CHAPTER 3

Iterative Methods

As we have seen in the previous two chapters, even for problems, which are theoretically well understood, such as computing the square root, one cannot provide the perfect answer in finite amount of time. One often applies iterative methods and analyses what happens in each iteration, within first \( k \) iterations, and in the large limit of the number of iterations (asymptotically). Iterative methods typically employ a variety of approximations, starting with approximating real numbers by floating-point numbers, and do not stop, unless we specify some stopping rules (termination criteria).

Most iterative method can be cast as:

```python
1 def method(initial = 0.0, stop = lambda x0, xk, k: False):
2     x = initial
3     k = 1
4     while not stop(initial, x, k):
5         x = T(x)
6         k += 1
7     return x
```

Let us use \( x_k \) to denote the \( x \) obtained in iteration \( k \). \( T \) is called the step. \( \text{stop} \) implements the termination criteria, e.g., that we no longer gain any precision, or that \( |x_k - x_{k+1}| \) is below a threshold, or just that \( k \) exceeded some threshold.

It is easy to see that the prototypical iterative method generates, in mathematical terms, a sequence \((x_0,x_1,\ldots,x_n,\ldots)\). One should ask whether the sequence is asymptotically convergent (whether there exists \( x \) such that \( x_k \to x \) as \( k \to \infty \), i.e., there exists \( \lim_{k \to \infty} x_k \)), what does it converge to (whether the limit is what we expect), what is the rate of convergence (how much precision one gains in an iteration), and how stable is it (how fast do errors propagate, or how much precision you lose per iteration). Much of the above should be expressed as functions of parameters of the instance of your problem (condition).

1. Convergence

We may view this algorithm as generating a sequence \((x_0,x_1,\ldots,x_n,\ldots)\) in some metric space \((X,d)\). Usually \((X,d)\) is a normed vector space \((X,\| \|)\), with the metric given by \( d(x,y) := \|x - y\| \) for all \( x,y \in X \). Given \( x_0 \), the sequence is generated as follows:

\[
x_1 = T(x_0), \quad x_2 = T(x_1), \ldots, \quad x_n = T(x_{n-1}).
\]

If this sequence converges to some limit \( x \in X \) then we say that this limit is approached by the process of successive approximation. We sometimes write this sequence as

\[
(x_n = T(x_{n-1})) \quad \text{or} \quad (x_n = T^n(x_0)).
\]

Here, \( T^n(x_0) = T(T(\cdots(x_0)\cdots)) \) means \( T \) applied \( n \) times in succession.

---

1For the sake of completeness: \( \{x_0,x_1,\ldots,x_n,\ldots\} \) denotes a countable set, where the ordering of the elements does not matter; a sequence \( (x_0,x_1,\ldots,x_n,\ldots) \), denoted \( (x_k) \), is simply a countable set where the ordering does matter.
Example 3.1. Here are some successive approximation sequences. For simplicity, we have taken \((X,d) = (\mathbb{R}, | |)\), that is, the metric is given by \(d(x,y) := |x-y|\) for all \(x, y \in \mathbb{R}\), where \(| |\) denotes the usual absolute value or modulus (a 1-dimensional norm). First, \(T(x) = \sqrt{x}\), with \(x_0 = 2\), gives the sequence 
\[(2.0, 1.4142, 1.1892, 1.0905, \ldots)\]
which has limit 1.0.

Note that at the limit \(x = 1\), we have \(x = T(x)\). Second, \(T(x) = x + x(1 - ax)\), with \(a = 3.0\) and \(x_0 = 0.5\), gives the sequence 
\[(0.50000, 0.25000, 0.312500, 0.3320313, 0.333328, 0.333333, \ldots)\]
which has limit 1/3.

Note again that at the limit \(x = 1/3\), we have \(x = T(x)\).\[\square\]

Indeed, for any \(a\) the mapping \(T\) has a fixed point \(x = 1/a\) such that \(x = T(x)\). Note also that this gives a method for calculating \(1/a\) using addition and multiplication only, important for implementation on computers. Next, consider \(T(x) = 1 + x\), with \(x_0 = 0\) gives the sequence 
\[(0, 1, 2, 3, 4, \ldots)\]
which has limit \(\infty\).

This sequence diverges for all \(x_0\). From this, it is clear that the convergence of a sequence generated by successive approximation depends on the transformation \(T\).

