Applications of Mathematics MA 794

N. Chernov and I. Knowles

Foreword

In this course, we will discuss various applications of mathematics in sciences, industry, medicine and business. The main objective is to prepare the students to possible employment in nonacademic world. The course will cover basic topics in numerical analysis, applied linear algebra, differential equations, statistics, computer programming, as well as discussion of specific applied problems on which our faculty are working. The course will give the students some experience and expertise in applied areas of mathematics. Even those pursuing an academic career may benefit from the experience gained in this course.

Tentative syllabus

In Fall 2002, we plan to cover the following topics:

- Basic concepts of numerical analysis: Computer arithmetic, Conditioning, Numerical stability, Rates of convergence.
- Solving equations in $\mathbb{R}^1$ and $\mathbb{R}^n$.
- Optimization algorithms: Golden section, Steepest descent, Newton-Raphson, Simplex method, Levenberg-Marquardt, Simulated annealing.
- Random number generators, Monte-Carlo methods.
- Statistical data processing. Maximum likelihood estimation. Least squares fit.
- Neural networks (time permitting).

In Spring 2003, the emphasis will be on learning programming languages and software packages and doing computer projects.

Students' work

The students are expected to do exercises assigned in class and turn in solutions in writing.

Individual reading projects will be assigned, on which the students will either write a report or give a presentation in class.

Computer projects will be assigned later (most likely, in Spring 2003).

Grading policy

In Fall 2002, the students will be graded on a pass/fail basis. In Spring 2003, everyone will receive a letter grade that will reflect his/her performance for the year.

References

1. L. Trefethen and D. Bau, III, *Numerical Linear Algebra*, SIAM 1997.

2. D. Kincaid and W. Cheney, *Numerical Analysis*, 2nd Ed., Brooks/Cole, 1996.

3. W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in Fortran*, 2nd Ed., Cambridge U. Press, Cambridge, 1992. Alternatively: *Numerical Recipes in C*, 2nd Ed., Cambridge U. Press, Cambridge, 1993. Available on-line at http://www.library.cornell.edu/nr/nr_index.cgi (as well as many other web sites).

Other books and/or journal papers may be used as well.

There is no need to buy or copy these articles. The instructor will provide copies of relevant pages and his own notes regularly.

1 Conditioning in Numerical Analysis

Numerical analysis deals with numerical (computer) solutions of mathematical problems and respective algorithms of such solutions. A good and passionate discussion of the nature and goals of numerical analysis can be found in book [1], on pp. 321–327. It is highly recommended for (hopefully, enjoyable) reading. Here we begin with basic definitions, following Lecture 12 in [1].

A *problem* in numerical analysis is to take the initial data (*input*) and find/compute a solution (*output*). Both input and output consist of one or more real numbers, so we can think of the input and the output as vectors in $\mathbb{R}^n$, $n \geq 1$.

1.1 Definition of a problem

Solving a *problem* is equivalent to constructing a function $f : X \to Y$ from one normed vector space X to another normed vector space Y.

The function f is usually continuous everywhere except some well defined bad points (singular points).

Unless stated otherwise, we assume here that $X = \mathbb{R}^n$ and $Y = \mathbb{R}^m$ with some $m, n \geq 1$, and the norm in X and Y is the Euclidean norm.

1.2 Example

Given a quadratic equation $ax^2 + bx + c = 0$, find its roots. This can be done by using the canonical formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(the case $a = 0$ requires a special treatment, though). The coefficients a, b, c are the *input data* and the roots x_1, x_2 are the *output data*. Hence, $X = \mathbb{R}^3$ and $Y = \mathbb{R}^2$. The function $X \to Y$ is defined on the domain $b^2 - 4ac \geq 0$. It is easy to see that it is continuous everywhere except $a = 0$.

1.3 Definition of an absolute condition number

The *absolute condition number* $\hat{\kappa}$ of a problem $f : X \to Y$ at a point $x \in X$ is

$$\hat{\kappa} = \hat{\kappa}(f, x) = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|}{\|\delta x\|}$$

where $\delta f = f(x + \delta x) - f(x)$.

If f is differentiable at x with derivative $J(x)$ (note that $J(x)$ is an $m \times n$ matrix), then $\hat{\kappa} = \|J(x)\|$. This follows from the definition of a norm of a matrix.

1.4 Definition of a relative condition number

The *relative condition number* κ of a problem $f : X \to Y$ at a point $x \in X$ is

$$\kappa = \kappa(f, x) = \lim_{\delta \to 0} \sup_{\|\delta x\| \le \delta} \left(\frac{\|\delta f\|}{\|f\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right)$$

where $\delta f = f(x + \delta x) - f(x)$.

If f is differentiable at x with derivative $J(x)$ (which is an $m \times n$ matrix), then

$$\kappa = \frac{\|J(x)\|}{\|f(x)\|/\|x\|}$$

1.5 Discussion: relative versus absolute condition numbers

If the input data are perturbed by a small amount, then the output will be perturbed by some amount, too. The absolute condition number gives a maximum factor by which perturbations of input are multiplied to produce perturbations of output. In other words, if the input data are known to be accurate to within ε, the output data will be accurate to within $\hat{\kappa}\,\varepsilon$. The relative condition number is a similar factor, but it corresponds to the relative accuracy of our data.

Now, do we really need two different condition numbers? Actually, one is enough. But which one should we use? There are many reasons why it is **relative** accuracy that we should care about in practical applications, rather than absolute accuracy. To demonstrate this, consider a very practical example - the price of merchandize. When you buy a bottle of soda, its price may be, for example, 89 cents or \$1.19. When you buy a TV set, it may cost \$139 or \$195. When you buy a car, it price tag is, for example, \$15,650 or \$24,990. When you buy a house, the seller may ask \$156,000 or \$225,900. Do you see the trend? In most cases, both sellers and buyers operate with 3–4 significant (nonzero) digits, i.e. their business has a natural level of relative accuracy – between 1% and 0.1%. The absolute accuracy does not seem to have practical value, even in the world of dollars and cents.

In Chapter 2 we will see that the computer arithmetic is also based on relative accuracy. We will only use relative condition number, and from now on forget the absolute condition number. Whenever we mention a condition number, we always mean the relative condition number.

Here is another interpretation of a (relative) condition number κ. It allows us to determine the number of digits in the output that we can trust. For example, if the input x is known to p decimal digits, i.e. $\|\delta x\|/\|x\| = \mathcal{O}(10^{-p})$, and $\kappa = 10^q$, then the output y will have about $p - q$ accurate digits, i.e. $\|\delta y\|/\|y\| = \mathcal{O}(10^{-(p-q)})$.

A problem is *well-conditioned* if κ is small (e.g., 1, 10, 10^2), and *ill-conditioned* if κ is large (e.g., 10^6, 10^{10}).

Note: Definition 1.4 is not perfect, it has some weak points when applied to scientific computations. We will discuss its weak points later.

1.6 Example

Consider the trivial problem of multiplication $x \mapsto cx$ where $c \neq 0$ is a fixed scalar. Then

$$\kappa = \frac{|c|}{|cx|/|x|} = 1$$

Hence, multiplication by a scalar is always well-conditioned.

1.7 Example

Consider the problem of multiplying two numbers $(x, y) \mapsto xy$. Then $J = (y, x)$ and $\|J\| = \sqrt{x^2 + y^2}$, so

$$\kappa = \frac{\sqrt{x^2 + y^2}}{|xy|/\sqrt{x^2 + y^2}} = \frac{|x|}{|y|} + \frac{|y|}{|x|}$$

Hence, the multiplication of two numbers is well-conditioned if they are comparable in absolute value, but it may be ill-conditioned otherwise.

This last conclusion is actually inaccurate, we will see later a different, more accurate viewpoint. Generally, multiplication (and division) are considered to be *always* well-conditioned in numerical analysis, unlike subtraction and division. The alleged ill-conditioning in this example just demonstrates the imperfection of our Definition 1.4.

1.8 Exercise

Find the condition number of the division $x \mapsto c/x$, where $c \neq 0$ is a scalar, and of the division of two numbers $(x, y) \mapsto x/y$.

1.9 Example

Consider the problem of computing the square root $x \mapsto \sqrt{x}$ for $x > 0$. We have

$$\kappa = \frac{1/(2\sqrt{x})}{\sqrt{x}/x} = \frac{1}{2}$$

This problem is always well-conditioned.

When the condition number happens to be less than 1, it might be also misleading. Literally, it tells us that the accuracy of the output y is higher than that of the input x. But this rarely happens in practice and is something clearly counterintuitive (we will elaborate later). It is safer to define the condition number κ by

$$\kappa = \max\left\{1, \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \left(\frac{\|\delta f\|}{\|f\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right) \right\}$$

(compare this to our Definition 1.4). Then the condition number in Example 1.9 would be equal to 1.

1.10 Example

Consider the subtraction of two numbers $(x, y) \mapsto x - y$. We have

$$\kappa = \frac{\sqrt{2}}{|x-y|/\sqrt{x^2+y^2}} = \frac{\sqrt{2x^2+2y^2}}{|x-y|}$$

This is well-conditioned if $x - y$ is comparable to $\max\{|x|, |y|\}$ and ill-conditioned otherwise.

The above ill-conditioning is *not* an artifact of our Definition 1.4, this is a real thing. It is well known in numerical analysis that subtracting two nearly equal numbers leads to a *catastrophic cancellation* and must be avoided if at all possible.

1.11 Exercise

Find the condition number for the addition $(x, y) \mapsto x + y$.

1.12 Example

A matrix A of size $m \times n$ defines a linear map $\mathbb{R}^n \to \mathbb{R}^m$. Its condition number at $x \in \mathbb{R}^n$ is

$$\kappa(A, x) = \|A\| \frac{\|x\|}{\|Ax\|}$$

If A is a square matrix ($m = n$) and is nonsingular ($\det A \neq 0$), we can take maximum with respect to x and obtain

$$\kappa(A) = \max_{x \neq 0} \kappa(A, x) = \|A\| \, \|A^{-1}\|$$

This is the definition of the condition number of a matrix adopted in linear algebra.

Recall: the condition number of a square $n \times n$ matrix A is the quotient of its largest and smallest singular values:

$$\kappa(A) = \frac{\sigma_1}{\sigma_n}$$

1.13 Theorem (see [1], page 94)

Let A be a square nonsingular matrix. Consider the problem of solving the system of linear equations $Ax = b$. Assuming that A is fixed, it defines a map $b \mapsto x$ according to $x = A^{-1}b$. The condition number of this problem is

$$\kappa = \|A^{-1}\| \frac{\|b\|}{\|x\|} \leq \|A\| \, \|A^{-1}\| = \kappa(A)$$

1.14 Theorem (see [1], page 95)

Let A be a square nonsingular matrix. Consider the problem of solving the system of linear equations $Ax = b$. It defines a map $(A, b) \mapsto x$ according to $x = A^{-1}b$. The condition number of this problem is

$$\kappa = \|A\| \, \|A^{-1}\| = \kappa(A)$$

The proofs of Theorems 1.13 and 1.14 are also provided in Applied Linear Algebra, MA 660.

1.15 Example

The calculation of the roots of a polynomial, given its coefficients, is a classic example of an ill-conditioned problem. Consider a rather simple polynomial

$$P(x) = \prod_{i=1}^{20} (x - i) = a_0 + a_1 x + \cdots + a_{19} x^{19} + x^{20}$$

studied by Wilkinson (see pages 92–93 of [1] and page 77 of [2]). This polynomial happens to have a huge condition number

$$\kappa \approx 5.1 \times 10^{13}$$

The illustration on page 93 of [1] shows a disastrous effect of ill-conditioning. A historical note on page 77 of [2] presents an interesting story of Wilkinson's discovery.

1.16 Exercise

Prove that the quadratic polynomial

$$P(x) = (x - 1)^2$$

has infinite condition number:

$$\kappa = \infty$$

(formally, here we consider a map $\mathbb{R}^3 \to \mathbb{R}^2$ defined in 1.2, and then κ is the respective condition number at the point $(1, 2, 1)$.)

1.17 Example

The calculation of eigenvalues and eigenvectors of a matrix is another example of an ill-conditioned problem. To illustrate this, consider two matrices

$$\begin{bmatrix} 1 & 1000 \\ 0 & 1 \end{bmatrix}, \qquad \begin{bmatrix} 1 & 1000 \\ 0.001 & 1 \end{bmatrix},$$

which have eigenvalues $\{1, 1\}$ and $\{0, 2\}$, respectively. On the other hand, if a matrix is symmetric (more generally, if it is normal), then its eigenvalues are well-conditioned.

Recall facts from Applied Linear Algebra: it is possible to define and compute the condition number of any eigenvalue for any square matrix. In particular, the condition number of a *simple* eigenvalue λ of a square matrix A is defined by $\kappa(\lambda) = 1/|y^* x|$ where x and y are the corresponding right and left unit eigenvectors. For symmetric and normal matrices, the vectors x and y are always parallel, hence $\kappa(\lambda) = 1$, but generally we can only say that $\kappa(\lambda) \geq 1$. We will not these facts, though.

2 Machine Arithmetic

2.1 Binary numbers

A *bit* is a *b*inary dig*it*, it can only take two values: 0 and 1. Any natural number N can be written, in the binary system, as a sequence of binary digits:

$$N = (d_n \cdots d_1 d_0)_2 = 2^n d_n + \cdots + 2d_1 + d_0$$

For example, $5 = 101_2$, $11 = 1011_2$, $64 = 1000000_2$, etc.

2.2 More binary numbers

To represent more numbers in the binary system (not just positive integers), it is convenient to use the numbers $\pm N \times 2^{\pm M}$ where M and N are positive integers. Clearly, with these numbers we can approximate any real number arbitrarily accurately. In other words, the set of numbers $\{\pm N \times 2^{\pm M}\}$ is dense on the real line. The number $E = \pm M$ is called the *exponent* and N the *mantissa*.

Note that $N \times 2^M = (2^k N) \times 2^{M-k}$, so the same real number can be represented in many ways in the form $\pm N \times 2^{\pm M}$ with different M and N.

2.3 Floating point representation

We return to our decimal system. Any decimal number (with finitely many digits) can be written as

$$f = \pm .d_1 d_2 \ldots d_t \times 10^e$$

where d_i are the decimal digits of f, and e is an integer. For example, $18.2 = .182 \times 10^2 = .0182 \times 10^3$, etc. This is called a *floating point* representation of decimal numbers. The part $.d_1 \ldots d_t$ is called the *mantissa* and e is the *exponent*. By changing the exponent e with a fixed mantissa, $d_1 \ldots d_t$, we can move ("float") the decimal point, for example $.182 \times 10^2 = 18.2$ and $.182 \times 10^1 = 1.82$.

2.4 Normalized floating point representation

To avoid unnecessary multiple representations of the same number (such as $.182 \times 10^2$ and $.0182 \times 10^3$ which represent one number, 18.2), we require that $d_1 \neq 0$. We say the floating point representation is *normalized* if $d_1 \neq 0$. Then $.182 \times 10^2$ is the only normalized representation of the real number 18.2.

For every positive real $f > 0$ there is a unique integer $e \in \mathbb{Z}$ such that $g := 10^{-e} f \in [0.1, 1)$. Then $f = g \times 10^e$ is the normalized representation of f. So, the normalized representation is unique.

2.5 Other number systems

Now suppose we are working in a number system with base $\beta \geq 2$. By analogy with 2.3, the floating point representation is

$$f = \pm .d_1 d_2 \dots d_t \times \beta^e$$

where $0 \leq d_i \leq \beta - 1$ are digits,

$$.d_1 d_2 \dots d_t = d_1 \beta^{-1} + d_2 \beta^{-2} + \cdots d_t \beta^{-t}$$

is the mantissa and $e \in \mathbb{Z}$ is the exponent. Again, we say that the above representation is normalized if $d_1 \neq 0$, this ensures uniqueness.

2.6 Machine floating point numbers

Any computer can only handle finitely many numbers. Hence, the number of digits d_i's is necessarily bounded, and the possible values of the exponent e are limited to a finite interval. Assume that the number of digits t is fixed (it characterizes the accuracy of machine numbers) and the exponent is bounded by $L \leq e \leq U$. Then the parameters β, t, L, U completely characterize the set of numbers that a particular machine system (or a particular computer) can handle. The most important parameter is t, the number of significant digits, or the length of the mantissa. (Note that the same computer can use many possible machine systems, with different values of β, t, L, U, see 2.8.)

2.7 Remark

The maximal (in absolute value) number that a machine system can handle is $M = \beta^U (1 - \beta^{-t})$. The minimal positive number is $m = \beta^{L-1}$.

