INTRODUCTION TO NUMERICAL ANALYSIS WITH C PROGRAMS

Attila Máté

Brooklyn College of the City University of New York

July 2004

Last Revised: August 2014

Typeset by $\mathcal{AMS}$-TEX

CONTENTS

PREFACE

These notes started as a set of handouts to the students while teaching a course on introductory numerical analysis in the fall of 2003 at Brooklyn College of the City University of New York. The notes rely on my experience of going back over 25 years of teaching this course. Many of the methods are illustrated by complete C programs, including instructions how to compile these programs in a Linux environment. These programs can be found at

`http://www.sci.brooklyn.cuny.edu/~mate/nml_progs/numanal_progs.tar.gz`

They do run, but many of them are works in progress, and may need improvements. While the programs are original, they benefited from studying computer implementations of numerical methods in various sources, such as [AH], [CB], and [PTVF]. In particular, we heavily relied on the array-handling techniques in C described, and placed in the public domain, by [PTVF]. In many cases, there are only a limited number of ways certain numerical methods can be efficiently programmed; nevertheless, we believe we did not violate anybody's copyright (such belief is also expressed in [PTVF, p. xvi] by the authors about their work).

New York, New York, July 2004
Last Revised: August 25, 2014

Attila Máté

1. FLOATING POINT NUMBERS

A floating point number in binary arithmetic is a number of form $2^e \cdot 0.m$, where e and m are integers written in binary (that is, base 2) notation, and the dot in $0.m$ is the binary "decimal" point, that is, if the digits of m are $m_1, m_1, \ldots, m_k$ ($m_i = 0$ or 1), then $0.m = m_1 \cdot 2^{-1} + m_1 \cdot 2^{-2} + \ldots m_1 \cdot 2^{-k}$. Here e is called the exponent and m is called the mantissa. In the standard form of a floating point number it is assumed that $m_1 = 1$ unless all digits of m are 0; a nonzero number in nonstandard form can usually be brought into standard form by lowering the exponent and shifting the mantissa (the only time this cannot be done is when this would bring the exponent e below its lower limit – see next). In the IEEE standard,[1] the mantissa m of a (single precision) floating point is 23 bits. For the exponent e, we have $-126 \leq e \leq 127$. To store the exponent, 8 bits are needed. One more bit is needed to store the sign, so altogether 40 bits are needed to store a single precision floating point number. Thus a single precision floating point number roughly corresponds to a decimal number having seven significant digits. Many programming languages define double precision numbers and even long double precision numbers.

As a general rule, when an arithmetic operation is performed the number of significant digits is the same as those in the operands (assuming that both operands have the same number of significant digits). An important exception to this rule is when two numbers of about the same magnitude are subtracted. For example, if

$$x = .7235523 \qquad \text{and} \qquad y = .7235291,$$

both having seven significant digits, then the difference

$$x - y = .0000232$$

has only three significant digits. This phenomenon is called loss of precision. Whenever possible, one must avoid this kind of loss of precision.

When evaluating an algebraic expression, this is often possible by rewriting the expression. For example, when solving the quadratic equation

$$x^2 - 300x + 1 = 0,$$

the quadratic formula gives two solutions:

$$x = \frac{300 + \sqrt{89996}}{2} \qquad \text{and} \qquad x = \frac{300 - \sqrt{89996}}{2}.$$

There is no problem with calculating the first root, but with the second root there is clearly a loss of significance, since $\sqrt{89996} \approx 299.993$. It is easy to overcome this problem in the present case. The second root can be calculated as

$$\begin{aligned} x = \frac{300 - \sqrt{89996}}{2} &= \frac{300 - \sqrt{89996}}{2} \cdot \frac{300 + \sqrt{89996}}{300 + \sqrt{89996}} \\ &= \frac{2}{300 + \sqrt{89996}} \approx .003, 333, 370, 371, 193, 4 \end{aligned}$$

and the loss of significance is avoided.

[1] IEEE is short for Institute of Electric and Electronics Engineers.

Problems

1. Calculate $x - \sqrt{x^2-1}$ for $x = 256,000$ with 10 significant digit accuracy. Avoid loss of significant digits.

Solution. We cannot use the expression given directly, since x and $\sqrt{x^2-1}$ are too close, and their subtraction will result in a loss of precision. To avoid this, note that

$$x - \sqrt{x^2-1} = \left(x - \sqrt{x^2-1}\right) \cdot \frac{x + \sqrt{x^2-1}}{x + \sqrt{x^2-1}} = \frac{1}{x + \sqrt{x^2-1}}.$$

To do the numerical calculation, it is easiest to first write that $x = y \cdot 10^5$, where $y = 2.56$. Then

$$\frac{1}{x + \sqrt{x^2-1}} = \frac{1}{y + \sqrt{y^2 - 10^{-10}}} \cdot 10^{-5} \approx 1.953,125,000,0 \cdot 10^{-6}.$$

2. Calculate $\sqrt{x^2+1} - x$ for $x = 1,000,000$ with 6 significant digit accuracy. Avoid the loss of significant digits.

Solution. We cannot use the expression given directly, since $\sqrt{x^2+1}$ and x are too close, and their subtraction will result in a loss of precision. To avoid this, note that

$$\sqrt{x^2+1} - x = \left(\sqrt{x^2+1} - x\right) \cdot \frac{\sqrt{x^2+1} + x}{\sqrt{x^2+1} + x} = \frac{1}{\sqrt{x^2+1} + x}.$$

To do the numerical calculation, it is easiest to first write that $x = y \cdot 10^6$, where $y = 1$. Then

$$\frac{1}{\sqrt{x^2+1} + x} = \frac{1}{\sqrt{y^2 + 10^{-12}} + y} \cdot 10^{-6} \approx 5.000,000,000 \cdot 10^{-7}.$$

3. Show how to avoid the loss of significance when solving the equation

$$x^2 - 1000x - 1 = 0.$$

4. Evaluate $e^{0.0002} - 1$ on your calculator. Hint: The best way to avoid the loss of precision is to use the Taylor series approximation of e^x. Using only two terms and the Lagrange remainder term, we have

$$e^x = 1 + x + \frac{x^2}{2} + e^{\xi}\frac{x^3}{6},$$

where ξ is between 0 and x. With $x = .0002 = 2 \cdot 10^{-4}$, we have $x^3 = 8 \cdot 10^{-12}$, and noting that e^{ξ} is very close to one, this formula will give accurate results up to 11 decimal places (i.e., the error is less than $5 \cdot 10^{-12}$).

5. Calculate

$$\sin 0.245735 - \sin 0.245712.$$

(Note that angles are measured in radians.) Hint: Rather than evaluating the sines of the angles given, it may be better to use the formula

$$\sin x - \sin y = 2\cos\frac{x+y}{2}\sin\frac{x-y}{2}$$

with $x = 0.245735$ and $y = 0.245712$. Then sine of a small angle may be more precisely evaluated using a Taylor series than using the sine function on your calculator. Note, however, that this approach does not avoid the loss of precision that results from calculating $x - y$. From what is given, this cannot be avoided.

6. Find $1 - \cos 0.008$ with 10 decimal accuracy.

Solution. Of course 0.008 means radians here. Using the value of $\cos 0.008$ here would lead to unacceptable loss of precision, since to value is too close to 1. Using the Taylor series of $\cos x$ gives a more accurate result:

$$\cos x = \sum_{n=0}^{\infty}(-1)^n \cdot \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

For $|x| \leq 1$ this is an alternating series, and so, when summing finitely many terms of the series, the absolute value of the error error will be less than that of the first omitted term. With $x = 0.008$, we have

$$\frac{x^6}{6!} < \frac{0.01^6}{720} = 10^{-12} \cdot \frac{1}{720} < 10^{-12} \cdot .00139 < 1.39 \cdot 10^{-15},$$

and so this term can safely be omitted. Thus, writing $x = 8 \cdot 10^{-3}$, with sufficient precision we have

$$1 - \cos x \approx x^2/2! - x^4/4! = 32 \cdot 10^{-6} - \frac{512}{3} \cdot 10^{-12} \approx 0.0000319998293$$

7. Find $1 - e^{-0.00003}$ with 10 decimal digit accuracy.

Solution. Using the value of $e^{-0.00003}$ would lead to an unnecessary and unacceptable loss of accuracy. It is much better to use the Taylor series of e^x with $x = -3 \cdot 10^{-5}$:

$$1 - e^x = 1 - \sum_{n=0}^{\infty}\frac{x^n}{n!} = -x - \frac{x^2}{2} - \frac{x^3}{6} - \dots .$$

For $x = -3 \cdot 10^{-5}$ this becomes an alternating series:

$$3 \cdot 10^{-5} - \frac{9 \cdot 10^{-10}}{2} + \frac{27 \cdot 10^{-15}}{6} - \dots .$$

When summing finitely many terms of an alternating series, the error will be smaller than the first omitted term. Since we allow an error no larger than $5 \cdot 10^{-11}$, the third term here can be safely omitted. Thus,

$$1 - e^{-0.00003} \approx 3 \cdot 10^{-5} - \frac{9 \cdot 10^{-10}}{2} = .000,029,999,55.$$

8. Calculate

$$\sum_{n=1}^{9999}\frac{1}{n^2}$$

Note. The issue here is that the proper order to calculate this sum is

$$\frac{1}{9999^2}+\frac{1}{9998^2}+\frac{1}{9997^2}+\frac{1}{9996^2}+\ldots+\frac{1}{2^2}+\frac{1}{1^2},$$

and not

$$\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{4^2}+\ldots+\frac{1}{9998^2}+\frac{1}{9999^2}.$$

To explain this, assume that the calculation used a floating point calculation with 10 significant digits in decimal arithmetic. When calculating the latter sum, after the first term is added, the size of the sum is at least $1 = 10^0$, and so no digits with place value less than 10^{-9} can be taken into account. Now, $1/9999^2 = 1.000,200,030,004,0 \cdot 10^{-8}$, so when one gets to this last term, only two significant digits of this term can be added. When one calculates the former sum, the first term added is $1/10000^2$, and all its ten significant digits can be taken into account. Of course, when one gets to the end of the calculation, all the digits smaller than 10^{-8} will be lost. However, when adding ten thousand numbers, the error in these additions will accumulate, and it will make a significant difference that the addition initially was performed greater precision (i.e., when adding the numbers by using the first method), since this will result in a smaller accumulated error.

This means that adding floating point numbers is not a commutative operation. Whenever practical, one should first add the smaller numbers so as to reduce the error.

Euler has proved that

$$\sum_{n=1}^{\infty}\frac{1}{n^2}=\frac{\pi^2}{6}.$$

We have $\pi^2/6 \approx 1.644,934,066,848$. If we calculate the sum

$$\sum_{n=1}^{9,999,999}\frac{1}{n^2}$$

in the forward order (i.e., by adding the terms for $n = 1$, then $n = 2$, then $n = 3$, ...), we obtain $1.644,725,322,723$ and if we calculate it in the backward order (i.e., by adding the terms for $n = 9,999,999$, then $n = 9,999,998$, then $n = 999,997$, ...), we obtain $1.644,933,938,980$. The former differs from the value of $\pi^2/6$ by $2.08744 \cdot 10^{-4}$, while the latter differs from it by $1.27868 \cdot 10^{-7}$, showing that the latter approximation is much better.

A computer experiment. The following computer experiment was done to produce the above results on a computer running Fedora Core 5 Linux, using the compiler GCC version `gcc (GCC) 4.1.0 20060304 (Red Hat 4.1.0-3)`.[2] The program `addfw.c` carried out the addition in forward order:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 main()
 5 {
 6   float sum, term;
 7   int i;
 8   sum=0.0;
 9   for(i=1; i < 10000000; i++)
10   {
```

[2]GCC originally stood for the GNU C Compiler (version 1.0 was released in 1987), but now that it can also handle many languages in addition to C, GCC stands for the GNU Compiler Collection.

```
11      term = (float) i;
12      term = term * term;
13      term = 1/term;
14      sum += term;
15   }
16   printf("The sum is %.12f\n", sum);
17 }
```

Here the number at the beginning of each line is a line number, and it is not a part of the file. In line 6, `float` is the type of a single precision floating point number, giving about seven significant decimal digits of precision. (the type double gives about 14 significant digits, and long double about 21 significant digits)[3]. In calculating the ith term of the series to be added, in line 11 the variable `i` is converted to floating point before its square is calculated in line 12; calculating the square of an integer would cause an overflow for large values of i. The program addbw.c was used to carry out the addition in the backward order;

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 main()
 5 {
 6   float sum, term;
 7   int i;
 8   sum=0.0;
 9   for(i=9999999; i >= 1; i--)
10   {
11      term = (float) i;
12      term = term * term;
13      term = 1/term;
14      sum += term;
15   }
16   printf("The sum is %.12f\n", sum);
17 }
```

Finally, in calculating $\pi^2/6$, we used the fact that $\pi/4 = \arctan 1$, and then used the C-function `atan2l` to do the calculation in the file `pi.c`:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 main()
 5 {
 6   long double pisqo6;
 7   double toprint;
 8   pisqo6 = atan2l(0.0, -1.0);
 9   pisqo6 *= pisqo6;
10   pisqo6 /= (long double) 6.0;
11   toprint = (double) pisqo6;
12   /* The conversion of the long double
13   value pisqoe6 to the double value toprint
14   is necessary, since printf apparently is
15   unable to handle long double variables;
```

[3]These values are only rough guesses, and are not based on the technical documentation of the GCC compiler

```
16   this may be a bug */
17   printf(" pi^2/6 is %.12f\n", toprint);
18 }
```

In this last program, the long double variable `pisqo6` is used to calculate $\pi^2/6$. First the `atan2` function, or, rather, the `atan2l` function (the final `l` here stands for "long," to indicate that this function takes arguments as `long double` floating point numbers, and gives its result as as `long double`. The meaning of the `atan2(y,x)` function is to give the polar angle of the point `(x,y)` in the plane; and, of course, the polar angle of the point $(-1, 0)$, given by `atan2(0.0,-1.0)` is π.[4] In line 11, the double long variable `pisqo6` is cast to (i.e., converted to) a double variable, since the function `printf` does not seem to work with long double variables.[5] In compiling these programs without optimization, we used the following `makefile`:

```
 1 all : addfw addbw pi
 2 addfw : addfw.o
 3         gcc -o addfw -s -O0 addfw.o -lm
 4 addfw.o : addfw.c
 5         gcc -c -O0 addfw.c
 6 addbw : addbw.o
 7         gcc -o addbw -s -O0 addbw.o -lm
 8 addbw.o : addbw.c
 9         gcc -c -O0 addbw.c
10 pi : pi.o
11         gcc -o pi -s -O4 pi.o -lm
12 pi.o : pi.c
13         gcc -c -O4 pi.c
14 clean : addfw addbw pi
15         rm *.o
```

This `makefile` does the compilation of the programs after typing the command

```
$ make all
```

Here the dollar sign `$` at the beginning of the line is the prompt given by the computer to type a command; in an actual computer, the prompt is most likely different (in fact, in Unix or Linux, the prompt is highly customizable, meaning that the user can easily change the prompt in a way she likes). The `makefile` contains a number of targets on the left hand-side of the colons in lines 1, 2, 4, 6, 8, 10, 12, 15. These targets are often names of files to be produced (as in lines 2 and 4, etc.), but they can be abstract objectives to be accomplished (as in line 1 – to produce all files – and in line 15 – to clean up the directory by removing unneeded files). On the right-hand side of the colon there are file names or other targets needed for (producing) the target. After each line naming the target, there are zero or more lines of rules describing how to produce the target. Each rule starts with a tab character (thus lines 3 and 5, etc. each start with a tab character, and not with a number of spaces), and then a computer command follows. No rule is needed to produce the target `all`, what is needed for `all` to be produced that the three files mentioned after the colon should be present. No action is taken if the target is already present, unless the target is a file name. In this case, it is checked if the target is newer than the files needed to produce it (as described on the right-hand side of the colon). If the target is older than at least one of the files needed to produce it, the rule is applied to produce a new target.[6]

---

[4] The `atan2` function is more practical for computer calculations than the *arctan* function of mathematics. Of course, *arctany* can be calculated as `atan2(y, 1.0)`.

[5] This bug has been fixed in newer releases of GCC; see

http://gcc.gnu.org/ml/gcc-bugs/2006-11/msg02440.html

The version we used was `gcc 4.1.0`.

[6] The idea is that the tasks described in the rules should not be performed unnecessarily.

For example, the rule needed to produce the target `addfw.o`, the object file (i.e., the compiled program) of the C program `addfw.c`, one uses the command

```
$ gcc -c -O0 addfw.c
```

This command specifies that the file `addfw.c` should be compiled with the optimization level 0, as specified by the `-O0` option (here the first letter is capital "oh" and the second letter is 0, the level of optimization). This means that no optimization is to be done. The rule used the produce the target file `addfw` is

```
$ gcc -o addfw -s -O0 addfw.o -lm
```

Here the `-o` option specifies the resulting object file should be called `addfw`, and the `-lm` option specifies that the object file should be linked to the mathematics library (denoted by the letter `m`) of the C compiler. The file `addfw` is an executable program, and one can run it simply by typing the command

```
$ addfw
```

When one works on this program, after each change in one of the C program files, one can type to command

```
$ make all
```

to recompile the executable programs. Only those programs will be recompiled for which recompilation is needed; that is, only those programs for which the source file (the C program), or one of the source files, have been changed. After finishing work on this program, one can type

```
$ make clean
```

which will remove all files of form `*.o`, that is those files whose name ends in `.o` (`*` is called a wild card character, that can be replaced with any string of characters.)

If the above programs are compiled with optimization, using the `makefile`

```
 1 all: addfw addbw pi
 2 addfw : addfw.o
 3         gcc -o addfw -s -O4 addfw.o -lm
 4 addfw.o : addfw.c
 5         gcc -c -O4 addfw.c
 6 addbw : addbw.o
 7         gcc -o addbw -s -O4 addbw.o -lm
 8 addbw.o : addbw.c
 9         gcc -c -O4 addbw.c
10 pi : pi.o
11         gcc -o pi -s -O4 pi.o -lm
12 pi.o : pi.c
13         gcc -c -O4 pi.c
14 clean : addfw addbw pi
15         rm *.o
```

the result will be different. In lines 4, 5, 7, etc., it is now indicated by the option `-O4` (minus capital "oh" four) that optimization level 4 is being used. In this case, the output of the program `addfw` and `addbw` will be identical: $1.644,933,966,848$. This differs from the value $1.644,934,066,848$ by about $1.00000 \cdot 10^{-7}$. This means that the optimizing compiler notices the problem with adding the terms of the sequence in the forward orders, and corrects this problem.[7]

[7] What the optimizing compiler `gcc` actually does is not clear to me. It must be doing something more complicated than just adding the terms of the series in reverse order, since the result is somewhat better that would be obtained by doing the latter.

2. ABSOLUTE AND RELATIVE ERRORS

The absolute error when calculating a quantity x is the difference $x-\bar{x}$, where $\bar{x}$ is the calculated value of x. The relative error is defined as $(x-\bar{x})/x$. Since x is usually not known, one usually estimates the relative error by the quantity $(x-\bar{x})/\bar{x}$; so if the size of the absolute error is known, in this way one can calculate also the relative error.[8]

The error in evaluating a function $f(x,y)$ can be estimated as follows. If we write $\Delta x = x-\bar{x}$ and $\Delta y = y-\bar{y}$, then

$$f(x,y) - f(\bar{x},\bar{y}) \approx \frac{\partial f(\bar{x},\bar{y})}{\partial x}(x-\bar{x}) + \frac{\partial f(\bar{x},\bar{y})}{\partial y}(y-\bar{y})$$

The right-hand side represents the approximate error in the function evaluation $f(x,y)$.

We will apply this with $f(x,y)$ replaced with xy and x/y respectively, writing $E(\cdot)$ for the error. For multiplication we obtain

$$E(xy) \approx y\Delta x + x\Delta y.$$

For the relative error, this means

$$\frac{E(xy)}{xy} \approx \frac{\Delta x}{x} + \frac{\Delta y}{y}.$$

In other words, the relative error of multiplication is approximately the sum of the relative errors of the factors.

For division, we get a similar result:

$$E(x/y) \approx \frac{1}{y}\Delta x - \frac{x}{y^2}\Delta y,$$

so

$$\left|\frac{E(x/y)}{x/y}\right| \lessapprox \left|\frac{\Delta x}{x}\right| + \left|\frac{\Delta y}{y}\right|,$$

that is the relative error of the division is roughly[9] less than the sums of the absolute values of the relative errors of the divisor and the dividend.

The *condition* of a function describes how sensitive a function evaluation to relative errors in its argument: It is "defined" as

$$\max\left\{\left|\frac{f(x)-f(\bar{x})}{f(x)} \Big/ \frac{x-\bar{x}}{x}\right| : x-\bar{x} \quad \text{is "small"}\right\} \approx \left|\frac{xf'(x)}{f(x)}\right|,$$

where x is the fixed point where the evaluation of f is desired, and $\bar{x}$ runs over values close to x. This is not really a definition (as the quotation marks around the word *defined* indicate above), since it is not explained what *small* means; it is meant only as an approximate description.

Problems

1. Evaluate

$$\cos(2x+y^2)$$

for $x = 2 \pm 0.03$ and $y = 1 \pm 0.07$. (The arguments of cosine are radians, and not degrees.)

---

[8] Of course, one does not usually know the exact value of the error, since to know the exact error and the calculated value is to know x itself, since $x = \bar{x} + \epsilon$, where ϵ is the error.

[9] The word *roughly* has to be here, since the equation on which this derivation is based is only an approximate equation. The sign $\lessapprox$ means "approximately less than."

Solution. There is no problem with the actual calculation. With $x = 2$ and $y = 1$ we have

$$\cos(2x + y^2) = \cos 5 \approx 0.283,662.$$

The real question is, how accurate this result is? Writing

$$f(x, y) = \cos(2x + y^2),$$

we estimate the error of f by its total differential

$$df(x, y) = \frac{\partial f(x, y)}{\partial x} dx + \frac{\partial f(x, y)}{\partial y} dy = -2 \sin(2x + y^2)\, dx - 2y \sin(2x + y^2)\, dy,$$

where $x = 2$, $y = 1$, and $dx = \pm 0.03$ and $dy = \pm 0.07$, that is, $|dx| \le 0.03$ and $|dy| \le 0.07$.[10] Hence

$$\begin{aligned} |df(x, y)| &\le |-2 \sin(2x + y^2)||dx| + |-2y \sin(2x + y^2)||dy| \\ &\lessapprox 2 \cdot 0.959 \cdot 0.03 + 2 \cdot 0.959 \cdot 0.07 = 0.192 \end{aligned}$$

Thus $f(x, y) \approx 0.284 \pm 0.192$.

2. Evaluate

$$s = x \left(1 - \cos \frac{y}{x}\right)$$

for $x = 30 \pm 3$ and $y = 20 \pm 2$.

Solution. There is no problem with the actual calculation, since there is no serious loss of precision:

$$30 \cdot \left(1 - \cos \frac{20}{30}\right) \approx 6.42338.$$

The real question is, how accurate this result is? Writing

$$f(x, y) = x \left(1 - \cos \frac{y}{x}\right),$$

we estimate the error of f by its total differential

$$df(x, y) = \frac{\partial f(x, y)}{\partial x} dx + \frac{\partial f(x, y)}{\partial y} dy = \left(1 - \cos \frac{y}{x} - \frac{y}{x} \sin \frac{y}{x}\right) dx + \sin \frac{y}{x}\, dy,$$

where $x = 30$, $y = 20$, and $dx = \pm 3$ and $dy = \pm 2$, that is, $|dx| \le 3$ and $|dy| \le 2$.[11] Thus

$$|df(x, y)| \le \left|1 - \cos \frac{y}{x} - \frac{y}{x} \sin \frac{y}{x}\right| |dx| + \left|\sin \frac{y}{x}\right| |dy| \lessapprox 0.198 \cdot 3 + 0.618 \cdot 2 \approx 1.8.$$

Thus $f(x, y) \approx 6.42 \pm 1.8$.

3. Find the condition of $y = x^3$ near $x = 2$.

4. Find the condition of $y = \tan x$ near $x = 1.5$.

[10] It is more natural to write Δx and Δy for the errors of x and y, but in the total differential below one customarily uses dx, and dy.

[11] It is more natural to write Δx and Δy for the errors of x and y, but in the total differential below one customarily uses dx, and dy.

3. ROUNDOFF AND TRUNCATION ERRORS

The main types of errors are *roundoff* error, and *truncation* error. The former refers to the fact that real numbers, which are usually infinite decimals, are represented by a finite number of digits in the computer. The latter refers to the error committed in a calculation when an infinite process is replaced by a finite process of calculation. A simple example for this latter is when the sum of an infinite series is evaluated by adding finitely many terms in the series. More generally, continuous processes in the computer are represented by finite, discrete processes, resulting in truncation error.

A third type of error, error resulting from a mistake or blunder is an altogether different matter; such errors ought to be avoided in so far as possible, whereas roundoff and truncation errors cannot be avoided.

4. THE REMAINDER TERM IN TAYLOR'S FORMULA

In Taylor's formula, a function $f(x)$ is approximated by a polynomial

$$\sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

The goodness of this approximation can be measured by the remainder term $R_n(x,a)$, defined as

$$R_n(x,a) \stackrel{def}{=} f(x) - \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

To estimate $R_n(x,a)$, we need the following lemma.

LEMMA. *Let $n \geq 1$ be an integer. Let U be an open interval in $\mathbb{R}$ and let $f : U \to \mathbb{R}$ be a function that is $n+1$ times differentiable. Given any $b \in U$, we have*

$$\frac{d}{dx}R_n(b,x) = -\frac{f^{(n+1)}(x)(b-x)^n}{n!} \tag{1}$$

for every $x \in U$.

PROOF. We have

$$R_n(b,x) = f(b) - \sum_{k=0}^{n} f^{(k)}(x)\frac{(b-x)^k}{k!} = f(b) - f(x) - \sum_{k=1}^{n} f^{(k)}(x)\frac{(b-x)^k}{k!}.$$

We separated out the term for $k=0$ since we are going to use the product rule for differentiation, and the term for $k=0$ involves no product. We have

$$\begin{aligned}
\frac{d}{dx}R_n(b,x) &= \frac{d}{dx}f(b) - \frac{d}{dx}f(x) \\
&\quad - \sum_{k=1}^{n}\left(\frac{df^{(k)}(x)}{dx}\frac{(b-x)^k}{k!} + f^{(k)}(x)\frac{d}{dx}\frac{(b-x)^k}{k!}\right) \\
&= -f'(x) - \sum_{k=1}^{n}\left(f^{(k+1)}(x)\frac{(b-x)^k}{k!} + f^{(k)}(x)\frac{-k(b-x)^{k-1}}{k!}\right) \\
&= -f'(x) - \sum_{k=1}^{n}\left(f^{(k+1)}(x)\frac{(b-x)^k}{k!} - f^{(k)}(x)\frac{(b-x)^{k-1}}{(k-1)!}\right).
\end{aligned}$$

Writing

$$A_k = f^{(k+1)}(x)\frac{(b-x)^k}{k!}$$

for k with $0 \le k \le n$, the sum (i.e., the expression described by $\sum_{k=1}^{n}$) on the right-hand side equals

$$\begin{aligned}\sum_{k=1}^{n}(A_k - A_{k-1}) &= (A_1 - A_0) + (A_2 - A_1) + \dots (A_n - A_{n-1}) \\ &= A_n - A_0 = f^{(n+1)}(x)\frac{(b-x)^n}{n!} - f'(x).\end{aligned}$$

Substituting this in the above equation, we obtain

$$\frac{d}{dx}R_n(b,x) = -f'(x) - \left(f^{(n+1)}(x)\frac{(b-x)^n}{n!} - f'(x)\right) = -f^{(n+1)}(x)\frac{(b-x)^n}{n!},$$

as we wanted to show.

COROLLARY 1. *Let $n \ge 1$ be an integer. Let U be an open interval in $\mathbb{R}$ and let $f : U \to \mathbb{R}$ be a function that is $n+1$ times differentiable. For any any $a, b \in U$ with $a \ne b$, there is a $\xi \in (a,b)$ (if $a < b$) or $\xi \in (b,a)$ (if $a > b$) such that*

$$R_n(b,a) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(b-a)^{n+1}. \tag{2}$$

PROOF. For the sake of simplicity, we will assume that $a < b$.[12] We have $b - a \ne 0$, and so the equation

$$R_n(b,a) = K \cdot \frac{(b-a)^{n+1}}{(n+1)!} \tag{3}$$

can be solved for K. Let K be the real number for which this equation is satisfied, and write

$$\phi(x) = R_n(b,x) - K \cdot \frac{(b-x)^{n+1}}{(n+1)!}$$

Then ϕ is differentiable in U; as differentiability implies continuity, it follows that f is continuous on the interval $[a,b]$ and differentiable in (a,b).[13] As $\phi(a) = 0$ by the choice of K and $\phi(b) = 0$ trivially, we can use Rolle's Theorem to obtain the existence of a $\xi \in (a,b)$ such that $\phi'(\xi) = 0$. Using (1), we can see that

$$0 = \phi'(\xi) = -\frac{f^{(n+1)}(\xi)(b-\xi)^n}{n!} - K \cdot \frac{-(n+1)(b-\xi)^n}{(n+1)!}.$$

Noting that $\frac{(n+1)}{(n+1)!} = \frac{1}{n!}$ and keeping in mind that $\xi \ne b$, we obtain $K = f^{n+1}(\xi)$ from here. Thus the result follows from (3).

[12] This assumption is never really used except that it helps us avoid circumlocutions such as $\xi \in (a,b)$ if $a < b$ or $\xi \in (b,a)$ if $b < a$.

[13] Naturally, ϕ is differentiable also at a and b, but this is not needed for the rest of the argument.

Note. The above argument can be carried out in somewhat more generality. Let $g(x)$ be a function that is differentiable on (a,b), $g'(x) \neq 0$ for $x \in (a,b)$, and $g(b) = 0$. Note that, by the Mean-Value Theorem,

$$-\frac{g(a)}{b-a} = \frac{g(b)-g(a)}{b-a} = g'(\eta)$$

for some $\eta \in (a,b)$. Since we have $g'(\eta) \neq 0$ by our assumptions, it follows that $g(a) \neq 0$. Instead of (3), determine K now such that

$$R_n(b,a) = Kg(a). \tag{4}$$

As $g(a) \neq 0$, it is possible to find such a K. Write

$$\phi(x) = R_n(b,x) - Kg(x).$$

We have $\phi(a) = 0$ by the choice of K. We have $\phi(b) = 0$ since $R_n(b,b) = 0$ and $g(b) = 0$ (the latter by our assumptions). As ϕ is differentiable on (a,b), there is a $\xi \in (a,b)$ such that $\phi'(\xi) = 0$. Thus, by (1) we have

$$0 = \phi'(\xi) = -\frac{f^{(n+1)}(\xi)(b-\xi)^n}{n!} - Kg'(\xi).$$

As $g'(\xi) \neq 0$ by our assumptions, we can determine K from this equation. Substituting the value of K so obtained into (4), we can see that

$$R_n(b,a) = -\frac{f^{(n+1)}(\xi)(b-\xi)^n}{n!} \cdot \frac{g(a)}{g'(\xi)}.$$

Note that in the argument we again assumed that $a < b$, but this was unessential. Further, note that the function g can depend on a and b. We can restate the result just obtained in the following

Corollary 2. *Let $n \geq 1$ be an integer. Let U be an open interval in $\mathbb{R}$ and let $f : U \to \mathbb{R}$ be a function that is $n+1$ times differentiable. For any any $a, b \in U$ with $a < b$, let $g_{a,b}(x)$ be a function such that $g_{a,b}$ is continuous on $[a,b]$ and differentiable on (a,b). Assume, further, that $g_{a,b}(b) = 0$ and $g'_{a,b}(x) \neq 0$ for $x \in (a,b)$. Then there is a $\xi \in (a,b)$ such that*

$$R_n(b,a) = -\frac{f^{(n+1)}(\xi)(b-\xi)^n}{n!} \cdot \frac{g_{a,b}(a)}{g'_{a,b}(\xi)}.$$

If $b < a$, then the same result holds, except that one needs to write (b,a) instead of (a,b) and $[b,a]$ instead of $[a,b]$ for the intervals mentioned (but the roles of a and b should not otherwise be interchanged).

This result is given in [B]; see also [M]. The Wikipedia entry [Wiki] also discusses the result (especially under the subheading *Mean value theorem*), without giving attributions. Taking $g_{a,b}(x) = (b-x)^r$ for an integer r with $0 < r \leq n+1$, we obtain that

$$R_n(b,a) = \frac{f^{(n+1)}(\xi)}{rn!}(b-\xi)^{n-r+1}(b-a)^r.$$

This is called the Roche–Schlömilch Remainder Term of the Taylor Series. Here ξ is some number in the interval (a,b) or (b,a); it is important to realize that the value of ξ depends on r. Taking $r = n+1$ here, we get formula (2); this is called Lagrange's Remainder Term of the Taylor Series. Taking $r = 1$, we obtain

$$R_n(b,a) = \frac{f^{(n+1)}(\xi)}{n!}(b-\xi)^n(b-a);$$

this is called Cauchy's Remainder Term of the Taylor Series.

The different forms of the remainder term of Taylor's series are useful in different situations. For example, Lagrange's Remainder Term is convenient for establishing the convergence of the Taylor series of e^x, $\sin x$, and $\cos x$ on the whole real line, but it is not suited to establish the convergence of the Taylor series of $(1+x)^\alpha$, where α is an arbitrary real number. The Taylor series of this last function is convergent on the interval $(-1, 1)$, and on this interval it does converge to the function $(1+x)^\alpha$ (this series is called the *Binomial Series*). This can be established by using Cauchy's Remainder Term.

5. LAGRANGE INTERPOLATION

Let $x_1, x_2, \ldots, x_n$ be distinct real numbers, and let $y_1, y_2, \ldots, y_n$ also be real numbers, these not necessarily distinct. The task of *polynomial interpolation* is to find a polynomial $P(x)$ such that $P(x_i) = y_i$ for i with $1 \le i \le n$. It is not too hard to prove that there is a unique polynomial of degree $n-1$ satisfying these conditions, while a polynomial of lower degree usually cannot be found. The polynomial interpolation problem was first solved by Newton. A different, elegant solution was later found by Lagrange. Here we consider the latter solution.

Lagrange considers the polynomials[14]

$$l_i(x) = \prod_{\substack{j=1 \\ j \ne i}}^{n} \frac{x - x_j}{x_i - x_j}$$

It is clear that $l_i(x)$ is a polynomial of degree n, since the numbers in the denominator do not depend on x. Further, for any integer j with $1 \le j \le n$ we have

$$l_i(x_j) = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{if } j \ne i. \end{cases}$$

Indeed, if $x = x_i$ then each of the fractions in the product expressing $l_i(x)$ is 1, and if $x = x_j$ for $j \ne i$ then one of the fractions in this product has a zero numerator. For this reason, the polynomial $P(x)$ defined as

$$P(x) = \sum_{i=1}^{n} y_i l_i(x)$$

satisfies the requirements; that is $P(x_i) = y_i$ for i with $1 \le i \le n$.

Example. Find a polynomial $P(x)$ of degree at most 3 such that $P(-1) = 7$, $P(2) = 3$, $P(4) = -2$, and $P(6) = 8$.

Solution. Writing $x_1 = -1$, $x_2 = 2$, $x_3 = 4$, and $x_4 = 6$, we have

$$l_1(x) = \frac{(x-2)(x-4)(x-6)}{(-1-2)(-1-4)(-1-6)} = -\frac{1}{105}(x-2)(x-4)(x-6),$$

$$l_2(x) = \frac{(x+1)(x-4)(x-6)}{(2+1)(2-4)(2-6)} = \frac{1}{24}(x+1)(x-4)(x-6),$$

$$l_3(x) = \frac{(x+1)(x-2)(x-6)}{(4+1)(4-2)(4-6)} = -\frac{1}{20}(x+1)(x-2)(x-6),$$

[14]These polynomials are sometimes called *Lagrange fundamental polynomials*.

$$l_4(x) = \frac{(x+1)(x-2)(x-4)}{(6+1)(6-2)(6-4)} = \frac{1}{56}(x+1)(x-2)(x-4).$$

Thus, the polynomial $P(x)$ can be written as

$$\begin{aligned} P(x) &= 7l_1(x) + 3l_2(x) - 2l_3(x) + 8l_4(x) = -\frac{7}{105}(x-2)(x-4)(x-6) \\ &+ \frac{3}{24}(x+1)(x-4)(x-6) - \frac{2}{20}(x+1)(x-2)(x-6) + \frac{8}{56}(x+1)(x-2)(x-4) \\ &= -\frac{1}{15}(x-2)(x-4)(x-6) + \frac{1}{8}(x+1)(x-4)(x-6) - \frac{1}{10}(x+1)(x-2)(x-6) \\ &+ \frac{1}{7}(x+1)(x-2)(x-4). \end{aligned}$$

Problems

1. Find the Lagrange interpolation polynomial $P(x)$ such that $P(-2) = 3$, $P(1) = 2$, $P(2) = 4$.

Solution. Write $x_1 = -2$, $x_2 = 1$, $x_3 = 2$. We have

$$l_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} = \frac{(x-1)(x-2)}{(-2-1)(-2-2)} = \frac{1}{12}(x-1)(x-2),$$

$$l_2(x) = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} = \frac{(x+2)(x-2)}{(1+2)(1-2)} = -\frac{1}{3}(x+2)(x-2),$$

$$l_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} = \frac{(x+2)(x-1)}{(2+2)(2-1)} = \frac{1}{4}(x+2)(x-1).$$

Thus, we have

$$\begin{aligned} P(x) &= P(-2)l_1(x) + P(1)l_2(x) + P(2)l_3(x) = 3 \cdot \frac{1}{12}(x-1)(x-2) + 2 \cdot \left(-\frac{1}{3}\right)(x+2)(x-2) \\ &+ 4 \cdot \frac{1}{4}(x+2)(x-1) = \frac{1}{4}(x-1)(x-2) - \frac{2}{3}(x+2)(x-2) + (x+2)(x-1). \end{aligned}$$

2. Find the Lagrange interpolation polynomial $P(x)$ such that $P(1) = -3$, $P(3) = -1$, $P(4) = 3$.

Solution. Write $x_1 = 1$, $x_2 = 3$, $x_3 = 4$. We have

$$l_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} = \frac{(x-3)(x-4)}{(1-3)(1-4)} = \frac{1}{6}(x-3)(x-4),$$

$$l_2(x) = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} = \frac{(x-1)(x-4)}{(3-1)(3-4)} = -\frac{1}{2}(x-1)(x-4),$$

$$l_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} = \frac{(x-1)(x-3)}{(4-1)(4-3)} = \frac{1}{3}(x-1)(x-3).$$

Thus, we have

$$\begin{aligned} P(x) &= P(1)l_1(x) + P(3)l_2(x) + P(4)l_3(x) = -3 \cdot \frac{1}{6}(x-3)(x-4) + (-1) \cdot \left(-\frac{1}{2}\right)(x-1)(x-4) \\ &+ 3 \cdot \frac{1}{3}(x-1)(x-3) = -\frac{1}{2}(x-3)(x-4) + \frac{1}{2}(x-1)(x-4) + (x-1)(x-3) = x^2 - 3x - 1. \end{aligned}$$

3. Find a polynomial $P(x)$ such that $P(-2) = 3$, $P(-1) = 5$, $P(2) = 7$, $P(4) = 3$, and $P(5) = -1$,

In describing the Lagrange interpolation polynomial, one usually writes $p(x) = \prod_{i=1}^{n}(x - x_i)$, since this allows one to write the polynomials $l_i(x)$ in a simple way. In fact,

$$l_i(x) = \frac{p(x)}{p'(x_i)(x - x_i)} \tag{1}$$

Indeed, using the product rule for differentiation,

$$p'(x) = \sum_{k=1}^{n} \prod_{\substack{j=1 \\ j \neq k}}^{n} (x - x_j).$$

If one substitutes $x = x_i$, all the terms of the sum except the one with $j = i$ will be contain a zero factor, so we obtain

$$p'(x_i) = \prod_{\substack{j=1 \\ j \neq i}}^{n} (x_i - x_j). \tag{2}$$

Substituting this into our original definition of $l_i(x)$, formula (1) follows. This formula, as stated, cannot be used at $x = x_i$, but the fraction on the right-hand side can clearly be reduced by dividing both the numerator and the denominator by $x - x_i$, after which the formula will be correct even for $x = x_i$.

Error of the Lagrange interpolation. The interpolation problem is usually used to approximate a function $f(x)$ by a polynomial $P(x)$ of degree $n - 1$ such that $P(x_i) = f(x_i)$ for i with $1 \leq i \leq n$. An important question is, how to estimate the error of this approximation at a point x. To find the error $f(x) - P(x)$ at x, assume x is different from any of the points x_i $(1 \leq i \leq n)$, since for $x = x_i$ the error is clearly 0. Write

$$F(t) = f(t) - P(t) - (f(x) - P(x))\frac{p(t)}{p(x)}$$

As is easily seen, $F(t) = 0$ for $t = x$ and $t = x_i$ $(1 \leq i \leq n)$.[15] Assuming f is differentiable n times, we can see that F is also differentiable n times. Applying Rolle's theorem repeatedly, we obtain that $F^{(n)}(t)$ has a zero in the open interval spanned by x and the x_i's $(1 \leq i \leq n)$. Indeed, $F'(t)$ has a zero between any two zeros of $F(z)$. Since $F(t)$ is known to have $n + 1$ (distinct) zeros in the closed interval spanned by x and the x_i's (namely x and the x_i's), $F'(t)$ must have at least n zeros in the in the open interval spanned by x and the x_i's, by Rolle's theorem. Applying Rolle's theorem again, $F''(t)$ must have at least $n-1$ zeros, $F'''(t)$ must have at least $n-2$ zeros, and so on. Repeating this argument n times, we can see that, indeed, $F^{(n)}(t)$ has at least a zero in the open interval spanned by x and the x_i's $(1 \leq i \leq n)$. Let ξ be such a zero. Then we have

$$0 = F^{(n)}(\xi) = f^{(n)}(\xi) - (f(x) - P(x))\frac{n!}{p(x)}.$$

[15]This means that the polynomial

$$Q(t) = P(t) + (f(x) - P(x))\frac{p(t)}{p(x)}$$

is the interpolation polynomial for $f(t)$ at $t = x$ and $t = x_i$ $(1 \leq i \leq n)$. That is, the polynomial $Q(t)$ interpolates $f(t)$ at the additional point $t = x$ in addition the where $P(t)$ interpolates $f(t)$.

Indeed, $P(t)$, being a polynomial of degree $n-1$, its nth derivative is zero, while $p(t)$, being a polynomial of form t^n+ lower order terms, its derivative is $n!$. Solving this for the error $f(x)-P(x)$ of the interpolation at x, we obtain

$$f(x) - P(x) = f^{(n)}(\xi)\frac{p(x)}{n!}.$$

This is the error formula for the Lagrange interpolation. Of course, the value of ξ is not known; for this reason, this only allows one to estimate the error, and not calculate this exactly.

Problems

4. Estimate the error of interpolating $\ln x$ at $x = 3$ with an interpolation polynomial with base points $x = 1$, $x = 2$, and $x = 4$, and $x = 6$.

Solution. We have $n = 4$,

$$p(x) = (x-1)(x-2)(x-4)(x-6),$$

and

$$\frac{d^4}{dx^4}\ln x = -\frac{6}{x^4}.$$

So the error equals

$$E = -\frac{6}{\xi^4}\frac{(3-1)(3-2)(3-4)(3-6)}{4!} = -\frac{3}{2\xi^4}$$

The value of ξ is not known, but it is known that $1 < \xi < 6$. Hence

$$-\frac{3}{2} < E < -\frac{3}{2\cdot 6^4} = -\frac{1}{864}.$$

5. Estimate the error of Lagrange interpolation when interpolating $f(x) = \sqrt{x}$ at $x = 5$ when using the interpolation points $x_1 = 1$, $x_2 = 3$, and $x_3 = 4$.

Solution. Noting that the third derivative of $\sqrt{x}$ equals $\frac{3}{8}x^{-5/2}$. So we have the error at $x = 5$ is

$$E(5) = \frac{(5-1)(5-3)(5-4)}{3!}\cdot\frac{3}{8}\xi^{-5/2} = \frac{\xi^{-5/2}}{2}$$

according to the error formula of the Lagrange interpolation, where ξ is some number in the interval spanned by x, x_1, x_2, and x_3, i.e., in the interval $(1, 5)$. Clearly, the right-hand side is smallest for $\xi = 5$ and largest for $x = 1$. Noting that $5^{5/2} = 25\sqrt{5}$, we have

$$\frac{1}{50\sqrt{5}} < E(5) < \frac{1}{2}.$$

We have strict inequalities, since the values $\xi = 1$ and $\xi = 5$ are not allowed.

6. Estimate the error of Lagrange interpolation when interpolating $f(x) = 1/x$ at $x = 2$ when using the interpolation points $x_1 = 1$, $x_2 = 4$, and $x_3 = 5$.

Solution. Noting that the third derivative of $1/x$ equals $-6/x^4$, with $f(x) = 1/x$ and with some ξ between 1 and 5, for the error at $x = 2$ we have

$$E(x) = f'''(\xi)\frac{(x-1)(x-4)(x-5)}{3!} = -\frac{6}{\xi^4}\,\frac{(2-1)(2-4)(2-5)}{6} = -\frac{6}{\xi^4}$$

according to the error formula of the Lagrange interpolation, where ξ is some number in the interval spanned by x, x_1, x_2, and x_3, i.e., in the interval $(1, 5)$. Clearly, the right-hand side is smallest for $\xi = 1$ and largest for $\xi = 5$. Thus we have

$$-6 < E(5) < -\frac{6}{625}.$$

We have strict inequalities, since the values $\xi = 1$ and $\xi = 5$ are not allowed.

7. Estimate the error of interpolating e^x at $x = 2$ with an interpolation polynomial with base points $x = 1$, $x = 3$, and $x = 4$, and $x = 6$.

8. Estimate the error of interpolating $\sin x$ at $x = 3$ with an interpolation polynomial with base points $x = -1$, $x = 1$, and $x = 2$, and $x = 4$.

9. Assume a differentiable function f has n zeros in the interval (a, b). Explain why f' has at least $n - 1$ zeros in the same interval.

Solution. By Rolle's Theorem, in any any open interval determined by adjacent zeros of f there is at least one zero of f'. The n zeros give rise to $n - 1$ intervals determined by adjacent zeros, thus giving rise to $n - 1$ zeros.

The result is also true if one takes multiplicities into account. One says that f has a k-fold zero at α if

$$f(\alpha) = f'(\alpha) = \ldots = f^{(k-1)}(\alpha) = 0.^{16}$$

Clearly, a k-fold zero of f gives rise a $k - 1$-fold zero of f' at the same location, so for a single multiple zero the assertion is true. Assuming that f has n zeros, some possibly multiple, one could carefully use this fact to compute the number of zeros[17] of f', but the calculation is unnecessarily messy. It is easier to note that if one scatters a k-fold zero, and replaces it by k nearby single zeros, the total count of n zeros will not change.[18] To avoid confusion, call the function so changed g. In counting the zeros of g', the count will not change either. The cluster of k single zeros of g will give rise to $k - 1$ zeros of g', just as a k-fold zero of f gave rise to $k - 1$ zeros of f'. As we already know that g' has (at least) $n - 1$ zeros, the same will also be true for f.

[16] This definition is motivated by the case of polynomials. If a polynomial $P(x)$ has the form

$$P(x) = (x - \alpha)^k Q(\alpha),$$

then it is easy to see that $P(x)$ has a k-fold zero in the sense of the above definition.

[17] More precisely, to give a lower bound for the number of zeros.

[18] One need not reflect how to do this analytically, i.e., one need not think up an expression for a function g that will have scattered zeros near the multiple zeros of f. One can simply redraw the graph of f near the multiple zero, and argue "geometrically."

6. NEWTON INTERPOLATION

Newton, in his solution to the interpolation problem, looked for the interpolation polynomial in the form

$$\begin{aligned} P(x) &= A_1 + A_2(x - x_1) + A_3(x - x_1)(x - x_2) + \ldots + A_n(x - x_1)(x - x_2)\ldots(x - x_{n-1}) \\ &= \sum_{i=1}^{n} A_i \prod_{j=1}^{i-1}(x - x_j). \end{aligned}$$

Given a function $f(x)$, we want to determine the coefficients A_i's in such a way that $P(x_i) = f(x_i)$ for each i with $1 \le i \le n$. This equation with $x = x_1$ gives $A_1 = f(x_1)$. Hence

$$f(x_2) = P(x_2) = f(x_1) + A_2(x_2 - x_1).$$

From now on, assume that the numbers x_1, x_2, ..., x_n are pairwise distinct. We then have

$$A_2 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{f(x_1) - f(x_2)}{x_1 - x_2}.$$

For arbitrary distinct real numbers t_1, t_2, ..., define the divided differences recursively as

$$f[t_1] = f(t_1), \tag{1}$$

$$f[t_1, t_2] = \frac{f[t_2] - f[t_1]}{t_2 - t_1}, \tag{2}$$

and, in general,

$$f[t_1, \cdots, t_{i-1}, t_i] = \frac{f[t_2, \cdots, t_i] - f[t_1, \cdots, t_{i-1}]}{t_i - t_1}. \tag{3}$$

If we write

$$f_{[t_2, t_3, \ldots, t_{i-1}]}(t) = f[t, t_2, \ldots t_{i-1}] \tag{4}$$

then we have

$$f[t_1, \cdots, t_{i-1}, t_i] = f_{[t_2, \cdots, t_{i-1}]}[t_1, t_i] \tag{5}$$

for $i > 2$.[19] If we write $f_{[\,]} = f$, then (5) is true even for $i = 2$. While this could be established directly, a more elegant approach is the following: we discard the original definition of divided differences, and take formulas (1), (2), (4), and (5) as the definition. We will establish (3) as a consequence later.

We can show by induction that

$$A_i = f[x_1, x_2, \cdots, x_i]. \tag{6}$$

[19] To make sense of this last formula, assume that the above formulas define the divided differences for an arbitrary function f, not just the specific function f for which we are discussing the Newton interpolation polynomial. We will take this point of view in what follows, in that we will consider divided differences also for functions other than f.

In fact, assume this result is true for all i with $i < n$; we will show it for $i = n$. In view of this assumption, the polynomial interpolating f at the points x_i for $1 \le i \le n-1$ is

$$P_{n-1}(x) = \sum_{i=1}^{n-1} f[x_1, \cdots, x_i] \prod_{j=1}^{i-1} (x - x_j).$$

Then the polynomial

$$P(x) = \sum_{i=1}^{n-1} f[x_1, \cdots x_i] \prod_{j=1}^{i-1} (x - x_j) + A_n \prod_{j=1}^{n-1} (x - x_j)$$

still interpolates $f(x)$ at the points x_i for $1 \le i \le n-1$, since the last term vanishes at these points. We need to determine A_n in such a way that the equation $P(x_n) = f(x_n)$ also holds. Clearly,

$$P(x) = A_n x^{n-1} + \text{ lower order terms},$$

For each integer k, with $Q(x) = x^{k+1}$ we have

$$Q[x, a] = x^k + \text{ lower order terms} \tag{7}$$

for $x \neq a$. In fact,

$$Q[x, a] = \frac{x^{k+1} - a^{k+1}}{x - a} = \sum_{j=0}^{k} x^{k-j} a^j.$$

Hence taking divided differences of $P(x)$ m times for $0 \le m \le n$, we have[20]

$$P[x, x_2, x_3, \ldots x_m] = A_n x^{n-m} + \text{ lower order terms}.$$

Indeed, this is true for $m = 1$ if we note that the left-hand side for $m = 1$ is $P(x)$.[21] Assume $m > 1$ and assume that this formula is true with $m - 1$ replacing m, that is, we have

$$P[x, x_2, x_3, \ldots x_{m-1}] = P_{[x_2, x_3, \ldots x_{m-1}]}(x) = A_n x^{n-m+1} + \text{ lower order terms}.$$

Taking divided differences and using (5) and (7), we obtain that

$$P[x, x_2, x_3, \ldots x_m] = P_{[x_2, x_3, \ldots x_{m-1}]}[x, x_m] = A_n x^{n-m} + \text{ lower order terms},$$

showing that the formula is also true for m. Hence the formula is true for all m with $1 \le m \le n$. Using this formula with $m = n$, we obtain that

$$P[x, x_2, x_3, \ldots x_n] = A_n.$$

This is true for every x; with $x = x_1$, this shows that

$$P[x_1, x_2, x_3, \ldots x_n] = A_n.$$

Now, clearly,

$$f[x_1, x_2, x_3, \ldots x_n] = P[x_1, x_2, x_3, \ldots x_n].$$

This is so, because the left-hand side is calculated by using the values $f(x_1)$, ..., $f(x_n)$, while the right-hand side is calculated by using the values $P(x_1)$, ..., $P(x_n)$, and these values are in turn the same. This completes the proof of (6) (for $i = n$; for $i < n$ its validity was assumed as the induction hypothesis).

To summarize, the Newton interpolation polynomial at the points x_1, x_2, ..., x_n we have

$$P(x) = \sum_{i=1}^{n} f[x_1, \cdots x_i] \prod_{j=1}^{i-1} (x - x_j).$$

[20] In the argument that follows, we need to assume that x is different from x_2, x_3, ..., x_n, but we may allow $x = x_1$.

[21] Here we take $[x_2, x_3, \ldots, x_m]$ for $m = 1$ to be the empty tuple. That is, for $m = 1$ we have $P_{[x_2, x_3, \ldots x_m]} = P_{[\,]}$ and $P[x, x_2, x_3, \ldots x_m] = P[x] = P(x)$.

Permuting the arguments of a divided difference. Divided differences are independent of the order in which the points are listed.

THEOREM. *Let x_1, x_2, ..., x_k be distinct points, and let x_{i_1}, x_{i_2}, ..., x_{i_k} be a permutation of these points (i.e., a listing of the same points in a different order). Let f be a function defined at these points. Then*

$$f[x_1, x_2, \dots, x_k] = f[x_{i_1}, x_{i_2}, \dots, x_{i_k}];$$

that is, a divided difference does not change if we rearrange the points.

PROOF. It is not hard to prove this by induction. But another, simpler argument is based on the observation that the leading term of the interpolating polynomial to f using the base points x_1, x_2, ..., x_k is

$$f[x_1, x_2, \dots, x_k]x^{k-1},$$

while the leading term of the interpolating polynomial to f using the base points x_{i_1}, x_{i_2}, ..., x_{i_k} is

$$f[x_{i_1}, x_{i_2}, \dots, x_{i_k}]x^{k-1}.$$

As these two interpolating polynomials are the same, the coefficients of their leading terms must agree. This establishes the assertion.

Note that so far, everything was proved by taking formulas (1), (2), (4), and (5) as the definition of divided differences. With the aid of the above Theorem, we can now easily establish (3). Indeed, using (1), (2), (4), and (5), and the Theorem above, we have

$$\begin{aligned} f[t_1, \cdots, t_{i-1}, t_i] &= f_{[t_2, \cdots, t_{i-1}]}[t_1, t_i] = \frac{f[t_i, t_2, \cdots, t_{i-1}] - f[t_1, t_2, \cdots, t_{i-1}]}{t_i - t_1} \\ &= \frac{f[t_2, \cdots, t_{i-1}, t_i] - f[t_1, t_2, \cdots, t_{i-1}]}{t_i - t_1}, \end{aligned}$$

showing that (3) also holds. Hence one can indeed use (1)–(3) to define divided differences.

Problems

1. Find the Newton interpolation polynomial $P(x)$ of degree at most 3 such that $P(1) = 2$, $P(2) = 4$, $P(4) = 6$, $P(5) = 9$.

Solution. We have

$$P[1,2] = \frac{P[2] - P[1]}{2 - 1} = \frac{4 - 2}{2 - 1} = 2,$$

$$P[2,4] = \frac{P[4] - P[2]}{4 - 2} = \frac{6 - 4}{4 - 2} = 1,$$

$$P[4,5] = \frac{P[5] - P[4]}{5 - 4} = \frac{9 - 6}{5 - 4} = 3,$$

$$P[1,2,4] = \frac{P[2,4] - P[1,2]}{4 - 1} = \frac{1 - 2}{4 - 1} = -\frac{1}{3},$$

$$P[2,4,5] = \frac{P[4,5] - P[2,4]}{5 - 2} = \frac{3 - 1}{5 - 2} = \frac{2}{3}$$

$$P[1,2,4,5] = \frac{P[2,4,5] - P[1,2,4]}{5 - 1} = \frac{\frac{2}{3} - \left(-\frac{1}{3}\right)}{5 - 1} = \frac{1}{4}.$$

We can summarize these values in a divided difference table:

x	$f[.]$	$f[.,.]$	$f[.,.,.]$	$f[x.,.,.,.]$
1	2			
		2		
2	4		$-\frac{1}{3}$	
		1		$\frac{1}{4}$
4	6		$\frac{2}{3}$	
		3		
5	9			

Thus,

$$\begin{aligned} P(x) &= P[1] + P[1,2](x-1) + P[1,2,4](x-1)(x-2) + P[1,2,4,5](x-1)(x-2)(x-4) \\ &= 2 + 2(x-1) - \frac{1}{3}(x-1)(x-2) + \frac{1}{4}(x-1)(x-2)(x-4). \end{aligned}$$

Observe that we can take these points in any other order. Since, as we know, $P[x_1, \dots, x_i]$ does not change if we permute its argument, we can use already calculated divided difference to write the result in different ways. For example

$$\begin{aligned} P(x) &= P[4] + P[4,2](x-4) + P[4,2,1](x-4)(x-2) + P[4,2,1,5](x-4)(x-2)(x-1) \\ &= 6 + 1(x-4) - \frac{1}{3}(x-4)(x-2) + \frac{1}{4}(x-4)(x-2)(x-1). \end{aligned}$$

2. Find the Newton interpolation polynomial $P(x)$ of degree 3 such that $P(2) = 3$, $P(4) = 5$, $P(7) = 2$, $P(8) = 3$.

3. The leading term of the Newton interpolation polynomial P to a function f with the nodes x_0, x_1, … x_n is

$$f[x_0, x_1, \cdots, x_n]x^n.$$

Using this, show that

$$f[x_0, x_1, \cdots, x_n] = \frac{f^{(n)}(\xi)}{n!}$$

for some ξ in the interval spanned by x_0, x_1, … x_n. (All the nodes x_0, x_1, … x_n are assumed to be distinct.)

Solution. Taking the nth derivative of the polynomial P, only the derivative of the leading term survives. That is,

$$P^{(n)}(x) = n! f[x_1, x_2, \cdots, x_n].$$

On the other hand, $f(x) - P(x)$ has at least $n+1$ zeros, x_0, x_1, … x_n. Hence $f^{(n)}(x) - P^{(n)}(x)$ has at least one zero in the interval spanned by x_0, x_1, … x_n. Writing ξ for such a zero, we have

$$0 = f^{(n)}(\xi) - P^{(n)}(\xi) = f^{(n)}(\xi) - n! f[x_1, x_2, \cdots, x_n].$$

Omitting the middle member of these equations and solving the remaining equality, we obtain

$$f[x_1, x_2, \cdots, x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

as we wanted to show. (*Note*: This result is stated as a lemma in the next section, Section 7 on Hermite interpolation.)

7. HERMITE INTERPOLATION

Hermite interpolation refers to interpolation where at some of the interpolation points not only the value of the function is prescribed, but also some of its derivatives. In the most general formulation of the problem, assume we are given pairwise distinct points x_1, x_2, ..., x_n, and for each i with $1 \le i \le n$, let a number $m_i > 0$ be given. Given a function f, we are looking for a polynomial P such that

$$P^{(l)}(x_i) = f^{(l)}(x_i) \qquad (1 \le i \le n, \quad \text{and} \quad 0 \le l < m_i) \tag{1}$$

Intuitively, the number m_i specifies how many values (including the value of the function itself and perhaps some of its derivatives) are prescribed at the point x_i. The lowest degree of the polynomial for which these equations are solvable is one less than the number of equations, that is, the degree of P must be at least

$$N = \sum_{i=1}^{n} m_i - 1.$$

The Newton interpolation polynomial of gives a fairly easy way of solving this problem. The idea is to replace each interpolation point x_i by m_i distinct closely situated points, and take the limiting case when these points all approach x_i.

To figure out how this works, we need the following simple

LEMMA. *Given an interval $[a, b]$ and a positive integer n, let f be a function that is continuous in $[a, b]$ and has at least n derivatives in the interval (a, b). Let x_0, x_1, ..., x_n be $n + 1$ distinct points in the interval $[a, b]$. Then there is a number $\xi \in (a, b)$ such that*

$$f[x_0, x_1, \dots, x_n] = \frac{1}{n!} f^{(n)}(\xi).$$

PROOF. Let P be the Newton interpolation polynomial interpolating f at the points x_0, x_1, ..., x_n. That is

$$P(x) = \sum_{i=0}^{n} f[x_0, \cdots x_i] \prod_{j=0}^{i-1} (x - x_j).$$

Then the function $f(x) - P(x)$ has at least $n + 1$ zeros in the interval $[a, b]$. Thus, similarly as in the argument used to evaluate the error of the Lagrange interpolation, there is a $\xi \in [a, b]$ such that

$$f^{(n)}(\xi) - P^{(n)}(\xi) = 0.$$

Here P is a polynomial of degree n; hence its nth derivative is $n!$ times its leading coefficient. That is

$$f^{(n)}(\xi) - n! f[x_0, \dots x_n] = 0.$$

This is clearly equivalent to the equation we wanted to prove.

An easy consequence of this Lemma is the following: If $f^{(n)}$ is continuous at a then

$$\lim_{\substack{x_0 \to a \\ x_1 \to a \\ \cdots\cdots \\ x_n \to a}} f[x_0, x_1, \dots, x_n] = \frac{1}{n!} f^{(n)}(a); \tag{2}$$

in the limit on the left-hand side, we need to assume that x_0, x_1, ..., x_n are pairwise distinct while approaching a, in order that the divided difference on the left-hand side be defined.

Motivated by the above Lemma, we can define the n-fold divided difference $f[x, x, \cdots, x]$ (x taken n times) as

$$f[x, x, \cdots, x] = \frac{1}{(n-1)!} f^{(n-1)}(x)$$

provided that the derivative on the right-hand side exists. Taking this into account, we can extend the definition of the divided difference to the case when several points agree. When x_1 and x_i are different, we can still use the formula

$$f[x_1, \cdots, x_{i-1}, x_i] = \frac{f[x_2, \cdots x_i] - f[x_1, \cdots, x_{i-1}]}{x_i - x_1},$$

That is, given arbitrary points $x_1, \ldots, x_n$ that are no longer assumed to be pairwise distinct, we define the divided differences by first assuming that $x_1 \le x_2 \le \ldots \le x_n$. For any $t_1, \ldots t_i$ with $t_1 \le t_2 \le \ldots \le t_i$ we put $f[t_1] = f(t_1)$ for $i = 1$, and for $i > 1$ we use a recursive defininition: we write

$$f[t_1, t_2, \cdots, t_i] = \frac{1}{(i-1)!} f^{(i-1)}(t_1) \qquad \text{if} \quad t_1 = t_i, \tag{3}$$

and

$$f[t_1, \cdots, t_{i-1}, t_i] = \frac{f[t_2, \cdots t_i] - f[t_1, \cdots, t_{i-1}]}{t_i - t_1} \qquad \text{if} \quad t_1 \ne t_i. \tag{4}$$

Finally, if $i_1, i_2, \ldots, i_k$ is a permutation of the numbers 1, 2, …, k then we put

$$f[t_{i_1}, t_{i_2}, \ldots, t_{i_k}] = f[t_1, t_2, \ldots, t_k]. \tag{5}$$

With this definition, writing

$$P_{[i_1, \ldots, i_n]}(x) = \sum_{k=1}^{n} f[x_{i_1}, \cdots, x_{i_k}] \prod_{j=1}^{k-1} (x - x_{i_j}) \tag{6}$$

for any permutation $i_1, i_2, \ldots, i_n$ of the numbers 1, 2, …, n, we have

$$P_{[i_1, \ldots, i_n]}(x) = P_{[1, \ldots, n]}(x) \tag{7}$$

for all x, i.e., the polynomial $P_{[i_1, \ldots, i_n]}$ does not depend on the permutation $i_1, i_2, \ldots, i_n$.

In order to prove this result, we may assume that f is continuously differentiable n times at each of the points $x_1, \ldots, x_n$. Indeed, if this is not already the case, then we can replace f by a function g with n continuous derivatives at these points such that $g^{(l)}(x_k) = f^{(l)}(x_k)$ for all those k and l with $1 \le k \le n$ and $1 \le l \le n$ for which the right-hand side exists. This will not change any of the divided differences. Making this assumption (about the continuity of the derivatives), we will define the divided differences in a different way, and then we will show that with new this definition formulas (3)–(7) are satisfied. Assuming that the points $t_1, t_2, \ldots, t_i$ ($i \le n$) come from among the points $x_1, x_2, \ldots, x_n$, we put

$$f[t_1, t_2, \ldots, t_i] = \lim_{\substack{y_1 \to t_1 \\ y_2 \to t_2 \\ \cdots\cdots \\ y_i \to t_i}} f[y_1, y_2, \ldots, y_i]; \tag{8}$$

in the limit on the right-hand side, we need to assume that $y_1, y_2, \ldots, y_i$ are pairwise distinct while approaching $t_1, t_2, \ldots, t_i$, respectively, in order that the divided difference on the right-hand side

be defined. It is not immediately clear that the limit on the right-hand side exists, but if it does, it *is clear* that the limit also exists and is the same if we permute the points t_1, t_2, ..., t_i, since in a divided difference with distinct arguments the points can be permuted. This establishes (5) as long as we can show that the limit on the right-hand side exists if $t_1 \le t_2 \le \ldots \le t_i$. Assuming that indeed $t_1 \le t_2 \le \ldots \le t_i$, if $t_1 = t_i$ the existence of the limit and the validity of (3) follows from (2), and if $t_1 < t_i$ then the existence of the limit and the validity (4) follows by induction on i and the definition of divided differences for pairwise distinct arguments.

This also establishes (7), since, according to (8), the polynomial interpolating at the points x_{i_1}, ..., x_{i_n} defined in (6) can be obtained as the limit of the polynomial interpolating at y_1, ..., y_n as $y_1 \to x_{i_1}$, ..., $y_n \to x_{i_n}$ for pairwise distinct y_1, ..., y_n; so we may simply write $P(x)$ instead of $P_{[i_1,\ldots,i_n]}(x)$. If t_1, ..., t_n is a permutation of the points x_1, ..., x_n such that $t_1 = \ldots = t_k$, then according to (3) we have

$$\begin{aligned} P(x) &= \sum_{i=1}^{n} f[t_1, \cdots, t_i] \prod_{j=1}^{i-1} (x - t_j) \\ &= \sum_{i=1}^{k} \frac{1}{(i-1)!} f^{(i-1)}(t_1)(x - t_1)^i + (x - t_1)^k \sum_{i=k+1}^{n} f[t_1, \cdots, t_i] \prod_{j=k+1}^{i-1} (x - t_j). \end{aligned}$$

This shows that for l with $0 \le l < k$ we have $P^{(l)}(t_1) = f^{(l)}(t_1)$; that is, the relation analogous to (1) is satisfied – note that the notation used now is different from the one used in (1) (in fact, there we assumed that the points x_1, ..., x_n were pairwise distinct, whereas here we do not make this assumption). This shows that the polynomial $P(x)$ defined here is indeed the Hermite interpolation polynomial.

One can also show that, with this extended definition of divided differences, the Lemma above remains valid even if the points x_0, ..., x_n are not all distinct but at least two of them are different, except that one needs to assume appropriate differentiability of f even at the endpoints a or b in case these points are repeated in the divided difference $f[x_0, x_1, \ldots, x_n]$, to ensure that this divided difference is defined. The proof is identical to the proof given above.

Problems

1. Find the Newton-Hermite interpolation polynomial $P(x)$ such that $P(2) = 3$, $P'(2) = 6$, $P''(2) = 4$, $P(4) = 5$ $P'(4) = 7$.

Solution. We have $P[2] = 3$ and $P[4] = 5$. Further

$$P[2,2] = \frac{P'[2]}{1!} = 6,$$

$$P[2,2,2] = \frac{1}{2}P''(2) = 2$$

$$P[2,4] = \frac{P[4] - P[2]}{4-2} = \frac{5-3}{4-2} = 1,$$

$$P[4,4] = \frac{P'[4]}{1!} = 7,$$

$$P[2,2,4] = \frac{P[2,4] - P[2,2]}{4-2} = \frac{1-6}{4-2} = -\frac{5}{2},$$

$$P[2,4,4] = \frac{P[4,4] - P[2,4]}{4-2} = \frac{7-1}{4-2} = 3,$$

$$P[2,2,2,4] = \frac{P[2,2,4] - P[2,2,2]}{4-2} = \frac{-\frac{5}{2} - 2}{4-2} = -\frac{9}{4},$$

$$P[2,2,4,4] = \frac{P[2,4,4] - P[2,2,4]}{4-2]} = \frac{3 - \left(-\frac{5}{2}\right)}{4-2} = \frac{11}{4},$$

$$P[2,2,2,4,4] = \frac{P[2,2,4,4] - P[2,2,2,4]}{4-2} = \frac{\frac{11}{4} - \left(-\frac{9}{4}\right)}{4-2} = \frac{20}{8} = \frac{5}{2}$$

Hence we can write the Newton-Hermite interpolation polynomial as

$$\begin{aligned} P(x) &= P[2] + P[2,2](x-2) + P[2,2,2](x-2)(x-2) + P[2,2,2,4](x-2)(x-2)(x-2) \\ &+ P[2,2,2,4,4](x-2)(x-2)(x-2)(x-4) = 3 + 6(x-2) + 2(x-2)^2 \\ &- \frac{9}{4}(x-2)^3 + \frac{5}{2}(x-2)^3(x-4). \end{aligned}$$

We can step through the coefficients in any other order. For example

$$\begin{aligned} P(x) &= P[2] + P[2,4](x-2) + P[2,4,2](x-2)(x-4) + P[2,4,2,4](x-2)(x-4)(x-2) \\ &+ P[2,4,2,4,2](x-2)(x-4)(x-2)(x-4) = 3 + (x-2) - \frac{5}{2}(x-2)(x-4) \\ &+ \frac{11}{4}(x-2)^2(x-4) + \frac{5}{2}(x-2)^2(x-4)^2. \end{aligned}$$

Note that, in evaluating the coefficients here, the order of the points does not make any difference; that is, $P[2,4,2,4] = P[2,2,4,4]$, and the latter was evaluated above.

2. Find the Newton-Hermite interpolation polynomial $P(x)$ such that $P(1) = 3$, $P'(1) = 6$, $P(3) = 6$, $P'(3) = 5$ $P''(3) = 8$, P(4)=5.

8. THE ERROR OF THE NEWTON INTERPOLATION POLYNOMIAL

The Newton interpolation formula describes the same polynomial as the Lagrange interpolation formula, so the error term for the Newton interpolation formula should be the same. Yet the Newton interpolation formula allows us to contribute to the discussion of the error of interpolation. Given distinct base points $x_1, \ldots, x_n$, the Newton interpolation polynomial $P(t)$ on these points can be written as

$$P(t) = \sum_{i=1}^{n} f[x_1, \cdots, x_i] \prod_{j=1}^{i-1} (t - x_j).$$

If we add the point x to the interpolation points, the polynomial $Q(t)$ interpolating f at the points $x, x_1, \ldots, x_n$ can be written as

$$Q(t) = P(t) + f[x_1, \cdots, x_n, x] \prod_{j=1}^{n} (t - x_j).$$

Clearly, $Q(x) = f(x)$, since x is one of the interpolation points for $Q(t)$. Thus, the error

$$E(x) = f(x) - P(x)$$

at x of the interpolating polynomial P can be written as

$$E(x) = Q(x) - P(x) = f[x, x_1, \cdots, x_n] \prod_{j=1}^{n} (x - x_j), \tag{1}$$

where, noting that the order of arguments in immaterial in the divided differences, we wrote x as the first argument. In view of the Lemma in Section 7 on Hermite interpolation, we have

$$E(x) = \frac{f^{(n)}(\xi)}{n!} \prod_{j=1}^{n} (x - x_j) \tag{2}$$

for some ξ in the open interval spanned by the points x, x_1, ..., x_n, assuming that f is n times differentiable in this open interval and is continuous in the closed interval spanned by these points.

Sometimes we need to differentiate the error term in (2). Since the dependence of ξ on x is not known, this cannot be done directly. However, formula (1) can provide a starting point. In fact, we have

LEMMA. *Let $n \geq 1$ be an integer, and let x_0, x_1, ..., x_n be pairwise distinct points, and that $f'(x_0)$ exists. Then*

$$\frac{d}{dx} f[x, x_1, \dots, x_n]\Big|_{x=x_0} = f[x_0, x_0, x_1, \dots, x_n]$$

REMARK. The assumption that the points x_0, x_1, ..., x_n be pairwise distinct can be omitted the price of additional differentiability requirements to make sure that the divided difference on the right-hand side is defined. As long as the right-hand side is meaningful, the statement in the Lemma is true. We will restrict the main discussion to the case of distinct points, and then describe what modifications are needed to establish the result in case the points are not necessarily distinct.

PROOF. Let t be a point such that $t \neq x_i$ for i with $0 \leq i \leq n$, and $Q_t(u)$ be the interpolation polynomial that interpolates $f(u)$ at the points t, and x_0, x_1, ... x_n. Write

$$P(x) = \lim_{t \to x_0} Q_t(x).$$

First, we need to show that this limit exists. To this end, note that for any k with $1 \leq k \leq n$ the limit

$$C_k = \lim_{t \to x_0} f[t, x_0, x_1, \dots x_k] = \lim_{t \to x_0} \frac{f[t, x_1, \dots x_k] - f[x_0, x_1, \dots x_k]}{t - x_0} = \frac{d}{dx} f[x, x_1, \dots, x_k]\Big|_{x=x_0}$$

exists. Indeed, $f[x, x_1, \dots, x_n]$ can be written as a fraction with the numerator being a polynomial of x and $f(x)$, with coefficients depending on x_1, ... x_n, and $f(x_1)$, ..., $f(x_n)$, and the denominator being a product of differences of the form $x - x_i$ and $x_i - x_j$ for $1 \leq i, j < k$ $(i \neq j)$. Since none of these differences is zero, and since $f(x)$ is differentiable at x_0, the existence of the derivative on the right-hand side follows.

Thus, in the Newton interpolation formula[22]

$$Q_t(x) = f[x_0] + f[x_0, t](x - x_0) + (x - x_0)(x - t) \sum_{k=1}^{n} f[x_0, t, x_1, \dots, x_k] \prod_{j=1}^{k-1} (x - x_j)$$

[22]We often use the fact that in the divided differences, the order of the arguments is immaterial; so, while above we used the order t, x_0, next we use the order x_0, t.

the coefficients converge when $t \to x_0$. Using the simple equation $\lim_{t\to x_0} f[x_0, t] = f'(x_0)$ and making $t \to x_0$, we obtain

$$P(x) = \lim_{t\to x_0} Q_t(x) = f(x_0) + f'(x_0)(x - x_0) + (x - x_0)^2 \sum_{k=1}^{n} C_k \prod_{j=1}^{k-1}(x - x_j)$$

with the C_k's just defined. It is clear from here that $P(x_0) = f(x_0)$ and $P'(x_0) = f'(x_0)$. Furthermore, we have $Q_t(x_i) = f(x_i)$ for any i with $1 \le i \le n$, and so $P(x_i) = f(x_i)$. Thus $P(x)$ is the Newton-Hermite polynomial interpolating f at the points x_0, x_0 (twice), x_1, x_2, ..., x_n.[23] Hence, we also have

$$P(x) = f[x_0] + f[x_0, x_0](x - x_0) + (x - x_0)^2 \sum_{k=1}^{n} f[x_0, x_0, x_1, \dots x_k] \prod_{j=1}^{k-1}(x - x_j).$$

The coefficients in these two representations of $P(x)$ are equal. In particular, the leading coefficients are equal; that is

$$C_n = f[x_0, x_0, x_1, \dots x_n].$$

Comparing this with the definition of C_k for $k = n$, the assertion of the lemma follows.

One can easily extend the above argument to establish the result when some of the points x_i are repeated. First, in the limit expressing C_k above, various derivatives of f may occur, but the existence of the limit still can easily be established. Second, if x_0 is equal to some of the x_i for $i > 0$, then some additional effort needs to be made to show that the right-hand side of the first formula describing $P(x)$ indeed represents a polynomial interpolating f at the points x_0, x_0, x_1, x_2, ..., x_n (since the point x_0 is repeated more than twice in this case, we need to show that the derivates of order higher than one of $f(x)$ and $P(x)$ agree at $x = x_0$).

As a consequence, we have the following

THEOREM (A. RALSTON).[24] *Let the points $x_1 < x_2 < \ldots < x_n$, and assume $x \in (x_1, x_n)$, and assume that $x \ne x_i$ for $1 \le i \le n$. Assume f is continuous in $[x_1, x_n]$ and that $f^{(n+1)}$ exists in $[x_1, x_n]$. Let $\xi \in (x_1, x_n)$ be such that formula* (2) *for the error of the interpolation polynomial at x is satisfied with ξ; then ξ depends on x. We have*

$$\frac{d}{dx} f^{(n)}(\xi) = \frac{f^{(n+1)}(\eta)}{n+1}$$

for some $\eta \in (x_1, x_n)$.

PROOF. We have

$$f^{(n)}(\xi) = n! f[x, x_1, x_2, \dots, x_n]$$

according to the definition of ξ, in view of (1) and (2) above. The Lemma just proved implies that

$$\frac{d}{dx} f^{(n)}(\xi) = n! \frac{d}{dx} f[x, x_1, x_2, \dots, x_n] = n! f[x, x, x_1, x_2, \dots, x_n].$$

Observing that Lemma in Section 7 on Hermite interpolation can easily be extended to the case when the point x is repeated, one can conclude that the right-hand side here equals

$$\frac{d}{dx} f^{(n)}(\xi) = \frac{f^{(n+1)}(\eta)}{n+1}$$

[23] As before, taking the point x_0 twice just means that we require both of the equations $P(x) = f(x)$ and $P'(x) = f'(x)$.

[24] See A. Ralston, [Ral].

for some $\eta \in (x_1, x_n)$, as claimed.

Indeed, when we stated the Lemma in question in Section 7, divided differences with repeated points had not yet been defined, so it was not possible to state the lemma with points repeated. In light of the later definition given for divided differences with points repeated, no essential modifications need to be made in the proof. If in that lemma we assume that $x_0 = x_1$, the function $f(x) - P(x)$ will have a double zero at $x = x_0$, and so $f'(x) - P'(x)$ will still have a single zero at x_0; $n-1$ further zeros of $f'(x) - P'(x)$ will be guaranteed by Rolle's theorem, resulting in the same zero count of n for $f'(x) - P'(x)$ as in the case when the points x_0, x_1, ..., x_n are all distinct.

REMARK. If $x = x_i$ for some i with $1 < i < n$, then $f^{(n)}(\xi)$ is not determined by (2). This is the main reason that $x = x_i$ is not allowed in the Theorem. Furthermore, if $x = x_i$ for some i, then the derivative of $E(x)$ in (2) does not depend on the derivative of $f^{(n)}(\xi)$ (since this derivative is multiplied by $\prod_{j=1}^{n}(x - x_j)$ in the derivative of $E(x)$).

9. FINITE DIFFERENCES

We will consider interpolation when the interpolation points are equally spaces. Given $h > 0$, let $x_k = x_0 + kh$ for every integer. We will introduce the operators I, E, Δ, and ∇, called, in turn, identity, forward shift, forward difference, and backward difference operators with the following meanings:

$$(If)(x) = f(x), \quad (Ef)(x) = f(x+h), \quad (\Delta f)(x) = f(x+h) - f(x), \quad (\nabla f)(x) = f(x) - f(x-h).$$

We will write E^{-1} for the inverse of the operator E:

$$(E^{-1}f)(x) = f(x-h).$$

We will drop some of these parentheses; e.g., we will write $\Delta f(x)$ instead of $(\Delta f)(x)$. These operators will multiply in a natural way. For example, $\Delta^0 f(x) = f(x)$ and, for any integer $k \geq 0$ we have

$$\Delta^{k+1} f(x) = \Delta(\Delta^k f(x)) = \Delta^k f(x+h) - \Delta^k f(x).$$

One can write $\Delta = E - I$. Using the Binomial Theorem, we have

$$\Delta^n = (E-I)^n = (E + (-1)I)^n = \sum_{i=0}^{n} \binom{n}{i} (-1)^{n-i} E^i.$$

Applying this to a function f we obtain

$$\begin{aligned} \Delta^n f(x) &= (E-I)^n f(x) = (E + (-1)I)^n f(x) \\ &= \sum_{i=0}^{n} \binom{n}{i} (-1)^{n-i} E^i f(x) = \sum_{i=0}^{n} \binom{n}{i} (-1)^{n-i} f(x+ih). \end{aligned}$$

This calculation appears to be purely formal, but it is not hard to justify the operator calculus on algebraic grounds. In fact, let A and B be two operators acting on functions on the real line, let r be a real number, and let f be a function on reals. Writing

$$[(A+B)f](x) \stackrel{def}{=} [Af](x) + [Bf](x), \quad [(rA)f](x) \stackrel{def}{=} r[Af](x), \quad [(AB)f](x) \stackrel{def}{=} [A(Bf)](x),$$

the operators will form a noncommutative ring. In justifying the Binomial Theorem above for operators, the only thing one needs to notice that the operators E and I commute (because $EI = IE = E$). The standard arguments used to establish the Binomial Theorem shows that we have

$$(A+B)^n = \sum_{i=1}^{n} \binom{n}{i} A^i B^{n-i}$$

whenever the operators A and B commute.

There are other uses of the Binomial Theorem. For example, we have $E = \Delta + I$. Hence

$$E^n = (\Delta + I)^n = \sum_{i=1}^{n} \binom{n}{i} \Delta^i$$

Applying this to a function f, we obtain

$$f(x+nh) = E^n f(x) = \sum_{i=1}^{n} \binom{n}{i} \Delta^i f(x).$$

This is called *Newton's forward formula*. Similarly, $E^{-1} = (I - \nabla)$. Thus

$$E^{-n} = (I + (-1)\nabla)^n = \sum_{i=1}^{n} \binom{n}{i} (-1)^i \nabla^i.$$

Applying this to f, we obtain

$$f(x-nh) = E^{-n} f(x) = \sum_{i=1}^{n} \binom{n}{i} (-1)^i \nabla^i f(x).$$

We have $\nabla = E^{-1}\Delta$. As ∇ and E^{-1} commute, we have

$$\nabla^i f(x) = \Delta^i E^{-i} f(x) = \Delta^i f(x - ih).$$

Hence the above formula can also be written as

$$f(x-nh) = E^{-n} f(x) = \sum_{i=1}^{n} \binom{n}{i} (-1)^i \Delta^i f(x - ih).$$

Problem

1. Express $\Delta^3 f(x)$ as a linear combination of $f(x)$, $f(x+h)$, $f(x+2h)$, and $f(x+3h)$.

Solution. We have

$$\Delta^3 f(x) = (E-I)^3 f(x) = (E^3 - 3E^2 + 3E - I) f(x) = f(x+3h) - 3f(x+2h) + 3f(x+h) - f(x).$$

10. NEWTON INTERPOLATION WITH EQUIDISTANT POINTS.

Let a point x_0 be given, and let $h > 0$. Consider the points $x_t = x_0 + th$ for a real t with $-\infty < t < \infty$. Given a function f on reals, write $f_t \stackrel{def}{=} f(x_t)$ and $\Delta^k f_t \stackrel{def}{=} \Delta^k f(x_t)$. We will be particularly interested in integer values of t, and we will consider interpolating f on the consecutive points x_i, x_{i+1}, ..., x_{i+n} for integers $n \geq 0$ and k. Denoting by P the interpolating polynomial on these points, we will write $P_t \stackrel{def}{=} P(x_t)$ and $\Delta^k P_t \stackrel{def}{=} \Delta^k P(x_t)$, similarly as for f.

In order to calculate the Newton interpolation polynomial, first we show that

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{1}{k!h^k}\Delta^k f_i. \tag{1}$$

for integers $k \geq 0$ and i.

In fact, it is easy to show this by induction on k. For $k = 0$ the result is obvious:

$$f[x_i] = f(x_i) = f_i = \frac{1}{0!h^0}\Delta^0 f_i,$$

since 0!=1 and Δ^0 is the identity operator. Now, assume that $k \geq 1$ and that (1) holds with $k-1$ replacing k. Then, using the definition of divided differences, we have

$$\begin{aligned} f[x_i, x_{i+1}, \ldots, x_{i+k}] &= \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, x_{i+1}, \ldots, x_{i+k-1}]}{x_{i+k} - x_i} \\ &= \frac{\frac{1}{(k-1)!h^{k-1}}(\Delta^{k-1} f_{i+1} - \Delta^{k-1} f_i)}{kh} = \frac{1}{k!h^k}\Delta\Delta^{k-1} f_i = \frac{1}{k!h^k}\Delta^k f_i \end{aligned}$$

Thus (1) is established.

Using this, the Newton interpolation formula on the points x_0, x_1, ..., x_n can be written as follows.

$$P_t = P(x_0 + th) = \sum_{i=0}^{n} f[x_0, x_1, \ldots x_i] \prod_{j=0}^{i-1}(x_t - x_j) = \sum_{i=0}^{n} \frac{1}{i!h^i}\left(\Delta^i f_0\right) \prod_{j=0}^{i-1}(t-j)h$$

Since h occurs in each factor of the product on the right-hand side, this produces a factor of h^i, which can be canceled against $1/h^i$ before this product. We define the binomial coefficient $\binom{t}{i}$ as

$$\binom{t}{i} \stackrel{def}{=} \prod_{j=0}^{i-1} \frac{t-j}{i-j} = \frac{1}{i!}\prod_{j=0}^{i-1}(t-j) \tag{2}$$

for real t and nonnegative integer i; note that this definition agrees with the customary definition of the binomial coefficient $\binom{t}{i}$ if t is also an integer. With this, the above formula becomes

$$P_t = \sum_{i=0}^{n} \binom{t}{i} \Delta^i f_0.$$

This is *Newton's forward formula*. This formula uses the interpolation points x_0, x_1, ..., x_n. If t is an integer with $0 \leq t \leq n$, then x_t is one of the interpolation points; that is, we have $f_t = f(x_t) = P(x_t) = P_t$. Hence

$$f_t = \sum_{i=0}^{n} \binom{t}{i} \Delta^i f_0 \qquad \text{if } t \text{ is an integer with } 0 \leq t \leq n$$

Recall that when interpolating through a sequence of points, one can take these points in any order in the Newton interpolation formula. Assume that we step through some of the points x_j in the order x_{i_0}, x_{i_1}, …, x_{i_n}. Then Newton's interpolation formula becomes

$$P_t = \sum_{k=0}^{n} f[x_{i_0}, x_{i_1}, \dots x_{i_k}] \prod_{j=0}^{k-1} (x_t - x_{i_j}) = \sum_{k=0}^{n} f[x_{i_0}, x_{i_1}, \dots x_{i_k}] h^k \prod_{j=0}^{k-1} (t - i_j).$$

If we want to be able to use formula (1) for the divided differences, we need to make sure that the points x_{i_0}, x_{i_1}, …, x_{i_k} are consecutive for any integer k with $0 \le k \le n$.[25] That is, for $k > 0$ we must have

$$i_k = \max(i_0, \dots i_{k-1}) + 1 \quad \text{or} \quad i_k = \min(i_0, \dots i_{k-1}) - 1.$$

We have

$$\{i_0, \dots i_k\} = \{j : i_k - k \le j \le i_k\} \quad \text{and} \quad \{i_0, \dots i_{k-1}\} = \{j : i_k - k \le j \le i_k - 1\}$$

in the former case, and

$$\{i_0, \dots i_k\} = \{j : i_k \le j \le i_k + k\} \quad \text{and} \quad \{i_0, \dots i_{k-1}\} = \{j : i_k + 1 \le j \le i_k + k\}$$

in the latter case.[26] Since the value of the divided differences is independent of the order the points are listed, it is easy to see from (1) and (2) that we have

$$f[x_{i_0}, x_{i_1}, \dots x_{i_k}] \prod_{j=0}^{k-1} (t - i_j) = h^k \binom{t - i_k + k}{k} \Delta^k f_{i_k - k},$$

the former case, and

$$f[x_{i_0}, x_{i_1}, \dots x_{i_k}] \prod_{j=0}^{k-1} (t - i_j) = h^k \binom{t - i_k - 1}{k} \Delta^k f_{i_k},$$

Note that these formulas work even in the case $k = 0$, since $\binom{u}{0} = 1$ for any real u.[27] Thus, we have

$$P_t = \sum_{k=0}^{n} \left\{ \begin{array}{ll} \binom{t - i_k + k}{k} \Delta^k f_{i_k - k} & \text{if } i_k = \max(i_0, \dots i_{k-1}) + 1 \text{ or } k = 0, \\ \\ \binom{t - i_k - 1}{k} \Delta^k f_{i_k} & \text{if } i_k = \min(i_0, \dots i_{k-1}) - 1. \end{array} \right\}.$$

With $i_k = k$ for each k with $0 \le k \le n$ this gives Newton's forward formula above. The choice $i_k = -k$ for each k with $0 \le k \le n$ gives *Newton's backward formula*:

$$P_t = \sum_{k=0}^{n} \binom{t + k - 1}{k} \Delta^k f_{-k}.$$

[25] Of course, this requirement is vacuous for $k = 0$.

[26] These equalities are equalities of sets, meaning that on either side the same integers are listed, even though the order they are listed is probably not the same.

[27] The empty product in (2) is 1 by convention. Thus, in the formula next, case $k = 0$ could have been subsumed in either of the alternatives given in the formula, since the binomial coefficient is 1 for $k = 0$ in both cases.

The choice $i_0 = 0$, $i_{2m-1} = m$ and $i_{2m} = -m$ $m \geq 1$, i.e., when the sequence i_0, i_1, ... start out as 0, 1, −1, 2, −2, 3 gives the formula

$$P_t = f_0 + \sum_{m=0}^{n} \left(\binom{t+m-1}{2m-1} \Delta^{2m-1} f_{-m+1} + \binom{t+m-1}{2m} \Delta^{2m} f_{-m} \right)$$

if the points $x_0, \ldots, x_{2n}$ are used to interpolate, or

$$P_t = f_0 + \sum_{m=0}^{n-1} \left(\binom{t+m-1}{2m-1} \Delta^{2m-1} f_{-m+1} + \binom{t+m-1}{2m} \Delta^{2m} f_{-m} \right) + \binom{t+n-1}{2n-1} \Delta^{2n-1} f_{-n+1}$$

if the points $x_0, \ldots, x_{2n-1}$ are used to interpolate ($n > 0$). This is called *Gauss's forward formula.* It starts out as

$$P_t = f_0 + \binom{t}{1} \Delta f_0 + \binom{t}{2} \Delta^2 f_{-1} + \binom{t+1}{3} \Delta^3 f_{-1} + \binom{t+1}{4} \Delta^4 f_{-2} + \ldots .$$

The choice $i_0 = 0$, $i_{2m-1} = -m$ and $i_{2m} = m$ $m \geq 1$, i.e., when the sequence i_0, i_1, ... start out as 0, −1, 1, −2, 2, −3,... gives the formula

$$P_t = f_0 + \sum_{m=0}^{n} \left(\binom{t+m-1}{2m-1} \Delta^{2m-1} f_{-m} + \binom{t+m}{2m} \Delta^{2m} f_{-m} \right)$$

if the points $x_0, \ldots, x_{2n}$ are used to interpolate, or

$$P_t = f_0 + \sum_{m=0}^{n-1} \left(\binom{t+m-1}{2m-1} \Delta^{2m-1} f_{-m} + \binom{t+m-1}{2m} \Delta^{2m} f_{-m} \right) \binom{t+n-1}{2n-1} \Delta^{2n-1} f_{-n}$$

if the points $x_0, \ldots, x_{2n-1}$ are used to interpolate ($n > 0$). This is called *Gauss's backward formula.* It starts out as

$$P_t = f_0 + \binom{t}{1} \Delta f_{-1} + \binom{t+1}{2} \Delta^2 f_{-1} + \binom{t+1}{3} \Delta^3 f_{-2} + \binom{t+2}{4} \Delta^4 f_{-2} + \ldots .$$

11. THE LAGRANGE FORM OF HERMITE INTERPOLATION

We consider the following special case of Hermite interpolation. Let n and r be a positive integer with $r \leq n$, and let x_1, x_2, ..., x_n be distinct points. We want to find a polynomial $P(x)$ such that $P(x) = f(x)$ for i with $1 \leq x \leq n$, and $P'(x) = f'(x)$ for i with $1 \leq x \leq r$. There are $n + r$ conditions here, and the lowest degree polynomial satisfying these condition has degree $n + r - 1$. For this polynomial we have

$$P(x) = \sum_{j=1}^{n} f(x_j) h_j(x) + \sum_{j=1}^{r} f'(x_j) \bar{h}_j(x), \tag{1}$$

where, writing

$$l_{jn}(x) = \prod_{\substack{i=1 \\ i \neq j}}^{n} \frac{x - x_i}{x_j - x_i} \qquad (1 \leq j \leq n),$$

$$l_{jr}(x) = \prod_{\substack{i=1 \\ i\neq j}}^{r} \frac{x - x_i}{x_j - x_i} \qquad (1 \le j \le r),$$

and

$$p_n(x) = \prod_{i=1}^{n}(x - x_i) \quad \text{and} \quad p_r(x) = \prod_{i=1}^{r}(x - x_i),$$

we have

$$h_j(x) = \begin{cases} \big(1 - (x - x_j)\big(l'_{jr}(x_j) + l'_{jn}(x_j)\big)\big) l_{jr}(x) l_{jn}(x) & \text{if } 1 \le j \le r, \\ l_{jn}(x) \frac{p_r(x)}{p_r(x_j)} & \text{if } r < j \le n. \end{cases} \tag{2}$$

and

$$\bar{h}_j(x) = (x - x_j) l_{jr}(x) l_{jn}(x) \qquad \text{if} \quad 1 \le j \le r. \tag{3}$$

To see this, first note that we have

$$h_j(x_i) = 0 \quad \text{if} \quad i \neq j \qquad (1 \le i, j \le n). \tag{4}$$

This is because $l_{jn}(x_i) = 0$ in this case, and both expressions on the of (2) are multiplied by $l_{jn}(x)$. Further, note that

$$h_j(x_j) = 1 \qquad (1 \le j \le n). \tag{5}$$

This is immediate by (2) in case $r < j \le n$, since $l_{jn}(x_j) = 1$. In case $1 \le j \le r$, by (2) we have

$$h_j(x_j) = l_{jr}(x_j) l_{jn}(x_j) = 1 \cdot 1 = 1.$$

Next, observe that

$$h'_j(x_i) = 0 \qquad (1 \le i \le r \text{ and } 1 \le j \le n). \tag{6}$$

Indeed, if $r < j \le n$, then $l_{jn}(x) = 0$ and $p_r(x) = 0$ for $x = x_i$; that is, each of these polynomials is divisible by $(x - x_i)$. Hence $h_j(x)$ is divisible by $(x - x_i)^2$ according to (2); therefore $h'_j(x_i) = 0$ as we wanted to show. A similar argument works in case $1 \le j \le r$ and $i \neq j$. In this case $l_{jr}(x)$ and $l_{jn}(x)$ are each divisible by $(x - x_i)$. Hence $h_j(x)$ is again divisible by $(x - x_i)^2$ according to (2); therefore $h'_j(x_i) = 0$ again.

Finally, consider the case $i = j$ (when $1 \le j \le r$). We have

$$\begin{aligned} h'_j(x) = {} & \big(-l'_{jr}(x_j) - l'_{jn}(x_j)\big) l_{jr}(x) l_{jn}(x) \\ & + (1 - (x - x_j)(l'_{jr}(x_j) + l'_{jn}(x_j)) l'_{jr}(x) l_{jn}(x)) \\ & + (1 - (x - x_j)(l'_{jr}(x_j) + l'_{jn}(x_j)) l_{jr}(x) l'_{jn}(x)). \end{aligned}$$

Substituting $x = x_j$ and noting that $l_{jr}(x_j) = 1$ and $l_{jn}(x_j) = 1$, it follows that

$$h'_j(x_i) = h'_j(x_j) = \big(-l'_{jr}(x_j) - l'_{jn}(x_j)\big) + l'_{jr}(x_j) + l'_{jn}(x_j) = 0;$$

thus (6) follows.

Further, note that[28]

$$\bar{h}_j'(x_i) = \delta_{ij} \qquad (1 \le i, j \le r) \tag{7}$$

Indeed, according to (3) we have

$$\bar{h}_j(x) = l_{jr}(x)l_{jn}(x) + (x - x_j)l_{jr}'(x)l_{jn}(x) + (x - x_j)l_{jr}(x)l_{jn}'(x)$$

For $x = x_j$ this gives

$$\bar{h}_j'(x_j) = l_{jr}(x_j)l_{jn}(x_j) = 1 \cdot 1 = 1.$$

On the other hand, for $x = x_i$ with $i \neq j$ the above equality implies $\bar{h}_j'(x_j) = 0$, because $l_{jr}(x_i) = l_{jn}(x_i) = 0$ in this case. Thus (7) follows.

Using (1) and (4)–(7), we can easily conclude that $P(x_i) = f(x_i)$ for i with $1 \le i \le n$ and $P'(x_i) = f'(x_i)$ for i with $1 \le i \le r$. Thus P is indeed the interpolation polynomial we were looking for.

Problems

1. Find the Hermite interpolating polynomial $P(x)$ in the Lagrange form such that $P(1) = 3$, $P'(1) = 5$, $P(3) = 4$, $P'(3) = 2$, and $P(4) = 7$.

Solution. Writing $x_1 = 1$, $x_2 = 3$, and $x_3 = 4$, we have $n = 3$, $r = 2$, and

$$p_3(x) = (x - 1)(x - 3)(x - 4) \quad \text{and} \quad p_2(x) = (x - 1)(x - 3).$$

First, we are going to calculate $h_1(x)$. We have

$$l_{13}(x) = \frac{(x - 3)(x - 4)}{(1 - 3)(1 - 4)} = \frac{(x - 3)(x - 4)}{6},$$

$$l_{13}'(1) = \left.\frac{(x - 4) + (x - 3)}{6}\right|_{x=1} = \left.\frac{2x - 7}{6}\right|_{x=1} = -\frac{5}{6},$$

and

$$l_{12}(x) = \frac{x - 3}{1 - 3} = -\frac{x - 3}{2} \quad \text{and} \quad l_{12}'(1) = -\frac{1}{2}.$$

Thus

$$\begin{aligned} h_1(x) &= (1 - (x - 1)(l_{13}'(1) + l_{12}'(1))l_{13}(x)l_{12}(x) \\ &= \left(1 - (x - 1)\left(-\frac{5}{6} - \frac{1}{2}\right)\right)\left(-\frac{x - 3}{2}\right)\frac{(x - 3)(x - 4)}{6} = -\frac{(4x - 1)(x - 3)^2(x - 4)}{36}. \end{aligned}$$

Next, we are going to calculate $h_2(x)$. We have

$$l_{23}(x) = \frac{(x - 1)(x - 4)}{(3 - 1)(3 - 4)} = -\frac{(x - 1)(x - 4)}{2},$$

$$l_{23}'(3) = -\left.\frac{(x - 1) + (x - 4)}{2}\right|_{x=3} = -\left.\frac{2x - 5}{2}\right|_{x=3} = -\frac{1}{2},$$

[28] δ_{ij}, called Kronecker's delta, is defined to be 1 is $i = j$ and 0 if $i \neq j$.

and

$$l_{22}(3) = \frac{x-1}{3-1} = \frac{x-1}{2} \quad \text{and} \quad l'_{22}(3) = \frac{1}{2}.$$

Thus

$$\begin{aligned} h_2(x) &= (1-(x-3)(l'_{23}(3)+l'_{22}(3))l_{23}(x)l_{22}(x) \\ &= \left(1-(x-1)\left(-\frac{1}{2}+\frac{1}{2}\right)\right)\frac{x-1}{2}\left(-\frac{(x-1)(x-4)}{2}\right) = -\frac{(x-1)^2(x-4)}{4}. \end{aligned}$$

The formula for, hence the calculation of, $h_3(x)$, is simpler:

$$h_3(x) = l_{33}(x)\frac{p_2(x)}{p_2(4)} = \frac{(x-1)(x-3)}{(4-1)(4-3)} \cdot \frac{(x-1)(x-3)}{(4-1)(4-3)} = \frac{(x-1)^2(x-3)^2}{9}.$$

The formula for $\bar{h}_1(x)$ is also simple:

$$\bar{h}_1(x) = (x-1)l_{12}(x)l_{13}(x) = (x-1)\cdot\frac{x-3}{1-3}\cdot\frac{(x-3)(x-4)}{(1-3)(1-4)} = -\frac{(x-1)(x-3)^3(x-4)}{12}.$$

Similarly for $\bar{h}_2(x)$:

$$\bar{h}_2(x) = (x-3)l_{22}(x)l_{23}(x) = (x-3)\cdot\frac{x-1}{3-1}\cdot\frac{(x-1)(x-4)}{(3-1)(3-4)} = -\frac{(x-1)^2(x-3)(x-4)}{4}.$$

So

$$\begin{aligned} P(x) &= 3h_1(x)+4h_2(x)+7h_3(x)+5\bar{h}_1(x)+2\bar{h}_2(x) = -3\frac{(4x-1)(x-3)^2(x-4)}{36} \\ &\quad -4\frac{(x-1)^2(x-4)}{4}+7\frac{(x-1)^2(x-3)^2}{9} \\ &\quad -5\frac{(x-1)(x-3)^3(x-4)}{12}-2\frac{(x-1)^2(x-3)(x-4)}{4} = \\ &= -\frac{(4x-1)(x-3)^2(x-4)}{12}-(x-1)^2(x-4)+7\frac{(x-1)^2(x-3)^2}{9} \\ &\quad -5\frac{(x-1)(x-3)^3(x-4)}{12}-\frac{(x-1)^2(x-3)(x-4)}{2}. \end{aligned}$$

2. Find the Hermite interpolating polynomial $P(x)$ in the Lagrange form such that $P(2) = 5$, $P'(2) = 2$, $P(4) = 3$, $P(5) = 4$, and $P'(5) = 6$.

Hint. In order to use (1), we may take $n = 3$, $r = 2$, and $x_1 = 2$, $x_2 = 5$, $x_3 = 4$. (Listing the points in the natural order as $x_1 = 2$, $x_2 = 4$, $x_3 = 5$ would not work, since the derivatives of P are specified at the first r points of the given n points. Of course it is possible to modify formula (1) in an appropriate way and then take the points in the natural order.)

12. SOLUTION OF NONLINEAR EQUATIONS

Bisection. Assume f is a continuous function, and $f(x_1)f(x_2) < 0$.[29] Then, by the Intermediate Value Theorem, the equation $f(x) = 0$ must have at least one solution in the interval (x_1, x_2). Such a solution may be approximated by successive halving of the interval. Namely, if $x_3 = (x_1+x_2)/2$, and $f(x_1)f(x_3) < 0$ then one of these solutions must be in the interval (x_1, x_3), and if $f(x_1)f(x_3) > 0$, then there must be a solution in the interval (x_3, x_2); of course if $f(x_1)f(x_3) = 0$ then x_3 itself is a solution. By repeating this halving of the interval, a root can be localized in successively smaller intervals. There is no guarantee that all roots are found this way, since both of the intervals (x_1, x_3) and (x_3, x_2) may contains roots of the equation $f(x) = 0$.

Newton's method. If f is differentiable and x_1 is an approximate solution of the equation $f(x) = 0$ then a better approximation might be obtained by drawing the tangent line to f at x_1 and taking the new approximation x_2 to be the point where the tangent line intersects the x axis. The equation of the tangent line to f at x_1 is

$$y - f(x_1) = f'(x)(x - x_1).$$

Solving this for $y = 0$, and writing $x = x_2$ for the solution, we obtain

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}.$$

This process can be repeated to get successively better approximations to the root.

The secant method. While Newton's method is usually considerably faster than bisection, its disadvantage is that the derivative of f needs to be calculated. The secant method usually is somewhat slower than Newton's method, but there is no need to calculate the derivative. Given two approximations x_1 and x_2 to the root, the secant line, i.e., the line through the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ is drawn, and the place x_3 where this line intersects the x axis is taken to be the next approximation. The equation of the secant line is

$$y - f(x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_2).$$

Solving this for $y = 0$ and writing $x = x_3$ for the solution, we obtain

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}.$$

This equation can also be written as

$$x_3 = \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)};$$

however, the former equation is preferred in numerical calculations to the latter, because of the loss of precision.[30] If x_{n-1} and x_n have been determined for $n \geq 2$, then analogously, one uses the formula

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

---

[29] This is just a simple way of saying that $f(x_1)$ and $f(x_2)$ have different signs.

[30] Loss of precision occurs when two nearly equal numbers are subtracted. Both expressions for x_3 may be affected by losses of precision, because each expression contains a number of subtractions where the result is likely be small. However, in the former expression, only the correction term

$$-f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

is affected by losses of precision. One would expect that this correction term in small in comparison to x_2, so the loss of precision is less important than it would be if the latter formula were used.

In the method of *regula falsi*,[31] one similarly uses the secant line, but one tries to make sure that successive iterations are on different sides of the root, so as to bracket the root. One starts with two approximations x_1 and x_2 to the root of $f(x) = 0$ such that $f(x_1)f(x_2) < 0$ (which is just a short way of saying that $f(x_1)$ and $f(x_2)$ have opposite signs). Assuming that f is continuous, then x_1 and x_2 enclose a solution of the equation $f(x) = 0$. Then one forms the sequence x_1, x_2, x_3, ..., as follows. Assuming that x_1, ... x_n have already been calculated for $n \geq 2$, and take i to be the largest integer with $1 \leq i < n$ such that $f(x_n)f(x_i) < 0$, and then calculated x_{n+1} by using the formula

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_i}{f(x_n) - f(x_i)} \tag{1}$$

(this is just the formula above for x_3 with i, n, $n+1$ replacing 1, 2, 3, respectively). The trouble with this method is that if f is convex,[32] say, and $x_1 < x_2$, and further, $f(x_1) < 0 < f(x_2)$, then all the successive iterates x_2, x_3, ... lie to the left of the root. That is, when x_{n+1} is calculated by the above formula, we will have $i = 1$, and so the above formula will use x_1 instead of a later, presumably better approximation of the root. This will make the method converge relatively slowly.

The method can be speeded up by using formula (1) for the first step (even if $f(x_1)f(x_2) > 0$) with 1, 2, 3 replacing i, n, and $n+1$. In successive steps one uses formula (1) calculate x_{n+1} with $i = n-1$ unless we have $f(x_{n-2})f(x_n) < 0$ and $f(x_{n-1})f(x_n) > 0$. On the other hand, if $f(x_{n-2})f(x_n) < 0$ and $f(x_{n-1})f(x_n) > 0$, then one uses a modified step as follows: x_{n+1} is taken to be the intersection of the straight line through the points $(x_{n-2}, \alpha f(x_{n-2}))$ and $(x_n, f(x_n))$, where different choices of the parameter α give rise to different methods. In the simplest choice, one takes $\alpha = 1/2$; the method so obtained is called the Illinois method. To get the equation for the "Illinois step," write $\bar{y}_{n-2} = \alpha f(x_{n-2})$. The equation of the line through the points $(x_{n-2}, \bar{y}_{n-2})$ and $(x_n, f(x_n))$ can be written as

$$y - f(x_n) = \frac{\bar{y}_{n-2} - f(x_n)}{x_{n-2} - x_n}(x - x_n).$$

Solving this for $y = 0$ and writing $x = x_{n+1}$ for the solution, we obtain

$$x_{n+1} = x_n - f(x_n)\frac{x_{n-2} - x_n}{\bar{y}_{n-2} - f(x_n)}.$$

As mentioned, here $\bar{y}_{n-2} = f(x_{n-2})/2.$ for the Illinois method.

Problems

1. An approximate solution of the equation $f(x) = 0$ with $f(x) = x^3 - 12x + 8$ is $x = 3$. Perform one step of Newton's method to find the next approximation.

---

[31] meaning false rule

[32] f is *convex* on an interval $[a, b]$ if for any $x, t \in [a, b]$ and for any λ with $0 < \lambda < 1$ we have

$$f(\lambda x + (1 - \lambda)t \leq \lambda f(x) + (1 - \lambda)f(t).$$

Geometrically, this means that the chord between x and t lies above the graph of the function. Such a function in elementary calculus courses is called *concave up*, but the term "concave up" is not used in the mathematical literature. If instead we have the opposite inequality

$$f(\lambda x + (1 - \lambda)t \geq \lambda f(x) + (1 - \lambda)f(t),$$

i.e., when the chord is below the graph, the function is called *concave* (in elementary calculus courses, one used the term "concave down.")

Solution. We have $f'(x) = 3x^2 - 12$, $x_1 = 3$, and

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = x_1 - \frac{x_1^3 - 12x_1 + 8}{3x_1^2 - 12} = 3 + \frac{1}{15} \approx 3.06667.$$

2. Consider the equation $f(x) = 0$ with $f(x) = 2 - x + \ln x$. Using Newton's method with $x_0 = 3$ as a starting point, find the next approximation to the solution of the equation.

Solution. We have

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{2 - x_0 + \ln x_0}{\frac{1}{x_0} - 1} = 3 - \frac{-1 + \ln 3}{-2/3} = \frac{3 + 3\ln 3}{2} \approx 3.14792.$$

The actual solution is approximately 3.14619.

3. Explain the Illinois modification of the secant method.

Solution. When solving the equation $f(x) = 0$, one starts with two approximations x_1 and x_2 for the solution of the equation, and further approximations x_3, x_4,... are generated. In the secant step, x_{n+1} is the intersection of the line connecting the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$ with the x axis. In the Illinois step, x_{n+1} is the intersection $(x_{n-2}, f(x_{n-2})/2)$ and $(x_n, f(x_n))$. The secant step is performed in case of $n = 3$ (in this case the Illinois step cannot be performed, since the Illinois step requires three earlier approximations), and for $n > 3$ it is performed when $f(x_{n-1})$ and $f(x_n)$ have different signs, or if $f(x_{n-2})$, $f(x_{n-1})$, and $f(x_n)$ all have the same sign. The Illinois step is performed otherwise, i.e., when $n > 3$ and $f(x_{n-1})$ and $f(x_n)$ have the same sign and $f(x_{n-2})$ has a different sign.

13. PROGRAMS FOR SOLVING NONLINEAR EQUATIONS

In the following C programs, the methods discussed above are used to solve the equation $f(x) = 0$, where

$$f(x) = \frac{1}{x} - 2^x.$$

We discuss these programs in a Linux environment (their discussion in any Unix environment would be virtually identical). The following file `funct.c` will contain the definition of the function f:[33]