One example of sequence, which are guaranteed to converge, are contraction mappings: the distance between a pair of transformed points \(T(x_1), T(x_2)\) needs to be less than the distance between the points \(x_1, x_2\) themselves. Formally: \(T : (X,d) \rightarrow (X,d)\) is a contraction mapping if, for some \(\lambda \in \mathbb{R}\) with \(0 \leq \lambda < 1\), we have \(d(T(x_1), T(x_2)) \leq \lambda d(x_1, x_2)\), for all \(x_1, x_2 \in X\). There, Banach’s fixed point theorem guarantees the convergence \(T\) in a complete metric space, and is particularly useful when we have an obvious metric for measuring the distance between successive points in the sequence (see algorithms for system of linear equations later). In such a case, the sequence converges to the unique fixed point \(x\) of \(T\). It cannot be emphasised enough that without results like Banach’s, we would have no guarantee of the convergence of an iterative algorithm: without such a theoretical grounding, a proposed iterative “algorithm” could not be trusted to perform as it claims.

Remark 3.2. Banach’s fixed point theorem has extensive applications outsied of proofs of convergence of iterative methods. See for example Mareček et al. [2015b] for an analysis of properties of some information provision schemes in intelligent transportation systems, which relies heavily on Banach’s fixed-point theorem.

2. Rate of Convergence

We assume that each iterative algorithm generates a sequence 
\[(x_0, x_1, \ldots, x_k, \ldots)\]
where the limit \(x\) is the solution to our problem. Define the error at stage \(k\) as 
\[e_k = d(x_k, x)\]
\(= \|x_k - x\|\) when \(X\) is a normed space.

Therefore, we may view an iterative algorithm as generating a sequence of non-negative real number errors 
\[(e_0, e_1, \ldots, e_k, \ldots)\]
with \(e_k \rightarrow 0\) as \(k \rightarrow \infty\).

In theory, this sequence is infinite; but in practice, finite precision forces us to stop at some stage \(k\) when \(e_k \leq \varepsilon\), for some chosen \(\varepsilon\).

One of the most important features of an iterative algorithm is the speed with which it converges. Roughly speaking, this is the number of iterations necessary to obtain a fixed point with a specified accuracy. Of course, speed of convergence as measured by number of iterations is not the only criterion by which we judge an algorithm: the amount of computation per iteration (i.e., complexity) is also important.
DEFINITION 3.3. Let \((x_0, x_1, \ldots, x_k, \ldots)\) be a sequence with \(x_k \to x\), and let \(e_k = d(x_k, x)\). If there exists a number \(p\) and a constant \(C \neq 0\) such that
\[
\lim_{k \to \infty} \frac{e_k}{e_{k-1}^p} = C,
\]
then \(p\) is called the order of convergence of \((x_k)\).

This is not the same as the “order” of complexity introduced earlier.

We assume that \(e_k < 1\), for all \(k\) from some point on (that is, each \(x_k\) is in a fairly small neighbourhood of \(x\)).

A more usable definition is: \((x_k)\) has order of convergence \(p\) if
\[
e_k = Ce_k^p,
\]
where \(C\) possibly depends on \(k\), but can be bounded above by a constant.

Sequences with \(p = 1\) are said to have Linear Convergence, while sequences with \(p > 1\) have Super-Linear Convergence (a good thing!).

For simplicity, we look at the case \(T : \mathbb{R} \to \mathbb{R}\), in one dimension. Consider the sequence \((x_k)\) with \(x_k = T(x_{k-1})\), and assume it converges to a fixed point \(x = T(x)\). Assume further that at the fixed point the derivatives \(T'(x), T''(x), \ldots, T^{(n)}(x)\) exist. Expanding \(T(x_k)\) in a Taylor series about the fixed point \(x\), we get
\[
T(x_k) = T(x) + \frac{1}{1!} T'(x)(x_k - x) + \frac{1}{2!} T''(x)(x_k - x)^2 + \cdots + \frac{1}{n!} T^{(n)}(x)(x_k - x)^n + R_{n+1},
\]
or
\[
T(x_k) - T(x) = \frac{1}{1!} T'(x)(x_k - x) + \frac{1}{2!} T''(x)(x_k - x)^2 + \cdots + \frac{1}{n!} T^{(n)}(x)(x_k - x)^n + R_{n+1}.
\]

Now \(|T(x_k) - T(x)| = |x_{k+1} - x| = e_{k+1}\). Recalling that \(|a + b| \leq |a| + |b|\) (triangle inequality), we get:
\[
e_{k+1} \leq |T'(x)|e_k + \frac{1}{2!} |T''(x)|e_k^2 + \cdots + \frac{1}{n!} |T^{(n)}(x)|e_k^n + |R_{n+1}|.
\]

This expression allows us to determine the order of convergence of the successive approximation sequence \((x_k = T(x_{k-1}))\) if we know the derivatives of \(T\) at the fixed point \(x\). Thus, we can deduce the following:

- If \(T'(x) \neq 0\) then \(e_{k+1} \approx T'(x)e_k\), that is, 1st order convergence.
- If \(T'(x) = 0, T''(x) \neq 0\) then \(e_{k+1} = |\frac{1}{2} T''(x)|e_k^2 = \text{const.} e_k^2\), that is, 2nd order convergence.
- If \(T'(x) = T''(x) = \cdots = T^{(n-1)}(x) = 0, T^{(n)}(x) \neq 0\) then \(e_{k+1} = |\frac{1}{n!} T^{(n)}(x)|e_k^n = \text{const.} e_k^n\), that is, \(n\)th order convergence.