2.8 Examples

Most computers use the binary system, $\beta = 2$. Many modern computers (e.g., all IBM compatible PC's) conform to the IEEE floating-point standard (ANSI/IEEE Standard 754-1985). This standard provides two systems. One is called *single precision*, it is characterized by $t = 24$, $L = -125$ and $U = 128$. The other is called *double precision*, it is characterized by $t = 53$, $L = -1021$ and $U = 1024$. The PC's equipped with the so called *numerical coprocessor* (which is built-in on all Pentium class chips) also use a different internal system called *temporary format*, it is characterized by $t = 65$, $L = -16381$ and $U = 16384$.

2.9 Relative errors

Let $x > 0$ be a positive real number with the normalized floating point representation with base β

$$x = .d_1 d_2 \dots \times \beta^e$$

where the number of digits may be finite or infinite. We need to represent x in a machine system with parameters β, t, L, U. If $e > U$, then x cannot be represented (an attempt to store x in a computer memory or perform calculation that results in x should terminate the computer program with error message OVERFLOW, which means that the attempted number is too large). If $e < L$, the system may either represent x by 0 (in many cases, this is a reasonable action) or terminate the program with error message UNDERFLOW – an attempted number is too small. If $e \in [L, U]$ is within the proper range, then the mantissa has to be reduced to t digits (if it is longer than t or infinite). There are two standard ways to do this reduction:

(i) just take the first t digits of the mantissa of x, i.e. $.d_1 \ldots d_t$, and drop ("chop off") the rest;

(ii) round off to the nearest available machine number, i.e. take the mantissa

$$\begin{cases} .d_1 \ldots d_t & \text{if } d_{t+1} < \beta/2 \\ .d_1 \ldots d_t + .0 \ldots 01 & \text{if } d_{t+1} \geq \beta/2 \end{cases}$$

Denote the obtained number by $\mathrm{fl}(x)$ (the computer floating point representation of x). The relative error in this representation can be estimated as

$$\frac{\mathrm{fl}(x) - x}{x} = \varepsilon \quad \text{or} \quad \mathrm{fl}(x) = x(1 + \varepsilon)$$

where the maximal possible value of ε is

$$\varepsilon_{\text{machine}} = \begin{cases} \beta^{1-t} & \text{for chopped arithmetic} \\ \frac{1}{2}\beta^{1-t} & \text{for rounded arithmetic} \end{cases}$$

The number $\varepsilon_{\text{machine}}$ is called the *machine epsilon* or *machine precision*.

2.10 Examples

a) For the IEEE floating-point standard with chopped arithmetic in single precision we have $\varepsilon_{\text{machine}} = 2^{-23} \approx 1.2 \times 10^{-7}$. In other words, approximately 7 decimal digits are accurate.

b) For the IEEE floating point standard with chopped arithmetic in double precision we have $\varepsilon_{\text{machine}} = 2^{-52} \approx 2.2 \times 10^{-16}$. In other words, approximately 16 decimal digits are accurate.

2.11 Experimental search

The quantity $\varepsilon_{\text{machine}}$ is the most important characteristic of a machine system. If you work on a computer with an unknown machine arithmetic, you can determine $\varepsilon_{\text{machine}}$ experimentally by running the following program:

Step 1: Set $x = 1$.

Step 2: Compute $y = 1 + x$.

Step 3: If $y = 1$, stop, otherwise reset $x = x/2$ and go back to Step 2.

The resulting number x will be (approximately) $\varepsilon_{\text{machine}}$.

The smallest and largest numbers m and M available in a machine system are less important, since practical computations rarely reach these limits. But if you need them, these numbers can also be determined experimentally, in an obvious manner.

2.12 Exact machine numbers

Which real numbers $x \in \mathbb{R}$ can be represented *exactly* in a machine system? Besides zero, these are rational numbers whose binary mantissa does not exceed t significant bits and whose binary exponent lies between L and U. For example:

(a) 1, -2, 355, -65001, $0.5 = 2^{-1}$, and $2^{10} + 2^{-7}$, are exact machine numbers in both single and double precision arithmetics (note that the binary mantissa of the number $2^{10} + 2^{-7}$ is 18 bits long);

(b) $2^{20} + 2^{-8}$ is an exact machine number in double precision but not in single precision (its binary mantissa is 29 bits long, which exceeds 24 bits);

(c) The following numbers are not exact in either single or double precision: 0.1 (its binary mantissa is infinite!), 2^{1200} (the number itself is too large), 2^{-1200} (this number is too small), 10^{45} (its binary mantissa is finite but too long), $2^{45} + 2^{-15}$ (its binary mantissa is 61 bits long).

In computer programming, it helps to keep these rules in mind.

2.13 Computational errors

Let x and y be two real numbers represented in a machine system by $\mathrm{fl}(x)$ and $\mathrm{fl}(y)$, respectively. An arithmetic operation $x * y$, where $*$ is one of $+, -, \times, \div$, is performed by a computer in the following way. The computer finds $\mathrm{fl}(x) * \mathrm{fl}(y)$ (first, exactly) and then represents that number in the machine system. The result is

$$z := \mathrm{fl}(\mathrm{fl}(x) * \mathrm{fl}(y))$$

Note that, generally, z is different from $\mathrm{fl}(x * y)$, which is the machine representation of the exact result $x*y$. Hence, z is not necessarily the best representation for $x*y$. In other words, the computer makes additional round-off errors during computations. Assuming that $\mathrm{fl}(x) = x(1+\varepsilon_1)$ and $\mathrm{fl}(y) = y(1+\varepsilon_2)$ we have

$$\mathrm{fl}(\mathrm{fl}(x) * \mathrm{fl}(y)) = (\mathrm{fl}(x) * \mathrm{fl}(y))(1+\varepsilon_3) = \Big([x(1+\varepsilon_1)] * [y(1+\varepsilon_2)]\Big)(1+\varepsilon_3)$$

where $|\varepsilon_1|, |\varepsilon_2|, |\varepsilon_3| \le \varepsilon_{\text{machine}}$.

2.14 Axioms of Floating Point Arithmetic

Textbook [1] states two axioms of machine number systems:

A1. For any real number $x \in \mathbb{R}$ within the proper range (either $m < |x| < M$ or $x = 0$) there exists ε with $|\varepsilon| < \varepsilon_{\text{machine}}$ such that

$$\text{fl}(x) = x(1 + \varepsilon)$$

A2. For all machine numbers x, y there exists ε with $|\varepsilon| < \varepsilon_{\text{machine}}$ such that

$$\text{fl}(x * y) = (x * y)(1 + \varepsilon)$$

2.15 Multiplication and Division

For multiplication, we have

$$z = xy(1 + \varepsilon_1)(1 + \varepsilon_2)(1 + \varepsilon_3) \approx xy(1 + \varepsilon_1 + \varepsilon_2 + \varepsilon_3)$$

so the relative error is (approximately) bounded by $3\varepsilon_{\text{machine}}$. A similar estimate can be made in the case of division.

2.16 Addition and Subtraction

For addition, we have

$$z = (x + y + x\varepsilon_1 + y\varepsilon_2)(1 + \varepsilon_3) = (x + y)\left(1 + \frac{x\varepsilon_1 + y\varepsilon_2}{x + y}\right)(1 + \varepsilon_3)$$

The relative error is now small if $|x|$ and $|y|$ are not much bigger than $|x + y|$. The error, however, can be very large if $|x + y| \ll \max\{|x|, |y|\}$. This effect is known as *catastrophic cancellation*. A similar estimate can be made in the case of subtraction $x - y$: if $|x - y|$ is not much smaller than $|x|$ or $|y|$, then the relative error is small, otherwise we may have a catastrophic cancellation.

2.17 Example

Consider the system of equations

$$\begin{pmatrix} 0.01 & 2 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

The exact solution is $x = 200/197 = 1.015\ldots$ and $y = 196/197 = 0.995\ldots$.

Now, to demonstrate the effect of round off errors, let us solve this system with the chopped arithmetic with base $\beta = 10$ and $t = 2$ (i.e. working with a two digit mantissa) by the standard Gaussian elimination method. We do not need a computer for this, the job can be done manually. The result is $x = 0$ and $y = 1$. The value of x is very inaccurate, its relative error is 100%.

Note: solving equations in machine arithmetic is different from solving them exactly. We 'pretend' that we are computers, and so we are subject to the strict rules of machine arithmetic, in particular we are limited to t significant digits. As we notice, these limitations may lead to unexpectedly large errors in the end.

2.18 Exercise

Solve the system in 2.17 in a machine arithmetic with $\beta = 10$ and $t = 3$. Any improvement? [Answer: $x = 2$. Still, the relative error in x is about 100%.]

2.20 Exercise

Compute the value $z = (x - y)^2$ in two ways:

$$z = (x - y) \times (x - y) \qquad \text{and} \qquad z = x \times x - 2 \times x \times y + y \times y$$

for $x = 1$ and $y = 0.98$ by using the system with the rounded arithmetic with base $\beta = 10$ and $t = 2$. How come one of the results is negative? See also a plot on page 79 of [2].

3 Numerical Stability

Consider again a problem $f : X \to Y$ of numerical analysis. When it is solved by a computer, the input data x has to be converted to its machine floating point version $\mathrm{fl}(x)$. Therefore, the computer can, at best, find $f(\mathrm{fl}(x))$, instead of $f(x)$. The relative error of the output is related to the relative error of the input, which is known to be less than $\varepsilon_{\mathrm{machine}}$, via the condition number defined in 1.4. This gives us the estimate

$$\frac{\|f(\mathrm{fl}(x)) - f(x)\|}{\|f(x)\|} \leq \kappa(f, x)\, \varepsilon_{\mathrm{machine}} \tag{1}$$

This may be bad enough already, when the problem is ill-conditioned. However, in reality things appear to be even worse, since the computer does not evaluate $f(\mathrm{fl}(x))$ precisely, apart from trivial cases. The computer executes a certain program that normally involves many arithmetic operations and other functions, like square root or logarithms. As the program runs, more and more round-off errors are made at each step and the errors compound toward the end. As a result, the computer program transforms the (machine) input $\mathrm{fl}(x)$ into a (machine) output $\tilde{y}$ which is generally different from $f(\mathrm{fl}(x))$. We denote the composite transformation $x \mapsto \mathrm{fl}(x) \mapsto \tilde{y}$ by $\tilde{y} = \tilde{f}(x)$, which defines another function $\tilde{f} : X \to Y$. It depends not only on the machine system but even more on the algorithm that is used to evaluate f.

Recall Exercise 2.20: the same function $z = (x - y)^2$ can be evaluated in two ways. Both ways are theoretically equivalent and practically appeared almost identical, but we obtained different results (one of them positive and the other, paradoxically, negative).

So, let a problem $f : X \to Y$ be solved on a computer with a particular algorithm defining another function $\tilde{f} : X \to Y$. The accuracy of the algorithm at a point $x \in X$ can be characterized by the relative error

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|}$$

Since the numerical algorithm for evaluating f cannot be more accurate than the exact function f, we do not expect the above relative error to be smaller than the relative error (1) estimated by $\kappa(f, x)\, \varepsilon_{\mathrm{machine}}$. But we may hope that it will not be any larger either. In other words, a good algorithm should not magnify the errors caused already by the round-off of x and conditioning, as expressed by (1). If this is the case, the algorithm is said to be stable.

3.1 Definition (stable algorithm)

An algorithm $\tilde{f}$ is *stable* if for each $x \in X$

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = \mathcal{O}(\varepsilon_{\mathrm{machine}})$$

for some $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\varepsilon_{\text{machine}})$$

In words,

> A stable algorithm gives nearly the right answer to nearly the right question.

3.2 Definition (backward stable algorithm)
An algorithm $\tilde{f}$ is *backward stable* if for each $x \in X$

$$\tilde{f}(x) = f(\tilde{x})$$

for some $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\varepsilon_{\text{machine}})$$

In words,

> A backward stable algorithm gives exactly the right answer to nearly the right question.

This is a tightening of the previous definition in that $\mathcal{O}(\varepsilon_{\text{machine}})$ is replaced by zero. Hence, a backward stable algorithm is always stable (but not vice versa).

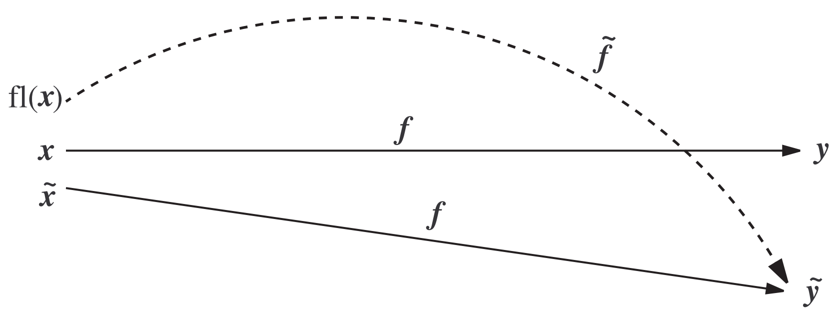

Figure 1: Numerical stability of an algorithm.

3.3 Remark

The notation $\mathcal{O}(\varepsilon_{\text{machine}})$ is understood in a usual way: we say that $g = \mathcal{O}(\varepsilon_{\text{machine}})$ if there is a constant $C > 0$ such that

$$|g/\varepsilon_{\text{machine}}| \le C$$

as $\varepsilon_{\text{machine}} \to 0$. Of course, the last limit is purely hypothetical, since in practice the machine accuracy is always a fixed quantity.

The constant $C > 0$ in Definitions 3.1 and 3.2 is supposed to be uniform in $x \in X$, see [1]. However, for practical purposes it is more important that $C > 0$ is reasonably small, something like 10 or 10^2 (in the same sense in which $\kappa(f, x)$ should be small for a problem to be well conditioned, recall Discussion 1.5).

The following relations are simple but useful:

(a) $(1 + \mathcal{O}(\varepsilon_{\text{machine}}))(1 + \mathcal{O}(\varepsilon_{\text{machine}})) = 1 + \mathcal{O}(\varepsilon_{\text{machine}})$

(b) $\sqrt{1 + \mathcal{O}(\varepsilon_{\text{machine}})} = 1 + \mathcal{O}(\varepsilon_{\text{machine}})$

(c) $(1 + \mathcal{O}(\varepsilon_{\text{machine}}))^{-1} = 1 + \mathcal{O}(\varepsilon_{\text{machine}})$

3.4 Theorem

The arithmetic operations $+, -, \times, \div$ are backward stable.

Proof. See [1], pp. 108–109, for the proof of backward stability for subtraction. The proofs for other operations are similar. $\square$

3.5 Examples

The scalar product of vectors $x, y \in \mathbb{R}^n$ defined by $c = x^T y$ is backward stable. The tensor product of vectors $A = xy^T$ is stable, but not backward stable. The operation $x \mapsto x + 1$ is stable but not backward stable. For details, see [1], page 109.

¿From practical viewpoint, there is little or no difference between stability and backward stability. If an algorithm is stable, it is good for all practical purposes. Backward stability just happens to be easier to verify analytically, this is why we care about it.

3.6 Theorem

Suppose a backward stable algorithm is applied to solve a problem $f : X \to Y$ with condition number κ. Then the relative errors satisfy

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = \mathcal{O}(\kappa(f, x)\,\varepsilon_{\text{machine}})$$

Proof. See [1], page 111.

3.7 Exercise

Does the above theorem hold for stable algorithms? Under what additional condition does it hold for stable algorithms?

[Answer: the condition is $\|f(x)\|/\|\tilde{f}(x)\| = \mathcal{O}(1)$.]

3.8 Exercise

Compare two algorithms for computing the function $z = (x-y)^2$:

$$z = (x-y) \times (x-y) \qquad \text{and} \qquad z = x \times x - 2 \times x \times y + y \times y$$

(again recall 2.20). Determine if each of them is *backward stable*, *stable but not backward stable* or *unstable*.

3.9 Example

Suppose we are going to find the larger root of a quadratic equation $x^2 + px + q = 0$ assuming (for simplicity) that $p^2 - 4q \gg \varepsilon_{\text{machine}}$. A classical formula

$$x = \frac{-p + \sqrt{p^2 - 4q}}{2}$$

gives a simple but, generally, numerically unstable algorithm (after the previous exercise, this should come as no surprise). Precisely, for $p < 0$ the above formula is stable, for $p > 0$ one should use a modified version

$$x = \frac{2q}{-p - \sqrt{p^2 - 4q}}$$

In this way one avoids catastrophic cancellations.

3.10 Example

Suppose one needs to compute

$$y = 1 - \cos x$$

for $x \approx 0$. Using the exact formula $y = 1 - \cos x$ would be very unstable and give very inaccurate results (figure out, why!). A numerically stable algorithm is based on the Taylor expansion:

$$y = \frac{x^2}{2!} - \frac{x^4}{4!} + \cdots$$

Of course, one cannot evaluate an infinite series on a computer, so a partial sum has to be used. How many terms one actually needs to take depends on a particular application. But even one term, $y \approx x^2/2$, would be much more accurate than the exact formula $y = 1 - \cos x$ for small x's. We emphasize this fact, which may be not so easily acceptable by mathematically minded people:

An approximate formula may be numerically more accurate than a theoretically exact formula.

