.

The format for a floating point number is

$$x = \pm z \times 2^p$$

where z is called the **mantissa** or **significand** This representation is not unique; for example,

$$1 \times 2^2 = 4 \times 2^0 = 8 \times 2^{-1}.$$

Therefore we make the rule that if $x \neq 0$, we **normalize** so that $1 \leq z < 2$, chosing the first of the three alternatives in the example.

POINTER 1.1. Modeling the Error.
Developing a realistic understanding of the errors in the data is often the most challenging part of scientific computing. If you are solving a spectroscopy problem, for example, ideally you would first want to take a sample for which the composition is known and obtain several sets of sample data from the spectrometer. Using that data, you could develop a model for the error and see how your algorithms behave.

POINTER 1.2. Matrix and Vector Notation.
Throughout this book we use the following notational conventions:

- All vectors are column vectors.
- Matrices are denoted by boldface upper case letters; vectors are boldface lower case.
- The elements of a matrix or vector are denoted by subscripted values: the element of $\boldsymbol{A}$ in row i and column j is a_{ij} or $A(i,j)$.
- The elements of matrices and vectors can be real or complex numbers.
- $\boldsymbol{I}$ is the identity matrix and $\boldsymbol{e}_i$ is the ith unit vector, the ith column of $\boldsymbol{I}$.
- $\boldsymbol{B} = \boldsymbol{A}^T$ means that $\boldsymbol{B}$ is the transpose of $\boldsymbol{A}$: $b_{ij} = a_{ji}$.
- $\boldsymbol{B} = \boldsymbol{A}^*$ means that $\boldsymbol{B}$ is the complex conjugate transpose of $\boldsymbol{A}$: $b_{ij} = \bar{a}_{ji}$, where the bar denotes complex conjugate. If $\boldsymbol{A}$ is real, then $\boldsymbol{A}^* = \boldsymbol{A}^T$.
- We'll use MATLAB notation when convenient. For example, $\boldsymbol{A}(i:j, k:\ell)$ denotes the submatrix of $\boldsymbol{A}$ with row entries between i and j and column entries between k and ℓ (inclusive), and $\boldsymbol{A}(:,5)$ denotes column 5 of the matrix $\boldsymbol{A}$.

To fit a floating point number in a single word, we need to limit the number of digits in the mantissa and the exponent. For these **single-precision** numbers, 24 digits are used to represent the mantissa, and the exponent is restricted to the range $-126 \leq p \leq 127$. This allows us to represent numbers as close to zero as 1.18×10^{-38} and as far as 1.70×10^{38}, a considerably larger range than for fixed-point.

If this range is not large enough, or if 24 digits of precision are not enough, we turn to **double-precision** numbers, stored in two words, using 53 digits for the mantissa, with an exponent $-1022 \leq p \leq 1023$. This allows us to represent numbers as close to zero as 2.23×10^{-308} and as large as 8.98×10^{307}.

If we perform a computation in which the exponent of the answer is outside

the allowed range, we have a more or less serious error.

- If the exponent is too big, then we cannot store the answer, and our computation has produced an **overflow** error. The answer is set to a special representation called `Inf` or `-Inf` to signal an error.
- If the exponent is too small, then the computation produced an **underflow**, and the default is to set the answer to zero.
- If we divide zero by zero, then the answer is set to a code indicating **not-a-number**, `NaN`.

In double precision, at most 2^{64} different numbers can be represented (including `NaN` and $\pm$`Inf`) so any other number must be approximated by one of the representable numbers. For example, numbers in the range $1 + 2^{-53} \leq x < 1 + (2^{-52} + 2^{-53})$ might be rounded to $x_m = 1 + 2^{-52}$, which can be represented exactly. This introduces a very small error: the **absolute error** in the representation is

$$|x - x_m| \leq 2^{-53}.$$

Similarly, numbers in the range $1024 + 2^{-43} \leq x < 1024 + (2^{-42} + 2^{-43})$ might be rounded to $x_m = 1024 + 2^{-42}$, with absolute error bound 2^{-43}, which is 1024 times bigger than the bound for numbers near 1. In each case, though, the **relative error**

$$\frac{|x - x_m|}{|x|}$$

is bounded by 2^{-53} when 53 digits are used for the mantissa.