```
1 #include <math.h>
2
3 double funct(double x)
4 {
5   double value;
6   value = 1.0/x-pow(2.0,x);
7   return(value);
8 }
```

Here `pow(x,y)` is C's way of writing x^y. Using decimal points (as in writing `1.0` instead of `1`), we indicate that the number is a floating point constant. The numbers at the beginning of these lines are not part of the file; they are line numbers that are occasionally useful in a line-by-line discussion

[33]The Perl programming language was used to mark up the computer files in this section for $\mathcal{AMS}$-TEX typesetting

of such a file. The program for bisection is in the file `bisect.c`, to be discussed later. The simplest way of compiling these programs is to write a *makefile*. In the present case, the makefile is called (surprise) `makefile` (this is the default name for a makefile), with the following content:

```
1 all: bisect
2 bisect : funct.o bisect.o
3        gcc -o bisect -s -O4 funct.o bisect.o -lm
4 funct.o : funct.c
5        gcc -c -O4 funct.c
6 bisect.o : bisect.c
7        gcc -c -O4 bisect.c
```

Line 1 here describes the file to make; namely, the file `bisect`. This file contains the compiled program, and the program can be run by typing its name on the command line, that is, by typing

```
$ bisect
```

Here the dollar sign `$` is the customary way of denoting the command line prompt of the computer, even though in actual life the command line prompt is usually different.[34] The second line contains the dependencies; the file `bisect` to be made depends on the files `funct.o` and `bisect.o` (the `.o` suffix indicates that these are object files, that is, already compiled programs – read on). The file on the left of the colon is called the *target* target file, and the files on the right are called the source files. When running the `make` command, by typing, say,

```
$ make all
```

on the command line, the target file is created only if it does not already exist, or if it predates at least one of the source files (i.e., if at least one of the source files has been change since the target file has last been created). Clearly, if the source files have not changed since the last creation of the target file, there is no need to create the target file again. Line 3 contains the rule used to create the target. One important point, a quirk of the `make` command, that the first character of line three is a tab character (which on the screen looks like eight spaces); the rule always must start with a tab character. The command on this line invokes the `gcc` compiler (the GNU C compiler) to link the already created programs `funct.o` and `bisect.o`, and the mathematics library (described as `-lm` at the end of the line). The `-o` option gives the name `bisect` to the file produced by the compiler. The option `-s` gives is passed to the loader[35] to strip all symbols from the compiled program, thereby making the compiled program more compact. The option `-O4` (the first character is "Oh" and not "zero") specifies the optimization level. Lines 4 and 6 contain the dependencies for creating the object files `funct.o` and `bisect.o`, and lines 5 and 7 describe the commands issued to the GNU C compiler to create these files from the source files `funct.c` and `bisect.c`. The compiler option `-c` means compile but do not link the assembled source files. These latter two files need to be written by the programmer; in fact, the file `funct.c` has already been described, and we will discuss the file `bisect.c` next:

```
1 #include <stdio.h>
2 #include <math.h>
3
4 double funct(double);
5 double bisect(double (*fnct)(double),
6    double x0, double x1, double xtol,
7    int maxits, double *fatroot,
8    int *noofits, int *converged);
```

[34] Unix is highly customizable, and you can set the prompt to be almost anything at will.

[35] The loader links together the object programs `funct.o` and `bisect.o`, and the mathematics library invoked by the `-lm` option. Apparently only linking and no compilation is done by line 3 of the `makefile`. The option `-O4` is probably useless on this line, since it is a compiler and not a loader option. The option `-s` of `gcc` is undocumented as of version 2.96, but our description of it is probably correct.

```
 9
10 main()
11 {
12   /* This program implements the bisection method
13      for solving an equation funct(x)=0. The function
14      funct() is defined separately.                  */
15   const double tol=5e-10;
16   double x, fx, root;
17   int its, success;
18   root=bisect(&funct, 0.01, 1.0, tol,
19         50, &fx,
20         &its, &success);
21   if ( success == 2 ) {
22     printf("The function has the same signs at "
23            "both endpoints.\n");
24   }
25   else if ( success ) {
26     printf("The root is %.12f. The value of "
27            " the function\nat the root is %.12f.\n", root, fx);
28     printf("%u iterations were used to find "
29            " the root\n", its);
30   }
31   else {
32     printf("The method cannot find the root.\n");
33   }
34 }
35
36 double bisect(double (*fnct)(double),
37    double x0, double x1, double xtol,
38    int maxits, double *fatroot,
39    int *itcount, int *converged)
40 {
41   double x, fx, f0, root;
42   int iterating=1, withintol=0;
43   *converged = 0;
44   f0 = fnct(x0);
45   fx = fnct(x1);
46   if ( (f0>=0 && fx>=0) || (f0<=0 && fx<=0) ) {
47   *converged = 2;
48   return 0;
49   }
50   for (*itcount = 0; *itcount < maxits; (*itcount)++) {
51     x = (x0+x1)/2; fx = fnct(x);
52     if ( (f0<=0 && fx>=0) || (f0>=0 && fx<=0) )
53        x1 = x;
54     else
55       { x0 = x; f0 = fx; }
56   /* The next two lines are included so as to monitor
57      the progress of the calculation. These lines
58      should be deleted from a program of
59      practical utility. */
```

```
60      printf("Iteration number %2u: ", *itcount);
61      printf("x=% .12f and f(x)=% .12f\n", x, fx);
62      if ( x1-x0 <= xtol ) {
63        *converged = 1;
64        break;
65      }
66    }
67    root = (x0+x1)/2;
68    *fatroot = fnct(x);
69    return root;
70  }
```

Lines 10 through 34 here is the calling program, and the bisection method itself is described in the function `bisect` through lines 36–65. The first parameter of `bisect` in line 36 is a pointer `double (*fnct)(double)` to a function, which is called by the address of the current function parameter, `&funct` in line 18.[36] The program is called with the initial values $x_1 = 0.01$ and $x_1 = 1$. The program itself is a fairly straightforward implementation of the bisection method. Note the additional parameter `success` to indicate the whether the program was successful in finding the root. Further, the number of iterations is limited to 50 is the calling statement in lines 18–20 to avoid an infinite loop. Note the output statements in lines 60–61; these were included in the program only for the purposes of illustration. Normally, these lines should be deleted (or, better yet, commented out, since they may need to be restored for debugging). The printout of the program is as follows:

```
 1 Iteration number  0: x= 0.505000000000 and f(x)= 0.561074663602
 2 Iteration number  1: x= 0.752500000000 and f(x)=-0.355806027449
 3 Iteration number  2: x= 0.628750000000 and f(x)= 0.044232545092
 4 Iteration number  3: x= 0.690625000000 and f(x)=-0.166018770776
 5 Iteration number  4: x= 0.659687500000 and f(x)=-0.063871145198
 6 Iteration number  5: x= 0.644218750000 and f(x)=-0.010624951276
 7 Iteration number  6: x= 0.636484375000 and f(x)= 0.016594102551
 8 Iteration number  7: x= 0.640351562500 and f(x)= 0.002933217996
 9 Iteration number  8: x= 0.642285156250 and f(x)=-0.003858575571
10 Iteration number  9: x= 0.641318359375 and f(x)=-0.000465872209
11 Iteration number 10: x= 0.640834960938 and f(x)= 0.001232872504
12 Iteration number 11: x= 0.641076660156 and f(x)= 0.000383300305
13 Iteration number 12: x= 0.641197509766 and f(x)=-0.000041335881
14 Iteration number 13: x= 0.641137084961 and f(x)= 0.000170969726
15 Iteration number 14: x= 0.641167297363 and f(x)= 0.000064813802
16 Iteration number 15: x= 0.641182403564 and f(x)= 0.000011738180
17 Iteration number 16: x= 0.641189956665 and f(x)=-0.000014799045
18 Iteration number 17: x= 0.641186180115 and f(x)=-0.000001530481
19 Iteration number 18: x= 0.641184291840 and f(x)= 0.000005103837
20 Iteration number 19: x= 0.641185235977 and f(x)= 0.000001786675
21 Iteration number 20: x= 0.641185708046 and f(x)= 0.000000128096
22 Iteration number 21: x= 0.641185944080 and f(x)=-0.000000701193
23 Iteration number 22: x= 0.641185826063 and f(x)=-0.000000286548
24 Iteration number 23: x= 0.641185767055 and f(x)=-0.000000079226
25 Iteration number 24: x= 0.641185737550 and f(x)= 0.000000024435
26 Iteration number 25: x= 0.641185752302 and f(x)=-0.000000027396
```

[36] The address operator `&` with functions is superfluous, since the name `funct` already refers to the address of this function; so we could have written simple `funct` in line 18. We wrote `&funct` for the sake of clarity.

```
Iteration number 26: x= 0.641185744926 and f(x)=-0.000000001480
Iteration number 27: x= 0.641185741238 and f(x)= 0.000000011477
Iteration number 28: x= 0.641185743082 and f(x)= 0.000000004998
Iteration number 29: x= 0.641185744004 and f(x)= 0.000000001759
Iteration number 30: x= 0.641185744465 and f(x)= 0.000000000139
The root 0.641185744696. The value of  the function
at the root is 0.000000000139.
30 iterations were used to find  the root
```

For Newton's method, in addition to the function, its derivative also needs to be calculated. The program to do this constitutes the file `dfunct.c`:

```
#include <math.h>

double dfunct(double x)
{
  double value;
  value = -1.0/(x*x)-pow(2.0,x)*log(2.0);
  return(value);
}
```

The program itself is contained in the file `newton.c`. The makefile to compile this program is as follows:

```
all: newton
newton : funct.o newton.o dfunct.o
        gcc -o newton -s -O4 funct.o dfunct.o newton.o -lm
funct.o : funct.c
        gcc -c -O4 funct.c
dfunct.o : dfunct.c
        gcc -c -O4 dfunct.c
        gcc -c -O4 bisect.c
newton.o : newton.c
        gcc -c -O4 newton.c
```

The file `newton.c` itself is as follows:

```
#include <stdio.h>
#include <math.h>

#define absval(x) ((x) >= 0.0 ? (x) : (-(x)))

double funct(double);
double dfunct(double);

double newton(double (*fnct)(double), double (*deriv)(double),
   double startingval, double xtol, int maxits, double *fx,
   int *itcount, int *outcome);

main()
{
  /* This program implements the Newton's method
     for solving an equation funct(x)=0. The function
     funct() and its derivative dfunct() is defined
     separately.                                    */
```

```
19   const double tol=5e-10;
20   double x, fx, root;
21   int its, success;
22   root = newton(&funct, &dfunct,
23     3.0, tol, 50, &fx,
24     &its, &success);
25   if ( success == 2 ) {
26     printf("The root is %.12f. The value of "
27            " the function\nat the root is %.12f.\n", root, fx);
28     printf("%u iterations were used to find "
29            " the root\n", its);
30   }
31   else if (success == 1) {
32     printf("The derivative is too flat at %.12f\n", x);
33   }
34   else if (success == 0) {
35     printf("The maximum number of iterations has been reached\n");
36   }
37 }
38
39 double newton(double (*fnct)(double), double (*deriv)(double),
40    double startingval, double xtol, int maxits, double *fx,
41    int *itcount, int *outcome)
42 {
43   double x, dx, dfx, assumedzero=1e-20;
44   *outcome = 0;
45   x = startingval;
46   for (*itcount = 0; *itcount < maxits; (*itcount)++) {
47     dfx = deriv(x);
48     if ( absval(deriv(x))<=assumedzero ) {
49       *outcome = 1; /* too flat */
50       break;
51     }
52     *fx = fnct(x);
53   /* The next two lines are included so as to monitor
54      the progress of the calculation. These lines
55      should be deleted from a program of
56      practical utility. */
57     printf("Iteration number %2u: ", *itcount);
58     printf("x=% .12f and f(x)=% .12f\n", x, *fx);
59     dx = -*fx/dfx; x = x+dx;
60     if ( absval(dx)/(absval(x)+assumedzero) <= xtol ) {
61       *outcome = 2; /* within tolerance */
62       *fx = fnct(x);
63       break;
64     }
65   }
66   return x; /* returning the value of the root */
67 }
```

The calling statement calls in lines 22-24 calls the bisection function, located in lines 39–67, with initial approximation $x_1 = 3$. The variable `success` keeps track of the outcome of the calculation,

the value 2 indicates that the root has been successfully calculated within the required tolerance, given as $5 \cdot 10^{-10}$ by the constant `tol` specified in line 19. The value 1 indicates that the tangent line is too flat at the current approximation (in which case the next approximation cannot be calculated with sufficient precision). Finally, the value 0 indicates that the maximum number of iterations (50 at present, specified in the calling statement in line 23) has been exceeded. This program produces the following output:

```
1 Iteration number  0: x= 3.000000000000 and f(x)=-7.666666666667
2 Iteration number  1: x= 1.644576458341 and f(x)=-2.518501258529
3 Iteration number  2: x= 0.651829979669 and f(x)=-0.037017477051
4 Iteration number  3: x= 0.641077330609 and f(x)= 0.000380944220
5 Iteration number  4: x= 0.641185733066 and f(x)= 0.000000040191
6 Iteration number  5: x= 0.641185744505 and f(x)= 0.000000000000
7 The root 0.641185744505. The value of  the function
8 at the root is -0.000000000000.
9 5 iterations were used to find  the root
```

The Illinois method, a modification of the secant method, is given in the file `secant.c`:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 5
 6 double funct(double);
 7 double secant(double (*fnct)(double),
 8   double x0, double x1, double xtol,
 9   int maxits, double *fatroot,
10   int *itcount, int *outcome);
11
12 main()
13 {
14   /* This program implements the Illinois variant of
15      the secant method for solving an equation funct(x)=0.
16      The function funct() is defined separately.        */
17   const double tol=5e-10;
18   double x, fx, root;
19   int its, success;
20   root = secant(&funct, 0.01, 1.0, tol,
21     50, &fx,
22     &its, &success);
23   if ( success == 2 ) {
24     printf("The root %.12f. The value of "
25            " the function\nat the root is %.12f.\n", root, fx);
26     printf("%u iterations were used to find"
27            " the root\n", its);
28   }
29   else if (success == 0) {
30     printf("The maximum number of iterations has been reached\n");
31   }
32 }
33
34 double secant(double (*fnct)(double),
```

```
35   double x0, double x1, double xtol,
36   int maxits, double *fatroot,
37   int *itcount, int *outcome)
38 {
39   double xold, xnew, dx, dfx, f0, f1, f2, fbar, fold,
40     root, assumedzero=1e-20, alpha;
41   *outcome = 0;
42   alpha = 0.5;
43   /* This called the Illinois method. There are other,
44      more complicated choices for alpha.            */
45     f0 = fnct(x0); f1 = fnct(x1);
46   for (*itcount = 0; *itcount < maxits; (*itcount)++) {
47   /* The next two lines are included so as to monitor
48      the progress of the calculation. These lines
49      should be deleted from a program of
50      practical utility. */
51     printf("Iteration number %2u: ", *itcount);
52     printf("x=% .12f and f(x)=%15.12f\n", x1, f1);
53     if ( *itcount == 0 || f0<=0 && f1>=0 || f0>=0 && f1<=0
54          || fold<=0 && f1<=0 || fold>=0 && f1>=0 ) {
55       xnew = x1-f1*(x1-x0)/(f1-f0);
56     }
57     else {
58       fbar = alpha*fold;
59       xnew= x1-f1*(xold-x1)/(fbar-f1);
60     }
61     if ( absval(xnew-x1)/(absval(x1)+assumedzero) <= xtol ) {
62       *outcome = 2; /* within tolerance */
63       root = xnew;
64       *fatroot = fnct(root);
65       break;
66     }
67     xold = x0; x0 = x1; x1 = xnew;
68     fold = f0; f0 = f1; f1 = fnct(xnew);
69   }
70   return root; /* returning the value of the root */
71 }
```

Here we still use the value 2 for the variable `success` (see line 23) to indicate that the method was successful. The value 1 for `success` is no longer used, but at the price of some additional calculations, one could add the option of telling the user that the secant line is too flat, so the root cannot be calculated. Eventually, the user will find it out either by having the maximum number of iterations exceeded or by receiving a floating point exception (overflow, because of dividing by a number too close to zero). The calling statement in lines 20–22 specifies the starting values $x_1 = 0.01$ and $x_2 = 1$ in line 20, the same starting values we used for bisection. The output of this program is as follows:

```
1 Iteration number  0: x= 1.000000000000 and f(x)=-1.000000000000
2 Iteration number  1: x= 0.990099311353 and f(x)=-0.976322026870
3 Iteration number  2: x= 0.971140749370 and f(x)=-0.930673220217
4 Iteration number  3: x= 0.584619671930 and f(x)= 0.210870175492
5 Iteration number  4: x= 0.656019294352 and f(x)=-0.051383435810
```

```
 6 Iteration number  5: x= 0.642029942985 and f(x)=-0.002963594600
 7 Iteration number  6: x= 0.640460359614 and f(x)= 0.002550386255
 8 Iteration number  7: x= 0.641186340346 and f(x)=-0.000002093441
 9 Iteration number  8: x= 0.641185744926 and f(x)=-0.000000001479
10 Iteration number  9: x= 0.641185744085 and f(x)= 0.000000001476
11 Iteration number 10: x= 0.641185744505 and f(x)=-0.000000000000
12 The root 0.641185744505. The value of  the function
13 at the root is -0.000000000000.
14 10 iterations were used to find the root
```

A single makefile, usually called `makefile`, can be used to compile all these programs:

```
 1 all: bisect newton secant
 2 bisect : funct.o bisect.o
 3        gcc -o bisect -s -O4 funct.o bisect.o -lm
 4 newton : funct.o newton.o dfunct.o
 5        gcc -o newton -s -O4 funct.o dfunct.o newton.o -lm
 6 secant : funct.o secant.o
 7        gcc -o secant -s -O4 funct.o secant.o -lm
 8 funct.o : funct.c
 9        gcc -c -O4 funct.c
10 dfunct.o : dfunct.c
11        gcc -c -O4 dfunct.c
12 bisect.o : bisect.c
13        gcc -c -O4 bisect.c
14 newton.o : newton.c
15        gcc -c -O4 newton.c
16 secant.o : secant.c
17        gcc -c -O4 secant.c
```

As we pointed out before, lines 3, 5, 7, 9, 11, 13, 15, 17 here start with a tab character, and there is no space character before the first letter in these lines.

14. NEWTON'S METHOD FOR POLYNOMIAL EQUATIONS

Given the polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^{n-k} = a_0 x^n + a_1 x^{n-1} + \ldots + a_n \\ &= (\ldots((a_0 x + a_1)x + a_2)x + \ldots + a_{n-1})x + a_n. \end{aligned}$$

it is easy to calculate the value of $P(x_0)$: put

$$b_0 = a_0 \quad \text{and} \quad b_k = a_k + b_{k-1}x_0 \quad \text{for } k \text{ with } 0 < k \leq n. \tag{1}$$

Then $P(x_0) = b_n$. This method of calculating the value of a polynomial is called Horner's rule. If we write

$$Q(x) = \sum_{k=0}^{n-1} b_k x^{n-1-k},$$

it is easy to see that

$$P(x) = (x - x_0)Q(x) + b_n. \tag{2}$$

Indeed, using the above expression for $Q(x)$, the right-hand side can be written as

$$(x-x_0)Q(x)+b_n=(x-x_0)\sum_{k=0}^{n-1}b_kx^{n-k-1}+b_n=x\sum_{k=0}^{n-1}b_kx^{n-k-1}+b_n-x_0\sum_{k=0}^{n-1}b_kx^{n-k-1}$$
$$=\sum_{k=0}^{n-1}b_kx^{n-k}+b_n-\sum_{k=0}^{n-1}b_kx^{n-k-1}x_0=\sum_{k=0}^{n}b_kx^{n-k}-\sum_{k=1}^{n}b_{k-1}x^{n-k}x_0.$$

Here, in obtaining the last equality, the term b_n was incorporated into the first sum, and in the second sum, k was replaced with $k+1$ (so, in the second sum on the right-hand side, k goes from 1 to n instead of going from 0 to $n-1$). If we make the assumption that $b_{-1}=0$ (b_{-1} has not been defined so far), we can extend the summation in the second sum on the right-hand side to $k=0$:

$$\sum_{k=0}^{n}b_kx^{n-k}-\sum_{k=0}^{n}b_{k-1}x^{n-k}x_0=\sum_{k=0}^{n}(b_kx^{n-k}-b_{k-1}x^{n-k}x_0)$$
$$=\sum_{k=0}^{n}(b_k-b_{k-1}x_0)x^{n-k}=\sum_{k=0}^{n}a_kx^{n-k}=P(x);$$

The penultimate[37] equality holds because we have

$$a_k=b_k-b_{k-1}x_0\qquad(0\le k\le n)$$

according to (1) (the case $k=0$ also follows, since we assumed that $b_{-1}=0$). Finally, taking the derivative of (2), we obtain

$$P'(x)=(x-x_0)Q'(x)+Q(x).$$

Substituting $x=x_0$, we obtain

$$P'(x_0)=Q(x_0).$$

These reflections allow us to draw the following conclusions. 1) Equations (1) allow a speedy evaluation of the polynomial $P(x)$ at x_0. 2) These equations also allow us to evaluate the coefficients of the polynomial $Q(x)$. Hence the derivative of $P(x)$ can also be easily evaluated at x_0 (since we only need to evaluate $Q(x_0)$ for this, using the same method). Hence, we can use Newton's method to approximate the zero of $P(x)$.

Once a root α_0 of $P(x)=0$ is found, one can look for further roots of the equation by looking for zeros of the polynomial $P(x)/(x-\alpha_0)$. Here one can observe that 3) the coefficients of $P(x)/(x-\alpha_0)$ can be evaluated by using equations (1) with α_0 replacing x_0. This is because the b_k's are the coefficients of the quotient of dividing $x-x_0$ into $P(x)$. Division by the factor $x-\alpha_0$ corresponding to the root α_0 is called *deflation*, and the quotient $P_1(x)$ is called the *deflated* polynomial. Once an approximate zero β_1 of the deflated polynomial is found (by Newton's method, say), one can use this approximation as the starting value for Newton's method to find the root of the equation $P(x)=0$. The reason for doing this is to eliminate the errors resulting from the fact that the coefficients of $P_1(x)$ are only approximate values. If one continues this procedure, and deflates a polynomial several times, the errors might accumulate to an unacceptable level unless each time one goes back to the original polynomial to refine the approximate root.

An illustration how this works is given in the following program to solve a polynomial equation. The file `newton.c` is essentially the same as the program performing Newton's method above (only some `fprint` statements used to monitor the progress of the calculation were deleted):

```
1 #include "newton.h"
```

[37] Last but one.

```
 2
 3 double newton(double (*fnct)(double), double (*deriv)(double),
 4    double startingval, double xtol, int maxits, double *fx,
 5    int *itcount, int *outcome)
 6 {
 7   double x, dx, dfx, assumedzero=1e-20;
 8   *outcome = 0;
 9   x = startingval;
10   for (*itcount = 0; *itcount < maxits; (*itcount)++) {
11     dfx = deriv(x);
12     if ( absval(deriv(x))<=assumedzero ) {
13       *outcome = 1; /* too flat */
14       break;
15     }
16     *fx = fnct(x);
17     dx = -*fx/dfx; x = x+dx;
18     if ( absval(dx)/(absval(x)+assumedzero) <= xtol ) {
19       *outcome = 2; /* within tolerance */
20       *fx = fnct(x);
21       break;
22     }
23   }
24   return x; /* returning the value of the root */
25 }
```

The first line here invokes the header file `newton.h`:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 #include <float.h>
 4 #include <stdlib.h>
 5
 6 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 7
 8 double funct(double);
 9 double dfunct(double);
10 double origfunct(double);
11 double dorigfunct(double);
12 double evalp(double a[], int n, double x, double b[]);
13 double newton(double (*fnct)(double), double (*deriv)(double),
14    double startingval, double xtol, int maxits, double *fx,
15    int *itcount, int *outcome);
```

The function `evalp` to evaluate polynomials is given in the file `horner.c`:

```
 1 #include <math.h>
 2
 3 double evalp(double a[], int n, double x, double b[])
 4 {
 5   double value;
 6   int k;
 7   b[0]=a[0];
 8   for ( k=1; k<=n; k++ ){
 9     b[k]=a[k]+b[k-1]*x;
```

```
10   }
11   return b[n];
12 }
```

As has been explained above, Horner's method can be used to evaluate a polynomial, evaluate its derivative, and to deflate it (i.e., to divide it by the linear factor corresponding to the root) once a root is known. The function `evalp()` takes returns the value of a polynomial at `x` with coefficient degree `n` and vector `a` (with `a[n]` being the coefficient of the highest degree term). The vector `b` is used to store the coefficients of the polynomial $Q(x)$ in equation (2) above.

The main program is contained in the file `callnewton.c`:

```
 1 #include "newton.h"
 2
 3 double aa[20], a[20];
 4 int nn, n;
 5
 6 main()
 7 {
 8   /* This program implements the Newton's method
 9      for solving an equation funct(x)=0. The function
10      funct() and its derivative dfunct() is defined
11      separately.                                    */
12   const double tol=5e-10;
13   double x, y, fx, root, b[20];
14   int its, k, success, random, count;
15   extern double aa[], a[];
16   extern int nn, n;
17   char s[20];
18   FILE *coefffile;
19   coefffile=fopen("coeffs", "r");
20   for (nn=0; fscanf(coefffile, "%s", s) !=EOF; nn++) {
21     aa[nn]=strtod(s,NULL);
22   }
23   nn--;
24   n = nn;
25     for (k=0; k<=n; k++) {
26       a[k] = aa[k];
27     }
28   while ( n > 0 ) {
29     root = newton(&funct, &dfunct,
30       0.0, tol, 50, &fx,
31       &its, &success);
32     if ( n < nn && success == 2 ) {
33       root = newton(&origfunct, &dorigfunct,
34         root, tol, 50, &fx,
35         &its, &success);
36     }
37     if ( success == 2 ) {
38       printf("Root number %d is %.12f. The value of "
39              " the function\nat the root is %.12f.\n",
40              nn-n+1, root, fx);
41     }
```

```
42     else {
43       printf("The procedure does not converge\n");
44       break;
45     }
46     evalp(a,n,root,b);
47     n--;
48     for (k=0; k<=n; k++) {
49       a[k] = b[k];
50     }
51   }
52 }
53 double origfunct(double x) {
54   extern double aa[];
55   extern int nn;
56   double b[20];
57   return evalp(aa, nn, x, b);
58   }
59 double dorigfunct(double x) {
60   extern double aa[];
61   extern int nn;
62   double b[20], c[20];
63   evalp(aa, nn, x, b);
64   return evalp(b, nn-1, x, c);
65 }
66 double funct(double x) {
67   extern double a[];
68   extern int n;
69   double b[20];
70   return evalp(a, n, x, b);
71   }
72 double dfunct(double x) {
73   extern double a[];
74   extern int n;
75   double b[20], c[20];
76   evalp(a, n, x, b);
77   return evalp(b, n-1, x, c);
78 }
```

In lines 4–5, the variables are declared outside the functions, because these variables are needed by several functions in this file. The integer `nn` will store the degree of the original polynomial, `n` will store the degree of the deflated polynomial (this will initially be `nn`, and will decrease gradually as more roots are found), the array will `aa` contain the coefficients of the original polynomial, and `a` will contain those of the deflated polynomial. The functions in lines 53–78 are used to provide an interface between the function `evalp()` mentioned above, implementing Horner's rule, and the functions `funct()` used by the file `newton.c` implementing Newton's method.[38] The functions `origfunct()` and `dorigfunct()` evaluate the original polynomial and its derivative, while the functions `funct()` and `funct()` evaluate the deflated polynomial and its derivative.

The coefficients of the polynomial are contained in the file `coeffs`; this file is opened for reading

[38] This was done so that the program implementing Newton's method could be used without change. An alternative, and perhaps more efficient, but possibly less readable, solution would be to rewrite this program, so that the functions providing these interfaces are not needed.

in line 18, and the coefficients are read into the array `aa[]` in lines 19–22. The numbers in this file are first read as character strings, and in line 21 they are converted to double[39] At the same time, the degree of the polynomial is calculated by counting the number of the coefficients read, and it is stored in the integer `nn`; `nn` needs to be decremented in line 23, since it was incremented at the end of the `for` loop in line 20. In lines 27–27 the array `aa[]` is copied into the array `a[]` since initially the deflated polynomial is the same as the original polynomial (`nn` is also copied to `n` in line 24).

The calculation of the roots and the deflation of the polynomial is performed in the loop in lines 28–51. In line 28, Newton's method is called with initial value 0.0 to calculate the next root of the deflated polynomial. If this calculation is successful then in lines 32–34 this root is refined by calling Newton's for the original polynomial with the the root of the deflated polynomial just calculated being used as initial value. This is done only for `n<nn`, since in case `n=nn` the deflated polynomial agrees with the original polynomial. If at any time in the process, Newton's method is unsuccessful, the loop is broken in line 44, If the determination of the current root is successful, then the result is printed out in lines 32–26.

Of course, there is no guarantee that Newton's method will find the root of a polynomial, so there is a good chance that the program will terminate unsuccessfully. Several measures could be taken to prevent this. There are ways to estimate the size of the roots of a polynomial. If at any time during the calculation, too large a value is used to approximate the root, Newton's method could be terminated and restarted with a new value. In this way one could make it likely that Newton's method calculates the real roots of a polynomial. The program was run with the following coefficient file `coeffs`:

```
1 1.0 4.33 -19.733. -74.7168
```

The `1` at the beginning of the line is a line number, and not a part of the file. The roots were successfully determined, and the printout was the following

```
1 Root number 1 is -3.200000000000. The value of  the function
2 at the root is -0.000000000000.
3 Root number 2 is 4.300000000000. The value of  the function
4 at the root is 0.000000000000.
5 Root number 3 is -5.430000000000. The value of  the function
6 at the root is 0.000000000000.
```

Problems

1. Evaluate the derivative of $P(x) = 2x^3 + 5x^2 + 3x + 4$ at $x = -2$ using Horner's method. Show the details of your calculation.

Solution. We have $a_0 = 2$, $a_1 = 5$, $a_2 = 3$, $a_3 = 4$, and $x_0 = -2$. Further, we have $b_0 = a_0$ and $b_k = a_k + b_{k-1}x_0$ for k with $0 < k \leq 3$. Therefore,

$$b_0 = a_0 = 2,$$

$$b_1 = a_1 + b_0 x_0 = 5 + 2 \cdot (-2) = 1,$$

$$b_2 = a_2 + b_1 x_0 = 3 + 1 \cdot (-2) = 1,$$

$$b_3 = a_3 + b_2 x_0 = 4 + 1 \cdot (-2) = 2.$$

Actually, we did not need to calculate b_3, since it is not used in calculating the derivative. The derivative as $x = -2$ is the value for $x = -2$ of the polynomial $b_0x^2 + b_1x + b_2$. Using Horner's

[39] the library function `fscan` seems to handle numbers of type `float` well, but not those of type `double`, so reading the numbers as character strings and then converting them appears to be safer.

rule, this can be calculated by first calculating the coefficients $c_0 = b_0$ and $c_k = b_k + c_{k-1}$ for k with $0 < k \leq 2$, and then value of the polynomial being considered will be c_2. That is,

$$c_0 = b_0 = 2,$$

$$c_1 = b_1 + c_0 x_0 = 1 + 2 \cdot (-2) = -3,$$

$$c_2 = b_2 + c_1 x_0 = 1 + (-3) \cdot (-2) = 7.$$

That is, $P'(-2) = c_2 = 7$. It is easy to check that this result is correct. There is no real saving when the calculation is done for a polynomial of such low degree. For higher degree polynomials, there is definitely a saving in calculation. Another advantage of the method, especially for computers, is that the formal differentiation of polynomials can be avoided.

2. Evaluate the derivative of $P(x) = x^3 - 4x^2 + 6x + 4$ at $x = 2$ using Horner's method. Show the details of your calculation.

Solution. We have $a_0 = 1$, $a_1 = -4$, $a_2 = 6$, $a_3 = 4$, and $x_0 = 2$. Further, we have $b_0 = a_0$ and $b_k = a_k + b_{k-1} x_0$ for k with $0 < k \leq 3$. Therefore,

$$b_0 = a_0 = 1,$$

$$b_1 = a_1 + b_0 x_0 = -4 + 1 \cdot 2 = -2,$$

$$b_2 = a_2 + b_1 x_0 = 6 + (-2) \cdot 2 = 2,$$

$$b_3 = a_3 + b_2 x_0 = 4 + 2 \cdot 2 = 8.$$

Actually, we did not need to calculate b_3, since it is not used in calculating the derivative. The derivative as $x = 2$ is the value for $x = 2$ of the polynomial $b_0 x^2 + b_1 x + b_2$. Using Horner's rule, this can be calculated by first calculating the coefficients $c_0 = b_0$ and $c_k = b_k + c_{k-1}$ for k with $0 < k \leq 2$, and then value of the polynomial being considered will be c_2. That is,

$$c_0 = b_0 = 1,$$

$$c_1 = b_1 + c_0 x_0 = -2 + 1 \cdot 2 = 0,$$

$$c_2 = b_2 + c_1 x_0 = 2 + 0 \cdot 2 = 2.$$

That is, $P'(2) = c_2 = 2$. It is easy to check that this result is correct. There is no real saving when the calculation is done for a polynomial of such low degree. For higher degree polynomials, there is definitely a saving in calculation. Another advantage of the method, especially for computers, is that the formal differentiation of polynomials can be avoided.

3. Let P and Q be polynomials, let x_0 and r be a numbers, and assume that

$$P(x) = (x - x_0)Q(x) + r.$$

Show that $P'(x_0) = Q(x_0)$.

Solution. We have

$$P'(x) = Q(x) + (x - x_0)Q'(x)$$

simply be using the product rule for differentiation. Substituting $x = x_0$, we obtain that $P'(x_0) = Q(x_0)$.

Note: The coefficients of the polynomial $Q(x)$ can be produced by Horner's method. By another use of Horner's method, we can evaluate $Q(x_0)$. This provides an efficient way to evaluate $P'(x)$ on computers without using symbolic differentiation.

15. FIXED-POINT ITERATION

The equation $x = f(x)$ can often be solved by starting with a value $x = x_1$, and using the simple iteration $x_{n+1} = f(x_n)$. This method is called fixed-point iteration, and it is important for theoretical reasons. This is partly because other methods can often be reformulated in terms of fixed-point iteration. For example, when using Newton's method

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)}$$

to solve the equation $g(x) = 0$, this can be considered as using fixed-point iteration to solve the equation

$$x = x - \frac{g(x)}{g'(x)}.$$

Furthermore, variants of fixed-point iteration are useful for solving certain systems of linear equations arising in practice (e.g., on account of cubic splines). A simple result describing the solvability of equations by fixed-point iteration is the following:

THEOREM. *Assume $x = c$ is a solution of the equation $x = f(x)$. Assume further that there are numbers $r > 0$ and q with $0 \le q < 1$ such that*

$$|f'(x)| \le q \quad \textit{for all} \quad x \textit{ with } c - r < x < c + r.$$

Then, starting with any value $x_1 \in (c - r, c + r)$ and putting $x_{n+1} = f(x_n)$ for $n \ge 1$, the sequence $\{x_n\}_{n=1}^{\infty}$ converges to c.

When one tries to use this result in practice, the interval $(c-r, c+r)$ of course cannot be known exactly. However, the fact that $|f'(x)| < q$ in an interval for some q with $0 \le q < 1$ makes it reasonable to try to use fixed-point iteration to find a solution of the equation $x = f(x)$ in this interval.

PROOF. Let $n \ge 1$, and assume that $x_n \in (c - r, c + r)$. By the Mean-Value Theorem of differentiation we have

$$f(x_n) - c = f(x_n) - f(c) = f'(\xi_n)(x_n - c),$$

where the first equality used the assumption that c is a root of $x = f(x)$, i.e., that $c = f(c)$; here ξ_n is some number between x_n and c. Clearly, we have $\xi_n \in (c - r, c + r)$, so we have $|f'(\xi_n)| \le q$. Therefore, noting that $x_{n+1} = f(x_n)$, we have

$$|x_{n+1} - c| \le q|x_n - c|. \tag{1}$$

Hence $x_{n+1} \in (c-r, c+r)$. Thus, given that $x_1 \in (c-r, c+r)$ by assumption, we can conclude that $x_n \in (c-r, c+r)$ for all positive integers n. Hence inequality (1) is valid for all positive integers n; thus we can conclude by induction that

$$|x_n - c| \le q^{n-1}|x_1 - c|$$

for all positive integers n. As q^{n-1} converges to 0 when n tends to zero, it follows that x_n converges to c.

A partial converse to this is the following

THEOREM. *Assume $x = c$ is a solution of the equation $x = f(x)$. Assume further that there are numbers a, b with $a < c < b$ such that*

$$|f'(x)| \geq 1 \quad \textit{for all} \quad x \textit{ with } a < x < b.$$

Then starting with any value x_1 and putting $x_{n+1} = f(x_n)$ for $n \geq 1$, the sequence $\{x_n\}_{n=1}^{\infty}$ does not converge to c unless $x_k = c$ for some positive integer k.

Of course, if $x_k = c$ for some positive integer k, then $x_n = c$ for all $n \geq k$. However, it is very unlikely that we accidentally end up with $x_k = c$ in practice, and so, if $|f'(x)| > 1$ near the solution of the equation $x = f(x)$, using fixed-point iteration to find the solution should be considered hopeless.

PROOF. Assuming x_n converges to c, we must have a positive integer N such that $x_n \in (a, b)$ for every $n \geq N$. So, for any $n \geq N$ we have, by the Mean-Value Theorem that

$$f(x_n) - c = f(x_n) - f(c) = f'(\xi_n)(x_n - c),$$

for some $\xi_n \in (a, b)$, and so, noting that $x_{n+1} = f(x_n)$ and that $|f'(\xi_n)| \geq 1$, we obtain that

$$|x_{n+1} - c| \geq |x_n - c|$$

for all $n \geq N$. Thus we have

$$|x_n - c| \geq |x_N - c|$$

for all $n \geq N$. Hence x_n cannot converge to c unless $x_N = c$.

Fixed point iteration is easy to implement on computer. In a C program doing fixed-point iteration to solve the equation $x = e^{1/x}$, the file `funct.c` implements the simple program calculating the function $e^{1/x}$:

```
#include <math.h>

double funct(double x)
{
  double value;
  value = exp(1.0/x);
  return(value);
}
```

The ideas used in the implementation of fixed-point iteration are similar to the ideas are similar to those used in earlier programs such as `newton.c`. The file fixedpoint.c contains the program:

```
#include <stdio.h>
#include <math.h>

#define absval(x) ((x) >= 0.0 ? (x) : (-(x)))

double funct(double);
double fixedpoint(double (*fnct)(double), double startingval,
     double xtol, int maxits, int *itcount, int *outcome);

main()
{
  /* This program implements fixed-point iteration
     for solving an equation x=funct(x). The function
     funct() is defined separately.                */
```

```
15   const double tol=5e-10;
16   double root;
17   int its, success;
18   root = fixedpoint(&funct, 1.0, tol, 50, &its, &success);
19   if ( success == 2 ) {
20     printf("The root is %.12f\n", root);
21     printf("%u iterations were used to find"
22            " the root\n", its);
23   }
24   else if (success == 0) {
25     printf("The maximum number of iterations has been reached\n");
26   }
27 }
28
29 double fixedpoint(double (*fnct)(double), double startingval,
30      double xtol, int maxits, int *itcount, int *outcome)
31 {
32   double x, oldx, assumedzero=1e-20;
33   *outcome = 0;
34   x = startingval;
35   for (*itcount = 0; *itcount < maxits; (*itcount)++) {
36     oldx = x;
37     x = fnct(x);
38   /* The next line is included so as to monitor
39      the progress of the calculation. This line
40      should be deleted from a program of
41      practical utility. */
42     printf("Iteration number %2u: ", *itcount);
43     printf("x=% .12f\n", x);
44     if ( absval(x-oldx)/(absval(oldx)+assumedzero) <= xtol ) {
45       *outcome = 2; /* within tolerance */
46       break;
47     }
48   }
49   return x; /* returning the value of the root */
50 }
```

The calling statement on line 18 uses the starting value $x = 1$, and limits the number of iterations to 50. The printout of the program is as follows:

```
 1 Iteration number  0: x= 2.718281828459
 2 Iteration number  1: x= 1.444667861010
 3 Iteration number  2: x= 1.998107789671
 4 Iteration number  3: x= 1.649502126004
 5 Iteration number  4: x= 1.833530851751
 6 Iteration number  5: x= 1.725291093281
 7 Iteration number  6: x= 1.785346181250
 8 Iteration number  7: x= 1.750874646996
 9 Iteration number  8: x= 1.770289539914
10 Iteration number  9: x= 1.759235513562
11 Iteration number 10: x= 1.765490799373
12 Iteration number 11: x= 1.761938693392
```

```
13 Iteration number 12: x= 1.763951807726
14 Iteration number 13: x= 1.762809621275
15 Iteration number 14: x= 1.763457255891
16 Iteration number 15: x= 1.763089906433
17 Iteration number 16: x= 1.763298230825
18 Iteration number 17: x= 1.763180076095
19 Iteration number 18: x= 1.763247085165
20 Iteration number 19: x= 1.763209080909
21 Iteration number 20: x= 1.763230634603
22 Iteration number 21: x= 1.763218410517
23 Iteration number 22: x= 1.763225343308
24 Iteration number 23: x= 1.763221411417
25 Iteration number 24: x= 1.763223641361
26 Iteration number 25: x= 1.763222376662
27 Iteration number 26: x= 1.763223093927
28 Iteration number 27: x= 1.763222687135
29 Iteration number 28: x= 1.763222917845
30 Iteration number 29: x= 1.763222786999
31 Iteration number 30: x= 1.763222861208
32 Iteration number 31: x= 1.763222819121
33 Iteration number 32: x= 1.763222842990
34 Iteration number 33: x= 1.763222829453
35 Iteration number 34: x= 1.763222837130
36 Iteration number 35: x= 1.763222832776
37 Iteration number 36: x= 1.763222835246
38 Iteration number 37: x= 1.763222833845
39 Iteration number 38: x= 1.763222834639
40 The root is 1.763222834639
41 38 iterations were used to find the root
```

Problems

1. Rearrange the equation $x + 1 = \tan x$ so that it be solvable by fixed-point iteration.

Solution. In order to solve the equation by fixed-point iteration, it needs to be written in the form $x = f(x)$, and we also must have $|f(x)| < q$ with some $q < 1$ near the solution. There are several ways to write the above equation in the form $x = f(x)$, for example

$$x = \tan x - 1.$$

This will definitely not work, since the derivative of the right-hand side is $1 + \tan^2 x$, and this is always greater than 1 except at $x = 0$. On the other hand, one can also write

$$x = \arctan(x + 1) + k\pi,$$

where k can be any integer (positive, negative, or zero). There are infinitely many solutions of this equation. It is easy to see by inspecting the graphs of $y = x$ and $y = \arctan x + k\pi$ that they will intersect exactly once for each value of k. The derivative of the right-hand side is

$$\frac{1}{(x+1)^2 + 1},$$

and for any value of x this is less or equal to 1; equalility holds only in case $x = -1$. Thus this form is well-suited for fixed-point iteration. For each value of k, one can use the starting value $x = 0$, for example.

2. The equation $e^x - x - 2 = 0$ has one positive solution and one negative solution. Rearrange the equation so that each of these solutions can be found by fixed-point iteration.

Solution. To find the negative solution, write $x = f(x)$ with $f(x) = e^x - 2$. Then $f'(x) = e^x$, and so, for any number $c < 0$ we have $0 < f'(x) \le e^c < 1$ for $x \le c$. Thus one can find the negative solution by using any starting point $x_1 < 0$ and using the iteration $x_{n+1} = e^{x_n} - 2$ (actually $x_1 = 0$ would also work, since then $x_2 = -1$).

To find the positive solution, write $f(x) = \ln(x + 2)$. Then we have $f'(x) = \frac{1}{x+2}$, and so for any $c > -1$ we have

$$0 < \frac{1}{x+2} \le \frac{1}{c+2} < 1$$

for any $x > c$. Thus, one can find the positive solution using any starting point $x_1 > -1$ and the iteration $x_{n+1} = \ln(x_n + 2)$; one might as well use the starting point $x_1 = 1$ also in this case. The approximate solutions are -1.84141 and 1.14619.

16. AITKEN'S ACCELERATION

Let α the solution of an equation, and let x_n be the nth approximation to this solution. Write $\epsilon_n = x_n - \alpha$ for the error of the nth approximation. In case we have $\lim_{n\to\infty} x_n = \alpha$ we say that the *order of convergence* of the approximation is λ if

$$\lim_{n\to\infty} \frac{|\epsilon_{n+1}|}{|\epsilon_n|^\lambda} = C \tag{1}$$

for some constant $C \neq 0$, with the additional requirement of $C < 1$ in case $\lambda = 1$. In case $\lambda = 1, 2, 3$ we also talk of *linear*, *quadratic*, or *cubic* convergence, respectively.[40] Intuitively, the higher the order of convergence of a method the the faster the convergence is. For example, in case of linear convergence (with $C = 1/10$), with each step one may expect one additional significant decimal digit in the solution, while with quadratic convergence the number of significant decimal digits would roughly double with each additional step.

With fixed point iteration, when it converges, one usually expect linear convergence. In fact, if $f(\alpha) = 0$ and $x_{n+1} = f(x + n)$, then

$$x_{n+1} - \alpha = f(x_n) - f(\alpha) = f'(\xi_n)(x_n - \alpha),$$

where the second equation is justified by the Mean-Value Theorem (under suitable differentiability assumptions), for some ξ_n between x_n and α, so that

$$\frac{x_{n+1} - \alpha}{x_n - \alpha} = f'(\xi_n) \approx f'(\alpha),$$

provided that f' is continuous at α and x_n (and so also ξ_n) is close enough to α. So, in case $0 < |f'\alpha)| < 1$ one would have linear convergence.

If one knows that we are dealing this linear convergence, there is a method, called Aitken's acceleration, to speed up this convergence. Assume we have

$$\frac{x_{n+1} - \alpha}{x_n - \alpha} = f'(\xi_n) \approx C$$

[40] If $\lambda < 1$ we obviously have divergence. In case $\lambda = 1$ and $C = 1$ we may or may not have convergence, but the convergence is usually much slower than in case $C < 1$. Of course, limit in (1) may not exists, or it may be 0 for each value of λ for which it exist. We do not intend to give a more inclusive definition or more precise definition of the order of convergence here.

for some C with $-1 < C < 1$.[41] Then we have

$$\frac{x_{n+1} - \alpha}{x_n - \alpha} \approx \frac{x_{n+2} - \alpha}{x_{n+1} - \alpha},$$

since both sides are approximately equal to C. Determine x_{n+3}, the next approximation to α, so that, with x_{n+3} replacing α, this equation becomes exact:

$$\frac{x_{n+1} - x_{n+3}}{x_n - x_{n+3}} = \frac{x_{n+2} - x_{n+3}}{x_{n+1} - x_{n+3}}.$$

Thus

$$(x_{n+2} - x_{n+3})(x_n - x_{n+3}) = (x_{n+1} - x_{n+3})^2,$$

i.e.,

$$x_{n+3} = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} = \frac{x_{n+2}x_n - x_{n+1}^2}{x_{n+2} - 2x_{n+1} + x_n}. \tag{2}$$

When one wants to solve the equation $x = f(x)$ with fixed-point iteration combined with Aitken's acceleration, one starts with an approximate solution, does two fixed-point iteration steps, and then does one Aitken acceleration step. That is, given an $n \geq 0$ divisible by 3, one takes $x_{n+1} = f(x_n)$, $x_{n+2} = f(x_{n+2})$, and one calculates x_{n+3} according to (2). When doing so, one uses the middle member of the equations in (2), since the right-hand side can cause greater loss of precision due to the subtraction of quantities of approximately the same size.[42]

Next we will discuss a C program implementation of the method. As in the fixed-point iteration example, we will use the method to find the solution of the equation $x = e^{1/x}$. The program defining this function, contained in the file `funct.c`, is the same as above:

```
1 #include <math.h>
2
3 double funct(double x)
4 {
5   double value;
6   value = exp(1.0/x);
7   return(value);
8 }
```

As for the program performing Aitken acceleration, this time we broke up the calling program and the program performing the work into two separate files and a third header file (so as not to have repeat the same declarations in the other files). The header file `aitken.h` is as follows:

```
1 #include <stdio.h>
2 #include <math.h>
3
4 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
5 #define square(x) ((x)*(x))
6
```

[41] Strictly speaking, this assumption is stronger than the assumption of linear convergence, since equation (1) has absolute values (which is really necessary there, since ϵ_n^λ may not be defined for noninteger values of λ if ϵ_n is negative). However, as we just saw in the example of fixed point iteration, in case of linear convergence the relation usually holds without absolute values.

[42] This is because in the middle member the loss of precision is likely to occur in the fraction that is being subtracted from x_n; since this fraction represents a small correction to the value of x_n, this loss of precision is not as fatal as might occur on the right-hand side.