Why does this happen? Here is an explanation. We need to distinguish between two types of errors in numerical calculations. When an infinite series $S = \sum_1^\infty a_i$ is approximated by a partial sum $S_n = \sum_1^n a_i$, the difference is $S_n - S$ called a *truncation error*. Similarly, suppose an iterative algorithm is designed to compute a sequence of numbers a_n converging to a limit a_*. Since the algorithm has to stop at some time, it will return the current value a_n as an approximation to a_*, and then $a_n - a_*$ will be the truncation error. Truncation errors occur "by design", they are inevitable and independent of the machine arithmetic (they would occur even if our computers had infinite precision). On the other hand, round-off errors occur in all floating point computations and depend on our machine precision.

In the above example, the direct formula $y = 1 - \cos x$ involves no truncation errors, but a large round-off error (for small x). On the contrary, the approximative formula $y \approx x^2/2$ involves a small truncation error and a small round off error. The combined error is still small, compared to the round off error of the direct formula.

3.11 Remarks.

As you now realize, the development of a numerically stable algorithm is a tricky business. It is, in a sense, finding a safe path through a mine field or through swamps (or both). Often one begins with a straightforward algorithm, which may or may not work well even in typical cases, and then one tests the algorithm, discovers various pitfalls and drawbacks, and find detours and corrections.

The history of numerical analysis is full of examples of problems for which it took years or decades to find a satisfactory (numerically stable and accurate) solution. A classical example is the linear least squares problem (studied in MA660). A simple (and theoretically exact) solution based on normal equations was used throughout until the 1970s. Then researchers came to a conclusion that it was dangerously unstable. Another algorithm, based on the QR decomposition, became popular. It was more expensive (in terms of computer time) but numerically stable and theoretically still exact. In the 1990s, yet another algorithm, based on SVD, made its way into the market. It is even more expensive than QR but its practical accuracy is higher than that of QR (a nice example is provided in [1] on pp. 137–143). It is interesting that the SVD algorithm is *not exact* in the theoretical sense, it involves an iterative procedure that converges to the exact solution. We will return to this issue in Chapter 8.

The above historical excursion is quite instructive. It frequently happens in applied mathematics that simple and theoretically exact algorithms are numerically unstable and inaccurate. In this case one needs to find a stable and accurate algorithm, even if the latter is more complicated, computationally expensive, and based on approximations rather than exact formulas.

3.12 Exercise

Suppose we want to compute

$$S = \sum_{n=1}^{\infty} \frac{1}{n^2}$$

(in theory, it equals $\pi^2/6$). Our computer program evaluates a partial sum

$$S' = \sum_{n=1}^{N} \frac{1}{n^2}$$

where N is the largest integer such that $1/N^2 > \varepsilon_{\text{machine}}$ (further terms, with $n > N$, may be ignored since they will not alter the result). Estimate the relative error of this algorithm. Is it $\mathcal{O}(\varepsilon_{\text{machine}})$ or larger?

3.13 Exercise

Now suppose that in the previous exercise the series is summed up from right to left:

$$S'' = \sum_{n=1}^{N} \frac{1}{(N-n+1)^2}$$

where N is the same as before. Estimate the relative error of this algorithm.

Compare 3.12 and 3.13. How would you evaluate the sum of a series on a computer – add numbers in an increasing order or in a decreasing order?

3.14 Exercise

The following series (known as harmonic series) is divergent:

$$S = \sum_{n=1}^{\infty} \frac{1}{n}$$

However, if one adds these numbers, from left to right, on a computer, one arrives at a finite answer, since the terms $1/n < \varepsilon_{\text{machine}}$ will no longer alter the sum. Can you estimate the result in single precision? Can you estimate the result in double precision? To answer these questions, you do not have to run a computer program. However, if you are familiar with any programming language (BASIC, FORTRAN, PASCAL, C) you can easily run the corresponding program on a computer and obtain a numerical result.

Note: it is common to verify the accuracy of a computer algorithm by running it in single precision and then in double precision. If the results are far from each other, as in the above example, neither one can be trusted. If they are close to each other, one hopes (fingers crossed!) that the results are accurate.

4 Rates of Convergence

Iterative algorithms were mentioned in the previous section. Many problems do not admit exact solutions, and then numerical schemes involving some sort of successive approximation are applied. (We have seen that even if exact solutions are available, sometimes iterative algorithms are more stable and/or accurate.)

A typical iterative procedure produces a sequence of points $y_n \in Y$ that supposedly converges to the exact solution y. Here we discuss the *rates of convergence*, i.e. the speed of convergence to zero of the sequence

$$e_n = \|y_n - y\|/\|y\|$$

4.1 Preliminaries

In mathematics, a sequence converging to zero may do so with different speeds. For example, $a_n = 1/n$ or, more generally, $a_n = 1/n^b$ (here $b > 0$ is a constant), converges to zero slowly. On the contrary, the sequence $a_n = 1/2^n$, or, more generally, $a_n = q^n$ (here $q < 1$ is a constant), converges rapidly. Rarely one encounters sequences that decrease faster than q^n for some $q < 1$.

However, in numerical analysis, the sequence $a_n = q^n$ is regarded as one of the slowest (!) to converge. One frequently encounters sequences that converge to zero as $a_n = q^{2^n}$ or $a_n = q^{3^n}$, etc. This is formalized in the following definitions.

4.2 Linear convergence in numerical analysis

A sequence a_n is said to converge to zero *linearly* if

$$\left|\frac{a_{n+1}}{a_n}\right| \leq \lambda$$

for some $\lambda < 1$. If the ratio $|a_{n+1}/a_n|$ is approximately λ (or if $|a_{n+1}/a_n| \to \lambda$ as $n \to \infty$), then λ is called the *rate* of convergence. In that case, asymptotically, $a_n \sim \text{const} \cdot \lambda^n$.

4.3 Quadratic convergence in numerical analysis.

A sequence a_n is said to converge to zero *quadratically* if

$$|a_{n+1}| \leq C|a_n|^2$$

for some $C > 0$. In that case, asymptotically, $a_n \sim \text{const} \cdot \lambda^{2^n}$ for some $\lambda > 0$, see Exercise below.

Similarly, one says that a convergence is cubic in the case

$$|a_{n+1}| \leq C|a_n|^3$$

for some $C > 0$. And more generally, a convergence of order $\beta > 0$ means that

$$|a_{n+1}| \leq C|a_n|^\beta$$

for some $C > 0$.

One may wonder if such a fast convergence is indeed possible. We will see later that many good numerical algorithms do achieve quadratic or even cubic convergence. Next we will explain what various speeds of convergence mean in terms of numerical accuracy.

4.4 Exercise

Let a sequence a_n converge to zero and satisfy $|a_{n+1}| \leq C|a_n|^2$ with some $C > 0$ for all $n \geq 1$. Prove that $a_n < C_1\lambda^{2^n}$ with some $C_1 > 0$ and $\lambda < 1$.

4.5 More on linear convergence

Suppose we have a sequence of approximations $\{y_n\}$ that converges to a limit y linearly at a rate $\lambda = 0.5$. Is it fast or slow? We see that each approximation is nearly twice as close to the limit as the previous one. Sounds good? But in terms of the number of accurate binary digits (bits) in the value of y_n, our convergence guarantees exactly one more accurate bit at each approximation. Suppose that the initial guess y_0 is at a relative distance 0.5 from the limit, i.e. $|y_0 - y|/|y| \approx 0.5$, i.e. just one significant bit is accurate. Then, in order to achieve a full accuracy in single precision, we need 22 approximations. In double precision, we will need as many as 51 approximations. (Note that if $\lambda = 0.8$ or 0.9, then the number of necessary approximations would be much higher than that.)

Normally, each approximation y_n requires one iteration of an algorithm. Thus, we are talking about 22 or even 51 iterations here. In many applications, every iteration is quite expensive and making 20-50 iterations is a luxury one cannot afford. Besides, the longer the program runs, the more round-off errors are compounding, and the accuracy is inevitably diluted. A good numerical algorithm should converge in no more than 10 iterations, and one regards a convergence in 3-5 iterations as a high quality certificate.

But how is such a fast convergence possible?

4.6 More on quadratic convergence

Suppose now we have a sequence of approximations $\{y_n\}$ that converges to a limit y quadratically, $|y_{n+1} - y| \leq C|y_n - y|^2$, and assume that $C = 1$ for simplicity. In terms of binary digits, our convergence guarantees that the number of accurate bits in the value of y_n *doubles* with each approximation. For example, if we start with just one accurate bit, then we get $2, 4, 8, 16, 32, 64, \ldots$ bits in subsequent approximations. In 4-5 iterations, one reaches full accuracy in single precision and in 5-6 iterations one reaches full accuracy in double precision! This is what one expects from a good algorithm.

If a sequence y_n converges cubically, then the number of accurate bits *triples* with each approximation. Starting with one accurate bit, one gets $3, 9, 27, 81, \ldots$ accurate bits at subsequent approximations. One could only wish that our computers handled data with such precision. But the cubic convergence is a top class performance that practical algorithms rarely achieve.

5 Solving Equations

The most fundamental problem in numerical analysis is to solve an equation

$$f(x) = 0$$

where $x \in \mathbb{R}$ is an independent variable and $f : \mathbb{R} \to \mathbb{R}$ a given function. This problem admits a natural generalization to $\mathbb{R}^n$, when one solves a system of equations

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

where $\mathbf{x} \in \mathbb{R}^n$ is an independent vector and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ a given map. In this case one normally requires that $m = n$ (the number of equations equals the number of unknowns, otherwise the system may be overdetermined for $m > n$ and underdetermined for $m < n$).

This topic is nicely covered in textbooks [2] (see chapter 3) and [3] (see chapter 9) to which we refer for a detailed reading. We only emphasize the main points here.

5.1 Formalities

At first sight, solving an equation $f(x) = 0$ does not look like a "problem" as defined by 1.1. What are input data? What are output data? Well, in practical applications the formula specifying the function $f(x)$ normally contains some variable coefficients or parameters, which we may consider as input data (for example, if f is a polynomial of degree k, then its $k + 1$ coefficients make input data).

Next, the solution of $f(x) = 0$ may not exist, and if it does it may not be unique. So what are the output data? Well, if there is no solution (for some particular values of the input data) then the map $f : X \to Y$ involved in Definition 1.1 is not defined at that particular input vector $x \in X$. A more difficult (conceptually) is the case of multiple solutions – what are the output data then? Well, in practical applications there is often a "desired" root (or a few "desired" roots) of an equation (as compared to many other roots which, while solving the equation $f(x) = 0$, have no practical significance – we call them "false roots"). The number of "desired" roots is usually well defined (often, just one). When practical considerations permit to distinguish the desired root from all the other roots, then we consider the desired root as the output data. This allows us to fit the topic of solving equations into the framework of Definition 1.1.

5.2 Practical remarks

When a function $y = f(x)$ is evaluated on a computer, a machine number $\tilde{y} = \tilde{f}(x)$ is obtained (as we explained earlier in the beginning of Section 3). Of course, $\tilde{y}$ here is generally different from $y = f(x)$ (hopefully, they differ by $\mathcal{O}(\varepsilon_{\text{machine}})$, but this is not always guaranteed). In particular, it is well possible that $f(x) = 0$ but $\tilde{f}(x) \neq 0$. Or vice versa, $\tilde{f}(x) = 0$ but $f(x) \neq 0$. Since the computer only deals with $\tilde{f}$ and has no clue about f, then how should it find a root x of the function f?

For example, the equation $f(x) = x^4 - 4x^3 + 6x^2 - 4x + 1 = 0$ has a unique root $x = 1$. The numerical version $\tilde{f}$ of this function f is plotted on page 79 of [2]. It shows that $\tilde{f}$

crosses the x axis many times (about 25 times on the plot in the book) and has multiple zeroes. Textbook [2] correctly states that in this example *any* number in the interval $(0.981, 1.026)$ could be taken as a good approximation to the true root. Numerically, all the numbers in this interval "solve" the original equation equally well, and no algorithm can (or should) find the root any more accurately.

The trouble in the above example is, of course, caused by the derivatives of f that vanished at the root.

On the other hand, when df/dx is large or infinite at a root (as, for instance, happens with the function $f(x) = (x - \pi)^{1/3}$ at the point $x = \pi$), then the function may cross the x axis too rapidly without even taking on a zero value or any sufficiently small value (say, of order $\varepsilon_{\text{machine}}$).

A very weird example of that sort is provided by the formula

$$f(x) = 3x^2 + \frac{1}{\pi^4} \ln\left[(\pi - x)^2\right] + 1 \tag{2}$$

see Section 3.0 in [3]. This function is smooth everywhere except $x = \pi$. Theoretically, it takes on all real values and has two distinct zeroes, both very close to π. However, if one plots this function by a graphical calculator or a computer software such as MAPLE, one would only see a parabola-like curve lying entirely above the line $y = 1$. In fact, the function dips below zero only in the ridiculously small interval of about $x = \pi \pm 10^{-667}$ (see Section 9.1 of [3] and a remark below). How would you detect such an interval (or a root, for that matter) on a computer whose machine precision is, say, 10^{-16}?

[In Fall 2002, our student Gaston Brouwer investigated the above function by using MAPLE and found the textbook formula (2) somewhat inaccurate. He considered the function

$$f_c(x) = 3x^2 + \frac{1}{\pi^4} \ln\left[(\pi - x)^2\right] + c$$

with a variable c. Then he writes

$$f_c(\pi + \varepsilon) = 3\pi^2 + 6\pi\varepsilon + 3\varepsilon^2 + \frac{2}{\pi^4} \ln|\varepsilon| + c$$

and for ε very small he obtains

$$f_c(\pi + \varepsilon) \approx \frac{2}{\pi^4} \ln|\varepsilon| + 3\pi^2 + c$$

Furthermore

$$\frac{2}{\pi^4} \ln|\varepsilon| + 3\pi^2 + c = 0 \quad \Longleftrightarrow \quad \varepsilon = \pm e^{-\pi^4(3\pi^2+c)/2}$$

With MAPLE he finds:

$$\begin{aligned} e^{-\pi^4(3\pi^2+c)/2} &\approx 0.362 \cdot 10^{-647} \quad \text{for } c = 1 \\ e^{-\pi^4(3\pi^2+c)/2} &\approx 0.255 \cdot 10^{-668} \quad \text{for } c = 2 \end{aligned}$$

Therefore, the free term in (2) should be equal to 2, rather than 1. This is an excellent example of numerical analysis!]

5.3 Machine approximations to solutions

The moral of the above examples is that the computer should classify x as a root whenever $\tilde{f}(x)$ is small (of order $\mathcal{O}(\varepsilon_{\text{machine}})$) or whenever the plot of $\tilde{f}$ crosses the x axis. So we arrive at *numerical criteria* for finding roots of equations: we call x a *good approximation to a root* of an equation $f(x) = 0$ if *either* of the following conditions holds:

- $|\tilde{f}(x)| \leq C\varepsilon_{\text{machine}}$.
- there are two nearby numbers $x' < x < x''$ such that $|x'' - x'|/|x| \leq C\varepsilon_{\text{machine}}$ and $\tilde{f}(x')$ and $\tilde{f}(x'')$ have opposite signs (or one of these values is zero).

Here C should be of order 10^k with some small k (such as $k = 1, 2, 3$). In fact, one can play with C to adjust the performance of computer algorithms. Setting C to a small value $C \approx 10$ makes the criteria very tight. While this may allow you to find a more accurate approximation to the true root in some cases, it may turn out "too tight" in some other cases, so that the computer will just fail to find any root! On the other hand, setting $C \approx 10^2$ or even $C \approx 10^3$ should guard against failures in most cases, but the the obtained approximation to a root may not be the best possible.

5.4 Bracketing

The second criterion in 5.3 leads to the following basic principle:

> If $\tilde{f}(a)$ and $\tilde{f}(b)$ for some $a < b$ have opposite signs, then there is at least one good approximation to a true root between a and b.

Here we do not suppose that a and b are close to each other or that $\tilde{f}(a)$ and $\tilde{f}(b)$ are small. This is a general principle. It is one of a few facts in numerical analysis that guarantee the existence of a solution to a problem. Textbook [3] strongly recommends that one holds on to this principle and includes it (as a safety check) in all computer programs for solving equations.

It is common to replace $\tilde{f}$ by f in the above principle (and other rules discussed below), assuming that the corresponding values are close enough. However, strictly speaking, the above principle is not always valid when one substitutes f for $\tilde{f}$.

5.5 Bisection (interval halving) method

The simplest algorithm for solving equations in one variable is based one the above principle alone. It is called bisection (or interval halving) method, see 3.1 in [2] or 9.1 in [3]. We only record here some important features and practical aspects of this method:

- The method converges linearly. More precisely, if x_n is the n-th approximation to the root x found by this method (i.e., x_n is the midpoint of the n-th interval), then $|x_n - x| \le C/2^n$, where C is the length of the initial interval. Linear convergence is slow. As a rule, it takes 20–25 iterations to approximate a root in single precision and 50–55 iterations in double precision, cf. 4.5.