Let's stop and consider the difference between the fixed point number system and the floating point number system.

CHALLENGE 1.1. For each machine-representable number `r`, define `f(r)` to be the next larger machine-representable number. Consider the following statements:

(a) For fixed point (integer) arithmetic, the distance between `r` and `f(r)` is constant.
(b) For floating point arithmetic, the relative distance `|(f(r)-r)/r|` is constant (for `r` $\neq 0$).

Are the statements true or false? Give examples or counterexamples to explain your reasoning.

This brings us to a very important parameter that characterizes machine precision: **machine epsilon** ϵ_m is defined as the gap between 1 and the next bigger number; for double precision, $\epsilon_m = 2^{-52}$. The relative error in rounding a number

POINTER 1.3. Floating point precision.
By default, MATLAB computes using double-precision floating point numbers, and that is what we use in all of our computations.

POINTER 1.4. IEEE Standard Floating Point Arithmetic.
Up until the mid-1980s, each computer manufacturer had a different representation for floating point numbers and different rules for rounding the answer to a computation. Therefore, a program written for one machine would not compute the same answers on other machines.
The situation improved somewhat with the introduction in 1987 of the **IEEE standard floating point arithmetic**, now used by virtually all computers.
For more detailed information on floating-point computer arithmetic, see the excellent book by Overton [102]. In particular, a careful reader might note that we seem to be storing 33 bits of floating-point information in a 32 bit word, and the trick that enables us to avoid storing the leading bit in the mantissa is explained in that book.

---

is bounded by $\epsilon_m/2$. Note that ϵ_m is much larger than the *smallest* positive number that the machine can store exactly!

The next two challenges provide some practice with floating point number systems, first in base 10 and then in base 2.

CHALLENGE 1.2. Assume you have a base 10 computer that stores floating point numbers using a 5 digit normalized mantissa (x.xxxx), a 4 digit exponent, and a sign for each.

(a) For this machine, what is machine epsilon?

(b) What is the smallest positive number that can be represented exactly in this machine?

CHALLENGE 1.3. Assume you have a base 2 computer that stores floating point numbers using a 6 digit (bit) normalized mantissa (x.xxxxx), a 4 digit exponent, and a sign for each.

POINTER 1.5. Internal Representation vs. Printed Numbers.
In interpreting MATLAB results, remember that if a number x is displayed as 1.0000, it is not necessarily equal to 1. All you know is that if you round the number to the nearest decimal number with 5 significant digits, you get 1. If you want to see whether it equals 1 exactly, then display $x - 1$. Alternatively, typing `format hex` changes the display to the internal machine representation.

(a) For this machine, what is machine epsilon?

(b) What is the smallest positive number that can be represented exactly in this machine?

(c) What mantissa and exponent are stored for the value 1/10? Hint:

$$\frac{1}{10} = \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \ldots.$$

We'll experiment a bit with the oddities of floating-point arithmetic.

CHALLENGE 1.4.

(a) Consider the following code fragment:

```
x = 1;
delta = 1 / 2^(53);
for j=1:2^(20),
  x = x + delta;
end
```

Using mathematical reasoning, we expect the final value of x to be $1 + 2^{-33}$. Use your knowledge of floating-point arithmetic to predict what it actually is. Verify by running the code. Explain the result.