```
 7 double funct(double);
 8 double aitken(double (*fnct)(double), double startingval,
 9      double xtol, int maxits, int *itcount, int *outcome);
```

The calling program, in the file `callaitken.c` is very similar to the calling program of the fixed-point iteration discussed before:

```
 1 #include "aitken.h"
 2
 3 main()
 4 {
 5   /* This program implements fixed-point iteration
 6      for solving an equation x=funct(x). The function
 7      funct() is defined separately.                  */
 8   const double tol=5e-10;
 9   double root;
10   int its, success;
11   root = aitken(&funct, 1.0, tol, 50, &its, &success);
12   if ( success == 2 ) {
13     printf("The root is %.12f\n", root);
14     printf("%u iterations were used to find"
15            " the root\n", its);
16   }
17   else if (success == 0) {
18     printf("The maximum number of iterations has been reached\n");
19   }
20 }
```

The difference here is that instead of including the function declarations and the standard *include* statements, all these are taken from the file `aitken.h`, included in line 1. Line 11 of the program uses the starting value 1 for the iteration, the same starting value that was used in the fixed-point iteration. The part of the program that performs the main work is contained in the file `aitken.c`:

```
 1 #include "fixedpoint.h"
 2
 3 double aitken(double (*fnct)(double), double startingval,
 4      double xtol, int maxits, int *itcount, int *outcome)
 5 {
 6   double x, x0, x1, oldx, assumedzero=1e-20;
 7   int i=0;
 8   *outcome = 0;
 9   x = startingval;
10
11   for (*itcount = 0; *itcount < maxits; (*itcount)++) {
12     oldx = x;
13     switch(i) {
14       case 0:
15         x0 = x; x = fnct(x);
16         break;
17       case 1:
18         x1 = x; x = fnct(x);
19         break;
20       case 2:
21         x = x0 - square(x1-x0)/(x-2*x1+x0);
```

```
22         break;
23       }
24     /* The next line is included so as to monitor
25        the progress of the calculation. This line
26        should be deleted from a program of
27        practical utility. */
28       printf("Iteration number %2u: ", *itcount);
29       printf("x=% .12f\n", x);
30       if ( absval(x-oldx)/(absval(oldx)+assumedzero) <= xtol ) {
31         *outcome = 2; /* within tolerance */
32         break;
33       }
34       i++;
35       if ( i==3 ) { i=0; }
36     }
37     return x; /* returning the value of the root */
38 }
```

The program is similar to the program performing fixed-point iteration. In the `switch` statement in lines 13–23 decides whether fixed-point iteration (in case `*itcount` is of form $3k$ or $3k+1$ for some integer k) should be used to calculate `x`, the next approximation, or whether the Aitken acceleration step (in case `*itcount` is of form $3k+2$) should be used.

The advantage of breaking up the program in all these files is that most files will never have to be changed when performing the calculation with different functions. In fact, only the files `funct.c` and `callaitken.c` need to be changed; the former when the function f in the equation $x = f(x)$ to be solved is changed, and the latter when one wants to change starting value, the precision of the solution required (described by the parameter `tol`) or the maximum number of iterations permitted. In fact, it would be easy to arrange for reading these parameters on the command line and never change the calling program if one only wants to solve one equation. However, often solving an equation is only a part of a larger calculation; in this case the calling program might still have to be changed. The following `makefile` can be used to compile the programs scattered in the files above:

```
1 all: aitken
2 aitken : funct.o aitken.o callaitken.o
3         gcc -o aitken -s -O4 callaitken.o funct.o \
4    aitken.o -lm
5 funct.o : funct.c
6         gcc -c -O4 funct.c
7 callaitken.o : callaitken.c aitken.h
8         gcc -c -O4 callaitken.c
9 aitken.o : aitken.c aitken.h
10        gcc -c -O4 aitken.c
```

The backslash `\` at the end of line 3 is the standard way of breaking up a line in Unix. The backslash quotes the newline character at the end of the line, and when quoted, the newline character no longer signifies the end of a command. That is, the command started in line 3 continues in line 4. Line 4 begins with three space characters; they are not required, but they make the layout more appealing. The printout of the program is as follows:

```
1 Iteration number  0: x= 2.718281828459
2 Iteration number  1: x= 1.444667861010
3 Iteration number  2: x= 1.986829971168
4 Iteration number  3: x= 1.654194746033
5 Iteration number  4: x= 1.830380270380
```

```
 6 Iteration number  5: x= 1.769373837327
 7 Iteration number  6: x= 1.759749886372
 8 Iteration number  7: x= 1.765197485602
 9 Iteration number  8: x= 1.763228455409
10 Iteration number  9: x= 1.763219646420
11 Iteration number 10: x= 1.763224642370
12 Iteration number 11: x= 1.763222834357
13 Iteration number 12: x= 1.763222834349
14 The root is 1.763222834349
15 12 iterations were used to find the root
```

One might compare this to fixed point iteration, where 38 iterations were used to find the solution of the same equation, using the same starting value and requiring the same precision.

Problem

1. Explain in how Aitken's acceleration for fixed point iteration works.

Solution. In solving the equation $f(x) = x$, one starts out with an approximation x_0 to the solution. Each time one does two steps of fixed-point iteration and then one step of Aitken acceleration. That is, if n is divisible by 3, then one takes $x_{n+1} = f(x_n)$ and $x_{n+2} = f(x_{n+1})$, and

$$x_{n+3} = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n}.$$

17. THE SPEED OF CONVERGENCE OF NEWTON'S METHOD

In solving the equation $f(x) = 0$, Newton's method starts with an approximate solution x_0, and for $n \geq 0$ one puts

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1}$$

Let α be a solution of the above equation, i.e., $f(\alpha) = 0$, and assume that α is not a multiple root, i.e., that $f'(\alpha) \neq 0$. Assuming that f' is continuous near α and that x_n is close enough to α, we will also have $f'(x_n) \neq 0$, so that equation (1) will make sense. Assume, further, that f is twice differentiable in an open interval containing α and x_n. By the Taylor expansion of f at x_n with the Lagrange remainder term we have

$$f(\alpha) = f(x_n) + (\alpha - x_n)f'(x_n) + \frac{f''(\xi_n)}{2}(\alpha - x_n)^2$$

with some ξ_n between x_n and α. Noting that $f(\alpha) = 0$, this equation can be written as

$$0 = f(x_n) + (\alpha - x_n)f'(x_n) + \frac{f''(\xi_n)}{2}(\alpha - x_n)^2.$$

Dividing by $f'(x_n)$ (recall that $f'(x_n) \neq 0$ by our assumptions above) and rearranging the equation, we obtain

$$x_n - \frac{f(x_n)}{f'(x_n)} - \alpha = \frac{f''(\xi_n)}{2f'(x_n)}(\alpha - x_n)^2.$$

Using equation (1), this leads to

$$x_{n+1} - \alpha = \frac{f''(\xi_n)}{2f'(x_n)}(\alpha - x_n)^2.$$

That is,

$$\frac{x_{n+1} - \alpha}{(x_n - \alpha)^2} = \frac{f''(\xi_n)}{2f'(x_n)}.$$

Assuming that $\lim_{n\to\infty} x_n = \alpha$, we can see that also $\lim_{n\to\infty} \xi_n = \alpha$ (since ξ_n is between x_n and α). Assuming, further, that f'' is continuous at α, it follows that $\lim_{n\to\infty} f''(\xi_n) = f''(\alpha)$, Hence,

$$\lim_{n\to\infty} \frac{x_{n+1} - \alpha}{(x_n - \alpha)^2} = \frac{f''(\alpha)}{2f'(\alpha)}.$$

Hence Newton's method has quadratic convergence for simple roots.

18. NUMERICAL DIFFERENTIATION OF TABLES

When differentiating a function given by a table (for example, of experimental data), one can approximate the derivative by using a polynomial interpolation formula:

$$f(x) = P(x) + E(x),$$

where $P(x)$ is the interpolation polynomial (given either in the Lagrange or the Newton form), and, for the error term we have

$$E(x) = f^{(n)}(\xi)\frac{p(x)}{n!} = f[x, x_1, \ldots, x_n]\, p(x), \tag{1}$$

where

$$p(x) = \prod_{j=1}^{n}(x - x_j),$$

and ξ is a point in the open interval spanned by the points x, x_1, …, x_n; see (1) and (2) in Section 8 on the error term of the Newton interpolation polynomial. Thus

$$f'(x) = P'(x) + E'(x).$$

There is no difficulty here in determining $P'(x)$. As for $E'(x)$, we have

$$E'(x) = \frac{df^{(n)}(\xi)}{dx}\frac{p(x)}{n!} + f^{(n)}(\xi)\frac{p'(x)}{n!}. \tag{2}$$

The first term on the right-hand side here presents some difficulties, since the dependence of ξ on x is not known; the only thing that is known that ξ is somewhere in the interval spanned by the points x, x_1, …, x_n. If x is one of the interpolation points, we have $p(x) = 0$, so the first term will be zero anyway. That is, at any one of the interpolation points, we have

$$E'(x) = f^{(n)}(\xi)\frac{p'(x)}{n!}.$$

Assume the points $x_1, \ldots, x_n$ are distinct and x belong to the open interval spanned by $x_1, \ldots, x_n$. Assume, further, that f is differentiable $n+1$ times. Then $E'(x)$ can be estimated even if $x \neq x_i$ for any i with $1 \leq i \leq n$. Using the the second expression in (1), the right-hand side of (2) can also be written as we obtain

$$\begin{aligned} E'(x) &= \frac{df[x, x_1, \ldots, x_n]}{dx}\, p(x) + f[x, x_1, \ldots, x_n]\, p'(x) \\ &= f[x, x, x_1, \ldots, x_n]\, p(x) + f[x, x_1, \ldots, x_n]\, p'(x) = f^{(n+1)}(\eta)\frac{p(x)}{(n+1)!} + f^{(n)}(\xi)\frac{p'(x)}{n!}, \end{aligned}$$

where the second equality holds according to the Lemma in Section 8, and the third equality holds for some ξ and η in the open interval spanned by the the numbers $x_1, \ldots, x_n$, according to (1) above and to the Theorem in Section 8 on the error term of the Newton interpolation polynomial. That is, on the right-hand side, ξ and η are appropriate numbers in the interval spanned by the the numbers $x_1, \ldots, x_n$.

Another, more complicated argument is needed to give an error estimate for the derivative of f if f is differentiable only n times. Instead of restricting us to the first derivative, we will discuss the kth derivative for k with $1 \leq k < n$, where n is the number of interpolation points. Assuming that f is n times differentiable, $E(x)$ must also be differentiable n times.[43] Furthermore, $E(x) = 0$ at each of the interpolation points $x_1, \ldots, x_n$. Therefore, by Rolle's Theorem, its kth derivative $E^{(k)}$ has at least $n - k$ zeros in the interval spanned by the points $x_1, \ldots, x_n$. Let these points be $\eta_1, \ldots, \eta_{n-k}$. Then,

$$f^{(k)}(x) = P^{(k)}(x) + E^{(k)}(x).$$

Here $P^{(k)}(x)$ is a polynomial of degree $\leq n - k - 1$, and the error term $E^{(k)}(x)$ is zero at the points $\eta_1, \ldots, \eta_{n-k}$. That is, $P^{(k)}(x)$ interpolates $f^{(k)}(x)$ at these points. Therefore, the error of this interpolation can be expressed as he error of the Lagrange interpolation[44] of the function $f^{(k)}(x)$ at the points $\eta_1, \ldots, \eta_{n-k}$:

$$E^{(k)}(x) = \frac{1}{(n-k)!} \cdot \frac{d^{n-k} f^{(k)}(t)}{dt^{n-k}}\Big|_{t=\xi} \prod_{i=1}^{n-k}(x - \eta_i) = \frac{1}{(n-k)!} \cdot f^{(n)}(\xi) \prod_{i=1}^{n-k}(x - \eta_i),$$

where ξ is in the interval spanned by x and by the points $\eta_1, \ldots, \eta_{n-k}$. The points $\eta_1, \ldots, \eta_{n-k}$ are not known here; but they are known to be located in the interval spanned by $x_1, \ldots, x_n$, and they do not depend on x (since they are the zeros of $E^{(k)}(t)$ guaranteed by Rolle's theorem).

In fact, a little more precise information can be obtained about the location of the η_i's. Assuming $x_1 < x_2 < \ldots < x_n$ and $\eta_1 < \eta_2 < \ldots < \eta_{n-k}$, we can say that

$$x_i < \eta_i < x_{i+k} \qquad \text{for } i \text{ with } 1 \leq i \leq n - k.$$

This can be proved by induction on k. For l with $0 \leq l < n$ write η_{li} with $1 \leq i \leq n - l$ for the zeros of $E^{(i)}(x)$ guaranteed by Rolle's Theorem.[45] Then $\eta_{0i} = x_i$ $(1 \leq i \leq n)$ and $\eta_i = \eta_{ki}$ $(1 \leq i \leq n-k)$. Clearly, $x_i < \eta_{1i} < x_{i+1}$, so for $k = 1$ the above claim about the location of the η_i's is correct. Moreover, for $k > 1$, $\eta_{k-1\,i} < \eta_{ki} < \eta_{k-1\,i+1}$. Assuming the assertion is valid with $k - 1$ replacing k, we have $x_i < \eta_{k-1\,i} < x_{i+k-1}$ and $x_{i+1} < \eta_{k-1\,i+1} < x_{i+k}$, so $x_i < \eta_{ki} < x_{i+k}$ follows.

[43] Because $E(x) = f(x) - P(x)$.

[44] Using the term Lagrange here is somewhat of a misnomer; the Newton interpolation has the same error, since the Newton interpolation polynomial is the same as the Lagrange interpolation polynomial, written in a different way. The name Lagrange was used only because this error formula was derived on account of a discussion of the Lagrange interpolation formula. The same formula could have been derived on account of the Newton interpolation formula.

[45] $E^{(i)}(x)$ may have more zeros than those guaranteed by Rolle's theorem; simply disregard the extra zeros, since their presence is not assured. For example, if $E'(x)$ has more than one zero between x_1 and x_2, disregard all but one.

19. NUMERICAL DIFFERENTIATION OF FUNCTIONS

Given a function f that is differentiable sufficiently many times, for given x and $h > 0$ we have according to Taylor's formula (with the Lagrange remainder term)

$$f(x+h) = \sum_{k=0}^{m} f^{(k)}(x)\frac{h^k}{k!} + f^{(m+1)}(\xi_1)\frac{h^{m+1}}{(m+1)!}$$

and

$$f(x-h) = \sum_{k=0}^{m} f^{(k)}(x)\frac{(-h)^k}{k!} + f^{(m+1)}(\xi_1)\frac{(-h)^{m+1}}{(m+1)!},$$

where ξ_1 is between x and $x+h$ and ξ_2 is between x and $x-h$, $m \geq 0$ and integer. Assume $m > 0$ is odd, say $m = 2n+1$, and subtract these formulas. Since every term for even k will cancel, we obtain

$$f(x+h) - f(x-h) = 2\sum_{k=0}^{n} f^{(2k+1)}(x)\frac{h^{2k+1}}{(2k+1)!} + \big(f^{(2n+2)}(\xi_1) - f^{(2n+2)}(\xi_2)\big)\frac{h^{2n+2}}{(2n+2)!}.$$

We will consider this formula for a fixed value of x, but the value of h will change. Divide both sides by $2h$, and rearrange the equation so as to isolate $f'(x)$ on the left hand side. Writing[46]

$$-c_k = \frac{f^{(2k+1)}(x)}{(2k+1)!} \quad \text{and} \quad -\big(f^{(2n+2)}(\xi_1) - f^{(2n+2)}(\xi_2)\big)\frac{h^{2n+1}}{(2n+2)!} = O(h^{2n+1}),$$

we obtain

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x-h)}{2h} + \sum_{k=0}^{n} c_k h^{2k} + O(h^{2n+1}) \\ &= \frac{f(x+h) - f(x-h)}{2h} + c_1 h^2 + c_2 h^4 + \ldots + c_n h^{2n} + O(h^{2n+1}). \end{aligned} \tag{1}$$

The assumption needed to make this formula valid is that x be differentiable $2n+2$ times near[47] x (n is a nonnegative integer here).

When using this formula to estimate $f'(x)$, one would like to choose h as small as possible. The limitation in choosing h too small is the loss of precision when calculating $f(x+h) - f(x-h)$. Assuming the absolute value of the error in calculation $f(t)$ for t near x is δ, and the error of formula (1) is about $|c_1|h^2$, the absolute value of the total error in calculating $f'(x)$ will be

$$\lessapprox \frac{\delta}{h} + |c_1|h^2,$$

where the first term is the error resulting from the numerical evaluation

$$\frac{f(x+h) - f(x-h)}{2h}.$$

[46] In a notation introduced by Edmund Landau in the early 20th century, $O(f(x))$ denotes a function $g(x)$ such that the ratio $g(x)/f(x)$ stays bounded (for certain values of x understood from the context), and $o(f(x))$ denotes a function $g(x)$ that tends to zero (when x tends to a limit understood from the context; usually when x tends to infinity or x tends to zero). When using the notation $O(f(x))$ or $o(f(x))$, one usually assumes that $f(x)$ is positive in the region considered. In the present case, we use the notation $O(h^{2n+1})$ for h close to zero or when $h \to 0$.

[47] In an open interval containing $x-h$ and $x+h$, say. This requirement can be weakened somewhat.

The above expression for the error assumes its minimum assumes its minimum when its derivative with respect to h is 0, that is when

$$-\frac{\delta}{h^2} + 2|c_1|h = 0,$$

i.e., when

$$h = \sqrt[3]{\frac{\delta}{2|c_1|}}.$$

the error in this case is

$$\lessapprox \delta \sqrt[3]{\frac{2|c_1|}{\delta}} + |c_1| \left(\frac{\delta}{2|c_1|}\right)^{2/3} = \frac{3}{2^{2/3}}|c_1|^{1/3}\delta^{2/3}.$$

While c_1 here is not usually known (though it may be possible to estimate it), this should make it clear that there are limitations to how small h can be chosen in a practical calculation. If no estimate for c_1 is available, it is not unreasonable to take $c_1 = 1$ in the above formula to find the optimal value of h.

Richardson extrapolation. For fixed x, write

$$F_1(h) = \frac{f(x+h) - f(x-h)}{2h}.$$

Then (1) above can be written as

$$f'(x) = F_1(h) + c_1h^2 + c_2h^4 + \ldots + c_nh^{2n} + O(h^{2n+1}).$$

The same formula with $2h$ replacing h says

$$f'(x) = F_1(2h) + 4c_1h^2 + 16c_2h^4 + \ldots + 2^{2n}c_nh^{2n} + O(h^{2n+1}).$$

If we take four times the first of these formulas and subtract the second one, the term corresponding to h^2 will drop out:

$$3f'(x) = 4F_1(h) - F_1(2h) - 12c_2h^4 + \ldots - (2^{2n} - 4)c_nh^{2n} + O(h^{2n+1}).$$

That is, writing

$$F_2(h) = \frac{4F_1(h) - F_1(2h)}{3} \quad \text{and} \quad c_{2,k} = \frac{4 - 2^{2k}}{3} \cdot c_k,$$

we have

$$f'(x) = F_2(h) + c_{2,2}h^4 + c_{2,3}h^6 + \ldots + c_{2,n}h^{2n} + O(h^{2n+1}).$$

This process can be continued. If we write this equation with $2h$ instead of h, we obtain

$$f'(x) = F_2(2h) + 16c_{2,2}h^4 + 64c_{2,3}h^6 + \ldots + 2^{2n}c_{2,n}h^{2n} + O(h^{2n+1}).$$

Multiplying the first equation by 16, subtracting the second equation, and dividing by 15, we obtain

$$f'(x) = F_3(h) + c_{3,3}h^6 + \ldots + c_{3,n}h^{2n} + O(h^{2n+1}),$$

where

$$F_3(h) = \frac{16F_2(h) - F_2(2h)}{15} \quad \text{and} \quad c_{2,k} = \frac{16 - 2^{2k}}{15} \cdot c_k,$$

This procedure can be continued along for a while, but eventually the procedure may become unstable, and continuing the procedure further would produce less accurate results.

Higher derivatives. According to the Lemma in the section on Hermite interpolation, we showed that

$$f[x_0, x_1, \ldots, x_n] = \frac{1}{n!} f^{(n)}(\xi)$$

for some ξ in the interval spanned by the numbers $x_0, \ldots, x_n$, provided f is differentiable n times in this interval. This formula can be used to estimate the nth derivative of a function:

$$f^{(n)}(x) \approx n! f[x_0, x_1, \ldots, x_n],$$

where one usually takes the points $x_0, \ldots, x_n$ to be equidistant, i.e., $x_k = x_0 + kh$ for some $h > 0$ ($0 \le k \le n$, and x is taken to be the midpoint of the interval (x_0, x_n), i.e., $x = x_0 + nh/2$. Using the forward difference operator $\Delta f(t) = f(t+h) - f(h)$ and the forward shift operator $Ef(x) = f(x+h)$, this formula can be written as

$$f^{(n)}(x) \approx \frac{\Delta^n f(x_0)}{h^n} = \frac{\Delta^n E^{-n/2} f(x)}{h^n}$$

according to formula (1) in Section 10 on Newton interpolation with equidistant points.[48] Taylor's formula can be used to estimate the error of this formula. For example, for $n = 1$ this formula gives

$$f'(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h},$$

which is the same formula we had above with $h/2$ replacing h. With $n = 2$, the same formula gives

$$f''(x) \approx \frac{f(x + h) - 2f(x) + f(x - h)}{h^2}.$$

Using Taylor's formula for $f(x + h)$ and $f(x - h)$ involving the first two derivatives and using the remainder term with the third derivative (i.e., using the Taylor expansions given at the beginning of this section with $m = 2$), we obtain

$$f''(x) = \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} + \frac{h}{6}(f''(\xi_2) - f''(\xi_1))$$

with some $\xi_1 \in (x - h, x)$ and $\xi_2 \in (x, x + h)$. If higher order derivatives also exist, one can again use Richardson extrapolation to obtain a more accurate result without having to use a smaller value of h.

Problems

1. Consider the formula

$$\bar{f}(x, h) = \frac{f(x + h) - f(x - h)}{2h}$$

to approximate the derivative of a function f. Assume we are able to evaluate f with about 5 decimal precision. Assume, further, that $f'''(x) \approx 1$. What is the best value of h to approximate the derivative?

Solution. We have

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} - \frac{f'''(x)}{3!} h^2 + O(h^3).$$

[48] The second equation uses the observation that $x = x_0 - nh/2$, and so $f(x_0) = E^{-n/2} f(x)$.

We are able to evaluate $f(x)$ with 5 decimal precision, i.e., with an error of $5 \cdot 10^{-6}$. Thus, the (absolute value of the maximum) error in evaluating $\frac{f(x+h)-f(x-h)}{2h}$ is $5 \cdot 10^{-6}/h$. So the absolute value of the total error (roundoff error plus truncation error) in evaluating $f'(x)$ is

$$\frac{5 \cdot 10^{-6}}{h} + \frac{|f'''(x)|}{6}h^2 \approx \frac{5 \cdot 10^{-6}}{h} + \frac{h^2}{6},$$

as $f'''(x) \approx 1$. The derivative of the right-hand side with respect to h is

$$-\frac{5 \cdot 10^{-6}}{h^2} + \frac{h}{3}.$$

Equating this with 0 gives the place of minimum error when $h^3 = 15 \cdot 10^{-6}$, i.e., $h \approx 0.0246$.

2. Consider the formula

$$\bar{f}(x,h) = \frac{f(x+h) - f(x-h)}{2h}$$

to approximate the derivative of a function f. Assume we are able to evaluate f with about 6 decimal precision. Assume, further, that $f'''(x) \approx 1$. What is the best value of h to approximate the derivative?

Solution. We have

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{f'''(x)}{3!}h^2 + O(h^3).$$

We are able to evaluate $f(x)$ with 6 decimal precision, i.e., with an error of $5 \cdot 10^{-7}$. Thus, the (absolute value of the maximum) error in evaluating $\frac{f(x+h)-f(x-h)}{2h}$ is $5 \cdot 10^{-7}/h$. So the absolute value of the total error (roundoff error plus truncation error) in evaluating $f'(x)$ is

$$\frac{5 \cdot 10^{-7}}{h} + \frac{|f'''(x)|}{6}h^2 \approx \frac{5 \cdot 10^{-7}}{h} + \frac{h^2}{6},$$

as $f'''(x) \approx 1$. The derivative of the right-hand side with respect to h is

$$-\frac{5 \cdot 10^{-7}}{h^2} + \frac{h}{3}.$$

Equating this with 0 gives the place of minimum error when $h^3 = 15 \cdot 10^{-7}$, i.e., $h \approx 0.011,447$.

3. Given a certain function f, we are using the formula

$$\bar{f}(x,h) = \frac{f(x+h) - f(x-h)}{2h}$$

to approximate its derivative. We have

$$\bar{f}(1, 0.1) = 5.135,466,136 \quad \text{and} \quad \bar{f}(1, 0.2) = 5.657,177,752$$

Using Richardson extrapolation, find a better approximation for $f'(1)$.

Solution. We have

$$f'(x) = \bar{f}(x,h) + c_1h^2 + c_2h^4 \dots$$
$$f'(x) = \bar{f}(x,2h) + c_1(2h)^2 + c_2(2h)^4 \dots$$

with some c_1, c_2, ... Multiplying the first equation by 4 and subtracting the second one, we obtain

$$3f'(x) = 4\bar{f}(x,h) - \bar{f}(x,2h) - 12c_2h^4 - \dots .$$

That is, with $h = 0.1$ we have

$$f'(x) \approx \frac{4\bar{f}(x,h) - \bar{f}(x,2h)}{3} \approx \frac{4 \cdot 5.135,466,136 - 5.657,177,752}{3} = 4.961,56$$

The function in the example is $f(x) = x \tan x$ and $f'(1) = 4.982,93$.

4. Given a certain function f, we are using the formula

$$\bar{f}(x,h) = \frac{f(x+h) - f(x-h)}{2h}$$

to approximate its derivative. We have

$$\bar{f}(2, 0.125) = 12.015625 \quad \text{and} \quad \bar{f}(2, 0.25) = 12.062500$$

Using Richardson extrapolation, find a better approximation for $f'(2)$.

Solution. We have

$$f'(x) = \bar{f}(x,h) + c_1h^2 + c_2h^4 \dots$$
$$f'(x) = \bar{f}(x,2h) + c_1(2h)^2 + c_2(2h)^4 \dots$$

with some c_1, c_2, ... Multiplying the first equation by 4 and subtracting the second one, we obtain

$$3f'(x) = 4\bar{f}(x,2h) - f(x,h) + 12c_2h^4 + \dots .$$

That is, with $x = 2$ and $h = 0.125$ we have

$$f'(x) \approx \frac{4\bar{f}(x,h) - \bar{f}(x,2h)}{3} \approx \frac{4 \cdot 12.015,625 - 12.062,500}{3} = 12.000,000$$

The function in the example is $f(x) = x^3$ and $f'(2) = 12$ is exact.[49]

20. SIMPLE NUMERICAL INTEGRATION FORMULAS

If we approximate a function f on an interval $[a, b]$ by an interpolation polynomial P, we can integrate the formula

$$f(x) = P(x) + E(x)$$

to approximate the integral of f. The polynomial P is easy to integrate. Often one can get reasonable estimates of the integral of the error E. We consider two simple rules: the trapezoidal rule, and Simpson's rule. When considering these rules, the function f will be assumed to be Riemann-integrable on $[a, b]$. Hence the error term $E(x) = f(x) - P(x)$ will also be Riemann-integrable, even though this might not be apparent from the formula we may obtain for $E(x)$.

[49] It is natural that one step of Richardson extrapolation gives the exact result for a cubic polynomial, after all all coefficients c_2, c_3, ... in the Taylor expansion of $f'(x)$ above are zero. So, if we eliminate the coefficient c_1, we must get an exact result.

The trapezoidal rule. Approximate f on the interval $[0, h]$ by the linear polynomial P with interpolating at the points 0 and h. Using the Newton interpolation formula, we have $f(x) = P(x) + E(x)$, where

$$P(x) = f[0] + f[0, h]x,$$

and, Assuming that f is continuous on $[0, h]$ and is twice times differentiable in the interval $(0, h)$, for the error term, we have

$$E(x) = f[x, 0, h]x(x - h) = \frac{1}{2}f''(\xi_x)x(x - h),$$

where $\xi_x \in (0, h)$ depends on x, according to Lemma 1 in the section on Hermite interpolation (Section 7). Integrating these on the interval $[0, h]$, we obtain

$$\int_0^h P(x)\,dx = \int_0^h (f[0]+f[0, h]x)\,dx = f[0]h+f[0, h]\frac{h^2}{2} = f(0)h+\frac{f(h) - f(0)}{h}\cdot\frac{h^2}{2} = \frac{h}{2}(f(0)+f(h)).$$

For the error, we need a simple Mean-Value Theorem for integrals that we will now describe. Given integrable functions ϕ and ψ on the interval $[a, b]$ such that

$$m < \phi(x) < M \quad \text{and} \quad \psi(x) \geq 0 \qquad \text{for } x \in (a, b),$$

we have

$$\int_a^b \phi(x)\psi(x)\,dx = H\int_a^b \psi(x)\,dx$$

for some number H with $m \leq H \leq M$. In fact, one can show that $m < H < M$.[50] In case of the usual form of error terms, this equality can be further simplified. Namely, if $\phi(x) = g'(\xi_x)$ for some function g that is differentiable on (a, b) and where $\xi_x \in (a, b)$ depends on x, we have

$$\int_a^b g'(\xi_x)\psi(x)\,dx = g'(\eta)\int_a^b \psi(x)\,dx \tag{1}$$

for some $\eta \in (a, b)$, provided that the integrals on both sides of this equation exist. This is because the derivative of a function has the intermediate-value property, that is, if $g'(\alpha) < c < g'(\beta)$ or $g'(\alpha) > c > g'(\beta)$ for some $\alpha, \beta \in (a, b)$ with $\alpha < \beta$, we have $g'(\xi) = c$ for some ξ with $\alpha < \xi < \beta$; this is true even if g' is not continuous. This means that the set

$$\{g'(x) : x \in (a, b)\}$$

[50]The value of $\phi(x)$ and $\psi(x)$ at a or b clearly have no influence on the value of the integrals, so it was sufficient to impose conditions on these values only for $x \in (a, b)$. If $\int_a^b \psi = 0$, then the integrals on both sides of the last equation are zero, so the value of H makes no difference. (This is difficult to explain without a more thorough knowledge of integration theory. Noting that we assumed that $\psi(x) \geq 0$, if one uses Lebesgue integration theory, then then assumption together with the equation $\int_a^b \psi = 0$ implies that $\psi = 0$ almost everywhere on $[a, b]$, and so $\int_a^b \phi(x)\psi(x)\,dx = 0$ for any ϕ. The explanation for Riemann integration theory is somewhat more complicated, but the statement is still true). If $\int_a^b \psi > 0$ then one can take

$$H = \frac{\int_a^b \phi(x)\psi(x)\,dx}{\int_a^b \psi(x)\,dx}.$$

Then $m \leq H \leq M$ can easily be shown. It can also be shows, however, that that $H = m$ or $H = M$ cannot happen.

is an interval (open, closed, or semi-closed). Hence the number H above with the choice of $\phi(x) = g'(\xi_x)$ can always be written as $g'(\eta)$ for some $\eta \in (a,b)$.[51] It is clear that (1) is also valid if we assume $\psi(x) \le 0$ for all $x \in (a,b)$ instead of assuming $\psi(x) \ge 0$.[52]

Using this Mean-Value Theorem, the integral of the error term can be written as

$$\int_0^h E(x)\,dx = \frac{1}{2}\int_0^h f''(\xi_x)x(x-h)\,dx = \frac{1}{2}f''(\eta)\int_0^h (x^2-hx)\,dx$$
$$= \frac{1}{2}f''(\eta)\left(\frac{h^3}{3} - h\cdot\frac{h^2}{2}\right) = -\frac{h^3}{12}f''(\eta)$$

With a simple change of variable $t = a + (b-a)x$ with $h = b-a$, we obtain

$$\int_a^b f = \frac{b-a}{2}(f(a)+f(b)) - \frac{(b-a)^3}{12}f''(\eta)$$

with some $\eta \in (a,b)$, provided f is continuous in $[a,b]$ and twice differentiable in (a,b).

Simpson's rule. In Simpson's rule, a function is interpolated by a quadratic polynomial at three points, $-h$, 0, and h. To get a better and simpler error estimate, we interpolate at the points $-h$, 0, 0, and h by a cubic polynomial, but the cubic part of the interpolation polynomial will make no contribution to the integral. The Newton-Hermite interpolation polynomial at the points 0, $-h$, h, 0 can be written as

$$f(x) = P(x) + E(x) = f[0] + f[0,-h]x + f[0,-h,h]x(x+h) + f[0,-h,h,0]x(x+h)(x-h)$$
$$+ f[x,0,-h,h,0]x^2(x+h)(x-h),$$

where the last term on the right-hand side is the error term. The main points of this equation is that the integral on the interval $[-h,h]$ of the fourth (cubic) term on the right-hand side is zero, while the polynomial in the error term (the fifth and last term) is ≤ 0 in the interval $(-h,h)$, so we can use the Mean-Value Theorem (1) in the error estimate. Thus, for the main term we have

$$\int_{-h}^h P(x)\,dx = \int_{-h}^h (f[0] + f[0,-h]x + f[0,-h,h](x^2+hx) + f[0,-h,h,0]x(x^2-h))\,dx$$
$$= f[0]\cdot 2h + f[0,-h,h]\cdot\frac{2h^3}{3} = \frac{h}{3}(f(-h)+4f(0)+f(h)).$$

Assuming that f is continuous on $[-h,h]$ and is four times differentiable in the interval $(-h,h)$, for the error term, we have

$$E(x) = f[x,0,-h,h,0]x^2(x+h)(x-h) = \frac{1}{4!}f^{(4)}(\xi_x)x^2(x^2-h^2)$$

for some $\xi_x \in (-h,h)$ depending on x, according to Lemma 1 in the section on Hermite interpolation (Section 7).[53] As the polynomial multiplying the (fourth) derivative on the right-hand side is nonpositive for $x \in (-h,h)$, we can use (1) to estimate the integral of the error:

$$\int_{-h}^h E(x)\,dx = \frac{1}{24}\int_{-h}^h f^{(4)}(\xi_x)(x^4-h^2x^2)\,dx = \frac{f^{(4)}(\eta)}{24}\int_{-h}^h (x^4-h^2x^2)\,dx$$
$$= \frac{f^{(4)}(\eta)}{24}\left(\frac{2h^5}{5} - h^2\cdot\frac{2h^3}{3}\right) = -\frac{h^5}{90}f^{(4)}(\eta)$$

[51] Above, we assumed that the function $\phi(x)$ is bounded. Here, $g'(\xi_x)$ replaces $\phi(x)$, and g' need not be bounded. However, if g' is bounded neither from above nor from below on (a,b) then (1) does not say anything useful, since $g'(\eta)$ can be any real number (because g' has the intermediate value property). If g' is bounded from above or from below, then the useful half of the argument above is still applicable so as to establish (1).

[52] To show this, use (1) with $-\psi$ replacing ψ.

[53] See the comment at the end of Section 7 (before the Problems) saying that the Lemma can be extended to the case of repeated interpolation points.

with some $\eta \in (-h, h)$. Writing $h = (b-a)/2$, with the change of variable $t = a+h+2hx$, we obtain

$$\int_a^b f = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right) - \frac{(b-a)^5}{2^5 \cdot 90} f^{(4)}(\eta)$$

with some $\eta \in (a, b)$, provided f is continuous in $[a, b]$ and five times differentiable in (a, b). Here $2^5 \cdot 90 = 2880$.

The above discussion assumed that the fourth derivative of f exists. If we assume that f has only three derivatives on $(-h, h)$, we to use the The Newton interpolation polynomial at the points 0, $-h$, h:

$$f(x) = P(x) + E(x) = f[0] + f[0, -h]x + f[0, -h, h]x(x+h) + f[x, -h, h, 0]x(x+h)(x-h),$$

and the error term here is

$$E(x) = f[x, -h, h, 0]x(x+h)(x-h) = \frac{1}{3!} f'''(\xi_x)x(x^2 - h)$$

for some $\xi_x \in (-h, h)$ depending on x. The problem here is that $x(x^2 - h)$ changes sign on $(-h, h)$, hence formula (1) above is not directly applicable. We need to apply formula (1) separately on the interval $[0, h]$ and $[-h, 0]$:

$$\begin{aligned}\int_0^h E(x)\,dx &= \frac{1}{6}\int_0^h f'''(\xi_x)(x^3 - h^2 x)\,dx = \frac{f'''(\eta_1)}{6}\int_0^h (x^3 - h^2 x)\,dx \\ &= \frac{f'''(\eta_1)}{6}\left(\frac{h^4}{4} - h^2 \cdot \frac{h^2}{2}\right) = -\frac{h^4}{24} f'''(\eta_1)\end{aligned}$$

with some $\eta_1 \in (-h, h)$.[54] Similarly, we have

$$\begin{aligned}\int_{-h}^0 E(x)\,dx &= \frac{1}{6}\int_{-h}^0 f'''(\xi_x)(x^3 - h^2 x)\,dx = \frac{f'''(\eta_2)}{6}\int_{-h}^0 (x^3 - h^2 x)\,dx \\ &= \frac{f'''(\eta_2)}{6}\left(-\frac{h^4}{4} + h^2 \cdot \frac{h^2}{2}\right) = \frac{h^4}{24} f'''(\eta_2)\end{aligned}$$

with some $\eta_2 \in (-h, h)$. So we obtain that

$$\int_{-h}^h E(x)\,dx = \frac{h^4}{24}\big(f'''(\eta_2) - f'''(\eta_1)\big),$$

for some $\eta_1, \eta_2 \in (-h, h)$.

21. COMPOSITE NUMERICAL INTEGRATION FORMULAS

In composite integration formulas, one divides the interval $[a, b]$ into equal parts, and on each part uses the same numerical integration formula.

Composite trapezoidal rule. Let f be a continuous function on the interval $[a, b]$, and assume that f is twice differentiable on (a, b). Let n be a positive integer, put $h = \frac{b-a}{n}$, and let $x_i = a + hi$ for i with $0 \le i \le n$. We use the trapezoidal rule on each of the intervals $[x_{i-1}, x_i]$ $(1 \le i \le n)$, and add the results. The calculation involving the main term is straightforward; as for the sum of the error terms, we have

$$-\frac{h^3}{12}\sum_{i=1}^n f''(\eta_i),$$

[54] Formula (1) above is not applicable literally, since we only have $\xi_x \in (-h, h)$ and so ξ_x need not belong to the interval of integration $(0, h)$. It is easy to show, however, that (1) can be generalized to the case where one assumes that ξ_x belongs to another interval (c, d), and not the interval (a, b) of integration.

where $\eta_i \in (x_{i-1}, x_i)$ for each i. Since the derivative of any function satisfies the intermediate-value property, the sum equals $nf''(\xi)$ for some $\xi \in (a, b)$. Thus, the error term can be written as

$$-\frac{(b-a)^3}{12n^2} f''(\xi).$$

Hence, we have

$$\int_a^b f = \frac{b-a}{2n}\Big(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\Big) - \frac{(b-a)^3}{12n^2} f''(\xi).$$

The error committed when using this formula to calculate an integral results from the error term and the accumulation of the errors committed in calculating each term on the right-hand side. To estimate this error, let δ be the maximum error in calculating the value of f. Then assuming the errors in evaluating function values are independent of one another, the error resulting of adding n function values can be considered a random variable having a normal distribution of mean 0 and standard deviation $\delta\sqrt{n/6}$.[55] This error will be less than $2.58\delta\sqrt{n/6}$ with 99% probability. Thus, the error in calculating the right-hand side of the above formula is about[56]

$$\frac{b-a}{2n} \cdot 2 \cdot 2.58\delta\sqrt{n/6} \approx 1.053\delta \cdot \frac{b-a}{\sqrt{n}} \approx \delta \cdot \frac{b-a}{\sqrt{n}}.$$

Noting that the truncation error (i.e., the error given by the error term) decreases in proportion of $1/n^2$ while the round-off error decreases in proportion of $1/\sqrt{n}$ (i.e., much more slowly), once the size of the roundoff error is about equal to the size of the truncation error, there is relatively little payoff in increasing the value of n.

Composite Simpson's rule. Let f be a continuous function on the interval $[a, b]$, and assume that f is four times differentiable on (a, b). Let n be a positive integer, and put $h = \frac{b-a}{2n}$; that is, we divide the interval $[a, b]$ in $2n$ equal parts. Let $x_i = a + hi$ for i with $0 \le i \le 2n$. Using Simpson's rule on each of the intervals $[x_{2k-1}, x_{2k}]$ for k with $1 \le k \le n$, and adding the results, we obtain

$$\int_a^b f = \frac{b-a}{6n}\Big(f(a) + 4f(x_1) + \sum_{k=1}^{n-1}\big(2f(x_{2k}) + 4f(x_{2k+1})\big) + f(b)\Big) - \frac{(b-a)^5}{2880n^4} f^{(4)}(\xi)$$

for some $\xi \in (a, b)$. The error calculation used the fact that $f^{(4)}$ satisfies the mean-value property, in a way similar to the error calculation in the composite trapezoidal rule.

As an example, Simpson's rule is used to calculate the integral

$$\int_0^4 \frac{dx}{\sqrt{1+x^3}}.$$

The function is defined in the file `funct.c`:

---

[55] The calculation on pp. 10–11 in A. Ralston–P. Rabinowitz [RR] concerns the case when $\delta = 1/2$.

[56] This calculation of the error assumes that n terms are added on the right-hand side of the above integration formula, and this sum is multiplied by 2. The number of terms actually added is $n + 1$, and not all of them are multiplied by 2.

```
 1 #include <math.h>
 2
 3 double funct(x)
 4   double x;
 5 {
 6   double value;
 7   value = 1.0/sqrt(1.0+x*x*x);
 8   return(value);
 9 }
```

The calling program and Simpson's rule are contained in the file `main.c`:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 double funct(double);
 5 double simpson(double (*fnct)(double), double lower,
 6     double upper, int n);
 7
 8 main()
 9 {
10   double lower = 0.0, upper = 4.0, x, integr;
11   int i, n;
12   printf("Type n: ");
13   scanf("%d", &n);
14   integr = simpson(&funct, lower,
15     upper,n);
16   printf("The integral is %.12f\n", integr);
17 }
18
19 double simpson(double (*fnct)(double), double lower,
20     double upper, int n)
21 {
22   double x,
23     h, integr = 0.0,
24     flower, fupper,middle,fmiddle;
25   int i;
26   h = (upper - lower)/ 2.0 / ((double) n);
27   x = lower;
28   integr = (*fnct)(x);
29   for(i=0; i < n; i++)
30   {
31      x += h;
32      integr += 4 * (*fnct)(x);
33      x += h;
34      integr += 2 * (*fnct)(x);
35   }
36   integr -= (*fnct)(upper);
37   return integr *= h / 3.0;
38 }
```

Lines 19–38 contain the function implementing Simpson's rule. It's prototype is given in lines 19–20. The parameters are a pointer to `(*fnct)`, the function to be integrated, the lower limit `lower`, the

upper limit `upper`, and the number `n` of the pairs of intervals used (i.e., the interval (a, b) is to be divided into $2n$ parts). The calculation is fairly straightforward; the only point to be mentioned that in lines 28, 32, and 34, the function is referred to as `(*fnct)`, the location pointed to by the pointer `fnct`. In the calling program, given in lines 8–17, the number `n` is entered by the user on line 13, so that the user can experiment with different values of n entered. For example, with $n = 40$, the printout is

```
1 Type n: 40
2 The integral is 1.805473405017
```

Here, the number 40 on line 1 is typed by the user. With $n = 200$, the printout is

```
1 Type n: 200
2 The integral is 1.805473301824
```

The actual value of the integral is $1.805, 473, 301, 658$.

Problems

1. We want to evaluate

$$\int_0^1 e^{-x^2}\, dx$$

using the composite trapezoidal rule with four decimal precision, i.e., with the absolute value of the error not exceeding $5 \cdot 10^{-5}$. What value of n should one use when dividing the interval $[0, 1]$ into n parts?

Solution. The error term in the composite trapezoidal formula when integrating f on the interval $[a, b]$ and dividing the interval into n parts is

$$-\frac{(b-a)^3}{12n^2} f''(\xi)$$

for some $\xi \in (a, b)$. We want to use this with $a = 0$, $b = 1$, and $f(x) = e^{-x^2}$. We have

$$f''(x) = (4x^2 - 2)e^{-x^2}.$$

We want to find the maximum of the absolute value of this in the interval $[0, 1]$. For the third derivative we have

$$f'''(x) = (12x - 8x^2)e^{-x^2} = 4x(3 - 2x^2)e^{-x^2} > 0$$

for $x \in (0, 1)$. Hence $f''(x)$ is increasing on $[0, 1]$. Thus

$$-2 = f''(0) < f''(x) < f''(1) = 2e^{-1} \approx 0.735, 759$$

for $x \in (0, 1)$. So, noting that $a = 0$ and $b = 1$, the absolute value of the error is

$$\frac{(b-a)^3}{12n^2}|f''(\xi)| = \frac{1}{12n^2}|f''(\xi)| \le \frac{1}{12n^2} \cdot 2 = \frac{1}{6n^2}.$$

In order to ensure that this error is less than $5 \cdot 10^{-5}$, we need to have $1/(6n^2) < 5 \cdot 10^{-5}$, i.e.,

$$n > \sqrt{\frac{10^5}{30}} \approx 57.735.$$

So one needs to make sure that $n \ge 58$. Thus one needs to divide the interval $[0, 1]$ into (at least) 58 parts in order to get the result with 4 decimal precision while using the trapezoidal rule. It is

interesting to compare this with the Simpson rule, which gives the result with 4 decimal precision if one divides the interval into (at least) 8 parts.

2. We want to evaluate

$$\int_0^1 e^{-x^2}\,dx$$

using the composite Simpson rule with four decimal precision. What value of n should one use when dividing the interval $[0,1]$ into $2n$ parts?

Solution. The error term in the composite Simpson formula when integrating f on the interval $[a,b]$ and dividing the interval into $2n$ parts is

$$-\frac{(b-a)^5}{2880n^4}f^{(4)}(\xi)$$

for some $\xi \in (a,b)$. We want to use this with $a=0$, $b=1$, and $f(x)=e^{-x^2}$. We have

$$f^{(4)}(x) = (16x^4 - 48x^2 + 12)e^{-x^2}.$$

We want to find the maximum the absolute value of this in the interval $[0,1]$. For the fifth derivative we have

$$f^{(5)}(x) = -(32x^5 - 160x^3 + 120x)e^{-x^2}.$$

One can easily find the zeros of this, since solving this equation amount to solving a quadratic equation. We find that the zeros are $x \approx \pm 0.958572$ and $x \approx \pm 2.02018$ and $x = 0$. Only one of these is in the interval $(0,1)$: $x \approx 0.958572$. The sixth derivative is

$$f^{(6)}(x) = (64x^6 - 480x^4 + 720x^2 - 120)e^{-x^2},$$

and for $x \approx 0.958572$ this is approximately 74.1949, that is, it is positive. Therefore, the fourth derivative has a local minimum at $x \approx 0.958572$. Since the fifth derivative is not zero anywhere else in the interval $(0,1)$, this is a place of an absolute minimum of the fourth derivative in $[0,1]$. The fourth derivative at $x \approx 0.958572$ is approximately -7.41948. At $x=0$ it is 12, and at $x=1$ is it is approximately -7.35759. Thus the absolute maximum of the fourth derivative on the interval $[0,1]$ is 12 (assumed at $x=0$). Using this in the above error formula, we find that the absolute value of the error is at most

$$\frac{1}{2880n^4}\cdot 12.$$

In order to achieve four decimal precision, this error should be less than $5\cdot 10^{-5}$, that is

$$\frac{1}{2880n^4}\cdot 12 < 5\cdot 10^{-5}.$$

Therefore, we must have

$$n^4 > \frac{12\cdot 10^5}{5\cdot 2880} = \frac{250}{3} \approx 83.3$$

This is satisfied with $n=4$ ($n=3$ comes close, since $3^4 = 81$). That is, when we use the composite Simpson rule with $n=4$ (i.e., when we divide the interval into eight parts), we will obtain the the result with four decimal precision.

22. ADAPTIVE INTEGRATION

When numerically approximating the integral of a function, one usually divides the interval of integration into parts, and on each part one uses simple integration formulas. Often dividing the interval into equal parts is not the best thing to do, since on certain parts of the interval of integration the function may be better behaved, requiring a courser subdivision, than in other parts, requiring a finer subdivision. This is the problem adaptive integration intends to deal with. Adaptive integration can be used with various simple integration rules, such as the trapezoidal rule and Simpson's rule. In the following discussion we consider it with the trapezoidal rule.

Suppose we want to determine the integral of a function f on an interval I with a maximum absolute error of ϵ. Assume the trapezoidal rule on this interval approximates the integral as $T(I)$. Then we divide I into two equal parts, I_0 and I_1, and consider the trapezoidal rule approximations $T(I_0)$ and $T(I_1)$ on each of these parts. Since $T(I)$ and $T(I_0) + T(I_1)$ both approximate $\int_I f$, we have

$$T(I) \approx T(I_0) + T(I_1).$$

This equation will, however, not be exact; one would expect that the right-hand side gives a better approximation, using a finer subdivision of the integral. In fact, one can use the quantity

$$T(I) - \big(T(I_0) + T(I_1)\big).$$

to estimate the accuracy of this approximation. We will consider it in more detail how this is done. Writing $|I|$ for the length of the interval I, for the errors of these approximations, we have

$$E(I) = -\frac{|I|^3}{12} f''(\xi_I),$$

where $\xi_I \in I$, and, noting that $|I_0| = |I_1| = |I|/2$, we have

$$E(I_0) = -\frac{|I|^3}{8 \cdot 12} f''(\xi_{I_0}), \quad E(I_0) = -\frac{|I|^3}{8 \cdot 12} f''(\xi_{I_0}),$$

where $\xi_{I_0} \in I_0$ and $\xi_{I_1} \in I_1$. Making the assumption that

$$f''(\xi) \approx f''(\xi_{I_0}) \approx f''(\xi_{I_1}), \tag{1}$$

it follows that

$$E(I_0) \approx E(I_1) \approx \frac{E(I)}{8}.$$

Hence, noting that

$$\int_I f = T(I) + E(I) = T(I_0) + T(I_1) + E(I_0) + E(I_1),$$

we have

$$T(I_0) + T(I_1) - T(I) = E(I) - E(I_0) - E(I_1) \approx (8 - 1 - 1) \cdot E(I_0) = 6E(I_0).$$

Hence

$$E(I_0) \approx E(I_1) \approx \frac{1}{6}(T(I_0) + T(I_1) - T(I)).$$

Thus the error in approximating $\int_I f$ by $T(I_0) + T(I_1)$ is

$$E(I_0) + E(I_1) \approx \frac{1}{3}(T(I_0) + T(I_1) - T(I)).$$

Thus, if

$$\frac{1}{3}|T(I_0) + T(I_1) - T(I)| < \epsilon. \tag{2}$$

then we accept the approximation

$$\int_I f = T(I_0) + T(I_1).$$

If this inequality is not satisfied, then we work with the intervals I_0 and I_1 separately. Noting that

$$\int_I f = \int_{I_0} f + \int_{I_1} f,$$

we try to determine the integral $\int_{I_0}$ with an allowable error less than $\epsilon/2$. Similarly, we try to determine $\int_{I_1}$ with an allowable error less than $\epsilon/2$. If we are successful, the total error committed on I will be less than ϵ.

When continuing this procedure, some parts of I may have to be subdivided many times, others maybe only a few times; in any case, the total error committed on I will be less than ϵ. Of course, there is no certainty that this will be so, since the assumption (1) may fail, especially if we did not make sure that we divided the interval into sufficiently many parts. Therefore, an additional precaution may be that (2) is only accepted as a guarantee that the error is small enough if the length of I itself is not too large.

A C program implementation of adaptive integration is contained in the file `adapt.c`. The header file `adapt.h` to this file is

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 5
 6 double funct(double);
 7 double trapadapt(double (*fnct)(double), double a,
 8     double b, double epsilon, double maxh,
 9     double minh, double fa, double fb,
10     double *bada, double *badb,
11     int *success);
```

On line 4, `absval(x)` to calculate the absolute value of `x` is defined, on line 6 the declaration of `funct`, the function to be integrated (defined in a separate file) is given, and in lines 7–11 the declaration of the function `trapadapt`, discussed next, is give. The file `adapt.c` contains the definition of this function:

```
 1 #include "adapt.h"
 2
 3 double trapadapt(double (*fnct)(double), double a,
 4     double b, double epsilon, double maxh,
 5     double minh, double fa, double fb,
 6     double *bada, double *badb,
 7     int *success)
 8 {
 9   double x, integr1, integr2, integr = 0.0,
10     mid, fmid, h, t1, t2;
11   h = b - a;
```

```
12   if ( h >= minh ) {
13     mid = (a + b)/2.0; fmid = (*fnct)(mid);
14     t1 = (fa+fb)*h/2.0;
15     t2 = (fa+2.0*fmid+fb)*h/4;
16     if ( h<=maxh && absval(t2-t1)<= 3.0*epsilon ) {
17        integr = t2;
18        *success = 1;
19     }
20     else {
21       integr1 = trapadapt(fnct, a, mid, epsilon/2.0, maxh,
22           minh, fa, fmid, bada, badb, success);
23       if ( *success ) {
24         integr2 = trapadapt(fnct, mid, b, epsilon/2.0, maxh,
25           minh, fmid, fb, bada, badb, success);
26       }
27       if ( *success ) {
28         integr = integr1+integr2;
29       }
30     }
31   }
32   else {
33     *success = 0; *bada = a; *badb = b;
34   }
35   return integr;
36 }
```

The function `trapadapt` has parameters (mentioned in lines 3–7) `(*fnct)`, a pointer to the function to be integrated, the limits of the integral `a` and `b`, the maximum allowable error `epsilon` on the interval `(a,b)`, the maximum length `max h` of a subdivision that need not be subdivided further, the minimum length `minh` of a subdivision (if the error is too large with an interval of length `minh`, the calculation will be declared a failure), the values `fa` and `fb`, of the function at `a` and `b`, respectively, pointers `*bada` and `*badb`, to the endpoints of the subinterval on which the calculation failed (the calling program can read these values, so it can be decided later to use a different method to determine the integral on the interval where the method failed), and the variable `success` that is 1 (true) if the method is successful and 0 (false) otherwise. In lines 15 and 16, the trapezoidal sums on the whole interval `(a,b)` and its two parts are calculated, and in line 16 it is decided whether the calculation is successful (when `h<=maxh` and the the absolute value of the difference `t2-t1` estimating the error is small enough). If this is not the case, the method is recursively called in lines 21–25 for the intervals `(a,mid)` and `(mid,b)` with `epsilon/2.0` instead of `epsilon` (since each of these subintervals allow only half the error of the whole interval. The sum of the two integrals on the subintervals, calculated in line 28, will be the integral on the interval `(a,b)`. If the calculation is not successful, the endpoints `a` and `b` are placed into the locations `*bada` and `badb` on line 33.

The file `funct.c` contains the definition of the function `funct`:

```
1 #include "adapt.h"
2
3 double funct(x)
4   double x;
5 {
6   double value;
7   value = 1.0/sqrt(1.0+x*x*x);
8   return(value);
```

```
9 }
```

That is, the integral of the function

$$\frac{1}{\sqrt{1+x^3}}$$

is being calculated. The main program, given in the file `main.c`, specifies the limits of the integral as $a = 0$ and $b = 4$:

```
 1 #include "adapt.h"
 2
 3 main()
 4 {
 5   double a = 0.0, b = 4.0, x, integr,
 6     epsilon=5e-10, maxh=1e-1, minh=1e-5, bada, badb,
 7     mid, fa, fb;
 8   int success;
 9   fa=funct(a); fb=funct(b);
10   integr=trapadapt(&funct, a, b, epsilon, maxh, minh,
11           fa, fb, &bada, &badb, &success);
12   if ( success ) {
13     printf("The integral is %.12f\n", integr);
14   }
15   else {
16
17     printf("The integral has trouble in the interval"
18       "(%.12f, %.12f)\n", bada, badb);
19   }
20 }
```

The makefile, called `makefile`, used to compile this program is given in

```
 1 all: adapt
 2 adapt : funct.o main.o adapt.o adapt.h
 3         gcc -o adapt -s -O4 adapt.o funct.o main.o -lm
 4 funct.o : funct.c adapt.h
 5         gcc -c -O4 funct.c
 6 adapt.o : adapt.c adapt.h
 7         gcc -c -O4 adapt.c
 8 main.o : main.c adapt.h
 9         gcc -c -O4 main.c
10 clean : adapt
11         rm funct.o main.o
```

To compile the program, one types the command

```
$ make all
```

If the program runs well, one may remove the object files `main.o`, and `funct.o` by typing

```
$ make clean
```

Once the program runs, there is no need for these files, and the file `adapt` is the only one needed. The file `funct.o` may be kept, since it can be reused if a different integral is to be calculated (for which the files `main.c` and `funct.c` may need to be rewritten, but the file `adapt.c` need not be changed. The printout of the program is a single line (the `1` at the beginning of the line is a line number, not part of the printout).

```
1 The integral is 1.805473301821
```

A more precise approximation of the integral is $1.805,473,301,658,077$; i.e, the result is accurate to nine decimal places, which is in accordance with out choice of the maximum allowable error $\epsilon = 5 \cdot 10^{-10}$ (i.e., `epsilon=5e-10`).

Problem

1. We would like to determine the integral

$$\int_0^1 \sqrt[3]{1+x^2}\, dx$$

with precision $\epsilon = 5 \cdot 10^{-4}$ using the adaptive trapezoidal rule. Writing $f(x) = \sqrt[3]{1+x^2}$, on the interval $[1/8, 2/8]$ we find that

$$T = \frac{1/8}{2}(f(1/8) + f(2/8)) \approx 0.126,599,701$$

and

$$\bar{T} = \frac{1/16}{2}(f(1/8) + 2f(3/16) + f(2/8)) = 0.126,523,853.$$

Do we have to subdivide the interval $[1/8, 2/8]$ further or the result is already precise enough.

Solution. The error allowed on the interval $[1/8, 2/8]$ is $\epsilon/8 = 6.25 \cdot 10^{-5}$. The error can be estimated as

$$\frac{1}{3}(\bar{T} - T) \approx -0.000,025,282,7$$

This is smaller than the permissible error, so the interval does not need to be further subdivided; one can accept $\bar{T}$ as the integral on this interval.

23. ADAPTIVE INTEGRATION WITH SIMPSON'S RULE

After of discussion of an implementation for adaptive integration for the trapezoidal rule, it will be easy to make the appropriate changes so as to implement adaptive integration with Simpson's rule. Suppose we want to determine the integral of a function f on an interval I with a maximum absolute error of ϵ. Assume Simpson's rule on this interval approximates the integral as $S(I)$. Then we divide I into two equal parts, I_0 and I_1, and consider the Simpson approximations $S(I_0)$ and $S(I_1)$ on each of these parts.[57] Since $S(I)$ and $S(I_0) + S(I_1)$ both approximate $\int_I f$, we have

$$S(I) \approx S(I_0) + S(I_1).$$

As in case of the trapezoidal rule, this equation will not be exact; one would expect that the right-hand side gives a better approximation, using a finer subdivision of the integral. In fact, one can use the quantity

$$S(I) - \big(S(I_0) + S(I_1)\big).$$

to estimate the accuracy of this approximation. We will consider it in more detail how this is done. Writing $|I|$ for the length of the interval I, for the errors of these approximations, we have

$$E(I) = -\frac{|I|^5}{2880} f^{(4)}(\xi_I),$$

[57] Of course, in order to calculate $S(I)$, one already has to divide I into two equal parts, I_0 and I_1. In order to calculate $S(I_0)$ and $S(I_1)$, one has to further subdivide each ot these part into two equal parts.

where $\xi_I \in I$, and, noting that $|I_0| = |I_1| = |I|/2$, we have

$$E(I_0) = -\frac{|I|^5}{2^5 \cdot 2880} f^{(4)}(\xi_{I_0}), \quad E(I_1) = -\frac{|I|^5}{2^5 \cdot 2880} f^{(4)}(\xi_{I_0}),$$

where $\xi_{I_0} \in I_0$ and $\xi_{I_1} \in I_1$. Making the assumption that

$$f^{(4)}(\xi) \approx f^{(4)}(\xi_{I_0}) \approx f^{(4)}(\xi_{I_1}), \tag{1}$$

it follows that

$$E(I_0) \approx E(I_1) \approx \frac{E(I)}{2^5}.$$

Hence, noting that

$$\int_I f = S(I) + E(I) = S(I_0) + S(I_1) + E(I_0) + E(I_1),$$

we have

$$S(I_0) + S(I_1) - S(I) = E(I) - E(I_0) - E(I_1) \approx (2^5 - 1 - 1) \cdot E(I_0) = 30E(I_1).$$

Hence

$$E(I_1) \approx E(I_2) \approx \frac{1}{30}(S(I_0) + S(I_1) - S(I)).$$

Thus the error in approximating $\int_I f$ by $S(I_0) + S(I_1)$ is

$$E(I_0) + E(I_1) \approx \frac{1}{15}(S(I_0) + S(I_1) - S(I)).$$

Thus, if

$$\frac{1}{15}|S(I_0) + S(I_1) - S(I)| < \epsilon. \tag{2}$$

then we accept the approximation

$$\int_I f = S(I_0) + S(I_1).$$

If this inequality is not satisfied, then we work with the intervals I_0 and I_1 separately. Noting that

$$\int_I f = \int_{I_0} f + \int_{I_1} f,$$

we try to determine the integral $\int_{I_0}$ with an allowable error less than $\epsilon/2$. Similarly, we try to determine $\int_{I_1}$ with an allowable error less than $\epsilon/2$. If we are successful, the total error committed on I will be less than ϵ.

When continuing this procedure, some parts of I may have to be subdivided many times, others maybe only a few times; in any case, the total error committed on I will be less than ϵ. Of course, there is no certainty that this will be so, since the assumption (1) may fail, especially if we did not make sure that we divided the interval into sufficiently many parts. Therefore, an additional precaution may be that (2) is only accepted as a guarantee that the error is small enough if the length of I itself is not too large.

A C program implementation of adaptive integration is contained in the file `adapt.c`. The header file `adapt.h` to this file is

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 5
 6 double funct(double);
 7 double simpsonadapt(double (*fnct)(double), double a,
 8     double b, double mid, double epsilon, double maxh,
 9     double minh, double fa, double fb, double fmid,
10     double *bada, double *badb,
11     int *success);
```

Similarly as in case of the trapezoidal rule, on line 4, `absval(x)` to calculate the absolute value of `x` is defined, on line 6 the declaration of `funct`, the function to be integrated (defined in a separate file) is given, and in lines 7–11 the declaration of the function `simpsonadapt`, discussed next, is given. The file `adapt.c` contains the definition of this function:

```
 1 #include "adapt.h"
 2
 3 double simpsonadapt(double (*fnct)(double), double a,
 4     double b, double mid, double epsilon, double maxh,
 5     double minh, double fa, double fb, double fmid,
 6     double *bada, double *badb,
 7     int *success)
 8 {
 9   double x, integr1, integr2, integr = 0.0,
10     mid1, mid2, fmid1, fmid2, h, s1, s2;
11   h = b - a;
12   if ( h >= minh ) {
13     mid1 = (a + mid)/2.0; fmid1 = (*fnct)(mid1);
14     mid2 = (mid + b)/2.0; fmid2 = (*fnct)(mid2);
15     s1 = (fa+4.0*fmid+fb)*h/6.0;
16     s2 = (fa+4.0*fmid1+2.0*fmid+4.0*fmid2+fb)*h/12.0;
17     if ( h<=maxh && absval(s2-s1)<= 15.0*epsilon ) {
18        integr = s2;
19        *success = 1;
20     }
21     else {
22       integr1 = simpsonadapt(fnct, a, mid, mid1, epsilon/2.0,
23           maxh, minh, fa, fmid, fmid1, bada, badb, success);
24       if ( *success ) {
25         integr2 = simpsonadapt(fnct, mid, b, mid2, epsilon/2.0,
26           maxh, minh, fmid, fb, fmid2, bada, badb, success);
27       }
28       if ( *success ) {
29         integr = integr1+integr2;
30       }
31     }
32   }
33   else {
34     *success = 0; *bada = a; *badb = b;
35   }
```

```
36   return integr;
37 }
```

The function `simpsonadapt` has parameters (mentioned in lines 3–7) `(*fnct)`, a pointer to the function to be integrated, the limits of the integral `a` and `b`, the midpoint `mid` of the interval of integration (while the midpoint is easy to calculate, the calculation is more efficient if the midpoint is not recalculated each time; the function `simsponadapt` will call itself recursively, and both the calling program and the called program will need the midpoint), the maximum allowable error `epsilon` on the interval `(a,b)`, the maximum length `max h` of a subdivision that need not be subdivided further, the minimum length `minh` of a subdivision (if the error is too large with an interval of length `minh`, the calculation will be declared a failure), the values `fa`, `fb`, `fmid` of the function at `a`, `b`, and `mid`, respectively, pointers `*bada` and `*badb` to the endpoints of the subinterval on which the calculation failed (the calling program can read these values, so it can be decided later to use a different method to determine the integral on the interval where the method failed), and the variable `success` that is 1 (true) if the method is successful and 0 (false) otherwise. In lines 15 and 16, the Simpson sums on the whole interval `(a,b)` and its two parts are calculated, and in line 17 it is decided whether the calculation is successful (when `h<=maxh` and the the absolute value of the difference `s2-s1` estimating the error is small enough). If this is not the case, the method is recursively called in lines 22–26 for the intervals `(a,mid)` and `(mid,b)` with `epsilon/2.0` instead of `epsilon` (since each of these subintervals allow only half the error of the whole interval. The sum of the two integrals on the subintervals, calculated in line 29, will be the integral on the interval `(a,b)`. If the calculation is not successful, the endpoints `a` and `b` are placed into the locations `*bada` and `badb` on line 34.