- In practical implementations, it is advisable to set a limit on the number of iterations, i.e. to terminate the program if the number of iterations exceeds a preset maximum value of, say, 60. This prevents the program from accidentally entering an infinite cycle which results in "freezing" the computer.

- In order to determine whether $f(a)$ and $f(b)$ have opposite signs, one *should not* multiply these numbers. Not only is the multiplication rather expensive, computationally, but it can result in underflow or overflow breaking down the execution of your program. A safer (and possibly cheaper) way is to compare $\text{sgn}(f(a))$ and $\text{sgn}(f(b))$, where **sgn** stands for a function extracting the sign of a real number, which is available in many computer languages.

- To find the midpoint of an interval (a, b) one *should not* compute $(a + b)/2$. There are documented examples where the so computed number lies *outside* of the interval $[a, b]$, due to round-off errors of addition, see p. 81 in [2]. A safer way is to keep record of the half interval length $e = (b - a)/2$ and then compute the midpoint by $a + e$. The half-length e can be easily updated at each iteration (we simply divide it by two).

5.6 The Newton method

A classical method for finding zeroes of differentiable functions is the Newton method (also called the Newton-Raphson method). See Section 3.2 in [2] and Section 9.4 in [3] for detailed descriptions with examples. We only record here some important features and practical aspects of this method:

- The method requires a starting point x_0 (called, colloquially, an *initial guess*), which must be provided by the user (i.e., by you). The performance of the method (and the final result) would usually depend on the point x_0 you provide, so beware! If x_0 is close enough to the desired root, the method will quickly find it, otherwise it may wander around forever, or converge to a false root, or dash to infinity (diverge), sometimes in spectacular ways, see examples in [2] and [3].

- When you select a good starting point x_0 making the method converge to the desired root, the speed of convergence is usually quadratic. More precisely, if x_n is the n-th approximation to the root x found by this method, then the errors $e_n = |x_n - x|$ are related by
$$e_{n+1} \simeq C e_n^2$$
where $C \approx f''(x)/2f'(x)$. Hence, the convergence is quadratic whenever $f'(x) \neq 0$.

- The quadratic convergence of the Newton method means that the number of accurate digits in the root approximation double at every iteration, see 4.6 and an example on p. 92 of [2]. Normally, 5-6 iterations are sufficient to achieve a maximal possible accuracy in double precision. This is very fast. The rapid convergence constitutes the main virtue of the algorithm. (Still, it is advisable to set a limit on the number of iterations to, say, 20, which would prevent the program from being accidentally trapped in an infinite cycle.)

- Despite the well known facts about possible divergence of the Newton method, it is widely applied *as is*, without any safety features or checkpoints. Occasional cases of divergence are attributed to a bad choice of the initial guess x_0, and one simply restarts the method from another initial point.

- There are, however, simple ways to guard against possible divergencies and failures. At the very least, make sure that $|f'(x_n)| > \varepsilon$ with some tolerance $\varepsilon > 0$ before dividing by $f'(x_n)$. Using bracketing in some way would be a good idea, too, see Section 9.4 in [3]. Textbook [3] suggests several hybrids of rapidly convergent but unsafe algorithms with the safe but slow bisection method (see Sections 9.3, 9.4 and 9.7 in [3]).

- Yet another way of controlling the convergence in the Newton method is based on the minimization of $[f(x)]^2$, it will be discussed below, in 5.22.

- When the derivative $f'(x_n)$ is evaluated inaccurately (which might happen by mistake or by design, see below), the Newton method may still converge but slow down dramatically – the convergence is frequently reduced from quadratic to linear. There are many attempts in various applications to save on (often expensive) computation of $f'(x_n)$ and approximate the derivative somehow. It often pays off but the resulting slowdown of convergence should be taken into account. A more detailed account of this issue is given in [3].

- When the derivative f' is not available, it is rather tempting to approximate $f'(x)$ by a finite difference scheme, such as

$$f'(x) \approx \frac{f(x_1) - f(x_2)}{x_1 - x_2}$$

  for some $x_1 \leq x \leq x_2$, and then use the Newton method anyway. This is *not advisable*. The resulting round off errors and/or truncation errors are usually too big and spoil the entire process, see [3].

- If $f'(x) = 0$ (which happens, for example, at the so called multiple roots of polynomials), then the convergence of the Newton method is slow (most likely, linear). In this case other, more specialized methods work better, but they are too special to be included in our course.

- There are some instances when the convergence of the Newton method is guaranteed theoretically – for example, when the function $f(x)$ is convex, see Theorem 2 on page 91 in [2]. Similar statements can be made about concave functions. If such a guarantee is available, one does not need additional safety features and can rely on the barebone Newton method – the fastest algorithm for solving equations.

As an exercise, state a theorem about the convergence of the Newton method for a function $f(x)$ concave on an interval (a, b) (no need to provide a proof).

5.7 Exercises

Do problems 6 and 14 on pages 96–97 of [2].

5.8 Remark

The Newton method for solving equations is approximative by design. It finds a sequence of numbers x_n that presumably converges to the root but normally *never reaches* the root. One might think that exact solutions, when available, are more accurate numerically. This is frequently wrong. To demonstrate the triumph of iterative methods over exact formulas, we will solve a simple quadratic equation

$$x^2 + px + q = 0$$

The following numerical experiment can be done on any PC. Let one of the roots be $x_1 = 1/3 = 0.333\ldots$ and define the other root to be $x_2 = x_1 + d$, where d is an exponential random variable with mean μ. One can generate a value of d (we plan to discuss random number generators later) and compute $x_2 = x_1 + d$. Then one computes the coefficients $p = -(x_1 + x_2)$ and $q = x_1 x_2$. After that, one can solve the above equation by any method and find the smaller root x_1 (which is exactly equal to $1/3$) numerically. Let us compare three particular methods:

(E1) theoretically exact but numerically unstable formula (recall Example 3.9)

$$x = \frac{-p - \sqrt{p^2 - 4q}}{2}$$

(E2) theoretically exact and numerically stable formula (recall Example 3.9)

$$x = \frac{2q}{-p + \sqrt{p^2 - 4q}}$$

(NM) Newton method starting at the point $x_0 = x_1 - d$ (a symmetric image of the other root across the root x_1) and making exactly 8 iterations (for simplicity).

The following table presents the average error (the average distance of the numerically computed solution from the true value of $x_1 = 1/3$) over 10^6 trials, for each of these three methods, as μ increases:

μ	E1	E2	NM
10^2	3×10^{-15}	2×10^{-17}	2×10^{-18}
10^4	3×10^{-13}	2×10^{-17}	2×10^{-18}
10^6	3×10^{-11}	2×10^{-17}	2×10^{-18}
10^8	3×10^{-9}	2×10^{-17}	2×10^{-18}
10^{10}	3×10^{-7}	2×10^{-17}	2×10^{-18}

One can see that the accuracy of the exact but unstable algorithm E1 deteriorates steadily as $\mu \to \infty$. The exact stable solution E2 and the Newton method NM remain very accurate. But the Newton method somehow happens to be 10 times more accurate than the exact solution...

5.9 Secant method and false position method

Two more algorithms covered in many numerical courses are the secant method and the false position method. See Section 3.3 in [2] and 9.2 in [3]. We only make a few comments here:

- These methods do not require the derivative of $f(x)$ but they somehow manage to direct the search for a root along the slope of f. In that, they are simpler than the "sophisticated" Newton method but more clever than the "dumb" bisection.

- These two methods neither converge as fast as the Newton method nor enjoy the guaranteed linear convergence of the bisection method.

- When the secant method converges, the convergence is of order

  $$\alpha = \frac{1+\sqrt{5}}{2} \approx 1.62$$

  Since $\alpha > 1$, the convergence is superlinear. But $\alpha < 2$, so the convergence is slower than quadratic. Therefore, in terms of speed, this method is somewhere between the bisection and the Newton. The secant method, just like the Newton, is prone to divergence and failures.

- The speed of convergence for the false position method cannot be estimated theoretically, its order may be anywhere between 1 and 2, i.e. between quadratic and linear, depending on the particular function f. Moreover, when the convergence is linear, it can be arbitrarily slow, i.e. it can be characterized by $e_{n+1} \simeq \rho e_n$ with ρ arbitrarily close to one. So the false position method may be even slower than the bisection. When applying the false position, you never know if you are lucky to converge fast or if it will take almost forever.

Overall, these two methods may occasionally provide good alternatives to the classical Newton and bisection, but they have many limitations and pitfalls.

5.10 Exercise
Do problem 7 on page 105 of [2].

5.11 Fixed-point method
Here is probably the simplest and most naive method for solving equations. Suppose we have an equation $f(x) = 0$. First, let us somehow transform it into the form $x = g(x)$. Then, starting at some point x_0, we apply the iterative scheme $x_{n+1} = g(x_n)$.

Assuming that the sequence x_n converges to some limit $x_* = \lim x_n$, it is easy to see that $x_* = g(x_*)$, provided g is continuous at x_*, hence x_* is a solution of $f(x) = 0$. What can be simpler?

5.12 Examples

(a) To solve a quadratic equation

$$x^2 - 3x + 2 = 0 \tag{3}$$

one can "split off" one x and rewrite it as $x = x^2 - 2x + 2$, then use the iterative scheme $x_{n+1} = x_n^2 - 2x_n + 2$.

(b1) To solve a quadratic equation

$$2x^2 + x - 1 = 0 \tag{4}$$

one can separate x from the other terms as $x = 1 - 2x^2$, then use the iterative scheme $x_{n+1} = 1 - 2x_n^2$.

(b2) Obviously, one can rewrite the same equation in many ways so that it takes form $x = g(x)$. The previous equation (4) can be written as $x = (1 - 2x^2 + x)/2$, and then one can use the iterative scheme $x_{n+1} = (1 - 2x_n^2 + x_n)/2$ to solve it. Thus, we now have two schemes for solving the same equation. How do they compare? We will see below.

(b3) Yet another modification of the same equation (4) is $x = (2x^2 - 1 + 10x)/9$, which leads to the iterative scheme $x_{n+1} = (2x_n^2 - 1 + 10x_n)/9$. Is it any better than the previous two? We will see below.

The crucial question here is: does a given fixed point scheme converge to the desired root? The answer is given by the following theorem:

5.13 Fixed point theorem
Suppose x_* is a fixed point of a function g, i.e. $x_* = g(x_*)$. Then:

(a) if $|g'(x_*)| < 1$, then x_* is a stable fixed point. The iterative scheme $x_{n+1} = g(x_n)$ converges to x_* provided the initial guess x_0 is close enough to x_*.

(b) if $|g'(x_*)| > 1$, then x_* is an unstable fixed point. The iterative scheme $x_{n+1} = g(x_n)$ never converges to x_*.

(c) if $|g'(x_*)| = 1$, then x_* is a neutral point. The iterative scheme $x_{n+1} = g(x_n)$ may or may not converge to x_*, depending on more subtle factors such as $g''(x_*)$.

There are other versions of this theorem based on contraction principle, see p. 108 of [2].

5.14 Remarks

If $x_n \to x_*$, the convergence is linear: $|x_{n+1} - x_*| \simeq \lambda |x_n - x_*|$ with $\lambda = |f'(x_*)|$. In one exceptional case, when $f'(x_*) = 0$, the convergence is superlinear (most likely, quadratic).

The moral of this theorem: one should rewrite a given equation $f(x) = 0$ as $x = g(x)$ in a clever way, so that g' will be as small as possible at the desired root. At least, $|g'|$ must be less than one, so that the convergence will be possible at all.

5.15 Examples

In Example 5.12(a), one root is $x = 1$, and at that root $g' = 0$, hence the convergence to $x = 1$ is superlinear. The other root is $x = 2$ and there $g' = 2$, so the point $x = 2$ is unstable. One can easily verify that the fixed point iterations converge to $x = 1$ from any starting point $x_0 \in (0, 2)$. If $x_0 < 0$ or $x_0 > 2$, then the iterations diverge to infinity.

Equation (4) has roots $x = -1$ and $x = 0.5$. For the fixed point scheme in Example 5.12(b1), both roots are unstable, so there is no convergence to either root. In fact, when the initial point x_0 is chosen randomly in $(-1, 1)$, then with probability one the sequence x_n will be dense in the interval $(-1, 1)$ and will have a chaotic behavior. This fact is known in the theory of dynamical systems and covered in MA 760.

However, equation (4) admits fixed point schemes that converge to its roots. The scheme in 5.12(b2) converges to the root $x = 0.5$ (which is then stable), and the one in 5.12(b3) converges to the other root $x = -1$.

5.16 Exercise

Do problem 7 on page 113 of [2].

5.17 Systems of equations: generalities

We now turn to equations in several variables, expressed by $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{x} \in \mathbb{R}^n$ is an independent vector and $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ a given map. One normally requires that $m = n$ (the number of equations equals the number of unknowns). For example, if $m = n = 2$, one has a system of two equations with two unknowns.

First of all, there is a bad news for us. There is no analogue of the safest bisection method, and no analogue of any kind of bracketing, see a discussion in 9.6 of [3]. Thus, solving systems of equations may be a much more unpleasant and frustrating task than solving one-variable equations. In many practical applications, one never feels safe and never knows if a solution exists in a given domain or not. No algorithm can guarantee that a root will be found. There are not many algorithms available to us in any case. Essentially, there are two general methods at our disposal: the fixed point method and the Newton method.

5.18 Fixed point method in several variables

Given an equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ with $m = n$, one may rewrite it as $\mathbf{x} = \mathbf{g}(\mathbf{x})$ with some $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$ and then apply the iterative procedure $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$.

This is simple, naive, but way too often used in practice...

5.19 Fixed point theorem in several variables

There is an analogue of Theorem 5.13 in several variables. However, its full version is very complicated, we only sketch here its main clauses. Let $\mathbf{x}_*$ be a fixed point of a function $\mathbf{g}$, i.e. $\mathbf{x}_* = \mathbf{g}(\mathbf{x}_*)$. Let $\mathbf{D} = \mathbf{Dg}(\mathbf{x}_*)$ be the derivative of $\mathbf{g}$ at the fixed point $\mathbf{x}_*$ (note: $\mathbf{D}$ is the $n \times n$ matrix of partial derivatives of all components of $\mathbf{g}$ with respect to all components of $\mathbf{x}$). Let $\lambda_1, \dots, \lambda_n$ be all the eigenvalues of the matrix $\mathbf{D}$.

(a) if $|\lambda_i| < 1$ for all i, then $\mathbf{x}_*$ is a stable fixed point. The iterative scheme $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ converges to $\mathbf{x}_*$ provided $\mathbf{x}_0$ is chosen close enough to $\mathbf{x}_*$.

(b) if $|\lambda_i| > 1$ for all i, then $\mathbf{x}_*$ is an unstable fixed point. The iterative scheme $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ never converges to $\mathbf{x}_*$.

(c) if $|\lambda_i| > 1$ for some i and $|\lambda_i| > 1$ for other i, then $\mathbf{x}_*$ is a saddle point. The iterative scheme $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ does not converge to $\mathbf{x}_*$, unless it starts in a special submanifold of $\mathbb{R}^n$ that is attracted to $\mathbf{x}_*$. This is very unlikely, and in a sense the probability of convergence is zero.

(d) If $|\lambda_i| = 1$ for all i, then $\mathbf{x}_*$ is a neutral point. The iterative scheme $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$ may or may not converge to $\mathbf{x}_*$, depending on more subtle factors such as the second derivative of $\mathbf{g}$ at $\mathbf{x}_*$.

Here we only listed principal cases, some other combinations of these cases may occur.

5.20 Practical remarks

The criteria of convergence listed in Theorem 5.19 are impractical, their verification is often out of the question (computing the eigenvalues of $\mathbf{D}$ is usually even more difficult than solving the equation itself). Therefore, one has to design a fixed point method intuitively, by trial and error method. The only reliable criterion of convergence is experimental testing.

5.21 Newton method in several variables

This method is more frequently called Newton-Raphson method. It is based on the iteration scheme

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{Df}(\mathbf{x}_k)]^{-1}\mathbf{f}(\mathbf{x}_k)$$

see a detailed description on pages 91–94 of [2] and in Section 9.6 of [3]. We only emphasize a few crucial points here:

- Practically, at each iteration one solves a system of linear equations

$$[\mathbf{Df}(\mathbf{x}_k)]\,\mathbf{p}_k = \mathbf{f}(\mathbf{x}_k)$$

  and then updates the approximation by $\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{p}_k$. Solving this system is usually done by the simplest method – Gaussian elimination (also called LU decomposition). There is no need to compute the inverse of the matrix $\mathbf{Df}(\mathbf{x}_k)$.