(b) Using mathematical reasoning, we know that for any positive number x, $2x$ is a number greater than x. Is this true of floating point numbers? Run this code fragment and explain your result:

```
x = 1;
twox = 2*x;
while (twox > x)
  x = twox;
  twox = 2*x;
end
```

(c) Using mathematical reasoning, we know that addition and multiplication are commutative

$$x + y = y + x, \quad xy = yx,$$

and associative

$$((x + y) + z) = x + (y + z), \quad (xy)z = x(yz)$$

and that multiplication distributes over addition:

$$x(y + z) = xy + xz.$$

Give examples of floating point numbers `x`, `y`, and `z` for which addition is not associative. Find a similar example for multiplication, and a third example showing that floating point multiplication does not always distribute over addition. (Avoid expressions that evaluate to $\pm$`Inf` or `NaN`, even though examples can be constructed using these values.)

(d) Write a MATLAB expression that gives an answer of `NaN` and one that gives `-Inf`.

(e) Given a floating point number `x`, what is the distance between `x` and the next larger floating point number? (Answer this either by analyzing the machine representation scheme or by experimenting in MATLAB.) Approximate your answer as a multiple of ϵ_m.

Our experiments have shown us that

- Unlike the fixed-point numbers, the numbers that we can store in floating point representation are not equally spaced.
- When we do a floating point operation (addition, subtraction, multiplication, or division), we get either exactly the right answer, or a rounded version of it, or `NaN`, or an indication of overflow.
- The main advantage of floating-point representation is the wide range of values that can be approximated with it.

Because of the errors introduced in floating point computation, small changes in the way the data is stored can make large changes in the answer, as we see in the next challenge.

CHALLENGE 1.5. Suppose we solve the linear system

$$\boldsymbol{A}\boldsymbol{x} \equiv \begin{bmatrix} 2.00 & 1.00 \\ 1.99 & 1.00 \end{bmatrix} \boldsymbol{x} = \begin{bmatrix} 1.00 \\ -1.00 \end{bmatrix} \equiv \boldsymbol{b}.$$

Now suppose that the units for $b(1)$ are centimeters, while the units for $b(2)$ are meters. If we convert the problem to meters we obtain the linear system

$$Cz \equiv \begin{bmatrix} 0.02 & 0.01 \\ 1.99 & 1.00 \end{bmatrix} z = \begin{bmatrix} 0.01 \\ -1.00 \end{bmatrix} \equiv d.$$

Solve both systems in MATLAB using the backslash operator and explain why x is not exactly equal to z.

If all data were exact and if computers did their arithmetic using real numbers, then mathematical analysis would tell us all we need to know. Because of uncertainty in data and use of the floating-point number system, we need to understand how errors propagate through computation.

1.4 How Errors Propagate

If answers to our calculations were always represented as the floating-point number closest to the true answer, then designing accurate algorithms would be easy. Unfortunately, the computed answer tends to drift away from the true answer due to accumulation of rounding error. This happens whenever the number of digits is limited, so for convenience, we'll look at examples in decimal arithmetic rather than binary.

Suppose we have measured two values:

$$a = 2.003 \pm 0.001,$$
$$b = 2.000 \pm 0.001\,.$$

The absolute error in each measurement is bounded by 0.001, and the relative error in the second is at most $.001/1.999 \approx 0.05\%$. The relative error in the first is also about 0.05%.

What can we conclude about the difference between the two values? The true difference is at most $2.004 - 1.999 = .005$ and at least $2.002 - 2.001 = .001$. We obtain the same information by taking the difference between the measurements and adding the uncertainties: $a - b = 0.003 \pm 0.002$.

When we subtracted the numbers, our bounds on the absolute errors were added. What happened to our bound on the relative error? If the true answer is 0.001, the relative error would be $(0.003 - 0.001)/0.001 = 200\%$. This enormous magnification of the relative error bound resulted from **catastrophic cancellation** of the significant digits in the two measurements: although the measured values have 4 significant digits, the difference has only 1. Any subsequent computation involving this difference propagates the error.

We could generalize this example to prove a theorem: when adding or subtracting, the bounds on absolute error add.

What about multiplication and division?