EXAMPLE 3.4 (Square Root). One option for calculating \(\sqrt{a}\) in Homework Assignment 1 was:
\[
x_k := T(x_{k-1}) = (x_{k-1} + a/x_{k-1})/2.
\]

The mapping \(T(x) = (x + a/x)/2\) has a fixed point \(x = \sqrt{a}\), since \(T(\sqrt{a}) = (\sqrt{a} + a/\sqrt{a})/2 = (\sqrt{a} + \sqrt{a})/2 = \sqrt{a}\).

The first derivative is \(T'(x) = (1 - a/x^2)/2\). This gives \(T'(\sqrt{a}) = (1 - a/a)/2 = 0\). Hence, this algorithm has 2nd order convergence. This is confirmed by showing that \(T''(\sqrt{a}) \neq 0\), in general.

The effect of 2nd order convergence is to double the number of correct digits at each iteration: \(e_k \approx e_{k-1}^2 \approx e_0^{2^k}\). If \(e_0 = 2^{-1}\) then \(e_k \approx e_0^{2^k} = 2^{-2^k}\). For \(k = 4\) we get \(e_k = 2^{-16} \approx 10^{-5}\). This means that \(k = 5\) iterations are sufficient to give full IEEE single precision.

REMARK 3.5. It can be shown (see later) that the Bisect algorithm for root-finding has order of convergence 1, while Newton’s algorithm has order of convergence 2.

Higher order of convergence generally requires knowing higher derivatives. However, computing these higher derivatives can actually be more work than is saved by a smaller number of iterations!
Remark 3.6. To complicate things: There are literally hundreds of related definitions of convergence and rates of convergence. Specifically, in non-convex optimisation, one distinguishes between the optimum one converges to, and hence between global convergence, i.e., from any starting point to the global optimum, and local convergence, i.e., from starting points within some neighbourhood of the global optimum to the global optimum, and to some local optimum from other starting points. In machine learning, randomised algorithms are common and one hence reasons about expectations over the choices made within the randomised algorithm. See Mareček et al. [2015a] for an example. We will introduce some of these notions formally in later chapters.

3. Errors

Most iterative methods employ at least three types of approximation:

- **Discretisation**: means modelling a continuous set by a discrete or finite set, such as modelling \( \mathbb{R} \) with floating-point numbers, or replacing derivatives by finite differences.
- **Truncation**: means cutting off some parts of a complete model, e.g., using only a finite number of terms from an infinite Taylor series, or stopping after a finite number of iterations.
- **Rounding**: a representation of real numbers so as to employ finite amount of space and time in processing them, or printing them out as a result.

Storing a number in floating point form introduces a *rounding error* because some digits are discarded. If \( \delta x \) is the rounding error, then the output of rounding \( x \) is \( x + \delta x \), where \( \delta x \) may be negative, zero, or positive. In the output of chopping, this \( \delta x \) is negative or zero. What is the maximum magnitude of this error?

**Example 3.7.** Consider a base-10 system with 4 significant digits, \( e_{\text{min}} = -8, e_{\text{max}} = +8 \). If \( x = +(0.1234999\ldots) \times 10^e \) then rounding produces \((0.1234) \times 10^e\) and \( \delta x = -(0.00004999\ldots) \times 10^e \). We can see that the error depends on the exponent \( e \).

The maximum relative roundoff error is derived as follows.

We have

\[
x = 0.(m_1m_2\ldots m_tm_{t+1}\ldots)_b \times b^e
\]

and

\[
x \text{ rounded } \approx 0.(m_1m_2\ldots m_t)_b \times b^e,
\]

so

\[
\text{error } = 0.(00\ldots 0m_{t+1}\ldots)_b \times b^e
\]

\[
\approx b^{-t} \times b^e \text{ in the worst case}
\]

\[
= b^{e-t}.
\]

The *relative roundoff error* is the absolute value of the roundoff error divided by the number-to-be-rounded:

\[
\frac{b^{e-t}}{0.(m_1m_2\ldots m_tm_{t+1}\ldots)_b \times b^e} = \frac{b^{e-t}}{b^{-t}} = b^{e-t}.
\]