- One stops iterations whenever one of the three conditions holds: (the norm of) the function $\|\mathbf{f}(\mathbf{x}_k)\|$ is small, or (the norm of) the update $\|\mathbf{h}_k\|$ is small, or the number of iterations exceeds a predefined limit. The choice of the norm is up to the user. One of the best choices is the 1-norm, i.e. $\|\mathbf{f}\| = \sum_{i=1}^{n} |f_i|$.

- The convergence is, generally, quadratic. Precisely, it is quadratic if the matrix $\mathbf{Df}(\mathbf{x})$ is nonsingular.

- The Newton method requires an initial guess $\mathbf{x}_0$ supplied by the user, and the performance of the method is usually very sensitive to the choice of $\mathbf{x}_0$, recall our discussion in 5.6.

- When the derivative of $\mathbf{f}$ is evaluated inaccurately, the Newton method may still converge but slows down.

- When the derivative $\mathbf{Df}$ is not available, it is tempting to approximate it by some finite difference schemes, but this is not advisable. The resulting round off errors and/or truncation errors are usually too big and spoil the entire process, see [3].

5.22 Improving the Newton method by an ad-hoc control

The Newton method is powerful but sometimes "too powerful". It has an unfortunate tendency to "overshoot" – see illustrations in 9.4 of [3]. Since there is no bracketing in several variables, one has to use another way to control the Newton procedure and redirect it if it overshoots.

One such idea of an ad-hoc control is to require the norm of the function $\mathbf{f}$ decrease at every iteration. Let

$$h = \frac{1}{2}\,\|\mathbf{f}\|_2 = \frac{1}{2}\,\mathbf{f}\cdot\mathbf{f}$$

(the 1/2 is put for convenience) be the half 2-norm of the function $\mathbf{f}$. As long as the Newton iterations make h decrease, i.e. $h(\mathbf{x}_{k+1}) < h(\mathbf{x}_k)$, they are accepted. However, if $h(\mathbf{x}_{k+1}) \geq h(\mathbf{x}_k)$ for some k, then the point $\mathbf{x}_{k+1}$ is rejected, and some other rule is used instead to recompute $\mathbf{x}_{k+1}$.

5.23 Backtracking

What exactly do we do if a point $\mathbf{x}_{k+1}$ is rejected? In most cases, this happens when $\mathbf{x}_{k+1}$ lands too far beyond the root. In this case one should attempt to "step back" somewhat.

Let us restrict the function $\mathbf{f}$ to the line from the current point $\mathbf{x}_k$ to the next attempted point $\mathbf{x}_{k+1}$. Denote by $\mathbf{p} = \mathbf{x}_{k+1} - \mathbf{x}_k$ the corresponding vector and consider the function

$$g(\lambda) = h(\mathbf{x}_k + \lambda \mathbf{p})$$

Note that $g(0) = h(\mathbf{x}_k)$ and $g(1) = h(\mathbf{x}_{k+1})$, and since the point $\mathbf{x}_{k+1}$ was rejected, we have $g(1) > g(0)$. All we want is find a $0 < \lambda < 1$ such that $g(\lambda) < g(0)$.

This procedure is called the Newton method with *backtracking*.

Until the early 1970s, standard practice was to minimize $g(\lambda)$ by using some general methods (we will learn some of them later). But those methods are often computationally expensive. In this case they are just wasteful, since the precise minimization of $g(\lambda)$ gives very little advantage (remember, we will have to resume the Newton method from the new point $\mathbf{x}_k + \lambda \mathbf{p}$ anyway). There are simpler and faster ways to find *some* $\lambda \in (0, 1)$ such that $g(\lambda) < g(0)$.

5.24 A fast line search

Here is one such way, see Section 9.7 in [3]. It is easy to see that

$$g'(0) = \nabla h \cdot \mathbf{p} < 0$$

Therefore, if λ is small enough, then $g(\lambda) < g(0)$. But at the same time we do not want λ to be too small, since then the improvement would be too little.

A reasonable strategy is this. One uses the available values of $g(0)$, $g(1)$ and the derivative $g'(0)$ to approximate the function g by a quadratic polynomial $q(\lambda)$ such that $q(0) = g(0)$, $q(1) = g(1)$ and $q'(0) = g'(0)$ (such a polynomial is unique, by the way). The minimum of $q(\lambda)$ can be easily found, we call it λ_1. Note that since $q(1) > q(0)$, we have $0 < \lambda_1 < 0.5$.

It can happen again that $g(\lambda_1) > g(0)$. Then one proceeds similarly, approximating g by a quadratic or cubic polynomial on the smaller interval $(0, \lambda_1)$ and minimizing that polynomial, until one finds λ such that $g(\lambda) < g(0)$. See details in 9.7 of [3].

5.25 Remarks

This additional control makes the Newton method safer and more stable. In fact, the method is forced to decrease the function h, hence it can no longer wander off or jump back and forth periodically. Does this enforce convergence to a root of the given equation? Not quite. The method actually tries to minimize the function h, and it may converge to a local minimum of h. The book [3] claims that this is quite rare in practice, though, and suggests that if this does happen, one should simply restart the Newton method with a new initial point $\mathbf{x}_0$. The book calls the Newton method with backtracking, somewhat ambitiously, a *globally convergent method for nonlinear systems of equations.*

6 Minimization of Functions

6.1 Introduction

In the end of the previous chapter (Sections 5.22–5.25), we have seen an instance of the so called *minimization problem*: given a function $F : \mathbb{R}^n \to \mathbb{R}$, one wants to find its minimum $F_{\min}$ and the point (or points) where F attains its minimum:

$$x_{\min} := \operatorname{argmin} F(x)$$

Alternatively, one may want to *maximize* a function F, i.e. find its maximum and the point (or points) where the maximum is attained:

$$x_{\max} := \operatorname{argmax} F(x)$$

Of course, by simply changing F to $-F$ one can convert the maximization of F to the minimization of $-F$, and vice versa. So it is enough to discuss one of these two (essentially, equivalent) problems, and it is customary to restrict the discussion to the minimization problem.

In applications, one often minimizes losses, expenses, cost, labor, etc.. One also maximizes profit, output, production, etc. This class of practical problems is referred to as *optimization*. Mathematically, this means the minimization or maximization of a certain function F, which is called the *objective function* or *cost function*.

In what follows, we restrict the discussion to the minimization problem.

6.2 General remarks

Not all functions have minima. In order to have a minimum, the function F must be at least bounded below, i.e. $F(x) \geq c$ for some $c \in \mathbb{R}$ and all x. In that case F, at least, has an infimum, but not necessarily a minimum. For example, the function $F(x) = (1 + x^2)^{-1}$ has $\inf F = 0$, but it does not have a minimum. That is, for every x there is another x' such that $F(x') < F(x)$. (This sounds all too trivial... until you run into such a function practically. Suppose your boss wants you to minimize this function for an applied problem, where you have to produce numerical values of $F_{\min}$ and $x_{\min}$, which are at least approximately correct; you obviously conclude that $F_{\min}$ should be set to 0, but which point would you present as an approximation to $x_{\min}$?)

Now we assume that a function F has a minimum $F_{\min}$. In many applications, it is not the value of $F_{\min}$, but rather the point $x_{\min} = \operatorname{argmin} F(x)$, which is of practical interest. Such a point exists, but may not be unique. The function F may attain its minimum at more than one point, for example, the function $y = \cos x$ has a minimum $y_{\min} = -1$ attained at points $x_{\min} = \pi + 2k\pi$, $k = 0, \pm 1, \pm 2, \ldots$. In this case we have so called *multiple minima*. Often any of them would give a satisfactory solution to a practical problem, but some confusion may arise.

Now assume, for simplicity, that F has a unique minimum, i.e. there is a unique point $x_{\min} = \operatorname{argmin} F(x)$. How do we find it? It may happen that there are other points $x' \neq x_{\min}$ where the function F has *local* or *relative* minima (as opposed to the *global* or *absolute* minimum at $x_{\min}$). Local minima may **not** be what we want to find, our objective is presumably the global minimum. But many numerical algorithms work "locally", examining values of the function F at selected points and their neighbors, as we will see below. Hence, numerical algorithms are very likely to find a local minimum of the objective function, but not necessarily a global one. See an illustration in Section 10.0 of [3]. There is, generally, no way a numerical algorithm can "see the entire function" $F(x)$, so it has no way of distinguishing a local minimum from a global one. Some recent algorithms attempt to do just that, see Simulated Annealing in Section 10.9 of [3], but they are quite complicated and slow (besides, they may also converge to a local minimum).

Most of the minimization algorithms are, therefore, designed to find a local minimum of the objective function. If this is not satisfactory in a particular application, some extra work needs to be done to find a global minimum, but this is usually not the responsibility of the algorithm per se. We will only discuss here algorithms that aim at finding a local minimum of a given function.

6.3 Warning

We have outlined several difficulties that may arise in a minimization problem. Now we give a mild warning. One might suggest a purely "numerical" (and unfortunately, almost useless) solution to a minimization problem as follows. A function $F(x)$ has its "computer implementation" $\tilde{F}(x)$, see the beginning of Section 3. The "numerical version" $\tilde{F}$ of the function F is defined on machine numbers and produces machine numbers. So, it has a finite domain and a finite range! Therefore, it always has a minimum. And one may now attempt to find that minimum $\tilde{F}_{\min}$ and the machine number

$$\tilde{x}_{\min} = \operatorname{argmin} \tilde{F}$$

at which it is attained. This is a bad idea. First of all, the amount of computations needed to find $\tilde{F}_{\min}$ and $\tilde{x}_{\min}$ is prohibitively high. Besides, *that* minimum is *not* what we want. We want a good approximation to an actual (global or local) minimum point $x_{\min}$. Such an approximation can be found much faster than $\tilde{F}_{\min}$ and $\tilde{x}_{\min}$. Our first question is this: when we say "good approximation", then how good is good?

6.4 Stopping Rule

It is common to assume that the function $F(x)$ is smooth, and hence its first derivative (gradient) vanishes at a local minimum $x_{\min}$. Therefore, it has a quadratic behavior

$$F(x_{\min} + dx) = F(x_{\min}) + \mathcal{O}(\|dx\|^2)$$

Since the function cannot be possibly computed more accurately than to within $\varepsilon_{\text{machine}}$, the argument $x_{\min}$ cannot be estimated more accurately than to within $\mathcal{O}(\sqrt{\varepsilon_{\text{machine}}})$. It is then common to adopt the following stopping rule for iterative minimization algorithms:

one stops iterations $\{x_n\}$ once

$$\|x_{n+1} - x_n\| \le \varepsilon_{\text{tolerance}} \approx \mathcal{O}(\sqrt{\varepsilon_{\text{machine}}}) \tag{5}$$

The book [3] recommends to set the value of $\varepsilon_{\text{tolerance}}$ to 10^{-4} in single precision and 3×10^{-8} in double precision. These requirements are significantly lower than those we imposed in Chapter 5 when solving equations. But there is no need to make the rule (5) more stringent – it is dictated by the smoothness considerations.

We now consider a one-variable minimization problem: minimize a function $F : \mathbb{R} \to \mathbb{R}$.

6.5 Golden section

It is remarkable that the one-variable minimization problem admits *bracketing* of the minimum of F, similar to the bracketing of a root in Section 5.4. One needs three points $a < b < c$ such that $F(b) < F(a)$ and $F(b) < F(c)$. If such points are found, there is a local minimum of F between a and b, as one can easily see. This simple fact can be used to restrict the search to the interval (a, c) and somehow narrow it down to pinpoint the local minimum. (In a way similar to the bisection algorithm for solving equations described in Section 5.5.)

In fact, there exists an analogue of the bisection method, which is called the *golden section* algorithm. It is described in 10.1 of [3]. We only make a few comments here:

- The method converges linearly. More precisely, if x_n is the n-th approximation to the minimum $x_{\min}$ found by this method, then $|x_n - x_{\min}| \le C\lambda^n$, where $\lambda = (\sqrt{5}-1)/2 \approx 0.618$ and $C > 0$ is the length of the initial interval $c-a$ bracketing the minimum. This convergence is a little slower than the convergence of the bisection method, where we had $|x_n - x_*| \le C\lambda^n$ with $\lambda = 0.5$. On the other hand, our stopping rule (5) is less restrictive than the stopping rule used in solving equations. Generally, we need about 20 iterations in single precision and 40–45 iterations in double precision to estimate a local minimum.

- The golden section method is absolutely stable and reliable. It cannot possibly fail, and it will always produce a local minimum of F in due course.

Below we mention some modifications of the golden section designed to speed up the convergence.

6.6 Brent method

This is a rather complicated and tricky algorithm based on the golden section and the idea of the secant method mentioned in Section 5.9. It is described in Section 10.2 of [3]. Its convergence is linear in worse cases and superlinear in better cases.

6.7 One-dimensional search with first derivatives

The golden section method and Brent method do not require the evaluation of the function's derivative. However, if we *can* evaluate it, we may want to use it to our advantage and improve the convergence. A particular algorithm of this sort is described in Section 10.3 of [3]. Its convergence again varies between linear and superlinear.

6.8 The Newton method

Another powerful method can be used if the *second derivative* of the objective function F is available. Then one can find the minimum by Newton's iterations

$$x_{n+1} = x_n - F'(x_n)/F''(x_n) \tag{6}$$

This is a classical Newton algorithm for minimizing smooth functions. Many remarks we made about the Newton method in Section 5.6 apply also to (6). In particular, the convergence is quadratic if the initial guess is chosen closely enough to a local minimum of F. We make additional important remarks here:

6.9 Remarks

Newton's method (6) is actually designed to solve the equation $F'(x) = 0$ rather than find a local minimum of F. Thus, it is just as likely to converge to a local maximum or a saddle point. More precisely, if $F'' > 0$, the iterations approach a local minimum, but if $F'' < 0$ they move toward a local maximum. Indeed, note that the Newton method is based on the approximation of the function F by a quadratic polynomial $P(x) = ax^2 + bx + c$ that agrees with F at the point x_n up to the second derivative, i.e. such that $P(x_n) = F(x_n)$, $P'(x_n) = F'(x_n)$, and $P''(x_n) = F''(x_n)$ (in this case one says that the parabola $y = ax^2 + bx + c$ is an *osculating curve* to the function $y = F(x)$). Then x_{n+1} is simply the vertex (extremum) of the polynomial $P(x)$. Now, it is easy to check that x_{n+1} is the minimum of P when $F''(x_n) > 0$ and the maximum of P when $F''(x_n) < 0$. Therefore, the Newton method logically leads to a local minimum of F when $F''(x_n) > 0$ and to a local maximum of F when $F''(x_n) < 0$.

6.10 An ad-hoc control

In 5.22 we discussed an ad-hoc control improving the behavior of the Newton method for solving equations. We now adapt it to the minimization problem. The control is based on the following ideas:

First of all, we see that the condition $F''(x_n) > 0$ is essential for convergence to a minimum and should be enforced. Second, we will require that the values of F decrease from iteration to iteration. That is, a Newton step should be accepted if $F(x_{n+1}) < F(x_n)$ and rejected otherwise. In the case of rejection, one needs to reduce the step, i.e. move x_{n+1} closer to x_n. This is equivalent to increasing $F''(x_n)$ in (6).

6.11 A scheme for improving the Newton method

The following practical algorithm combines the principles outlined above. It uses an additional (control) parameter $\lambda > 0$. The Newton step (6) is redefined to be

$$x_{n+1} = x_n - \frac{F'(x_n)}{F''(x_n) + \lambda} \tag{7}$$

It is accepted under two conditions:

$$F''(x_n) + \lambda > 0 \qquad \text{and} \qquad F(x_{n+1}) < F(x_n) \tag{8}$$

If either one fails, we reject x_{n+1}, increase λ (say, by multiplying it by a large constant factor, such as 10) and recompute x_{n+1}. We keep increasing λ until the above two conditions hold. This is bound to happen since for very large λ we will be making a tiny step in the direction where $F(x)$ decreases.

When both conditions (8) hold, we accept the new approximation x_{n+1} and decrease λ (by multiplying it by a small constant factor, such as 0.1 or even 0.01). Then we proceed to the next iteration.

Note that when the iterations x_n are far from a local minimum, the algorithm (7) is forced to make smaller steps in the direction where F decreases (this prevents aimless wandering so characteristic of the classical Newton method (6)). Once the iterations x_n get close enough to a local minimum, λ will rapidly decrease and the algorithm will speed up to the minimum at a quadratic speed.

Now we discuss the multivariate minimization problem of minimizing a function $F : \mathbb{R}^d \to \mathbb{R}$, where $d \geq 2$.