CHALLENGE 1.6. Suppose x and y are true (nonzero) values and $\tilde{x}$ and $\tilde{y}$ are our approximations to them. Let's express the errors as

$$\tilde{x} = x(1 - r),$$
$$\tilde{y} = y(1 - s),$$

(a) Show that the relative error in $\tilde{x}$ is $|r|$ and the relative error in $\tilde{y}$ is $|s|$.

(b) Show that we can bound the relative error in $\tilde{x}\tilde{y}$ as an approximation to xy by

$$\left|\frac{\tilde{x}\tilde{y} - xy}{xy}\right| \leq |r| + |s| + |rs|.$$

Since we expect the relative errors r and s to be much less than 1, the quantity $|rs|$ is expected to be very small compared to $|r|$ and $|s|$. Therefore, when multiplying or dividing, the bounds on relative errors (approximately) add.

Notice that these statements about errors after arithmetic operations assume that the computed solution is stored exactly; additional error may result from rounding to the nearest floating-point number.

CHALLENGE 1.7. Consider the following MATLAB code:

```
x = .1;
sum = 0;
for i=1:100
  sum = sum + x;
end
```

Is the final value of `sum` equal to 10? If not, why not?

In computations where error build-up can occur, it is good to rearrange the computation to avoid cancellation whenever possible. We'll consider a familiar example, finding the roots of a quadratic polynomial, next.

1.5 Mini Case Study: Avoiding Catastrophic Cancellation

Suppose we are asked to find the roots of the polynomial

$$x^2 - 56x + 1 = 0.$$

The usual formula, which you may have learned in an algebra class, computes

$$\begin{aligned} x_1 &= 28+\sqrt{783} \approx 28+27.982 = 55.982 \quad (\pm 0.0005), \\ x_2 &= 28-\sqrt{783} \approx 28-27.982 = 0.018 \quad (\pm 0.0005). \end{aligned}$$

The error arose from approximating $\sqrt{783}$ by its correctly rounded value, 27.982. The absolute error bounds are the same, but the relative error bounds are about 10^{-5} for x_1 and 0.02 for x_2 – vastly different!

The problem, of course, was catastrophic cancellation in the computation of x_2, and it is easy to convince yourself that for any quadratic with real roots, the quadratic formula causes some cancellation during the computation of one of the roots.

We can avoid this cancellation by using other facts about quadratic equations and about square roots. We consider three possibilities.

- Use an alternate formula. The product of the two roots equals the constant term in the polynomial, so $x_1 x_2 = 1$. If we compute

  $$x_2 = \frac{1}{x_1},$$

  then our relative error is bounded by 10^{-5}, the relative error in our value for x_1, so we obtain

  $$x_2 \approx .0178629(\pm 2 \times 10^{-7}),$$

  accurate to the same relative error.

- Rewrite the formula. Notice that

  $$x_2 = 28-\sqrt{783} = \sqrt{784}-\sqrt{783}.$$

  Let's derive a better formula for the difference of these square roots:

  $$\begin{aligned} \sqrt{z+e}-\sqrt{z} &= (\sqrt{z+e}-\sqrt{z})\frac{\sqrt{z+e}+\sqrt{z}}{\sqrt{z+e}+\sqrt{z}} \\ &= \frac{z+e-z}{\sqrt{z+e}+\sqrt{z}} \\ &= \frac{e}{\sqrt{z+e}+\sqrt{z}}. \end{aligned}$$

  Therefore, letting $z = 783$ and $e = 1$, we calculate

  $$x_2 = \frac{1}{28+\sqrt{783}},$$

  giving the same result as above but from a different approach.

- Use Taylor series. Let $f(x) = \sqrt{x}$. Then

  $$f(z+e) - f(z) = f'(z)e + \frac{1}{2}f''(z)e^2 + \cdots,$$

  so we can approximate the difference by $f'(z)e = 1/(2\sqrt{783})$.