The file `funct.c` contains the definition of the function `funct`; this file is the same as in case of adaptive integration for the trapezoidal rule. That is, the integral of the function

$$\frac{1}{\sqrt{1+x^3}}$$

is being calculated. The main program, given in the file `main.c`, specifies the limits of the integral as $a = 0$ and $b = 4$, as before:

```
 1 #include "adapt.h"
 2
 3 main()
 4 {
 5   double a = 0.0, b = 4.0, x, integr,
 6     epsilon=5e-10, maxh=1e-1, minh=1e-5, bada, badb,
 7     mid, fa, fb, fmid;
 8   int success;
 9   fa=funct(a); fb=funct(b);
10   mid=(a+b)/2.0; fmid=funct(mid);
11   integr=simpsonadapt(&funct, a, b, mid, epsilon, maxh,
12           minh, fa, fb, fmid, &bada, &badb, &success);
13   if ( success ) {
14     printf("The integral is %.12f\n", integr);
15   }
16   else {
17
18     printf("The integral has trouble in the interval"
19       "(%.12f, %.12f)\n", bada, badb);
20   }
21 }
```

There are very few changes in the calling program compared to the one used in case of adaptive integration for the trapezoidal rule; one difference is that now the calling program has to calculate

the value of the midpoint of the interval `(a,b)` and has to calculate the value of the function there; this is done on line 10. The makefile is the same as in case of adaptive integration for the trapezoidal rule. The printout of the program is a single line (the `1` at the beginning of the line is a line number, not part of the printout).

```
1 The integral is 1.805473301724
```

The first nine digits of the result coincide with those given by adaptive integration with the trapezoidal rule.

Problems

1. We would like to evaluate the integral

$$\int_0^4 \sqrt{1+x^3}\,dx$$

with a precision of $\epsilon = 5 \cdot 10^{-4}$ using the adaptive Simpson Rule. Writing $f(x) = \sqrt{1+x^3}$, on the interval $[2,3]$ we find that

$$S_{[2,3]} = \frac{1}{6}\left(f(2) + 4f\left(2\frac{1}{2}\right) + f(3)\right) \approx 4.100,168,175.$$

and

$$S_{\left[2,2\frac{1}{2}\right]} + S_{\left[2\frac{1}{2},3\right]} = \frac{1}{12}\left(f(2) + 4f\left(2\frac{1}{4}\right) + 2f\left(2\frac{1}{2}\right) + 4f\left(2\frac{3}{4}\right) + f(3)\right) = 4.100,102,756.$$

Do we have to subdivide the interval $[2,3]$ further or the result is already precise enough.

Solution. The error allowed on the interval $[2,3]$ is $\epsilon/4 = 1.25 \cdot 10^{-4}$. The error can be estimated as

$$\frac{1}{15}\left(S_{\left[2,2\frac{1}{2}\right]} + S_{\left[2\frac{1}{2},3\right]} - S_{[2,3]}\right) \approx \frac{4.100,102,756 - 4.100,168,175}{15} = -4.361,27 \cdot 10^{-6}.$$

This is considerably smaller in absolute value than the permissible error, so the interval does not need to be further subdivided; one can accept $S_{\left[2,2\frac{1}{2}\right]} + S_{\left[2\frac{1}{2},3\right]} = 4.100,102,756$ as the integral on this interval.

24. THE EULER-MACLAURIN SUMMATION FORMULA

Let a and b be integers, $a < b$, let $N > 0$ be an integer, and assume f is a function that is N times differentiable on the interval $[a,b]$, and $f^{(N)}$ is continuous on (a,b).[58] Then Euler-Maclaurin summation formula states that we have[59]

$$\begin{aligned}\frac{f(a)}{2} + \sum_{i=a+1}^{b-1} f(i) + \frac{f(b)}{2} &= \int_a^b f(x)\,dx + \sum_{\substack{m=2 \\ m \text{ is even}}}^{N} \frac{B_m}{m!}\left(f^{(m-1)}(b) - f^{(m-1)}(a)\right) \\ &\quad - \frac{(-1)^N}{N!}\int_a^b P_N(x) f^{(N)}(x)\,dx.\end{aligned} \tag{1}$$

[58] The continuity of $f^{(N)}$ is needed to justify the integration by parts involving $f^{(N)}$ in the proof of (1) given later in this section. Insofar as this integration by parts can be justified under weaker assumptions, the continuity requirement can be relaxed. It is sufficient to assume, for example, that $f^{(N)}$ is Riemann-integrable on $[a,b]$ instead of assuming that it is continuous on (a,b).

[59] The notation used will be explained below.

Here $P_m(x) = B_m(x - [x])$, were $[x]$ is the integer part of x (i.e., the largest integer not exceeding x), and the Bernoulli polynomials $B_m(x)$ are defined by the equation[60]

$$\frac{ze^{xz}}{e^z - 1} = \sum_{m=0}^{\infty} \frac{B_m(x)}{m!} z^m. \tag{2}$$

In other words, the function

$$\frac{ze^{xz}}{e^z - 1}$$

on the left-hand side, considered as a function of z, can be expanded into a Taylor series at $z = 0$, and it turns out that the coefficients $B_m(x)/m!$ of this Taylor series are polynomials of degree m of x; these coefficients identify the Bernoulli polynomials.[61] Finally, the Bernoulli numbers B_m are defined as $B_m(0)$. Substituting $x = 0$ in the above series, this shows that the Bernoulli numbers are defined by the Taylor expansion

$$\frac{z}{e^z - 1} = \sum_{m=0}^{\infty} \frac{B_m}{m!} z^n. \tag{3}$$

It turns out that $B_0(x) = 1$, $B_1(x) = x - \frac{1}{2}$, $B_2(x) = x^2 - x + \frac{1}{6}$, and $B_0 = 1$, $B_1 = -1/2$; for odd $m > 1$ we have $B_m = 0$, and $B_2 = 1/6$, $B_4 = -1/30$, $B_6 = 1/42$, $B_8 = -1/30$, $B_{10} = 5/66$, $B_{12} = -691/2730$, $B_{14} = 7/6$, ... To see that $B_m = 0$ for odd $m > 1$, observe that the function

$$\frac{z}{e^z - 1} + \frac{z}{2} = \frac{1}{2} z \frac{e^z + 1}{e^z - 1} = \frac{1}{2}(-z)\frac{e^{-z} + 1}{e^{-z} - 1}$$

is an even function[62] of z, so the odd powers of z are missing from its Taylor expansion at $z = 0$. The proof of the Euler-Maclaurin summation formula will be given later in this section.

Writing $N = 2M$ or $N = 2M + 1$ with an integer M, (1) can be rewritten as

$$\frac{f(a)}{2} + \sum_{i=a+1}^{b-1} f(i) + \frac{f(b)}{2} = \int_a^b f(x)\,dx + \sum_{m=1}^{M} \frac{B_{2m}}{(2m)!}\Big(f^{(2m-1)}(b) - f^{(2m-1)}(a)\Big) + R, \tag{4}$$

where the remainder term R can be written as

$$R = -\frac{1}{(2M)!} \int_a^b P_{2M}(x) f^{(2M)}(x)\,dx \tag{5}$$

in case $M > 0$, f is $2M$ times differentiable on $[a, b]$, and $f^{(2M)}$ is continuous on (a, b), and

$$R = \frac{1}{(2M+1)!} \int_a^b P_{2M+1}(x) f^{(2M+1)}(x)\,dx \tag{6}$$

---

[60] In the mathematical literature, there a number of different definitions of the Bernoulli polynomials, giving rise to different, but closely related sets of polynomials. The one given here is the modern version. When reading a statement involving Bernoulli polynomials in the mathematical literature, one needs to check carefully which definition of these polynomials is taken.

[61] The expression

$$\frac{ze^{xz}}{e^z - 1},$$

regarded as a function of z, is not defined for $z = 0$, but taking its value to be 1 for $z = 0$, it becomes differentiable any number of times, and its Taylor series at $z = 0$ will converge for $|z| < 2\pi$ (this value for the radius of convergence is a simple consequence of results from complex function theory). That $B_m(x)$ is indeed a polynomial of x will be a by-product of the proof of formula (1), given later in this section. See footnote 63 on p. 87.

[62] A function $f(t)$ is called even if $f(-t) = f(t)$. $f(t)$ is called odd if $f(-t) = -f(t)$

in case $M \geq 0$, f is $2M+1$ times differentiable on $[a,b]$, and $f^{(2M+1)}$ is continuous on (a,b).

By making the substitution $t = hx$ in the integral in (1), and writing $x_i = a + ih$, one can obtain a version of the trapezoidal rule with new error terms. When making the substitution $t = hx$ in the Euler summation formula (1) above, we also change to notation, and write A, B, and F instead of a, b, and f. We obtain

$$\frac{F(A)}{2} + \sum_{i=A+1}^{B-1} F(i) + \frac{F(B)}{2} = \int_{Ah}^{Bh} F(t/h)\frac{1}{h}\,dt + \sum_{\substack{m=2 \\ m \text{ is even}}}^{N} \frac{B_m}{m!}\Big(F^{(m-1)}(B) - F^{(m-1)}(A)\Big) - \frac{(-1)^N}{N!}\int_{Ah}^{Bh} P_N(t/h)F^{(N)}(t/h)\frac{1}{h}\,dt.$$

Write $a = Ah$, $b = Bh$, and $f(t) = F(t/h)$, and note that $f^{(k)}(t) = h^{-k}F^{(k)}(t/h)$, i.e., $h^k f^{(k)}(t) = F^{(k)}(t/h)$. Multiplying both sides by h, and writing

$$T(f,n) = \frac{h}{2}\big(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\big)$$

for the trapezoidal sum, we obtain

$$T(f,n) = \int_a^b f(t)\,dt + \sum_{\substack{m=2 \\ m \text{ is even}}}^{N} \frac{B_m h^m}{m!}\Big(f^{(m-1)}(b) - f^{(m-1)}(a)\Big) - \frac{(-1)^N}{N!}\int_a^b P_N(t/h)h^N f^{(N)}(t)\,dt.$$

An interesting point about the error term here is that if the function f is periodic with period $b-a$, then all the terms in the sum on the right-hand side are zero. In this case, we obtain for the integral that

$$\int_a^b f = T(f,n) + O(h^N).$$

That is, when integrating a periodic, many times differentiable function along its whole period, the trapezoidal rule gives a much better error term, than, for example, Simpson's rule. Further, Romberg integration (discussed in the next section) does not improve on the trapezoidal rule in this case. For example, in calculating the integral

$$\int_0^{2\pi} e^{\sin x}\,dx$$

one would use the trapezoidal rule, and not Simpson's rule or Romberg integration.

Proof of the Euler-Maclaurin summation formula. For any integer c with $a \leq c < b$ we have

$$\begin{aligned}\int_c^{c+1} f(t)\,dt &= \int_c^{c+1} \left(t - c - \frac{1}{2}\right)' f(t)\,dt \\ &= \left(t - c - \frac{1}{2}\right) f(t)\Big|_{t=c}^{t=c+1} - \int_c^{c+1} \left(t - c - \frac{1}{2}\right) f'(t)\,dt \\ &= \frac{1}{2}(f(c) + f(c+1)) - \int_c^{c+1} B_1(t-c)f'(t)\,dt.\end{aligned} \tag{7}$$

The second equality here follows by integration by parts, and the third equality holds since $B_1(x) = x - \frac{1}{2}$. According to (2) above we have

$$\sum_{m=0}^{\infty} \frac{B_m(x)}{m!} z^{m+1} = \frac{z^2 e^{xz}}{e^z - 1} = \frac{\partial}{\partial x} \frac{z e^{xz}}{e^z - 1} = \sum_{m=0}^{\infty} \frac{B'_m(x)}{m!} z^m.$$

Here the first equality is just (2) multiplied by z and with the sides reversed, and the third equality is obtained by taking the partial derivative of (2) with respect to x. Equating the coefficients of z^{m+1} on the extreme members of these equations, we obtain that $B'_{m+1}(x) = (m+1)B_m(x)$ for all $m \geq 0$.[63] Hence, using integration by parts,

$$\begin{aligned}(m+1)\int_c^{c+1} B_m(t-c) f^{(m)}(t)\,dt &= \int_c^{c+1} B'_{m+1}(t-c) f^{(m)}(t)\,dt \\ &= B_{m+1}(t-c) f^{(m)}(t)\Big|_{t=c}^{t=c+1} - \int_c^{c+1} B_{m+1}(t-c) f^{(m+1)}(t)\,dt \\ &= B_{m+1}(1) f^{(m)}(c+1) - B_{m+1}(0) f^{(m)}(c) - \int_c^{c+1} B_{m+1}(t-c) f^{(m+1)}(t)\,dt.\end{aligned}$$

Observe that $B_m(0) = B_m(1)$ holds for $m > 1$. This is because we have

$$\sum_{m=0}^{\infty} \frac{B_m(1)}{m!} z^m = \frac{z e^z}{e^z - 1} = z + \frac{z}{e^z - 1} = z + \sum_{m=0}^{\infty} \frac{B_m(0)}{m!} z^m.$$

The first equality here holds according to (2) with $x = 1$, and the third equation holds according to (2) with $x = 0$. Equating the coefficients of powers of z on the extreme members of these equations, we obtain that indeed $B_{m+1}(0) = B_{m+1}(1)$ holds for $m \geq 1$. Hence, noting that $B_{m+1} = B_{m+1}(0)$ by the definition of the Bernoulli numbers, and writing $K = m$, we obtain for $K \geq 1$ that

$$\begin{aligned}&\int_c^{c+1} B_K(t-c) f^{(K)}(t)\,dt \\ &\qquad = \frac{1}{K+1}\Big(B_{K+1} \cdot \big(f^{(K)}(c+1) - f^{(K)}(c)\big) - \int_c^{c+1} B_{K+1}(t-c) f^{(K+1)}(t)\,dt\Big).\end{aligned}$$

Using this equation, the formula

$$\begin{aligned}\int_c^{c+1} f(t)\,dt &= \frac{1}{2}(f(c) + f(c+1)) \\ &\quad - \sum_{m=2}^{K} (-1)^m \frac{B_m}{(m)!}\big(f^{(m-1)}(c+1) - f^{(m-1)}(c)\big) + \frac{(-1)^K}{K!}\int_c^{c+1} B_K(t-c) f^{(K)}(t)\,dt.\end{aligned}$$

can be established for $K = 1, 2, \ldots, N$ by induction. Indeed, for $K = 1$, the formula is equivalent to (7), and the induction step from K to $K+1$ follows by expressing the integral on the right-hand side with the aid of the the equation above. For $K = N$, this equation is equivalent to (1).

To obtain (1), take this last formula for $K = N$ and sum it for $c = a, a+1, \ldots, b-1$. To see that indeed (1) results this way, one only needs to point out that $P_N(t) = B_N(t - [t]) = B_N(t - c)$ for t with $c \leq t < c + 1$, and that $B_m = 0$ for odd $m > 1$. □

[63] This equation shows that $B_m^{(m)}(x) = m! B_0(x) = m!$, so that $B_m(x)$ is indeed a polynomial of degree m of x.

Connection with finite differences. The Euler-Maclaurin summation formula can be explained in terms of the calculus of finite differences. While this explanation can be developed into a full-featured proof of the formula, we will not do this here. Let f be a function, let E be the forward shift operator (with $h = 1$, see Section 9), that is, $Ef(x) = f(x+1)$, and let D be the differentiation operator, i.e., $Df(x) = f'(x)$; finally, let I be the identity operator, that is $If(x) = f(x)$. Then, expressing $f(x+1)$ into a Taylor series expansion at x, we have

$$f(x+1) = \sum_{n=0}^{\infty} \frac{1}{n!} f^{(n)}(x);$$

of course, this is valid only under some restrictive assumptions on f, but for now we will assume that this formula is valid. In terms of the operators D and E, this can be written as

$$Ef = \Big(\sum_{n=0}^{\infty} \frac{1}{n!} D^n\Big) f = e^D f,$$

where the last equality can be taken as the definition of the exponential e^D of the operator D in terms of the Taylor series of the exponential function. Writing F for an antiderivative of f (that is, $F' = f$), and using the above formula for F instead of f, we obtain

$$\int_x^{x+1} f = F(x+1) - F(x) = (e^D - I)F(x).$$

Multiplying both sides by $(e^D - I)^{-1}$ we can write that[64]

$$F(x) = (e^D - I)^{-1}(F(x+1) - F(x)).$$

Multiplying both sides by D on the left, we obtain

$$DF(x) = D(e^D - I)^{-1}(F(x+1) - F(x)).$$

Noting that

$$D(e^D - I)^{-1} = \sum_{m=0}^{\infty} \frac{B^m}{m!} D^m$$

according to (3), the last equation can be written as

$$DF(x) = \sum_{m=0}^{\infty} \frac{B_m}{m!} D^m (F(x+1) - F(x))$$

In view of the equations $DF = F' = f$, $B_0 = 1$, and $B_1 = -1/2$, this formula can be written as

$$f(x) = F(x+1) - F(x) - \frac{1}{2}(f(x+1) - f(x)) + \sum_{m=2}^{\infty} \frac{B_m}{m!} (f^{(m-1)}(x+1) - f^{(m-1}(x)).$$

Observing that $F(x+1) - F(x) = \int_x^{x+1} f$, it follows that

$$\frac{1}{2}\big(f(x) + f(x+1)\big) = \int_x^{x+1} f + \sum_{m=2}^{\infty} \frac{B_m}{m!} (f^{(m-1)}(x+1) - f^{(m-1}(x)), \tag{8}$$

[64] At this point all these calculations should be taken only as symbolic manipulations – the question whether the inverse of $e^D - I$ is meaningful should be bypassed for now.

which equivalent the Euler-Maclaurin summation formula if on the right-hand side of (1) we make $N \to \infty$, and we ignore the last integral on the right-hand side. Unfortunately, this cannot be done, since the sum on the right-hand side of (1) usually diverges when $N \to \infty$. Nevertheless, this argument works in case f is a polynomial, since in this case high-order derivatives of f are zero. In fact, when f is a polynomial, all the arguments above are rigorous, since all the infinite series in fact turn into finite series. So this argument indeed gives a rigorous proof (of the form (8)) of the Euler-Maclaurin formula polynomials.

If one wants to get a proof of the general form if the Euler-Maclaurin summation formula, one may try to derive formula (1) from formula (8) by finding a polynomial P that approximates f, f', …, $f^{(N)}$ uniformly on the interval $[a, b]$ by the Weierstrass approximation theorem.[65] But then one runs into technical difficulties as to how to obtain the integral on the right-hand side of (1). This integral can be considered in a sense a "remainder term" of the infinite series on the right-hand side of (1). On may probably obtain this integral in a way similar to the way one obtains the integral remainder term of the Taylor formula.[66] However, it is questionable whether this is an avenue worth pursuing. Namely, a proof of the Taylor formula with integral remainder term can be obtained by repeated integration by parts, in much the same way as was done in the first proof of the Euler-Maclaurin formula; in fact, such a proof is in some sense more satisfying than the one we gave for the Taylor formula in Section 4 using telescoping sums. It is better to think of the above argument leading to formula (8) as an intuitive justification of the appearance of the Bernoulli numbers, defined by (3), in the Euler-Maclaurin summation formula.

Zeros of the Bernoulli polynomials. According to (2), we have

$$\frac{ze^{xz}}{e^z - 1} - \frac{ze^{(1-x)z}}{e^z - 1} = \sum_{m=0}^{\infty} \frac{B_m(x) - B_m(1-x)}{m!} z^m.$$

The left-hand side here can also be written as

$$\frac{z(e^{(x-1/2)z} - e^{(1/2-x)z})}{e^{z/2} - e^{-z/2}}.$$

It is easy to see that this is an odd function[67] of z. Therefore, in its Taylor expansion at $z = 0$ the coefficients of even powers of z are zero. Hence $B_m(x) = B_m(1-x)$ for even m. We saw above, in the proof of the Euler-Maclaurin summation formula, that we have $B'_m(x) = mB_{m-1}(x)$ for all $m > 0$. Hence, for even $m > 1$, we have

$$mB_{m-1}(x) = B'_m(x) = -B'_m(1-x) = -mB_{m-1}(1-x).$$

Thus $B_m(x) = -B_m(1-x)$ for odd $m > 0$. It therefore follows that $B_m(1/2) = -B_m(1/2)$ for odd m. Hence $B_m(1/2) = 0$ for odd m. Thus, for odd $m > 1$, $B_m(x)$ has zeros at $x = 0$, $1/2$, 1; this is because we saw above that $B_m(0) = B_m = 0$ for odd m, and then $B_m(1) = B_m(1-1) = B_m(0) = 0$ also holds. We are going to show that $B_m(x)$ has no more zeros in the interval $[0, 1]$ for odd m.

---

[65] The Weierstrass approximation theorem says if f is continuous on $[a, b]$ then for every $\epsilon > 0$ there is a polynomial P such that $|f(x) - P(x)| < \epsilon$ for all $x \in [a, b]$.

[66] The Taylor formula with integral remainder term says can be written as

$$f(b) = \sum_{k=0}^{n} \frac{f^k(a)}{k!} + \int_a^b \frac{f^{(n+1)}(x)\,(b-x)^n}{n!}\,dx,$$

provided f^k is exists on $[a, b]$ (if $a < b$) or $[b, a]$ (if $b < a$) and $f^{(n+1)}$ is continuous on $[a, b]$ or $[b, a]$. This can be proved from (1) in the Lemma in Section 4 by expressing $R_n(b, x)$ for $x = a$ as the integral of its derivative.

[67] See footnote 62 on p. 85.

Indeed, assume that $m > 1$ is odd, and $B_m(x)$ has at least for zeros (counted with multiplicity) in the interval $[0, 1]$. Then $B'_m(x) = mB_{m-1}$ has at least three zeros in the interval $(0, 1)$ by Rolle's theorem, and $B'_{m-1}(x) = (m-1)B_{m-2}(x)$ has at lest two zeros in $(0, 1)$. If $m - 2 = 1$ then this is a contradiction, since them, $B_{m-2}(x) = B_1(x)$, being a linear polynomial, can have at most one zero. If $m-2 > 1$, then $B_{m-2}(x)$ also has zeros at $x = 0$ and $x = 1$; thus, $B_{m-2}(x)$ has at least four zeros in the interval $[0, 1]$; therefore we can repeat the same argument. In the end, we will reach the contradictory conclusion that $B_(x)$ has at least two zeros in $(0, 1)$. Hence, indeed, the only zeros of $B_m(x)$ for odd $m > 1$ in the interval $[0, 1]$ are $x = 0$, $1/2$, and 1.

Now, if $m > 0$ is even, then $B_m(x) = \frac{1}{m+1}B'_{m+1}(x)$ must have at least one zero in the interval $(0, 1/2)$, and another zero in the interval $(1/2, 1)$, by Rolle's theorem. It cannot have more zeros in $[0, 1]$, since then $mB_{m-1}(x) = B'_m(x)$ would have at least two zeros in $(0, 1)$ by Rolle's theorem, and we know that this is not the case. That is, for $m > 0$ even, $B_m(x)$ has exactly two zeros in the interval $[0, 1]$, one in $(0, 1/2)$ and one in $(1/2, 1)$.

Mean-value form of the remainder term. The remainder term (6) to formula (4) can be written as

$$R = \frac{1}{(2M+1)!}\sum_{c=a}^{b-1}\int_c^{c+1/2} P_{2M+1}(x)f^{(2M+1)}(x)\,dx + \frac{1}{(2M+1)!}\sum_{c=a}^{b-1}\int_{c+1/2}^{c+1} P_{2M+1}(x)f^{(2M+1)}(x)\,dx$$

Above we saw that the only zeros of $B_m(x)$ for odd $m > 1$ in the interval $[0, 1]$ are $x = 0$, $1/2$ and 1 (and the only zero of $B_1(x)$ is $x = 1/2$), it follows that $P_{2M+1}(x) = B_{2M+1}(x - [x])$ does not change sign on any of the intervals $[c, c + 1/2]$ and $[c + 1/2, c + 1]$ for integer c. Hence the integrals in the above sums can be estimated by the Mean-Value Theorem discussed in formula (1) in Section 20 (p. 69 – take $\xi = x$ in that formula):

$$\int_c^{c+1/2} P_{2M+1}(x)f^{(2M+1)}(x)\,dx = f^{(2M+1)}(\xi_c)\int_c^{c+1/2} P_{2M+1}(x)\,dx$$
$$= f^{(2M+1)}(\xi_c)\frac{B_{2M+2}(1/2) - B_{2M+2}(0)}{2M+2} = f^{(2M+1)}(\xi_c)\frac{B_{2M+2}(1/2) - B_{2M+2}}{2M+2}.$$

The first equality holds for some $\xi_c \in (c, c + 1/2)$ by the Mean-Value Theorem quoted. The second equality holds because $B'_m(x) = mB_{m-1}(x)$ for all $m > 0$, and the third equality holds since $B_m = B_m(0)$ by the definition of the Bernoulli numbers. Similarly,

$$\int_{c+1/2}^{c+1} P_{2M+1}(x)f^{(2M+1)}(x)\,dx = f^{(2M+1)}(\eta_c)\int_{c+1/2}^{c+1} P_{2M+1}(x)\,dx$$
$$= f^{(2M+1)}(\eta_c)\frac{B_{2M+2}(1) - B_{2M+2}(1/2)}{2M+2} = -f^{(2M+1)}(\eta_c)\frac{B_{2M+2}(1/2) - B_{2M+2}}{2M+2}.$$

The first equality holds for some $\eta_c \in (c+1/2, c+1)$ by the Mean-Value Theorem quoted. The third equality holds since $B_m(1) = B_m(0) = B_m$ for all $m \geq 0$. Hence we have

$$R = \frac{B_{2M+2}(1/2) - B_{2M+2}}{(2M+2)!}\sum_{c=a}^{b-1}\Big(f^{(2M+1)}(\xi_c) - f^{(2M+1)}(\eta_c)\Big),$$

where $\xi_c \in (c, c + 1/2)$ and $\eta_c \in (c + 1/2, c + 1)$. Below, we will show that

$$B_m\left(\frac{1}{2}\right) = (2^{1-m} - 1)B_m. \tag{9}$$

With the aid of this relation, the above expression for R can be simplified as

$$R = -\frac{2-2^{-1-2M}}{(2M+2)!}B_{2M+2}\sum_{c=a}^{b-1}\Big(f^{(2M+1)}(\xi_c) - f^{(2M+1)}(\eta_c)\Big). \tag{10}$$

In order that the form (6) be a valid expression for the remainder term in (4), we needed to assume that $f^{(2M+1)}$ be continuous on (a,b). It may be of some interest to point that that for the validity of (10) one needs to assume only that $f^{(2M+1)}(x)$ should exist for all $x \in (a,b)$. Of course, if one wants to derive (10) under this weaker assumption, we cannot use (6). Instead, the last integration by parts in the derivation of the Euler-Maclaurin summation formula needs to be replaced by a Mean-Value Theorem. The formula for integration by parts says that

$$\int_\alpha^\beta f'g = f(x)g(x)\Big|_{x=\alpha}^{x=\beta} - \int_\alpha^\beta fg'.$$

In case f does not change sign on (α,β),[68] one can write this as

$$\int_\alpha^\beta f'g = f(x)g(x)\Big|_{x=\alpha}^{x=\beta} - g'(\xi)\int_\alpha^\beta f \tag{11}$$

for some $\xi \in (\alpha,\beta)$ by the same Mean-Value Theorem for integrals that we used above. The point is that this last equation is valid even without the assumption that the integration by parts formula above is valid. In fact, (11) is valid if f does not change sign on (α,β), f' and g are continuous on $[\alpha,\beta]$, and $g'(x)$ exists for all $x \in (\alpha,\beta)$. That is, it may happen that fg' is not integrable on $[\alpha,\beta]$, in which case the integration by parts formula is certainly not valid, and yet (11) is still valid. As is clear from the proof of the Euler-Maclaurin summation formula, in deriving (6), one integrates (5) by parts (in case $M > 0$; in case $M = 0$, one uses the integration by parts occurring in (7)). This integration by parts needs to be replaced by (11) (on the intervals $[c, c+1/2]$, and $[c+1/2, c+1]$ for c with $a \le c < b$) when deriving (10) with assuming only that $f(2M+1)(x)$ exists for $x \in (a,b)$.

To establish (11) under these weaker assumptions, first assume that $f(x) \neq 0$ for $x \in (\alpha,\beta)$. choose A such that

$$\int_\alpha^\beta f'g = f(t)g(t)\Big|_{t=\alpha}^{t=\beta} - A\int_\alpha^\beta f.$$

Such an A exists since $\int_\alpha^\beta f \neq 0$, given that f is either always positive or always negative on the interval (α,β). Then, writing

$$F(x) = \int_\alpha^x f'g - f(t)g(t)\Big|_{t=\alpha}^{t=x} + A\int_\alpha^x f,$$

we have $F(\alpha) = 0$ and $F(\beta) = 0$ (the latter by the choice of A). Since F is differentiable on (α,β), by Rolle's theorem there is a $\xi \in (\alpha,\beta)$ such that $F'(\xi) = 0$. That is,

$$0 = F'(\xi) = f'(\xi)g(\xi) - (f'(\xi)g(\xi) + f(\xi)g'(\xi)) + Af(\xi) = (A - g'(\xi))f(\xi).$$

This equation implies that $A = g'(\xi)$, showing that (11) holds.

In case $f(x) = 0$ for some $x \in (\alpha,\beta)$, the interval (a,b) can be decomposed as a union of (finitely many or infinitely many) intervals $I_n = [\alpha_n, \beta_n]$ such that any two of these intervals have at most one endpoint in common (i.e., they have no internal points in common),[69] $f(x) = 0$ at the endpoints of each of these intervals unless these endpoints equal α or β (when $f(x)$ may or may not be zero), and, for each n, either $f(x) \neq 0$ for all $x \in (\alpha_n, \beta_n)$ or $f(x) = 0$ for all $x \in [\alpha_n, \beta_n]$. Since the formula analogous to (11) has already been established for f and g on each I_n with some $\xi_n \in (\alpha_n, \beta_n)$ (for those intervals I_n on which f identically vanishes, (11) is trivially true), we have

$$\int_\alpha^\beta f'g = f(x)g(x)\Big|_{x=\alpha}^{x=\beta} - \sum_n g'(\xi_n)\int_{I_n} f$$

(the terms $f(x)g(x)$ for $x = \alpha_n$ or $x = \beta_n$ with $x \neq \alpha$ and $x \neq \beta$ on the right-hand side do not appear, since $f(x) = 0$ for those values of x). Write

$$S = \sup\left\{g'(\xi) : \int_{I_n} f \neq 0\right\}$$

[68] That is, either $f(x) \ge 0$ for all $x \in (\alpha,\beta)$, or $f(x) \le 0$ for all $x \in (\alpha,\beta)$.

[69] Note that I_{n+1} need not be adjacent to I_n; in fact, possibly there are infinitely many intervals I_k between I_n and I_{n+1}.

and

$$s = \inf\left\{g'(\xi) : \int_{I_n} f \neq 0\right\};$$

we allow $s = -\infty$ or $S = +\infty$, so that we do not assume that these sets are bounded, but we assume that these sets are not empty – i.e., we disregard the trivial case when $f(x) = 0$ for all $x \in (\alpha, \beta)$. Write

$$H = \frac{\sum_n g'(\xi_n) \int_{I_n} f}{\int_\alpha^\beta f};$$

as $\int_\alpha^\beta f = \sum_n \int_{I_n} f$, it is easy to see that $s \le H \le S$. In fact, we have $s < H < S$ unless $s = S$. In any case, there are i and j such that $g'(\xi_i) \le H \le g'(\xi_j)$. Since a derivative always has the intermediate-value property,[70] there is a $\xi \in (\alpha, \beta)$ such that $H = g'(\xi)$. Then (11) is satisfied with this ξ.

Evaluation of $B_m(1/2)$. Next we will prove formula (9). With $t = 2z$, we have

$$\frac{z}{e^z - 1} + \frac{z}{e^z + 1} = \frac{2ze^z}{e^{2z} - 1} = \frac{te^{t\cdot(1/2)}}{e^t - 1} = \sum_{m=0}^{\infty} \frac{B_m(1/2)}{m!} t^m = \sum_{m=0}^{\infty} \frac{B_m(1/2)}{m!} 2^m z^m,$$

where the third equality holds according to (2). Similarly,

$$\frac{z}{e^z - 1} - \frac{z}{e^z + 1} = \frac{2z}{e^{2z} - 1} = \frac{t}{e^t - 1} = \sum_{m=0}^{\infty} \frac{B_m}{m!} t^m = \sum_{m=0}^{\infty} \frac{B_m}{m!} 2^m z^m,$$

where the third equality holds according to (3). Hence

$$\sum_{m=0}^{\infty} \frac{2B_m}{m!} z^m = 2 \cdot \frac{z}{e^z - 1} = \sum_{m=0}^{\infty} \frac{B_m(1/2) + B_m}{m!} 2^m z^m;$$

the first equality here holds again according to (3), and the second equality can be obtained by adding the two equations above. Equating coefficients of z^m on the two sides, formula (9) follows.

Problem

1. Euler's constant γ is defined as

$$\gamma = \lim_{n\to\infty} \Big(\sum_{i=1}^{n} \frac{1}{i} - \log n.\Big)$$

Evaluate γ with 10 decimal precision.

Solution. Writing $f(x) = 1/x$, we have $f^{(m)}(x) = (-1)^m m!/x^{m+1}$. Hence, given positive integers a and b, according to the Euler-Maclaurin summation formula (4) with remainder term (10) we have

$$\frac{1}{2a} + \sum_{c=a+1}^{b-1} \frac{1}{c} + \frac{1}{2b} - \int_a^b \frac{dx}{x} = -\sum_{m=2}^{M} \frac{B_{2m}}{2m} \Big(\frac{1}{b^{2m}} - \frac{1}{a^{2m}}\Big) + R,$$

Note the absence of the factorial in the denominator after the sum sign on the right-hand side; this is because of cancelation with the factorial in $f^{(2m-1)}(x)$; also, the negative sign on the right-hand

[70] That is, if g is differentiable on $[u, v]$ and H is such that $g'(u) \le H \le g'(v)$ or $g'(u) \ge H \ge g'(v)$, then there is a ξ in $[a, b]$ such that $g'(\xi) = H$. This is true even if g' is not continuous.

side occurs because $(-1)^{2m-1} = -1$ occurs in the expression for $f^{(2m-1)}(x)$. Formula (10) for the remainder term gives

$$R = \frac{2 - 2^{-1-2M}}{2M+2} B_{2M+2} \cdot S,$$

where

$$S = \sum_{c=a}^{b-1} \Big(\frac{1}{\xi_c^{2M+2}} - \frac{1}{\eta_c^{2M+2}} \Big).$$

Again the absence of the factorial and of the minus sign (the one that occurs in (10) in the first of these formulas is explained by the form of f^{2M+2}. Here $\xi_c \in (c, c+1/2)$ and $\eta_c \in (c+1/2, c+1)$. If we put $\xi_c = c$ and $\eta_c = c+1$, we get a sum larger than S; if we put $\xi_c = c+1/2$ and $\eta_c = c+1/2$, we get a sum smaller than S. These latter sums are easily evaluated since almost everything in them cancels. Thus we obtain

$$0 < S < \frac{1}{a^{2M+2}} - \frac{1}{b^{2M+2}}. \tag{12}$$

Hence, for any integer $a \geq 1$ we have

$$\begin{aligned}
\gamma &= \lim_{n\to\infty} \Big(\sum_{i=1}^{n} \frac{1}{i} - \log n \Big) = \lim_{n\to\infty} \Big(\sum_{i=1}^{n} \frac{1}{i} - \int_1^n \frac{dx}{x} \Big) \\
&= \sum_{i=1}^{a-1} \frac{1}{i} + \frac{1}{2a} - \int_1^a \frac{dx}{x} + \lim_{n\to\infty} \Big(\frac{1}{2a} + \sum_{i=a+1}^{n-1} \frac{1}{i} + \frac{1}{2n} - \int_a^n \frac{dx}{x} \Big) \\
&= \sum_{i=1}^{a-1} \frac{1}{i} + \frac{1}{2a} - \log a + \sum_{m=2}^{M} \frac{B_{2m}}{2m} \cdot \frac{1}{a^{2m}} + R,
\end{aligned}$$

where, by making $b \to \infty$ in (12) we have

$$0 < R \leq \frac{2 - 2^{-1-2M}}{2M+2} \cdot \frac{B_{2M+2}}{a^{2M+2}} \qquad \text{or} \qquad 0 > R \geq \frac{2 - 2^{-1-2M}}{2M+2} \cdot \frac{B_{2M+2}}{a^{2M+2}}$$

depending on whether B_{2M+2} is positive or negative.[71] For $a = 10$ and $M = 3$ this gives

$$0 > R \geq \frac{2 - 2^{-7}}{10} \cdot \frac{-1/30}{10^{10}} \approx -6.640 \cdot 10^{-11},$$

which is slightly worse than what is required for ten decimal precision. For $a = 10$ and $M = 4$ we have

$$0 < R \leq \frac{2 - 2^{-9}}{10} \cdot \frac{5/66}{10^{10}} \approx 1.514 \cdot 10^{-12},$$

and this gives an acceptable result. Using the latter values for a and M, we obtain that

$$0.577,215,664,900,80 \lessapprox \gamma \lessapprox 0.577,215,664,902,31.$$

To compare this with what one can do without the Euler-Maclaurin summation formula, note that for the actual value of γ up to 14 decimals we have $\gamma = 0.577,215,664,901,53$, while

$$\sum_{i=1}^{n} \frac{1}{i} - \log n$$

for $n = 10,000$ gives $0.577,265,664,068,20$; for $n = 100,000$, it gives $0.577,220,664,893,20$; for $n = 1,000,000$, it gives $0.577,216,164,901,45$; for $n = 10,000,000$, it gives $0.577,215,714,901,53$. Even in the last result, the seventh decimal place deviates from the decimal expansion of γ.

[71] It is known that B_{2m} for $m > 0$ is positive if m is odd, and it is negative if m is even.

25. ROMBERG INTEGRATION

Let f be a function on the interval $[a, b]$, and assume that f is at least $2N-1$ times continuously differentiable for some $N > 0$. Let $n > 0$ be an integer, let $h = (b-a)/n$ and $x_i = a + ki$ for $0 \le i \le n$, and write $T(f, n)$ for the trapezoidal sum approximating the integral of f on $[a, b]$:

$$T(f,n) = \frac{h}{2}\Big(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\Big)$$

The Euler (or Euler-Maclaurin) summation formula, discussed in Section 24, generalizes the trapezoidal rule in that it gives a more precise error term. According to this formula, for h close to 0 we have

$$\int_a^b f = T(f,n) + \sum_{l=1}^{N-1} c_l h^{2l} + O(h^{2N-1}), \tag{1}$$

where the constants c_l and the error term $O(h^{2N-1})$ depend on f.[72] This equation allows one to do Richardson extrapolation on the trapezoidal sum for various values of h.

The organization of this calculation is as follows. One calculates the approximations $S_{k,j}$ to $\int_a^b f$ as follows, where k and j are integers with $0 \le j \le k$. For $j = 0$, this approximation is the trapezoidal sum $T(f, 2^k)$. Writing $h_k = (b-a)/2^k$ for $k \ge 0$, we have

$$S_{0,0} = \frac{h_0}{2}(f(a) + f(b))$$

and

$$S_{k,0} = \frac{S_{k-1,0}}{2} + h_k \sum_{i=1}^{2^{k-1}} f(a + (2i-1)h_k) \tag{2}$$

for $k > 0$; this is because the sum expressing $S_{k,0}$ reuses all the partition points occurring in the sum $S_{k-1,0}$, so there is no reason to do repeat function evaluations used in calculating According to (1), we have

$$\int_a^b f = S_{k,0} + \sum_{l=1}^{N-1} c_l h_k^{2l} + O(h^{2N-1}).$$

The quantity $S_{k,0}$ will be the zeroth Richardson extrapolation. In the jth Richardson extrapolation $S_{k,j}$ the first j terms h_k^{2l} are eliminated:

$$\int_a^b f = S_{k,j} + \sum_{l=j+1}^{N-1} c_{j,l} h_k^{2l} + O(h_k^{2N-1}) \qquad (j \le k); \tag{3}$$

we assume here that $k \le N-2$. For $j > 0$, one obtains $S_{k,j}$ from $S_{k-1,j-1}$ and $S_{k,j}$ by eliminating the coefficient of h_k^{2j}. We have

$$\int_a^b f = S_{k,j-1} + c_{j-1,j} h_k^{2j} + \sum_{l=j+1}^{N-1} c_{j-1,l} h_k^{2l} + O(h_k^{2N-1})$$

[72] The Euler summation formula gives the values of these constants and that of the error term explicitly.

and

$$\begin{aligned}\int_a^b f &= S_{k-1,j-1} + c_{j-1,j}h_{k-1}^{2j} + \sum_{l=j+1}^{N-1} c_{j-1,l}h_{k-1}^{2l} + O(h_{k-1}^{2N-1}) \\ &= S_{k-1,j-1} + c_{j-1,j}2^{2j}h_k^{2j} + \sum_{l=j+1}^{N-1} c_{j-1,l}2^{2l}h_k^{2l} + 2^{2N-1}O(h_k^{2N-1})\end{aligned}$$

where we used the equation $h_{k-1} = 2h_k$. Multiplying the first equation by $4^j = 2^{2j}$ and subtracting the second equation, the coefficient of h^{2j} will be eliminated. I.e.

$$(4^j - 1)\int_a^b f = 4^j S_{k,j-1} - S_{k-1,j-1} + \sum_{l=j+1}^{N-1} c_{j-1,l}(4^j - 4^l)h_k^{2l} + (4^j + 2^{2N-1})O(h_k^{2N-1});$$

note that in the error term, the coefficient is $4^j + 2^{2N-1}$ rather than $4^j - 2^{2N-1}$ since in only a bound on the absolute value of the error is known.[73] Writing

$$S_{k,j} = \frac{4^j S_{k,j-1} - S_{k-1,j-1}}{4^j - 1} \tag{4}$$

and

$$c_{j,l} = \frac{4^j - 4^l}{4^j - 1}c_{j-1,l},$$

we have

$$\int_a^b f = S_{k,j} + \sum_{l=j+1}^{N-1} c_{j,l}h_k^{2l} + O(h_k^{2N-1}),$$

provided we can write

$$\frac{4^j + 2^{2N-1}}{4^j - 1}O(h_k^{2N-1})$$

as $O(h_k^{2N-1})$,[74] which agrees with (3).

When using Romberg's method to calculate an integral, one compares the values $S_{n-1,n-1}$ and $S_{n,n}$ for a positive integer n. When the absolute value of the difference of these values is less than a permissible error, one accepts $S_{n,n}$ as the value of the integral. In selecting n, one usually assumes that $n \leq 30$. Taking a value of n larger than 30 is in general not advisable, since for large n the method becomes unstable, and error in approximating the integral by $S_{n,n}$ increases for large n. This is because the coefficients in the Richardson extrapolation will grow too fast for large n. If one does not obtain a sufficiently small error for $n \leq 30$, one abandons the method and decides that the integral cannot be calculated with this method.

In the computer implementation of Romberg integration, we will calculate the integral

$$\int_0^1 \frac{dx}{1+x^2}.$$

[73]That is, the error in one of the formulas may be positive, and in the other formula it may be negative. Hence, even though the two formulas are subtracted, the errors may add up.

[74]This is justified for the single use of (4) in calculating $S_{k,j}$ from $S_{k,j-1}$ and $S_{k-1,j-1}$, but it is somewhat questionable when the process is repeated so as to get $S_{n,n}$ from $S(k,0)$ with $0 \leq k \leq n$. All one can hope is that the errors do not all add up. This hope is probably justified if the function is smooth enough. In practice, this means that if the function is not sufficiently smooth, Richardson extrapolation after a certain point may produce no improvement.

The header file `romberg.h` is used in all the files implementing the program:

```
#include <stdio.h>
#include <math.h>

#define absval(x) ((x) >= 0.0 ? (x) : (-(x)))

double funct(double);
double romberg(double (*fnct)(double), double a,
    double b, double tol, int maxrow, int *success);
```

This includes declarations form the functions `funct` and `romberg` defined soon. The integrand is defined in the file `funct.c`:

```
#include "romberg.h"

double funct(x)
  double x;
{
  double value;
  value = 1.0/(1.0+x*x);
  return(value);
}
```

Romberg integration is defined in the file `romberg.c`:

```
#include "romberg.h"
#define MAXROW 30
#define F(x) ((*fnct)(x))

double romberg(double (*fnct)(double), double a,
    double b, double tol, int maxrow, int *success)
{
  int j, k, p, state, two_to_kmin1;
  double olds, news, hcoeff, h, s[MAXROW],
       four_to_j, sigma;
  *success=0;
  if ( maxrow > MAXROW ) {
    printf("Too many rows");
    return 0;
  }
  k=1; two_to_kmin1=1; h=(b-a)/2.0;
  news=h*(F(a)+F(b)); s[0]=news;
  /* state==0: more splitting is needed;
     state==1: result is within tolerance;
     state==2: last row reached */
  for (state=0; state == 0;) {
    sigma=0.0; hcoeff=-1.0;
    for (p=1; p <= two_to_kmin1; p++) {
      hcoeff+=2.0;
      sigma+=F(a+hcoeff*h);
    }
    olds=s[0];
    /* A new row of the interpolation table is calculated.
```

```
29        There is no need to keep the old row around. */
30      news=olds/2.0+h*sigma; s[0]=news;
31      four_to_j=1.0;
32      for (j=1; j<k; j++) {
33        four_to_j *= 4.0;
34        news=(four_to_j * news-olds)/(four_to_j-1.0);
35        olds=s[j]; s[j]=news;
36      }
37      /* The new diagonal element is calculated and compared
38         to the old diagonal element. If they are close,
39         the result is accepted */
40      four_to_j *= 4.0;
41      news=(four_to_j * news-olds)/(four_to_j-1.0);
42      if ( absval(news-olds) <= tol ) {
43        state = 1;
44      }
45      else if ( k==maxrow ) {
46        state = 2;
47      }
48      else {
49        /* The new diagonal element is entered in the table */
50        s[k++] = news;
51        h /= 2.0;
52        two_to_kmin1 *= 2;
53      }
54    }
55    if ( state == 1 ) { *success = 1; }
56    /* Some diagnostic information is printed out next.
57       This part is best commented out in a production
58       version -- or else the parameters need to be
59       returned to the calling program, and the calling
60       program can decide what to print. */
61    printf ("The number of rows is %d\n", k);
62    printf ("The value of h is %.12f\n", h);
63    printf ("The number of division points is %d\n",
64         2*two_to_kmin1);
65    return news;
66  }
```

The function prototype `romberg`, described in lines 5–6, has a pointer to the function `(*fnct)` to be integrated. The other parameters are the lower limit `a`, the upper limit `b`, the permissible error `tol`, the maximum number of rows `maxrow` allowed, and a pointer to an integer `*success` that will be true if the integral was successfully calculated. In the implementation, `maxrow` is not allowed to be larger than `MAXROW`, defined to be 30 in line 2 (because `MAXROW` number of locations have been reserved to the array `s[]` holding a column of the matrix of approximation sums $S_{k,j}$). There is an integer `two_to_kmin1` declared in line 8 to hold the current value of 2^{k-1}, and a real `four_to_j` of type double to hold the current value 4^j is declared in line 10. Further, an integer `state` describing the current state of the process, i.e., whether more interval splitting is needed (`state==0`), or the result is within the tolerance defined by `tol` (`state==1`, or whether no more rows are allowed (`state==2`).

In line 3, `F(x)` is defined to be `(*fnct)(x)`; this allows to simplify the writing when the function pointed to by `fnct` needs to be evaluated. In line 11, `*success` assigned 0, and it will stay 0 unless

the test in line 55 shows that the calculation reached `state==1`. If on line 12 it turns out that `maxrow` is larger than `MAXROW`, the calculation is terminated unsuccessfully. The actual calculation starts on line 17 by calculating $S_{0,0}$, and the main part of the calculation is done in the loop between lines 21 and 54. Lines 21–26 calculate $_(k,0)$ using formula (2) (`k` is incremented on line 50), and $S_{k,j}$ is calculated in lines 31–36 using formula (4). Before the calculation, the array element `s[j]` contains $S_{k-1,j}$, this is moved to `olds`, and $S_{k,j}$ is calculated in `news`, and the result is moved into `s[j]`, updating the array `s[]`. As mentioned above, on line 42 it is tested if the new diagonal element is close enough to the old diagonal element; a yes answer will finish the calculation. In lines 50–52 the new diagonal element `news` is stored in `s[k]`, then `k` is incremented in the same statement, and the values of `h` and `two_to_kmin1` are updated.

In line 55, the value of `*success` is set. The printing statements on lines 61–63 have only diagnostic purpose, and in the final version of the program they can be commented out (they should probably still be left there, since they might be needed temporarily when the program is maintained or updated). Line 65 returns the value of the integral. The calling program is contained in the file `main.c`:

```
 1 #include "romberg.h"
 2
 3 main()
 4 {
 5   double integr, tol=5e-13;
 6   int success;
 7   integr = romberg(&funct,0.0,1.0, tol, 26, &success);
 8   if ( success ) {
 9     printf("The integral is %.12f\n", integr);
10     printf("The calculated value of pi is %.12f\n", 4.0*integr);
11     printf("The actual value of pi is     %.12f\n", 4.0*atan2(1,1));
12     printf("The tolerance used is         %.12g\n", tol);
13   }
14   else {
15     printf("It is not possible to calculate the integral "
16            "with the\n    required precision\n");
17   }
18 }
```

The tolerance `tol` is set to be $5 \cdot 10^{-13}$ and in line the romberg integration routine is called with limits 0 and 1, and a maximum 26 rows is allowed. As we mentioned, the integral to be calculated is

$$\int_0^1 \frac{dx}{1+x^2}.$$

The value of this integral is well known to be $\pi/4$. The result of the calculation is compared to the value of π obtained as $4 \arctan 1$ calculated on line 11. The printout of the program is

```
1 The number of rows is 7
2 The value of h is 0.007812500000
3 The number of division points is 128
4 The integral is 0.785398163397
5 The calculated value of pi is 3.141592653590
6 The actual value of pi is     3.141592653590
7 The tolerance used is         5e-13
```

This shows that all 12 decimal digits of the result (i.e., four times the integral) agree with the value of π.

Problems

1. Write the expression for $S_{3,3}$ to evaluate the integral

$$\int_0^8 f(x)\,dx$$

using Romberg integration.

Solution. We have $h_0 = 8$, and

$$S_{0,0} = \frac{h_0}{2}(f(0) + f(8)) = 4(f(0) + f(8)).$$

Further,

$$S_{1,0} = 2(f(0) + 2f(4) + f(8)),$$

$$S_{1,1} = \frac{4S_{1,0} - S_{0,0}}{3} = \frac{4}{3}(f(0) + 4f(4) + f(8)),$$

$$S_{2,0} = f(0) + 2f(2) + 2f(4) + 2f(6) + f(8),$$

$$S_{2,1} = \frac{4S_{2,0} - S_{1,0}}{3} = \frac{2}{3}(f(0) + 4f(2) + 2f(4) + 4f(6) + f(8))$$

$$S_{2,2} = \frac{16S_{2,1} - S_{1,1}}{15} = \frac{1}{45}(28f(0) + 128f(2) + 48f(4) + 128f(6) + 28f(8)),$$

$$S_{3,0} = \frac{1}{2}(f(0) + 2f(1) + 2f(2) + 2f(3) + 2f(4) + 2f(4) + 2f(6) + 2f(7) + f(8)),$$

$$S_{3,1} = \frac{4S_{3,0} - S_{2,0}}{3} = \frac{1}{3}(f(0) + 4f(1) + 2f(2) + 4f(3) + 2f(4) + 4f(5) + 2f(6) + 4f(7) + f(8)),$$

$$S_{3,2} = \frac{1}{45}(14f(0) + 64f(1) + 24f(2) + 64f(3) + 28f(4) + 64f(5) + 24f(6) + 64f(7) + 14f(8)),$$

$$\begin{aligned} S_{3,3} = \frac{64S_{3,2} - S_{2,2}}{63} = \frac{1}{2835}(&868f(0) + 4096f(1) + 1408f(2) + 4096f(3) \\ &+ 1744f(4) + 4096f(5) + 1408f(6) + 4096f(7) + 868f(8)). \end{aligned}$$

2. We would like to determine the integral

$$\int_0^1 \sqrt{x^3 + x}\,dx$$

Explain why Romberg integration would not work. Explain why adaptive integration would be a better choice.

Solution. The integrand behaves badly near $x = 0$ in that all its derivatives become infinite at $x = 0$. So no integration method is likely to work that divides the interval into equal parts. Near $x = 0$ one needs to divide the interval into much smaller parts than away from 0 because the error terms of both the trapezoidal rule and Simpson's rule becomes infinite at 0. Even adaptive integration does not work unless one excludes a small neighborhood of $x = 0$.

3. Even adaptive integration will have trouble with the integral in the preceding problem on the interval $[0, 1]$, but one can successfully use the adaptive trapezoidal rule to calculate

$$\int_{10^{-5}}^{1} \sqrt{x^3 + x}\, dx = 1.847,750,122,11$$

with an error less than $2.5 \cdot 10^{-10}$. Explain how one can use Taylor's formula to approximate the missing part of the integral

$$\int_{0}^{10^{-5}} \sqrt{x^3 + x}\, dx$$

with the same error $2.5 \cdot 10$, and use this result to find

$$\int_{0}^{1} \sqrt{x^3 + x}\, dx$$

with an error less than $5 \cdot 10^{-10}$.

Solution. We have

$$\sqrt{x^3 + x} = \sqrt{x}\sqrt{1 + x^2} = \sqrt{x}\left(1 + \frac{x^2}{2} + O(x^4)\right),$$

where we took the Taylor expansion of $\sqrt{1 + x^2}$ at $x = 0$. When integrating this on the interval $[0, 10^{-5}]$, we can ignore $x^2/2$ in the Taylor expansion, since its maximum of the corresponding term $\sqrt{x} \cdot x^2/2 = x^{5/2}/2$ in the integrand on this interval is $10^{-25/2} < 10^{-12}$, and when we multiply this by the length of the interval of integration, i.e., 10^{-5}, the result is $< 10^{-17}$, which is much smaller that the permissible error of error is $< 2.5 \cdot 10^{-10}$.[75] Thus

$$\int_{0}^{10^{-5}} \sqrt{x^3 + x} \approx \int_{0}^{10^{-5}} \sqrt{x}\, dx = \frac{2}{3} 10^{-15/2} = 3.152 \cdot 10^{-8}$$

Hence

$$\int_{0}^{1} \sqrt{x^3 + x}\, dx \approx 1.847,750,122,11 + 0.000,000,031,52 = 1.847,750,153.63.$$

4. Briefly explain in what situation would you prefer adaptive integration over Romberg integration, and conversely, when would you prefer Romberg integration over adaptive integration.

[75] For the sake of simplicity, we used $O(x^4)$ instead of the remainder term of the Taylor formula. An exact proof showing that the contribution of the remaining terms is too small to matter would need to use the formula for the remainder term, or some other way of estimating the error. The remainder of the Taylor formula would be

$$\frac{f''(\xi)}{2} t^2$$

with $f(t) = \sqrt{1 + t}$ and $t = x^2$ (it is technically simpler to calculate the Taylor series of $\sqrt{1 + t}$ and at $t = 0$ and then take $t = x^2$ than to calculate the Taylor series of $\sqrt{1 + x^2}$, but the result will of course be the same); it is fairly easy to see that $f''(\xi)$ is very close to 1 when ξ is in the interval $[0, 10^{-10}]$, the interval to which $t = x^2$ belongs when x belongs to the interval $[0, 10^{-5}]$.

26. INTEGRALS WITH SINGULARITIES

Improving singularities by integration by parts. On occasion, one needs to evaluate integrals with singularities, such as

$$I = \int_0^{\pi/2} \frac{\sin x\, dx}{x^{3/2}}.$$

The trouble is that the usual numerical integration methods do not work well in this situation, since the integrand tends to infinity at $x = 0$. The situation is even worse for the derivatives of the integral; for example, the fourth derivative, occurring in the error estimate for Simpson's rule, tends to infinity as fast as $x^{-9/2}$ at $x = 0$.[76]

One can improve the situation by integration by parts. Indeed, integration by parts gives[77]

$$\begin{aligned} I &= \int_0^{\pi/2} x^{-3/2} \sin x\, dx = \lim_{\epsilon \searrow 0} \left[-2x^{-1/2} \sin x \right]_{x=\epsilon}^{\pi/2} + \int_0^{\pi/2} 2x^{-1/2} \cos x\, dx \\ &= -\sqrt{8/\pi} + \int_0^{\pi/2} 2x^{-1/2} \cos x\, dx. \end{aligned}$$

The integral on the right is just as bad as the as the integral on the left, but another integration by parts will indeed improve the situation:

$$I = -\sqrt{8/\pi} + 4x^{1/2} \cos x \Big|_{x=0}^{\pi/2} + \int_0^{\pi/2} 4x^{1/2} \sin x\, dx = -\sqrt{8/\pi} + \int_0^{\pi/2} 4x^{1/2} \sin x\, dx;$$

the last equation holds, since this time the contribution of the integrated-out part is zero.

The situation is now better, since the integrand on the right-hand tends to zero as fast as $x^{3/2}$ when x tends to zero. So, one can say that the integral no longer has a singularity. However, the fourth derivative at $x = 0$ still tends to infinity as fast as $x^{-5/2}$, so Simpson's rule is still not applicable; even the trapezoidal rule is not applicable, since the second derivative tends to infinity at $x = 0$. Two more integration by parts are needed to make the trapezoidal rule applicable, and four more integration by parts are needed to make Simpson's rule applicable.[78]

Subtraction of singularity. Performing repeated integrations by parts is laborious, and it is simpler to use another method, called the *subtraction of singularity*. Using the Taylor expansion

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^7),$$

[76] This is because the integrand tends to infinity at $x = 0$ as fast as $x^{-1/2}$, and the fourth derivative of this is constant times $x^{-9/2}$.

[77] In order to deal with the singularity at $x = 0$, instead of $x = 0$ at the lower limit we take ϵ tending to 0 from the right. In the integration part, the factor $x^{-3/2}$ will be integrated, and the factor $\sin x$ will be differentiated. Finally, since we have

$$\lim_{x \to 0} \frac{\sin x}{x} = 1,$$

we have

$$\lim_{\epsilon \searrow 0} \epsilon^{-1/2} \sin \epsilon = 0.$$

[78] This latter statement is true whether one uses the error term involving the fourth derivative that we obtained for Simpson's rule, or whether one uses a less precise error term involving the third derivative for Simpson's rule.

convergent on the whole real line, where $O(x^7)$ is used to indicate the error when $x \to 0$, we have

$$\int_0^{\pi/2} x^{-3/2} \sin x \, dx = \int_0^{\pi/2} x^{-3/2} \left(\sin x - x + \frac{x^3}{6} - \frac{x^5}{120} \right) dx + \int_0^{\pi/2} \left(x^{-1/2} - \frac{x^{3/2}}{6} + \frac{x^{7/2}}{120} \right) dx.$$

Here, the first integral can be calculated by Simpson's rule; indeed, at $x = 0$, the integrand behaves as $x^{-3/2} \cdot O(x^7) = O(x^{11/2})$, so its fourth derivative behaves as $O(x^{11/2-4}) = O(x^{3/2})$, so the fourth derivative tends to zero when x tends to zero.[79] The second integral can be calculated directly, without the need of numerical integration.[80]

When one wants to evaluate the integral

$$I = \int_0^2 \frac{\arctan x \, dx}{x^{3/2}},$$

repeated integration by parts would still work to prepare the integral for numerical evaluation, but it would be much more laborious than even in the previous example, since the repeated derivatives of $\arctan x$ look much more complicated than the repeated derivatives of $\sin x$. On the other hand, subtraction of singularity works fairly simply. Using the Maclaurin expansion[81]

$$\arctan x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{2n+1} = x - \frac{x^3}{3} + \frac{x^5}{5} + O(x^7),$$

convergent on the interval $(-1, 1]$ (only conditionally at $x = 1$), we have

$$\int_0^2 x^{-3/2} \arctan x \, dx = \int_0^2 x^{-3/2} \left(\arctan x - x + \frac{x^3}{3} - \frac{x^5}{5} \right) dx + \int_0^2 \left(x^{-1/2} - \frac{x^{3/2}}{3} + \frac{x^{7/2}}{5} \right) dx.$$

the first integral can be evaluated by Simpson's rule, and the second one directly, without the use of numerical methods.[82]

There are also numerical methods that can deal with singularities, but removing the singularities by one of the methods outlined above is usually more accurate.

Infinite intervals of integration. There are numerical methods that work for integrals on infinite intervals, but it is usually more accurate to transform the integral to a finite interval of integration. For example, the integral

$$I = \int_0^{\infty} e^{-x^{3/2}} \, dx$$

[79] One needs to be careful here: just because a function behaves as $O(x^2)$ near $x = 0$, it does not follow that its derivative behaves as $O(x)$. Consider for example the function $f(x) = x^2 \sin \frac{1}{x}$ when $x \neq 0$ and $f(0) = 0$. This function is differentiable everywhere, but its derivative does not tend to 0 when $x \to 0$. The argument here works only because we are considering functions of the form $f(x) = x^\alpha g(x)$, where α is a real number, and the function $g(x)$ has a convergent Taylor series near 0. More generally, the argument also works for functions of form $f(x) = x^\alpha g(x^\beta)$ where α and β are reals, $\beta > 0$, and $g(x)$ can be approximated by Taylor polynomial (i.e., by a finite number of terms of the Taylor series) near 0 – the Taylor series need not even be convergent anywhere.

[80] Note that the integral $\int_0^{\pi/2} x^{-1/2} \, dx$ is a convergent improper integral.

[81] i.e., the Taylor expansion at $x = 0$.

[82] The fact that the Maclaurin series of $\arctan x$ does not converge on the whole interval [0.2] of integration has no importance here; the only important point is that the Maclaurin converges near $x = 0$. In fact, one can even weaken this condition, and consider functions whose Taylor series never converges, but in this case it is still important that the first few terms of the Taylor series approximate the function well near 0, and the first few derivatives of the function can similarly be approximated by the appropriate derivatives of the Taylor series.

can be handled by first splitting the interval of integration as

$$I = I_1 + I_2 = \int_0^1 e^{-x^{3/2}}\,dx + \int_1^\infty e^{-x^{3/2}}\,dx$$

While first integral is continuous at $x = 0$, its derivatives (beginning with the second derivative) tend to infinity when $x \to 0$, so we need to prepare this integral for numerical integration. This can be done by the subtraction of singularity. We have the Maclaurin series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + O(x^3),$$

and so

$$e^{-x^{3/2}} = 1 - x^{3/2} + \frac{x^3}{2} + O(x^{9/2}).$$

Hence we can write

$$I_1 = \int_0^1 e^{-x^{3/2}}\,dx = \int_0^1 \left(e^{-x^{3/2}} - 1 + x^{3/2} - \frac{x^3}{2}\right)\,dx + \int_0^1 \left(1 - x^{3/2} + \frac{x^3}{2}\right)\,dx.$$

The first integral on the right-hand side can be handled by Simpson's rule, and the second integral can be evaluated directly.

As for the second integral above, we can use the substitution $t = 1/x$ to obtain

$$I_2 = \int_1^\infty e^{-x^{3/2}}\,dx = -\int_1^0 t^{-2} e^{-t^{-3/2}}\,dt = \int_0^1 t^{-2} e^{-t^{-3/2}}\,dt\,;$$

indeed, when substituting, we have $x = t^{-1}$, so $dx = -t^{-2}\,dx$; the lower limit 1 stays the same, and the upper limit ∞ moves to 0. The integral on the right-hand side behaves well; in fact, the integrand and all its derivatives tend to 0 when $t \searrow 0$. Hence this integral can be handled even by Romberg integration (which is more efficient than Simpson's rule if the integrand behaves nicely).[83]

27. NUMERICAL SOLUTION OF DIFFERENTIAL EQUATIONS

The initial value problem for the differential equation

$$y' = f(x, y)$$

is to find a function ϕ such that

$$\phi'(x) = f(x, \phi(x))$$

satisfying a given initial condition $\phi(x_0) = y_0$ for some given numbers x_0 and y_0. To simplify the notation, one often write $y(x)$ instead of $\phi(x)$. One usually looks for the solution in a neighborhood of the point x_0. One of the simplest method of approximating the solution is Taylor's method, which consists in approximating the solution $y = y(x)$ with its Taylor series at x_0:

$$y(x) \approx \sum_{k=0}^{n} \frac{y^{(k)}(x_0)}{k!}(x - x_0)^k.$$

[83]For $t = 0$, the integrand needs to be taken to be 0.

The coefficient's here involve higher derivatives of y at x_0, but this is easy to calculate using implicit differentiation (provided that f is differentiable sufficiently many times).

For example, if one wants to solve the differential equation

$$y' = \sqrt{x+y^2}, \qquad y(3) = 1,$$

one can differentiate to obtain

$$y'' = \frac{1+2yy'}{2\sqrt{x+y^2}} = \frac{1+2y\sqrt{x+y^2}}{2\sqrt{x+y^2}} = \frac{1}{2\sqrt{x+y^2}} + y,$$

where the second equality was obtained by substituting y' from the original differential equation. We can differentiate the right-hand side again to obtain

$$y''' = -\frac{1}{4}(1+2yy')(x+y^2)^{-3/2} + y' = -\frac{1}{4}(x+y^2)^{-3/2} - \frac{y}{2}(x+y^2)^{-1} + (x+y^2)^{1/2}.$$

Substituting $x = 3$, in which case $y = 1$ according to the initial condition, we obtain

$$y'(3) = 2, \qquad y''(3) = \frac{5}{4}, \qquad y'''(3) = \frac{59}{32}.$$

Hence

$$y(x) \approx y(3) + y'(3)(x-3) + \frac{y''(3)}{2}(x-3)^2 + \frac{y'''(3)}{6}(x-3)^3 = 1 + 2(x-3) + \frac{5}{8}(x-3)^2 + \frac{59}{192}(x-3)^3.$$

One can use this equation to determine $y(x)$ at a point close to $x = 3$, say at $x = 3.1$. If one wants to determine $y(4)$, one might divide the interval $(3, 4)$ into parts, say into ten equal parts, and then determine $y(3.1)$ first. Then writing a similar equation at $x = 3.1$, one can determine $y(3.2)$, etc. Repeating this step ten times, one can obtain an approximation to $y(4)$. One can even estimate the error of this approximation using the remainder term of Taylor's formula.

We will include a short discussion of Taylor's method in general. We have $y' = f(x, y)$, and using the (two-variable) chain-rule

$$y'' = f_x(x,y) + f_y(x,y)y' = f_x(x,y) + f_y(x,y)f(x,y) = f_x + f_y f,$$

where f_x and f_y denote the partial derivatives of f. The second equality uses the differential equation $y' = f(x, y)$. The calculations are easier to follow if one suppresses the arguments of f (i.e., if one writes f instead of $f(x, y)$, f_x instead of $f_x(x, y)$, etc.). Differentiating once more, we have

$$\begin{aligned} y''' &= (f_x + f_y f)_x + (f_x + f_y f)_y y' = (f_{xx} + f_{yx}f + f_y f_x) + (f_{xy} + f_{yy}f + f_y f_y)y' \\ &= (f_{xx} + f_{yx}f + f_y f_x) + (f_{xy} + f_{yy}f + f_y f_y)f = f_{xx} + 2f_{xy}f + f_x f_y + f_{yy}f^2 + f_y^2 f, \end{aligned}$$

where for the third equality we used the equation $y' = f$, and for the fourth equality we used the fact that the order the mixed derivatives are taken is irrelevant.[84] Thus, writing $h = x_1 - x_0$, for h close to 0 we have

$$y_1 = y_0 + hf + \frac{h^2}{2}(f_x + f_y f) + \frac{h^3}{6}(f_{xx} + 2f_{xy}f + f_x f_y + f_{yy}f^2 + f_y^2 f) + O(h^4), \tag{1}$$

[84]The equation $f_{xy}(x_0, y_0) = f_{xy}(x_0, y_0)$ is valid if, for example, f_{xy} and f_{xy} exist at (x_0, y_0) and the first derivatives f_x and f_y are continuous at (x_0, y_0).

where $y_1 = y(x_1)$, and f and its derivatives are evaluated at (x_0, y_0) on the right-hand side. This formula can be continued by calculating the fourth and higher derivatives of y.

One can obtain a general expression for $y^{(n)}$ in terms of differential operators. Namely, assuming that the function $y = y(x)$ satisfies the above differential equation, that is $y' = f(x, y)$, then for the derivative of an arbitrary function $g(x, y)$ we can write

$$\frac{d}{dx}g(x, y(x)) = \frac{\partial}{\partial x}g(x, y) + \frac{dy(x)}{dx}\frac{\partial}{\partial y}g(x, y) = \frac{\partial}{\partial x}g(x, y) + f(x, y)\frac{\partial}{\partial y}g(x, y) = \left(\frac{\partial}{\partial x} + f\frac{\partial}{\partial y}\right)g$$

where, for better readability, arguments are suppressed at some places.[85] Since g was an arbitrary function, in terms of differential operators, this means

$$\frac{d}{dx} = \frac{\partial}{\partial x} + f\frac{\partial}{\partial y}.$$

Therefore, for $n \geq 1$ we have

$$\frac{d^n y}{dx} = \left(\frac{d}{dx}\right)^{(n-1)} f(x, y) = \left(\frac{\partial}{\partial x} + f\frac{\partial}{\partial y}\right)^{n-1} f$$

While this formula can speed up the evaluation of higher-order derivatives of y, it is is not as useful as one might think at first time. Namely, one cannot use the Binomial Theorem to evaluate the power of the differential operator on the right-hand side. The proof of the Binomial Theorem expressing $(A+B)^n$ depends on the commutativity $AB = BA$, and these differential operators do not commute:

$$\frac{\partial}{\partial x}\left(f\frac{\partial}{\partial y}\right)g = f_x g_y + f g_{yx} = f_x g_y + f g_{xy}, \qquad \text{while} \qquad \left(f\frac{\partial}{\partial y}\right)\frac{\partial}{\partial x}g = f g_{xy},$$

Problems

1. Write a third order Taylor approximation at $x = 0$ for the solution of the differential equation $y' = x + y$ with initial condition $y(0) = 2$.

Solution. We have $y(0) = 2$, $y'(0) = x + y = 2$; the right-hand side was obtained by substituting $x = 0$ and $y = 2$. Differentiating and again substituting $x = 0$ and $y = 2$, we obtain

$$y''(x) = 1 + y' = 1 + x + y = 3.$$

Differentiating and substituting $x = 0$ and $y = 2$, we again obtain

$$y'''(x) = 1 + y' = 1 + x + y = 3.$$

[85] Certain conditions are needed in order that the the chain rule be valid. Namely, for the above formula to be valid for $x = x_0$ and $y_0 = y(x_0)$ the function $y(x)$ needs to be differentiable at x_0 and $g(x, y)$ needs to be differentiable at (x_0, y_0). This latter condition means that there are numbers A and B

$$g(x_0 + h, y_0 + h) - g(x_0, y_0) = Ah + Bk + E(h, k)$$

where $E(h, k)$ is a function such that

$$\lim_{\substack{h\to 0\\ k\to 0}} \frac{E(h, k)}{|h| + |k|} = 0.$$

For the first of these equations to hold, we must have $A = g_x(x_0, y_0)$ and $B = g_y(x_0, y_0)$. Furthermore, a sufficient condition for the differentiability of g a (x_0, y_0) is that partial derivatives g_x and g_y be continuous at (x_0, y_0).

$$y(x) = y(0) + y'(0)x + y''(0)\frac{x^2}{2} + y'''(0)\frac{x^3}{6} + O(x^4) = 2 + 2x + 3\frac{x^2}{2} + 3\frac{x^3}{6} + O(x^4)$$
$$= 2 + 2x + \frac{3}{2}x^2 + \frac{1}{2}x^3 + O(x^4)$$

for x near 0.

2. Write a third order Taylor approximation at $x = 0$ for the solution of the differential equation $y' = \sin x + \cos y$ with initial condition $y(0) = 1$.

Solution. We have $y(0) = 1$, $y'(0) = \sin 0 + \cos 1 \approx 0.540,302$

$$y''(x) = \cos x - y' \sin y = \cos x - (\sin x + \cos y) \sin y.$$

Substituting $x = 0$ and $y = 1$ we obtain $y''(0) \approx 0.545,351$. Differentiating again, we obtain

$$y'''(x) = -\sin x - (\cos x - y' \sin y) \sin y - (\sin x + \cos y)y' \cos y$$
$$= -\sin x - (\cos x - (\sin x + \cos y) \sin y) \sin y - (\sin x + \cos y)(\sin x + \cos y) \cos y$$
$$- \sin x - \cos x \sin y + \sin x \sin^2 y + \cos y \sin^2 y - \sin^2 x \cos y - 2 \sin x \cos^2 y - \cos^3 y$$

Substituting $x = 0$ and $y = 1$ we obtain $y'''(x) \approx -0.616,626$. Thus,

$$y(x) = 1 + 0.540,302x + 0.545,351\frac{x^2}{2} - 0.616,626\frac{x^3}{6} + O(x^4)$$
$$= 1 + 0.540,302x + 0.272,676\frac{x^2}{2} - 0.102,771\frac{x^3}{6} + O(x^4)$$

for x near 0.

28. RUNGE-KUTTA METHODS

The difficulty with Taylor's method is that it involves a substantial amount of symbolic calculation, a task usually more complicated for computers than direct numerical calculation. Runge-Kutta methods seek to remedy this weakness of Taylor's method in that they achieve the same precision without using any symbolic calculations. Given the differential equation

$$y = f(x, y), \qquad y(x_0) = y_0,$$

one seeks to determine the solution $y_1 = y(x_1)$ at a nearby point $x_1 = x_0 + h$ in the form

$$y_1 = y_0 + hK, \tag{1}$$

where K is sought in the form

$$K = \sum_{i=1}^{r} \gamma_i K_i, \tag{2}$$

where

$$K_1 = f(x_0, y_0), \tag{3}$$

and

$$K_i = f\Big(x_0 + h\alpha_i, y_0 + h\sum_{j=1}^{i-1} \beta_{ij} K_j\Big). \tag{4}$$

In the last equation, we have $\alpha_1 = 0$, so as to be consistent with equation (3). One seeks to determine the coefficients γ_i, α_i, and β_{ij} in such a way as to achieve a high degree of agreement with Taylor's formula

$$y_1 = y(0) + hy'(0) + \frac{h^2}{2} y''(0) + \dots .$$

The method described above (with appropriately chosen coefficients) is called an r-stage Runge-Kutta method. If an agreement with the Taylor formula up to and including the term h^m is achieved, then m is called the order of the method. For a given m and r, appropriate coefficients cannot always be found. A necessary (but not sufficient) condition that appropriate coefficients can be found is $m \le r$ (we will not prove this).