6.12 The Newton method in several variables

The Newton method 6.8 and its improvement 6.11 extend to multivariate functions almost without changes. We only highlight the differences here. The derivative F' is now replaced by the gradient vector $\nabla F = (\partial F/\partial x_1, \dots, \partial F/\partial x_d)$, and the second derivative F'' is replaced by a $d \times d$ matrix (called the Hessian matrix)

$$\mathbf{H} = \left(\frac{\partial^2 F}{\partial x_i \partial x_j} \right), \qquad 1 \leq i, j \leq d$$

The Newton step is expressed by

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H}(\mathbf{x}_n)]^{-1} \nabla F(\mathbf{x}_n)$$

Note: in practice, we do not need to invert the Hessian matrix, but rather solve the system of linear equations $\mathbf{H}(\mathbf{x}_n)\,\mathbf{u} = \nabla F(\mathbf{x}_n)$ for $\mathbf{u}$. Since the Hessian matrix $\mathbf{H}(\mathbf{x}_n)$ is symmetric (and should also be positive definite, see below), we can apply fast and stable routines such as Cholesky decomposition to solve this system and find $\mathbf{u}$.

An improvement of the Newton method uses an additional control parameter $\lambda > 0$. The modified Newton step now looks like

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H}(\mathbf{x}_n) + \lambda I]^{-1} \nabla F(\mathbf{x}_n) \tag{9}$$

where I is the identity matrix. We accept $\mathbf{x}_{n+1}$ if

$$F(\mathbf{x}_{n+1}) < F(\mathbf{x}_n) \qquad \text{and} \qquad \mathbf{H}(\mathbf{x}_n) + \lambda I > 0$$

where the last inequality means that the matrix $\mathbf{H}(\mathbf{x}_n) + \lambda I$ is positive definite.

We note that as λ increases, the displacement vector $\mathbf{x}_{n+1} - \mathbf{x}_n$ not only shrinks but also aligns in the direction opposite to the gradient vector $\nabla F(\mathbf{x}_n)$. This guarantees that for sufficiently large λ we have $F(\mathbf{x}_{n+1}) < F(\mathbf{x}_n)$. In other aspects, the method described in 6.9–6.11 remains unchanged in any dimension.

The above scheme (9) using the control parameter λ is known as Levenberg-Marquardt correction to the Newton method.

6.13 Steepest descent

Another method for minimizing functions of several variables is called the steepest descent. Here one takes a current approximation $\mathbf{x}_n$ and restricts the function F to the line passing through $\mathbf{x}_n$ in the direction of to the gradient vector $\nabla F(\mathbf{x}_n)$. One gets a one variable function

$$f(\eta) = F(\mathbf{x}_n - \eta\,\nabla F(\mathbf{x}_n))$$

where $\eta > 0$ is a new variable (compare this to Section 5.23). Then one finds the minimum of $f(\eta)$ by using any of the one-dimensional search schemes (e.g., the golden section, or Brent method, or others) and takes the point $\mathbf{x}_{n+1} = \mathbf{x}_n - \eta\,\nabla F(\mathbf{x}_n)$ as the next approximation.

This method is very old and known to be quite slow. Not only it requires expensive minimization of a one variable function at *every* iteration, but also the number of iterations is very likely to be quite high in typical cases, see an illustration in Section 10.5 of [3]. This method is therefore not recommended in most cases.

6.14 Final remarks

Two more minimization algorithms, the *simplex method* and the *simulated annealing*, are not discussed in these notes. They are assigned as independent projects and will be presented in class by the students who select them.

We have now covered the basics of numerical analysis, except matrix computations, which are the subject of Applied Linear Algebra MA 660.

Next we move to applications of mathematics in real world. Two major areas where mathematics is traditionally (and extensively) applied to practical problems (in science, industry, business, medicine, etc.) are these:

- Modelling by differential equations and numerical solutions of such equations.
- Probabilistic modelling and using statistical methods to process experimental data.

In the next chapter we describe some statistical applications.

7 Statistical Data Processing

For those who are not familiar with statistics, the following toy example demonstrates its basic principles. (We assume, however, some knowledge of probability theory.)

7.1 A toy example

Suppose a few people are polled before an election day on their political affiliation. The results of the poll are recorded by a sequence

$$D\ R\ R\ D\ R\ R\ D\ R$$

where D stands for democrats and R for republicans. What can one conclude? Is there anything a mathematician can do with these experimental data?

First, we set up a probabilistic model. Assume that there is a large pool of voters, each supporting either democrats or republicans. This is our probability space. It is divided into two subsets (events), D and R. Denote the proportion of voters supporting democrats (the probability of D) by p and the proportion of voters supporting republicans (the probability of R) by $q = 1-p$. Next, assume that our eight persons have been selected randomly and independently for the poll. In other words, an independent sample of $n = 8$ points is taken from the probability space (one says that the sample has *size* 8). Each point may belong to D with probability p or to R with probability q. The probability of the entire sequence recorded in the poll is thus $pqqpqqpq = p^3q^5 = p^3(1-p)^5$.

Of course, p is unknown, so the probability $L = p^3(1-p)^5$ is a function of the unknown parameter p. It is called the *likelihood function*.

In statistics, one tries to estimate (predict) the unknown parameter(s). This can be done, for example, by maximizing the likelihood function. Such a method is called the *maximum likelihood estimation (MLE)*.

In our toy example, L has a unique maximum. To find it, we take logarithm of L

$$\ln L = 3\ln p + 5\ln(1-p)$$

then the maximum of L is found by a simple calculation

$$\frac{d\ln L}{dp} = \frac{3}{p} - \frac{5}{1-p} = 0$$

Solving this equation gives $p = 3/8$. This is an estimate of the unknown parameter p found from the available experimental data. To distinguish an estimate of p from the true (and still unknown) value of p, the estimate is denoted by $\hat{p}$, so in our case $\hat{p} = 3/8$.

7.2 Analysis of our toy example

Several important observations should be made at this point:

- The likelihood function L only depends on the number of D's and the number of R's. The order in which D's and R's were observed in the poll is irrelevant.

Furthermore, it is enough to know the number of D's (call it n_D), then the number of R's can be easily computed by $n - n_D$, where $n = 8$ is the size of the sample. Hence one can simplify the recording of experimental data: there is no need for keeping track of the entire sequence of D's and R's, it is enough to record the number of D's alone. Such a number is called a *sufficient statistic*.

- The likelihood function L depends on the unknown parameter(s). The point in the parameter domain where L is maximized is called the *maximum likelihood estimate* (MLE). In almost all practical applications, the MLE is the best possible estimate of the unknown parameters, see below.

- For the practical maximization of L, it is often convenient to take its logarithm first, and then differentiate with respect to the parameter(s).

7.3 A more serious example

Suppose one has several experimentally obtained real numbers $x_1, x_2, \ldots, x_n$ representing some random quantity. What can one conclude?

Again, we need to set up a probabilistic model. Usually, the numbers $x_1, \ldots, x_n$ are obtained from the same experiment that was repeated n times. Then $x_1, \ldots, x_n$ are values of a random variable obtained independently. If the random variable has density function $f_\theta(x)$, where $\theta \in \mathbb{R}^k$ is a parameter (vector), then the joint density function of our n values $x_1, \ldots, x_n$ is $L(\theta) = f_\theta(x_1) f_\theta(x_2) \cdots f_\theta(x_n)$. Again, this is called the likelihood function corresponding to the given sample $x_1, \ldots, x_n$.

[A remark on terminology: In probability theory, $f_\theta(x_1) \cdots f_\theta(x_n)$ is called a joint probability density function. It depends on $x_1, \ldots, x_n$, while θ is not regarded as its argument. In statistics, $x_1, \ldots, x_n$ are known (from experiment), and θ is a primary goal, so the same function now depends on θ only and goes by a different name – the likelihood function.]

Next, in most cases of statistical analysis, the probability distribution is assumed to be normal (gaussian). Probability theory provides a basis for this assumption – the central limit theorem. Later we discuss some other distributions.

Assuming that $x_1, \ldots, x_n$ are normal random variables with mean μ and variance σ^2 (these are our parameters) gives

$$L(\mu, \sigma^2) = \left(\frac{1}{2\pi\sigma^2}\right)^{n/2} \exp\left[-\frac{\sum (x_i - \mu)^2}{2\sigma^2}\right]$$

Taking logarithm gives

$$\ln L = -\frac{n}{2} \ln(2\pi\sigma^2) - \frac{\sum (x_i - \mu)^2}{2\sigma^2}$$

It is convenient to modify this expression by noticing that

$$\sum (x_i - \mu)^2 = \sum x_i^2 - 2\mu \sum x_i + n\mu^2$$

The values $M_1 = \sum x_i$ and $M_2 = \sum x_i^2$ are called the first and second moments of the data sample, respectively. Then we obtain

$$\ln L = -\frac{n}{2}\ln(2\pi\sigma^2) - \frac{M_2 - 2\mu M_1 + n\mu^2}{2\sigma^2}$$

Since the likelihood function L only depends on M_1 and M_2, they are our sufficient statistics. We no longer need to know the entire sample $x_1, \dots, x_n$, it is enough to store the two moments M_1 and M_2. [Furthermore, these moments can be conveniently computed "on-line", as the data arrives: after receiving an experimental number x_i one updates the moments by the obvious rules $M_1 = M_1 + x_i$ and $M_2 = M_2 + x_i^2$, and then x_i can be discarded (erased from the computer memory).]

The unknown parameters μ and σ^2 can be estimated by maximizing the likelihood function. A direct calculation gives

$$\hat{\mu} = \frac{M_1}{n} = \frac{1}{n}\sum x_i$$

and

$$\hat{\sigma}^2 = \frac{M_2}{n} - \left(\frac{M_1}{n}\right)^2 = \frac{1}{n}\sum (x_i - \hat{\mu})^2$$

7.4 Remark

One can estimate unknown parameters in many different ways. For example, μ can be estimated by any of these formulas:

$$\hat{\mu}_1 = \frac{x_1 + x_2}{2}, \qquad \hat{\mu}_2 = \frac{x_{\max} + x_{\min}}{2}, \qquad \hat{\mu}_3 = x_{\text{med}}, \qquad \hat{\mu}_4 = x_1^{100} - x_2^3$$

here x_{med} is the median of the sample. One may argue that the first three formulas make some sense, while the last one is completely ridiculous. Generally, however, any function $g(x_1, \dots, x_n)$ may be called an estimate of an unknown parameter, see next.

7.5 Definition (Estimate)

If $x_1, \dots, x_n$ is a sample from a random variable whose distribution depends on an unknown parameter $\theta \in \mathbb{R}^k$, then any function $\mathbb{R}^n \to \mathbb{R}^k$ is called an estimate of θ based on $x_1, \dots, x_n$.

7.6 Remarks

With such a general definition, of course, we need some ways to assess the quality of an estimate and compare various estimates, since some of them are definitely better than others. We first emphasize that an estimate $\hat{\theta}$ is a function of random numbers $x_1, \dots, x_n$, hence it is a random variable itself. As such, it has mean value and variance. The mean values of the estimates $\hat{\mu}$ and $\hat{\sigma}^2$ in Example 7.3 can be easily computed (we leave this as an exercise):

$$E(\hat{\mu}) = \mu, \qquad E(\hat{\sigma}^2) = \frac{n-1}{n}\sigma^2$$

This means that, on the average, our estimate $\hat{\mu}$ coincides with μ, and our estimate $\hat{\sigma}^2$ differs from σ^2 by a small relative error $1/n$.

7.7 Definition (Unbiased estimates)
An estimate $\hat{\theta}$ of an unknown parameter θ is said to be *unbiased* if $E(\hat{\theta}) = \theta$. If an estimate is biased, then the difference $E(\hat{\theta}) - \theta$ is called the *bias* of $\hat{\theta}$.

7.8 Remarks
The estimate $\hat{\mu}$ is unbiased. So is $\hat{\mu}_1$ in 7.4. The estimates $\hat{\mu}_2$ and $\hat{\mu}_3$ are also unbiased for the normal random variables of 7.3.

The estimate $\hat{\sigma}^2$ is biased, but one can make it unbiased by a slight modification. Let

$$\hat{\sigma}^2 = \frac{M_2}{n-1} - \frac{M_1^2}{n(n-1)} = \frac{1}{n-1} \sum (x_i - \hat{\mu})^2$$

This estimate does not maximize the likelihood function L, but it is unbiased. Most experimenters prefer the unbiased version of $\hat{\sigma}^2$ to the maximum likelihood version. In statistics, $\hat{\mu}$ is called the *sample mean* and the unbiased version of $\hat{\sigma}^2$ is called the *sample variance.*

7.9 Definition (Variance of estimates)
The accuracy of an estimate $\hat{\theta}$ can be measured by the mean square error

$$Q(\hat{\theta}) = E[(\hat{\theta} - \theta)^2]$$

It is easy to derive (we leave this as an exercise) that

$$Q(\hat{\theta}) = \mathrm{Var}(\hat{\theta}) + [\mathrm{Bias}(\hat{\theta})]^2$$

The mean square error, therefore, is made up by the variance and the squared bias of $\hat{\theta}$. The variance of $\hat{\theta}$ characterizes random (statistical) errors, while the bias constitutes a systematic error.

In most applications, the bias is relatively easy to eliminate. Often unbiased estimates are available, and if not, one can make the bias much smaller than typical statistical errors, i.e. ensure that $|\,\mathrm{Bias}(\hat{\theta})\,| \ll \sigma_{\hat{\theta}} = \sqrt{\mathrm{Var}(\hat{\theta})}$. Therefore, for such estimates

$$Q(\hat{\theta}) \approx \mathrm{Var}(\hat{\theta})$$

Best estimates have smaller variances. Optimal estimates have minimum variance.

7.10 Theorem (Rao-Cramer lower bound)
It turns out, that the minimal variance of an estimate can be found exactly in each problem. Given a density function $f_\theta(x)$, there is a precise formula for the minimum

variance of estimates $\hat{\theta}$ (assuming that those estimates are unbiased). This formula is known as Rao-Cramer lower bound:

$$\begin{aligned}\operatorname{Var}(\hat{\theta}) &\geq \frac{1}{n\int[\partial \ln f_\theta(x)/\partial\theta]^2 f_\theta(x)\,dx} \\ &= \frac{-1}{n\int[\partial^2 \ln f_\theta(x)/\partial\theta^2] f_\theta(x)\,dx}\end{aligned}$$

The integrals, of course, are taken over $\mathbb{R}$. (Miraculously, these two integrals are negative of each other, this is why the above two fractions are equal.)

If $\hat{\theta}$ is biased but its bias is small enough (in the sense $|\operatorname{Bias}(\hat{\theta})| \ll \sigma_{\hat{\theta}}$), then the above bound is valid approximately.

7.11 Remarks

(a) The two integrals in the respective denominators of 7.10 are the expectations

$$E\left[\frac{\partial \ln f_\theta(x)}{\partial\theta}\right]^2 \qquad \text{and} \qquad E\left[-\frac{\partial^2 \ln f_\theta(x)}{\partial\theta^2}\right]$$

sometimes one is easier to compute than the other.

(b) While we stated the Rao-Cramer lower bound for continuous random variables, it is also valid for discrete distributions with summations replacing integrations.

(c) When $\theta \in \mathbb{R}^k$ is a parameter vector, i.e. $k \geq 2$, then the integrands in 7.10 become $k \times k$ matrices. Precisely, the square of the first partial derivative $[\partial \ln f_\theta(x)/\partial\theta]^2$ is replaced by the outer (tensor) product of the gradient vector with itself, $\nabla_\theta f_\theta(x) \otimes \nabla_\theta f_\theta(x)$. The second partial derivative $\partial^2 \ln f_\theta(x)/\partial\theta^2$ is replaced by the Hessian matrix of $f_\theta(x)$. Taking the reciprocals in 7.10 is replaced by taking the inverses of the corresponding matrices.

(d) For large n, the Rao-Cramer bound is $\mathcal{O}(1/n)$, hence the average errors of good estimates are of order $1/\sqrt{n}$. This is expressed in a "rule of thumb" popular in practice: n experimental data render statistical estimates accurate to within $1/\sqrt{n}$. For example, a sample of size $n = 100$ may give one accurate digit, then a sample of size $n = 10,000$ would give two digits, etc. To obtain one additional digit we need to increase n by a factor of 100.

7.12 Definition (Statistical efficiency)

In many problems, an estimate whose variance attains the Rao-Cramer lower bound can be computed. Such estimates are said to be *statistically efficient* or *statistically optimal*. The above estimate of μ, see 7.3, is statistically efficient.

7.13 Exercise

Compute the variance of $\hat{\mu}$ in 7.3.

7.14 Exercise

Compute the Rao-Cramer lower bounds for $\hat{\mu}$ and $\hat{\sigma}^2$ in Problem 7.3. (Note: when differentiating the density function, treat σ^2 as a parameter, not σ. Hint: the corresponding matrix, after integration, must be diagonal.) Compare the variance of $\hat{\mu}$ with its lower bound (Hint: they must be equal).