This expression achieves its maximum when the denominator is a minimum, i.e., when \( |x| = 0.(100\ldots 0)_b = b^{-1} \). Hence, relative rounding error is \( \frac{b^{e-t}}{b^{-1}} = b^{1-t} \). We call this relative error the *unit roundoff* and denote it by \( u_r \), where

\[
u_r = \begin{cases} b^{1-t} & \text{using chopping,} \\ \frac{1}{2} b^{1-t} & \text{using rounding.} \end{cases}
\]
3.1. Machine Epsilon and Unit Roundoff. Roundoff error occurs when a number \( x \) falls between two adjacent floating point numbers. The maximum distance between these two floating point numbers is \( \varepsilon_m |x| \). Hence, the maximum roundoff error is half the distance between them. Hence, relative roundoff error is

\[
\frac{1}{2} \frac{\varepsilon_m |x|}{|x|} = \frac{1}{2} \varepsilon_m = u_r.
\]

Thus, we see that the unit roundoff error \( u_r \) is half machine epsilon, i.e.,

\[
u_r = \frac{\varepsilon_m}{2} = \frac{1}{2} b^{1-t}.
\]

Exercise 3.8. Determine the decimal unit roundoff \( u_r \) for each of the following number systems:

1. base-2, 24 significant digits
2. base-8, 8 significant digits
3. base-16, 6 significant digits.

Which system do you prefer? Why? If \( w = 32 \) bits are used to store these numbers, what is the maximum value of the biased exponent in each case?

Exercise 3.9. Generate all the numbers representable in the following number systems:

1. base-10, 1 significant digit, \( e_{\min} = -2, e_{\max} = +2 \),
2. base-10, 2 significant digits, \( e_{\min} = -1, e_{\max} = +1 \), and
3. base-2, 3 significant digits, \( e_{\min} = -1, e_{\max} = +2 \).

Exercise 3.10. Given a computer with 3-state memory devices, design two different floating point number systems, where each number is stored in a word with \( w = 3^k \) ternary digits and \( k = 3, 4, 5 \). Assume that the exponent occupies one third (i.e., \( \frac{w}{3} = 3^{k-1} \)) of these digits. Can we represent \( \frac{1}{3} \) precisely in these systems? Can we represent \( \frac{1}{3} \) precisely in base 2 systems? If not, why not?

3.2. Floating Point Arithmetic Operations. Errors occur when performing arithmetic operations even if the operands and the operation are exact. This is because the result of the operation may not be representable.

For example, the numbers 103.366 and 33.2977 are both representable in a base-10 number system with 6 significant digits, but 103.366 + 33.2977 = 136.6637 is not, as it has 7 digits of precision. This number would be rounded to the nearest representable number, 136.664, giving an error of 0.0003.

In general, the result of a floating-point arithmetic operation is

\[
\text{round}(a \text{ op } b) = (a \text{ op } b)(1 + \delta),
\]

where \( \text{op} \) is one of \{\(=, +, -, \times, /\)\} and \(|\delta| \leq u_r\).

This is called the Standard Model of floating point arithmetic. The interpretation of the model is: \( \text{round}(a \text{ op } b) \) is the result of calculating \( a \text{ op } b \) exactly and then rounding this exact result.

Ideally, \( x \text{ op } y = \text{round}(x \text{ op } y) \) (i.e., floating point operations \( \text{op} \) produce correctly rounded results). Many machines, such as those satisfying the IEEE floating point standard (see below) achieve this ideal as long as \( x \text{ op } y \) is within the range of the floating point system.

However, the standard axioms of arithmetic do not always hold. In particular, floating point addition and multiplication are commutative but not associative. This makes forward error analysis difficult.

Example 3.11. For example, with base-10 number system with 3 significant digits, \( e_{\min} = -5, e_{\max} = +5 \) and \( x = -1.00, y = 1.00, \) and \( z = 0.001 \),

\[
\text{round(\text{round}(x + y) + z)} \neq \text{round}(x + \text{round}(y + z)),
\]

because

\[
\text{round}(x + y) = \text{round}(-1.0 + 1.0) = 0.0, \quad \text{round}(\text{round}(x + y) + z) = 0.001,
\]
and
\[ \text{round}(y + z) = \text{round}(1.0 + 0.001) = 1.0, \quad \text{round}(x + \text{round}(y + z)) = \text{round}(-1.0 + 1.0) = 0.000. \]

In general, if \( \delta \) is a positive floating point number a little less than \( \varepsilon_m \), then \((1 + \delta) + \delta = 1 \) but \( 1 + (\delta + \delta) > 1 \). Small errors creep in for numbers, which are periodic in base-2, as we have seen in the previous chapter. As a genera rule: Try to avoid the equality test in dealing with floating-point numbers (and consider \( \varepsilon_m \) when testing the absolute values of differences as well).