POINTER 1.6. Symbolic Computation.
Some folks claim that the pitfalls in floating-point arithmetic are best avoided by avoiding floating-point arithmetic altogether, and instead using **symbolic computation** systems such as MAPLE (`http://www.maplesoft.com`) (included in MATLAB) or MATHEMATICA (`www.wolfram.com`). These systems are incredibly useful, but eventually they produce a formula that needs to be evaluated using arithmetic. These systems have pitfalls of their own: the computation can use a tremendous amount of time and storage, and they can produce formulas that lead to unnecessarily high relative and absolute errors.

CHALLENGE 1.8. Write a MATLAB function that computes the two roots of a quadratic polynomial with good relative precision.

1.6 How Errors Are Measured

Error analysis determines the cumulative effects of error. We have been using forward error analysis, but there are alternatives, including backward error analysis.

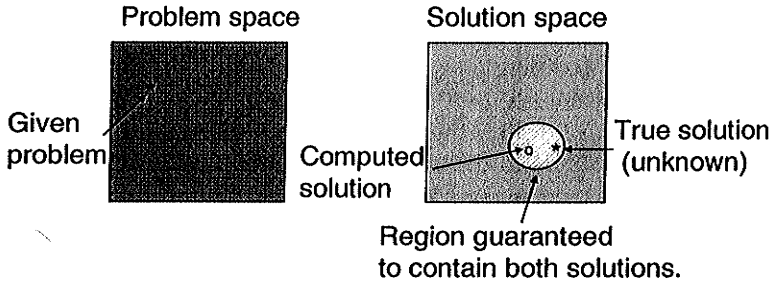
- In **forward error analysis**, we find an estimate for the answer and bounds on the error. Schematically, we see in the top of Figure 1.2 that we have a space of all possible problems and a space of their solutions. We are given a true problem whose true solution is unknown. We compute a solution, and report that solution along with a bound on the distance between the computed solution and the true solution. For example, we might compute the answer 5.348 and determine that the true answer is $5.348 \pm .001$. Or, for a vector solution, we might report that $\|\boldsymbol{x}_c - \boldsymbol{x}_{true}\| \leq 10^{-5}$, where $\boldsymbol{x}_c$ is the computed solution and $\boldsymbol{x}_{true}$ is the true solution.

- In **backward error analysis**, we again are given a true problem whose true solution is unknown. We compute a solution, and report that solution along with a bound on the difference between the problem we solved and the true problem. This is illustrated in the bottom of Figure 1.2.

Let's determine forward and backward error bounds for a simple problem.

CHALLENGE 1.9. Suppose the sides of a rectangle have lengths $3.2 \pm .005$ and $4.5 \pm .005$. Consider approximating the area of the rectangle by $A = 14$.

Forward Error Analysis:
Report the computed solution and a region known to contain both the true and computed solutions.



Backward Error Analysis:
Report the computed solution and a region known to contain both the given and solved problems.

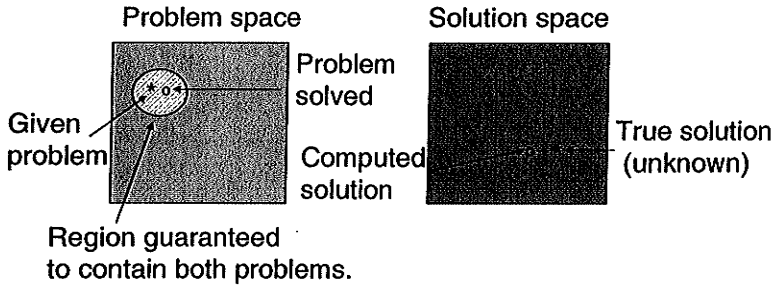


Figure 1.2. *In forward error analysis, we find bounds on the distance between the computed solution and the true solution. In backward error analysis, we find bounds on the distance between the problem we solved and the problem we wanted to solve.*

(a) Give a forward error bound for A as an approximation to the true area.

(b) Give a backward error bound.