7.15 Definition (Asymptotic efficiency)

In many cases, the efficiency can only be achieved asymptotically, as $n \to \infty$. An estimate $\hat{\theta}$ is said to be *asymptotically efficient* if its variance satisfies

$$\begin{aligned}\liminf_{n\to\infty} [n \operatorname{Var}(\hat{\theta})] &\geq \frac{1}{\int [\partial \ln f_\theta(x)/\partial\theta]^2 f_\theta(x)\, dx} \\ &= \frac{-1}{\int [\partial^2 \ln f_\theta(x)/\partial\theta^2] f_\theta(x)\, dx}\end{aligned}$$

The estimates of σ^2 introduced in 7.3 and 7.8 are (both) asymptotically efficient, see next.

7.16 Exercise

Compute the variances of the estimates $\hat{\sigma}^2$ discussed in 7.3 and 7.8.

This is a lengthy exercise and can be done is several steps:

(a) Consider a more general estimate

$$\hat{\sigma}^2 = C \sum (x_i - \hat{\mu})^2 = C(M_2 - M_1^2/n)$$

with some $C = C(n)$. Note that $C = 1/n$ and $C = 1/(n-1)$ for the particular estimates introduced in 7.3 and 7.8.

(b) Show that

$$\hat{\sigma}^2 = C \left(\frac{n-1}{n} \sum x_i^2 - \frac{1}{n} \sum_{i\neq j} x_i x_j \right)$$

(c) Verify that so defined $\hat{\sigma}^2$ is invariant under the shift of data, i.e. it remains unchanged if all x_i are replaced by $x_i + b$ with a constant b. Hence we can replace x_i by $x_i - \mu$, and so effectively assume that $\mu = 0$.

(d) Compute $E(\hat{\sigma}^2)$.

(e) Show that

$$[\hat{\sigma}^2]^2 = C^2 \left(\frac{(n-1)^2}{n^2} \sum x_i^4 - \frac{n^2 - 2n + 2}{n^2} \sum_{i\neq j} x_i^2 x_j^2 + \cdots \right)$$

where the terms not shown involve products of x_i's with odd degrees, hence with zero mean.

(e) Compute $E[\hat{\sigma}^2]^2$. Recall that $E(x_i^4) = 3\sigma^4$.

(f) Compute $\mathrm{Var}(\hat{\sigma}^2)$.

7.17 Exercise
Compute the accuracy of $\hat{\sigma}^2$ introduced in 7.16 (a) by using the definition 7.9. Show that the biased version of $\hat{\sigma}^2$ introduced in 7.3 is more accurate than the unbiased version found in 7.8. Find the value of C for which the corresponding estimate is the *most accurate*.

7.18 Theorem (Efficiency of MLE estimates)
Under very general conditions, maximum likelihood estimates are asymptotically efficient. We refer to statistical textbooks for precise statements. This well known theorem is the main reason for the wide popularity of MLE in practical problems.

7.19 Exercise
Sometimes MLE's take quite an unexpected form. Let $x_1, \ldots, x_n$ be sampled from a uniform distribution on an unknown interval (a, b). Estimates a and b by maximizing the likelihood function.

7.20 Weighted averages
Let us modify Example 7.3 slightly. Assume now that for each $i = 1, \ldots, n$ the value x_i is normal $N(\mu, d_i\sigma^2)$, where μ and σ^2 are unknown parameters, and d_i are some known factors. This happens, for example, when the $x_1, \ldots, x_n$ are experimental measurements made with different precision (by different tools/guages). One may know relative accuracies of those measurements, i.e. $\mathrm{Var}(x_i)/\mathrm{Var}(x_j)$. In this case, as one can easily compute, the MLE of μ is

$$\hat{\mu} = \frac{\sum w_i x_i}{\sum w_i}, \qquad w_i = \frac{1}{d_i}$$

So $\hat{\mu}$ is a *weighted average* of x_i's, with weights given by $w_i = 1/d_i$. Weighted averages are used in many practical applications, see below.

7.21 MLE of a mean value for an arbitrary distribution.
Here we generalize 7.3 further. Assume that $x_1, \ldots, x_n$ have an arbitrary distribution with an unknown mean value. Precisely, let the density function be $f(x-\mu)$, where $f(x)$ is a fixed density function satisfying $\int x f(x)\, dx = 0$, so that the mean value of each x_i is indeed μ. Here μ will be the only unknown parameter. Then the likelihood function is

$$L(\mu) = f(x_1 - \mu) \cdots f(x_n - \mu)$$

Taking logarithm and differentiating with respect to μ shows that the MLE satisfies the equation

$$\sum -\frac{f'(x_i-\mu)}{f(x_i-\mu)} = 0 \tag{10}$$

Generally, there is no obvious ways of solving this equation for μ. However, one trick is very common in practice. One rewrites the above equation as

$$\sum -\frac{f'(x_i-\mu)}{(x_i-\mu)\,f(x_i-\mu)}\,(x_i-\mu) = 0$$

and then obtains

$$\mu = \frac{\sum w_i x_i}{\sum w_i}, \qquad w_i = -\frac{f'(x_i-\mu)}{(x_i-\mu)\,f(x_i-\mu)} \tag{11}$$

This gives μ as a weighted average of x_i's (we retain the negative sign here to make the weights positive, at least for any unimodal function f). Of course, our "weights" w_i depend on μ itself, so this is not really a "solution". However, this suggests a fixed-point scheme for a numerical solution of (10), recall our 5.11–5.15.

When implementing the fixed-point method, one needs an initial guess μ_0. Then one computes w_i's by using μ_0, and then finds μ_1 by the weighted average formula (11). Next one recomputes w_i's by using μ_1 and finds μ_2 as the weighted average. Since the weights are recomputed at every iterations, such an algorithm is referred to as *reweight procedure*. It is a common method for computing the MLE of a mean value in many applications.

7.22 Exercise

Show that w_i =const in (11) if and only if f is a normal density. Therefore, the classical sample mean is only appropriate for normal random variables.

7.23 Example: A contaminated sample

In some applications, a sample $x_1, \dots, x_n$ from a normal distribution $N(\mu, \sigma^2)$ is contaminated by "noise". That is, most x_i's are indeed sampled from $N(\mu, \sigma^2)$, but a few come from a very different distribution, usually with a variance much larger than σ^2. We call them *noisy observations*. In many cases, those can be seen graphically as points lying far from the bulk of the sample, such points are also called *outliers*. There is, however, no a priory rule for detection and removal of all noisy observation.

The model of a contaminated sample is a mixture of two densities: a normal $N(\mu, \sigma^2)$ density $f(x)$ (the main part) and another density $h(x)$ (the background noise):

$$g(x) = pf(x) + qh(x) \tag{12}$$

where $p+q=1$. Here p and q represent proportions of "good" and "bad" observations in the sample (note that p/q is called the *signal-to-noise ratio*). The "noisy" density $h(x)$ can be also normal $N(\mu_1, \sigma_1^2)$ with some $\sigma_1 \gg \sigma$. Alternatively, $h(x)$ may be a uniform density

$$h(x) = 1/L \qquad \text{for} \qquad a < x < a+L$$

and we assume that $a < \mu < a + L$ and $\sigma \ll L$. We are interested in estimating μ, and for simplicity regard σ^2, a, L, p and q as known quantities. A typical example: $p = 0.9$, $q = 0.1$, $a = 0$, $L = 100$, $\sigma = 2$.

The density (12) resembles a gaussian bell-shaped curve, but unlike the normal density, it has *heavy tails*.

The MLE of μ can be found by the reweight procedure described in 7.21, with the weights w_i given by the formula (11):

$$w_i = -\frac{g'(x_i - \mu)}{(x_i - \mu)\, g(x_i - \mu)} = \frac{1}{\sigma^2} \frac{pf(x_i - \mu)}{pf(x_i - \mu) + qh(x_i - \mu)}$$

(the constant factor $1/\sigma^2$ is, of course, irrelevant and can be removed). It is not hard to plot w_i as a function of x_i (by using MAPLE, for instance). It is a unimodal function with a maximum at $x_i = \mu$, it is almost symmetric about $x_i = \mu$, and it practically vanishes for $|x_i - \mu| \geq 4\sigma$. Since the weight w_i characterizes the "contribution" of x_i to the average (11), the values x_i's far from μ (outliers) are practically eliminated by our weighting system. Estimates that detect and eliminate outliers (noise) are said to be *robust*, see next.

7.24 Definition (Robust estimates)

An estimate $\hat{\mu}$ is said to be *robust* if it remains accurate when the sample is contaminated by outliers. (Almost) equivalently, we can say that a robust estimate remains accurate for probability distributions different from normal, especially for those with heavy tails. (It is customary to say that robust estimates are "distribution-free".)

7.25 Examples of robust estimates

A simple example of a robust estimate is *trimming*, in which one removes a certain proportion (say, 5% or 10%) of the observations from the upper end and/or from the lower end of the sample. Precisely, let

$$x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$$

be order statistics (they are obtained simply by ordering the given sample $x_1, \ldots, x_n$). Then one estimates μ by

$$\hat{\mu} = \frac{x_{(n_1)+1} + \cdots + x_{(n_2)}}{n_2 - n_1}$$

with $n_1 = [r_1 n]$ and $n_2 = n - [r_2 n]$ for some $r_1, r_2 \geq 0$. This removes $r_1 n$ smallest observations and $r_2 n$ largest observations.

For $r_1 = r_2 = 0$ we recover the classical (nonrobust) sample mean. When r_1 and r_2 grow, the estimate becomes increasingly robust. For $r_1 = r_2 = 0.5$ one obtains the median x_{med} (recall Remark 7.4), which is an extremely robust estimate. In many applications, where the distribution of x_i's is completely unknown and is expected to behave "wildly", x_{med} is the safest estimate of the mean value.

However, robust estimates are less accurate than the classical sample mean in the case of pure (not contaminated) normal random variables. In particular, the median is not statistically efficient, one can compute its variance and find that it is larger than the Rao-Cramer lower bound.

7.26 Practical algorithms

The reweight procedure described in 7.21 requires an initial guess. It is clear that the wrong guess μ_0 (for example, such that $|\mu_0 - \mu| > 10\sigma$) could screw up the whole procedure, because it would immediately suppresses all good observations and "focus" on a few noisy points converging to their average.

To improve the performance of the reweight procedure, one can use a version of "simulated annealing". It requires an external control parameter T ("temperature"). First, the temperature is set to a high enough value, such as $T = L$. The reweight procedure is applied with with σ replaced by σT. Since the initial value of T is very large, no weights w_i will be too small and no observation will be suppressed. Then one gradually decreases the temperature T, so that the reweight procedure will be restricted to a smaller and smaller window that captures a part of the sample and suppresses the rest. The window will be selected automatically, based on the previous approximations and the current value of T. Thus the procedure can adjust itself, and in typical cases it will focus on parts of the sample with higher density of observations.

Here is a possible implementation of this procedure. Let $\beta < 1$ be a constant slightly less than one, such as $\beta = 0.95$. Let $K \geq 1$ be a fixed integer, such as $K = 10$.

Step 1 Set $T = L$ and select an arbitrary initial value for μ.

Step 2 Apply K iterations of the reweight procedure with σ replaced by σT.

Step 3 Replace T by βT. If $T > 1$, repeat Step 2, otherwise exit.

8 Fitting Curves

A typical statistical problem is fitting curves to experimental data. The curve is usually defined by a simple equation $f_\theta(x,y)=0$, where $\theta \in \mathbb{R}^k$ is a parameter (vector) describing the family of available curves. Given n data points $(x_1,y_1),\dots,(x_n,y_n)$ one wants to find the best fitting curve, i.e. estimate the parameter θ.

There are two probabilistic models available for this class of applied problems. If the values of one variable, say x, are known precisely (for example, they are chosen by the experimenter), then one uses a *classical regression model.* However, if both coordinates x and y are random (measured experimentally or "corrupted with errors"), then one uses an *orthogonal regression model.*

8.1 Classical regression

Suppose an unknown function $y = f_\theta(x)$ must be determined experimentally. One can measure the value of y at some selected points $x_1,\dots,x_n$. The corresponding experimental values $y_1,\dots,y_n$ are assumed to satisfy

$$y_i = f_\theta(x_i) + e_i, \qquad i=\overline{1,n}$$

where e_i are random (statistical) errors. It is standard to assume that $e_1,\dots,e_n$ are independent normal random variables with zero mean and common variance σ^2. Then y_i is normal with mean $E(y_i)=f(x_i)$ and variance $\mathrm{Var}(y_i)=\sigma^2$. The likelihood function is

$$L(\theta)=\left(\frac{1}{2\pi\sigma^2}\right)^{n/2}\exp\left[-\frac{\sum[y_i-f_\theta(x_i)]^2}{2\sigma^2}\right]$$

and its logarithm

$$\ln L(\theta) = -\frac{n}{2}\ln(2\pi\sigma^2)-\frac{\sum[y_i-f_\theta(x_i)]^2}{2\sigma^2}$$

The quantity σ^2 can also be regarded as a parameter of the model, but it is not of a primary interest.

The maximum likelihood estimate of θ is

$$\hat\theta = \mathrm{argmax}\ln L(\theta)$$

It is easy to see that the maximization of L is equivalent to the minimization of

$$F(\theta)=\sum[y_i-f_\theta(x_i)]^2$$

The value $\hat\theta =$ argmin $F(\theta)$ is called the *least squares estimate* of θ.

In other words, the best fitting curve $y=f_\theta(x)$ minimizes the sum of squares of the distances to the experimental points (measured along the y axis). This principle is often called the least squares fit (LSF).

We emphasize that the variables x and y are substantially different. The values of x are known precisely or selected by the experimenter, they are fixed (nonrandom) constants, but the values of y are measured experimentally and are random. We call x the *control variable* and y the *response variable.*

8.2 Orthogonal regression

Now assume that both variables x and y are random and we need to find an unknown function $f_\theta(x, y) = 0$ experimentally, where $\theta \in \mathbb{R}^k$ is the vector of parameters describing the family of available functions. One can think that several points on the graph of the function are measured experimentally and their coordinates (x_i, y_i), $1 \leq i \leq n$, are recorded.

The sampled points (x_i, y_i) are perceived as random perturbations of some true (ideal) points $(\bar{x}_i, \bar{y}_i)$ lying on the unknown curve $f_\theta(x, y) = 0$. Assume that the statistical errors $e'_i = x_i - \bar{x}_i$ and $e''_i = y_i - \bar{y}_i$ are independent and normally distributed with zero mean and variance σ^2. Then the likelihood function is

$$L = \left(\frac{1}{2\pi\sigma^2}\right)^n \exp\left[-\frac{\sum[x_i - \bar{x}_i]^2 + [y_i - \bar{y}_i]^2}{2\sigma^2}\right]$$

Taking logarithm gives

$$\ln L = -n\ln(2\pi\sigma^2) - \frac{\sum[x_i - \bar{x}_i]^2 + [y_i - \bar{y}_i]^2}{2\sigma^2}$$

Denote by $d_i^2 = [x_i - \bar{x}_i]^2 + [y_i - \bar{y}_i]^2$ the square of the distances from the experimental point to the corresponding "true" point, $1 \leq i \leq n$. The maximization of L is equivalent to the minimization of

$$F(\theta) = \sum d_i^2 \tag{13}$$

The "true" points $(\bar{x}_i, \bar{y}_i)$ must belong to the "true" curve, i.e.

$$f_\theta(\bar{x}_i, \bar{y}_i) = 0, \qquad i = \overline{1, n} \tag{14}$$

The points $(\bar{x}_i, \bar{y}_i)$ are unknown and, technically, need to be estimated, too, along with the main parameters $\theta \in \mathbb{R}^k$. But the "true" points $(\bar{x}_i, \bar{y}_i)$ are constrained by the equations (14). This constraint gives a simple way of eliminating the "true" points from the analysis. For each θ, the best choice of the point $(\bar{x}_i, \bar{y}_i)$ is the closest point of the curve $f_\theta(x, y) = 0$ to the observed point (x_i, y_i). Then d_i in (13) is the geometric (orthogonal) distance from the point (x_i, y_i) to the curve $f_\theta(x, y) = 0$. The minimization of (13) is called *orthogonal least squares problem.* The value $\hat{\theta} = \operatorname{argmin} F(\theta)$ is called the *orthogonal least squares estimate* of θ. The curve minimizing the sum of squares of the distances to the experimental points is called the *orthogonal least squares fit.*

Next we show how regression problems can be solved, emphasizing computational aspects of our analysis.

8.3 Classical linear regression

In many applications, the function $y = f_\theta(x)$ depends on the parameter θ linearly. For example, if you fit a polynomial $y = a_0 + a_1 x + \cdots + a_m x^m$, then the coefficients $a_0, \ldots, a_m$ are your unknown parameters.

More generally, let

$$y = a_1 g_1(x) + a_2 g_2(x) + \cdots + a_k g_k(x)$$

where $g_1(x), \ldots, g_k(x)$ are arbitrary known functions. If this happens in your applied problem, consider yourself lucky, since the problem has a direct, precise, and relatively simple solution.