3.3. Composition and Propagation of Error. Performing a floating point arithmetic operation \( \text{round}(x \text{ op } y) \) involves two sources of error:

- errors in the operands \( x \) and \( y \), and
- an error in storing the result, attributed to the operator.

Let \( x \) and \( y \) be the true operands and \( \hat{x} \) and \( \hat{y} \) be the actual operands used in the operation, where \( \delta_x, \delta_y \leq u_r \). Let \((x \text{ op } y) = (x \text{ op } y)(1 + \delta)\) be the result of the floating point operation \( \text{op} \), where \( \delta \leq u_r \).

The total error in the result is
\[ E = |(x \text{ op } y) - (\hat{x} \text{ op } \hat{y})|, \]
\[ = |(x \text{ op } y - \hat{x} \text{ op } \hat{y}) - (\hat{x} \text{ op } \hat{y} - \hat{x} \text{ op } \hat{y})|, \]

where \( |\delta| \leq u_r \).

The generated error is
\[ |\hat{x} \text{ op } \hat{y} - \hat{x} \text{ op } \hat{y}| = |\hat{x} \text{ op } \hat{y} - (\hat{x} \text{ op } \hat{y})(1 + \delta)| \leq |\hat{x} \text{ op } \hat{y}||\delta|, \]

3.4. Propagated Error of Addition and Subtraction. To develop the propagated error further we must look at the individual operations. If \( E_{AS} \) is the error propagated by addition or subtraction then we can show (see Appendix)
\[ E_{AS} \leq (|x| + |y|)u_r. \]

This gives no cause for concern but it hides a problem, known as cancellation error, that may occur when subtracting floating point numbers that are close to each other.

First, we emphasise that subtracting close numbers is not necessarily dangerous.

Consider base-10 numbers with 3 significant digits and two representable numbers \( x = 102 \) and \( y = 101 \). These numbers are as close as can be in the floating-point system we consider. The result of subtracting these two numbers is
\[ \text{round}((x - y)) = \text{round}(102 - 101) = \text{round}(1.00), \]

and the error is 0.

Example 3.12. Consider \( x = 100.5 \) and \( y = 100.4 \) and the ratio
\[ \frac{x + y}{x - y}. \]

The exact result is \( \frac{100.5 + 100.4}{100.5 - 100.4} = \frac{200.9}{0.1} = 2009 \). What if the computer used base-10 floating-point numbers with 3 significant digits? \[ \diamond \]
If this result is rounded to 3 digits it becomes 2010.0 with a relative error of $\frac{1}{2009}$, about $\frac{1}{20}$%, and attributable to the subtraction operation. However, $x$ and $y$ are not representable in base-10 numbers with 3 significant digits and are rounded to 101 and 100 respectively. The result is now $\frac{101+100}{101-100} = 201$ which is representable but has a relative error of $\frac{2009-201}{2009} = \frac{1808}{2009}$, about 90%. This huge error is due entirely to the rounding that occurred before the subtraction took place and is not the fault of the subtraction operation (it can work only with what it’s given).

This error is called cancellation error because if the first $k$ digits of the operands are equal then the subtraction operation sets the first $k$ leading digits of the result to zero. When this result is normalised it contains only $t - k$ significant digits.

It can be shown that the relative propagated error of subtraction is

$$E_{S_r} \leq 2u \max\{|x|, |y|\} \frac{1}{|x - y|}.$$  

$E_{S_r}$ will be large if $x$ is close to $y$. Also, the larger $\max\{|x|, |y|\}$ is, the greater is $E_{S_r}$.

**Theorem 3.13 (Cancellation Error).** In a base-$b$ floating point system with $t$ significant digits, the relative error in $\text{round}(x - y)$ can be as large as $b - 1$.

This tells us that in base 2 ($b = 2$ so $b - 1 = 1$) the relative error can be as much as 1, *i.e.*, 100% out.

### 4. Condition and stability

We must separate the difficulties associated with a problem which requires a numerical solution from those associated with a computational algorithm which is used to obtain the numerical solution.

These difficulties are described by the condition of a problem and the stability of an algorithm.

#### 4.1. Condition.

Consider the problem of solving the system of two linear equations:

\[
\begin{align*}
(A) \quad (2.0000)x + (0.6667)y & = 2.0000 \\
(1.0000)x + (0.3333)y & = 1.0000
\end{align*}
\]

If we subtract twice the second equation from the first, we obtain $(0.0001)y = 0.0000$ which yields the unique solution

\[
\begin{align*}
x & = 1.0000 \\
y & = 0.0000
\end{align*}
\]

Now consider the system

\[
\begin{align*}
(B) \quad (2.0000)x + (0.6666)y & = 2.0000 \\
(1.0000)x + (0.3333)y & = 1.0000
\end{align*}
\]

which differs from (A) only in the fourth significant digit of the coefficient 0.6667. This system has rank one (the coefficients of the first equation are twice those of the second equation) and so has an infinite number of solutions. The system reduces to the equations

\[
\begin{align*}
x & = 1.0000 - (0.3333)\lambda \\
y & = \lambda
\end{align*}
\]

giving a solution for every possible choice of real number $\lambda$.