The error committed in calculating the true value of $y(x_1)$, that is $y(x_1) - y_1$ is called the *local truncation error*. So the local truncation error of an m order method is $O(h^{m+1})$. The *global* truncation error is the error committed after a number of steps. For example, if one wants to calculate the solution on an interval (a, b), one might divide this interval into n equal parts and taking $h = (b-a)/n$, and taking n steps to calculate the value at b from the known value at a. The error committed in n steps is about n times the local truncation error. Thus, if the local truncation error is $O(h^{m+1})$, then global truncation error is $nO(h^{m+1}) = O(h^m)$.[86]

We will discuss the equations that the requirement of an agreement with the Taylor series method imposes on the above coefficients. However, if one wants to obtain a Runge-Kutta method along these lines that are useful in practice, the equations that one needs to solve are quite complicated; hence we are only going to present the basic principles involved.

In order to describe the equations for the coefficients involved, we need to express the quantities K_i in terms of the two dimensional Taylor series. According to this

$$F(x_0+h, y_0+k) = \sum_{i=0}^{n} \frac{1}{i!} \left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^i F(x,y)\Big|_{\substack{x=x_0\\y=y_0}} + \frac{1}{(n+1)!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^{n+1} F(x,y)\Big|_{\substack{x=x_0+\theta h\\y=y_0+\theta k}}.$$

This formula can easily be derived by writing $G(t) = F(x_0 + th, y_0 + tk)$, and expressing $G(1)$ in terms of the Taylor formula for G at $t = 0$. The remainder term involves the value of $G(\theta)$ for some θ in the interval $(0, 1)$.[87] The derivatives after the sum sign can be evaluated by using the Binomial Theorem.[88] For example,

$$\begin{aligned}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^3 F &= \left(h\frac{\partial}{\partial x}\right)^3 F + 3\left(h\frac{\partial}{\partial x}\right)^2 k\frac{\partial}{\partial y}F + 3h\frac{\partial}{\partial x}\left(k\frac{\partial}{\partial y}\right)^2 F + \left(h\frac{\partial}{\partial y}\right)^3 F \\ &= h^3\frac{\partial^3 F}{\partial x^3} + 3h^2k\frac{\partial^3 F}{\partial x^2\partial y} + 3hk^2\frac{\partial^3 F}{\partial x\partial y^2} + k^3\frac{\partial^3 F}{\partial y^3} = h^3F_{xxx} + 3h^2kF_{xxy} + 3hk^2F_{xyy} + k^3F_{yyy}.\end{aligned}$$

[86] This is not strictly true, since the errors do not add up: earlier errors may be magnified in subsequent calculations. In an unstable method, earlier errors may cause excessive global errors even though the local errors (truncation errors and round-off errors) may be small; for a stable method this is not supposed to happen. The Runge-Kutta methods are quite stable.

[87] It is customary to use θ for an unknown number between $(0, 1)$. Usually, one used ξ in the remainder term of the single-variable Taylor formula, and θ in the multi-variable Taylor formula. We used the Lagrange remainder term of the single-variable Taylor formula to obtain the multi-variable remainder term. Other single-variable remainder terms can be used to obtain different multi-variable remainder terms.

[88] The proof of the Binomial Theorem relies on the commutativity of multiplication. That is, it relies on the fact that $\frac{\partial}{\partial x}\frac{\partial}{\partial y} = \frac{\partial}{\partial y}\frac{\partial}{\partial x}$. This is valid if the function being differentiated is sufficiently nice, so that the mixed derivatives do not depend on the order of differentiation.

Using this formula with $n = 2$ for K_i above, we have

$$K_i = f\Big(x_0 + h\alpha_i, y_0 + h\sum_{j=1}^{i-1}\beta_{ij}K_j\Big) = f + \Big(f_x\alpha_i + f_y\sum_{j=1}^{i-1}\beta_{ij}K_j\Big)h$$
$$+ \frac{1}{2}\Big(f_{xx}\alpha_i^2 + 2f_{xy}\alpha_i\sum_{j=1}^{i-1}\beta_{ij}K_j + f_{yy}\Big(\sum_{j=1}^{i-1}\beta_{ij}K_j\Big)^2\Big)h^2 + O(h^3),$$

where the derivatives are taken at the point (x_0, y_0), and the symbol $O(h^3)$ is considered in a neighborhood of $h = 0$. The right-hand side here can be further expanded by considering the Taylor expansion of the K_j on the right-hand side, using the same formula that we just gave for K_i:

$$K_j = f + \Big(f_x\alpha_j + f_y\sum_{l=1}^{j-1}\beta_{jl}K_l\Big)h + O(h^2) = f + \Big(f_x\alpha_j + f_y\sum_{l=1}^{j-1}\beta_{jl}f\Big)h + O(h^2),$$

where, in the second member of these equalities, the expansion $K_l = f + O(h)$ was taken to obtain the right-hand side. Substituting this expansion into the expansion of K_i (in the terms multiplied by h^2, we only need to take $K_j = f + O(h)$), we obtain

$$K_i = f + \left(f_x\alpha_i + f_y\sum_{j=1}^{i-1}\beta_{ij}\Big(f + \Big(f_x\alpha_j + f_y\sum_{l=1}^{j-1}\beta_{jl}f\Big)h\Big)\right)h$$
$$+ \frac{1}{2}\Big(f_{xx}\alpha_i^2 + 2f_{xy}\alpha_i\sum_{j=1}^{i-1}\beta_{ij}f + f_{yy}\Big(\sum_{j=1}^{i-1}\beta_{ij}f\Big)^2\Big)h^2 + O(h^3)$$
$$= f + \Big(f_x\alpha_i + ff_y\sum_{j=1}^{i-1}\beta_{ij}\Big)h + \Big(f_xf_y\sum_{j=1}^{i-1}\alpha_j\beta_{ij} + ff_y^2\sum_{j=1}^{i-1}\sum_{l=1}^{j-1}\beta_{ij}\beta_{jl}$$
$$+ \frac{1}{2}f_{xx}\alpha_i^2 + ff_{xy}\alpha_i\sum_{j=1}^{i-1}\beta_{ij} + \frac{1}{2}f^2f_{yy}\Big(\sum_{j=1}^{i-1}\beta_{ij}\Big)^2\Big)h^2 + O(h^3).$$

Substituting this into (1) and (2), we obtain

$$y_1 = y_0 + fh\sum_{i=1}^{r}\gamma_i + \Big(f_x\sum_{i=1}^{r}\gamma_i\alpha_i + ff_y\sum_{i=1}^{r}\gamma_i\sum_{j=1}^{i-1}\beta_{ij}\Big)h^2 +$$
$$\Big(f_xf_y\sum_{i=1}^{r}\gamma_i\sum_{j=1}^{i-1}\alpha_j\beta_{ij} + ff_y^2\sum_{i=1}^{r}\gamma_i\sum_{j=1}^{i-1}\sum_{l=1}^{j-1}\beta_{ij}\beta_{jl} + \frac{1}{2}f_{xx}\sum_{i=1}^{r}\gamma_i\alpha_i^2 + ff_{xy}\sum_{i=1}^{r}\gamma_i\alpha_i\sum_{j=1}^{i-1}\beta_{ij}$$
$$+ \frac{1}{2}f^2f_{yy}\sum_{i=1}^{r}\gamma_i\Big(\sum_{j=1}^{i-1}\beta_{ij}\Big)^2\Big)h^3 + O(h^4).$$

To obtain equations for the coefficients, one may compare this equation with equation (1) in Section 27 concerning Taylor's method on the Numerical solution of Differential Equations. Since the present equation must hold for every (small) h and for every f, the coefficients of fh, f_xh^2, ff_yh^2, $f_xf_yh^3$, $ff_y^2h^3$, $f_{xx}h^3$, $ff_{xy}h^3$, and $f^2f_{yy}h^3$ must agree. We obtain

$$\sum_{i=1}^{r}\gamma_i = 1, \quad \sum_{i=1}^{r}\gamma_i\alpha_i = \frac{1}{2}, \quad \sum_{i=1}^{r}\gamma_i\sum_{j=1}^{i-1}\beta_{ij} = \frac{1}{2}, \quad \sum_{i=1}^{r}\gamma_i\sum_{j=1}^{i-1}\alpha_j\beta_{ij} = \frac{1}{6}, \quad \sum_{i=1}^{r}\gamma_i\sum_{j=1}^{i-1}\sum_{l=1}^{j-1}\beta_{ij}\beta_{jl} = \frac{1}{6},$$
$$\sum_{i=1}^{r}\gamma_i\alpha_i^2 = \frac{1}{3}, \quad \sum_{i=1}^{r}\gamma_i\alpha_i\sum_{j=1}^{i-1}\beta_{ij} = \frac{1}{3}, \quad \sum_{i=1}^{r}\gamma_i\Big(\sum_{j=1}^{i-1}\beta_{ij}\Big)^2 = \frac{1}{3}.$$

More equations can be obtained if more terms of the to Taylor expansions are calculated and compared. In the classical Runge-Kutta method one has

$$\begin{aligned}
K_1 &= f(x_0, y_0),\\
K_2 &= f(x_0 + h/2, y_0 + hK_1/2),\\
K_3 &= f(x_0 + h/2, y_0 + hK_2/2),\\
K_4 &= f(x_0 + h, y_0 + hK_3),\\
y_1 &= y_0 + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4).
\end{aligned}$$

This is fourth order method.[89] The disadvantage of this method is that it is difficult to estimate the error of the approximation of y_1 to the true solution of the differential equation.

One way of estimating the error of a Runge-Kutta approximation is to use the same method with step size h and step size $h/2$ and compare the results, but this involves too many function evaluations. A better way is to use two Runge-Kutta approximations with the same step size, one method of order m and another of order $m + 1$. The higher order method then calculates the approximation with much greater precision, so the difference of the two approximations can be taken as the error. If the error is too large, one should use a smaller step size. Using two different methods normally still involves too much calculation. However, there are pairs of Runge-Kutta methods where most of the calculations used in making the lower order step can be re-used form making the higher order step. In the Runge-Kutta-Fehlberg method, one has a fourth-order method combined with a fifth order method. In the fourth order method, one puts

$$\begin{aligned}
K_1 &= f(x_0, y_0),\\
K_2 &= f(x_0 + h/4, y_0 + hK_1/4),\\
K_3 &= f(x_0 + 3h/8, y_0 + h(3K_1 + 9K_2)/32),\\
K_4 &= f(x_0 + 12h/13, y_0 + h(1932K_1 - 7200K_2 + 7296K_3)/2197),\\
K_5 &= f(x_0 + h, y_0 + h(439K_1/216 - 8K_2 + 3680K_3/513 - 845K_4/4104)),\\
y_1 &= y_0 + h(25K_1/216 + 1408K_3/2565 + 2197K_4/4104 - K_5/5).
\end{aligned}$$

Here y_1 is the approximation given by the method to the correct value of $y(x_1)$, where $x_1 = x_0 + h$. For the fifth-order method one needs to do only one more function evaluation[90]

$$\begin{aligned}
K_6 &= f(x_0 + h/2, y_0 + h(-8K_1/27 + 2K_2 - 3544k_3/2565 + 1859K_4/4104 - 11K_5/40)),\\
\bar{y}_1 &= y_0 + h(16K_1/135 + 6656K_3/12825 + 28561K_4/56430 - 9K_5/50 + 2K_6/55),
\end{aligned}$$

where we wrote $\bar{y}_1$ for the approximation given by the method to the correct value of $y(x_1)$. For the local truncation error of the former method we have $y(x_1) - y_1 = O(h^5)$, while for that of the latter method we have $y(x_1) - \bar{y}_1 = O(h^6)$. Since the latter is much smaller than the former, one can take $\bar{y}_1 - y_1$ as a good estimate for the $y(x_1) - y_1$. This gives the following estimate for the error

$$E = h(K_1/360 - 128K_3/4275 - 2197K_4/75240 + k5/50 + 2K_6/55).$$

[89] This means, in particular, that the equations for the coefficients we listed above are not sufficient to derive this method. The above equations reflect only agreement with the Taylor expansion of $y(x)$ with error $O(h^4)$, and we would need agreement with error of $O(h^5)$ to derive a fourth-order method.

[90] In formula (8.52) on p. 282 in [AH], the denominator in the term involving K_4 on the right-hand side of the equation for y_1 is in error. They erroneously give this term as $28561K_4/56437$.

In a practice, one uses the fourth order method with this estimate of the error.[91]

When looking for the solution of a differential equation on an interval (a, b) (i.e., given an initial condition $y(a) = c$, one is looking for $y(b)$), one starts out with an initial step size h, and if the local truncation error is too large, one halves the step size, i.e., takes $h/2$ as the new value of h (perhaps one needs to do this a number of times). If at one point one finds that the local truncation error is too small, then one doubles the step size.

Next we discuss a computer implementation of the Runge-Kutta-Fehlberg method with step size control. The declarations and the included resources are contained in the file `rk45.h`, where `rk45` refers to Runge-Kutta 4th and 5th order:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 5
 6 double funct(double x, double y);
 7 double rk45(double (*fnct)(double,double), double x0,
 8     double y0, double x1, double epsilon, double *calling_h,
 9     double min_h, int *countsucc,
10     int *countfail, int *success);
```

In this file, the definition of the abbreviation `absval(x)` to calculate the absolute value of `x` is given on line 4. Line 6 declares the function `funct`, which stands for the function f in the differential equation $y' = f(x, y)$. On line 7, the function `rk45` is declared to perform the Runge-Kutta-Fehlberg method with step size control. The function itself is described in the file `rk45.c`:

```
 1 #include "rk45.h"
 2 #define F(x,y) ((*fnct)(x,y))
 3
 4 double rk45(double (*fnct)(double,double), double x0,
 5     double y0, double x1, double epsilon, double *calling_h,
 6     double min_h, int *countsucc,
 7     int *countfail, int *success)
 8 {
 9   /* This program implements the Runge-Kutta_Fehlberg
10      method to solve an ordinary differential equation */
11   const double assumedzero=1e-20;
12   double k1,k2,k3,k4,k5,k6,err,err1,x,y=y0,h=*calling_h,
13          epsilonover32=epsilon/32.;
14   int state, shortrange=0;
15     /* state==0: stepping
16        state==1: h is too small
17        state==2: x1 is reached */
18   for ( state=0; !state ;) {
19     k1=F(x,y);
20     k2=F(x+h/4.,y+h*k1/4.);
21     k3=F(x+3.*h/8.,y+h*(3.*k1+9.*k2)/32.);
22     k4=F(x+12.*h/13.,y+h*(1932.*k1-7200.*k2+7296.*k3)/2197.);
23     k5=F(x+h,y+h*(439.*k1/216.-8.*k2+3680.*k3/513.
```

[91] One might think that it is better to use the fifth-order method, since it is more accurate than the fourth-order method. However, no error estimate is available for the fifth-order method. While it is true that the error of the fifth could be estimated by the above formula for E, but combining the fifth order method with the error estimate for the fourth order method would lead to numerical instabilities.

```
24                                      -845.*k4/4104.));
25      k6=F(x+h/2.,y+h*(-8.*k1/27.+2.*k2-3544.*k3/2565.
26                                  +1859.*k4/4104.-11.*k5/40.));
27      err1=h*(k1/360.-128.*k3/4275.-2197.*k4/75240.
28                                     +k5/50.+2*k6/55.);
29      err=absval(err1);
30      if ( err>h*epsilon ) {
31        h /= 2.;
32        if ( h<min_h ) { state=1; }
33        (*countfail)++;
34      }
35      else {
36        (*countsucc)++;
37        x += h;
38        y +=h*(25.*k1/216.+1408.*k3/2565.+2197.*k4/4104.-k5/5.);
39        if ( absval(x1-x) <= assumedzero ) state=2;
40        else {
41          if ( err <= h*epsilonover32 ) h *= 2.;
42          if ( h > x1-x ) shortrange=1;
43          if ( shortrange ) {
44            *calling_h=h; h=x1-x;
45          }
46        }
47
48      }
49    }
50    if ( !shortrange ) *calling_h=h;
51    if ( state==2 ) {
52      *success=1;
53      return y;
54    }
55    else *success=0;
56    return 0.0;
57  }
```

The definition of the function `rk45` starts on line 4. Its first parameter is `(*fnct)`, pointer to the function f in the differential equation $y' = f(x, y)$ to be solved, `x0`, the starting point, `y0`, the initial value of the solution, `x1`, the point where the solution is sought (the interval `(x0,x1)` will be divided into smaller parts), `epsilon`, the global error allowed at `x1`, a pointer `calling_h` to the step size the function uses (the starting value `*calling_h` of the location pointed to the by `calling_h` is the initial step size specified by the calling program; when the process finished, this location will contain the value of the step size that was used last). The parameter `min_h` contains the smallest permissible step size (if the step size needed to achieve the required accuracy needs to be smaller, the process is abandoned; the location pointed to by the pointer `countsucc` counts the number of successful steps (i.e., when the result of the calculated for the given step is accepted), while `countfail` points to the location containing the number of failed steps (i.e., where the step needs to be repeated with a smaller step size. The location pointed to by `success` will contain `1` (true) if the value of the solution at `x1` was successfully calculated, and `0` (false) if the calculation was not successful. The function `rk45` returns the calculated value of the solution at `x1`.

In lines 15–17 the comments describe the meaning of the variable `state`: it will be 0 if there are more steps to be calculated, it will be 1 is the step size needed to achieve the required accuracy would be too small), Lines 18–48 contain a look that is performed if `state` is 1 (i.e., stepping),

when successive steps of the method are applied. Inside this loop, in lines 19–25, the quantities `k1` through `k6` (corresponding to K_1 through K_6) are evaluated;[92] in these calculations, `F(x,y)` stands for `(*fnct)(x,y)` according to the definition on line 2, to simplify the notation. In line 27, the error of the fourth-oder method is evaluated, and in line 29 its absolute value is taken. If on line 30 it is found that the local error is too large (i.e., larger that `h*epsilon`, corresponding to $h\epsilon$), then the step size is halved. If the value of `h` thereby becomes less than `min_h`, on line 32 the integer `state` is changed to 1, and the count of failures is incremented on line 33. Otherwise the count of successes is incremented on line 36, that of `x` is incremented by `h` on the next line, and `y`, the value of $y(x)$ at the next step, is calculated on line 38. If the new value of `x` is too close to `x1` (the endpoint of the interval on which the solution is calculated), `state` is changed to 2, indicating that `x1` has been reached, and the calculation is finished (on line 51, based on the value of `state`).

If on line 41 it is found that the error is too small, error step size is doubled. The criterion used for considering the error too small is that it is smaller than `epsilonover32`, corresponding to $\epsilon/32$ (cf. line 13, where `epsilonover32` is initialized). The reason is that a fourth order method is used, so the local truncation error is proportional to h^5, so doubling the step size will result in a 32-fold increase of the error.[93]

If `x` is close to `x1` than the current step size `h`, the integer `shortrange` is changed to 1 on line 42. The next and last step to be taken will be `x1-x` instead of the current step size `h`, but the current value of `h` is preserved at the location `calling_h` on line 44, so that the calling program can take this value from this location. If the function `rk45` is called again to continue the calculation, this value of `h` can be used for further calculations as a starting value.

The loop ends on line 48, and in line 48, if the last step taken was not too short (i.e., `shortrange` is 0; `shortrange` was initialized to be 0 on line 14, but this value may have been changed on line 43), the last used value of `h` is loaded into the location `calling_h` (if the last step taken was too short, this location will contain the value of `h` before the short step was taken; this value was assigned on line 44).

If the loop ended with `state` equaling 2, assigned on line 39 (rather than 1, assigned on line 32), the variable `*success` is assigned 1 (true) on line 52, and the value `y` of the solution at `x1` is returned. Otherwise, `*success` is assigned 0, and the value 0 is returned on line 56.

The file `funct.c` contains the definition of the function $f(x, y)$.

```
 1 #include "rk45.h"
 2
 3 double funct(double x,double y)
 4 {
 5   const assumedzero=1e-20;
 6   double value;
 7   if ( absval(x+1.0)<=assumedzero ) {
 8     value=0.0;
 9   }
10   else {
11     value=-(x+1.0)*sin(x)+y/(x+1.0);
```

[92] One might think that instead of giving the coefficients as common fractions in these calculations, one should calculate these fractions as floating point numbers, and put these floating point numbers in the program. The reason for this would be that these fractions should be calculated only once, and not every time the loop is executed. However, any optimizing compiler would take care of this, and in fact these fractions would be evaluated only once, at compile time.

[93] This maybe trying to control the size of the error too tightly: if the error is only slightly smaller than $\epsilon/32$, doubling the step size will make the error only slightly smaller than ϵ; that is, in a few steps the error might become larger than ϵ, and the step size would then be halved. This might result in thrashing, i.e., frequent halving and doubling of the step size. Therefore it might be reasonable to give some additional leeway, and only consider the error too small if it is less than $\epsilon/64$. We will consider the effect of this on an example.

```
12   }
13   return(value);
14 }
```

This file defines

$$f(x,y) = -(x+1)\sin x + \frac{y}{x+1}.$$

The calling program is contained in the file `main.c`:

```
 1 #include "rk45.h"
 2 double sol(double x);
 3
 4 main()
 5 {
 6   double min_h=1e-5, epsilon=5e-10,
 7       x0=0.0,x1,xn=10.0,y0=1.0,y1,h,h0,yx;
 8   int i, n=10, success, countsucc=0,countfail=0, status;
 9   /* status==0: failed;
10      status==1: integrating;
11      status==2: reached the end of the interval; */
12   status=1;
13   printf("Solving the differential equation "
14       "y'=-(x+1)sin x+y/(x+1):\n");
15   printf("   x                y       "
16       "    exact solution          error\n\n");
17   yx=sol(x0);
18   printf("%6.3f %20.16f %20.16f %20.16f\n",
19          x0, y0, yx, y0-yx);
20   h0=(xn-x0)/((double) n); h=h0;
21   for (i=1; status==1; i++) {
22     x1=x0+h0;
23     y1=rk45(&funct, x0, y0, x1, epsilon, &h,
24           min_h, &countsucc, &countfail, &success);
25     if ( success ) {
26       x0=x1; y0=y1;
27       yx=sol(x0);
28       printf("%6.3f %20.16f %20.16f %20.16f\n",
29             x0, y0, yx, yx-y0);
30       if ( i>=n )  status=2;
31     }
32     else {
33       status=0;
34       printf("No more values could be calculated.\n");
35     }
36   }
37   printf("\n" "Parameters\n");
38   printf("epsilon:                  "
39          "                         "
40          "%.16f\n", epsilon);
41   printf("smallest step size allowed:"
42          "                        "
43          "%.16f\n", min_h);
44   printf("number of successful steps: %8d\n", countsucc);
```

```
45   printf("number of failed steps:     %8d\n", countfail);
46 }
47
48 double sol(double x) {
49   /* This function is the exact solution of the
50      differential equation being solved, to compare
51      with the numerical solution */
52   return (x+1.0)*cos(x);
53 }
```

This program will solve the differential equation $y' = f(x, y)$ with the above choice of $f(x, y)$ with the initial condition $y(0) = 1$. The exact solution of this equation with the given initial condition is

$$y(x) = (x + 1)\cos x. \tag{5}$$

Indeed, differentiating this, we obtain

$$y'(x) = \cos x - (x + 1)\sin x. \tag{6}$$

According to the above equation, we have

$$\cos x = \frac{y(x)}{x + 1}.$$

Replacing $\cos x$ in (6) with the right-hand side, we obtain the differential equation we are solving.

The exact solution using equation (5) is calculated in lines 48–53 as `sol(x)`, so that the calculated solution can be compared with the exact solution. The calculated solution and the exact solution is printed out at the points $x = 1, 2, 3, \ldots, 10$. This is accomplished by setting up a loop in lines 21–35. Inside the loop, the function `rk45`, discussed above, is called with initial values x_0 and ending values $x_1 = x + 1$ with $x_0 = 0, 1, \ldots, 10$. On line 38, the value of x, the calculated value $\bar{y}(x)$ of $y(x)$, the exact value of $y(x)$, and the error $y(x) - \bar{y}(x)$ is printed out. Finally, in lines 37–46, the value of ϵ, h, the smallest step size allowed, the number of successful steps, and the number of failed steps is allowed. On line 6, the smallest step size is specified as 10^{-10} and ϵ is given as 10^{-5} (the way the program was written, ϵ means the error allowed on an interval of length 1). The printout of the program is

```
 1 Solving the differential equation y'=-(x+1)sin x+y/(x+1):
 2    x               y           exact solution          error
 3
 4  0.000   1.0000000000000000   1.0000000000000000   0.0000000000000000
 5  1.000   1.0806046116088004   1.0806046117362795   0.0000000001274790
 6  2.000  -1.2484405098784015  -1.2484405096414271   0.0000000002369743
 7  3.000  -3.9599699865316182  -3.9599699864017817   0.0000000001298364
 8  4.000  -3.2682181044399328  -3.2682181043180596   0.0000000001218732
 9  5.000   1.7019731125069812   1.7019731127793576   0.0000000002723763
10  6.000   6.7211920060799493   6.7211920065525623   0.0000000004726129
11  7.000   6.0312180342683925   6.0312180347464368   0.0000000004780446
12  8.000  -1.3095003046372715  -1.3095003042775217   0.0000000003597498
13  9.000  -9.1113026190614956  -9.1113026188467696   0.0000000002147257
14 10.000  -9.2297868201511246  -9.2297868198409763   0.0000000003101476
15
16 Parameters
17 epsilon:                                          0.0000000005000000
```

```
18 smallest step size allowed:                          0.0000100000000000
19 number of successful steps:      353
20 number of failed steps:           10
```

If one makes a slight change in the file `rk45.c` by replacing the definition of the variable `epsilonover32=epsilon/32.` given in line 13 with `epsilonover32=epsilon/64.`, the printout changes only slightly:

```
 1 Solving the differential equation y'=-(x+1)sin x+y/(x+1):
 2    x               y           exact solution          error
 3
 4  0.000   1.0000000000000000   1.0000000000000000   0.0000000000000000
 5  1.000   1.0806046116088004   1.0806046117362795   0.0000000001274790
 6  2.000  -1.2484405098375091  -1.2484405096414271   0.0000000001960819
 7  3.000  -3.9599699864770947  -3.9599699864017817   0.0000000000753129
 8  4.000  -3.2682181043739900  -3.2682181043180596   0.0000000000559304
 9  5.000   1.7019731125861131   1.7019731127793576   0.0000000001932445
10  6.000   6.7211920061722683   6.7211920065525623   0.0000000003802938
11  7.000   6.0312180343918813   6.0312180347464368   0.0000000003545558
12  8.000  -1.3095003044983455  -1.3095003042775217   0.0000000002208238
13  9.000  -9.1113026189071356  -9.1113026188467696   0.0000000000603657
14 10.000  -9.2297868199575586  -9.2297868198409763   0.0000000001165816
15
16 Parameters
17 epsilon:                                             0.0000000005000000
18 smallest step size allowed:                          0.0000100000000000
19 number of successful steps:      361
20 number of failed steps:           10
```

Problem

1. Consider the differential equation $y' = f(x, y)$ with initial condition $y(x_0) = y_0$. Show that, with $x_1 = x_0 + h$, the solution at x_1 can be obtained with an error $O(h^3)$ by the formula

$$y_1 = y_0 + hf\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}f(x_0, y_0)\right).$$

In other words, this formula describes a Runge-Kutta method of order 2.[94]

Solution. Writing f, f_x, f_y for f and its derivatives at (x_0, y_0), we have We have

$$f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}f(x_0, y_0)\right) = f + \frac{h}{2}f_x + \frac{h}{2}f \cdot f_y + O(h^2).$$

according to Taylor's formula in two variables. Substituting this into the above formula for y_1, we obtain

$$y_1 = y_0 + hf + \frac{h^2}{2}(f_x + ff_y) + O(h^3).$$

This agrees with the Taylor expansion of y_1 (given in the preceding section) with error $O(h^3)$, showing that this is indeed a correct Runge-Kutta method of order 2.

[94]This method is called the *modified Euler method*. The *Euler method* simply takes $y_1 = y_0 + hf(x_0, y_0)$.

29. PREDICTOR-CORRECTOR METHODS

Consider the differential equation $y' = f(x, y)$, and assume that its solution $y = y(x)$ is known at the points $x_i = x_0 + ih$ for $h = 0, 1, 2, 3$. Write $y_i = y(x_i)$.[95] We would like to calculate y_4. Clearly,

$$y_4 = y_3 + \int_{x_3}^{x_4} f(x, y(x))\, dx. \tag{1}$$

We intend to evaluate the integral on the right-hand side by approximating $f(x, y(x))$ with an interpolation polynomial. We will consider two different interpolation polynomial for this: first, we will use the polynomial $P(x)$ interpolating at the points x_0, x_1, x_2, x_3, and, second we will use the polynomial $Q(x)$ interpolating at the points x_1, x_2, x_3, x_4. The interpolation by $Q(x)$ will give a better approximation, because interpolation inside an interval determined by the nodes is more accurate than interpolation outside this interval.[96] On the other hand, interpolation by $Q(x)$ will involve the unknown value y_4. This will not be a big problem, since y_4 will occur on both sides of the equation, and it can be determined by solving an equation.

Write $\bar{f}(x) = f(x, y(x))$, and write $f_i = f(x_i, y_i)$. To simplify the calculation assume that $x_2 = 0$; in this case $x_0 = -2h$, $x_1 = -h$, $x_3 = h$, $x_4 = 2h$. It will help us to make the calculations that follow more transparent to also use the notation $\bar{f}_i = \bar{f}(ih)$. Note, that, this will mean that

$$f_i = \bar{f}_{i-2} \tag{2}$$

Writing E_P for the error of the Newton interpolation polynomial P interpolating $\bar{f}$ at x_0, x_1, x_2, x_3 we have

$$\begin{aligned}\bar{f}(x) = P(x) + E_P(x) &= \bar{f}[0] + \bar{f}[0,-h]x + \bar{f}[0,-h,h]x(x+h) + \bar{f}[0,-h,h,-2h]x(x+h)(x-h)\\
&+ \bar{f}[0,-h,h,-2h,x]x(x+h)(x-h)(x+2h) = \bar{f}_0 + \frac{\bar{f}_0 - \bar{f}_{-1}}{h}x\\
&+ \frac{\bar{f}_1 - 2\bar{f}_0 + \bar{f}_{-1}}{2h^2}(x^2 + xh) + \frac{\bar{f}_1 - 3\bar{f}_0 + 3\bar{f}_{-1} - \bar{f}_{-2}}{6h^3}(x^3 - h^2x)\\
&+ \frac{\bar{f}^{(4)}(\xi_x)}{4!}(x^4 + 2hx^3 - h^2x^2 - 2h^3x)\end{aligned}$$

for some $\xi_x \in (x_0, x_4) = (-2h, 2h)$ (assuming that $x \in (x_3, x_4)$).[97] Integrating this, we obtain

$$\begin{aligned}\int_h^{2h} \bar{f} &= \bar{f}_0 h + \frac{\bar{f}_0 - \bar{f}_{-1}}{h} \cdot \frac{3h^2}{2} + \frac{\bar{f}_1 - 2\bar{f}_0 + \bar{f}_{-1}}{2h^2} \cdot \frac{23h^3}{12} + \frac{\bar{f}_1 - 3\bar{f}_0 + 3\bar{f}_{-1} - \bar{f}_{-2}}{6h^3} \cdot \frac{3h^4}{8}\\
&+ \frac{\bar{f}^{(4)}(\xi)}{4!} \cdot \frac{251}{30}h^5 = \frac{h}{24}(55\bar{f}_1 - 59\bar{f}_0 + 37\bar{f}_{-1} - 9\bar{f}_{-2}) + \bar{f}^{(4)}(\xi)\frac{251}{720}h^5\\
&= \frac{h}{24}(55f_3 - 59f_2 + 37f_1 - 9f_0) + \bar{f}^{(4)}(\xi)\frac{251}{720}h^5\end{aligned}$$

---

[95] More or fewer points could be considered, but we want to confine our attention to a case where we can derive formulas of considerable importance in practice.

[96] In fact, for $P(x)$, the interval determined by the interpolation points is (x_0, x_3), and so x in the interval (x_3, x_4) of integration is outside the interval of interpolation, while, for $Q(x)$, the interval determined by the interpolation points is (x_1, x_4), and so x is in the interval (x_3, x_4) of integration is inside the interval of interpolation.

[97] To ease the calculation of differences, one may note that

$$f[x_i, x_{i+1}, \dots, x_{i+k}] = \frac{1}{k!h^k}\Delta^k f_i.$$

according to (1) in Section 10 on Newton Interpolation With Equidistant Points, and

$$\Delta^n f(x) = \sum_{i=0}^{n} \binom{n}{i}(-1)^{n-i} f(x + ih).$$

according to a formula in Section 7 on Finite Differences.

for some $\xi \in (x_0, x) \subset (x_0, x_4)$; for the last equality we used (2). In calculating the remainder term, note that the polynomial

$$x^4 + 2hx^3 - h^2x^2 - 2h^3x = x(x+h)(x-h)(x+2h)$$

is positive on the interval $(h, 2h)$, and so we can use the Mean-Value Theorem for Integrals given in formula (1) in Section 20 on Simple Numerical Integration Formulas.[98] Recalling that $\bar{f}(x) = f(x, y(x))$ and substituting this into (1), we obtain

$$y_4 = y_3 + \frac{h}{24}(55f_3 - 59f_2 + 37f_1 - 9f_0) + \bar{f}^{(4)}(\xi)\frac{251}{720}h^5 \tag{3}$$

for some $\xi \in (x_0, x_4)$. This is the Adams-Bashforth predictor formula.

Similarly as above writing E_Q for the error of the Newton interpolation polynomial Q interpolating $\bar{f}$ at x_1, x_2, x_3, x_4, we have

$$\begin{aligned}\bar{f}(x) = Q(x) + E_Q(x) &= \bar{f}[0] + \bar{f}[0,-h]x + \bar{f}[0,-h,h]x(x+h) + \bar{f}[0,-h,h,2h]x(x+h)(x-h)\\ &+ \bar{f}[0,-h,h,2h,x]x(x+h)(x-h)(x-2h) = \bar{f}_0 + \frac{\bar{f}_0 - \bar{f}_{-1}}{h}x\\ &+ \frac{\bar{f}_1 - 2\bar{f}_0 + \bar{f}_{-1}}{2h^2}(x^2 + xh) + \frac{\bar{f}_2 - 3\bar{f}_1 + 3\bar{f}_0 - \bar{f}_{-1}}{6h^3}(x^3 - h^2x)\\ &+ \frac{\bar{f}^{(4)}(\eta_x)}{4!}(x^4 - 2hx^3 - h^2x^2 + 2h^3x)\end{aligned}$$

for some $\eta_x \in (x_1, x_4) = (-2h, 2h)$ (assuming that $x \in (x_3, x_4)$). Integrating this, we obtain

$$\begin{aligned}\int_h^{2h} \bar{f} &= \bar{f}_0 h + \frac{\bar{f}_0 - \bar{f}_{-1}}{h}\cdot\frac{3h^2}{2} + \frac{\bar{f}_1 - 2\bar{f}_0 + \bar{f}_{-1}}{2h^2}\cdot\frac{23h^3}{12} + \frac{\bar{f}_2 - 3\bar{f}_1 + 3\bar{f}_0 - \bar{f}_{-1}}{6h^3}\cdot\frac{3h^4}{8}\\ &+ \frac{\bar{f}^{(4)}(\eta)}{4!}\cdot\frac{19}{30}h^5 = \frac{h}{24}(9\bar{f}_2 + 19\bar{f}_1 - 5\bar{f}_0 + 9\bar{f}_{-1}) - \bar{f}^{(4)}(\eta)\frac{19}{720}h^5\\ &= \frac{h}{24}(9f_4 + 19f_3 - 5f_2 + 9f_1) - \bar{f}^{(4)}(\eta)\frac{19}{720}h^5\end{aligned}$$

for some $\eta \in (x_1, x_4)$, where (2) was used to obtain the last equality. In calculating the remainder term, note that the polynomial

$$x^4 - 2hx^3 - h^2x^2 + 2h^3x = x(x+h)(x-h)(x-2h)$$

is negative on the interval $(h, 2h)$, and so we can again use the Mean-Value Theorem for Integrals given in formula (1) in the Section 20 on Simple Numerical Integration Formulas. Recalling that $\bar{f}(x) = f(x, y(x))$ and substituting this into (1), we obtain

$$y_4 = y_3 + \frac{h}{24}(9f_4 + 19f_3 - 5f_2 + 9f_1) - \bar{f}^{(4)}(\eta)\frac{19}{720}h^5 \tag{4}$$

for some $\eta \in (x_1, x_4)$. This is the Adams-Moulton corrector formula.

[98] What we need here is a slight extension of that result. In discussing that formula, we assumed that ξ_x belongs to the interval of integration, which is not the case at present. The same argument can be extended to the situation when ξ_x belongs to some other, arbitrary interval, since the only thing that was used in the argument was that the derivative of a function satisfies the Intermediate-Value Theorem. Thus, the Mean-Value Theorem in question can be extended to cover the present situation.

In a practical method for solving differential equations, one uses the corrector formula (4) as an equation for y_4 (since y_4 occurs on both sides). One solves this equation by fixed-point iteration, but one only does one step of fixed point iteration, with a starting value obtained by the predictor formula (3). We will explain below how exactly this is done, but first we need to discuss the error estimates for these formulas. Write y_{pred} for the value calculated for y_4 by the predictor formula (i.e., y_{pred} equals the right-hand side of (3) except for the error term), and y_{corr} of the value calculated for y_4 by the corrector formula (using y_{pred} to calculated f_4. That is, write

$$\begin{aligned} y_{\text{pred}} &= y_3 + \frac{h}{24}(55f_3 - 59f_2 + 37f_1 - 9f_0), \\ f_{\text{pred}} &= f(x_4, y_{\text{pred}}), \\ y_{\text{corr}} &= y_3 + \frac{h}{24}(9f_{\text{pred}} + 19f_3 - 5f_2 + 9f_1). \end{aligned} \tag{5}$$

In order to estimate the error, we make the assumption that

$$\bar{f}^{(4)}(\eta) \approx \bar{f}^{(4)}(\xi) \qquad \text{and} \qquad y_4 \approx y_{\text{corr}} - \bar{f}^{(4)}(\eta)\frac{19}{720}h^5 \tag{6}$$

holds.

The reason the latter equation is not exact is that we used f_{pred} to calculate y_{corr}, rather than f_4, as on the right-hand side of (4). For this reason,

$$\begin{aligned} y_4 - y_{\text{corr}} &= \frac{9h}{24}(f_4 - f_{\text{pred}}) - \bar{f}^{(4)}(\eta)\frac{19}{720}h^5 = \frac{9h}{24}f_y(x_4, \lambda)(y_4 - y_{\text{pred}}) - \bar{f}^{(4)}(\eta)\frac{19}{720}h^5 \\ &= \frac{9h}{24}f_y(x_4, \lambda)\bar{f}^{(4)}(\xi)\frac{251}{720}h^5 - \bar{f}^{(4)}(\eta)\frac{19}{720}h^5 = -\bar{f}^{(4)}(\eta)\frac{19}{720}h^5 + f_y(x_4, \lambda)\bar{f}^{(4)}(\xi)\frac{251}{1920}h^6 \end{aligned}$$

where the second equation follows with λ between y_4 and y_{pred} from the Mean-Value Theorem for the function $f(x_4, y)$ considered as a function of the single variable y only, and the third equation uses the error term given in (3). For small h, the second term on the right-hand side is much smaller than the first term, so we obtain the second formula in (6).

Using these estimates, we obtain

$$y_{\text{corr}} - y_{\text{pred}} = (y_4 - y_{\text{pred}}) - (y_4 - y_{\text{corr}}) \approx \bar{f}^{(4)}(\eta)\left(\frac{251}{720} + \frac{19}{720}\right)h^5 = \bar{f}^{(4)}(\eta)\frac{270}{720}h^5,$$

where (3) and (5) were used to express $y_4 - y_{\text{pred}}$, and the second formula in (6) was used to estimate $y_4 - y_{\text{corr}}$. Hence

$$y_4 - y_{\text{corr}} \approx -\bar{f}^{(4)}(\eta)\frac{19}{720}h^5 = -\frac{19}{270} \cdot \bar{f}^{(4)}(\eta)\frac{270}{720}h^5 \approx -\frac{19}{270}(y_{\text{corr}} - y_{\text{pred}})$$

The left-hand side is the error E_{corr} of the corrector formula. That is,

$$E_{\text{corr}} \approx -\frac{19}{270}(y_{\text{corr}} - y_{\text{pred}}) \approx -\frac{1}{14}(y_{\text{corr}} - y_{\text{pred}}). \tag{7}$$

For the last approximate equation, note that $19/270 \approx 0.070,370,4$ and $1/14 \approx 0.071,428,6$.[99]

[99]There seems to be hardly any practical difference whether one uses the value 19/270 or 1/14. The latter value is easier for humans to remember; for computers, this is not an issue. The computer evaluates these fractions only once, at compile time. Nevertheless, we will use the latter value in the computer program below.

The method combining formulas (5) and (7) above for y_{pred}, y_{corr}, and E_{corr} is called the Adams-Bashforth-Moulton predictor-corrector method. Since the formulas rely on four previous values values for y_i, the method is used in combination with another method, such as the Runge-Kutta-Fehlberg method, that can supply these values. The corrector formula (using fixed point iteration to solve the equation for y_4) is used only once; if the error is not small enough after one iteration, it is better to decrease the step size than to do more iteration. After the Runge-Kutta-Fehlberg method supplies the initial values, the Adams-Bashforth-Moulton method can work on its own until the step size needs to be changed (decreased because the error is too large or increased because the error is too small – to speed up the calculation). When the step size is changed, the Runge-Kutta-Fehlberg method can again be invoked to supply new starting-valued with the changed step size.

Next we will discuss a computer implementation of this method. We will use the method solve the same differential equation

$$f(x, y) = -(x + 1) \sin x + \frac{y}{x + 1};$$

the file `funct.c` defining this function is identical to the file with the same name considered for the Runge-Kutta-Fehlberg method. The header file `abm.h` contains the header file for a number of files used in the implementation of this method:

```
#include <stdio.h>
#include <math.h>

#define absval(x) ((x) >= 0.0 ? (x) : (-(x)))

double funct(double x, double y);
double abmrkf(double (*fnct)(double,double), double x0,
    double y0, double x1, double epsilon, double *h,
    double min_h, int *countabsucc, int *countabfail,
    int *countrksucc, int *countrkfail, int *success);
double abmstep(double (*fnct)(double,double), double x4,
    double y3, double f0, double f1, double f2, double f3,
    double h, double *error);
double rkfstep(double (*fnct)(double,double), double x0,
    double y0, double *x1, double h, double *k1addr,
    double *error);
```

This file contains the declarations of the functions needed for the program; these declarations will be discussed in detail below. One step of the Runge-Kutta-Fehlberg method is implemented in the file

```
#include "abm.h"
#define F(x,y) ((*fnct)(x,y))
#define K1 (*k1addr)

double rkfstep(double (*fnct)(double,double), double x0,
    double y0, double *x1, double h, double *k1addr,
    double *error)
{
  /* This program implements the Runge-Kutta_Fehlberg
     method to solve an ordinary differential equation */
  double k2, k3, k4, k5, k6, y1;
  K1=F(x0,y0);
  k2=F(x0+h/4.,y0+h*K1/4.);
```

```
14   k3=F(x0+3.*h/8.,y0+h*(3.*K1+9.*k2)/32.);
15   k4=F(x0+12.*h/13.,y0+h*(1932.*K1-7200.*k2+7296.*k3)/2197.);
16   k5=F(x0+h,y0+h*(439.*K1/216.-8.*k2+3680.*k3/513.
17                                  -845.*k4/4104.));
18   k6=F(x0+h/2.,y0+h*(-8.*K1/27.+2.*k2-3544.*k3/2565.
19                             +1859.*k4/4104.-11.*k5/40.));
20   *error=absval(h*(K1/360.-128.*k3/4275.-2197.*k4/75240.
21                                +k5/50.+2*k6/55.));
22   *x1=x0+h;
23   y1=y0+h*(25.*K1/216.+1408.*k3/2565.+2197.*k4/4104.-k5/5.);
24   return y1;
25 }
```

The implementation of the Runge-Kutta-Fehlberg method has been discussed earlier. This file only implements a single step of the method, and the step size control will be performed in the calling program. The parameters of this function are the pointer `funct` describing the function $f(x, y)$ in the differential equation to be solved, `x0` for the starting point, `y0` for the initial value of the solution at that point, `*x1` is end point of the step where the solution is calculated (this value is calculated in here, and the calling program can read it from the location `x1`, the step size `h` (so `*x1=x0+h`), `k1addr` for the address of the quantity $K_1(x_0, y_0)$ (this is calculated here, and the calling program can get it from this location; K_0 can be reused even if the step size is changed), and the `*error` of the step (readable by the calling program). To simplify the notation, in lines 2 and 3 the symbols `F(x,y)` and `K1` are defined. Lines 11–23 perform the calculation the same way as was already discussed above on account of the Runge-Kutta-Fehlberg method. On line 24, the value `y1` of the solution is returned to the calling program.

One step of the Adam-Bashforth-Moulton method is implemented in the file `abmstep.c`:

```
 1 #include "abm.h"
 2 #define F(x,y) ((*fnct)(x,y))
 3
 4 double abmstep(double (*fnct)(double,double), double x4,
 5     double y3, double f0, double f1, double f2, double f3,
 6     double h, double *error)
 7 {
 8   double predy4, predf4, y4;
 9   predy4=y3+h*(55.*f3-59.*f2+37.*f1-9.*f0)/24.;
10   predf4=F(x4,predy4);
11   y4=y3+h*(9.*predf4+19.*f3-5.*f2+f1)/24.;
12   *error=absval(y4-predy4)*1./14.;
13   return y4;
14 }
```

The function `abmstep` is quite simple. Its parameters are a pointer `fnct` representing the function $f(x, y)$, `x4`, `f1`, `f2`, `f3`, `h`, corresponding to x_4, f_1, f_2, f_3, and h, respectively, and the pointer `error` to the location containing the error as determined by formula (7) on line 12. The file `abm.c` implements step size control for the method:

```
 1 #include "abm.h"
 2 #define F(x,y) ((*fnct)(x,y))
 3 #define H (*h)
 4
 5 double abmrkf(double (*fnct)(double,double), double x0,
 6     double y0, double x1, double epsilon, double *h,
 7     double min_h, int *countabsucc, int *countabfail,
```

```
 8     int *countrksucc, int *countrkfail, int *success)
 9 {
10   /* This solves an ordinary differential equation
11      using the Adams-Bashforth-Moulton predictor corrector
12      method. Automatic step size control is incorporated.
13      The starting values or the initial values when
14      the step size is changed are supplied by the
15      Runge-Kutta-Fehlberg method. The main problem
16      in coupling these two methods is to ensure that
17      most of the steps are performed by the predictor
18      corrector method and not by the Runge-Kutta method.
19      This is a difficult problem; the main steps to
20      ensure this were: 1) the Runge-Kutta method is
21      used only to decrease the step size, and never to
22      increase it; 2) the predictor corrector method is
23      allowed to increase the step size only if it was
24      found on 8 consecutive occasions that the step size
25      is too small. */
26   const double assumedzero=1e-20;
27   double  error,
28           x[5], y[5], f[4], working_h,
29           epsilonover64=epsilon/64.;
30   int i=0, upscalecount=0, iserror_ok, state, shortrange=0, j;
31   /* state==0: stepping
32      state==1: h is too small
33      state==2: x1 is reached */
34   /* iserror_ok==0: error is too small
35      iserror_ok==1: error is OK
36      iserror_ok==2: error is too large */
37   /* using Runge-Kutta-Fehlberg to generate starting values: */
38   x[0]=x0; y[0]=y0;
39   for ( state=0; !state ;) {
40     if ( i==3 ) {
41       f[3]=F(x[3],y[3]);
42       x[4]=x[3]+H;
43       y[4]=abmstep(fnct,x[4],y[3],f[0],f[1],f[2],f[3],
44                    H,&error);
45       iserror_ok=1;
46       if ( error <= H*epsilonover64 ) iserror_ok=0;
47       if ( error >  H*epsilon )       iserror_ok=2;
48       if ( iserror_ok < 2 ) {
49         (*countabsucc)++;
50         for (j=0; j<4; j++) {
51           x[j]=x[j+1];
52           y[j]=y[j+1];
53         }
54         for (j=0; j<3; j++) {
55           f[j]=f[j+1];
56         }
57       }
58       else (*countabfail)++;
```

```
    }
    else {
      y[i+1]=rkfstep(fnct,x[i],y[i],&x[i+1],
               H,&f[i],&error);
      if ( error > H*epsilon ) {
        iserror_ok=2;
        (*countrkfail)++;
      }
      else {
        iserror_ok=1;
        i++;
        (*countrksucc)++;
      }
    }
    if ( iserror_ok==2 ) {
      H /= 2.;
      if ( H < min_h ) state=1;
      x[0]=x[i]; y[0]=y[i]; i=0;
    }
    else {
      if ( absval(x1-x[i]) <= assumedzero ) state=2;
      else {
        if ( iserror_ok==0 ) {
          upscalecount++;
          if ( upscalecount==8 ) {
            H *= 2.;
            x[0]=x[i]; y[0]=y[i]; i=0;
          }
        }
        else upscalecount=0;
        if ( H > x1-x[i] ) {
          shortrange=1;
          working_h=H; H=x1-x[i];
          x[0]=x[i]; y[0]=y[i]; i=0;
        }
        else shortrange=0;
      }
    }
  }
  if ( shortrange ) H=working_h;
  if ( state==2 ) {
     *success=1;
     return y[i];
  }
  else {
    *success=0;
    return 0.0;
  }
}
```

The function `abmrkf` has its parameters arranged in a similar way to the parameters of the function `rk45` in the file `rk45.c` discussed above, in the section on Runge-Kutta methods. In fact, the

function `abmrkf` here and the earlier function `rk45` has many features in common. The parameters of `abmrkf` are a pointer `fnct` to the function representing $f(x, y)$, the starting point `x0`, the initial value `y0` at this point, the point `x1` where the solution is desired (this point is reached as a result of many steps, so its meaning is very different from the meaning of `x1`, reached in a single step, in the functions `rkfstep` and `abmstep`), the global error `epsilon` allowed on an interval of length 1, a pointer `h` to the value of h, the smallest step size `min_h` allowed, a pointer `countabsucc` to the number of successful Adams-Bashforth-Moulton steps, a pointer `countabfail` to the number of failed Adams-Bashforth-Moulton steps, a pointer `countrksucc` to the number of successful Runge-Kutta-Fehlberg steps, a pointer `countrkfail` to the number of failed Runge-Kutta-Fehlberg steps, and a pointer to an integer `*success`, which is 1 is the determination of the value of the solution at `x1` is successful, and 0 is it is unsuccessful. The function returns the value of the solution at `y1` if successful (returns `0.0` otherwise).

Meaning of the integer `state` is explained in the comments in lines 31–33 (0 if the method is stepping, 1 is the value of h is smaller than the allowed step size, and 2 if the point x_1 is reached. The meaning of the integer `iserror_ok` is explained in the comments in lines 34–36 (0 if the error is too small, 1 is the error is OK, and 2 if the error is too large. The loop in lines 39–97 is performed as long as useful calculations can be done, i.e. until the calculation is finished (the point x_1 is reached) or when the calculation cannot be continued (since h would have to be smaller than allowed). The test on line 40 `i==0` is evaluated true when enough starting values have been obtained to do an Adams-Bashforth-Moulton step, and in line 40 the program doing this is called. To read these lines, note the definitions in lines 2–3 defining `F(x,y)` and `H`. If the error is this step is smaller than or equal to $h\epsilon/64$ (the parameter `epsilonover64` has value $\epsilon/64$) the error is too small (`iserror_ok` is set to 0), and if the error is larger than $h\epsilon$, the error is too large (`iserror_ok` is set to 2); if the error is not too large, the variable the variable `*countabsucc` incremented on line 49, and the arrays `x[]`, `y[]`, and `f[]` are updated in lines 50–56. On line 58, `*countabfail` is implemented because the error was found too large (through the failure of the test on line 48).

On line 61, a Runge-Kutta-Felberg step is taken (because the test on line 40 failed, i.e., showed that there are not enough values available to take an Adams-Bashforth-Moulton step), and in lines 64–66 the integer `iserror_ok` is set to 2, and `*countrkfail` is incremented in case the error is found to be too large on line 63, otherwise `iserror_ok` is set to 1, and `*countrksucc` is incremented in lines 68–70. The variable `i` is incremented on line 69 to indicate that the successful Runge-Kutta-Fehlberg step made an additional starting value available. In line 73 it is tested if the error is too large; this may have happened either in a Runge-Kutta-Fehlberg step or an Adams-Bashforth-Moulton step. The step size must be decreased if this is the case, and this is done in line 74. In line 76, `i` is set to 0 to indicate that only 1 (i.e., $i + 1$) starting value is available (if the step size changes, earlier starting values become useless.[100])

If the step size does not need to be changed in lines 73–77, on line 79 it is tested if we are close enough to target point `x1` (we should of course not test exact equality with floating point numbers), or whether more steps need to be taken. If the latter is the case, on line 81 it is tested if the error is too small. If so, the step size may need to be increased. Only an Adams-Bashforth-Moulton step can increase the step size (in lines 64 and 68 we saw that the Runge-Kutta-Fehlberg method cannot set `iserror_ok` to 0). The step size will be increased only if it is found on eight occasions in a row that the error is too small; the number of successive occurrences of a too small error is counted by the variable `upscalecount`. On the eighth occurrence, the step size is doubled on line 84, and `i` is set to 0 to indicate that there is only one (i.e., $i + 1$) available starting value.

In line 89, it is tested if the current step size to larger than the step needs to be taken to reach x_1;

[100] With more expensive book keeping, some of the old starting values could still be used even if the step size is halved. If the step size is doubled, then it is possible that there are still four starting values available among the already calculated values, but to keep track of them is more expensive also in this case. Such a more expensive book keeping hardly seems justified unless a function evaluation is enormously costly. As will appear from the printout of the present program, it seems that the method is tuned in a way that a change in step size is needed only rarely.

if so, the variable `shortrange` is set to 1, and the step size is changed on line 91 to the size of the actual step that needs to be taken. The current value of h is preserved in the variable `working_h` so that the program can communicate this value of h to the calling program. On line 92, `i` is set to be zero to ensure that the next (and last) step is a Runge-Kutta-Fehlberg step. In line 98, the working value of h is restored in case the last step was a short step, so that the calling program can read this value in the location `h` (recall that `H` stands for `*h`, as indicated on line 3). In line 99 the variable `state` is tested to see whether the determination of `y1` was successful; if so, the calculated value is returned to the calling program on line 101.

The main program contained in the file `main.c` is very similar to the file by the same name discussed on account of the Runge-Kutta-Fehlberg method in the preceding section:

```
 1 #include "abm.h"
 2 double sol(double x);
 3
 4 main()
 5 {
 6   double min_h=1e-5, epsilon=5e-10,
 7       x0=0.0,x1,xn=10.0,y0=1.0,y1,h,h0,yx;
 8   int i, n=10, success, countabsucc=0,countabfail=0,
 9       countrksucc=0,countrkfail=0, status;
10   /* status==0: failed;
11      status==1: integrating;
12      status==2: reached the end of the interval; */
13   status=1;
14   printf("Solving the differential equation "
15       "y'=-(x+1)sin x+y/(x+1):\n");
16   printf("   x                y       "
17       "    exact solution          error\n\n");
18   yx=sol(x0);
19   printf("%6.3f %20.16f %20.16f %20.16f\n",
20          x0, y0, yx, y0-yx);
21   h0=(xn-x0)/((double) n); h=h0;
22   for (i=1; i<=n ; i++) {
23     x1=x0+h0;
24     y1=abmrkf(&funct, x0, y0, x1, epsilon, &h,
25     min_h, &countabsucc, &countabfail,
26     &countrksucc, &countrkfail, &success);
27     if ( success ) {
28       x0=x1; y0=y1;
29       yx=sol(x0);
30       printf("%6.3f %20.16f %20.16f %20.16f\n",
31              x0, y0, yx, y0-yx);
32     }
33     else {
34       printf("No more values could be calculated.\n");
35       break;
36     }
37   }
38   printf("\n" "Parameters\n");
39   printf("epsilon:                  "
40          "                         "
41          "%.16f\n", epsilon);
```

```
42   printf("smallest step size allowed:"
43          "                        "
44          "%.16f\n", min_h);
45   printf("number of ABM successful steps: %8d\n", countabsucc);
46   printf("number of ABM failed steps:     %8d\n", countabfail);
47   printf("number of RK successful steps:  %8d\n", countrksucc);
48   printf("number of RK failed steps:      %8d\n", countrkfail);
49 }
50
51 double sol(double x) {
52   /* This function is the exact solution of the
53      differential equation being solved, to compare
54      with the numerical solution */
55   return (x+1.0)*cos(x);
56 }
```

The only difference between this file and the file `main.c` discussed on account of the Runge-Kutta-Felberg method, that here the separate step counts are printed out for Runge-Kutta-Fehlberg steps and Adams-Bashforth-Moulton steps in lines 45–48. The printout of this program is as follows:

```
 1 Solving the differential equation y'=-(x+1)sin x+y/(x+1):
 2    x               y           exact solution          error
 3
 4  0.000   1.0000000000000000   1.0000000000000000   0.0000000000000000
 5  1.000   1.0806046121476161   1.0806046117362795   0.0000000004113366
 6  2.000  -1.2484405092421744  -1.2484405096414271   0.0000000003992527
 7  3.000  -3.9599699859316626  -3.9599699864017817   0.0000000004701192
 8  4.000  -3.2682181037536910  -3.2682181043180596   0.0000000005643685
 9  5.000   1.7019731136560623   1.7019731127793576   0.0000000008767047
10  6.000   6.7211920076259482   6.7211920065525623   0.0000000010733860
11  7.000   6.0312180359864973   6.0312180347464368   0.0000000012400602
12  8.000  -1.3095003030395211  -1.3095003042775217   0.0000000012380006
13  9.000  -9.1113026175358165  -9.1113026188467696   0.0000000013109534
14 10.000  -9.2297868184153185  -9.2297868198409763   0.0000000014256585
15
16 Parameters
17 epsilon:                                          0.0000000005000000
18 smallest step size allowed:                       0.0000100000000000
19 number of ABM successful steps:     2033
20 number of ABM failed steps:            4
21 number of RK successful steps:        52
22 number of RK failed steps:             6
```

Problem

1. For a certain predictor-corrector method, the error of the predicted value y_1 is $\frac{1}{3}h^3Y'''(\xi_1)$, where $Y(x)$ is the true solution of the equation, and the error of the corrected value $\bar{y}_1$ is $-\frac{1}{12}h^3Y'''(\xi_2)$, where ξ_1 and ξ_2 are some unknown numbers near where the solution is being calculated. Estimate the error in terms of the difference $\bar{y}_1 - y_1$.

Solution. We have

$$Y(x_1) - y_1 = \frac{1}{3}h^3Y'''(\xi_1),$$

and

$$Y(x_1) - \bar{y}_1 = -\frac{1}{12}h^3 Y'''(\xi_2).$$

Hence, assuming $Y'''(\xi_1) \approx Y'''(\xi_2)$,

$$\bar{y}_1 - y_1 = (Y(x_1) - y_1) - (Y(x_1) - \bar{y}_1) \approx \frac{1}{3}h^3 Y'''(\xi_2) - \left(-\frac{1}{12}\right) h^3 Y'''(\xi_2) = \frac{5}{12}h^3 Y'''(\xi_2).$$

Therefore

$$Y(x_1) - \bar{y}_1 = -\frac{1}{12}h^3 Y'''(\xi_2) = -\frac{1}{5} \cdot \frac{5}{12}h^3 Y'''(\xi_2) \approx -\frac{1}{5}(\bar{y}_1 - y_1).$$

2. For a certain predictor-corrector method, the error of the predicted value y_1 is $-\frac{14}{45}h^5 Y^{(5)}(\xi_1)$, where $Y(x)$ is the true solution of the equation, and the error of the corrected value $\bar{y}_1$ is $\frac{1}{90}h^5 Y^{(5)}(\xi_2)$, where ξ_1 and ξ_2 are some unknown number near where the solution is being calculated. Estimate the error in terms of the difference $\bar{y}_1 - y_1$.

Solution. Write $y(x_1)$ for the correct value of $y(x)$ at x_1. Assume that $Y^{(5)}(\xi_1) \approx Y^{(5)}(\xi_2)$. We then have

$$y(x_1) - y_1 = -\frac{14}{45}h^5 Y^{(5)}(\xi_1),$$

and

$$y(x_1) - \bar{y}_1 = \frac{1}{90}h^5 Y^{(5)}(\xi_2).$$

Hence

$$\bar{y}_1 - y_1 = (y(x_1) - y_1) - (y(x_1) - \bar{y}_1) \approx -\frac{14}{45}h^5 Y^{(5)}(\xi_2) - \frac{1}{90}h^5 Y^{(5)}(\xi_2) = -\frac{29}{90}h^5 Y^{(5)}(\xi_2).$$

Therefore

$$y(x_1) - \bar{y}_1 = \frac{1}{90}h^5 Y^{(5)}(\xi_2) = -\frac{1}{29} \cdot \left(-\frac{29}{90}\right) h^5 Y^{(5)}(\xi_2) \approx -\frac{1}{29}(\bar{y}_1 - y_1).$$

30. RECURRENCE EQUATIONS

An equation of form

$$\sum_{k=0}^{m} a_k y_{n+k} = 0 \qquad (a_0 \neq 0,\ a_m \neq 0,\ m > 0) \tag{1}$$

is called a *recurrence equation*, more precisely, a *homogeneous* recurrence equation. (If the right-hand side is replaced with some function of n that is not identically zero, then what he get is called an *inhomogeneous* recurrence equation.[101] In this section, we will only discuss homogeneous recurrence equations.) Here a_k for integers k with $0 \leq k \leq m$ are given numbers, and we seek solutions y_n such that these equations are satisfied for all nonnegative integers n. m is called the order of this

[101] A recurrence equation can often be written in terms of the difference operators introduced in Section 9 on Finite Differences. In this case, a recurrence equation is also called a *difference equation*.

equation. The assumptions $a_0 \neq 0$ and $a_m \neq 0$ are reasonable in the sense that if either of these assumptions fail, the equation can be replaced with a lower order equation. It will be advantageous to work with complex numbers; i.e., the numbers a_k and y_n will be allowed to be complex. It is convenient to consider a solution of this equation as a vector

$$\mathbf{y} = \langle y_0, y_1, y_2, \dots \rangle$$

with infinitely many components. These vectors can be added componentwise, that is

$$\langle y_0, y_1, y_2, \dots \rangle + \langle z_0, z_1, z_2, \dots \rangle = \langle y_0 + z_0, y_1 + z_1, y_2 + z_2, \dots \rangle,$$

and can be multiplied by scalars, that is

$$\alpha\langle y_0, y_1, y_2, \dots \rangle = \langle \alpha y_0, \alpha y_1, \alpha y_2, \dots \rangle.$$

The solution vectors form an n-dimensional vector space. First, they form a vector space, since if $\mathbf{y}$ and $\mathbf{z}$ are solutions then $\alpha\mathbf{y} + \beta\mathbf{z}$ is also a solution. It is also clear that the dimension of this vector space is m since each solution is determined if we specify the number y_i for each integer i with $0 \leq i \leq m-1$ (indeed, y_n for $n \geq m$ is then determined by the recurrence equation, as $a_m \neq 0$), and these numbers y_i can be specified arbitrarily.

Write

$$P(\zeta) = \sum_{k=0}^{m} a_k \zeta^k. \tag{2}$$

The polynomial $P(\zeta)$ is called the characteristic polynomial of the recurrence equation (1), and the polynomial equation $P(\zeta) = 0$ is called its characteristic equation. Here ζ is a complex variable.[102]

The forward shift operator E on functions of n is defined by writing $Ef(n) = f(n+1)$; the forward shift operator was introduced in Section 9 on page 28. The difference there was that the variable used was x, and x was a real variable, whereas here n is a nonnegative integer variable.[103] The powers of the operator E can be defined as they were in Section 9; in addition, we can also use the identity operator I. Polynomials of the operator E will be called difference operators.[104] y_n will be considered as a function of n, and the operator E on y_n will act according to the equation $Ey_n = y_{n+1}$.[105] The recurrence equation (1) can be written in terms of the operator E as

$$\Big(\sum_{k=0}^{m} a_k E^k\Big) y_n = 0. \tag{3}$$

By solving the characteristic equation, the characteristic polynomial can be factored as the product of m linear factors; assuming that λ_j is a zero[106] of multiplicity m_j of the characteristic polynomial for j with $1 \leq j \leq N$ (the λ_j's are assumed to be pairwise distinct), we have

$$\sum_{k=0}^{m} a_k \zeta^k = a_m \prod_{j=1}^{N} (\zeta - \lambda_j)^{m_j}, \qquad \text{where} \quad \sum_{j=1}^{N} m_j = m;$$

[102] Or an indeterminate, from an alternative viewpoint. An indeterminate is a symbolic variable used in defining a polynomial ring, and is not tu be interpreted as representing a number.

[103] The forward shift operator is always associated with a variable; if more than one variable were associated with forward shift operators, the notation should indicate the variable in question as well, for example E_x would shift the variable x forward, while E_y would shift the variable y, etc.

[104] This is because these operators can also be expressed in terms of of the forward difference operator Δ described in Section 9.

[105] It would be formally more correct, but less convenient, to say that E acts on vectors $\langle y_0, y_1, y_2, \dots \rangle$, and

$$E\langle y_0, y_1, y_2, \dots \rangle = \langle y_1, y_2, y_3, \dots \rangle.$$

[106] A *zero* of a polynomial is a *root* of the equation obtained by equating the polynomial to zero.

the second equation here just says that the above polynomial equation (of degree m) has m roots, counting multiplicities. The difference operator in recurrence equation (3) has a corresponding factorization:

$$\sum_{k=0}^{m} a_k E^k = a_m \prod_{j=1}^{N} (E - \lambda_j)^{m_j};$$

here $E - \lambda_j$ could also have been written as $E - \lambda_j I$, but the identity operator is often omitted when is has a number coefficient. This is because the rules of algebra involving polynomials of the variable ζ and polynomials of the forward shift operator E are the same.[107]

The degree of a polynomial $P(n)$ of n will be denoted by $\deg P(n)$; the constant polynomial that is not identically zero will have degree zero, and the identically zero polynomial will have degree -1. Then we have

LEMMA. *Let λ and η be nonzero complex numbers, and let $P(n)$ be a polynomial of n that is not identically zero. Then*

$$(E - \lambda)P(n)\eta^n = Q(n)\eta^n,$$

where $Q(n)$ is another polynomial of n such that $\deg Q(n) = \deg P(n)$ if $\lambda \neq \eta$ and $\deg Q(n) = \deg P(n) - 1$ if $\lambda = \eta$.

PROOF. Given an integer $k \geq 0$, we have

$$(E - \lambda)n^k\eta^n = (n+1)^k\eta^{n+1} - \lambda n^k\eta^n = \sum_{j=0}^{k} \binom{k}{j} n^j\eta^{n+1} - \lambda n^k\eta^n$$

$$= \left((\eta - \lambda)n^k + \eta \sum_{j=0}^{k-1} \binom{k}{j} n^j \right) \eta^n;$$

the second equality was obtained by using the Binomial Theorem. This equation says it all; if $\lambda = \eta$ then the term involving n^k *will cancel*, and if $\lambda \neq \eta$ then this term *will not cancel*. In the former case, the operator lowers the degree of n^k in the term $n^k\eta^n$ by one. (In this case, if n^k is the term of the highest degree of the polynomial $P(n)$, then the resulting term $\eta\binom{k}{1}n^{k-1}\eta^n$ will not cancel against the terms resulting from lower degree terms of $P(n)$, since the degrees of those terms will also be lowered.) The proof is complete. □

Linear independence of certain functions. Functions here mean functions on nonnegative integers; instead of the word "function" we could have used the word "sequence." The lemma just established has several important corollaries.

COROLLARY (LINEAR INDEPENDENCE). *Let $r \geq 1$ be an integer. Let $f_k(n) = P_k(n)\lambda_k^n$ be functions of n for k with $1 \leq k \leq r$, where $P_k(n)$ is a polynomial of n that is not identically zero, and λ_k is a nonzero complex number, such that if $1 \leq k < l \leq r$ then either $\lambda_k \neq \lambda_l$, or if $\lambda_k = \lambda_l$ then $\deg P_k(n) \neq \deg P_l(n)$. Then the functions f_k are linearly independent.*

PROOF. Assume, on the contrary, that we have

$$\sum_{k=1}^{r} c_k P_k(n)\lambda_k^n \equiv 0,$$

[107] In particular, given complex numbers λ and η the operators $E - \lambda$ and $E - \eta$ commute; that is

$$(E - \lambda)(E - \eta) = (E - \eta)(E - \lambda).$$

Note that E does not commute with expressions involving n. For example, $nEn^2 = n(n+1)^2$, and $n^2En = n^2(n+1)$.

where not all the complex coefficients c_k are zero ($\equiv$ here means that equality holds identically; in the present case this means that equality holds for every nonnegative integer n). We will show that this equation cannot hold. To this end, without loss of generality, we may assume that none of the coefficients are zero, since the terms with zero coefficients can simply be discarded. Further, we may assume that among the terms $P_k(n)\lambda_k^n$ the polynomial $P_1(n)$ is the one that has the highest degree (other polynomials $P_k(n)$ with nonzero c_k for $\lambda_k \neq \lambda_1$ may have the same degree, but not higher). Let d be the degree of $P_1(x)$. Then

$$(E-\lambda_1)^d\Big(\prod_{\substack{k:2\le k\le r,\\ \lambda_k\neq\lambda_1}}(E-\lambda_k)^{d+1}\Big)\sum_{k=1}^{r}c_kP_k(n)\lambda_k^n = c\lambda_1^n,$$

with a nonzero c. The product is taken for all k for which λ_k is different from λ_1.[108] The reason for this equation is that the difference operator $(E-\lambda_k)^{d+1}$ annihilates the term $P_k(n)\lambda_k^n$ when $\lambda_k \neq \lambda_1$ according to the Lemma above, (since $\deg P_k(n) \le d$). These operators will not change the degree of the polynomial in the term $P_1(n)\lambda_1^n$ according to the same Lemma (because $\lambda_k \neq \lambda_1$). The operator $(E-\lambda_1)^d$ will annihilate the term $P_k(n)\lambda_k^n$ in case $\lambda_k = \lambda_1$ and $k \neq 1$ (since $\deg P_k(n) < d$ in this case, according to our assumptions). Finally, the operator $(E-\lambda_1)^d$ lowers the degree of $P_1(n)$ by d in the term $P_1(n)\lambda_1^n$ according to the Lemma (while none of the other operators change the degree of $P_1(n)$ in this term, as we mentioned). Hence, after the application of the above difference operators, the resulting function will be $c\lambda_1^n$ with $c \neq 0$; this confirms the above equation. So, applying the difference operator to both sides of the equation expressing linear dependency, we obtain that

$$c\lambda_1^n \equiv 0,$$

while $c \neq 0$. This is a contradiction since $\lambda_1 \neq 0$ according to assumptions, showing that the functions in question are linearly independent. □

The solution of the recurrence equation.

Corollary (Solution of the Homogeneous Equation). *Assuming*

$$\sum_{k=0}^{m}a_k\zeta^k = a_m\prod_{j=1}^{N}(\zeta-\lambda_j)^{m_j}, \qquad \text{where} \quad \sum_{j=1}^{N}m_j = m,$$

and the λ_j's are pairwise distinct, the functions $n^r\lambda_j^n$ for r and j with $0 \le r < m_j$ and $1 \le j \le N$ represent m linearly independent solutions of the differential equation

$$\Big(\sum_{k=0}^{m}a_kE^k\Big)y_n = 0.$$

Proof. The linear independence of the functions claimed to be representing the solutions have been established in the first Corollary. Since a recurrence equation of order m can have at most m linearly independent solutions, these functions will represent a complete set of linearly independent solutions. To see that each of these functions is a solution, it is enough to note according to the equation

$$\sum_{k=0}^{m}a_kE^k = a_n\prod_{j=1}^{N}(E-\lambda_j)^{m_j}$$

[108] This arrangement is of course highly redundant, because if $\lambda_k = \lambda_l$, there is no need to take both of the factors $(E-\lambda_k)^{d+1}$ and $(E-\lambda_l)^{d+1}$, but such redundancy is harmless and it serves to simplify the notation.

that, in view of the Lemma above, the difference operator

$$(E - \lambda_j)^{m_j}$$

annihilates the function $n^r \lambda_j^n$ for $r < m_j$. $\square$

Thus we exhibited m linearly independent solutions of equation (1). If follows that any solution of (1) is a linear combination of these solutions.

Problems

1. The Fibonacci numbers y_n, $n = 0, 1, 2, \ldots$ are defined by the equations $y_0 = 0$, $y_1 = 1$ and $y_{n+2} = y_n + y_{n+1}$ for every integer $n \geq 0$. Write a formula expressing y_n.

Solution. The characteristic equation of the recurrence equation $y_{n+2} = y_n + y_{n+1}$ is $\zeta^2 = 1 + \zeta$. The solutions of this equation are

$$\zeta_1 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad \zeta_2 = \frac{1-\sqrt{5}}{2}.$$

Thus, the general solution of the above recurrence equation is

$$y_n = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

The initial conditions $y_0 = 0$ and $y_1 = 1$ lead to the equations

$$C_1 + C_2 = 0$$

and

$$C_1 \frac{1+\sqrt{5}}{2} + C_2 \frac{1-\sqrt{5}}{2} = 1.$$

It is easy to solve these equations. Multiplying the first equation by $1/2$ and subtracting it from the second equation, we obtain

$$\frac{\sqrt{5}}{2}(C_1 - C_2) = 1,$$

that is

$$C_1 - C_2 = \frac{2}{\sqrt{5}},$$

Adding the first equation to this, we obtain $2C_1 = 2/\sqrt{5}$, or else $C_1 = 1/\sqrt{5}$. Substituting this into the first equation, we obtain $C_2 = -1/\sqrt{5}$. With these values for C_1 and C_2, the formula for y_n gives

$$y_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

2. Write a difference operator that annihilates all but the first term in the expression

$$c_1 n^3 \cdot 3^n + c_2 n^4 \cdot 2^n + c_3 n^2 \cdot 5^n,$$

while it reduces the first term to $c \cdot 3^n$, where c is a nonzero constant (it is assumed that $c_1 \neq 0$).

Solution. The difference operator

$$(E-3)^3$$

will lower the degree of the polynomial in the first term to 0 (i.e., it will change the term into $c \cdot 3^n$ with a nonzero c), while it will not change the degrees of the other polynomials. The difference operator

$$(E-2)^5$$

will annihilate the second term, while it will not change the degrees of the polynomials in the other terms. Finally, the difference operator

$$(E-5)^3$$

will annihilate the third term, while it will not change the degrees of the polynomials Hence the product of these differential operators,

$$(E-3)^3(E-2)^5(E-5)^3$$

will change the first term into $c \cdot 3^n$ with a nonzero c, while it will annihilate the second and the third terms.

This argument can be used to show that if

$$c_1 n^3 \cdot 3^n + c_2 n^4 \cdot 2^n + c_3 n^2 \cdot 5^n \equiv 0,$$

then we must have $c_1 = 0$. Similar arguments can be used to show that we must also have $c_2 = 0$ and $c_3 = 0$; hence the terms $n^3 \cdot 3^n$, $n^4 \cdot 2^n$, and $n^2 \cdot 5^n$ are linearly independent.

31. NUMERICAL INSTABILITY

Numerical instability is the phenomenon when small errors committed early in the calculation cause excessively large errors later. An example for a numerically unstable method for solving the differential equation $y' = f(x, y)$ is Milne's method. Let $x_n = x_0 + nh$ with some x_0 and some $h > 0$. Write y_n for the calculated solution of the above equation at x_n, and put $f_n = f(x_n, y_n)$. Writing $\bar{y}_{n+1}$ for the predicted value of y at x_{n+1}, Milne's method has the predictor

$$\bar{y}_{n+1} = y_{n-3} + \frac{4h}{3}(2f_n - f_{n-1} + 2f_{n-2}).$$

Setting $\bar{f}_{n+1} = f(x_{n+1}, \bar{y}_{n+1})$, the corrector equation is

$$y_{n+1} = y_{n-1} + \frac{h}{3}(\bar{f}_{n+1} + 4f_n + f_{n-1}).$$

The predictor equation does not play too much role in out analysis of the method, since it only provides a starting value for the corrector equation. For the discussion that follows, instead of considering what happens during one application of the corrector equation, we will consider the behavior of the method when the equation

$$y_{n+1} = y_{n-1} + \frac{h}{3}(f_{n+1} + 4f_n + f_{n-1}). \tag{1}$$

is solved exactly, where we use f_{n+1} rather than $\bar{f}_{n+1}$ on the right-hand side to indicate this is an equation for y_n. Of course, this equation just expresses the use of Simpson's method in calculating the integral in the equation

$$y_{n+1} = y_{n-1} + \int_{x_{n-1}}^{x_{n+1}} f(x, y(x))\, dx.$$

We will consider what happens when equation (1) is used to solve the differential equation $y' = -Ky$ for some positive constant K. That is, we will put

$$f(x, y) = -Ky.$$

Equation (1) then becomes

$$y_{n+1} = y_{n-1} - \frac{hK}{3}(y_{n+1} + 4y_n + y_{n-1}). \tag{2}$$

The characteristic equation of this recurrence equation is

$$\zeta^2 = 1 - \frac{hK}{3}(\zeta^2 + 4\zeta + 1),$$

that is

$$(3 + hK)\zeta^2 + 4hK\zeta - (3 - hK) = 0.$$

The solutions of this equation are

$$Z_1 = \frac{-2hK + \sqrt{3h^2K^2 + 9}}{3 + hK} \quad \text{and} \quad Z_2 = \frac{-2hK - \sqrt{3h^2K^2 + 9}}{3 + hK}.$$

We will approximate these with error $O(h^2)$ by using the Taylor series for them at $h = 0$. Rather than directly calculating the Taylor series for Z_1 and Z_2, an easier way to do this may be to use the Taylor expansions

$$\frac{1}{3 + hK} = \frac{1}{3} \cdot \frac{1}{1 - (-hK/3)} = \frac{1}{3}(1 + (-hK/3) + O(h^2)) = \frac{1}{3}(1 - Kh/3 + O(h^2))$$

and

$$\sqrt{3h^2K^2 + 9} = 3\sqrt{1 + h^2K^2/3} = 3(1 + O(h^2)),$$

and substituting these into the expressions for Z_1 and Z_2. We obtain

$$Z_1 = 1 - Kh + O(h^2) \quad \text{and} \quad Z_2 = -1 - \frac{Kh}{3} + O(h^2).$$

Hence the general solution of equation (2) is

$$y_n = c_1 Z_1^n + c_2 Z_2^n = c_1(1 - Kh + O(h^2))^n + c_2(-1)^n(1 + Kh/3 + O(h^2))^n.$$

Suppose we are trying to solve the differential equation $y' = -y$ (that is, we are taking $K = 1$) with initial condition $x_0 = 0$, $y_0 = 1$, and, seeking the solution for some $x > 0$, we use $h = x/n$ for some large n. Noting that

$$\left(1 + \frac{u}{n} + O\left(\frac{1}{n^2}\right)\right)^n = e^u + O\left(\frac{1}{n}\right),$$

as $n \to \infty$,[109] and using this with $u = -x$ and $u = x/3$, we obtain that

$$y_n = c_1 e^{-x} + c_2(-1)^n e^{x/3} + O\left(\frac{1}{n}\right).$$

[109] Using the Taylor expansion $\log(1 + t) = t + O(t^2)$ as $t \to 0$ (here $\log t$ denotes the natural logarithm of t), we have

$$n \log\left(1 + \frac{u}{n} + O\left(\frac{1}{n^2}\right)\right) = u + O\left(\frac{1}{n}\right)$$

for fixed u when $n \to \infty$. Take the exponential of both sides and note that

$$e^{u + O(1/n)} = e^u \cdot e^{O(1/n)} = e^u \cdot (1 + O(1/n)) = e^u + e^u \cdot O(1/n).$$

Finally, $e^u \cdot O(1/n) = O(1/n)$ for fixed u.

The values of c_1 and c_2 are determined with the aid of the initial conditions. The choice $c_1 = 1$ and $c_2 = 0$ correctly gives the solution $y(x) = e^{-x}$ when $n \to \infty$. If, on the other hand, one only has $c_1 \approx 1$ and $c_2 \approx 0$ then, for large enough x, the term $c_2(-1)^n e^{x/3}$ will have much larger absolute value than the term $c_1 e^{-x}$, and the calculated solution will be nowhere near the actual solution of the equation.

The solution $c_1 e^{-x}$ is called the *main solution*, while solution $c_2(-1)^n e^{x/3}$ is called the *parasitic solution* of the recurrence equation (2). In a practical calculation one will not be able to suppress the parasitic solution (by ensuring that $c_2 = 0$). Because of normal roundoff errors, the parasitic solution will arise during the calculation even if the choice of initial conditions ensure that $c_2 = 0$ at the beginning.[110]

32. GAUSSIAN ELIMINATION

Consider the following system of linear equations

$$\begin{aligned} 3x_1 + \ 6x_2 - 2x_3 + 2x_4 &= -22 \\ -3x_1 - \ 4x_2 + 6x_3 - 4x_4 &= \ 34 \\ 6x_1 + 16x_2 + \ x_3 + \ x_4 &= -33 \\ -6x_1 - 18x_2 - 2x_3 - 2x_4 &= \ 36 \end{aligned}$$

Writing a_{ij} for the coefficient of x_j in the ith equation and b_i for the right-hand side of the ith equation, this system can be written as

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \qquad (1 \le i \le n) \tag{1}$$

with $n = 4$. To solve this system of equation, we write $m_{i1} = a_{i1}/a_{11}$, and subtract m_{i1} times the first equation from the ith equation for $i = 2$, 3, and 4. We have $m_{21} = -1$, $m_{31} = 2$, and $m_{41} = -2$. The following equations will result (the first equation is written down unchanged):

$$\begin{aligned} 3x_1 + 6x_2 - 2x_3 + 2x_4 &= -22 \\ 2x_2 + 4x_3 - 2x_4 &= \ 12 \\ 4x_2 + 5x_3 - 3x_4 &= \ 11 \\ -6x_2 - 6x_3 + 2x_4 &= -\ 8 \end{aligned}$$

Write $a_{ij}^{(2)}$ for the coefficient of x_j in the ith equation. We will continue the above process: writing $m_{i2} = a_{i2}^{(2)}/a_{22}^{(2)}$ and subtract m_{i2} times the second equation from the ith equation for $i = 3$ and $i = 4$. We have $m_{32} = 2$ and $m_{42} = -3$. We obtain the equations

$$\begin{aligned} 3x_1 + 6x_2 - 2x_3 + 2x_4 &= -22 \\ 2x_2 + 4x_3 - 2x_4 &= \ 12 \\ -3x_3 + \ x_4 &= -13 \\ 6x_3 - 4x_4 &= \ 28 \end{aligned}$$

[110] In order to start the method, one needs initial conditions for y_0 and y_1. In so far as one obtains y_1 by some approximate solution of the differential equation, it is likely that there will be some truncation error in the calculation, so one will not have $c_2 = 0$. In fact, even if one enters the correct solution $y(x_1)$ as the value of y_1, the parasitic solution will not be suppressed, since the correct solution of the differential equation will not be the same as the main solution of the recurrence equation, since the latter only approximates the solution of the differential equation. That is, the situation is hopeless even without roundoff errors.

Write $a_{ij}^{(3)}$ for the coefficient of x_j in the ith equation. Write $m_{43} = a_{43}^{(3)}/a_{33}^{(3)}$, and subtract m_{43} times the third equation from the fourth equation. We have $m_{43} = -2$, and the following equations result:

$$\begin{aligned} 3x_1 + 6x_2 - 2x_3 + 2x_4 &= -22 \\ 2x_2 + 4x_3 - 2x_4 &= \quad 12 \\ -3x_3 + \ x_4 &= -13 \\ -2x_4 &= \quad 2 \end{aligned}$$

The above method of solving is called *Gaussian elimination*, named after the German mathematician Carl Friedrich Gauss. The last system of equation is called *triangular* because of the shape formed by the nonzero coefficients That is, writing a_{ij}^4 for the coefficient of x_j and $b_i^{(4)}$ for the write hand side in the ith equation in the last system of equations, we have $a_{ij}^{(4)} = 0$ for $j < i$.

The resulting equations are easy to solve. From the fourth equation we obtain

$$x_4 = \frac{b_4^{(4)}}{a_{44}^{(4)}} = \frac{2}{-2} = -1.$$

Using this value of x_4, we can determine x_3 from the third equation

$$x_3 = \frac{b_3^{(4)} - a_{34}^{(4)} x_4}{a_{33}^{(4)}} = \frac{-13 - 1 \cdot (-1)}{-3} = 4.$$

Next, using the value of x_3 and x_4, we can determine the value of x_2 from the second equation:

$$x_2 = \frac{b_2^{(4)} - a_{23}^{(4)} x_3 - a_{24}^{(4)} x_4}{a_{22}^{(4)}} = \frac{12 - 4 \cdot 4 - (-2) \cdot (-1)}{2} = -3.$$

Finally, using the values of x_2, x_3, and x_4, we can determine x_1 from the first equation:

$$x_1 = \frac{b_1^{(4)} - a_{12}^{(4)} x_2 - a_{13}^{(4)} x_3 - a_{14}^{(4)} x_4}{a_{11}^{(4)}} = \frac{-22 - 6 \cdot (-3) - (-2) \cdot 4 - 2 \cdot (-1)}{3} = 2.$$

This way is of solving a triangular system of equations is called *back substitution*.

In a general description of the method of solving equations (1), one starts with the coefficients a_{kj} of the equations. Taking $a_{kj}^{(1)} = a_{kj}$, one defines the coefficients m_{ki} when $1 \le i < k < n$ and $a_{kj}^{(i)}$ when $1 \le k \le n$, $1 \le j \le n$, and $1 \le i \le k$ as follows: when it is the turn to use the ith equation to eliminate the coefficients of x_i in the kth equation for $i < k \le n$, one defines

$$m_{ki} = a_{ki}^{(i)}/a_{ii}^{(i)} \tag{2}$$

(assuming that $a_{ii}^{(i)} \neq 0$), and one defines the new value of a_{kj} as

$$a_{kj}^{(i+1)} = a_{kj}^{(i)} - m_{ki} a_{ij}^{(i)} \qquad (1 \le i < k \le n,\ 1 \le j \le n); \tag{3}$$

note that for $i = j$ this gives $a_{ki}^{(i+1)} = 0$, showing that x_i is eliminated from the kth equation for $k > i$. It is now easy to see by induction on i from (3) that $a_{kj}^{(i)} = 0$ if $j < i \le k$. The final value of the coefficient a_{ij} in these calculation will be $a_{ij}^{(i)}$. We put

$$a_{ij}^{\text{new}} \stackrel{def}{=} a_{ij}^{(i)}. \tag{4}$$

According to (4), equation (3) can be rewritten as

$$a_{kj}^{(i+1)} = a_{kj}^{(i)} - m_{ki} a_{ij}^{\text{new}} \qquad \text{for } i < k.$$

By a repeated application of this formula, we obtain from (4) that

$$a_{kj}^{\text{new}} = a_{kj} - \sum_{i=1}^{k-1} m_{ki} a_{ij}^{\text{new}}. \tag{5}$$

Write $m_{kk} = 1$ and $m_{ki} = 0$ for $k < i$; then the last equation can also be written as

$$a_{kj} = \sum_{i=1}^{n} m_{ki} a_{ij}^{\text{new}}.$$

One can also express these equations in matrix form. Writing

$$A = (a_{ij}), \quad L = (m_{ij}), \quad \text{and} \quad U = (a_{ij}^{\text{new}})$$

for the $n \times n$ matrices with the indicated elements, the last equation in matrix form can be written as

$$A = LU.$$

This is called the *LU-factorization* of A. The name L is used because the matrix it designates is a *lower-triangular* matrix, i.e., a matrix whose elements above the main diagonal are zero, and the letter U is used because the matrix it denotes is an *upper-triangular* matrix, i.e., a matrix whose elements below the main diagonal are zero.

Equation (1) can be written in the form

$$A\mathbf{x} = \mathbf{b},$$

where x is the column vector of unknowns, and b is the column vector of the right-hand sides, i.e.,

$$\mathbf{x} = (x_1, x_2, \dots, x_n)^T \quad \text{and} \quad \mathbf{b} = (b_1, b_2, \dots, b_n)^T,$$

where T in superscript stands for transpose.[111] As $A = LU$, this equation can now be solved as

$$\mathbf{x} = U^{-1} L^{-1} \mathbf{b}$$

Multiplication by the inverse matrices U^{-1} and L^{-1} is easy to calculate. Writing

$$\mathbf{b}^{\text{new}} = (b_1^{\text{new}}, b_2^{\text{new}}, \dots, b_n^{\text{new}}) \stackrel{def}{=} L^{-1}\mathbf{b}$$

the elements b_i^{new} are calculated analogously to the way the coefficients a_{ij}^{new} are calculated in formula (5):

$$b_i^{\text{new}} = b_i - \sum_{l=1}^{i-1} m_{il} b_l^{\text{new}}.$$

This formula is called *forward substitution*. In the numerical example above, we calculated the right-hand sides simultaneously with the coefficients in the LU-factorization. Often, one wants to solve systems of equations with the same left-hand side but different right-hand sides; for this reason, it is usually advantageous to separate the calculation of the left-hand side from that of the right-hand side. The solution of the equation then can be written in the form $x = U^{-1}\mathbf{b}^{\text{new}}$. Multiplication by the inverse of U can be calculated with the aid of back-substitution, already mentioned above:

$$x_i = \frac{b_i^{\text{new}} - \sum_{j=i+1}^{n} a_{ij}^{\text{new}} x_j}{a_{ii}^{\text{new}}}.$$

[111] Writing column vectors takes up too much space in text, and so it is easier to describe a column vector as the transpose of a row vector. In general, the transpose of an $m \times n$ matrix $A = (a_{ij})$ is an $n \times m$ matrix $B = (b_{ij})$ with $b_{ij} = a_{ji}$ for any i, j with $1 \le i \le n$ and $1 \le j \le m$. A row vector can be considered a $1 \times n$ matrix, while a column vector, an $n \times 1$ matrix. Hence a column vector can be written as the transpose of a row vector.

Pivoting. In equation (2), it was assumed that $a_{ii}^{(i)} \neq 0$. This may, of course, not be the case, and one cannot proceed further along the lines described. One gets into trouble even if a_{ii} is only close to zero, and not actually equal to zero. To deal with this problem, one interjects the following step. Instead of using the ith equation in continuing the calculations, one looks for the number with with the largest absolute value among the coefficients $a_{ki}^{(i)}$ for $k \geq i$. If this coefficient is $a_{ri}^{(i)}$ then one interchanges the ith equation and the rth equation before continuing the calculation. It may happen that we have $a_{ki}^{(i)} = 0$ for all $k \geq i$; in this case, however, the system of equations is not uniquely solvable.

That is, the system of equations is either unsolvable or it has infinitely many solutions. To see this, assume that the system formed by $i+1$st through the nth equations at *this stage* of the process is solvable for x_k with $i < k \leq n$ (these are the only unknowns occurring in the latter equations). In this case we can chose x_i arbitrarily, and then we can solve the system for the unknowns x_{i-1}, x_{i-2}, ..., x_1 by back substitution. If on the other hand, the system formed by $i+1$st through the nth equations at *this stage* of the process is not solvable for x_k with $i < k \leq n$, then the original system of equations in not solvable, either.

This method of interchanging of equations is called *partial pivoting*, and the new element moved in the position of a_{ii} is called the *pivot* element.

Instead of partial pivoting, one can use what is called *full pivoting*. In full pivoting one looks for the coefficient with the larges absolute value among all the coefficients $a_{kj}^{(i)}$ for $k \geq i$ and $j \geq i$, and moves this element into the position of the element a_{ii} by interchanging rows and columns of the coefficient matrix. (The interchange of rows corresponds to interchanging equations, and the interchange of columns corresponds to interchanging unknowns.) In practice, full pivoting is rarely done because it is much more complicated to implement than partial pivoting, and there is in practice no significant advantage of full pivoting over partial pivoting. Partial pivoting can lead one astray in the system of equations

$$\begin{aligned} .000,01x_1 + \quad .000,001x_2 &= -\ .000,02 \\ .000,02x_1 - 10 \qquad\quad x_2 &= \quad 3 \end{aligned}$$

Interchanging the two equations in this case would be a mistake. In fact, dividing each equation by the coefficient of the maximum absolute value on its left-hand side we obtain an equivalent system of equations.

$$\begin{aligned} x_1 + .1x_2 &= -2 \\ -.000,002x_1 + \quad x_2 &= -\ .3 \end{aligned}$$

Using partial pivoting with this system of equations, one would not interchange the equations. The point is that the working of partial pivoting may be distorted if the equations are "differently scaled." If this is the case, it may be simpler to rescale each equation by dividing through with the coefficient with the maximum absolute value on its left-hand side.

In fact, even the working of full pivoting is affected by the equations being "differently scaled." Indeed, with the former system, one would use the coefficient -10 of x_2 in the second equation as pivot element, whereas with the latter, rescaled system, one would use the coefficient 1 of x_1 in the first equation as pivot element. However, both of these are good choices, and full pivoting can work well without rescaling the equations.

In a theoretical discussion of partial pivoting, it is helpful to imagine that all row interchanges happen in advance, and then one carries out Gaussian elimination on the obtained system of equations without making further interchanges.[112] A sequence of interchanges of rows of a matrix results in a permutation of the rows of a matrix. Such a permutation can be described in terms of a *permutation matrix* $P = (p_{ij})$. A permutation matrix can be characterized by saying that each row and each column has exactly one element equal to 1, with the rest of the elements being equal to 0.

[112]Of course, in practice this cannot be done, but in a theoretical discussion of algorithms one often considers *oracles*. An oracle is a piece of information, unobtainable in practice, that determines a step to be taken in a algorithm.

If $p_{ij} = 1$ in an $n \times n$ permutation matrix then, given an $n \times n$ matrix A, in the matrix PA the ith row of the matrix PA will be the jth row of the matrix A, as one can easily check by the equation

$$(PA)_{il} = \sum_{k=1}^{n} p_{ik} a_{kl} = a_{jl},$$

where $(PA)_{il}$ denotes the element in the ith row and lth column of the matrix PA.

Given a system of equations $A\mathbf{x} = \mathbf{b}$, Let P is the permutation matrix describing the row interchanges performed during the solution of the equations. Write $A' = PA$. Then one obtains an LU-factorization of the matrix A', i.e., one finds a representation $A' = LU$ where L is a lower triangular matrix, and U is an upper triangular matrix, with the additional assumption that the elements in the main diagonal of the matrix L all equal to 1. That is, one can write $PA = LU$, or

$$A = P^{-1}LU.$$

It is easy to see that the inverse of a permutation matrix is its transpose; that is, finding the inverse of a permutation matrix presents no problems in practice.

Suppose $p_{ij} = 1$ in the permutation matrix P. Writing $A' = PA$, then the ith row of the matrix A' will be the jth row of the matrix A. Writing $A'' = P^T A'$ then $(P^T)_{ji} = 1$; so the jth row of the matrix A'' will be the i row of the matrix A'. Therefore, the matrix A'' is the same as the matrix A. Hence $P^T P = I$, where I is the $n \times n$ identity matrix. Similarly, we have $PP^T = (P^T)^T P^T = I$, so P^T is the inverse of P.

This last step is perhaps unnecessary, since if a matrix B has a left inverse, then it is nonsingular, and if it is nonsingular, then it must also have a right inverse. If Q is a left inverse, and R is a right inverse of a matrix B, then $QB = BR = I$, so $Q = QI = Q(BR) = (QB)R = IR = R$, i.e., the left and right inverses must agree. But this argument is more complicated than the above argument involving the double transpose.

When one performs Gaussian elimination on the matrix A with partial pivoting, one in effect obtains a permuted LU-factorization $A = P^{-1}LU$ of the matrix A. Given the equation

$$A\mathbf{x} = \mathbf{b},$$

one can then find the solution in the form

$$\mathbf{x} = U^{-1}L^{-1}P\mathbf{b}.$$

Calculating the vector $P\mathbf{b}$ amounts to making the same interchanges in the components of the vector $\mathbf{b}$ that were made to the rows of the matrix A. We explained above that multiplying by L^{-1} can be calculated by forward substitution, and multiplying by U^{-1} can be calculated by back substitution.

The inverse of the matrix A can be written as $U^{-1}L^{-1}P$. However, one usually does not want to evaluate the inverse matrix; that is, if one needs to calculate $A^{-1}B$ for some matrix B, then one calculates the matrices PB, $L^{-1}(PB)$, and $U^{-1}(L^{-1}PB)$ rather than first evaluating A^{-1} and then evaluating $A^{-1}B$.

Iterative improvement. In a practical calculation, the elements of the matrices L and U in the factorization $A = P^{-1}LU$ can only be approximately determined. Therefore the solution obtained as

$$\mathbf{x}^{(0)} = U^{-1}L^{-1}P\mathbf{b}.$$

will only be approximate. Writing

$$\delta\mathbf{b}^{(1)} = \mathbf{b} - A\mathbf{x}^{(0)},$$

one can then obtain an approximate solution of the equation

$$A\delta\mathbf{x}^{(1)} = \delta\mathbf{b}^{(1)}$$

as

$$\delta\mathbf{x}^{(1)} = U^{-1}L^{-1}P\delta\mathbf{b}^{(1)}.$$

One can then expect that

$$\mathbf{x}^{(1)} = x^{(0)} + \delta\mathbf{x}^{(1)}$$

is a better approximation of the true solution of the equation $A\mathbf{x} = \mathbf{b}$ than vector $\mathbf{x}^{(1)}$ is. This calculation is called (one step of) iterative improvement. One can perform this step repeatedly to obtain a better approximation to the solution. The calculation of $\delta\mathbf{b}^{(1)}$ above should be done with double or extended precision; since the vectors $\mathbf{b}$ and $A\mathbf{x}^{(0)}$ are close to each other, using extended precision helps one to avoid the loss of significant digits. The rest of the calculation can be done with the usual (single or double) precision.[113]

Computer implementation. We use the header file `gauss.h` in the computer implementation of Gaussian elimination:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 #include <stdlib.h>
 4
 5 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 6
 7 void swap_pivot_row(double **a, int col, int n);
 8 void LUfactor(double **a, int n, int *success);
 9 void LUsolve(double **a, int n, double *b);
10 void backsubst(double **c, int n, double *b, double *x,
11           double *deltab);
12 int improve(double **a, int n, double **c, double *b,
13     double *x, double tol, int maxits, int *success);
14 double **allocmatrix(int n);
15 double *allocvector(int n);
```

Lines 5-15 contain the declarations of functions used in the program. These declarations will be explained later, when discussing the respective functions. The file `alloc.c` contains the routines allocating memory for the vectors and matrices used in the programs:

```
 1 #include "gauss.h"
 2
 3 double **allocmatrix(int n)
 4   /* Allocate matrix. The following code segment allocates
 5      a contiguous block of pointers for the whole matrix.
 6      There is no real advantage for this here, because with
 7      pivoting, rows will be interchanged. So memory for rows
 8      could be allowcated separately. On the other hand, even
 9      the largest matrix for which Gaussian elimination
10      is feasible occupies a relatively modest amount of
11      memory, there does not seem to be any disadvantage
12      in asking for a contiguous block. */
13 {
14   int i;
15   double **a;
```

[113] Most or all the programs in these notes use data type `double` for floating point numbers. This is already double precision, single precision being represented by data type `float`. Extended precision in this case can be represented by data type `long double`. Indeed, `long double` is used in the program below to calculate $\delta\mathbf{b}^{(1)}$.

```
16   a=(double **) malloc((size_t)((n+1)*sizeof(double*)));
17   if (!a) {
18     printf("Allocation failure 1 in matrix\n");
19     exit(1);
20   }
21   /* Allocate the whole matrix: */
22   a[1]=(double *) malloc((size_t)(n*(n+1)*sizeof(double)));
23   if (!a[1]) {
24     printf("Allocation failure 2 in matrix\n");
25     exit(1);
26   }
27   /* Initialize the pointers pointing to each row: */
28   for (i=2; i<=n;i++) a[i]=a[i-1]+n+1;
29   /* Save the beginning of the matrix in a[0]; the
30      rows will be interchanged, the values of a[i]
31      for 1<=i<=n may change:                      */
32   a[0]=a[1];
33   return a;
34 }
35
36 double *allocvector(int n)
37 {
38   double *b;
39   b=(double *) malloc((size_t)((n+1)*sizeof(double)));
40   if (!b) {
41     printf("Allocation failure 1 in vector\n");
42     exit(1);
43   }
44   return b;
45 }
```

We will first discuss the function `allocvector` in lines 36–45. A vector of reals (of type `double`) will be represented a by a block of $n+1$ reals pointed to by a pointer (called `b` on line 38); here n is the number of unknowns and also the number of equations we are dealing with. The elements of this matrix can be referred to as `b[i]` for $0 \leq i \leq n$ (they can also be referred to as `*(b+i)`). In line 40, it is tested whether `!b` is true, i.e., whether `b` is the `NULL` pointer, indicating that the memory was not successfully allocated.

In lines 3–34 the function `allocmatrix` first allocates a block of $n+1$ pointers to reals (of the type `double`) on line 16, and then a block of $n(n+1)$ reals is allocated, and the address of the beginning of this block is placed in the pointer `a[1]`. Then the pointers `a[i]` are initialized for $2 \leq i \leq n$ on line 28 in such a way that each pointer `a[i]` will point to a block of $n+1$ reals; these reals can be invoked as `a[i][j]` for $0 \leq j \leq n$; the element `a[i][0]` will have special uses (at times it will contain the right-hand side of the equation, at other times it will contain the solution vector), and the element `a[i][j]` for $1 \leq i, j \leq n$ will refer to the matrix element a_{ij}. Thus `a[i]` can be said to refer to row i of the matrix; the interchange of row i and j can be handled by simply switching the values of `a[i]` and `a[j]`. On line 32, the pointer `a[0]` is given the value of `a[1]`; thus `a[0]` will also point to the beginning of the block of $n(n+1)$ reals. This is important when deallocating this block: the value of `a[0]` will not change, while that of `a[1]` may change when interchanging rows.

The file `lufactor.c` contains the functions doing the LU-factorization of the matrix A:

```
1 #include "gauss.h"
2
```

```
 3 void swap_pivot_row(double **a, int col, int n)
 4   /* Scans the elements a[col][i] for col<=i<=n to find the
 5      element c[col][pivi] with the largest absolute value,
 6      and then the rows a[col] and a[pivi] are interchanged. */
 7 {
 8   double maxel, *rowptr;
 9   int i,pivi;
10   maxel=absval(a[col][col]); pivi=col;
11   for (i=col+1;i<=n;i++) {
12     if ( absval(a[i][col])>maxel ) {
13       maxel=absval(a[i][col]); pivi=i;
14     }
15   }
16   if ( pivi !=col ) {
17     rowptr=a[col]; a[col]=a[pivi]; a[pivi]=rowptr;
18   }
19 }
20
21 void LUfactor(double **a, int n, int *success)
22   /* Uses partial pivoting */
23 {
24   const double assumedzero=1e-20;
25   double pivot, mult;
26   int i, j, k;
27   for (j=1;j<n;j++) {
28     swap_pivot_row(a, j, n); pivot=a[j][j];
29     if ( absval(pivot)<=assumedzero ) {
30       *success=0; return;
31     }
32     else *success=1;
33     for (i=j+1; i<=n; i++) {
34       mult = a[i][j] /= pivot;
35       for (k=j+1;k<=n;k++) a[i][k]-=mult*a[j][k];
36     }
37   }
38 }
```

In lines 3–19, the function `swap_pivot_row` is defined. The parameters of this function is the matrix `**a` (described as a pointer of pointers to `double`, as explained above), the column `col` and the size of the matrix `n`. In line 10, `absval` finds the absolute value of a number (as defined in line 5 of the file `gauss.h` above). In lines 10–14 the element of the largest absolute value in or below the main diagonal of column `col` found, and if this element is `a[pivi][col]` (`pivi` refers to pivoting `i`, rows `col` and `pivi` are interchanged in lines 16–18. The interchange is performed simply by exchanging the values of the pointers `a[col]` and `a[pivi]`. In the end, one can find out what interchanges have been made simply by examining the values of the pointers `a[i]` for i with $1 \leq i \leq n$.

In lines 21–38 to function LUfactor performs LU-factorization with partial pivoting of the matrix given as parameter `**a`. The other parameters of this function are the size of the matrix `n`, and a pointer to the integer `*success`; this will become true if LU-factorization was successfully performed. The outer loop in lines 27–37 first places the pivot element in the main diagonal in line 28 by interchanging appropriate rows. If no appropriate pivot element if found, the value of the variable `*success` is changed to 0 (false) on line 30. In the loop in lines 33–45, row i of the matrix for $j = 1 \leq i \leq n$ is updated: In line 34, the element m_{ij} is calculated; m_{ij} will be stored in location

`a[i][j]`, that is, as element a_{ij} of the matrix; recall that $i > j$ at this point. That is, for $i > j$, the element a_{ij} will be replaced by the element m_{ij} of the matrix L. For $i \leq j$, the element a_{ij}^{new} of the matrix U will replace the matrix element a_{ij}. The zero matrix elements (m_{ij} for $i < j$) and a_{ij}^{new} for $i > 0$) do not need to be stored. The elements m_{ii} are equal to 1, and these do not need to be stored, either. Finally, in line 35, an appropriate multiple of row i is subtracted from row j, and the inner loop ends on line 36.

The file `lusolve.c` contains the function needed to solve the equation once the LU-factorization is given, and those needed for iterative improvement.

```
 1 #include "gauss.h"
 2
 3 void LUsolve(double **a, int n, double *b)
 4 {
 5   int i, j;
 6   /* We want to solve P^-1 LUx=b. First we calculate c=Pb.
 7      The elements of c[i] will be stored in a[i][0], not used
 8      in representing the matrix. */
 9   for (i=1;i<=n;i++) a[i][0]=b[1+(a[i]-a[0])/(n+1)];
10   /* Here is why we preserved the beginning of the matrix
11      in a[0]. The quantity 1+(a[i]-a[1])/(n+1) gives
12      the original row index of what ended up as row i
13      after the interchanges. Next we calculate y=Ux=L^-1 c.
14      y will be stored the same place c was stored:
15      y[i] will be a[i][0]. */
16   for (i=2;i<=n;i++)
17     for (j=1;j<i;j++) a[i][0] -= a[i][j]*a[j][0];
18   /* Finally, we calculate x=U^-1 y. We will put x in
19      the same place as y: x[i]=a[i][0]. */
20   a[n][0] /= a[n][n];
21   for (i=n-1;i>=1;i--) {
22     for (j=i+1;j<=n;j++) a[i][0] -= a[i][j]*a[j][0];
23       a[i][0] /= a[i][i];
24   }
25 }
26
27 void backsubst(double **c, int n, double *b, double *x,
28              double *deltab)
29 {
30   long double longdelta;
31   int i,j;
32   for (i=1;i<=n;i++) {
33     longdelta=b[i];
34     for (j=1;j<=n;j++) longdelta -= c[i][j]*x[j];
35     deltab[i]=longdelta;
36   }
37 }
38
39 int improve(double **a, int n, double **c, double *b,
40     double *x, double tol, int maxits, int *success)
41 {
42   const double assumedzero=1e-20;
```

```
43   int i, j;
44   double maxx, maxdeltax, *deltab, rel_error;
45   deltab=allocvector(n);
46   for (i=1;i<=n;i++) {
47     x[i]=0.; deltab[i]=b[i];
48   }
49   *success=0;
50   /* If the process is successful, *success will be
51      changed to 1.                                    */
52   for (j=1; j==1 || (j<=maxits);j++) {
53     LUsolve(a, n, deltab);
54     maxx=0.; maxdeltax=0.;
55     for (i=1;i<=n;i++) {
56       x[i] += a[i][0];
57       if ( absval(x[i])>maxx ) maxx=absval(x[i]);
58       if ( absval(a[i][0])>maxdeltax ) maxdeltax=absval(a[i][0]);
59     }
60     rel_error=maxdeltax/(maxx+assumedzero);
61     if ( j==2 && rel_error > 0.5 ) {
62       printf("Iterative refinement will not converge\n");
63       break;
64       /* Cannot return from here since memory must be freed */
65     }
66     if ( rel_error <= tol ) {
67       *success=1;
68       break;
69     }
70     backsubst(c, n, b, x, deltab);
71   }
72   free(deltab);
73   return j;
74 }
```

The function `LUsolve` is given in lines 3–25. The parameters for this function are the matrix `**a`, the size `n` of this matrix, and the vector `*b` of the right-hand side of the equation. When this function is called, the matrix `**a` should contain the LU-factorization of the original coefficient matrix; i.e., in the lower half, the matrix L is stored, while in the upper half the matrix U is stored; the details were explained above. In line 9, the elements of the vector `*b` are permuted the way the rows of the matrix were permuted, and they are placed in column zero of the matrix `**a`; this column is not used to store matrix elements. The comment in lines 10–14 explains how this permutation is preserved in the values of the pointers `a[i]`. In lines 16–17, the vector $L^{-1}P\mathbf{b}$ is calculated (by doing the same transformation to the elements of the vector `*b` as was done to the rows of the matrix); the elements of this vector will replace the elements of the vector $P\mathbf{b}$ in column zero of the matrix `**a`. Finally, backward substitution is performed in lines 20–24. Again, column zero of the matrix is used to store the solution vector $\mathbf{x}$ of the system of equations; when the value of x_i is placed in location `a[i][0]`, this location is no longer needed to store the corresponding element of the vector $\mathbf{b}$ (or, rather, the vector $L^{-1}P\mathbf{b}$, since this vector replaced the elements of the vector $P\mathbf{b}$ originally placed here).

In line 27 the function `backsubst` substitutes the solutions of the system of equation into the original equation; the coefficients of the original equation are stored in the matrix `**c` (the original coefficient matrix needed to be copied to another location, since this matrix was overwritten when LU-factorization was performed). The other parameters of the function `backsubst` are the size of the matrix `n`, the right-hand side `*b` of the equation, the solution vector `*x` of the equation,

and `*deltab`, the difference the actual right-hand side and the one calculated by substituting the solutions into the equation; that is the vector $\delta\mathbf{b}$ obtained as $\delta\mathbf{b} = \mathbf{b} - A\delta x$. The important point here is that the the components of this vector are calculated with extended precision; that is, the identifier `longdelta` is declared `long double` on line 30, and this the values obtained during the calculation of the components of the vector $\delta\mathbf{b}$ are accumulated in `longdelta`.

Iterative improvement is performed by the function `improve` in lines 39–74. The parameters of this function are `**a`, which at this time holds the LU-factorization on the coefficient matrix, the size of the matrix `n`, the original coefficient matrix `**c`, the original right-hand side `*b`, a vector `*x` for the solution vector (eventually returned to the calling program), the maximum allowed number of iterations `maxits`, and the integer `*success` indicating success or failure of iterative improvement. In line 42, the constant `assumedzero` is defined to be 10^{-20}, to use to test when floating point numbers are too close to usefully distinguish. In line 45, the function `allocvector` (discussed above) is called to allocate memory to the vector `deltab`, which stands for $\delta\mathbf{b}$; on line 72, this memory is freed. In lines 46–48, the vector $\mathbf{x}$ is initially set to 0, while $\delta\mathbf{b}$ made equal to $\mathbf{b}$, the original right-hand side of the equation. In line 49, `*success` is given the value 0 (false), and it will be explicitly changed to 1 later if iterative improvement is successful. The loop in lines 52–71 does repeated iterative improvement. Whatever the value of `maxits`, the maximum number of iterations, this loop is performed at least once, to calculate the solution of the system of equations; iterative improvement is done for values of $j > 1$. On line 54, `LUsolve` is called to solve the system of equations with $\delta\mathbf{b}$ as right-hand side (which is $\mathbf{b}$ for $j = 1$), and then, in lines 55-59 the new value of the vector $\mathbf{x}$ (the current solution) is calculated: The function `LU-solve` stored $\delta\mathbf{x}$, the solution of the equation $A\delta\mathbf{x} = \delta\mathbf{b}$ in column zero of the matrix A on line 53. On line 56, the new value $\mathbf{x}+\delta\mathbf{x}$ is given to the vector $\mathbf{x}$. Then the l^∞ norm[114] `maxx`$= \|x\|$ of the vector $\mathbf{x}$, and the l^∞ norm `maxdeltax` $\|\delta\mathbf{x}\|$ of $\delta\mathbf{x}$ is calculated. In line 61, the relative error of the solution is calculated. This is done by taking the ratio $\|\delta\mathbf{x}\|/\|\mathbf{x}\|$, but the constant `assumedzero` is added to the denominator to avoid a *floating-point exception* caused by dividing by 0. In lines 61 it is checked whether the initial relative error (for $j = 2$, when the first values of $\mathbf{x}$ and $\delta\mathbf{x}$ have been calculated[115]) is greater than 1/2. If it is, the relative error is too large, iterative improvement is unlikely to be useful, and it is abandoned.[116] If the relative error is less than `tol`, the integer `*success` is given value 1 to indicate the success of the calculation, and the loop in lines 52–73 is terminated. In line 72, the memory used for the vector $\delta\mathbf{b}$ is freed, and the number of steps used for iterative improvement is returned on line 73.

```
 1 #include "gauss.h"
 2
 3 main()
 4 {
 5   /* This program reads in the coefficient matrix
 6      of a system of linear equations. The first
 7      entry is the required tolerance, and the
 8      second entry is n, the number of unknowns.
 9      The rest of the entries are the coefficients
10      and the right-hand sides; there must be n(n+1)
11      of these. The second entry must be an unsigned
12      integer, the other entries can be integers or
13      reals.                                      */
14   double **a, *b, *x, **c, tol, temp;
15   int i, j, n, readerror=0, success;
```

[114] The l^∞ norm of a vector is the maximum of the absolute values of its components. The l^∞ norm of a vector $\mathbf{v}$ is usually denoted as $\|v\|_\infty$; when it is clear from the context what type of norm is taken, one may simply write $\|v\|$ instead.

[115] For $j = 1$ at this point both $\mathbf{x}$ and $\delta\mathbf{x}$ are equal to the initial solution of the equation $A\mathbf{x} = \mathbf{b}$

[116] The inequality $\|\delta\mathbf{x}/\mathbf{x}\| > 1/2$ means roughly that even the first significant binary digit of $\|\mathbf{x}\|$ is wrong.

```
16  char s[25];
17  FILE *coefffile;
18  coefffile=fopen("coeffs", "r");
19  fscanf(coefffile, "%s", s);
20  tol=strtod(s,NULL);
21  printf("Tolerance for relative L-infinity"
22         " error used: %s\n", s);
23  fscanf(coefffile, "%u", &n);
24  /* Create pointers to point to the rows of the matrix: */
25  a=allocmatrix(n);
26  /* Allocate right-hand side */
27  b=allocvector(n);
28  for (i=1;i<=n;i++) {
29    if ( readerror ) break;
30    for (j=1;j<=n+1;j++) {
31      if (fscanf(coefffile, "%s", s)==EOF) {
32        readerror=1;
33        printf("Not enough coefficients\n");
34        printf("i=%u j=%u\n", i, j);
35        break;
36      }
37      if ( j<=n ) {
38        a[i][j]=strtod(s,NULL);
39      }
40      else b[i]=strtod(s,NULL);
41    }
42  }
43  fclose(coefffile);
44  if ( readerror ) printf("Not enough data\n");
45    else {
46    /* duplicate coefficient matrix */
47    c=allocmatrix(n);
48    for (i=1;i<=n;i++)
49      for (j=1;j<=n;j++) c[i][j]=a[i][j];
50    /* allocate vector for iterative improvement of solution */
51    x=allocvector(n);
52    LUfactor(a, n, &success);
53    /* Call improve here */
54    if ( success ) j=improve(a,n,c,b,x,tol,100,&success);
55    if ( success ) {
56      printf("%u iterations were used to find"
57             " the solution.\n", j);
58      printf("The solution is:\n");
59      for (i=1;i<=n;i++)
60        printf("x[%3i]=%16.12f\n",i,x[i]);
61    }
62    else
63      printf("The solution could not be determined\n");
64    /* must free c[0], c, and x inside this else statement,
65       since if this statement is not entered, space is not
66       reserved for x and c */
```

```
67     free(c[0]);
68     free(c);
69     free(x);
70   }
71   /* We must free a[0] and not a[1], since a[1] may
72      have been changed. */
73   free(a[0]);
74   /* free matrix */
75   free(a);
76   free(b);
77 }
```

The file main.c has the program reading the input data: the tolerance `tol`, value of n, the coefficient matrix, and the right-hand side of the equation, and then calls the programs to solve the equation. The input data are contained in the file `coeffs`: the first entry is `tol`, the second one is n, and then coefficients and the right-hand sides of the equations. The data can be separated by any kind of *white space* (i.e. spaces, newlines, tabs; it is helpful to write an equation an its right-hand side on a single line, but this is not required). On line 18, the file `coeffs` is opened, and `tol` is read in lines 19–20 (it is read as a character string on line 18, and it is converted to `double` on line 20). On line 25, memory is reserved for the coefficient matrix, and on line 27, this is done for the right-hand side vector of the equation. the coefficient matrix and the right-hand side is read in the loop in lines 28–42 (as character strings, then converted to `double` and placed in the correct location in lines 38 and 40). The integer `readerror` indicates whether there is a sufficient number of input data in the file `coeffs`. In line 43, the file `coeffs` is closed.

If on line 44 one finds that there was a reading error, there is not much one can do other than print that there are "not enough data." Otherwise, on line 47 one allocates space for a copy `**c` of the coefficient matrix, and and in lines 48–49 the coefficient matrix is duplicated. In line 51, memory is allocated for the solution vector `*x`. On line 52, the function `LUfactor` is called, and if LU-factorization is successful, then `improve` is called to perform iterative improvement on line 54; note that the parameter `100` indicates that at most 100 steps of iterative improvement are allowed. If this is done successfully, the solution is printed out in lines 55–61, otherwise a message saying that "the solution could not be determined" is printed on line 63. On line 67–69, memory reserved for `**c` and `*x` is freed. This must be done before the end of the current `else` statement, since memory for these variables was reserved in the current block. Finally, in lines 73–76, memory allocated for the matrix `**a` and the vector `*b` is freed.

The above program was called with the following input file `coeffs`:

```
 1 8.308315e-17
 2 10
 3  2  3  5 -2  5    3 -4 -2  1  2    -3
 4  1  4 -2 -1  3   -2  1  3  4  1     2
 5  3 -1  2  1  3   -1  2  3  2 -1    -1
 6  9 -2 -1 -1  4    2  3 -1  1  4     8
 7 -1  2 -1  2 -1    3 -1 -3 -4  2    -3
 8 -4 -5  2  3  1    1  2  3 -1 -1    -4
 9 -1 -4 -2  3  4    1  2 -4 -3 -8     3
10 -9 -4 -3 -1  9   -2 -2 -3  4  8     2
11 -1 -2  9  8 -7   -8  2 -4  3  1     5
12  8 -7  7  0  3   -5  3 -2  4  9     7
```

Note that the first column of numbers 1–12 here are line numbers, as usual, and not part of the file. Line 1 contains the tolerance; after some experimentation, tolerance was set nearly as low as possible without causing iterative improvement to fail. Line 2 contains the value of n, and each of

the rest of the lines contain the coefficients of one of the ten equations and its right-hand side. With this input file, we obtain the following output:

```
 1 Tolerance for relative L-infinity error used: 8.308315e-17
 2 3 iterations were used to find the solution.
 3 The solution is:
 4 x[  1]= -0.270477452322
 5 x[  2]=  0.126721910153
 6 x[  3]=  0.014536223326
 7 x[  4]= -1.039356801309
 8 x[  5]= -0.542881432027
 9 x[  6]=  0.900056527510
10 x[  7]=  2.429714763221
11 x[  8]= -1.669529286724
12 x[  9]=  1.731205614034
13 x[ 10]= -0.163886551028
```

Problems

1. Find the LU-factorization of the matrix

$$A = \begin{pmatrix} 2 & 3 & 1 \\ 4 & 8 & 5 \\ 6 & 13 & 12 \end{pmatrix}.$$

Do not make any row interchanges.

Solution. Writing $A = (a_{ij})$ and $L = (l_{ij})$, we have $l_{21} = a_{21}/a_{11} = 4/2 = 2$ and we have $l_{31} = a_{31}/a_{11} = 6/2 = 3$. We obtain the matrix $A^{(2)}$ by subtracting $l_{21} = 2$ times the first row from the second row, and $l_{21} = 3$ times the first row from the third row:

$$A^{(2)} = \begin{pmatrix} 2 & 3 & 1 \\ 0 & 2 & 3 \\ 0 & 4 & 9 \end{pmatrix}.$$

Writing $A^{(2)} = a_{ij}^{(2)}$, we have $l_{32} = a_{32}^{(2)}/a_{22}^{(2)} = 4/2 = 2$. We obtain the matrix $U = A^{(3)}$ by subtracting the $l_{32} = 2$ times the second row from the third row in he matrix $A^{(2)}$:

$$U = A^{(3)} = \begin{pmatrix} 2 & 3 & 1 \\ 0 & 2 & 3 \\ 0 & 0 & 3 \end{pmatrix}.$$

The matrix L can be written by noting that the elements of L in the main diagonal are all 1's, i.e., $l_{ii} = 1$ for $i = 1, 2, 3$, and the elements in the upper triangle of L are zero, i.e., $l_{ij} = 0$ whenever $1 \leq i < j \leq 3$. The values of the remaining elements of L ($l_{21} = 2$, $l_{31} = 3$, and $l_{32} = 2$) have been given above. That is, we have

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix}.$$

2. Find the LU-factorization of the matrix

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 9 & 7 & 5 \\ 6 & 6 & 9 \end{pmatrix}.$$

Do not make any row interchanges.

Solution. Writing $A = (a_{ij})$ and $L = (l_{ij})$, we have $l_{21} = a_{21}/a_{11} = 9/3 = 3$ and we have $l_{31} = a_{31}/a_{11} = 6/3 = 2$. We obtain the matrix $A^{(2)}$ by subtracting $l_{21} = 3$ times the first row from the second row, and $l_{31} = 2$ times the first row from the third row:

$$A^{(2)} = \begin{pmatrix} 3 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 2 & 7 \end{pmatrix}.$$

Writing $A^{(2)} = a_{ij}^{(2)}$, we have $l_{32} = a_{32}^{(2)}/a_{22}^{(2)} = 2/1 = 2$. We obtain the matrix $U = A^{(3)}$ by subtracting the l_{32} times the second row from the third row in he matrix $A^{(2)}$:

$$U = A^{(3)} = \begin{pmatrix} 3 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 3 \end{pmatrix}.$$

The matrix L can be written by noting that the elements of L in the main diagonal are all 1's, i.e., $l_{ii} = 1$ for $i = 1, 2, 3$, and the elements in the upper triangle of L are zero, i.e., $l_{ij} = 0$ whenever $1 \leq i < j \leq 3$. The values of the remaining elements of L ($l_{21} = 2$, $l_{31} = 3$, and $l_{32} = 2$) have been given above. That is, we have

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 2 & 1 \end{pmatrix}.$$

3. Solve the equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 3 & 1 \end{pmatrix} \begin{pmatrix} 3 & 1 & 1 \\ 0 & 4 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 37 \\ 47 \end{pmatrix}.$$

Solution. Write the above equation as $LU\mathbf{x} = \mathbf{b}$, and write $\mathbf{y} = U\mathbf{x}$. Then this equation can be written as $L\mathbf{y} = \mathbf{b}$, that is

$$\begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 3 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 37 \\ 47 \end{pmatrix}.$$

This equation can be written as

$$\begin{aligned} y_1 &= 10 \\ 3y_1 + y_2 &= 37 \\ 2y_1 + 3y_2 + y_3 &= 47 \end{aligned}$$

The solution of these equations is straightforward (the method for solving them is called forward substitution). From the first equation, we have $y_1 = 10$. Then, from the second equation we have $y_2 = 37 - y_1 = 37 - 3 \cdot 10 = 7$. Finally, from the third equation $y_3 = 47 - 2y_1 - 3y_2 = 47 - 2 \cdot 10 - 3 \cdot 7 = 6$. That is, $\mathbf{y} = (10, 7, 6)^T$. Thus, the equation $U\mathbf{x} = \mathbf{y}$ can be written as

$$\begin{pmatrix} 3 & 1 & 1 \\ 0 & 4 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 7 \\ 6 \end{pmatrix},$$

or else

$$\begin{aligned} 3x_1 + x_2 + x_3 &= 10 \\ 4x_2 + x_3 &= 7 \\ 2x_3 &= 6 \end{aligned}$$

This can easily be solved by what is called back substitution. From the third equation, we have $x_3 = 6/2 = 3$. Substituting this into the second equation, we have

$$x_2 = \frac{7 - x_3}{4} = \frac{7 - 3}{4} = 1.$$

Finally, substituting these into the second equation, we obtain

$$x_1 = \frac{10 - x_2 - x_3}{3} = \frac{10 - 3 - 1}{3} = 2.$$

That is, we have

$$\mathbf{x} = (2, 1, 3)^T.$$

4. Solve the equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 11 \\ 27 \\ 28 \end{pmatrix}.$$

Solution. Write the above equation as $LU\mathbf{x} = \mathbf{b}$, and write $\mathbf{y} = U\mathbf{x}$. Then this equation can be written as $L\mathbf{y} = \mathbf{b}$, that is

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 3 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 11 \\ 27 \\ 28 \end{pmatrix}.$$

This equation can be written as

$$\begin{aligned} y_1 &= 11 \\ 2y_1 + y_2 &= 27 \\ y_1 + 3y_2 + y_3 &= 28 \end{aligned}$$

The solution of these equations is straightforward (the method for solving them is called forward substitution). From the first equation, we have $y_1 = 11$. Then, from the second equation we have $y_2 = 27 - 2y_1 = 27 - 2 \cdot 11 = 5$. Finally, from the third equation $y_3 = 28 - y_1 - 3y_2 = 28 - \cdot 11 - 3 \cdot 5 = 2$. That is, $\mathbf{y} = (11, 5, 2)^T$. Thus, the equation $U\mathbf{x} = \mathbf{y}$ can be written as

$$\begin{pmatrix} 2 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 11 \\ 5 \\ 2 \end{pmatrix},$$

or else

$$\begin{aligned} 2x_1 + x_2 + 2x_3 &= 11 \\ x_2 + 2x_3 &= 5 \\ x_3 &= 2 \end{aligned}$$

This can easily be solved by what is called back substitution. From the third equation, we have $x_3 = 2$. Substituting this into the second equation, we have

$$x_2 = 5 - 2x_3 = 5 - 4 = 1.$$

Finally, substituting these into the second equation, we obtain

$$x_1 = \frac{11 - x_2 - 2x_3}{2} = \frac{11 - 1 - 4}{2} = 3.$$

That is, we have

$$\mathbf{x} = (3, 1, 2)^T.$$

5. Given

$$A = LU = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 6 & 4 \\ 0 & 4 & 8 \\ 0 & 0 & 3 \end{pmatrix},$$

write A as $L'U'$ such that L' is a lower triangular matrix and U' is an upper triangular matrix such that the elements in the main diagonal of U' are all 1's.

Solution. Write D for the diagonal matrix whose diagonal elements are the same as those of U. That is,

$$D = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 3 \end{pmatrix}.$$

We have $A = LU = (LD)(D^{-1}U)$. Note that the inverse of a diagonal matrix is a diagonal matrix formed by the reciprocal of the elements in the diagonal:

$$D^{-1} = \begin{pmatrix} 1/2 & 0 & 0 \\ 0 & 1/4 & 0 \\ 0 & 0 & 1/3 \end{pmatrix}.$$

Now, we have

$$LD = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 4 & 4 & 0 \\ 2 & 12 & 3 \end{pmatrix}.$$

Furthermore,

$$D^{-1}U = \begin{pmatrix} 1/2 & 0 & 0 \\ 0 & 1/4 & 0 \\ 0 & 0 & 1/3 \end{pmatrix} \begin{pmatrix} 2 & 6 & 4 \\ 0 & 4 & 8 \\ 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Thus,

$$A = \begin{pmatrix} 2 & 0 & 0 \\ 4 & 4 & 0 \\ 2 & 12 & 3 \end{pmatrix} \begin{pmatrix} 1 & 3 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

6. Briefly explain how iterative improvement works.

Solution. In iterative improvement, one takes the calculated solution $\mathbf{x}^{(0)}$ of the equation $A\mathbf{x} = \mathbf{b}$, and then one calculates the difference

$$\delta\mathbf{b}^{(1)} = \mathbf{b} - A\mathbf{x}^{(0)}.$$

The inner products used in the calculation of the components of this vector should be accumulated in a location with double or extended precision; only one scalar variable needs to be declared for this (e.g., one can use `double` in C if one uses `float` for the rest of the calculation, or one can use `long double` if one uses `double` for the rest of the calculation). One then solves the equation $A\delta\mathbf{x}^{(1)} = \delta\mathbf{b}^{(1)}$, and takes $\mathbf{x}(1) = \mathbf{x}^{(0)} + \delta\mathbf{x}^{(1)}$ as an improved approximation of the solution of the equation. This process can be repeated, by calculating

$$\delta\mathbf{b}^{(2)} = \mathbf{b} - A\mathbf{x}^{(1)},$$

and then solving the equation $A\delta\mathbf{x}^{(2)} = \delta\mathbf{b}^{(2)}$, and so on.

33. THE DOOLITTLE AND CROUT ALGORITHMS. EQUILIBRATION

The Doolittle algorithm. The Doolittle algorithm is a slight modification of Gaussian elimination. In Gaussian elimination, one starts with the system of equations $A\mathbf{x} = \mathbf{b}$, where $A = (a_{ij})$ is an $n \times n$ matrix, and $\mathbf{x} = (x_1, x_2, \dots, x_n)^T$ and $\mathbf{b} = (b_1, b_2, \dots, b_n)^T$ are column vectors. Then one determines a factoring LU of the matrix A, where U is an upper triangular matrix and L is a lower triangular *unit* matrix, i.e., a lower triangular matrix with all 1s in the diagonal. (Similarly, an upper triangular matrix with all 1s in the diagonal will be called an upper triangular *unit* matrix.) One determines the elements of the matrices U and L row by row, at the ith step for making $a_{kj}^{(i+1)} = 0$ for $k > i$ of the elements of the matrix $A^{(i+1)}$ (see equation (3) of the section on Gaussian elimination).

One can change the order of calculations by doing the calculations column by column. That is, instead of calculating the elements of the matrices $A^{(1)}$, $A^{(2)}$, ..., $A^{(n)}$, one can directly calculate the elements of the matrix $L = A^{(\text{new})}$ by doing the calculations in different order. Namely, first one calculates the first column of L and the first row U, column of L and the second row of U, and so on, by using the formulas derived above. In particular, one takes

$$u_{1s} = a_{1s} \quad \text{for} \quad s \geq 1,$$

and

$$l_{s1} = \frac{a_{s1}}{u_{11}} \quad \text{for} \quad s > 1.$$

For $i > 1$ one calculates

$$u_{is} = a_{is} - \sum_{k=1}^{i-1} l_{ik}u_{ks} \quad \text{for} \quad s \geq i, \tag{1}$$

and

$$l_{si} = \frac{a_{si} - \sum_{k=1}^{i-1} l_{sk}u_{ki}}{u_{ii}} \quad \text{for} \quad s > i. \tag{2}$$

If one uses the last two formulas for $i = 2, 3, \ldots$, in turn, the elements on the right-hand side have already been calculated by the time one needs them.[117]

The point of doing the calculations in this order is that one can use a location with extended precision to calculate the partial sums of the right-hand sides of the formulas for u_{is} an l_{is}, in the same way as this was done in the function `backsubst` in the implementation of iterative improvement in the file `lusolve.c` of the section of Gaussian elimination.

One gets into trouble with this calculation when one wants to implement partial pivoting, since when calculating u_{ss}, one may have to interchange the equations first, to see which row will give the largest absolute value of u_{ss}. The way to get around this is to tentatively interchange row s with each of the rows $s+1$, $s+2$, $\ldots$, and keep the interchange that gives the largest absolute value for u_{ss}. The downside of this is that u_{ss} needs to be calculated for each of the tentative interchanges, and later the results of all these calculations will be discarded except the one involving the actual interchange made. This waste is avoided in the

Crout algorithm. We will describe the Crout algorithm with partial pivoting and *implicit equilibration*. Partial pivoting does not work well if the equations are "differently scaled," that is, if the coefficients in some of the equations have sizes very different from the coefficients in some other equations. To avoid this, before doing any (variant of) Gaussian elimination, one may want to rescale the equations by dividing each equation by its coefficient of maximum absolute value. Doing this is called *equilibration*. This, however, is very costly, since it involves n^2 divisions. Instead of doing equilibration, one can simulate its effects by doing what is called *implicit equilibration* by first calculating

$$d_i = \max\{|a_{ij}| : 1 \le j \le n\} \quad \text{for } i \text{ with} \quad 1 \le i \le n. \tag{3}$$

We will describe how to simulate equilibration in the context of the Crout algorithm. In the Crout algorithm, one factors the matrix A as $P^{-1}L'U'$, where P is a permutation matrix, L' is a lower triangular matrix, and U' is an upper triangular unit matrix, as opposed to the factorization A=$P^{-1}LU$ obtained in Gaussian elimination, where L is an lower triangular unit matrix, and U is an upper triangular matrix. Given the factorization $A = P^{-1}LU$, one can obtain the factorization $A = P^{-1}L'U'$ as was done in Problem 5 of the section on Gaussian elimination, but the point here is to obtain the latter factorization directly. The equations can be obtained by combining the equations of Gaussian elimination with the method used in Problem 5 just mentioned.

As the first step, we put

$$l'_{s1} = a_{s1} \quad \text{for} \quad s \ge 1,$$

and pick the pivot row as the row s for which

$$\left|\frac{l'_{s1}}{d_s}\right| \quad \text{for} \quad s \ge 1$$

is largest, and then interchange row 1 and s. Then change the notation appropriately to reflect this row interchange (so that elements the new first row, for example, are again denoted by a_{1s}), and write

$$u'_{1s} = \frac{a_{1s}}{l'_{11}} \quad \text{for} \quad s > 1.$$

For $i > 1$, write

$$l'_{si} = a_{si} - \sum_{k=1}^{i-1} l'_{sk}u'_{ki} \quad \text{for} \quad s \ge i. \tag{4}$$

[117] The starting equations above are just equations (1) and (2) with $i = 1$, when these equations contain an empty sum. So there was no need to state the starting equations; perhaps stating them makes the explanation clearer.

Then select the row s such that

$$\left|\frac{l'_{si}}{d_s}\right| \quad \text{for} \quad s \geq i$$

is the largest as the pivot row. Interchange row s and row i, and change the notation to reflect the row interchange. Then calculate

$$u'_{is} = \frac{a_{is} - \sum_{k=1}^{i-1} l'_{ik} u'_{ks}}{l'_{ii}} \quad \text{for} \quad s > i. \tag{5}$$

The elements whose calculation is not indicated here are not stored (these are the above diagonal elements of the matrix L' and the below diagonal elements of U', all of these being 0, and the diagonal elements of U', all of these being 1).[118]

The main point of the Crout algorithm is that finding the pivot row does not involve tentative calculations, as it would in the Doolittle algorithm. If one does not want to do implicit equilibration, then instead of (3) one takes $d_i = 1$ for all i. Equations (4) and (5) allow one first to store the partial products in a location with extended precision, and then to transfer the value to the appropriate location in the coefficient matrix (at this point, some precision is lost, but the reason to use extended precision was to control the accumulation of roundoff errors in the calculation of the matrix element in question).

So, what is the real reason that partial pivoting works smoothly in the Crout algorithm, but requires extra calculations in the Doolittle algorithm? To understand this, one needs to take another look at Problem 5 in the section on Gaussian elimination. It is seen in this problem that when one converts an LU factorization LU with L being an lower diagonal unit matrix into and a factorization $L'U'$ with U' an upper diagonal unit matrix that the diagonal elements of L' are the same as those of U. When doing Gaussian elimination, these diagonal elements are the elements that turned out to be pivot elements. At the ith step, the pivot element is chosen from among the elements of column i of the matrix $A^{(i)}$ on or below the diagonal, and only the pivot element among these occurs in the matrix U obtained at the end of Gaussian elimination, the other elements were changed in the continued course of the algorithms. On the other hand, these elements are the same as the elements of the ith column of the matrix L' on or below the diagonal; these elements are calculated at the ith step of the Crout algorithm, and are available when one needs to determine the pivot row. In the Doolittle algorithm, these elements are never available, since only the final elements of the matrix are calculated naturally, so only the element actually chosen as pivot element is calculated naturally. The other elements of the ith column at the ith step need to be calculated tentatively to find the pivot element, and then these calculations will be discarded after the pivot element is found.

34. DETERMINANTS

Cyclic permutations. A *permutation* is a one-to-one mapping of a set onto itself. A *cyclic permutation*, or a *cycle*, or a *k-cycle*, where $k \geq 2$ is an integer, is a permutation σ where for some elements i_1, i_2, ..., i_k, we have $\sigma(i_1) = i_2$, $\sigma(i_2) = i_3$, ..., $\sigma(i_{k-1}) = i_k$, $\sigma(i_k) = i_1$. A standard notation for this permutation is $\sigma = (i_1 i_2 \dots i_k)$. One often considers this σ as being a permutation of some set M that includes the set $\{i_1, i_2, \dots, i_k\}$ such that $\sigma(i) = i$ for any element of M that is not in this set. The two-cycle (ii) is the identity mapping. Such a two-cycle is not a *proper* two-cycle, a proper two-cycle being a two-cycle (ij) for i and j distinct. A proper two-cycle is also called a *transposition*.

[118] Similarly as above in case of the Doolittle algorithm, the starting equations are just equations (4) and (5) with $i = 1$.

LEMMA. *Every permutation of a finite set can be written as a product of transpositions.*

PROOF. It is enough to consider permutations of the set $M_n \stackrel{def}{=} \{1, 2, \dots, n\}$. We will use induction on n. The Lemma is certainly true for $n = 1$, in which case every permutation of M_n (the identity mapping being the only permutation) can be written as a product of zero number of transpositions (the *empty* product of mappings will be considered to be the identity mapping). Let σ be a permutation of the set M_n, and assume $\sigma(n) = i$. Then

$$(in)\sigma(n) = n$$

(since $i = \sigma(n)$ and $(in)(i) = n$). So the permutation $\rho = (in)\sigma$ restricted to the set M_{n-1} is a permutation of this latter set. By induction, it can be written as a product $\tau_1 \dots \tau_m$ of transpositions, and so

$$\sigma = (in)^{-1}\rho = (in)\rho = (in)\tau_1 \dots \tau_m,$$

where for the second equality, one needs to note that, clearly, the inverse $(in)^{-1}$ of (in) is (in) itself. □

One can use the idea of the proof of the above Lemma, or else one can use direct calculation, to show that if i, j, and k are distinct elements then

$$(ijk) = (ik)(ij).$$

For example, $(ij)(j) = i$ and $(ik)(i) = k$, and so $(ik)(ij)(j) = k$, which agrees with the equation $(ijk)(j) = k$.

Even and odd permutations. We start with the following

LEMMA. *The identity mapping cannot be written as a product of an odd number of transpositions.*

PROOF. We will assume that the underlying set of the permutations is M_n, and we will use induction on n. For $n = 1$ the statement is certainly true; namely, there are no transpositions on M_1, so the identity mapping can be represented only as a product of a zero number of transpositions. According to the last displayed equation, if i, j, and n are distinct elements of M_n, we have $(nij) = (nj)(ni)$, and we also have $(nij) = (ijn) = (in)(ij) = (ni)(ij)$, and so

$$(nj)(ni) = (ni)(ij).$$

Moreover, in a similar way, $(nij) = (jni) = (ji)(jn) = (ij)(nj)$. Comparing this with one of the expressions equal to (nij) above we obtain

$$(ij)(nj) = (ni)(ij).$$

Further, we clearly have

$$(ni)(ni) = \iota,$$

where ι (the Greek letter *iota*) stands for the identity mapping (often represented by the empty product. Finally, if i, j, k, n are distinct, then we have

$$(ij)(nk) = (nk)(ij).$$

Using these four displayed equation, all transpositions containing n in the product can be moved all the way to the left in such a way that in the end either no transposition will contain n, or at most one transposition containing n will remain all the way to the left. Each application of these identities changes the number of transpositions in the product by an even number; in fact, the third

of these identities decreases the the number of transpositions by two, and the others do no change the number of transpositions.

A product $\sigma = (ni)\tau_1 \dots \tau_m$, where i is distinct from n, and the transpositions $\tau_1, \dots \tau_m$ do not contain n cannot be the identity mapping (since $\sigma(n) = i$), so the only possibility that remains is that we end up with a product of transpositions representing the identity mapping where none of the transpositions contain n. Then we can remove n from the underlying set; since, by the induction hypothesis, the identity mapping on M_{n-1} can only be represented as a product of an even number of transpositions, we must have started with an even number of transpositions to begin with.

COROLLARY. *A permutation cannot be written as a product both of an even number of transpositions and of an odd number of transpositions.*

PROOF. Assume that for a permutation σ we have $\sigma = \tau_1 \dots \tau_k$ and $\sigma = \rho_1 \dots \rho_l$ where k is even, l is odd, and $\tau_1, \dots, \tau_k$, and $\rho_1, \dots, \rho_l$, are transpositions. The the identity mapping can be written as a product

$$\iota = \sigma\sigma^{-1} = \tau_1 \dots \tau_k(\rho_1 \dots \rho_l)^{-1} = \tau_1 \dots \tau_k(\rho_l)^{-1} \dots (\rho_1)^{-1} = \tau_1 \dots \tau_k\rho_l \dots \rho_1.$$

Since the right-hand side contains an odd number of transpositions, this is impossible according to the last Lemma. □

A permutation that can be written as a product of an even number of transpositions is called an *even permutation*, and a permutation that can be written as a product of an odd number of transpositions is called an *odd permutation*. The function sgn (*sign*, or Latin *signum*) on permutations of a finite set is defined as $\operatorname{sgn}(\sigma) = 1$ if σ is an even permutation, and $\operatorname{sgn}(\sigma) = -1$ if σ is an odd permutation, Often, it is expedient to extend the signum function to mappings of a finite set into itself that are not permutations (i.e, that are not one-to-one) by putting $\operatorname{sgn}(\sigma) = 0$ if σ is not one-to-one.

The distributive rule. Let (a_{ij}) be an $m \times n$ matrix. Then

$$\prod_{i=1}^{n}\sum_{j=1}^{m} a_{ij} = \sum_{\sigma}\prod_{i=1}^{n} a_{i\sigma(i)},$$

where σ runs over all mappings of the set $M_n = \{1, 2, \dots, n\}$ into the set $M_n = \{1, 2, \dots, m\}$. The left-hand side represent a product of sums. The right-hand side multiplies out this product by taking one term out of each of these sums, and adding up all the products that can be so formed. The equality of these two sides is obtained by the distributivity of multiplication over addition. For example, for $m = n = 2$, the above equation says that

$$(a_{11} + a_{12})(a_{21} + a_{22}) = a_{1\sigma_1(1)}a_{2\sigma_1(2)} + a_{1\sigma_2(1)}a_{2\sigma_2(2)} + a_{1\sigma_3(1)}a_{2\sigma_3(2)} + a_{1\sigma_4(1)}a_{2\sigma_4(2)},$$

where $\sigma_1(1) = 1$, $\sigma_1(2) = 1$, $\sigma_2(1) = 1$, $\sigma_2(2) = 2$, $\sigma_3(1) = 2$, $\sigma_3(2) = 1$, $\sigma_4(1) = 2$, $\sigma_4(2) = 2$. We will use this rule with $m = n$ below.

Determinants. The determinant $\det A$ of an $n \times n$ matrix $A = (a_{ij})$ is defined as

$$\det A = \sum_{\sigma} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)},$$

where σ runs over all mappings (not necessarily one-to-one) of the set $M_n = \{1, 2, \dots, n\}$ into itself. The above formula is called the *Leibniz formula* for determinants. Sometimes one writes $\det(A)$ instead of $\det A$. Since $sgn(\sigma) = 0$ unless σ is a permutation, one may say instead that σ runs through all permutations of the set of the set $M_n = \{1, 2, \dots, n\}$. However, for some considerations (e.g., for the application of the distributive rule above) it may be expedient to say that σ runs through all mappings, rather than just permutations.

Multiplications of determinants. The product of two determinants of the same size is the determinant of the product matrix; that is:

LEMMA. *Let $A = (a_{ij})$ and $B = (b_{ij})$ be two $n \times n$ matrices. Then*

$$\det(AB) = \det(A)\det(B).$$

PROOF. With σ and ρ running over all permutations of M_n, we have

$$\begin{aligned}\det(A)\det(B) &= \sum_{\sigma,\rho} \operatorname{sgn}(\sigma)\operatorname{sgn}(\rho)\Big(\prod_{i=1}^n a_{i\sigma(i)}\Big)\Big(\prod_{i=1}^n b_{i\rho(i)}\Big)\\ &= \sum_{\sigma,\rho} \operatorname{sgn}(\rho)\operatorname{sgn}(\sigma)\Big(\prod_{i=1}^n a_{i\sigma(i)}\Big)\Big(\prod_{i=1}^n b_{\sigma(i)\rho\sigma(i)}\Big) = \sum_{\sigma,\rho} \operatorname{sgn}(\rho\sigma)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\rho\sigma(i)}\end{aligned}$$

On the right-hand side of the second equality, the product $\prod_{i=1}^n b_{\sigma(i)\rho\sigma(i)}$ is just a rearrangement of the product $\prod_{i=1}^n b_{i\rho(i)}$, since σ is a one-to-one mapping. The third equality takes into account that $\operatorname{sgn}(\rho)\operatorname{sgn}(\sigma) = \operatorname{sgn}(\rho\sigma)$ (since, if ρ can be written as the product of k transpositions and σ as the product of l transpositions, $\rho\sigma$ can be written as the product of $k+l$ transpositions). Writing $\rho\sigma = \pi$, the permutations π and σ uniquely determine ρ, so the right-hand side above can be written as

$$\sum_{\sigma,\pi} \operatorname{sgn}(\pi)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(i)} = \sum_{\sigma}\sum_{\pi} \operatorname{sgn}(\pi)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(i)}.$$

Here σ and π run over all permutations of M_n. Now we want to change our point of view in that we want to allow σ to run over all mappings of M_n into itself on the right-hand side, while we still restrict π to permutations.[119] To show that this is possible, we will show that the inner sum is zero whenever σ is not a permutation. In fact, assume that for some distinct k and l in M_n we have $\sigma(k) = \sigma(l)$. Then, denoting by (kl) the transposition of k and l, the permutation $\pi(kl)$ will run over all permutations of M_n as π runs over all permutations of M_n. Hence, the first equality next is obvious:

$$\begin{aligned}\sum_{\pi} \operatorname{sgn}(\pi)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(i)} &= \sum_{\pi} \operatorname{sgn}(\pi(kl))\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(kl)(i)}\\ &= -\sum_{\pi} \operatorname{sgn}(\pi)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(kl)(i)} = -\sum_{\pi} \operatorname{sgn}(\pi)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(i)}.\end{aligned}$$

The second equality expresses the fact that $\operatorname{sgn}(\pi(kl)) = -\operatorname{sgn}(\pi)$, and the third equality expresses the fact that $\sigma(k) = \sigma(l)$, and the equation just reflects the interchange of the factors corresponding to $i = k$ and $i = l$ in the product. Rearranging this equation, it follows that

$$2\sum_{\pi} \operatorname{sgn}(\pi)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(i)} = 0,$$

and so

$$\sum_{\pi} \operatorname{sgn}(\pi)\prod_{i=1}^n a_{i\sigma(i)}b_{\sigma(i)\pi(i)} = 0,$$

[119] One might ask why did we not extend the range of σ to all mappings earlier, when this would have been easy since we had $\operatorname{sgn}(\sigma)$ in the expression, which is zero if σ is not one-to-one. The answer is that we wanted to make sure that $\pi = \rho\sigma$ is a permutation, and if σ is not one-to-one then $\pi = \rho\sigma$ is not one-to-one, either.

as we wanted to show.[120]

Therefore, in the last sum expressing $\det(A)\det(B)$ one can allow σ to run over all mappings of M_n into itself, and not just over permutations (while π will still run over all permutations). We have

$$\det(A)\det(B) = \sum_{\sigma}\sum_{\pi}\operatorname{sgn}(\pi)\prod_{i=1}^{n} a_{i\sigma(i)}b_{\sigma(i)\pi(i)} = \sum_{\pi}\operatorname{sgn}(\pi)\sum_{\sigma}\prod_{i=1}^{n} a_{i\sigma(i)}b_{\sigma(i)\pi(i)}$$
$$= \sum_{\pi}\operatorname{sgn}(\pi)\prod_{i=1}^{n}\sum_{k=1}^{n} a_{ik}b_{k\pi(i)} = \det\left(\sum_{k=1}^{n} a_{ik}b_{kj}\right)_{i,j} = \det(AB);$$

the third equality follows by the distributive rule mentioned above (with $m = n$), and the last equality holds in view of the definition of the product matrix AB. $\square$

Simple properties of determinants. For a matrix A, $\det A = \det A^T$, where A^T denotes the transpose of A. Indeed, if $A = (a_{ij})$ then, with σ running over all permutations of M_n, we have

$$\det A^T = \sum_{\sigma}\operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{\sigma(i)i} = \sum_{\sigma}\operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{i\sigma^{-1}(i)} = \sum_{\sigma}\operatorname{sgn}(\sigma^{-1})\prod_{i=1}^{n} a_{i\sigma^{-1}(i)}$$
$$= \sum_{\sigma}\operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{i\sigma(i)} = \det A;$$

here the second equality represents a change in the order in of the factors in the product, the third equality is based on the equality $\operatorname{sgn}(\sigma) = \operatorname{sgn}(\sigma^{-1})$, and the fourth equality is obtained by replacing σ^{-1} with σ, since σ^{-1} runs over all permutations of M_n, just as σ does.