It might be hard to imagine a situation in which backward error analysis provides any useful information, but think back to our bridge. Suppose we compute a solution to a problem for which the measurements differ from our measurements by 10^{-5}. If the error bounds in our measurements are greater than 10^{-5}, then we may have computed the stresses for the true bridge! In any case, the solution we computed is as reasonable as one for any other problem in the uncertainty intervals, so we can be quite satisfied with the outcome. In general, backward error statements are quite useful when the data has uncertainty.

Backward error estimates also tend to be less pessimistic than forward error estimates, since they don't involve taking a worst-case bound after every computation. Backward error estimates are usually derived at the end of the algorithm. For example, if we compute an approximate solution $\boldsymbol{x}_c$ to a linear system of equations

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

then we can test how good it is by evaluating the **residual**

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_c.$$

If $\boldsymbol{x}_c$ equals the true solution, then $\boldsymbol{r} = \boldsymbol{0}$; if it is a good approximation, then we expect $\boldsymbol{r} \approx \boldsymbol{0}$. In any case, we know that our computed solution $\boldsymbol{x}_c$ is the exact solution to the nearby problem

$$\boldsymbol{A}\boldsymbol{x}_c = \boldsymbol{b} - \boldsymbol{r},$$

so $\|\boldsymbol{r}\|$ gives us a backward error bound.

Here are three examples to provide some experience in computing error bounds.

CHALLENGE 1.10. Bound the backward error in approximating the solution to

$$\begin{bmatrix} 2 & 1 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5.244 \\ 21.357 \end{bmatrix} \text{ by } \boldsymbol{x}_c = \begin{bmatrix} 1 \\ 3 \end{bmatrix}.$$

CHALLENGE 1.11. Suppose that you have measured the length of the side of a cube as $(3.00 \pm .005)$ meters. Give an estimate of the volume of the cube and a (good) bound on the absolute error in your estimate.

1.7 Conditioning and Stability

It is important to distinguish between bad problems and bad algorithms.

We say that a problem is **well-conditioned** if small changes in the data always make small changes in the solution; otherwise it is **ill-conditioned**. Similarly, an algorithm is **stable** if it always produces the solution to a nearby problem, and **unstable** otherwise.[1]

To illustrate these ideas, consider the linear system of equations

$$\begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

where $\delta < \epsilon_m/2$. If we solve this system using Gauss elimination without pivoting, we compute

$$\begin{bmatrix} \delta & 1 \\ 0 & -1/\delta \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1/\delta \end{bmatrix}$$

so

$$x_2 = 1, \;\; x_1 = 0.$$

The true solution is

$$x_{true} = \begin{bmatrix} -\frac{1}{1-\delta} \\ \frac{1}{1-\delta} \end{bmatrix},$$

so our answer is very bad. The problem is well-conditioned, though; we can see this graphically in Figure 1.3, since small changes in any of the coefficients of the two lines move the intersection point by just a little. Therefore, Gauss elimination without pivoting must be an unstable algorithm. If we use pivoting, our answer improves: the linear system is rewritten as

$$\begin{bmatrix} 1 & 1 \\ \delta & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

so the elimination gives us

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

from which we determine that

$$x_2 = 1, \;\; x_1 = -1\,.$$

This is quite close to the true solution.

Consider a second example with 3-digit decimal arithmetic:

$$\begin{bmatrix} 0.661 & 0.991 \\ 0.500 & 0.750 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.330 \\ 0.250 \end{bmatrix} \tag{1.1}$$

[1] Actually, for historical reasons, well-conditioned problems are sometimes called stable in some areas of scientific computing, but it is best to use the term well-conditioned to avoid confusion.