The point here is that a “small” perturbation in a single coefficient of (A) changes the problem from one with a unique solution to one with an infinity of solutions. This is a mathematical property of the system (A) which is completely independent of any algorithm used.

**Definition 3.14.** The condition of a problem $p$ is a measure of the change in the solution $s$ to a change in the problem $p$.

More precisely, assume we have a way $\|\|$ to measure the “size” $\|p\|$ of a problem $p$. 7
Definition 3.15. If \( p \) is changed to \( p + \Delta p \), where \( \Delta p \) is small, and the new solution is \( s + \Delta s \), then the condition number of \( p \) is
\[
\text{cond}(p) = \max_{\Delta p \neq 0} \frac{||\Delta s||}{||\Delta p||}.
\]
A more useful definition is:

Definition 3.16. The relative condition of a problem is defined as
\[
\text{cond}_{\text{rel}}(p) = \max_{\Delta p \neq 0} \frac{||\Delta s||/||s||}{||\Delta p||/||p||}.
\]
This is a non-negative real number which measures the relative change in the solution due to a relative change in the problem data. If this number is small, then the problem is well-conditioned. Otherwise, it is ill-conditioned.

Note 3.17. We can make the following points:

- The condition of a problem is a property of the problem and does not depend on the behaviour of the algorithm that solves \( p \).
- This is a very general (and vague) definition and needs to be refined for different problem types.

Example 3.18 (Function Evaluation). Suppose we want to evaluate a given function \( f(x) \) for any value of \( x \). We regard \( x \) as the problem data \( p \) and we want to see how the solution \( s = f(x) \) changes with \( x \).

The relative condition for this problem is
\[
\text{cond}_{\text{rel}}(p) = \left| \frac{f(x + \delta x) - f(x)}{f(x)} \right| \approx \frac{|x f'(x)|}{|f(x)|},
\]
for small \( \delta x \).

Let us look at some particular functions:

1. Let \( f(x) = \frac{10}{1-x^2} \) so \( f'(x) = 10(-1)(1-x^2)^{-2}(-2x) = \frac{20x}{(1-x^2)^2} \). Then \( f \) has relative condition
\[
\frac{2x^2}{|1-x^2|}.
\]
This problem is ill-conditioned for values of \( |x| \) near 1 (why?). Notice that the relative condition changes with \( x \).

2. \( f(x) = \sqrt{x} \) has relative condition \( \frac{1}{2} \) which means that calculating the square root is a well-conditioned problem because the change in the solution is actually smaller than the change in the problem. For example, to 32 significant digits,
\[
\sqrt{2.0000000} = 1.4142135623730950488016887242097
\]
\[
\sqrt{2.0000001} = 1.414213597728436661873377512028
\]
Calculating cond\(_{\text{rel}}\)(\( p \)) for these numbers gives \( \approx 0.5 \) (check this yourself).

3. \( f(x) = e^x \) has relative condition \( x \) (show this yourself). Is this ill- or well-conditioned?

Thus the relative condition depends on the particular function.

Example 3.19 (Root finding). The roots of \((x-1)^4 = 0\) are \( x_1 = x_2 = x_3 = x_4 = 1 \) (a single repeated root of multiplicity 4).

However, \((x-1)^4 = 10^{-8}\) has roots which satisfy \((x-1)^2 = \pm 10^{-4}\). Hence \( x-1 = \pm \sqrt[4]{\pm 10^{-2}} \) and we get the roots (where we write \( i = \sqrt{-1} \)) \( x_1 = 1+10^{-2} = 1.01, x_2 = 1-10^{-2} = 0.99, x_3 = 1+10^{-2}i = 1+0.01i, x_4 = 1-10^{-2}i = 1-0.01i \).

These are four distinct roots, each 0.01 away from 1 (in the directions E, W, N and S respectively), on the Argand diagram of complex numbers \( \mathbb{C} \).

A change of \( 10^{-8} \) in the problem has caused a change of \( 10^{-2} \) in the solutions \( x_1, \ldots, x_4 \) so the problem has a condition number \( 10^{-2}/10^{-8} = 10^6 \). This problem is ill-conditioned.

\[\text{In economics, } x f'(x)/f(x) \text{ is sometimes called the elasticity of } f.\]
4.2. Stability. An unstable algorithm is one which gives accurate solutions for some instances of a problem and inaccurate solutions for others. This is a description rather than a definition because stability means different things in different circumstances. We now give some classic examples that illustrate instability and show how the instability may be eliminated.