8.4 Normal equations

In the previous problem, we denote $x_{ij} = g_j(x_i)$ for all $1 \le j \le k$ and $1 \le i \le n$. We are to minimize the function

$$F(a_1, \ldots, a_k) = \sum_{i=1}^{n} (y_i - a_1 x_{i1} - a_2 x_{i2} - \cdots - a_k x_{ik})^2$$

Taking partial derivatives with respect to a_j, $1 \le j \le k$ one arrives at a linear system

$$\begin{aligned} X_{11}a_1 + X_{12}a_2 + \cdots + X_{1k}a_k &= Y_1 \\ X_{21}a_1 + X_{22}a_2 + \cdots + X_{2k}a_k &= Y_2 \\ &\cdots\cdots \\ X_{k1}a_1 + X_{k2}a_2 + \cdots + X_{kk}a_k &= Y_k \end{aligned} \tag{15}$$

where

$$X_{jm} = \sum_{i=1}^{n} x_{ij} x_{im}, \qquad \text{and} \qquad Y_j = \sum_{i=1}^{n} y_i x_{ij}$$

We call (16) the system of *normal equations*.

The equations (16) can be written in matrix form, this will illuminate many important properties of this system. Denote by

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{pmatrix}$$

the $n \times k$ *data matrix* (also called the *design matrix*) and by $\mathbf{Y} = (y_1, \ldots, y_n)^T$ the *response* vector. Let $\mathbf{A} = (a_1, \ldots, a_k)$ be the parameter vector. Then the system of normal equations takes form

$$\mathbf{X}^T \mathbf{X}\, \mathbf{A} = \mathbf{X}^T \mathbf{Y}$$

and its solution is

$$\mathbf{A} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

Next we present three numerical algorithms for computing $\mathbf{A}$.

8.5 First algorithm (by Cholesky factorization)

The matrix $\mathbf{N} = \mathbf{X}^T\mathbf{X}$ is symmetric and positive semidefinite (the latter means that $\mathbf{u}^T\mathbf{N}\mathbf{u} \geq 0$ for any vector $\mathbf{u} \in \mathbb{R}^k$). If the design matrix $\mathbf{X}$ has full rank (which it does in generic cases), then $\mathbf{N}$ is positive definite, which means that $\mathbf{u}^T\mathbf{N}\mathbf{u} > 0$ for any nonzero vector $\mathbf{u} \neq \mathbf{0}$.

For such matrices $\mathbf{N}$, there are many fast algorithms for solving systems of equations $\mathbf{N}\mathbf{x} = \mathbf{y}$. A standard algorithm on this occasion is Cholesky factorization covered in Numerical Linear Algebra, MA 660. It is simple, fast and theoretically exact.

This algorithm requires about

$$(k^2 + 3k)n + k^3/3 + 2k^2$$

arithmetic operations (also called floating point operations, or *flops* for brevity). Here is a precise flop count: the multiplication $\mathbf{X}^T\mathbf{X}$ takes $(k^2 + k)n$ flops, the product $\mathbf{X}^T\mathbf{Y}$ takes $2kn$ flops, the Cholesky factorization of the matrix $\mathbf{X}^T\mathbf{X}$ takes $k^3/3$ flops, then one finishes computing $\mathbf{A}$ by a backward substitution (k^2 flops) and a forward substitution (another k^2 flops). We note that k is usually a small fixed number (such as 3 or 4), while n is usually large and variable.

This method has been standard until about 1970. It is still widely used today, due to its simplicity and efficiency in most practical cases. However, one must be aware of its drawbacks. This method is numerically unstable. Whenever the matrix $\mathbf{X}$ is ill-conditioned, the above method may easily break down. More precisely, if the condition number $\kappa(\mathbf{X})$ is large, then the condition number $\kappa(\mathbf{N})$ becomes huge, because

$$\kappa(\mathbf{N}) = [\kappa(\mathbf{X})]^2$$

For example, if $\kappa(\mathbf{X}) = 10^q$, then any solution of the least squares problem in double precision will yield at most $15 - q$ accurate digits. However, the above solution based on the normal equations will only yield $15 - 2q$ accurate digits. There is an unwanted loss of additional q digits, caused exclusively by the method. (Note that it is not the Cholesky algorithm that is at fault here, it is the formation of the normal equations where the loss of accuracy occurs.)

8.6 Second algorithm (by QR decomposition)

There is an alternative solution that is numerically stable. It avoids an explicit use of the normal equations (16), and instead employs the QR decomposition of the design matrix $\mathbf{X}$, as studied in Numerical Linear Algebra, MA 660. Precisely, there is a $n \times k$ matrix $\mathbf{Q}$ whose columns are orthonormal vectors (i.e. its columns are unit vectors orthogonal to each other) such that

$$\mathbf{X} = \mathbf{Q}\mathbf{R}$$

where R is an upper triangular $k \times k$ matrix. Then the least squares problem can be solved by

$$\mathbf{A} = \mathbf{R}^{-1}\mathbf{Q}^T\mathbf{Y}$$

This is proved in Numerical Linear Algebra, MA 660.

The QR method is more complicated and costly. Its cost is

$$(2k^2 + 2k)n + k^2$$

flops. Precisely, the QR decomposition of $\mathbf{X}$ requires $2k^2n$ flops, the multiplication $\mathbf{Q}^T\mathbf{Y}$ takes $2kn$ flops, then one finishes computing $\mathbf{A}$ by a backward substitution in k^2 flops. We see that the QR method is about twice as expensive as the previous method based on the normal equations.

However, the QR method is numerically stable and accurate. It is also theoretically exact, it does the job in finitely many computations (it does not involve iterations or approximations of any kind). We note that in the modern world of fast computers, the complexity of an algorithm is not so a critical issue as before, while its accuracy and stability are essential.

The only instance when the QR method fails is the so called rank-deficient case, when $\operatorname{rank} \mathbf{X} < k$. In that case the least squares problem has multiple solutions. Its solutions form a linear subspace in $\mathbf{L} \subset \mathbb{R}^k$ (we note that $\mathbf{L}$ does not pass through the origin). In practical problems, one either picks an arbitrary solution $\mathbf{A} \subset \mathbf{L}$ or selects the solution with the minimal norm:

$$\mathbf{A}_{\min} = \operatorname{argmin}_{\mathbf{A} \subset \mathbf{L}} \|\mathbf{A}\|$$

The QR method can be modified (by using the so called pivoting, i.e. a suitable reordering of the columns of $\mathbf{X}$) to handle this special case, but this modification is not very satisfactory. First, it involves a significant amount of extra computations. Second, there are documented examples where the fails anyway, for some strange and purely numerical reasons.

8.7 Third algorithm (by SVD decomposition)

In the last decade, another method emerged as an attractive alternative to the QR. It is based on the SVD decomposition studied in Numerical Linear Algebra, MA 660. For any real $n \times k$ matrix $\mathbf{X}$, there are $n \times n$ and $k \times k$ orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$, respectively, and a diagonal $n \times k$ matrix $\mathbf{D}$ such that

$$\mathbf{X} = \mathbf{UDV}$$

Then the least squares problem can be solved by

$$\mathbf{A} = \mathbf{V}^T[\mathbf{D}^-]^T\mathbf{U}^T\mathbf{Y} \tag{16}$$

where $\mathbf{D}^-$ is a diagonal $n \times k$ matrix whose entries on the main diagonal are the reciprocals of those of $\mathbf{D}$ (with one exception: when a diagonal entry of $\mathbf{D}$ is zero, then the corresponding diagonal entry of $\mathbf{D}^-$ must be zero as well).

The SVD method is even more complicated than the QR. The computation of an SVD is usually done in two steps. At first, one transforms the given matrix $\mathbf{X}$ into a bidiagonal form by $\mathbf{X} = \mathbf{U}'\mathbf{B}\mathbf{V}'$, where $\mathbf{U}'$ and $\mathbf{V}'$ are orthogonal matrices and $\mathbf{B}$ is a bidiagonal matrix. This takes $2k^2n + 2k^3$ flops. At the second stage one reduces $\mathbf{B}$ to a diagonal matrix $\mathbf{D}$ by an iterative procedure. This takes Ck^2 flops, where the factor C depends on the machine precision. After the SVD is done, one computes $\mathbf{A}$ by (16), which takes $2kn + k^2 + k$ flops. The total cost of this algorithm is thus

$$(2k^2 + 2k)n + 2k^3 + (C + 1)k^2 + k$$

flops. We see that it is slightly more expensive than the QR method.

We note that the SVD method involves an iterative procedure, hence it cannot compute the result in a finite number of steps. In other words, the computation of an SVD is **not** theoretically exact – it involves a truncation error, in addition to the inevitable round-off errors, recall 3.10. This all sounds quite discouraging, right?

However, the SVD method is numerically stable. It is extremely stable. In most practical computations, it outperforms the QR method and yields more accurate results. A very detailed example is provided in [1] on pp. 137–143. Also, the SVD automatically handles the rank-deficient case in the proper manner and returns the minimum norm solution $\mathbf{A}_{\min}$ mentioned above. Overall, it is currently the most reliable algorithm for solving the least squares problems. Its superior numerical performance is recognized by almost everyone.

The codes (in C and FORTRAN) for an SVD decomposition are freely available on the Internet. A good source for this and other useful computer programs is www.netlib.org, a free software repository. The SVD routine is also available in the Numerical Recipes (NR) software package on our departmental server catbert.math.uab.edu. The NR package also includes codes for Cholesky factorization and QR decomposition.

8.8 Linear orthogonal regression

This is a continuation of 8.2. We restrict ourselves to the simplest case – fitting a straight line. A line is usually defined by an equation $y = a + bx$, but this is not good for our purposes. First, it does not include vertical lines. Second, the roles of x and y in the equation $y = a + bx$ are clearly different, but we would like to have a complete symmetry between x and y, due to the nature of the problem. So we prefer another equation:

$$Ax + By + C = 0 \tag{17}$$

Obviously, the line defined by (17) remains unchanged if we multiply the parameters A, B, C by a nonzero scalar. In this case one says that A, B, C are *indeterminate*, i.e. they cannot be determined by a line they represent. To rectify this problem, we impose the constraint

$$A^2 + B^2 = 1$$

Alternatively, A and B can be replaced by a single parameter θ so that $A = \cos\theta$ and $B = \sin\theta$, with $0 \le \theta < 2\pi$. Note that θ is a cyclic parameter, i.e. $\theta \in S^1$. A line can now be defined by

$$x\cos\theta + y\sin\theta + d = 0$$

with two parameters, θ and d. These parameters have a clear geometric interpretation: θ is the slope of the normal vector to the line, and $|d|$ is the distance from the line to the origin $(0,0)$.

The sum of squares of the distances from the experimental data (x_i, y_i) to the line is given by

$$F(A,B,C) = \frac{1}{A^2+B^2}\sum_{i=1}^{n}(Ax_i + By_i + C)^2 \tag{18}$$

in the parametrization A, B, C without imposing the constraint $A^2 + B^2 = 1$, and by

$$F(\theta, d) = \sum_{i=1}^{n}(x_i\cos\theta + y_i\sin\theta + d)^2 \tag{19}$$

in the parametrization θ, d. Our goal is to find the minimum of F.

8.9 Existence of the orthogonal least squares fit

We pose for a second to discuss the existence of a solution of our minimization problem 8.8. Obviously, the function F depends continuously (in fact, smoothly) on parameters in both parametrization schemes (18) and (19). It is known that a nonnegative continuous function on a compact topological space always attains a minimum (possibly, not unique). However, our parameter spaces are not compact, so additional care must be taken to show the existence of the minimum. Note that the set of experimental points (x_i, y_i) is finite, hence bounded, so all the data points (x_i, y_i) lie in some square (or rectangle) K, which we assume to be closed. Now, consider all lines that intersect the region K. We can disregard other lines, since the function F clearly takes larger values on those than on the lines intersecting K. Since $|d|$ is the distance from the line to the origin, we may impose restriction $|d| < d_{\max}$ where $d_{\max}$ is the distance from the origin $(0,0)$ to the most remote point of K. Now, the θd parameter space becomes a cylinder $S^1 \times [-d_{\max}, d_{\max}]$, which is compact. This proves that the function F takes a minimum, hence the orthogonal least squares fit always exists.

8.10 Implementation of the orthogonal least squares fit

Now we develop a numerical algorithm for computing the minimum of F. We use methods of linear algebra. The minimization of (18) is equivalent to the problem

$$F_0(A,B,C) = \sum_{i=1}^{n}(Ax_i + By_i + C)^2 \;\to\; \min$$

subject to the constraint $A^2 + B^2 = 1$. The function F_0 is a quadratic polynomial in A, B, C and can we written as

$$F_0 = M_{xx}A^2 + M_{yy}B^2 + 2M_{xy}AB + 2M_xAC + 2M_yBC + nC^2$$

where $M_{\dots}$ denote various moments of the data sample: $M_{xx} = \sum x_i^2$, $M_{xy} = \sum x_i y_i$, etc. In matrix form, $F_0 = \mathbf{A}^T \mathbf{M} \mathbf{A}$, where $\mathbf{A} = (A, B, C)^T$ is the vector of parameters and $\mathbf{M}$ is the matrix of moments:

$$\mathbf{M} = \begin{pmatrix} M_{xx} & M_{xy} & M_x \\ M_{xy} & M_{yy} & M_y \\ M_x & M_y & n \end{pmatrix}$$

Note that $\mathbf{M}$ is symmetric and positive semidefinite. This can be easily seen from the fact that $\mathbf{M} = \mathbf{X}^T \mathbf{X}$, where

$$\mathbf{X} = \begin{pmatrix} x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{pmatrix}$$

is the $n \times 3$ data matrix. The constraint $A^2 + B^2 = 1$ can be written as $\mathbf{A}^T \mathbf{I}_0 \mathbf{A} = 1$, where

$$\mathbf{I}_0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

is the "reduced" identity matrix. Now introducing a Lagrange multiplier η we minimize the function

$$\tilde{F}_0 = \mathbf{A}^T \mathbf{M} \mathbf{A} - \eta(\mathbf{A}^T \mathbf{I}_0 \mathbf{A} - 1)$$

Differentiating with respect to $\mathbf{A}$ gives

$$\mathbf{M}\mathbf{A} - \eta \mathbf{I}_0 \mathbf{A} = 0$$

In this case one says that η is a generalized eigenvalue for the matrix pair $(\mathbf{M}, \mathbf{I}_0)$. Its value can be found from the equation

$$\det(\mathbf{M} - \eta \mathbf{I}_0) = 0 \tag{20}$$

This is a quadratic equation for η. It can be proven that it always has two real nonnegative solutions. Which one corresponds to the minimum of F_0? Well, note that

$$F_0 = \mathbf{A}^T \mathbf{M} \mathbf{A} = \eta \mathbf{A}^T \mathbf{I}_0 \mathbf{A} = \eta$$

hence the minimum of $\mathcal{F}_0$ corresponds to the *smaller* root of the equation (20).

Lastly, $\mathbf{A}$ can be chosen as a null vector of $\mathbf{M} - \eta \mathbf{I}_0$. When the corresponding null space is one-dimensional, the solution will be unique. When it is two-dimensional, then the problem has multiple solutions, see the exercises below.

8.11 Remark

A simple trick can be used to make the above calculations easier and numerically more stable. Let us translate the origin of the coordinate system to the center of mass (also called the *centroid*) of the data set. This can be done by computing

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i, \qquad \bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$$

and then transforming $x_i \mapsto x_i - \bar{x}$ and $y_i \mapsto y_i - \bar{y}$ for all $1 \le i \le n$. If we do that, we make $M_x = M_y = 0$. Also, we reduce the values of the other moments thus minimizing round-off errors.

8.12 Exercise

(a) Prove that if $M_x = M_y = 0$, then $C = 0$ and $d = 0$.

(b) Prove that the quadratic equation (20) always has two real nonnegative roots.

(c) Prove that the least squares line is unique if and only if the roots of (20) are distinct.

(d) Show that the roots of (20) coincide whenever two conditions hold simultaneously:

$$M_{xx} - M_x^2/n = M_{yy} - M_y^2/n \qquad \text{and} \qquad M_{xy} - M_x M_y/n = 0$$

(e) Give a nontrivial example where these conditions hold (Hint: place the data points at vertices of a regular n-gon).

(f) Describe all the solutions of the least squares problem 8.8 when the above conditions hold, i.e. describe the family of the corresponding lines.

8.13 Remarks

(a) When solving the quadratic equation (20), one needs to chose the minus sign before the square root of its discriminant (since we are interested in the smaller root). The other, larger, root of (20) corresponds to a saddle point of the function F_0.

(b) One can implement a different algorithm for minimizing F. First one can eliminate d from (19) by differentiating with respect to d and then solving the equation $\partial F/\partial d = 0$. Next, one minimizes the resulting function $F_1(\theta)$ with respect to the parameter θ alone. Work out the details.

(c) In terms of the algorithm described in 8.13 (b), the larger root of equation (20) will correspond to the maximum of $F_1(\theta)$.