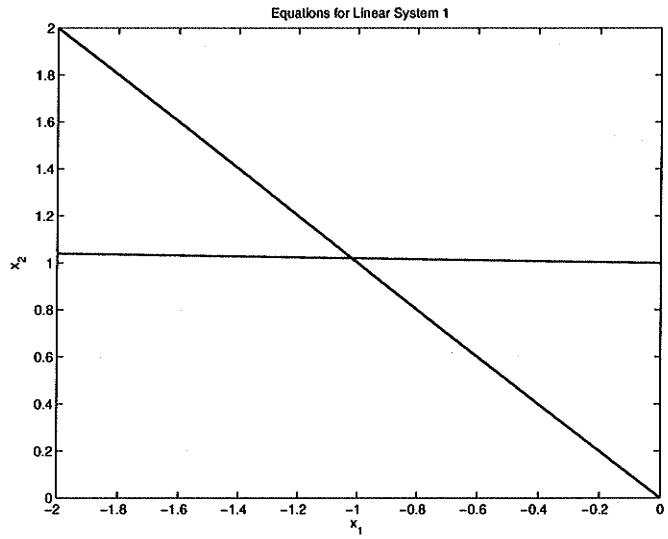


Figure 1.3. *Plot of a well-conditioned system of linear equations. Small changes in the data move the intersection of the two lines by a small amount.*

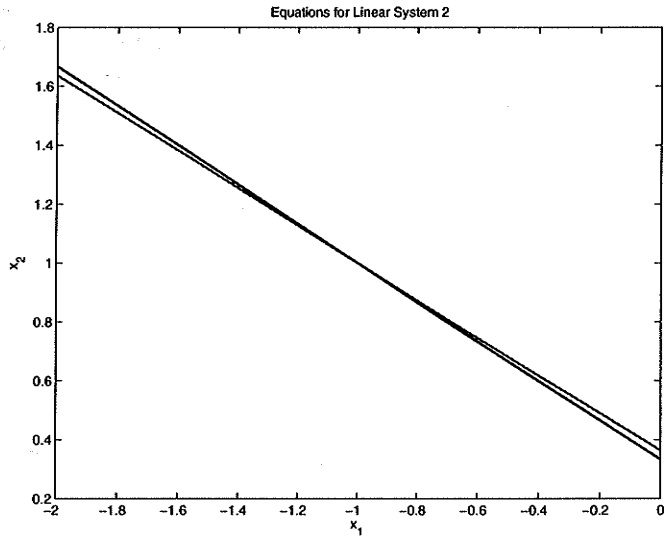


Figure 1.4. *Plot of an ill-conditioned system of linear equations. Small changes in the data can move the intersection of the two lines by a large amount. This is the example in (1.1) except that the (1,1) coefficient in the matrix has been changed from* 0.661 *to* 0.630 *so that the two lines could be distinguished visually.*

If we compute the solution with pivoting, truncating all intermediate results to 3 digits, we obtain

$$x_c = \begin{bmatrix} -.470 \\ .647 \end{bmatrix},$$

whereas the true solution is quite far from this:

$$x_{true} = \begin{bmatrix} -1.000 \\ 1.000 \end{bmatrix}.$$

But when we plug our computed solution back into (1.1), we see that the residual, or difference between the left and right sides, is

$$r = \begin{bmatrix} -.000507 \\ -.000250 \end{bmatrix}.$$

Gauss elimination with pivoting produced a small residual because it is a stable algorithm, so it is guaranteed to solve a nearby problem. But the x-error is not small, since the problem is ill-conditioned. We can see this graphically, in Figure 1.4; if we wiggle the coefficients of the two lines, we can make the intersection move quite a bit.

Sometimes we have additional information about the solution to a problem that gives us some guidance about improving a computed solution, as in the next challenge.

CHALLENGE 1.12. Suppose you solve the nonlinear equation $f(x) = 0$ using a MATLAB routine, and the answers are complex numbers with small imaginary parts. If you know that the true answers are real numbers, what would you do?

Life may toss us some ill-conditioned problems, but there is no good reason to settle for an unstable algorithm.

In the next chapter we illustrate various ways of measuring the sensitivity or conditioning of a problem.