Example 3.20 (Roots of a Quadratic). Calculate the two roots of \( ax^2 + bx + c = 0 \). This has the solution \( x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \).

Let us consider the following code:

```python
import math
def roots(a, b, c):
    s = math.sqrt(b*b - 4.0*a*c)
    x1 = (-b + s) / (2.0*a)
    x2 = (-b - s) / (2.0*a)
    return (x1, x2)
```

If this code is run with \( a = 1.0, c = 2.0 \) and single precision instead of the double precision suggest, we get the following results, for various values of \( b \), rounded to a precision of 5 decimal digits:

<table>
<thead>
<tr>
<th>( b )</th>
<th>( s )</th>
<th>Computed ( x_1 )</th>
<th>True ( x_1 )</th>
<th>Relative Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2123</td>
<td>4.3781</td>
<td>-0.4171</td>
<td>-0.41708</td>
<td>0.004</td>
</tr>
<tr>
<td>52.123</td>
<td>52.046</td>
<td>-0.0385</td>
<td>-0.038399</td>
<td>2.9</td>
</tr>
<tr>
<td>1212.3</td>
<td>1212.3</td>
<td>0.0</td>
<td>-0.0016498</td>
<td>100.0</td>
</tr>
</tbody>
</table>

The last result is completely wrong: 100% error. The problem would appear to be the subtraction of nearly-equal numbers \( b \) and \( \sqrt{b^2 - 4ac} \) when calculating \( x_1 \). However, as we saw in Chapter 2, this indicates (propagation) errors in previous calculations, in this case \( s \).

We can patch up this code by noting that since \( ax^2 + bx + c \) has roots \( x_1, x_2 \), so it factorises as

\[
ax^2 + bx + c = a(x - x_1)(x - x_2) = ax^2 - a(x_1 + x_2)x + ax_1x_2.
\]

Comparing the \( x \) and constant coefficients gives

\[
b = -a(x_1 + x_2) \quad \text{and} \quad c = ax_1x_2.
\]

It follows that \( x_1x_2 = c/a \). This fact allows us to avoid the subtraction operation.

Example 3.21 (Evaluating \( e^x \) using its Taylor series expansion). The Taylor series expansion of \( e^x \) is

\[
e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}
\]

If we use (a truncated form of) this series to calculate \( e^x \) to machine single precision we get the following results:

<table>
<thead>
<tr>
<th>( x )</th>
<th>Error</th>
<th>Rel.Err(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-11.00000000</td>
<td>0.00000000</td>
<td>0.00000001</td>
</tr>
<tr>
<td>-10.00000000</td>
<td>0.00000000</td>
<td>0.00000001</td>
</tr>
<tr>
<td>1.00000000</td>
<td>0.00000000</td>
<td>0.00000001</td>
</tr>
<tr>
<td>10.00000000</td>
<td>0.00000000</td>
<td>0.00000001</td>
</tr>
</tbody>
</table>

It can be seen that this algorithm is very unstable for negative values of \( x \). To avoid these negative values we note that \( e^{-x} = 1/e^x \); so when \( x \) is negative we calculate \( e^{|x|} \) and then \( 1/e^{|x|} \).
5. Termination Criteria

Often, one would like to stop the computation at some point. Considering one can keep running many numerical methods ad infinitum, in principle, we need to build in some rule(s) to stop. These are often called termination criteria or stopping rules.

Example 3.22. For example, an algorithm to calculate the sum $S = \sum_{i=1}^{n} |x_i|$ could sort the numbers by their absolute values, start with the largest absolute value, and stop once the absolute value drops below a threshold.

In the case of zero-finding algorithms, we know that $\lim_{k \to \infty} f(x_k) = 0$, so we can test explicitly for the closeness of $f(x_k)$ to 0.

However, in iterative algorithms for zero-finding, optimisation or any other purpose, we do not know $\lim_{k \to \infty} x_k$ (this is what the algorithm is trying to find) so we must test the sequence $(x_k)$ in the Cauchy sequence sense: we test for how close together are the successive approximations $x_{k-1}$ and $x_k$. That is, stop when $\|x_{k+1} - x_k\| \leq \varepsilon_x$.

This is a valid thing to do in a complete metric space, where every Cauchy sequence converges.

These mathematical stopping rules must be modified to take into account the effects of finite-precision floating point arithmetic.

Let us examine a test that is too often seen in textbooks and amateur programs:

\[
\text{if } |x_k - x_{k-1}| \leq 10^{-6} \text{ then stop}
\]

If $|x_k| = 1000$ then the floating point number spacing around $x_k$ is $\varepsilon_m \times |x_k| = 10^{-7} \times 10^3 = 10^{-4}$ (recall that $\varepsilon_m$ denotes machine epsilon). This means that the two floating point numbers $x_k$, $x_{k-1}$ can never get closer than $10^{-4}$ and so passing this test is impossible. If this was the only test used in the algorithm then it would never stop.