If one interchanges two columns of a determinant, the value determinant gets multiplied by -1. Formally, if $k, l \in M_n$ are distinct, then $\det(a_{ij})_{i,j} = -\det(a_{i\,(kl)(j)})_{i,j}$, where, as usual, (kj) denotes a transposition. Indeed, with σ running over all permutations of M_n, we have

$$\det(a_{i\,(kl)(j)})_{i,j} = \sum_{\sigma}\operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{i\,\sigma(kl)(i)} = -\sum_{\sigma}\operatorname{sgn}(\sigma(kl))\prod_{i=1}^{n} a_{i\,\sigma(kl)(i)}$$
$$= -\sum_{\sigma}\operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{i\sigma(i)} = -\det A;$$

the second equality holds, since $\operatorname{sgn}(\sigma(kl)) = -\operatorname{sgn}(\sigma)$, and the third equality is obtained by replacing $\sigma(kl)$ by σ, since $\sigma(kl)$ runs over all permutations of M_n, just as σ does. Of course, since $\det A^T = \det A$, a similar statement can be made when one interchanges rows.

If two columns of A are identical, then $\det A = 0$. Indeed, by interchanging the identical columns, one can conclude that $\det A = -\det A$.[121]

If we multiply a row of a determinant by a number c, then the determinant gets multiplied by c. Formally: if $A = (a_{ij})$ $B = (b_{ij})$, and for some $k \in M_n$ and some number c, we have $b_{ij} = a_{ij}$ if

[120] The theory of determinants can be developed for arbitrary rings. For rings of characteristic 2 (in which one can have $a + a = 0$ while $a \neq 0$ – in fact, $a + a = 0$ holds for every a), the last step in the argument is not correct. Is is, however, easy to change the argument in a way that it will also work for rings of characteristic 2. To this end, one needs to split up the summation for π into two parts such that in the first part π runs over all even permutations; then $\pi(kl)$ will run over all odd permutations, and then one needs to show that these two parts of the sum cancel each other.

[121] This argument does not work for rings of characteristic 2. In order to establish the result for this case as well, one needs to split up the sum representing the determinant into sums containing even and odd permutations, respectively, as pointed out in the previous footnote.

$i \neq k$ and we have $b_{kj} = ca_{kj}$ then $\det B = c \det A$. This is easy to verify by factoring out c from each of the products in the defining equation of the determinant $\det B$.

If two determinants are identical except for one row, then the determinant formed by adding the elements in the different rows, while keeping the rest of the elements unchanged, the two determinants get added. Formally, if $A = (a_{ij})$ $B = (b_{ij})$, and for some $k \in M_n$ we have $b_{ij} = a_{ij}$ if $i \neq k$,and $C = (c_{ij})$, where we have $c_{ij} = a_{ij}$ if $i \neq k$,and and $c_{kj} = a_{kj} + b_{kj}$, then $\det C = \det A + \det B$. This is again easy to verify from the defining equation of determinants.

If one adds the multiple of a row to another row in a determinant, then the value of the determinant does not change. To see this, note that adding c times row l to row k to a determinant amounts to adding to the determinant c times a second determinant in which rows k and l are identical; since this second determinant is zero, nothing will change.

Expansion of a determinant by a row. Given an $n \times n$ matrix $A = (a_{ij})$, denote by $A(i, j)$ the $(n-1) \times (n-1)$ matrix obtained by deleting the ith row and jth column of the matrix.

LEMMA. *If a_{11} is the only (possibly) nonzero element of the first row of A, then*

$$\det A = a_{11} \det A(1, 1).$$

By "(possibly) nonzero" we mean that the Lemma of course applies also in the case when we even have $a_{11} = 0$.

PROOF. With σ running over all permutations of M_n and ρ running over all permutations of $\{2, \dots, n\}$, we have

$$\begin{aligned}\det A &= \sum_{\sigma} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)} = \sum_{\sigma:\sigma(1)=1} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)} \\ &= \sum_{\sigma:\sigma(1)=1} a_{11} \operatorname{sgn}(\sigma) \prod_{i=2}^{n} a_{i\sigma(i)} = a_{11} \sum_{\rho} \operatorname{sgn}(\rho) \prod_{i=2}^{n} a_{i\rho(i)} = a_{11} \det A(1, 1);\end{aligned}$$

for the second equality, note that the product on the left-hand side of the second equality is zero unless $\sigma(1) = 1$. For the fourth equality, note that if ρ is the restriction to the set $\{2, \dots, n\}$ of a permutation σ of M_n with $\sigma(1) = 1$, then $\operatorname{sgn}(\rho) = \operatorname{sgn}(\sigma)$.

COROLLARY. . *If for some $k, l \in M_n$, a_{kl} is the only (possibly) nonzero element in the kth row of A, then* $\det A = (-1)^{k+l} a_{kl} \det A(k, l)$.

PROOF. The result can be obtained from the last Lemma by moving the element a_{kl} into the top left corner (i.e., into position $(1, 1)$) of the matrix A. However, when doing this, it will not work to interchange the kth row of the matrix A with the first row, since this will change the order of rows in the submatrix corresponding to the element. In order not to disturb the order of rows in the submatrix $A(k, l)$, one always needs to interchange adjacent rows. Thus, one can move the kth row into the position of the first row by first interchanging rows k and $k-1$, then rows $k-1$ and $k-2$, then rows $k-2$ and $k-3$, etc. After bringing the element a_{kl} into the first row, one can make similar column interchanges. While doing so, one makes altogether $k-1+l-1$ row and column interchanges, hence the factor $(-1)^{k+l} = (-1)^{(k-1)+(l-1)}$ in the formula to be proved.

The following theorem describes the expansion of a determinant by a row. It is usually attributed to Pierre-Simon Laplace (1749–1827), but it was known to Gottfried Wilhelm Leibniz (1646–1716), who invented determinants of order greater than two.

THEOREM. *For any integer $k \in M_n$ we have*

$$\det A = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det A(i,j).$$

PROOF. Let the matrix B_j be the matrix that agrees with A except for row k, and in row k all elements are zero except that the element in position (k,j) is a_{kj}. In view of the last Corollary, the equation to be proved can be written as

$$\det A = \sum_{j=1}^{n} \det B_j;$$

this equation can be established by the (repeated) use of the addition rule of determinants.

The number $A_{ij} \stackrel{def}{=} (-1)^{i+j} \det A(i,j)$ is often called the *cofactor* of the entry a_{ij} of the matrix A; then the above equation can be written as

$$\det A = \sum_{j=1}^{n} a_{ij} A_{ij}.$$

Since $\det A = \det A^T$, one can obtain a similar expansion of a determinant by a column:

$$\det A = \sum_{i=1}^{n} a_{ij} A_{ij}.$$

The expansion

$$\sum_{i=1}^{n} a_{ik} A_{ij}.$$

for some j and k with $1 \le j,k \le n$ represent the expansion of a determinant by the jth column that has the elements a_{ik} in this column instead of the elements a_{ij}. If $j = k$ then this is in fact the determinant of the matrix A; if $j \neq k$ then this represent a determinant whose jth and kth columns are identical, and one of the simple properties of determinants says that such a determinant is 0. Therefore

$$\sum_{i=1}^{n} a_{ik} A_{ij} = \delta_{jk} \det A,$$

where δ_{jk} is *Kronecker's delta*, defined as $\delta_{jk} = 1$ if $j = k$ and $\delta_{jk} = 0$ if $j \neq k$. This equation can also be written as

$$\sum_{i=1}^{n} a_{ik} \frac{A_{ij}}{\det A} = \delta_{jk}.$$

This equation can also be expressed as a matrix product. In fact, with the matrix $B = (b_{ij})$ with $b_{ij} = A_{ji}/\det A$, this equation can be written simply as $AB = I$, where I is the $n \times n$ identity matrix. That is, the matrix B is the inverse of A. In other words, we have

$$A^{-1} = \left(\frac{A_{ij}}{\det A} \right)^T = \left(\frac{A_{ij}}{\det A} \right)_{j,i} = \left(\frac{A_{ji}}{\det A} \right)_{i,j}.$$

To explain the notation here, the subscripts j,i on the outside in the middle member indicates that the entry listed between the parenthesis is in the j row and the ith column. The matrix in the middle is obviously the same as the one on the right, where the subscripts outside indicate that the entry listed between the parenthesis is in the i row and the jth column.[122]

[122] If, as usual, the subscripts on the outside are omitted, some agreed-upon unspoken assumption is made. For example, one may assume that the letter that comes first in the alphabet refers to the rows.

Cramer's rule. Consider the system $A\mathbf{x} = \mathbf{b}$ of linear equations, where $A = (a_{ij})$ is an $n \times n$ matrix, and $\mathbf{x} = (x_i)^T$ and $\mathbf{b} = (b_i)^T$ are column vectors. For a fixed k with $1 \le k \le n$, multiplying the ith equation $\sum_{j=1}^n a_{ij}x_j = b_i$ by the cofactor A_{ik}, and adding these equations for i with $1 \le i \le n$ we obtain for the left-hand side

$$\sum_{i=1}^n A_{ik} \sum_{j=1}^n a_{ij}x_j = \sum_{j=1}^n x_j \sum_{i=1}^n a_{ij}A_{ik} = \sum_{j=1}^n x_j \delta_{jk} \det A = x_k \det A$$

So we obtain the equation

$$x_k \det A = \sum_{i=1}^n b_i A_{ik}.$$

Assuming that $\det A \neq 0$,[123] we have

$$x_k = \frac{\sum_{i=1}^n b_i A_{ik}}{\det A} = \frac{\det B_k}{\det A},$$

where B_k is the matrix where the kth column of A has been replaced by the right-hand side $\mathbf{b}$; the second equation holds because the numerator in the middle member represents the expansion of $\det B_k$. This determinant is called the *determinant of the unknown* x_k; the determinant $\det A$ is called the *determinant of the system*. The above equation expressing x_k is called *Cramer's rule*. Cramer's rule is of theoretical interest, but it is not a practical method for the numerical solution of a system of linear equations, since the calculations of the determinants in it are time consuming; the practical method for the solution of a system of linear equations is Gaussian elimination.

35. POSITIVE DEFINITE MATRICES

An $n \times n$ matrix A with real entries is called *positive definite* if $\mathbf{x}^T A\mathbf{x} > 0$ for any n-dimensional column vector $\mathbf{x}$. In what follows, all matrices will be assumed to have real entries. If $A = (a_{ij})$ and $\mathbf{x} = (x_1, \dots, x_n)^T$, then

$$\mathbf{x}^T A\mathbf{x} = \sum_{i,j=1}^n x_i a_{ij} x_j.$$

Hence, if A is a positive definite matrix then $a_{kk} > 0$ for all k with $1 \le k \le n$. This follows by choosing $x_i = \delta_{ki}$. Further, note that if A is positive definite then the matrix obtained by deleting the kth row and the kth column of the matrix for any k with $1 \le k \le n$ is also positive definite; this can be seen by choosing $x_k = 0$ in the equation above.

A *minor* of a matrix is the determinant of a submatrix obtained by taking the intersection of a number of rows and the same number of columns in the matrix. A *principal minor* is the determinant obtained by taking the intersection of the first k rows and first k columns of the matrix, for some integer k; this principal minor is called the kth principal minor. A matrix $A = (a_{ij})$ is symmetric is $a_{ij} = a_{ji}$.

The following is an important characterization of positive definite matrices:

[123] If $\det A = 0$, then it is easy to show that the system of equations $A\mathbf{x} = \mathbf{b}$ does not have a unique solution – i.e., either it has no solution, or it has infinitely many solutions. In fact, in this case the rank of the matrix A is less than n.

THEOREM. *All principal minors of a positive definite matrix are positive. Conversely, a symmetric matrix is positive definite if all its principal minors are positive.*

PROOF. We will use induction on n, the size of the matrix. The result is obviously true in case $n = 1$. Assume $n > 1$, and assume the result is true for every $k \times k$ matrix with $1 \leq k < n$. We will then prove that it is also true for $n \times n$ matrices. Let $A = (a_{ij})$ be an $n \times n$ matrix. At this point, we will only assume that A has one of the properties in the theorem, i.e., that it is either positive definite or that all its principal minors are positive; then we will have to show that it has the other property. Our first claim is that the element a_{11} is then nonzero. We know that this is true is A is positive definite. On the other hand, a_{11} is just the first principal minor of A.

Form the matrix B by subtracting appropriate multiples of the first row of A from the other rows in such a way that all but the first element of the first column will be zero (exactly as is done in Gaussian elimination). Writing $m_{i1} = a_{i1}/a_{11}$, this amounts to forming the matrix $B = (b_{ij}) = SA$, where $S = (s_{ij})$ is the matrix whose first column is $(1, -m_{21}, -m_{31}, \ldots, -m_{n1})^T$, and whose other elements are $s_{ij} = \delta_{ij}$. Indeed, the element b_{ij} is obtained as the product of the ith row of S and the jth column of A, i.e., for $i > 1$,

$$b_{ij} = -m_{i1}a_{1j} + \sum_{k=2}^{n} \delta_{ik}a_{kj} = a_{ij} - m_{i1}a_{1j},$$

and the right-hand side here is zero in case $j = 1$. Next, form the matrix $C = BS^T$. Note that all elements of the elements of the first column of $C = (c_{ij})$ are the same as those of B; this is because the elements of the first column of C are obtained by multiplying the rows of B by the first column of S^T; since the first column of S^T is $(1, 0, 0, \ldots, 0)^T$, in each case the result of this multiplication is the first element of the row of B in question.

Thus, $c_{11} = a_{11}$, and $c_{i1} = 0$ for $i > 1$. Further, $C = SAS^T$. S is a lower triangular matrix with all its diagonal elements equal to 1, so $\det(S) = \det(S^T) = 1$. Hence

$$\det(C) = \det(SAS^T) = \det(S)\det(A)\det(S^T) = \det(A).$$

Next, we claim that C is positive definite if and only if A is positive definite. In fact, if $\mathbf{x}$ is an n-dimensional column vector and $\mathbf{y} = S^T\mathbf{x}$ then $\mathbf{x}$ is a nonzero vector if and only if $\mathbf{y}$ is so; this is because S is nonsingular (as $\det(S) = 1$), and so $\mathbf{x} = (S^T)^{-1}\mathbf{y}$. Further,

$$\mathbf{y}^T A\mathbf{y} = \mathbf{x}^T C\mathbf{x};$$

so assuming the left-hand side is positive for all nonzero $\mathbf{y}$ is the same as assuming that the right-hand side is positive for all $\mathbf{x}$, establishing our claim.

Next assume that A is positive definite. Then all its principal minors other than $\det(A)$ are positive by the induction hypothesis; we need to prove that $\det(A)$ is also positive. As we just saw, C is also positive definite. Hence the matrix C' obtained by deleting the first row and first column of C is also positive definite. Then, by the induction hypothesis the determinant of C' is positive; further, $c_{11} > 0$, since, as we remarked above, all diagonal elements of a positive definite matrix are positive. Hence

$$\det(A) = \det(C) = c_{11}\det(C') > 0;$$

here the second equality can be seen by expanding the determinant C by its first column, and noticing that all but the first element of this column is zero.

Finally, assume that all principal minors of A are positive. For an arbitrary positive integer $m \leq n$, writing A_m,S_m, C_m, for the submatrices of A, S, C formed by the first m rows and m columns, note that the equation $C = SAS^T$ implies that $C_m = S_m A_m (S_m)^T$; this second equation would not be true for arbitrary matrices A, S, and C satisfying the first equation; it is true at

present because the block formed by the first m rows and the last $n-m$ columns of S contain all zeros, so the elements in this block do not contribute to the product on the right-hand side of the second equation.[124] Now, $\det(S_m) = \det((S_m)^T) = 1$, since S_m is a lower triangular matrix with ones in the diagonal. So,

$$\det(C_m) = \det(S_m)\det(A_m)\det((S_m)^T) = \det(A_m) > 0.$$

As for the principal minor associated with the submatrix C'_{m-1} of the matrix C' obtained by deleting the first row and first column of C (that is, C'_{m-1} consists of the first $m-1$ rows and $m-1$ columns of C', i.e., of the rows and columns of index 2 through m of C), we have

$$0 < \det(C_m) = c_{11}\det(C'_{m-1}) = \det(C_1)\det(C'_{m-1});$$

the first equation holds by expanding C_m by its first column, and the second equation holds because the only entry of the matrix C_1 is c_{11}. Since $\det(C_1) > 0$, the inequality $\det(C'_{m-1}) > 0$ follows. So all principal minors of C' are positive.

Now, assume also that A is a symmetric matrix, i.e., that $A^T = A$. It then follows that C is also symmetric, since

$$C^T = (SAS^T)^T = (S^T)^T A^T S^T = SA^T S^T = SAS^T.$$

Hence C' is also symmetric; since we just saw that all the principal minors of C' are positive, it follows by the induction hypothesis that C' is positive definite. Further, as we know that all but the first element in the first column of C is zero, it follows by the symmetry of C that all but the first element in the first row of C is also zero; that is, for $i > 1$, $c_{i1} = c_{1i} = 0$. Hence, for an arbitrary nonzero vector $\mathbf{x} = (x_1, \dots, x_n)^T$, we have

$$\sum_{i,j=1}^{n} x_i c_{ij} x_j = c_{11}x_1^2 + \sum_{i,j=2}^{n} x_i c_{ij} x_j > 0;$$

the inequality holds since $c_{11} = \det(C_1) > 0$ and the matrix C' is positive definite. Therefore, it also follows that C is positive definite. As we remarked above, this implies that A is positive definite. The proof of the theorem is complete. □

The second conclusion of the Theorem is not valid without assuming that A is symmetric, as the example of the matrix

$$A = \begin{pmatrix} 1 & -3 \\ 0 & 1 \end{pmatrix}$$

with positive principal minors shows. This matrix is not positive definite, since for the vector $\mathbf{x} = (x_1, x_2)^T = (1, 1)^T$ we have

$$\mathbf{x}^T A \mathbf{x} = x_1^2 - 3x_1x^2 + x_2^2 = -1.$$

[124] It may be easier to see this second equation directly, without considering block matrices. Indeed

$$c_{ij} = \sum_{k,l=1}^{n} s_{ik} a_{kl} s_{jl}.$$

Assuming $i \le m$ and $j \le m$, if $k > m$, we have $s_{ik} = 0$, since $s_{ik} \neq 0$ only if $k = 1$ or $i = k$, neither of which is the case; so the terms for $k > m$ make no contribution. Similarly, if $l > m$, we have $s_{jl} = 0$, so, again, the terms for $l > m$ do not make any contribution.

If one wants to argue with block matrices, all one needs to recall that matrices can be multiplied in blocks. That is

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

for matrices $A, \dots, H$ of the appropriate size (so that they can be multiplied).

Cholesky factorization. The Theorem has an important consequence concerning Gaussian elimination.

COROLLARY. *Assume A is a positive definite matrix. Then Gaussian elimination on A can be performed without pivoting (i.e., without row interchanges). Further, in the LU factorization $A = LU$, where L is a lower triangular matrix with all 1s in the diagonal, and U is an upper diagonal matrix, the diagonal elements of U are all positive.*

PROOF. We start out with Gaussian elimination without pivoting, and at the kth step (with $1 \le k \le n$, A being an $n \times n$ matrix); we will show that we can continue the procedure in that in the matrix $A^{(k)} = (a_{ij}^{(k)})$ obtained at this point we have $a_{kk}^{(k)} \neq 0$; note that this is true in case $k = 1$, since $a_{11}^{(k)} = a_{11} > 0$, as A is positive definite. The operations in Gaussian elimination (subtracting a multiple of a row from another row) do not change the values of the principal minors (since either the subtracted row is in the principal minor in question, or both the subtracted row and the row from which we subtract are outside this minor, and in this latter case the principal minor is not changed). All principal minors of A being positive according to the Theorem above, the same holds for the principal minors of $A^{(k)}$. In particular, we have $\det(A_k^{(k)}) > 0$, where $A_k^{(k)}$ is the matrix found in the intersection of the first k rows and the first k columns of $A^{(k)}$. This matrix is, however, upper triangular, so its determinant is the product of its diagonal elements. Hence, none of these diagonal elements can be zero; so $a_{kk}^{(k)} \neq 0$. Hence Gaussian elimination can be continued as claimed.

To see that $u_{ll} > 0$ for each l with $1 \le l \le n$ in the LU factorization LU obtained, note that $U = A^{(n)}$, so we have $\det(U_l) > 0$ for the lth principal minor of U, by what we said about the principal minors of $A^{(k)}$ above. Since U is a triangular matrix, we have

$$\det(U_l) = u_{ll} \det(U_{l-1});$$

as $\det(U_l) > 0$ and $\det(U_{l-1}) > 0$, the inequality $u_{ll} > 0$ follows.[125] □

Observe that if it is possible to obtain an LU decomposition of a matrix $A = LU$ (without pivoting, as this equation holds without a permutation matrix only if no pivoting was done), then this decomposition is unique, because the steps in the Gaussian elimination are uniquely determined.[126] Form the diagonal matrix[127] D by taking the diagonal elements of U. Then $D^{-1}U$ is an upper diagonal matrix whose diagonal elements are ones, and LD is a lower diagonal matrix, and we have $A = (LD)(D^{-1}U)$.[128]

Assume now that A is a symmetric matrix; then

$$A = A^T = (D^{-1}U)^T(LD)^T = (U^T(D^{-1})^T)(D^TL^T) = (U^TD^{-1})(DL^T);$$

the fourth equation here holds, since D and D^{-1} are diagonal matrices, so they are their own transposes. This is an LU decomposition of the matrix A with the diagonal elements of U^TD^{-1} being ones, this decomposition is identical to LU; hence $U = DL^T$. That is,

$$A = LDL^T.$$

[125] This argument works even in case $l = 1$ if we take U_0 to be the empty matrix (i.e., a 0×0 matrix) and we stipulate that $\det(U_0) = 1$. Of course, we also know that $u_{11} = a_{11} > 0$, so this argument is not really needed in case $l = 1$.

[126] Another way of putting this is that the equation $A = LU$ and the facts that L is a lower diagonal matrix, U is an upper diagonal matrix, and the diagonal elements of L are ones, can be used to derive the equations used in the Doolittle algorithm (without pivoting) to determine the entries of L and U.

[127] A matrix all whose elements outside the diagonal are zero.

[128] In fact, this is the decomposition obtained by the Crout algorithm.

Now assume that A is positive definite (in addition to being symmetric). Then all diagonal entries of D are positive, according to the Corollary above. Let E be the diagonal matrix formed by taking the square roots of the diagonal elements of D. Then $D = E^2$. Then writing $L'' = LE$, we have

$$A = L''(L'')^T,$$

where L'' is a lower triangular matrix. This representation of a positive definite symmetric matrix A is called its *Cholesky factorization*. For the derivation of the formulas below, it may also be worth noting that we have $L'' = LE = U^T(D^{-1})E = U^T E^{-1}$. The advantage of Cholesky factorization over Gaussian elimination is that instead of n^2 matrix elements, only about $n^2/2$ matrix elements need to be calculated. If one is only interested in solving the equation $A\mathbf{x} = \mathbf{b}$, then it is better to use the factorization $A = LDL^T$ mentioned above than the Cholesky factorization, since for this factorization one does not need to calculate the square roots of the entries of the diagonal matrix D. The formulas used in the Doolittle algorithm can easily be modified to obtain the factorization $A = LDL^T$ or the Cholesky factorization $A = L''(L'')^T$.

For the former, with $A = a_{ij}$, $L = (l_{ij})$ and $D = (d_{ij})$, where $d_{ij} = 0$ unless $i = j$, $U = (u_{ij})$ we have

$$u_{1s} = a_{1s} \quad \text{for} \quad s \geq 1,$$

and

$$l_{s1} = \frac{u_{1s}}{u_{11}} \quad \text{for} \quad s > 1.$$

For $i > 1$ one calculates

$$u_{is} = a_{is} - \sum_{k=1}^{i-1} l_{ik}u_{ks} \quad \text{for} \quad s \geq i$$

and

$$l_{si} = \frac{u_{is}}{u_{ii}} \quad \text{for} \quad s > i,$$

If one uses the last two formulas for $i = 2, 3, \ldots$, in turn, the elements on the right-hand side have already been calculated by the time one needs them.[129] Finally, $d_{ii} = u_{ii}$ (so $d_{ij} = u_{ii}\delta_{ij}$).

For the Cholesky factorization with $L'' = (l''_{ij})$ we have

$$l''_{11} = \sqrt{a_{11}},$$

and

$$l''_{s1} = \frac{a_{s1}}{l''_{11}} \quad \text{for} \quad s > 1.$$

For $i > 1$ we put

$$l''_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} (l''_{ik})^2},$$

and

$$l''_{si} = \frac{a_{si} - \sum_{k=1}^{i-1} l''_{ik}l''_{sk}}{l''_{ii}} \quad \text{for} \quad s > i$$

[129] The starting equations above are just equations (1) and (2) with $i = 1$, when these equations contain an empty sum. So there was no need to state the starting equations; perhaps stating them makes the explanation clearer. These equations can also be used in a different order, and the elements of L can also be calculated row by row. That is, for $s = 1$, one calculates u_{11} by the above formulas, and then, for $s > 1$, one can calculate u_{is} for $i \leq s$ and l_{is} for $i < s$. This order of calculations is possible because one does not intend to do any pivoting. Instead of looking at the equations of the Doolittle algorithm, the above equations can also be directly obtained from the matrix equation $A = LU = LDL^T$.

Again, the latter two formulas can also be used for $i = 1$ as well, so the two starting equations are not needed. Here we wrote a_{s1} and a_{si} instead of a_{1s} and a_{is}; this is possible, because the matrix A is symmetric.[130] This emphasizes the point that the upper half of A (i.e., a_{is} for $s > i$) never needs to be stored. This is also true for the LDL^T factorization, but there one needs also to store the matrix U, so the whole square block normally reserved for the elements of a nonsymmetric matrix is needed. The elements of L and U can be stored in the locations originally occupied by the elements of A, just as in other variants of the Gaussian elimination; this is also true for the elements of L'' in Cholesky factorization, where only the lower triangular block representing the symmetric matrix A is needed, first to store A, and then to store L''.

36. JACOBI AND GAUSS-SEIDEL ITERATIONS

In certain cases, one can solve the system of equations

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \qquad (1 \le i \le n)$$

by iteration. In Jacobi iteration, the kth approximation $x_j^{(k)}$ to the solution x_j of this equation is calculated as

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1,\ j\neq i}^{n} a_{ij}x_j^{(k)}}{a_{ii}} \qquad (1 \le i \le n),$$

and in Gauss-Seidel iteration one takes

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}}{a_{ii}} \qquad (1 \le i \le n);$$

that is, in Gauss-Seidel iteration, one makes use of $x_j^{(k+1)}$ as soon as it is available. One can use any starting values with these iterations; for example, $x_i^{(0)} = 0$ for each i is a reasonable choice.

Call an $n \times n$ matrix $A = (a_{ij})$ *row-diagonally dominant* if

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|$$

for each i with $1 \le i \le n$.

LEMMA. *Consider the equation $A\mathbf{x} = \mathbf{b}$, where A is an $n \times n$ row-diagonally dominant matrix, and $\mathbf{b}$ is an n-dimensional column vector. When solving this equation for the n-dimensional column vector $\mathbf{x}$, both Jacobi and Gauss-Seidel iterations will converge with any starting vector $\mathbf{x}^{(0)}$.*

PROOF. Write

$$q = \max_{1\le i\le n} \frac{\sum_{j=1,\ j\neq i}^{n} |a_{ij}|}{|a_{ii}|}.$$

According to the assumption that A is row-diagonally dominant, we have $0 \le q < 1$. Assume that for some $k \ge 1$ and for some C we have

$$|x_i^{(k)} - x_i^{(k-1)}| \le C \qquad \text{for each } i \text{ with } 1 \le i \le n. \tag{1}$$

[130] Again, as no pivoting is intended, these equations can be used in a different order, as outlined in the preceding footnote.

In case of Jacobi iteration, we have

$$x_i^{(k+1)} - x_i^{(k)} = \frac{\sum_{j=1,\, j\neq i}^{n} a_{ij}\big(x_j^{(k-1)} - x_j^{(k)}\big)}{a_{ii}} \qquad (1 \le i \le n),$$

and so

$$|x_i^{(k+1)} - x_i^{(k)}| \le \frac{\sum_{j=1,\, j\neq i}^{n} |a_{ij}|C}{|a_{ii}|} \le qC \qquad (1 \le i \le n).$$

Thus, if $|x_i^{(1)} - x_i^{(0)}| \le C_0$, then we have $|x_i^{(k+1)} - x_i^{(k)}| \le q^k C_0$. Since the series $\sum_{k=0}^{\infty} q^k C_0$ is convergent, given m and n with $1 \le m \le n$, it follows that for all i with $1 \le i \le n$ we have

$$|x_i^{(n)} - x_i^{(m)}| = \left|\sum_{k=m}^{n-1} (x_i^{(k+1)} - x_i^{(k)})\right| \le \sum_{k=m}^{n-1} |x_i^{(k+1)} - x_i^{(k)}| \le \sum_{k=m}^{n-1} C_0 q^k \le \sum_{k=m}^{\infty} C_0 q^k = \frac{C_0 q^m}{1-q}.$$

Since the limit of the right-hand side is 0 when $m \to \infty$, it follows by the Cauchy convergence criterion that the sequence $\{x_i^{(k)}\}_{k=1}^{\infty}$ is convergent. This proves the convergence of Jacobi iteration.

The proof of Gauss-Seidel iteration is slightly more subtle. For Gauss-Seidel iteration, we have

$$x_i^{(k+1)} - x_i^{(k)} = \frac{\sum_{j=1}^{i-1} a_{ij}\big(x_j^{(k)} - x_j^{(k+1)}\big) - \sum_{j=i+1}^{n} a_{ij}\big(x_j^{(k-1)} - x_j^{(k)}\big)}{a_{ii}} \qquad (1 \le i \le n).$$

Consider a fixed i with $1 \le i \le n$. Assume in addition to (1) that we have

$$|x_j^{(k+1)} - x_j^{(k)}| \le C \qquad \text{for each } j \text{ with } 1 \le j < i; \tag{2}$$

note that this is vacuous for $i = 1$. Then

$$|x_i^{(k+1)} - x_i^{(k)}| \le \frac{\sum_{j=1}^{i-1} |a_{ij}|C + \sum_{j=i+1}^{n} |a_{ij}|C}{|a_{ii}|} \le qC \qquad (1 \le i \le n).$$

As $q < 1$, this establishes (2) with $i + 1$ replacing i (in that it shows the inequality in (2) with i replacing j); thus, (2) follows by inductions. Hence the latter inequality also follows, assuming (1) only (since (2) has been proved, it is no longer necessary to assume it). Therefore, similarly as in the case of the Jacobi iteration, given m and n with $1 \le m \le n$, for all i with $1 \le i \le n$ we have

$$|x_i^{(n)} - x_i^{(m)}| \le \frac{C_0 q^m}{1-q}.$$

Since the limit of the right-hand side is 0 when $m \to \infty$, it again follows by the Cauchy convergence criterion that the sequence $\{x_i^{(k)}\}_{k=1}^{\infty}$ is convergent. This proves the convergence of the Gauss-Seidel iteration.

Jacobi and Gauss-Seidel iterations exhibit linear convergence, and Aitken's acceleration (see Section 16) can be adapted to matrices and used to accelerate the convergence of these methods.

Problems

1. *Question* 1. Explain why the system of equations

$$\begin{aligned} 6x + 2y - 3z &= -8 \\ 3x - 7y + 2z &= 9 \\ -2x - 4y + 9z &= 5 \end{aligned}$$

can be solved by Gauss-Seidel iteration.

Question 2. Write the equations describing the Gauss-Seidel iteration to solve the above system of equations. Do not solve the equations.

Solution. *Question* 1. The system of equation is row-diagonally dominant. That is, the absolute value of the each coefficient in the main diagonal is larger than the sum of the absolute values of all the other coefficients on the left-hand side of the same equation. Such a system of equations can always be solved by both Jacobi iteration and Gauss-Seidel iteration.

Question 2. In Gauss-Seidel iteration, one can start with $x^{(0)} = y^{(0)} = z^{(0)} = 0$, and for each integer $k \geq 0$ one can take

$$\begin{aligned} x^{(k+1)} &= \frac{-8 - 2y^{(k)} + 3z^{(k)}}{6} \\ y^{(k+1)} &= \frac{9 - 3x^{(k+1)} - 2z^{(k)}}{-7} \\ z^{(k+1)} &= \frac{5 + 2x^{(k+1)} + 4y^{(k+1)}}{9} \end{aligned}$$

2. *Question* 1. Explain why the system of equations

$$\begin{aligned} 8x - 3y - 2z &= -9 \\ 2x + 9y - 3z &= \quad 6 \\ -3x + 2y + 7z &= \quad 3 \end{aligned}$$

can be solved by Gauss-Seidel iteration.

Question 2. Write the equations describing the Gauss-Seidel iteration to solve the above system of equations.

Question 3. Write the equations describing Jacobi iteration to solve the above system of equations.

Solution. *Question* 1. The system of equation is row-diagonally dominant, similarly as in the preceding problem.

Question 2. In Gauss-Seidel iteration, one can start with $x^{(0)} = y^{(0)} = z^{(0)} = 0$, and for each integer $k \geq 0$ one can take

$$\begin{aligned} x^{(k+1)} &= \frac{-9 + 3y^{(k)} + 2z^{(k)}}{8} \\ y^{(k+1)} &= \frac{6 - 2x^{(k+1)} + 3z^{(k)}}{9} \\ z^{(k+1)} &= \frac{3 + 3x^{(k+1)} - 2y^{(k+1)}}{7}. \end{aligned}$$

Question 3. In Jacobi iteration, one can start with $x^{(0)} = y^{(0)} = z^{(0)} = 0$, and for each integer $k \geq 0$ one can take

$$\begin{aligned} x^{(k+1)} &= \frac{-9 + 3y^{(k)} + 2z^{(k)}}{8} \\ y^{(k+1)} &= \frac{6 - 2x^{(k)} + 3z^{(k)}}{9} \\ z^{(k+1)} &= \frac{3 + 3x^{(k)} - 2y^{(k)}}{7}. \end{aligned}$$

3. Explain how to solve the system of equations

$$\begin{aligned} x - 2y - 7z &= \quad 7 \\ 8x - \ y + 3z &= -2 \\ 2x + 6y + \ z &= -4 \end{aligned}$$

by Gauss-Seidel iteration.

Solution. By changing the order of equations, the system of equation can be made to be row-diagonally dominant. Moving the first equation to the last place (when the second equation will become the first one) and the third one will become the second one, we obtain the following system of equations:

$$\begin{aligned} 8x - \ y + 3z &= -2 \\ 2x + 6y + \ z &= -4 \\ x - 2y - 7z &= \quad 7 \end{aligned}$$

This system of equation is row-diagonally dominant. That is, the absolute value of the each coefficient in the main diagonal is larger than the sum of the absolute values of all the other coefficients on the left-hand side in the same row. Such a system of equations can always be solved by both Jacobi iteration and Gauss-Seidel iteration.

When solving these equations with Gauss-Seidel iteration, one can start with $x^{(0)} = y^{(0)} = z^{(0)} = 0$, and for each integer $k \geq 0$ one can take

$$\begin{aligned} x^{(k+1)} &= \frac{-2 + y^{(k)} - 3z^{(k)}}{8} \\ y^{(k+1)} &= \frac{-4 - 2x^{(k+1)} - z^{(k)}}{6} \\ z^{(k+1)} &= \frac{7 - x^{(k+1)} + 2y^{(k+1)}}{-7} \end{aligned}$$

When solving them by Jacobi iteration, one can start with $x^{(0)} = y^{(0)} = z^{(0)} = 0$, and for each integer $k \geq 0$ one can take

$$\begin{aligned} x^{(k+1)} &= \frac{-2 + y^{(k)} - 3z^{(k)}}{8} \\ y^{(k+1)} &= \frac{-4 - 2x^{(k)} - z^{(k)}}{6} \\ z^{(k+1)} &= \frac{7 - x^{(k)} + 2y^{(k)}}{-7} \end{aligned}$$

37. CUBIC SPLINES

Let $[a, b]$ be an interval, and assume we are given points x_0, x_1, ..., x_n with $a = x_0 < x_1 < \ldots < x_n = b$ for some positive integer n. and corresponding values y_0, y_1, ..., y_n, we would like to determine cubic polynomials

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i \tag{1}$$

for i with $0 \le i < n$ satisfying the following conditions:

$$S_i(x_i) = y_i \quad \text{for } i \text{ with } 0 \le i \le n-1, \tag{2}$$

$$S_{n-1}(x_n) = y_n, \tag{3}$$

$$S_i^{(k)}(x_{i+1}) = S_{i+1}^{(k)}(x_{i+1}) \qquad \text{for } i \text{ with } 0 \le i \le n-2 \text{ and } 0 \le k \le 2. \tag{4}$$

These represent $(n+1)+3(n-1) = 4n-2$ equations. The two remaining equations can be obtained by setting conditions on S_0 at x_0 and S_{n-1} at x_n. For example, if one wants to approximate a function f with $f(x_i) = y_i$, then one can require that

$$S_0'(x_0) = f'(x_0) \quad \text{and} \quad S_{n-1}'(x_n) = f'(x_n); \tag{5}$$

these are called *correct boundary conditions*. The drawback of these conditions is that $f'(x_0)$ and $f'(x_n)$ may not be known. Alternatively, one may require that

$$S_0''(x_0) = 0 \quad \text{and} \quad S_{n-1}''(x_n) = 0; \tag{6}$$

these are called *free* or *natural boundary conditions*. The function $S(x)$ defined as

$$S(x) = S_i(x) \quad \text{whenever} \quad x \in [x_i, i+1] \qquad (1 \le i < n)$$

is called a *cubic spline*. The name spline is that of a flexible ruler forced to match a certain shape by clamping it at certain points, and used to draw curves.

It is not too hard to solve the above equations. Write

$$h_i = x_{i+1} - x_i \qquad (0 \le i \le n-1).$$

In view of equation (1), equations (2) say that $y_i = d_i$ for i with $0 \le i \le n-1$ Write $d_n = y_n$; so far, d_n has not been defined. From now on, we will use d_i instead of y_i. Equations (3) and (4) can be rewritten as

$$d_{i+1} = a_i h_i^3 + b_i h_i^2 + c_i h_i + d_i \qquad (0 \le i \le n-1), \tag{7}$$

$$c_{i+1} = 3a_i h_i^2 + 2b_i h_i + c_i \qquad (0 \le i \le n-2), \tag{8}$$

$$b_{i+1} = 3a_i h_i + b_i \qquad (0 \le i \le n-2). \tag{9}$$

The equation for $i = n-1$ in (7) corresponds to equation (3); the other equations in (7)–(9) correspond the equations (4) with $k = 0$, 1, and 2.

By (9) we have

$$a_i = \frac{b_{i+1} - b_i}{3h_i} \qquad (0 \le i \le n-2); \tag{10}$$

This can be used to determine the a_i's once the b_i's are known. Substituting this into (8), we obtain

$$c_{i+1} = (b_{i+1} + b_i)h_i + c_i \qquad (0 \le i \le n-2); \tag{11}$$

According to (7),

$$\frac{d_{i+1} - d_i}{h_i} = a_i h_i^2 + b_i h_i + c_i \qquad (0 \le i \le n-1). \tag{12}$$

In case $i \neq 0$, we can substitute (11) with $i-1$ replacing i on the right-hand side to obtain

$$\frac{d_{i+1}-d_i}{h_i} = a_i h_i^2 + b_i h_i + (b_i + b_{i-1})h_{i-1} + c_{i-1}, \qquad (1 \le i \le n-1).$$

Equation (12) with $i-1$ replacing with i says

$$\frac{d_i - d_{i-1}}{h_{i-1}} = a_{i-1}h_{i-1}^2 + b_{i-1}h_{i-1} + c_{i-1}. \qquad (1 \le i \le n)$$

Subtracting the last two equations, we obtain

$$\frac{d_{i+1}-d_i}{h_i} - \frac{d_i - d_{i-1}}{h_{i-1}} = a_i h_i^2 - a_{i-1}h_{i-1}^2 + b_i(h_i + h_{i-1}) \qquad (1 \le i \le n-1).$$

Substituting into this the value for a_i and a_{i-1} given by (10) and multiplying both sides by 3, we have

$$3\frac{d_{i+1}-d_i}{h_i} - 3\frac{d_i - d_{i-1}}{h_{i-1}} = (b_{i+1}-b_i)h_i - (b_i - b_{i-1})h_{i-1} + 3b_i(h_i + h_{i-1}) \qquad (1 \le i \le n-2);$$

we cannot allow $i = n-1$ here since (10) is not valid for $i = n-1$; but, as we will see soon, (10) will be extended to $i = n-1$, and so this equation will also be extended to $i = n-1$. This can also be written as

$$b_{i-1}h_{i-1} + 2b_i(h_i + h_{i-1}) + b_{i+1}h_i = 3\frac{d_{i+1}-d_i}{h_i} - 3\frac{d_i - d_{i-1}}{h_{i-1}} \qquad (1 \le i \le n-2). \tag{13}$$

This gives $n-2$ equations for b_i, $0 \le i \le n-1$. Two more equations can be obtained from the boundary conditions (5) or (6). For example, the free boundary conditions given by equation (6) gives

$$b_0 = 0$$

and

$$3a_{n-1}h_{n-1} + b_{n-1} = 0.$$

Introducing the new variable b_n and setting $b_n = 0$, this amounts to extending (9) to $i = n-1$. Hence we can also extend (10) to $i = n-1$:

$$a_{n-1} = \frac{b_n - b_{n-1}}{3h_n} \quad \text{and} \quad b_n = 0.$$

By extending equation (10) to $i = n-1$, equation (13) will also be extended to $i = n-1$. Thus equation (13) for i with $1 \le i \le n-1$ plus the equations $b_0 = b_n = 0$ allow us to determine the values of b_i; Once b_i are known, the other coefficients can be determined with the aid of equations (10) and (12). Note that the matrix of the system (13) of equations is row-diagonally dominant, and so Gauss-Seidel iteration can be used to solve these equations.

A program implementing cubic spline interpolation (using free boundary conditions) is described next. The header file `spline.h` is given as

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4
5 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
```

```
 6
 7 struct rows{
 8    double a;
 9    double b;
10    double c;
11    double d;
12 };
13
14 struct rows *allocrows(int n);
15 double *allocvector(int n);
16 int gauss_seidel_tridiag(struct rows *eqn, double *x,
17        int n, float tol, int maxits, int *converged);
18 int free_spline(double *x, double *f, struct rows *s, int n,
19        float tol, int maxits, int *success);
20 double evalspline(double xbar, double *x, struct rows *s,
21        int n);
```

Line 5 defines the absolute value function $|x|$. In lines 7–12, the structure rows is defined to hold the coefficients of the cubic polynomials constituting the splines on the one-hand, and it will hold the coefficients and the right-hand side of the coefficient matrix in equations (1)3 on the other hand. The coefficient matrix is a *tridiagonal matrix*; that is, it has nonzero elements only in the main diagonal and the two adjacent diagonals. Lines 14–21 give the declarations of the functions occurring in the program. The memory allocation is handled by the functions in the file `alloc.c`:

```
 1 #include "spline.h"
 2
 3 struct rows *allocrows(int n)
 4 {
 5   struct rows *b;
 6   b=(struct rows *) malloc((size_t)((n+1)*sizeof(struct rows)));
 7   if (!b) {
 8     printf("Allocation failure 1 in rows\n");
 9     exit(1);
10   }
11   return b;
12 }
13
14 double *allocvector(int n)
15 {
16   double *b;
17   b=(double *) malloc((size_t)((n+1)*sizeof(double)));
18   if (!b) {
19     printf("Allocation failure 1 in vector\n");
20     exit(1);
21   }
22   return b;
23 }
```

The function `allocrows` in lines 3–12 allocates memory for a structure of type `rows` defined in the file `spline.h`. The parameter `n` indicates that a block of $n+1$ structures (indexed from 0 to n) will be reserved. In line 7 it is tested if the allocation was successful. In line 11, a pointer to the first of these structures will be returned. In lines 14–23, the function `allocvector` is defined. This reserves memory for a block of $n+1$ reals of the type `double`. This function is identical to the one used in

Gaussian elimination. The main functions are defined in the file `spline.c`:

```
#include "spline.h"

int gauss_seidel_tridiag(struct rows *eqn, double *x,
       int n, float tol, int maxits, int *converged)
{
  int itcount, i;
  double newx;
  for (i=0;i<=n+1;i++) x[i]=0.;
  *converged=0;
  for (itcount=0; itcount<maxits && !(*converged); itcount++) {
    *converged=1;
    for (i=1;i<=n;i++) {
      newx = (eqn[i].d - eqn[i].a * x[i-1]
              - eqn[i].c * x[i+1])/ eqn[i].b;
      if ( absval(newx-x[i])>tol ) *converged=0;
      x[i]=newx;
    }
  }
  return itcount;
}

int free_spline(double *x, double *f, struct rows *s, int n,
       float tol, int maxits, int *success)
{
  int i, noofits;
  struct rows *eqn;
  double *bvec, hi, hiprev, deltaf, prevdeltaf;
  eqn=allocrows(n-1);
  bvec=allocvector(n);
  hiprev=x[1]-x[0]; prevdeltaf=f[1]-f[0];
  for (i=1;i<n;i++) {
    hi=x[i+1]-x[i]; deltaf=f[i+1]-f[i];
    eqn[i].a=hiprev; eqn[i].b=2.*(hiprev+hi); eqn[i].c=hi;
    eqn[i].d=3.*(deltaf/hi-prevdeltaf/hiprev);
    hiprev=hi; prevdeltaf=deltaf;
  }
  noofits = gauss_seidel_tridiag(eqn, bvec, n-1, tol,
                        maxits, success);
  if ( *success ) {
    for (i=0;i<n;i++) {
      hi=x[i+1]-x[i];
      s[i].a=(bvec[i+1]-bvec[i])/(3.*hi);
      s[i].b=bvec[i];
      s[i].c=(f[i+1]-f[i])/hi-(bvec[i+1]+2.*bvec[i])*hi/3.;
      s[i].d=f[i];
    }
  }
  free(bvec);
  free(eqn);
  return noofits;
```

```
51 }
52
53 double evalspline(double xbar, double *x, struct rows *s,
54        int n)
55 {
56   int i;
57   double dx;
58   for (i=1; i<n && x[i]<xbar; i++) ;
59   dx=xbar-x[--i];
60   return s[i].d + dx * (s[i].c + dx * (s[i].b + dx * s[i].a));
61 }
```

In linew 3–17, Gauss-Seidel iteration for a tridiagonal matrix is implemented. The coefficients $a_{i-1\,i}$, a_{ii}, and $a_{i\,i+1}$ are represented by the structure elements `eqn[i].a`, `eqn[i].b`, `eqn[i].c`, and the right-hand side is represented by the structure element `eqn[i].d`. In the equation, it is assumed that $x_0 = x_{n+1} = 0$. This corresponds to the situation that in equations (13) with b_i the unknowns we have $b_0 = b_n = 0$ (the n in these equation will correspond to $n-1$ in the program).

The function `gauss_seidel_tridiag` is called with parameters `*eqs` describing the coefficient matrix and the right-hand side of the equation as just described, the vector `*x` to store the solution of the equations, the integer `maxits` to store the maximum permissible number of iterations, and the integer `*converged` indicating whether the calculation was successful (1 is successful, 0 if not). The function returns the actual number of iterations performed. In line 11, the integer `*converged` is set to be 1, but in line 15 it is set to be 0 unless the solution `x[i]` converges for every value of i ($1 \le i \le n$). On line 19, the number of iterations `itcount` is returned.

The function `free_spline` in lines 22–51 calculates the coefficient matrix for the equations (1)3 in lines 30–36, it calls the function `gauss_seidel_tridiag` on line 37 to solve these equations, and in lines 40–45 it calculates the coefficients of the polynomials (1) in lines 39–46. The function has parameters `*x`, the vector where the points $x_0, \ldots, x_n$ are stored, the function values y_i at these points stored in the vector `*f`, the structure `*s` to store the coefficients of the polynomials (1), the integer `n` (the number of interpolation points minus 1), the tolerance `tol` indicating the precision required for the solution, the integer `maxits` giving the maximum permissible number of iterations, and the integer `*success` indicating whether the determination of the polynomials (1) was successful. This variable gets its values when it calls `gauss_seidel_tridiag` on line 37, to indicate whether this function was successful in solving the system of equations. In lines 26, memory for the structure holding the coefficients of the equation is reserved, and in line 27 memory is reserved for the vector `bvec` to hold the solution of equations (1)3 (that is, the coefficients b_i). The memory is freed in lines 48–49. On line 50 the number of iterations `noofit` is returned. This variable obtains its value from the function call of `gauss_seidel_tridiag` in line 37, and indicates the number of iterations used in solving the system of equations.

In lines 53–61 the function `evalspline` is defined. This function has the parameters `xbar`, the value where the spline is to be evaluated, the `double *x` of interpolation points, the structure `*s` holding the coefficients of the polynomials (1), and the integer `n`. In line 58, the value of i is searched for which the point $\bar{x}$ (or `xbar` is located in the interval $[x_i, x_{i+1})$, so that the value of the spline can be returned on line 60.

The main program calling these programs in contained in the file `main.c`:

```
1 #include "spline.h"
2
3 main()
4 {
5   const float tol=5e-12;
6   const int maxits=100;
```

```
 7   int i,k,n, success, itcount;
 8   double *x, *f, xbar, y, expxbar, theerror;
 9   struct rows *s;
10   n=100;
11   x=allocvector(n);
12   f=allocvector(n);
13   s=allocrows(n-1);
14   printf("Approximating sin x by free cubic spline "
15          "interpolation using %d points\n", n+1);
16   for (i=0;i<=n;i++) {
17     x[i] = (double) i/10.;
18     f[i] = sin(x[i]);
19     printf("%7.2f ",x[i]);
20     if ( i%5==0 ) printf("\n");
21   }
22   printf("\n\n");
23   itcount=free_spline(x, f, s, n, tol, maxits,
24            &success);
25   if ( success ) {
26     printf("The number of Gauss-Seidel iterations "
27            "is %u\n\n", itcount);
28     printf("    x           sin x          "
29            "   spline(x)           error\n\n");
30     for (i=-1;i<=n/10;i++) {
31       xbar=0.55+(double) i;
32       y=evalspline(xbar,x,s,n);
33       expxbar=sin(xbar);
34       theerror=expxbar-y;
35       printf("%7.4f %18.13f %18.13f %18.13f\n",
36          xbar, expxbar, y, theerror);
37     }
38   }
39   else
40     printf("The coefficients of the cubic spline "
41       "count not be calculated\n"
42       "Probably more iterations are needed.\n");
43   free(s);
44   free(f);
45   free(x);
46 }
```

The tolerance `tol` used is specified on line 5 to be $5{\cdot}10^{-12}$, the maximum number of allowed iterations will be 100. The function $\sin x$ will be interpolated at 100 equidistant points in the interval $[0, 10]$. In lines 11–13 memory is allocated for the vectors `*x` holding the interpolation points and `*f` holding the function values at the interpolation points, and on line 13, memory is allocated for the structure `*s` containing the coefficients of the spline polynomials. In lines 17 and 18 `x[i]` and `f[i]` are allocated. In line 23, the function `free_spline` is called to find the coefficients of the spline polynomials, and in lines 30–37 the spline is evaluated at 12 points. Two evaluations take place outside the interval of interpolation, the rest of the evaluations take place inside, at the middle of certain partition intervals $[x_{i-1}, x_i]$. If the evaluation of the coefficients of the spline polynomial is unsuccessful, a message to this effect is printed in lines 40–41. In lines 43–45 the reserved memory is freed. The printout of this program is as follows:

```
 1 Approximating sin x by free cubic spline interpolation using 101 points
 2    0.00
 3    0.10    0.20    0.30    0.40    0.50
 4    0.60    0.70    0.80    0.90    1.00
 5    1.10    1.20    1.30    1.40    1.50
 6    1.60    1.70    1.80    1.90    2.00
 7    2.10    2.20    2.30    2.40    2.50
 8    2.60    2.70    2.80    2.90    3.00
 9    3.10    3.20    3.30    3.40    3.50
10    3.60    3.70    3.80    3.90    4.00
11    4.10    4.20    4.30    4.40    4.50
12    4.60    4.70    4.80    4.90    5.00
13    5.10    5.20    5.30    5.40    5.50
14    5.60    5.70    5.80    5.90    6.00
15    6.10    6.20    6.30    6.40    6.50
16    6.60    6.70    6.80    6.90    7.00
17    7.10    7.20    7.30    7.40    7.50
18    7.60    7.70    7.80    7.90    8.00
19    8.10    8.20    8.30    8.40    8.50
20    8.60    8.70    8.80    8.90    9.00
21    9.10    9.20    9.30    9.40    9.50
22    9.60    9.70    9.80    9.90   10.00
23
24
25 The number of Gauss-Seidel iterations is 22
26
27     x           sin x             spline(x)           error
28
29 -0.4500   -0.4349655341112   -0.4348249101747   -0.0001406239365
30  0.5500    0.5226872289307    0.5226870924736    0.0000001364570
31  1.5500    0.9997837641894    0.9997835031776    0.0000002610118
32  2.5500    0.5576837173914    0.5576835717979    0.0000001455935
33  3.5500   -0.3971481672860   -0.3971480636032   -0.0000001036828
34  4.5500   -0.9868438585032   -0.9868436008696   -0.0000002576336
35  5.5500   -0.6692398572763   -0.6692396825590   -0.0000001747173
36  6.5500    0.2636601823728    0.2636601135395    0.0000000688333
37  7.5500    0.9541522662795    0.9541520171807    0.0000002490989
38  8.5500    0.7674011568675    0.7674009565259    0.0000002003416
39  9.5500   -0.1248950371168   -0.1248937203815   -0.0000013167352
40 10.5500   -0.9023633099588   -1.1815860484840    0.2792227385251
```

38. OVERDETERMINED SYSTEMS OF LINEAR EQUATIONS

Assume we are given a system of linear equations

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \qquad (1 \le i \le m),$$

where $m > n$; that is, we have more equations than unknown. This situation can occur when we make more measurements to determine the quantities x_i than absolutely necessary. If the measurements

were exact, all these equations would be satisfied; however, in view of measurement errors, the determination of the coefficients has errors, and for these reason not all equations can be satisfied.

In such a situation one looks for the most reasonable solution of these equations. One can write this system of equations in matrix form as

$$A\mathbf{x} = \mathbf{b},$$

where $A = (a_{ij})$ is an $m \times n$ matrix, $\mathbf{x} = (x_1, \dots x_n)^T$ is an n-dimensional column vector, and $\mathbf{b} = (b_1, \dots b_m)^T$ is an m-dimensional column vector. While this equation is unsolvable, one may look for a vector $\mathbf{x}$ for which the norm of the vector $\mathbf{b} - A\mathbf{x}$ is the smallest possible. One can take various vector norms; we will discuss the case of l^2 norm.[131] The square of the l^2 norm of the vector $\mathbf{b} - \mathbf{A}\mathbf{x}$ can be written as

$$M = \sum_{i=1}^{m} \Big(b_i - \sum_{j=1}^{n} a_{ij} x_j \Big)^2.$$

The minimum of this will occur when the partial derivatives of M with respect to each of the quantities x_k $(1 \le k \le n)$ are zero:

$$\frac{\partial M}{\partial x_k} = -2 \sum_{i=1}^{m} a_{ik} \Big(b_i - \sum_{j=1}^{n} a_{ij} x_j \Big) = 0 \qquad (1 \le k \le n).$$

In matrix form, this equation can be written, with T in superscript indicating transpose, as $-2A^T(\mathbf{b} - A\mathbf{x}) = 0$, or else as

$$A^T A\mathbf{x} = A^T \mathbf{b}.$$

These represent n equations for n unknowns, and in principle they are solvable for $\mathbf{x}$. The problem is that the matrix $A^T A$ is usually ill-conditioned. Hence, we will seek another way to solve the problem of minimizing the l^2 norm of the vector $\mathbf{b} - \mathbf{A}\mathbf{x}$.

An $n \times n$ matrix Q is called *orthogonal* if $Q^T Q = I$, where I is the identity matrix of the same size.[132] In this case, Q^T is the inverse of the matrix Q, and so we also have $QQ^T = I$. If Q_1 and Q_2 are orthogonal matrices then $Q_1 Q_2$ is also orthogonal, since $(Q_1 Q_2)^T = Q_2^T Q_1^T$, and so

$$(Q_1 Q_2)^T Q_1 Q_2 = Q_2^T Q_1^T Q_1 Q_2 = Q_2^T I Q_2 = Q_2^T Q_2 = I.$$

If Q is an orthogonal $n \times n$ matrix, then for any n-dimensional vector $\mathbf{x}$ we have $\|Q\mathbf{x}\|_2 = \|\mathbf{x}\|_2$. Indeed,

$$\|Q\mathbf{x}\|_2^2 = (Q\mathbf{x})^T (Q\mathbf{x}) = \mathbf{x}^T Q^T Q \mathbf{x} = \mathbf{x}^T \mathbf{x} = \|\mathbf{x}\|_2^2.$$

In other word, multiplication on the left by Q is an *isometry*.[133]

Given an n-dimensional column vector $\mathbf{v}$ such that $\mathbf{v}^T \mathbf{v} = 1$, the matrix

$$H = H_{\mathbf{v}} = I - 2\mathbf{v}\mathbf{v}^T$$

is symmetric and orthogonal. Indeed,

$$H^T = I^T - 2(\mathbf{v}\mathbf{v}^T)^T = I - 2(\mathbf{v}^T)^T \mathbf{v}^T = I - 2\mathbf{v}\mathbf{v}^T = H.$$

[131] The l^2 norm $\|\mathbf{v}\|_2$ of the vector $\mathbf{v} = (v_1, \dots, v_n)^T$ is the square root of $\sum_{i=1}^n v_i^2$.

[132] The reason for calling these matrices orthogonal is that for any two distinct column vectors $\mathbf{x}$ and $\mathbf{y}$ of such a matrix we have $\mathbf{x}^T \mathbf{y} = 0$; i.e., such vectors are perpendicular, or orthogonal. The word perpendicular is rarely used for vectors of dimension greater than three; the word orthogonal literally means having a right angle.

[133] An isometry is a transformation of a space that leaves the distance between two points unchanged. In the present case, the distance between points represented by position vectors (i.e., vectors with initial points at the origin and endpoints at the given points) $\mathbf{x}$ and $\mathbf{y}$ is $\|\mathbf{y} - \mathbf{x}\|_2$.

Furthermore,

$$H^TH = HH = (I - 2\mathbf{v}\mathbf{v}^T)^2 = I - 2\cdot 2\mathbf{v}\mathbf{v}^T + 4\mathbf{v}\mathbf{v}^T\mathbf{v}\mathbf{v}^T$$
$$= I - 4\mathbf{v}\mathbf{v}^T + 4\mathbf{v}(\mathbf{v}^T\mathbf{v})\mathbf{v}^T = I - 4\mathbf{v}\mathbf{v}^T + 4\mathbf{v}\mathbf{v}^T = I;$$

the parenthesis in the third term of the fifth member can be placed anywhere since multiplication of matrices (and vectors) is associative, and the fifth equation holds since $\mathbf{v}^T\mathbf{v} = 1$. The matrix H is called a *Householder transformation*. Given two vectors $\mathbf{x}$ and $\mathbf{y}$ with $\|\mathbf{x}\|_2 = \|\mathbf{y}\|_2$ and $\mathbf{x} \neq \mathbf{y}$, there is a Householder transformation that maps $\mathbf{x}$ to $\mathbf{y}$ and $\mathbf{y}$ to $\mathbf{x}$. Namely, if one takes

$$\mathbf{v} = \frac{\mathbf{x} - \mathbf{y}}{\|\mathbf{x} - \mathbf{y}\|_2},$$

then $H_{\mathbf{v}}$ maps $\mathbf{x}$ to $\mathbf{y}$. Indeed,

$$H_{\mathbf{v}}\mathbf{x} = \left(I - 2\frac{\mathbf{x}-\mathbf{y}}{\|\mathbf{x}-\mathbf{y}\|_2}\cdot\frac{(\mathbf{x}-\mathbf{y})^T}{\|\mathbf{x}-\mathbf{y}\|_2}\right)\mathbf{x} = \left(I - 2\frac{(\mathbf{x}-\mathbf{y})(\mathbf{x}^T-\mathbf{y}^T)}{(\mathbf{x}-\mathbf{y})^T(\mathbf{x}-\mathbf{y})}\right)\mathbf{x}$$
$$= \left(I - 2\frac{(\mathbf{x}-\mathbf{y})(\mathbf{x}^T-\mathbf{y}^T)}{\mathbf{x}^T\mathbf{x} - \mathbf{x}^T\mathbf{y} - \mathbf{y}^T\mathbf{x} + \mathbf{y}^T\mathbf{y}}\right)\mathbf{x} = \mathbf{x} - 2\frac{(\mathbf{x}-\mathbf{y})(\mathbf{x}^T\mathbf{x} - \mathbf{y}^T\mathbf{x})}{\mathbf{x}^T\mathbf{x} - \mathbf{x}^T\mathbf{y} - \mathbf{y}^T\mathbf{x} + \mathbf{y}^T\mathbf{y}}.$$

In the denominator on the right-hand side we have $\mathbf{x}^T\mathbf{x} = \|\mathbf{x}\|_2^2 = \|\mathbf{y}\|_2^2 = \mathbf{y}^T\mathbf{y}$ in view of our assumption, and, noting that $\mathbf{x}^T\mathbf{y}$ is a scalar, and so it equals its own transpose, we have $\mathbf{x}^T\mathbf{y} = (\mathbf{x}^T\mathbf{y})^T = \mathbf{y}^T\mathbf{x}$; hence the denominator on the right equals $2\mathbf{x}^T\mathbf{x} - 2\mathbf{y}^T\mathbf{x}$. Thus, the right-hand side equals

$$\mathbf{x} - (\mathbf{x} - \mathbf{y}) = \mathbf{y},$$

showing that $H_{\mathbf{v}}\mathbf{x} = \mathbf{y}$, as we wanted to show.

In finding the solution $\mathbf{x}$ of the equation $A\mathbf{x} = \mathbf{b}$ for which the norm $\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2$ of the error is minimal,[134] we will find an orthogonal matrix P of size $m \times m$ for which PA has form $(R^T, \mathbf{0}^T)^T$ such that R is an $n \times n$ upper triangular matrix and $\mathbf{0}$ is an $n \times (m-n)$ matrix with all its entries 0. Write $P\mathbf{b} = (\mathbf{c}^T, \mathbf{d}^T)^T$, where $\mathbf{c}$ is an n-dimensional column vector and $\mathbf{d}$ is an $m-n$-dimensional column vector. For the vector $\mathbf{r} \overset{def}{=} \mathbf{b} - A\mathbf{x}$ we then have

$$P\mathbf{r} = \begin{pmatrix}\mathbf{c}\\ \mathbf{d}\end{pmatrix} - \begin{pmatrix}R\\ \mathbf{0}\end{pmatrix}\mathbf{x} = \begin{pmatrix}\mathbf{c} - R\mathbf{x}\\ \mathbf{d}\end{pmatrix}.$$

We have $\|\mathbf{r}\|_2 = \|P\mathbf{r}\|_2$ since P is an orthogonal matrix. Hence, for the square of $\|\mathbf{r}\|_2$ we have

$$\|\mathbf{r}\|_2^2 = \left\|\begin{pmatrix}\mathbf{c} - R\mathbf{x}\\ \mathbf{d}\end{pmatrix}\right\|_2^2 = \|\mathbf{c} - R\mathbf{x}\|_2^2 + \|\mathbf{d}\|_2^2.$$

This will be minimal when $\mathbf{c} - R\mathbf{x} = 0$. Thus, in order to minimize the norm of $\mathbf{r}$, we have to solve the equation

$$R\mathbf{x} = \mathbf{c}.$$

The orthogonal matrix P will be found as the product of Householder matrices $H_{\mathbf{v}}$ of size $m \times m$. Starting with the matrix A, we will form the matrices $A_1 = H_{\mathbf{v}_1}A$, $A_2 = H_{\mathbf{v}_2}A_1, \ldots, A_n = H_{\mathbf{v}_n}A_{n-1}$, and we will write $P = H_{\mathbf{v}_n}H_{\mathbf{v}_{n-1}}\ldots H_{\mathbf{v}_1}$. Since the product of orthogonal matrices is orthogonal, P will be orthogonal. In the matrix $A_k = (a_{ij}^{(k)})_{\substack{1\le i\le m\\ 1\le j\le n}}$ the elements below the main diagonal in the first k column will be 0. It will be seen that the process will not break down provided the columns

[134] Such a solution is called the *least square solution* of the system of equations.

of the starting matrix A are linearly independent. This assumption will also guarantee that the diagonal elements of the triangular matrix R in the upper square of the matrix PA are nonzero, and so the equation $R\mathbf{x} = \mathbf{c}$ is solvable.[135]

Starting with the matrix A, we look for a Householder transformation that maps the first column $\mathbf{x} = \mathbf{a}_1$ of the matrix A to the m-dimensional vector $\mathbf{y} = (\pm||\mathbf{a}_1||_2, 0, \dots, 0)^T$. This is possible, since the two vectors have the same length. The sign $\pm$ is chosen to be the opposite of that of a_{11}, so that the first components of $\mathbf{x}$ and $\mathbf{y}$ get added in absolute value when forming the vector

$$\mathbf{v}_1 = \frac{\mathbf{x} - \mathbf{y}}{\|\mathbf{x} - \mathbf{y}\|_2},$$

for the Householder transformation $H_{\mathbf{v}_1}$. Assume that we have formed the matrix A_{k-1} in such a way that all elements in the first $k-1$ columns below the main diagonal of A_{k-1} are zero. We will choose the vector $\mathbf{v}_k$ in such a way that the first $k-1$ components of $\mathbf{v}_k$ will be zero. In this case, with I denoting the $m \times m$ identity matrix, the Householder matrix

$$H_{\mathbf{v}_k} = I - 2\mathbf{v}_k\mathbf{v}_k^T$$

will have zeros in the first $k-1$ rows and columns except for ones in the main diagonal. Thus, in the product $A_k = H_{\mathbf{v}_k}A_{k-1}$, the first $k-1$ columns of the matrix A_{k-1} will not change.

To see this, note that the verbal description just given for $H_{\mathbf{v}_k}$ means that this matrix can be written in block matrix form as

$$H_{\mathbf{v}_k} = \begin{pmatrix} I & \mathbf{0} \\ \mathbf{0} & H \end{pmatrix},$$

where the matrix is partitioned into $k-1$ and $m-k+1$ rows and $k-1$ and $m-k+1$ columns. Here I denotes an identity matrix, and the two $\mathbf{0}$'s indicate zero matrices, each of the appropriate size.[136] The matrix A_{k-1}, partitioned into $k-1$ and $m-k+1$ rows and $k-1$ and $n-k+1$ columns, can be written as

$$A_{k-1} = \begin{pmatrix} U & B \\ \mathbf{0} & C \end{pmatrix},$$

where U is an upper diagonal square matrix, and $\mathbf{0}$ is a zero matrix, each of the appropriate size.[137] This equation expresses the fact that the the elements in the first $k-1$ columns of A_{k-1} under the main diagonal are zero. Now,

$$A_k = H_{\mathbf{v}_k}A_{k-1} = \begin{pmatrix} I & \mathbf{0} \\ \mathbf{0} & H \end{pmatrix}\begin{pmatrix} U & B \\ \mathbf{0} & C \end{pmatrix} = \begin{pmatrix} IU + \mathbf{0}\mathbf{0} & IB + \mathbf{0}C \\ \mathbf{0}U + H\mathbf{0} & \mathbf{0}B + HC \end{pmatrix} = \begin{pmatrix} U & B \\ \mathbf{0} & HC \end{pmatrix};$$

here the right-hand side is partitioned into $k-1$ and $m-k+1$ rows and $k-1$ and $m-k+1$ columns, i.e., the same way as A_{k-1} was partitioned. This shows that the first $k-1$ columns of A_{k-1} and A_k are indeed identical, as claimed.

---

[135] This latter assertion is easy to see. If the kth diagonal element of the triangular matrix R is zero, then the rows of the block formed by the intersection of the first k rows and first k columns of R are linearly dependent, since the kth row of this block consists entirely of zeros. Hence its columns are also linearly dependent. Hence the first k columns of the whole matrix R are linearly dependent, since the entries in these columns outside the block just described are all zero. Thus the columns of R are linearly dependent; therefore the columns of PA are also linearly dependent, since these columns outside R contain only zero entries. On the other hand, P being an orthogonal matrix, it is invertible. Hence, assuming that the columns of A are linearly independent, the columns of the matrix PA cannot be linearly dependent, since a zero linear combination of the columns (which can be written as the equation $PA\mathbf{y} = \mathbf{0}$, with the components of the column vector $\mathbf{y}$ being the coefficients of such a linear combination) of this matrix would be mapped by P^{-1} to a zero linear combination of the columns of the matrix $A = P^{-1}(PA)$ (that is, $A\mathbf{y} = P^{-1}(PA\mathbf{y}) = P\mathbf{0} = \mathbf{0}$).

[136] Thus, I here is the $(k-1) \times (k-1)$ unit matrix, the $\mathbf{0}$ to the right is the $(k-1) \times (m-k+1)$ zero matrix, the $\mathbf{0}$ below I is the $(m-k+1) \times (k-1)$ zero matrix, and H is an $(m-k+1) \times (m-k+1)$ matrix. H is actually a Householder matrix, but this fact will not be used directly.

[137] Thus, the size of U is $(k-1) \times (k-1)$, that of $\mathbf{0}$ is $(m-k+1) \times (k-1)$, that of B is $(k-1) \times (n-k+1)$, and that of C is $(m-k+1) \times (n-k+1)$

That is, the matrix A_k will have zeros in the first $k-1$ columns below the main diagonal, since the same was true for A_{k-1}. To make sure that in A_k the kth column also has zeros below the main diagonal, choose a Householder transformation that maps the kth column

$$\mathbf{x} = \mathbf{a}_k^{(k-1)} = \left(a_{1k}^{(k-1)}, \dots, a_{mk}^{(k-1)}\right)^T$$

to

$$\mathbf{y} = \mathbf{a}_k^{(k)} = \left(a_{1k}^{(k-1)}, \dots, a_{k-1\,k}^{(k-1)}, \pm\sqrt{\sum_{i=k}^{m}(a_{ik}^{(k-1)})^2}, 0 \dots, 0\right)^T,$$

where the zeros occupy the last $m-k$ positions (i.e., all the positions corresponding to places below the main diagonal of the kth column. These two vectors having the same l^2 norms, there is a Householder transformation mapping $\mathbf{x}$ to $\mathbf{y}$. The $\pm$ sign is chosen to be the opposite of the sign of $a_{kk}^{(k-1)}$, so that, when forming the difference

$$\bar{\mathbf{v}} = \mathbf{x} - \mathbf{y} = \left(0, \dots, 0, a_{kk}^{(k-1)} \mp \sqrt{\sum_{i=k}^{m}(a_{ik}^{(k-1)})^2}, a_{k+1\,k}^{(k-1)}, \dots, a_{mk}^{(k-1)}\right)^T,$$

quantities of the same sign are added in the kth component, so as to maximize the norm of the vector $\mathbf{v}$. Then we form the Householder transformation $H_{\mathbf{v}_k}$ with the vector

$$\mathbf{v}_k = \frac{1}{\|\bar{\mathbf{v}}\|_2}\bar{\mathbf{v}}.$$

This process will break down only if the norm in the denominator here is zero, that is, if we have $a_{ik}^{(k-1)} = 0$ for $i \geq k$. This means that the first k columns of the matrix A_{k-1} are linearly dependent, since these columns have nonzero elements only in the first $k-1$ places, and k vectors of $k-1$ dimension must be linearly dependent. Thus the columns of the matrix A_{k-1} are linearly dependent; this can only happen if the columns of the starting matrix A are linearly dependent, since $A_{k-1} = P'A$ for some orthogonal matrix (see the footnote above). Thus this algorithm of triangularizing the top square of A will not fail if the columns of A are linearly independent.

The method will work even if A is a square matrix, giving a new way of triangularizing a matrix without row interchanges. This gives a new way of solving the equation $A\mathbf{x} = \mathbf{b}$ even for square matrices. However, the method is much more expensive in calculation than Gaussian elimination.

In programming the method, the header file `household.h` is used:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 #include <stdlib.h>
 4
 5 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 6 #define square(x) ((x)*(x))
 7
 8 int household(double **a,double **p,int m,int n);
 9 void pb(double **p, double *b, int m, int n);
10 void triangsolve(double **a, double *b, double *x, int n);
11 double **allocmatrix(int m, int n);
12 double *allocvector(int n);
```

On line 5 the absolute value, and on line 6 the square is defined. The function declarations in lines 8–12 will be discussed below. The memory allocation functions are described in the file `alloc.c`:

```
 1 #include "household.h"
 2
 3 double **allocmatrix(int m, int n)
 4   /* Allocate matrix. The following code segment allocates
 5      a contiguous block of pointers for the whole matrix
 6      of size m times n.                                  */
 7 {
 8   int i;
 9   double **a;
10   a=(double **) malloc((size_t)((m+1)*sizeof(double*)));
11   if (!a) {
12     printf("Allocation failure 1 in matrix\n");
13     exit(1);
14   }
15   /* Allocate the whole matrix: */
16   a[1]=(double *) malloc((size_t)(m*(n+1)*sizeof(double)));
17   if (!a[1]) {
18     printf("Allocation failure 2 in matrix\n");
19     exit(1);
20   }
21   /* Initialize the pointers pointing to each row: */
22   for (i=2; i<=m; i++) a[i]=a[i-1]+n+1;
23   /* Save the beginning of the matrix in a[0]; the
24      rows will be interchanged, the values of a[i]
25      for 1<=i<=n may change:                        */
26   a[0]=a[1];
27   return a;
28 }
29
30 double *allocvector(int n)
31 {
32   double *b;
33   b=(double *) malloc((size_t)((n+1)*sizeof(double)));
34   if (!b) {
35     printf("Allocation failure 1 in vector\n");
36     exit(1);
37   }
38   return b;
39 }
```

The function `allocmatrix` in lines 3–38 differs from the earlier function `allocmatrix` discussed on account of Gaussian elimination in that the matrix allocated here has size $m \times n$. This changes very little; on line 10, $m+1$ pointers are allocated, and in line 16, the size of the allocated block for the matrix is $m(n+1)$ (no memory needs to be allocated to the pointed `a[0]`; this pointer only needed for freeing the memory, and on line 22, each row pointer is incremented by $n+1$. The function `allocvector` in lines 30–39 is identical to the function by the same name discussed on account of Gaussian elimination. The file `household.c` contains the main functions implementing the method:

```
1 #include "household.h"
2
3 int household(double **a,double **p,int m,int n)
4 /* The matrices are size m times n */
```

```
 5 {
 6   const double near_zero=1e-20;
 7   int i,j,k,l,success;
 8   double **c, norm;
 9   c=allocmatrix(m,n);
10   success=1;
11   for(k=1;k<=n;k++) {
12     /* The entries of the matrix p could be stored in
13        the entries of the matrix a except for its main
14        diagonal -- at the price of some loss of
15        transparency in the method. The next line is
16        commented out, since it is superfluous. It
17        stores 0's in a part of the matrix p that is
18        supposed to be zero; but since these entries
19        of p are never used, it is not necessary to
20        do this.
21     for (i=1;i<k;i++) p[i][k]=0.; */
22     p[k][k]=0.;
23     for (i=k;i<=m;i++) p[k][k] += square(a[i][k]);
24     p[k][k]=sqrt(p[k][k]);
25     if ( a[k][k]<0. ) p[k][k]=-p[k][k];
26     p[k][k] += a[k][k];
27     if (absval(p[k][k])<=near_zero) {
28       success=0;
29       break;
30     }
31     for (i=k+1;i<=m;i++) p[i][k]=a[i][k];
32     norm=0.;
33     for (i=k;i<=m;i++) norm += square(p[i][k]);
34     norm=sqrt(norm);
35     for (i=k;i<=m;i++) p[i][k] /= norm;
36     for (i=k;i<=m;i++)
37       for (j=k;j<=n;j++) {
38         c[i][j]=a[i][j];
39         for (l=k;l<=m;l++)
40           c[i][j] -= 2.*p[i][k]*p[l][k]*a[l][j];
41       }
42     for (i=k;i<=m;i++)
43       for (j=k;j<=n;j++) a[i][j]=c[i][j];
44   }
45   free(c[0]);
46   free(c);
47   return success;
48 }
49
50 void pb(double **p, double *b, int m, int n)
51 {
52   int i,j,k,l;
53   double *c;
54   c=allocvector(m);
55   for(k=1;k<=n;k++) {
```

```
56      for (i=k;i<=m;i++) {
57          c[i]=b[i];
58          for (l=k;l<=m;l++) c[i] -= 2.*p[i][k]*p[l][k]*b[l];
59      }
60      for (i=k;i<=m;i++) b[i]=c[i];
61    }
62    free(c);
63  }
64
65  void triangsolve(double **a, double *b, double *x, int n)
66  {
67    int i,j,k,l;
68    x[n]= b[n]/a[n][n];
69    for (i=n-1;i>=1;i--) {
70      x[i]=b[i];
71      for (j=i+1;j<=n;j++) x[i] -= a[i][j]*x[j];
72      x[i] /= a[i][i];
73    }
74  }
```

The function `household` in lines 3–63 implements the Householder transformation of the matrix `**a`, the first parameter of this function. The next parameter `**p` contains the column vectors $\mathbf{v}_k$ $1 \le k \le n$ for the Householder transformations $H_{\mathbf{v}_k}$.[138] The last two parameters, `m` and `n`, contain the size of the matrix. The function returns 1 is the calculation is successful, otherwise it returns 0. The loop in lines 11–44 deals with the kth column of the matrix `**a`. In line 21, zeros are placed in the first `k` places of the column vector `p[.][k]`, but note that this line is commented out, so it has no effect. It perhaps makes the program easier to read by a human reader, or perhaps it could be useful in a modification of the method. In lines 22–31, the column vector $\bar{v}$ is calculated in the kth column of the matrix `**p`, and in lines 31–34 the norm of this vector is calculated. Before calculating the norm, in line 27 it is tested whether `p[k][k]` is too close to zero (the constant `near_zero` is set to 10^{-20}; if it is, the norm of this vector will also be too close to 0. In this case, the variable `success` will be set to 0 (false) (it was set to 1 on line 10); the calculation is then abandoned on line 29. Finally the vector is divided by its norm in line 35 to obtain the vector $\mathbf{v}_k$. In line 9, a matrix `**c` is allocated, to hold a temporary copy of the matrix `**a` while doing the matrix calculations. The calculation of the product $H_{\mathbf{v}_k} A_{k-1}$ is performed in lines 36–41; the result is placed in the matrix `**c`, and in lines 42–43, this result is copied into the matrix `**a`. The matrix `**c` is freed in lines 45–46, and the integer `success` is returned on line 47.