If $|x_k| = 0.001$ then the floating point number spacing around $x_k$ is $10^{-7} \times 10^{-3} = 10^{-10}$ and so the test is valid but crude by a factor of $10^4$.

This example shows that we must account for floating point number spacing when choosing $\varepsilon_x$. We do this by making sure that the test never requires two numbers to be closer than the number spacing. From Chapter 2 we know that the number spacing around $x_k$ is $\varepsilon_m |x_k|$. Thus, the test

\[
\text{if } |x_k - x_{k-1}| \leq 4.0 \times \varepsilon_m \times |x_k| \text{ then stop}
\]

is valid for all magnitudes of $x_k$. The floating point number 4.0 in the test is a safety factor. This test ensures that we get almost full precision, i.e., the last two terms of the sequence are as close as they can possibly be in the floating point number system being used.

Sometimes we may not need convergence to full precision because a crude approximation $\varepsilon$ to the limit will suffice. This is elegantly handled by this modified test:

\[
\text{if } |x_k - x_{k-1}| \leq \varepsilon + 4.0 \times \varepsilon_m \times |x_k| \text{ then stop}
\]

To see how this test works, suppose that $|x_k| = 0.1$ and we want to stop when successive terms of the sequence get within $10^{-3}$ of each other. Set $\varepsilon = 10^{-3}$ and the test becomes

\[
\text{if } |x_k - x_{k-1}| \leq 10^{-3} + 4.0 \times 10^{-7} \times 10^{-1} \approx 10^{-3} \text{ then stop}
\]

Also, the test correctly deals with the case if $\varepsilon$ is set too small. Suppose, in the example above, that $\varepsilon = 10^{-10}$. This is an impossible requirement because the machine spacing is $10^{-7} \times 10^{-1} = 10^{-8}$. The test is now

\[
\text{if } |x_k - x_{k-1}| \leq 10^{-10} + 4.0 \times 10^{-7} \times 10^{-1} = 4.01 \times 10^{-8} \text{ then stop}
\]
and $4.01 \times 10^{-8}$ is greater than the spacing about $x_k = 10^{-1}$.

The modified test ($\ast$) above takes account both of the user’s desired tolerance and the floating point number spacing at the relevant part of the number line. It should be used in implementing any numerical algorithm that uses a tolerance test.

It may happen that an iterative algorithm may not pass a tolerance test in a “reasonable” time for our purposes. This may indicate some effect such as cycling or divergence (see later), or it may simply mean convergence is happening too slowly for our algorithm to pass a tolerance test in the time we have given it.

To prevent an algorithm running for longer than we want to, we generally pick a maximum number of iterations we will allow the algorithm to run for, called $maxits$ in our algorithms later, and wrap the algorithm inside a “for $k := 1$ to $maxits$” loop, thus ensuring no more than $maxits$ iterations are performed.

**Remark 3.23.** We will look at specific examples of stopping rules as we encounter particular algorithms in the next few chapters. Sometimes, these may disappoint sophisticated number analysts. Especially in machine learning, there has been a recent shift towards very crude stopping rules, e.g. “100 epochs” (which stands for a number of iteration equal to 100 times the dimension), “2 hours,” or even “2 seconds”. This is quite understandable: In situations, where new data are streaming in frequently (e.g. every second), there may be more benefit in re-running the computation with the updated data, than in finishing the computation using the possibly out-dated data to a better precision. The handling of such trade-offs is the subject of a recent patent application Marecek et al. [2015].

### 5.1. A Summary.

Implementing numerical methods in a programming language so that they run correctly is not always easy. Here are some facts and questions that may be worth reminding yourself of in the translation of an algorithm to a program:

Numbers have finite precision. With floating-point numbers, rules of algebra do not always work. Where could you reach the limits of machine epsilon, mantissa precision, or largest and smallest floating point numbers? Subtraction of nearly-equal numbers and division by small numbers, may cause difficulties, as they may lead to loss of precision. Use a sensible tolerance test.

One should also note that many approximations may occur before a calculation begins.

**Modelling:** may be approximate in that it omits some features to make it easier to work with, e.g., assume 0 transaction costs in equity trading, independence of default in constructing mortgage-backed securities, sinusoidal demands and currents in power systems, etc.

**Empirical measurements:** accuracy of data may be affected by limited precision of measuring instruments, small sample size, systematic bias, etc.

These issues may be outside of our scope, but are at least as important as the issues of numerical analysis.
Bibliography