The function `pb` in lines 50–63 has the matrix `**p` as its first parameter, the second parameter is the column vector `*b`, the last two parameters `m` and `n` indicate the size of the matrix. When calling this function, the matrix `**p` is expected to contain the Householder transformations in its columns representing the matrix P (P is the product of these Householder transformations). `*b` contains the right-hand side of the equation $A\mathbf{x} = \mathbf{b}$ when the function is called, and it will contain the vector $P\mathbf{b}$ when the function returns. The calculation in lines 55–61 parallels those in lines 35–44. In line 44, memory for a vector `*c` is allocated for temporary storage, and this memory freed in line 62. The function `triangsolve` in lines 65–74 solves a system of linear equation whose left-hand side is represented by a triangular matrix; the method, backward substitution, was described on account of Gaussian elimination.

[138] At some loss of transparency, it might be possible to store the matrix `**p` in the part of the matrix `**a` that will become zero, but arranging this is considerably more complicated than in the case of Gaussian elimination; besides, the columns of the matrix `**p` contain one more nonzero element that there are available spaces, so the matrix `**a` would need to be enlarged by one row to make this possible.

```
 1 #include "household.h"
 2
 3 main()
 4 {
 5   /* This program reads in the coefficient matrix
 6      of a system of linear equations. The first
 7      entry is m, the number of equations, and
 8      second entry is n, the number of unknowns.
 9      The rest of the entries are the coefficients
10      and the right-hand sides; there must be m(n+1)
11      of these. The second entry must be an unsigned
12      integer, the other entries can be integers or
13      reals.                                        */
14   double **a, *b, **p, *x;
15   int i, j, m, n, readerror=0, success;
16   char s[25];
17   FILE *coefffile;
18   coefffile=fopen("coeffs", "r");
19   fscanf(coefffile, "%u", &m);
20   fscanf(coefffile, "%u", &n);
21   /* Create pointers to point to the rows of the matrix: */
22   a=allocmatrix(m,n);
23   /* Allocate right-hand side */
24   b=allocvector(m);
25   for (i=1;i<=m;i++) {
26     if ( readerror ) break;
27     for (j=1;j<=n+1;j++) {
28       if (fscanf(coefffile, "%s", s)==EOF) {
29         readerror=1;
30         printf("Not enough coefficients\n");
31         printf("i=%u j=%u\n", i, j);
32         break;
33       }
34       if ( j<=n ) {
35         a[i][j]=strtod(s,NULL);
36       }
37       else b[i]=strtod(s,NULL);
38     }
39   }
40   fclose(coefffile);
41   if ( readerror ) printf("Not enough data\n");
42   else {
43     p=allocmatrix(m,n);
44     success=household(a,p,m,n);
45     if ( success ) {
46       x=allocvector(n);
47       pb(p,b,m,n);
48       triangsolve(a,b,x,n);
49       printf("The solution is:\n");
50       for (i=1;i<=n;i++)
51         printf("x[%3i]=%16.12f\n",i,x[i]);
```

```
52       free(x);
53     }
54     else
55       printf("The solution could not be determined\n");
56     free(p[0]);
57     free(p);
58   }
59   /*
60   printf("\n");
61   for (i=1;i<=m;i++) {
62     for (j=1;j<=n;j++)
63       printf("%5.2f ",a[i][j]);
64     printf("  %5.2f\n",b[i]);
65   }
66   */
67   free(a[0]);
68   free(a);
69   free(b);
70 }
```

In line 20 the file `coeff` is opened, then the coefficients are read in. The first two entries of the file contain the size of the coefficient matrix, the rest of the entries contain the coefficients. In lines 25–39, the coefficients are read, and the file is closed on line 40. If there was a reading error (because the end of file was reached before the expected number of data were read, no calculation is done. Otherwise, lines 42–58 are executed. In line 43, the matrix `**p` is allocated to contain the vectors describing the Householder transformations, then the function `household` is invoked to calculate the entries of the matrix `**p` and triangularize the top square of the matrix `**a`. If this function was successful, then lines 45–43 are executed. In line 46, the vector `*x` to contain the solution of the equations is allocated memory space (this is freed on line 52), the right-hand side of the equation is calculated on line 47, and the equations are solved on line 48, and then the solutions are printed out. The matrix `**p` is freed in lines 56–57. In lines 60–65 the entries of the transformed matrix `**a` are printed out; however, these lines are commented out, so nothing is actually done here. These lines were useful for debugging, and rather than deleting them, they are only commented out. In a later modification of the program, they can again be useful for debugging. The matrix `**a` and the vector `*b` are freed in lines 67–69.

The program was run with the following input file `coeffs`:

```
 1 12 10
 2  2  3  5 -2  5    3 -4 -2  1  2    -3
 3  1  4 -2 -1  3   -2  1  3  4  1     2
 4  3 -1  2  1  3   -1  2  3  2 -1    -1
 5  9 -2 -1 -1  4    2  3 -1  1  4     8
 6 -1  2 -1  2 -1    3 -1 -3 -4  2    -3
 7 -4 -5  2  3  1    1  2  3 -1 -1    -4
 8 -1 -4 -2  3  4    1  2 -4 -3 -8     3
 9 -9 -4 -3 -1  9   -2 -2 -3  4  8     2
10 -1 -2  9  8 -7   -8  2 -4  3  1     5
11  8 -7  7  0  3   -5  3 -2  4  9     7
12  9 -6  8  1  2   -4  6 -3  5  8     9
13 -6 -4 -2  2  8    2 -3 -1  3  6     5
```

It is important to keep in mind that the first column represent line numbers and is not part of the file. The program produced the following printout:

```
 1 The solution is:
 2 x[  1]=  0.708141957416
 3 x[  2]= -0.677022884362
 4 x[  3]= -0.948640836111
 5 x[  4]=  0.611332408213
 6 x[  5]= -0.508849337986
 7 x[  6]=  0.471529531514
 8 x[  7]= -0.325609870356
 9 x[  8]= -0.863288481097
10 x[  9]=  1.870101220300
11 x[ 10]= -0.173441867530
```

Problem

1. Describe what an orthogonal matrix is.

39. THE POWER METHOD TO FIND EIGENVALUES

Let $n > 1$ be an integer. Given an $n \times n$ matrix A, an n-dimensional nonzero column vector $\mathbf{v}$ is called an *eigenvector* of A if there is a real number λ such that

$$A\mathbf{v} = \lambda\mathbf{v}.$$

The number λ is called an *eigenvalue*[139] of A. Writing I for the $n \times n$ identity matrix, the above equation can be written as

$$(A - \lambda I)\mathbf{v} = 0.$$

If $\mathbf{a}_i$ denotes the ith column vector of A, $\mathbf{e}_i$ denotes the ith column of the identity matrix I, and v_i denotes the ith component of v, then this equation can also be written as

$$\sum_{i=1}^{n} v_i(\mathbf{a}_i - \lambda\mathbf{e}_i) = 0;$$

i.e., the vectors $\mathbf{a}_i - \lambda\mathbf{e}_i$ $(1 \le i \le n)$ are linearly dependent. In other words, the determinant formed by these column vectors

$$p(\lambda) = \det(A - \lambda I)$$

is zero. $p(\lambda)$ is a polynomial of λ of degree n. The coefficients of this polynomial of course depend on the entries of the matrix A. This polynomial is called the *characteristic polynomial* of the matrix, and the equation $p(\lambda) = 0$ is called its *characteristic equation*. The characteristic equation has n solutions (which are usually complex numbers) when these solutions are counted with their multiplicities. The multiplicity of a solution of the characteristic equation is also called the multiplicity of the same eigenvalue. Thus an $n \times n$ matrix has n eigenvalues when these eigenvalues are counted with their multiplicities.

[139] The German word *eigen* means something like *owned by self*. Thus, eigenvalue of a matrix would mean something like a special value owned by the matrix. There have been attempts to translate the term *eigenvalue* into English as *characteristic value* or *proper value*. The former expression is sometimes used, the latter is almost never; in any case, the the mixed German-English term *eigenvalue* predominates in English (the German term is *Eigenwert*).

If the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_m$ (there may also be other eigenvalues) of an $n \times n$ matrix are distinct, then the corresponding eigenvectors are linearly independent. Indeed, writing $\mathbf{v}_i$ $(1 \leq i \leq m)$ for the corresponding eigenvectors,[140] assume that we have

$$\sum_{i=1}^{m} \alpha_i \mathbf{v}_i = 0,$$

where not all the numbers α_i are zero. Assume that the α_i's here are so chosen that the smallest possible number among them is not zero. Let the number of nonzero coefficients among the α_i be k, where $1 \leq k \leq n$. By rearranging the vectors and the corresponding eigenvalues, we may assume that $\alpha_i \neq 0$ for i with $1 \leq i \leq k$ and $\alpha_i = 0$ for $k < i \leq m$, where $k \geq 1$. That is,

$$\sum_{i=1}^{k} \alpha_i \mathbf{v}_i = 0.$$

Of course, $k = 1$ is not possible here, since it would mean that $\mathbf{v}_1 = 0$, whereas we assumed that an eigenvector cannot be the zero vector. Multiplying this equation by the matrix A on the left, and noting that $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$, we obtain

$$\sum_{i=1}^{k} \alpha_i \lambda_i \mathbf{v}_i = 0.$$

Multiplying the former linear relation by λ_k and subtracting from it the latter, we obtain

$$\sum_{i=1}^{k-1} (\alpha_i \lambda_k - \alpha_i \lambda_i) \mathbf{v}_i = 0.$$

None of the coefficients $\alpha_i \lambda_k - \alpha_i \lambda_i$ $(1 \leq i \leq k-1)$ is zero, since $\lambda_i \neq \lambda_k$ by our assumption. This relation contradicts the minimality of k.

That an $n \times n$ matrix does not need to have n linearly independent eigenvectors is shown by the matrix

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

The characteristic polynomial of this matrix is

$$\begin{vmatrix} 1-\lambda & 1 \\ 0 & 1-\lambda \end{vmatrix} = (\lambda - 1)^2,$$

so 1 is a double eigenvalue of this matrix, and so its only eigenvalue. The equation

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \lambda \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

with $\lambda = 1$ can be written as[141]

$$\begin{aligned} v_1 + v_2 &= v_1, \\ v_2 &= v_2. \end{aligned}$$

The only solution of these equations is $v_2 = 0$ and v_1 is arbitrary. That is, aside from its scalar multiples, $(1, 0)^T$ is the only eigenvector of the above matrix.[142]

[140] In case λ_i is a multiple eigenvalue (i.e., if it is a multiple root of the characteristic equation), then the eigenvector $\mathbf{v}_i$ may not be unique.

[141] We must have $\lambda = 1$ in this equation since 1 is the only eigenvalue of the above matrix.

[142] $\mathbf{v}^T$ is the transpose of the vector v. When describing a column vector, one can save space by writing it as the transpose of a row vector.

The companion matrix of a polynomial. Let

$$p(\lambda) = \lambda^n + \sum_{k=0}^{n-1} a_k \lambda^k = \lambda^n + a_{n-1}\lambda^{n-1} + \ldots + a_1\lambda + a_0 \qquad (n \geq 1)$$

be a polynomial . The $n \times n$ matrix

$$A = \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & -a_0 \\ 1 & 0 & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & 0 & \ldots & 0 & -a_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & -a_{n-1} \end{pmatrix}$$

is called the companion matrix of the polynomial p (for $n = 1$ take $A = -a_0$). The interesting point is that the characteristic polynomial of the matrix A is $(-1)^n p(\lambda)$. The importance of the companion matrix is that any method to find eigenvalues of a matrix can be used as a method to find zeros of polynomials.

We are going to prove the above statement about the characteristic polynomial of the matrix A. To do this, we have to show that the determinant

$$\det(A - \lambda I) = \begin{vmatrix} -\lambda & 0 & 0 & \ldots & 0 & -a_0 \\ 1 & -\lambda & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & -\lambda & \ldots & 0 & -a_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & -\lambda - a_{n-1} \end{vmatrix}$$

equals $(-1)^n p(\lambda)$. To this end, assuming $n > 1$ first, we will expand this determinant by its first row (see the Theorem on p. 158 in Section 34 on determinants). We obtain that this determinant equals

$$-\lambda \begin{vmatrix} -\lambda & 0 & 0 & \ldots & 0 & -a_1 \\ 1 & -\lambda & 0 & \ldots & 0 & -a_2 \\ 0 & 1 & -\lambda & \ldots & 0 & -a_3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -\lambda & -a_{n-2} \\ 0 & 0 & 0 & \ldots & 1 & -\lambda - a_{n-1} \end{vmatrix} + (-1)^{n+1}(-a_0) \begin{vmatrix} 1 & -\lambda & 0 & \ldots & 0 & 0 \\ 0 & 1 & -\lambda & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & \ldots & 1 & -\lambda \\ 0 & 0 & 0 & \ldots & 0 & 1 \end{vmatrix}.$$

The first determinant is the of form $\det(A_1 - \lambda I)$ where the $(n-1) \times (n-1)$ matrix A_1 is obtained from the matrix A by deleting the first row and the first column of A. So we can make the induction hypothesis that this determinant equals $(-1)^{n-1} p_1(\lambda)$, where

$$p_1(\lambda) = \lambda^{n-1} + \sum_{k=0}^{n-2} a_{k+1}\lambda^k = \lambda^{n-1} + a_{n-1}\lambda^{n-2} + \ldots + a_2\lambda + a_1.$$

The second determinant is 1, since it is the determinant of a triangular matrix, so the value of this determinant is just the product of this diagonal elements. Hence it follows that indeed

$$\det(A - \lambda I) = -\lambda \cdot (-1)^{n-1} p_1(\lambda) + (-1)^{n+1}(-a_0) = (-1)^n p(\lambda).$$

To complete the proof by induction, one needs to check that the statement is true for $n = 1$. In this case A is the determinant of the 1×1 matrix $(-a_0)$, so $\det(A - \lambda I) = \det(-a_0 - \lambda) = -(\lambda + a_0)$, verifying the assertion also in this case. Another proof can be given by expanding the determinant of the matrix $A - \lambda I$ by its last column; see Problem 8 below.

The power method. Let A be an $n \times n$ matrix with real numbers as entries. We will assume that the eigenvectors of the matrix A span the vector space R^n, and all its eigenvalues are real; in this case, the components of the eigenvectors are also real.[143] In an important case, namely when $A = (a_{ij})$ is a *symmetric matrix*, i.e., when $a_{ij} = a_{ji}$, this is valid. We will also assume that the matrix A has a single eigenvalue of largest absolute value. That is, we assume that A has a real eigenvalue λ of multiplicity 1 such that for all other eigenvalues ρ we have $|\rho| < |\lambda|$.[144] If this is the case, one can determine the eigenvalue of the matrix by what is called the power method.

To describe the method, assume that the eigenvalues of A are λ_i, $1 \le i \le n$, and assume that $|\lambda_1| > |\lambda_i|$ for all i with $2 \le i \le n$. We do not assume that the λ_i's for i with $2 \le i \le n$ are distinct. Let $\mathbf{v}_i$, $1 \le i \le n$ be the corresponding eigenvectors. Let $\mathbf{x}^{(0)}$ be an arbitrary n-dimensional column vector; for lack of a better choice, we may choose

$$\mathbf{x}^{(0)} = (1, 1, \dots, 1)^T.$$

As we assumed that the eigenvectors are linearly independent, we have

$$\mathbf{x}^{(0)} = \sum_{i=1}^{n} \alpha_i^{(0)} \mathbf{v}_i$$

for some numbers $\alpha_i^{(0)}$. If $\alpha_1^{(0)} = 0$, our choice for $\mathbf{x}^{(0)}$ is unlucky, and we must choose another starting value.[145]

We form the vectors

$$\mathbf{x}^{(k)} = \sum_{i=1}^{n} \alpha_i^{(k)} \mathbf{v}_i \tag{1}$$

and $\mathbf{u}^{(k)}$ as follows. We start with the arbitrarily chosen vector $\mathbf{x}^{(0)}$, already discussed above. If $\mathbf{x}^{(k)}$ has already been determined, we put

$$\mathbf{u}^{(k+1)} \stackrel{def}{=} A\mathbf{x}^{(k)} = \sum_{i=1}^{n} \lambda_i \alpha_i^{(k)} \mathbf{v}_i$$

and, denoting by μ_{k+1} the component of largest absolute value of the vector $\mathbf{u}^{(k+1)}$,[146] we put

$$\mathbf{x}^{(k+1)} = \sum_{i=1}^{n} \alpha_i^{(k+1)} \mathbf{v}_i = \frac{1}{\mu_{k+1}} \mathbf{u}^{(k+1)} = \sum_{i=1}^{n} \frac{\lambda_i \alpha_i^{(k)}}{\mu_{k+1}} \mathbf{v}_i,$$

This step normalizes the vector $\mathbf{x}^{(k+1)}$; that is, it ensures that the component of largest absolute value of $\mathbf{x}^{(k+1)}$ is 1. As the vectors $\mathbf{v}_i$ were assumed to be linearly independent, the last displayed equation means that

$$\alpha_i^{(k+1)} = \frac{\lambda_i \alpha_i^{(k)}}{\mu_{k+1}} \qquad \text{for each } i \text{ with } 1 \le i \le n.$$

---

[143] This is easy to see. The components of the eigenvector $\mathbf{v}$ corresponding to the eigenvalue λ are the coefficients of a linear combination showing that the column vectors of the matrix $A - \lambda I$ are linearly dependent. Given that these column vectors are in a vector space over the real numbers, these coefficients can be chosen to be real.

[144] The method converges even if λ has multiplicity greater than 1. We need to assume, however, that λ is real.

[145] If we perform the procedure described below in exact arithmetic, then the vectors obtained during the procedure will all be inside the subspace spanned by $\mathbf{v}_2, \dots, \mathbf{v}_n$, and thus we will not be able to determine the eigenvalue λ_1. In practice, roundoff errors will lead us out of the subspace spanned by $\mathbf{v}_2, \dots, \mathbf{v}_n$, and so the procedure may still converge to the largest eigenvalue, albeit slowly.

[146] If there is more than one component of $\mathbf{u}^{(k+1)}$ of the largest absolute value, pick μ_{k+1} to be the first one (i.e., the one with the smallest subscript) among them.

Thus

$$\frac{\alpha_i^{(k+1)}}{\alpha_1^{(k+1)}} = \frac{\lambda_i}{\lambda_1}\frac{\alpha_i^{(k)}}{\alpha_1^{(k)}}.$$

for each i with $1 \le i \le n$. The case $i = 1$ is of course of no interest. Using this equation repeatedly, we obtain

$$\frac{\alpha_i^{(k)}}{\alpha_1^{(k)}} = \left(\frac{\lambda_i}{\lambda_1}\right)^k \frac{\alpha_i^{(0)}}{\alpha_1^{(0)}}.$$

We assumed that λ_1 has absolute value larger than any other eigenvalue Thus,

$$\lim_{k\to\infty} \frac{\alpha_i^{(k)}}{\alpha_1^{(k)}} = \frac{\alpha_i^{(0)}}{\alpha_1^{(0)}} \cdot \lim_{k\to\infty} \left(\frac{\lambda_i}{\lambda_1}\right)^k = 0$$

for each i with $2 \le i \le n$. Hence, according to (1), we have[147]

$$\lim_{k\to\infty} \frac{\mathbf{1}}{\alpha_{\mathbf{1}}^{(\mathbf{k})}}\mathbf{x}^{(k)} = \sum_{i=1}^{n} \frac{\alpha_i^{(k)}}{\alpha_1^{(k)}}\mathbf{v}_i = \mathbf{v}_1.$$

As the component of the largest absolute value of $\mathbf{x}^{(k)}$ is 1, it follows that the limit

$$\alpha_1 \stackrel{def}{=} \lim_{k\to\infty} \alpha_1^{(k)} \tag{2}$$

exists,[148] and $\alpha_1 \neq 0$, at least in case $\mathbf{v}_1$ has a single component of the largest absolute value. Indeed, if $v_{1,r}$ is this component of $\mathbf{v}_1$ then the component of the largest absolute value of $\mathbf{x}^{(k)}$ is $x_r^{(k)} = 1$ for large enough k, and

$$\lim_{k\to\infty} \frac{\mathbf{1}}{\alpha_{\mathbf{1}}^{(\mathbf{k})}} x_r^{(k)} = v_{1,r}.$$

The limit in (2) is likely to exist even if $\mathbf{v}_1$ has several components of the largest absolute value. To ensure this it was important to always pick the first component of the largest absolute value of $\mathbf{u}^{(k+1)}$ as μ_{k+1}, though even this may not be enough to guarantee the existence of the limit in (2).

What may happen is that the components of $\mathbf{v}_1$ of the largest absolute value are $v_{1,r}$ and $v_{1,s}$ with $v_{1,r} = -v_{1,s}$, and for large k the largest component of $\mathbf{u}^{(k+1)}$ is its rth or sth component, depending on the value of k. Even in this case the limit

$$\alpha_1' = \lim_{k\to\infty} |\alpha_1^{(k)}|$$

will exist, and with slight modifications of the description given here one can still determine the eigenvector $\mathbf{v}_1$ and the associated eigenvalue λ_1. In such a modification, instead of choosing the component of the largest absolute value of $\mathbf{u}^{(k+1)}$, for large k one would always choose the same component of $\mathbf{u}^{(k+1)}$, and one would be satisfied if this component is close to having the largest absolute value among the components of $\mathbf{u}^{(k+1)}$.

If the limit in (2) exists, for large n we have $\mathbf{x}^{(k)} \approx \alpha_1\mathbf{v}_1$. Since any nonzero scalar multiple of the eigenvector $\mathbf{v}_1$ is an eigenvector, this means that $\mathbf{x}^{(k)}$ is nearly an eigenvector of A for the eigenvalue λ_1. This allows us to (approximately) determine λ_1 and a corresponding eigenvector.

In writing a program to calculate the eigenvalue of an $n \times n$ matrix using the power method, we use the header file `power.h`:

[147] The limit $\mathbf{z} = \lim_{k\to\infty} \mathbf{z}^{(k)}$ of a sequence of vectors can be defined componentwise, that is, by saying that the limit of each component of the vector $\mathbf{z}^{(k)}$ is the corresponding component of the $\mathbf{z}$.

[148] When a limit is infinite, we also say that the limit does not exist; so, by saying that the limit exists, we are also saying that it is finite.

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 #include <stdlib.h>
 4
 5 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 6
 7 double **allocmatrix(int n);
 8 double *allocvector(int n);
 9 int power(double **a, double *x, int n, double *lambda,
10         float tol, int maxits, int *success);
```

This file defines the absolute value function on line 5, and contains the declarations of the functions used in the program in lines 7–10. The file `alloc.c` is almost the same as the one that was used for Gaussian elimination:

```
 1 #include "power.h"
 2
 3 double **allocmatrix(int n)
 4   /* Allocate matrix. The following code segment allocates
 5      a contiguous block of pointers for the whole matrix.
 6      There is no real advantage for this here, because with
 7      pivoting, rows will be interchanged. So memory for rows
 8      could be allocated separately. On the other hand, even
 9      the largest matrix for which Gaussian elimination
10      is feasible occupies a relatively modest amount of
11      memory, there does not seem to be any disadvantage
12      in asking for a contiguous block. */
13 {
14   int i;
15   double **a;
16   a=(double **) malloc((size_t)((n+1)*sizeof(double*)));
17   if (!a) {
18     printf("Allocation failure 1 in matrix\n");
19     exit(1);
20   }
21   /* Allocate the whole matrix: */
22   a[1]=(double *) malloc((size_t)(n*(n+1)*sizeof(double)));
23   if (!a[1]) {
24     printf("Allocation failure 2 in matrix\n");
25     exit(1);
26   }
27   /* Initialize the pointers pointing to each row: */
28   for (i=2; i<=n; i++) a[i]=a[i-1]+n+1;
29   /* Save the beginning of the matrix in a[0]; the
30      rows will be interchanged, the values of a[i]
31      for 1<=i<=n may change:                      */
32   a[0]=a[1];
33   return a;
34 }
35
36 double *allocvector(int n)
37 {
```

```
38   double *b;
39   b=(double *) malloc((size_t)((n+1)*sizeof(double)));
40   if (!b) {
41     printf("Allocation failure 1 in vector\n");
42     exit(1);
43   }
44   return b;
45 }
```

In fact, the only change is line one, which now includes the file `power.h` rather than `gauss.h`. The file `power.c` contains the function `power` performing the power method.

```
 1 #include "power.h"
 2
 3 int power(double **a, double *x, int n, double *lambda,
 4         float tol, int maxits, int *success)
 5 {
 6   int i, j, itcount;
 7   double m, max, newx, *y, *u;
 8   u=allocvector(n);
 9   for (i=1;i<=n;i++) x[i]=1.0;
10   for (itcount=1;itcount<=maxits;itcount++) {
11     /* Form y=Ax abd record maximum element max */
12     max=0.;
13     for (i=1;i<=n;i++) {
14       m=0.;
15       for (j=1;j<=n;j++) m += a[i][j]*x[j];
16       if ( absval(m)>absval(max) ) max=m;
17       u[i]=m;
18     }
19     /* Normalize u and test for convergence */
20     *success=1;
21     for (i=1;i<=n;i++) {
22       newx=u[i]/max;
23       if ( absval(newx-x[i])>tol ) *success=0;
24       x[i]=newx;
25     }
26     if ( *success ) {
27       *lambda=max;
28       break;
29     }
30   }
31   free(u);
32   return itcount;
33 }
```

The function `power` introduced in lines 3–4 returns the number of iterations performed. Its parameters of include the pointer `**a` to the coefficients of the matrix A, the vector `*x` that will store the components of the eigenvector, the integer `n` specifying the size of the matrix, the pointer `*lambda` to the location where the eigenvalue will be calculated, the tolerance `tol`, the maximum number of iterations `maxits` allowed. and the pointer to the integer `success` indicating whether the procedure converged.

On line 8, space is allocated for the vector **u**. In line 9 the vector **x** is given the initial value

$(1, 1, \ldots, 1)^T$. The iterations to find the new values of **u** and **x** are carried out in the loop in lines 10–30. In lines 13–18, the vector **u** is calculated, and in lines 21–25, the next value of the vector **x** will be calculated. On line 20, `*success` is set to be 1 (true), but on line 23 it will be made 0 (false) unless the ith component of the iteration converges; thus, it will stay one on line 26 if the iteration converged for every value of i. If on line 26 it is found that the iteration converged, `*lambda` is given its value as the largest component of **u** (it is easy to see that this is close to the eigenvalue, since the largest component of the previous value of **x** was 1). The vector **x** at this point will contain an approximation of the eigenvector, and the loop is exited on line 28. On line 31, the space allocated for **u** is freed, and on line 32, the number of iterations is returned.

The function `power.c` is called in the file `main.c`:

```
 1 #include "power.h"
 2
 3 main()
 4 {
 5   /* This program reads in the elements of a square
 6      matrix the eigenvalue of which is sought. The
 7      first entry is the required tolerance, and the
 8      second entry is n, the size of the matrix. The
 9      rest of the entries are the elements of the
10      matrix. The second entry must be an unsigned
11      integer, the other entries can be integers or
12      reals.                                        */
13   double **a, *x, lambda;
14   float tol;
15   int n, i, j, subs, readerror=0, success, itcount;
16   char s[25];
17   FILE *coefffile;
18   coefffile=fopen("coeffs", "r");
19   fscanf(coefffile, "%f", &tol);
20   printf("Tolerance used: %g\n", tol);
21   fscanf(coefffile, "%u", &n);
22   a=allocmatrix(n);
23   for (i=1;i<=n;i++) {
24     if ( readerror ) break;
25     for (j=1;j<=n;j++) {
26       if (fscanf(coefffile, "%s", s)==EOF) {
27         readerror=1;
28         printf("Not enough coefficients\n");
29         printf("i=%u j=%u\n", i, j);
30         break;
31       }
32       a[i][j]=strtod(s,NULL);
33     }
34   }
35   fclose(coefffile);
36   if ( readerror ) printf("Not enough data\n");
37   else {
38     x=allocvector(n);
39 itcount=power(a,x,n,&lambda,tol,100,&success);
40     if ( success ) {
41       printf("%u iterations were used to find"
```

```
42              " the largest eigenvalue.\n", itcount);
43       printf("The eigenvalue is %16.12f\n", lambda);
44       printf("The eigenvector is:\n");
45       for (i=1;i<=n;i++)
46         printf("x[%3i]=%16.12f\n",i,x[i]);
47     }
48     else
49       printf("The eigenvalue could not be determined\n");
50     free(x);
51   }
52   free(a[0]); free(a);
53 }
```

The matrix A is contained in the file `coeffs`. The first entry of this file is the tolerance `tol`, the second entry is the size `n` of the matrix (this must be an integer, the first entry must be of type `float`, and the rest of the entries can be real numbers such as `float` or `double` or integers). The file `coeffs` is opened on line 18. After `tol` and `n` are read in and printed out in lines 19–21, space for the matrix A is allocated on line 22. In lines 23–34, the matrix is read in, and on line 35 the file `coeffs` is closed. If there were not enough coefficients, an error message is printed out on line 36, otherwise the calculation is continued in line 37. On line 38, space is allocated for the vector `*x`; this will contain the eigenvector of the matrix in the end. On line 39, the power method is called, and, if successful, the results of the calculation are printed out in line 40–47; otherwise an error message is printed on line 49. The space reserved for the vector `x` is freed on line 50 (this must be freed inside the `else` statement of lines 37–51, where it was allocated). The space allocated for the matrix is freed on line 52.

The following input file `coeffs` was used to run the program:

```
1 5e-14
2 5
3 1  2  3  5  2
4 3  4 -2  2  3
5 1  1  1  1 -1
6 1  2  1 -1  3
7 2  1 -1  1  2
```

The output obtained was as follows:

```
1 Tolerance used: 5e-14
2 28 iterations were used to find the largest eigenvalue.
3 The eigenvalue is   8.310753728109
4 The eigenvector is:
5 x[  1]=  0.829693363275
6 x[  2]=  1.000000000000
7 x[  3]=  0.253187383172
8 x[  4]=  0.478388026222
9 x[  5]=  0.457090784061
```

Problems

1. *Question* 1. Find the characteristic polynomial of the matrix

$$A = \begin{pmatrix} 3 & 1 \\ 4 & 3 \end{pmatrix}.$$

Question 2. Find the eigenvalues of the matrix A.

Solution. *Question* 1. The characteristic polynomial is

$$p(\lambda) = \begin{vmatrix} 3-\lambda & 1 \\ 4 & 3-\lambda \end{vmatrix} = (3-\lambda)(3-\lambda) - 1 \cdot 4 = 9 - 6\lambda + \lambda^2 - 4 = \lambda^2 - 6\lambda + 5.$$

Question 2. The eigenvalues are the solutions of the equation $p(\lambda) = 0$. One can use the quadratic formula to solve this equation; however, the characteristic polynomial is easy to factor:

$$p(\lambda) = \lambda^2 - 6\lambda + 5 = (\lambda - 1)(\lambda - 5).$$

Thus, the solutions of the equation $p(\lambda) = 0$ are $\lambda = 1$ and $\lambda = 5$; i.e., the eigenvalues of A are 1 and 5.

2. *Question* 1. Find the characteristic polynomial of the matrix

$$A = \begin{pmatrix} 2 & 2 \\ 6 & 3 \end{pmatrix}.$$

Question 2. Find the eigenvalues of the above matrix.

Solution. *Question* 1. The characteristic polynomial is

$$p(\lambda) = \begin{vmatrix} 2-\lambda & 2 \\ 6 & 3-\lambda \end{vmatrix} = (2-\lambda)(3-\lambda) - 2 \cdot 6 = 6 - 5\lambda + \lambda^2 - 12 = \lambda^2 - 5\lambda - 6.$$

Question 2. The eigenvalues are the solutions of the equation $p(\lambda) = 0$. One can use the quadratic formula to solve this equation; however, the characteristic polynomial is easy to factor:

$$p(\lambda) = \lambda^2 - 5\lambda - 6 = (\lambda + 1)(\lambda - 6).$$

Thus, the solutions of the equation $p(\lambda) = 0$ are $\lambda = -1$ and $\lambda = 6$; i.e., the eigenvalues of A are -1 and 6.

3. Find the eigenvectors of the matrix

$$A = \begin{pmatrix} 2 & 3 & 1 \\ 0 & 1 & 4 \\ 0 & 0 & 3 \end{pmatrix}.$$

Solution. The eigenvalues of a triangular matrix are the elements in the main diagonal. The easiest way to see this is to note that the determinant of a triangular matrix is the product of the elements in the main diagonal. Thus, the characteristic polynomial of the above matrix is

$$p(\lambda) = \begin{vmatrix} 2-\lambda & 3 & 1 \\ 0 & 1-\lambda & 4 \\ 0 & 0 & 3-\lambda \end{vmatrix} = (2-\lambda)(1-\lambda)(3-\lambda).$$

Thus, the eigenvalues, i.e., the solutions of the equation $p(\lambda) = 0$, are 2, 1, 3. The eigenvectors are the solutions of the equations $A\mathbf{x} = \lambda\mathbf{x}$ with these values of λ; i.e., the solutions of the equations $A\mathbf{x} = 2\mathbf{x}$, $A\mathbf{x} = \mathbf{x}$, and $A\mathbf{x} = 3\mathbf{x}$. The equation $A\mathbf{x} = 2\mathbf{x}$ can be written as

$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 1 & 4 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = 2 \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

This equation can be written as

$$\begin{aligned} 2x_1 + 3x_2 + \ \ x_3 &= 2x_1 \\ x_2 + 4x_3 &= 2x_2 \\ 3x_3 &= 2x_3 \end{aligned}$$

The third equation here can be satisfied only with $x_3 = 0$. Substituting this into the second equation, this equation becomes $x_2 = 2x_2$. This can also be satisfied only with $x_2 = 0$. Substituting $x_2 = x_3 = 0$ into the first equation, this equation becomes $2x_1 = 2x_1$. This is satisfied for any value of x_1. I.e., the eigenvector $\mathbf{x}$ corresponding to the eigenvalue 2 is

$$\mathbf{x} = (x_1, 0, 0)^T = x_1(1, 0, 0)^T.$$

That is, an eigenvector corresponding to the eigenvalue 2 is

$$\mathbf{x} = (1, 0, 0)^T.$$

As we just saw, any scalar multiple of this is also an eigenvector; this, however, is hardly worth pointing out, since a scalar multiple of an eigenvector is always an eigenvector.

As for the eigenvector corresponding to the eigenvalue 1, the equation $A\mathbf{x} = \mathbf{x}$ can be written as

$$\begin{aligned} 2x_1 + 3x_2 + \ \ x_3 &= x_1 \\ x_2 + 4x_3 &= x_2 \\ 3x_3 &= x_3 \end{aligned}$$

The third equation here can only be satisfied with $x_3 = 0$. Substituting this into the second equation, this equation becomes $x_2 = x_2$. We might as well choose $x_2 = 1$; any other choice would only give a scalar multiple of the eigenvector we are going to obtain. Substituting $x_3 = 0$ and $x_2 = 1$ into the first equation, this equation becomes $2x_1 + 3 = x_1$, i.e., $x_1 = -3$. That is, the eigenvector corresponding to the eigenvalue 1 is

$$\mathbf{x} = (-3, 1, 0)^T.$$

As for the eigenvector corresponding to the eigenvalue 3, the equation $A\mathbf{x} = 3\mathbf{x}$ can be written as

$$\begin{aligned} 2x_1 + 3x_2 + \ \ x_3 &= 3x_1 \\ x_2 + 4x_3 &= 3x_2 \\ 3x_3 &= 3x_3 \end{aligned}$$

The third equation here is satisfied for any value of x_3. We might as well choose $x_3 = 1$, since any other choice of x_3 would only give a scalar multiple of the eigenvector we are going to obtain. The second equation then becomes $x_2 + 4 = 3x_2$, i.e., $x_2 = 2$. Substituting this into the first equation, we obtain $2x_1 + 3 \cdot 2 + 1 = 3x_1$, i.e., $x_1 = 7$. This gives the eigenvector

$$\mathbf{x} = (7, 2, 1)^T.$$

Thus, the eigenvectors are $(1, 0, 0)^T$, $(-3, 1, 0)^T$, $(7, 2, 1)^T$, corresponding to the eigenvalues 2, 1, 3, in turn.

4. Find all eigenvectors of the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Solution. The only eigenvalue of A is 1, since A is a triangular matrix, and the only element in the main diagonal is 1. The equation $A\mathbf{x} = \mathbf{x}$ with $\mathbf{x} = (x_1, x_2, x_3, x_4)^T$ can be written as

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}.$$

This equation can be written as the following system of equations:

$$\begin{aligned} x_1 + x_2 &= x_1 \\ x_2 + x_3 &= x_2 \\ x_3 + x_4 &= x_3 \\ x_4 &= x_4 \end{aligned}$$

The first equation here implies $x_2 = 0$, the second equation implies $x_3 = 0$, and the third equation implies $x_4 = 0$. The fourth equation is always satisfied. In the first equation we can take any value of x_1. We must have $x_1 \neq 0$, since an eigenvector cannot be the zero vector. Thus we have $\mathbf{x} = x_1(1, 0, 0, 0)^T$. We might as well take $x_1 = 1$ here, since an eigenvector is only determined up to a scalar multiple. That is, aside from its scalar multiples, $(1, 0, 0, 0)^T$ is the only eigenvector.

5. Given the matrix

$$A = \begin{pmatrix} 2 & 1 \\ 8 & 9 \end{pmatrix},$$

do one step of the power method to determine its eigenvalues, using the starting vector $\mathbf{x}^{(0)} = (1, 1)^T$.

Solution. We have

$$\mathbf{u}^{(1)} = A\mathbf{x}^{(0)} = (3, 17)^T,$$

and

$$\mathbf{x}^{(1)} = \frac{1}{17}u^{(1)} = \left(\frac{3}{17}, 1\right)^T.$$

This gives the first approximation 17 (the component of $\mathbf{u}^{(1)}$ with the largest absolute value) to the largest eigenvalue. The actual values of the eigenvalues are 10 and 1.

6. Explain what it means for a column vector $\mathbf{v}$ to be an eigenvector of a square matrix (of the same size).

7. Assume A is an invertible $n \times n$ matrix, and B is an arbitrary $n \times n$ matrix. Show that the matrices AB and BA have the same eigenvalues.

Solution. The eigenvalues of AB are the zeroes of its characteristic polynomial

$$\det(AB - \lambda I),$$

where λ is the unknown, and I is the $n \times n$ identity matrix, and $\det(C)$ denotes the determinant of the matrix C. Noting that we have

$$\det(CD) = \det(C) \cdot \det(D)$$

for any two $n \times n$ matrices C and D, we have

$$\begin{aligned}\det(AB - \lambda I) &= \det(A(B - \lambda A^{-1})) = \det(A)\det(B - \lambda A^{-1}) = \det(B - \lambda A^{-1})\det(A) \\ &= \det((B - \lambda A^{-1})A) = \det(BA - \lambda I),\end{aligned}$$

showing that the characteristic polynomials of AB and BA are the same.

Note. The result is true even if A is not invertible. In fact, even if A is not invertible, the matrix $A - \eta I$ is invertible except for finitely many values of η (namely, it is invertible exactly when η is not an eigenvalue of A). So, the matrix $A - \eta I$ will be invertible if $\eta \neq 0$ is close enough to 0. For such an η, we have

$$\det((A - \eta I)B - \lambda I) = \det(B(A - \eta I) - \lambda I).$$

Taking the limit when $\eta \to 0$, we can see that

$$\det(AB - \lambda I) = \det(BA - \lambda I).$$

Even more generally, one can show that given any commutative ring $\mathbf{R}$ and two $n \times n$ matrices A and B over $\mathbf{R}$, the equation

$$\det(AB - \lambda I) = \det(BA - \lambda I).$$

is an identity in the polynomial ring $\mathbf{R}[\lambda]$. This follows from from the fact that this is an identity over the ring of integers, usually denoted as $\mathbb{Z}$ (we mean $\mathbb{Z}$ itself and not the polynomial ring $\mathbb{Z}[\lambda]$; that is, λ in this latter context is an integer, and not an abstract variable). See Section 1.1 in [Ma, p. 2].

8. Show that the characteristic polynomial of the companion matrix A of the polynomial

$$p(\lambda) = \lambda^n + \sum_{k=0}^{n-1} a_k \lambda^k = \lambda^n + a_{n-1}\lambda^{n-1} + \ldots + a_1\lambda + a_0$$

is $(-1)^n p(\lambda)$ by expanding the determinant $A - \lambda I$ by its last column.

Hint. By crossing out the row and the column of the element $-a_k$ (or $-\lambda - a_{n-1}$ in case $k = n-1$) in the last column of the matrix $A - \lambda I$, we obtain the matrix

$$C = \begin{pmatrix} P & \mathbf{0} \\ \mathbf{0}^T & Q \end{pmatrix},$$

where P is a $k \times k$ lower triangular matrix with all its diagonal elements equal to $-\lambda$, Q is an $(n-k-1) \times (n-k-1)$ upper triangular matrix with all its diagonal elements equal to 1, $\mathbf{0}$ is $k \times (n-k-1)$ matrix with all its elements 0, and $\mathbf{0}^T$ is its transpose. It is easy to show by the Leibniz formula we used to define determinants (see p. 154 in Section 34) that $\det P = (-\lambda)^k$, $\det Q = 1$, and $\det C = \det P \cdot \det Q$. For this discussion to make sense in case $k = 0$ (when P is an empty matrix, or a 0×0 matrix) or $k = n - 1$ (when Q is an empty matrix), take the determinant of an empty matrix to be 1.

40. THE INVERSE POWER METHOD

The *inverse power method* is the power method applied with the matrix $(A - sI)^{-1}$ for some number s.[149] Assuming λ is an eigenvalue of A corresponding to the eigenvector $\mathbf{v}$, this vector is also an eigenvector of $(A - sI)^{-1}$ with eigenvalue $(\lambda - s)^{-1}$. Indeed, we have

$$(A - sI)^{-1}\mathbf{v} = (\lambda - s)^{-1}\mathbf{v}.$$

To see this, note that

$$(A - sI)\mathbf{v} = A\mathbf{v} - sI\mathbf{v} = \lambda\mathbf{v} - s\mathbf{v} = (\lambda - s)\mathbf{v}.$$

Multiplying this equation by $(\lambda - s)^{-1}(A - sI)^{-1}$ on the left, we obtain the equation above (after reversing the sides).[150]

If the eigenvalues of A are $\lambda_1, \ldots, \lambda_n$, then the eigenvalues of $(A - sI)^{-1}$ will be $(\lambda_1 - s)^{-1}$, $\ldots$, $(\lambda_n - s)^{-1}$.[151] Thus, if λ_1 is the closest eigenvalue of A to s, $(\lambda - s)^{-1}$ will be the eigenvalue of $(A - sI)^{-1}$ with the largest absolute value. Therefore, the power method will be well-suited to determine this eigenvalue of $(A - sI)^{-1}$, assuming that λ_1 is real.

That is, as in the power method, one starts with an arbitrary vector $\mathbf{x}^{(0)}$. As before, unless a better choice is available, we may put

$$\mathbf{x}^{(0)} = (1, 1, \ldots, 1)^T.$$

Having determined $\mathbf{x}^{(k)}$, we write

$$\mathbf{u}^{(k+1)} = (A - sI)^{-1}\mathbf{x}^{(k)},$$

and

$$\mathbf{x}^{(k+1)} = \frac{1}{\mu_{k+1}}\mathbf{u}^{(k+1)},$$

where μ_{k+1} is the component of the largest absolute value of the vector $\mathbf{u}^{(k+1)}$.

When using the inverse power method, one does not need to determine the inverse of the matrix $A - sI$. Instead, one finds the LU-factorization of the matrix $A - sI$, and then finds

$$\mathbf{u}^{(k+1)} = (A - sI)^{-1}\mathbf{x}^{(k)}$$

by solving the equation

$$(A - sI)\mathbf{u}^{(k+1)} = \mathbf{x}^{(k)}.$$

The LU-factorization need to be calculated only once, and then can be used repeatedly for solving this equation with different k.

In a program implementing the inverse power method, we use the header file `inv_power.h`

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4
5 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
```

[149] There are variants of the inverse power method for which this statement is not quite accurate.

[150] In this argument, we need to assume that s itself is not an eigenvalue of A; this will guarantee that the matrix $A - sI$ is invertible.

[151] One can reverse the above argument to show that if μ is an eigenvalue of the matrix $(A - sI)^{-1}$, then $\mu^{-1} + s$ is an eigenvalue of A. Writing $\lambda = \mu^{-1} + s$, we have $\mu = (\lambda - s)^{-1}$. That is, every eigenvalue of $(A - sI)$ is of form $(\lambda - s)^{-1}$ for some eigenvalue λ of A.

```
 6
 7 double **allocmatrix(int n);
 8 double *allocvector(int n);
 9 int inv_power(double **a, double *x, int n, float s,
10         double *lambda, float tol, int maxits, int *success);
11 void swap_pivot_row(double **a, int col, int n);
12 void LUfactor(double **a, int n, int *success);
13 void LUsolve(double **a, int n, double *b);
```

Line 5 defines the function `absval` to take the absolute value. The file `alloc.c` handling memory allocation is virtually identical to the one used for Gaussian elimination.

```
 1 #include "inv_power.h"
 2
 3 double **allocmatrix(int n)
 4   /* Allocate matrix. The following code segment allocates
 5      a contiguous block of pointers for the whole matrix.
 6      There is no real advantage for this here, because with
 7      pivoting, rows will be interchanged. So memory for rows
 8      could be allowcated separately. On the other hand, even
 9      the largest matrix for which Gaussian elimination
10      is feasible occupies a relatively modest amount of
11      memory, there does not seem to be any disadvantage
12      in asking for a contiguous block. */
13 {
14   int i;
15   double **a;
16   a=(double **) malloc((size_t)((n+1)*sizeof(double*)));
17   if (!a) {
18     printf("Allocation failure 1 in matrix\n");
19     exit(1);
20   }
21   /* Allocate the whole matrix: */
22   a[1]=(double *) malloc((size_t)(n*(n+1)*sizeof(double)));
23   if (!a[1]) {
24     printf("Allocation failure 2 in matrix\n");
25     exit(1);
26   }
27   /* Initialize the pointers pointing to each row: */
28   for (i=2; i<=n; i++) a[i]=a[i-1]+n+1;
29   /* Save the beginning of the matrix in a[0]; the
30      rows will be interchanged, the values of a[i]
31      for 1<=i<=n may change:                      */
32   a[0]=a[1];
33   return a;
34 }
35
36 double *allocvector(int n)
37 {
38   double *b;
39   b=(double *) malloc((size_t)((n+1)*sizeof(double)));
40   if (!b) {
```

```
41     printf("Allocation failure 1 in vector\n");
42     exit(1);
43   }
44   return b;
45 }
```

The only difference is that the file `inv_power.h` is included on line 1 instead of `gauss.h`, an inessential change (since the latter would also work, since none of the functions declared in the file `inv_power.h` are needed in the file `alloc.c` The file `lufactor.c` implements LU-factorization:

```
 1 #include "inv_power.h"
 2
 3 void swap_pivot_row(double **a, int col, int n)
 4   /* Scans the elements a[col][i] for col<=i<=n to find the
 5      element c[col][pivi] with the largest absolute value,
 6      and then the rows a[col] and a[pivi] are interchanged. */
 7 {
 8   double maxel, *rowptr;
 9   int i,pivi;
10   maxel=absval(a[col][col]); pivi=col;
11   for (i=col+1;i<=n;i++) {
12     if ( absval(a[i][col])>maxel ) {
13       maxel=absval(a[i][col]); pivi=i;
14     }
15   }
16   if ( pivi !=col ) {
17     rowptr=a[col]; a[col]=a[pivi]; a[pivi]=rowptr;
18   }
19 }
20
21 void LUfactor(double **a, int n, int *success)
22   /* Uses partial pivoting */
23 {
24   const double assumedzero=1e-20;
25   double pivot, mult;
26   int i, j, k;
27   for (j=1;j<n;j++) {
28     swap_pivot_row(a, j, n); pivot=a[j][j];
29     if ( absval(pivot)<=assumedzero ) {
30       *success=0; return;
31     }
32     else *success=1;
33     for (i=j+1; i<=n; i++) {
34       mult = a[i][j] /= pivot;
35       for (k=j+1;k<=n;k++) a[i][k]-=mult*a[j][k];
36     }
37   }
38 }
```

This is identical to the file with the same name given on account of Gaussian elimination, except that the first line now includes the file `inv_power.h` instead of `gauss.h`.[152] The file `lusolve.c`

[152] In fact, there is no need to change the file `lusolve.c`, and the program would work even if the file `gauss.h` were

implements the solution of a linear system of equation given the LU-factorization of the matrix of the equation:

```
 1 #include "inv_power.h"
 2
 3 void LUsolve(double **a, int n, double *b)
 4 {
 5   int i, j;
 6   /* We want to solve P^-1 LUx=b. First we calculate c=Pb.
 7      The elements of c[i] will be stored in a[i][0], not used
 8      in representing the matrix. */
 9   for (i=1;i<=n;i++) a[i][0]=b[1+(a[i]-a[0])/(n+1)];
10   /* Here is why we preserved the beginning of the matrix
11      in a[0]. The quantity 1+(a[i]-a[1])/(n+1) gives
12      the original row index of what ended up as row i
13      after the interchanges. Next we calculate y=Ux=L^-1 c.
14      y will be stored the same place c was stored:
15      y[i] will be a[i][0]. */
16   for (i=2;i<=n;i++)
17     for (j=1;j<i;j++) a[i][0] -= a[i][j]*a[j][0];
18   /* Finally, we calculate x=U^-1 y. We will put x in
19      the same place as y: x[i]=a[i][0]. */
20   a[n][0] /= a[n][n];
21   for (i=n-1;i>=1;i--) {
22     for (j=i+1;j<=n;j++) a[i][0] -= a[i][j]*a[j][0];
23       a[i][0] /= a[i][i];
24   }
25 }
```

This file is again similar to the one used for Gaussian elimination. Again, the file `inv_power.h` is included on line 1 instead of `gauss.h` However, we retained only the function `LUsolve`; that is, we deleted the functions `backsubst` and `improve` used for iterative improvement, since iterative improvement is not appropriate in the present case.[153] The inverse power method itself is implemented in the file `inv_power.c`:

```
 1 #include "inv_power.h"
 2
 3 int inv_power(double **a, double *x, int n, float s,
 4         double *lambda, float tol, int maxits, int *success)
 5 {
 6   int i, j, itcount;
 7   double m, max, newx, *y, *u;
 8   /* Form A-sI and place the result in A */
 9   for (i=1;i<=n;i++) a[i][i] -= s;
10   LUfactor(a,n,success);
11   if ( !*success ) return 0;
12 /* I am here */
13   u=allocvector(n);
14   for (i=1;i<=n;i++) x[i]=1.0;
```

included instead of `inv_power.h`. In fact, the file `lusolve` uses no function defined outside the file.

[153] Whatever one may get out of iterative improvement, it is much more efficient to improve the approximation to the eigenvalue and the eigenvector is to perform the inverse power method with a closed approximation s to the eigenvalue λ being calculated.

```
15   for (itcount=1;itcount<=maxits;itcount++) {
16     /* Solve (A-sI)y=x and place the result in u */
17     LUsolve(a,n,x);
18     max=0.;
19     for (i=1;i<=n;i++) {
20       u[i] = a[i][0];
21       if ( absval(u[i])>absval(max) ) max=u[i];
22     }
23     /* Normalize u and test for convergence */
24     *success=1;
25     for (i=1;i<=n;i++) {
26       newx=u[i]/max;
27       if ( absval(newx-x[i])>tol ) *success=0;
28       x[i]=newx;
29     }
30     if ( *success ) {
31       *lambda=s+1./max;
32       break;
33     }
34   }
35   free(u);
36   return itcount;
37 }
```

The function `inv_power` defined in lines 3–37 is very similar to the `power` function used in the power method. It is useful to make a line-by-line comparison between this file and the file `power.c` discussed on account of the power method. The parameter list is the same as the one used for the power method. That is, the first parameter `**a` represents the entries of the matrix, `*x` will contain the eigenvector at the end of the calculation, `n` is the size of the matrix, `tol` is the tolerance, or permissible error, in the calculation, `maxits` is the maximum allowed number of iterations, and `*success` indicates whether the method was successful. In line 9, the matrix $A - sI$ is calculated, and stored in the same locations where A itself was stored, and in line 10 its LU-factorization is calculated (and, again, it is stored in the same locations). In line 11, if LU-factorization was unsuccessful, there is nothing more to be done (except, perhaps, to try the inverse power method with a different value of s). Then, on line 13, memory for the vector $\mathbf{u}$ is allocated. The vector $\mathbf{x}$ is initialized on line 14, and in lines 15–34, successive values of the vectors $\mathbf{u}$ and $\mathbf{x}$ are calculated. On lines 17–22, the vector $\mathbf{u} = (A - sI)^{-1}\mathbf{x}$ is calculated; note that `LUsolve` stores the solution of this equation in column zero of the array `**a`, and this solution is copied into the vector `*x` in lines 19–21. The element of maximum absolute value is also calculated in these lines, and stored as the variable `max`. In lines 25–29 the normalized vector $\mathbf{u}$ is stored in $\mathbf{x}$, and convergence is tested. For the convergence test, the integer `*success` is set to be 1 (true) on line 24, and it is changed to false on line 27 if the test on this line fails for any value of i. If the process is successful, on line 31, the eigenvalue of A is calculated, using the equation

$$\lambda = s + \frac{1}{\mu},$$

(i.e., $\mu = (\lambda - s)^{-1}$)m where μ is the eigenvalue calculated for the matrix $(A - sI)^{-1}$. On line 35, the memory allocated for the vector `u` is freed, and the number of iterations is returned on line 35.

The calling program is contained in the file `main.c`, to be discussed soon. This program read the values of the entries of the matrix from the file `coeffs`

```
1 9.2
```

```
 2 5e-14
 3 5
 4 1  2  3  5  2
 5 3  4 -2  2  3
 6 1  1  1  1 -1
 7 1  2  1 -1  3
 8 2  1 -1  1  2
```

(Recall that the numbers in the first column are line numbers.) The first entry in this file is the value of `s`, the second one is the tolerance `tol`, the third one is the value of `n`, and the rest of the entries are the entries of the matrix A. The file `main.c` is as follows:

```
 1 #include "inv_power.h"
 2
 3 main()
 4 {
 5   /* This program reads in the elements of a square
 6      matrix from the file coeffs the eigenvalue of
 7      which is sought. The first entry is an
 8      approximate eigenvalue, the second entry
 9      entry is the required tolerance, and the
10      third entry is n, the size of the matrix.
11      The rest of the entries are the elements of the
12      matrix. The first entry must be of type float,
13      the third entry an unsigned integer, the other
14      entries can be integers or reals. The inverse
15      power method is used to determine the
16      eigenvalue of the matrix           */
17   double **a, *x, lambda;
18   float s, tol;
19   int n, i, j, subs, readerror=0, success, itcount;
20   char ss[25];
21   FILE *coefffile;
22   coefffile=fopen("coeffs", "r");
23   fscanf(coefffile, "%f", &s);
24   printf("Approximate eigenvalue used: %8.5f\n", s);
25   fscanf(coefffile, "%f", &tol);
26   printf("Tolerance used: %g\n", tol);
27   fscanf(coefffile, "%u", &n);
28   a=allocmatrix(n);
29   for (i=1;i<=n;i++) {
30     if ( readerror ) break;
31     for (j=1;j<=n;j++) {
32       if (fscanf(coefffile, "%s", ss)==EOF) {
33         readerror=1;
34         printf("Not enough coefficients\n");
35         printf("i=%u j=%u\n", i, j);
36         break;
37       }
38       a[i][j]=strtod(ss,NULL);
39     }
40   }
```

```
41   fclose(coefffile);
42   if ( readerror ) printf("Not enough data\n");
43   else {
44     x=allocvector(n);
45 itcount=inv_power(a,x,n,s,&lambda,tol,100,&success);
46     if ( success ) {
47       printf("%u iterations were used to find"
48              " the largest eigenvalue.\n", itcount);
49       printf("The eigenvalue is %16.12f\n", lambda);
50       printf("The eigenvector is:\n");
51       for (i=1;i<=n;i++)
52         printf("x[%3i]=%16.12f\n",i,x[i]);
53     }
54     else
55       printf("The eigenvalue could not be determined\n");
56     free(x);
57   }
58   free(a[0]); free(a);
59 }
```

The program `main.c` is nearly identical to the one used on account of the power method. One difference is that in lines 23–24 the parameter `s` is read from the input file `coeffs` and then it is printed out; in the present file, the name of the character string declared on line 30 was changes to `ss` to avoid conflict (in the file `main.c` associated with the power method, this character string was called `s`). On line 45 the function `inv_power` is called (rather than `power` power method earlier). With the above input file, the output of the program is the following:

```
 1 Approximate eigenvalue used:  9.20000
 2 Tolerance used: 5e-14
 3 15 iterations were used to find the largest eigenvalue.
 4 The eigenvalue is   8.310753728109
 5 The eigenvector is:
 6 x[  1]=  0.829693363275
 7 x[  2]=  1.000000000000
 8 x[  3]=  0.253187383172
 9 x[  4]=  0.478388026222
10 x[  5]=  0.457090784061
```

41. WIELANDT'S DEFLATION

In Wielandt's deflation, after determining the eigenvalue and the corresponding eigenvector of a square matrix, the order of the matrix is reduced, and then one can look for the remaining eigenvalues. To explain how this work, let A be a square matrix, and assume that the column vector $\mathbf{x}$ is an eigenvector of A with eigenvalue λ; that is,

$$A\mathbf{x} = \lambda\mathbf{x}.$$

One then looks for a column vector $\mathbf{z}$ such that

$$\mathbf{z}^T\mathbf{x} = 1,$$

and then forms the matrix

$$B = A - \lambda \mathbf{x}\mathbf{z}^T.$$

One may want to keep in mind that if A is an $n \times n$ matrix and $\mathbf{z}$, $\mathbf{x}$ are $n \times 1$ column vectors, then $\mathbf{z}^T\mathbf{x}$ is a number, and $\mathbf{x}\mathbf{z}^T$ is an $n \times n$ matrix.

One can make the following observations. First, 0 is an eigenvalue of B, with eigenvector $\mathbf{x}$. Indeed,

$$B\mathbf{x} = A\mathbf{x} - \lambda(\mathbf{x}\mathbf{z}^T)\mathbf{x} = \lambda\mathbf{x} - \lambda\mathbf{x}(\mathbf{z}^T\mathbf{x}) = \lambda\mathbf{x} - \lambda\mathbf{x} = 0.$$

Second, if $\rho \neq 0$ is an eigenvalue of A different from λ then ρ is also an eigenvalue of B. In fact, if

$$A\mathbf{y} = \rho\mathbf{y}$$

for some nonzero vector $\mathbf{y}$, then we will show that

$$B\mathbf{w} = \rho\mathbf{w}$$

with

$$\mathbf{w} = \rho\mathbf{y} - \lambda(\mathbf{z}^T\mathbf{y})\mathbf{x}.$$

Note that $\mathbf{w} \neq 0$. Indeed, as the $\rho \neq \lambda$, the eigenvectors $\mathbf{y}$ and $\mathbf{x}$ of A corresponding to the eigenvalues ρ and λ must be linearly independent. $\mathbf{w}$ being one of their linear combinations, we must have $\mathbf{w} \neq 0$. We have

$$\begin{aligned}
B\mathbf{w} &= A(\rho\mathbf{y} - \lambda(\mathbf{z}^T\mathbf{y})\mathbf{x}) - \lambda(\mathbf{x}\mathbf{z}^T)(\rho\mathbf{y} - \lambda(\mathbf{z}^T\mathbf{y})\mathbf{x}) \\
&= \rho^2\mathbf{y} - \lambda^2(\mathbf{z}^T\mathbf{y})\mathbf{x} - \lambda\rho(\mathbf{x}\mathbf{z}^T)\mathbf{y} + \lambda^2(\mathbf{z}^T\mathbf{y})(\mathbf{x}\mathbf{z}^T)\mathbf{x} \\
&= \rho^2\mathbf{y} - \lambda^2(\mathbf{z}^T\mathbf{y})\mathbf{x} - \lambda\rho(\mathbf{x}\mathbf{z}^T)\mathbf{y} + \lambda^2(\mathbf{z}^T\mathbf{y})\mathbf{x}(\mathbf{z}^T\mathbf{x}) \\
&= \rho^2\mathbf{y} - \lambda^2(\mathbf{z}^T\mathbf{y})\mathbf{x} - \lambda\rho(\mathbf{x}\mathbf{z}^T)\mathbf{y} + \lambda^2(\mathbf{z}^T\mathbf{y})\mathbf{x} = \rho^2\mathbf{y} - \lambda\rho(\mathbf{x}\mathbf{z}^T)\mathbf{y} \\
&= \rho^2\mathbf{y} - \lambda\rho\mathbf{x}(\mathbf{z}^T\mathbf{y}) = \rho\mathbf{w}.
\end{aligned}$$

In the last term of the second member of these equations, the matrix $\mathbf{x}\mathbf{z}^T$ and the scalar $\mathbf{z}^T\mathbf{y}$ were interchanged to obtain the last term of the third member. Then the associative rule for matrix products was used to obtain the last term of the fourth member. Next, the equality $\mathbf{z}^T\mathbf{x} = 1$ was used to obtain the fifth member of these equations. To obtain the seventh member, the associative rule was used. To obtain the last equation, one needs to note that the vector $\mathbf{x}$ and scalar $\mathbf{x}^T\mathbf{y}$ commute.[154]

Finally, we have

$$\mathbf{z}^T\mathbf{w} = \rho(\mathbf{z}^T\mathbf{y}) - \lambda(\mathbf{z}^T\mathbf{y})(\mathbf{z}^T\mathbf{x}) = \rho(\mathbf{z}^T\mathbf{y}) - \lambda(\mathbf{z}^T\mathbf{y}) = (\rho - \lambda)(\mathbf{z}^T\mathbf{y}).$$

Hence, assuming that $\rho \neq \lambda$, we have

$$\mathbf{z}^T\mathbf{y} = \frac{1}{\rho - \lambda}(\mathbf{z}^T\mathbf{w}).$$

Thus, if we also have $\rho \neq 0$, then

$$\mathbf{y} = \frac{1}{\rho}(\mathbf{w} + \lambda(\mathbf{z}^T\mathbf{y})\mathbf{x}) = \frac{1}{\rho}\left(\mathbf{w} + \frac{\lambda}{\rho - \lambda}(\mathbf{z}^T\mathbf{w})\mathbf{x}\right). \tag{1}$$

[154]The assumption $\rho \neq \lambda$ was only used to show that $\mathbf{x}$ and $\mathbf{y}$ were linearly independent, and this was needed to ensure that $\mathbf{w} \neq 0$. That is, the argument works even if $\rho = \lambda \neq 0$ and $\mathbf{x}$ and $\mathbf{y}$ are linearly independent. That is, if λ is a multiple eigenvalue with at least two linearly independent eigenvectors, then λ will still be an eigenvalue of B.

Note: If $\rho = \lambda$ and $\mathbf{y} = c\mathbf{x}$ then $\mathbf{w} = \rho\mathbf{y} - \lambda(\mathbf{z}^T\mathbf{y})\mathbf{x} = \rho\mathbf{y} - \lambda(\mathbf{z}^T\mathbf{x})\mathbf{y} = 0$, so we cannot allow $\mathbf{x}$ and $\mathbf{y}$ to be linearly dependent.

This equation expresses the eigenvector $\mathbf{y}$ associated with the eigenvalue ρ of A in terms of the eigenvector $\mathbf{w}$ of B belonging to the same eigenvalue.

In one step of Wielandt deflation, the order of an $n \times n$ matrix $A = (a_{ij})$ is reduced by one. First, one finds a nonzero eigenvalue λ and the corresponding eigenvector $\mathbf{x} = (x_1, \ldots, x_n)^T$. Then one finds the component x_r of $\mathbf{x}$ of maximum absolute value; this will guarantee that $x_r \neq 0$, since $\mathbf{x}$, being an eigenvector, cannot be a zero vector. Then one picks

$$\mathbf{z} = \frac{1}{\lambda x_r}(a_{r1}, a_{r2}, \ldots, a_{rn})^T$$

Then $\mathbf{z}^T\mathbf{x}$ is the rth component of the vector

$$\frac{1}{\lambda x_r} A\mathbf{x} = \frac{1}{\lambda x_r}\lambda\mathbf{x} = \frac{1}{x_r}\mathbf{x}.$$

The second equation holds because $\mathbf{x}$ is an eigenvector of A with eigenvalue λ. The rth component of this vector is $\frac{1}{x_r}x_r = 1$. Thus, the equation $\mathbf{z}^T\mathbf{x} = 1$ is satisfied, as was required above.

The entry b_{ij} of the matrix $B = A - \lambda\mathbf{x}\mathbf{z}^T$ can be expressed as

$$b_{ij} = a_{ij} - \lambda x_i \cdot \frac{1}{\lambda x_r} a_{rj} = a_{ij} - \frac{1}{x_r} x_i a_{rj}. \tag{2}$$

It can be seen that all elements in the rth row are zero:

$$b_{rj} = a_{rj} - \frac{1}{x_r} x_r a_{rj} = 0$$

Now, we are looking for another eigenvalue $\rho \neq 0$ of the matrix A eigenvector y. As mentioned above, then the vector $\mathbf{w} = \rho\mathbf{y} - \lambda(\mathbf{z}^T\mathbf{y})\mathbf{x}$ is an eigenvector of B with eigenvalue ρ. Write $\mathbf{w} = (w_1, \ldots, w_n)^T$. The rth component of the vector $\rho\mathbf{w} = B\mathbf{w}$ is

$$\rho w_r = \sum_{j=1}^{n} b_{rj} w_j = 0,$$

where the last equation holds because $b_{rj} = 0$ for each j. As $\rho \neq 0$, it follows that $w_r = 0$.

It follows that the rth *column* of the matrix B plays does not enter into the calculation of the eigenvalue ρ and the corresponding eigenvector. Indeed, the equation $\rho\mathbf{w} = B\mathbf{w}$ can be written componentwise as

$$\rho w_i = \sum_{j=1}^{n} b_{ij} w_j = \sum_{\substack{j=1 \\ j \neq r}}^{n} b_{ij} w_j,$$

where the second equality holds because $w_r = 0$. Therefore, if one deletes the rth row and rth column of the matrix B to obtain B', and if one deletes the rth component of the vector $\mathbf{w}$ to obtain $\mathbf{w}'$, one will have $B'\mathbf{w}' = \rho\mathbf{w}'$. Hence, one can calculate the eigenvalues and eigenvectors of B' to find ρ and $\mathbf{w}'$. Having found $\mathbf{w}'$, and can find $\mathbf{w}$ by restoring the deleted zero component. Finally, one can calculate the corresponding eigenvector of $\mathbf{y}$ of A by using equation (1).

Thus, having found one eigenvalue λ and a corresponding eigenvector $\mathbf{x}$ of the matrix A, one can reduce the order of the matrix A by one, that is, one can *deflate* the matrix, and look for the remaining eigenvalues of a lower order matrix. In this way, one can find all the eigenvalues one by one. Using equation (1) to find the corresponding eigenvector is, however, probably not worth doing. This is because the eigenvalues of the deflated matrix should only be considered approximations of those of the original matrix A, since successive deflations introduce roundoff errors. One can use the

inverse power method with the original matrix to obtain better approximations of the eigenvalues; if $\bar{\rho}$ is the calculated approximation of the eigenvalue ρ of A, then one would use the inverse power method to calculate the largest eigenvalue of the matrix $(A-\bar{\rho}I)^{-1}$. Because of roundoff errors, we will have $\bar{\rho} \neq \rho$, and this matrix will not be singular (so the inverse can be calculated). The fact that the matrix $A-\bar{\rho}I$ is nearly singular (and so its inverse cannot be calculated very accurately) will not adversely affect the calculation of the eigenvalue. Since the inverse power method also calculates the eigenvector of the original matrix, there is little reason to keep track of eigenvectors during deflation.

In the computer implementation of Wielandt's deflation, the header file `wielandt.h` is used:

```
 1 #include <stdio.h>
 2 #include <math.h>
 3 #include <stdlib.h>
 4
 5 #define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
 6
 7 double **allocmatrix(int n);
 8 double *allocvector(int n);
 9 /* int inv_power(double **a, double *x, int n, float s,
10         double *lambda, float tol, int maxits, int *success); */
11 int power_maxel(double **a, double *x, int n, double *lambda,
12         int *r, float tol, int maxits, int *success);
13 void wielandt(double **a, double *x, int n, int k);
14 int wielandt_defl(double **a, double *x, int n,
15         double *lambda, float tol, int maxits);
```

Here the function `inv_power` is commented out on lines 9–10, since it is not used in the present implementation. As mentioned above, after calculating the eigenvalues by deflation, one should use the inverse power method with the original matrix to improve the accuracy of the solution; for the sake of simplifying the discussion, we omitted this step. The file `power_maxel.c` contains a slightly modified version of the power method:

```
 1 #include "wielandt.h"
 2
 3 int power_maxel(double **a, double *x, int n, double *lambda,
 4         int *r, float tol, int maxits, int *success)
 5 /* In this variation of the power method, the subscript
 6    of the maximal element of the eigenvector is returned
 7    as r */
 8 {
 9   int i, j, itcount;
10   double m, max, newx, *y, *u;
11   u=allocvector(n);
12   for (i=1;i<=n;i++) x[i]=1.0;
13   for (itcount=1;itcount<=maxits;itcount++) {
14     /* Form y=Ax and record maximum element max */
15     max=0.;
16     for (i=1;i<=n;i++) {
17       m=0.;
18       for (j=1;j<=n;j++) m += a[i][j]*x[j];
19       if ( absval(m)>absval(max) ) max=m;
20       u[i]=m;
21     }
22     /* Normalize u and test for convergence */
```

```
23      *success=1;
24      for (i=1;i<=n;i++) {
25        newx=u[i]/max;
26        if ( absval(newx-x[i])>tol ) *success=0;
27        x[i]=newx;
28      }
29      if ( *success ) {
30        *lambda=max;
31        /* In principle, we would want to find the value of r
32           for which x[r] has the largest absolute value. However,
33           we know that x[r] for this r has value 1; so it
34           is easier to look for x[r] with the largest value.
35           The calling program will assume x[r]==1. If we were
36           looking for x[r] with the largest absolute value,
37           it might happen that we find another component
38           with x[r]==-1; theoretically, this should not happen,
39           but in practice it might happen because of rounding
40           errors.                                              */
41        *r=1; max=x[1];
42        for (i=1;i<=n;i++)
43          if ( x[i]>max) {
44            *r=i;
45            max=x[i];
46          }
47        break;
48      }
49    }
50    free(u);
51    return itcount;
52  }
```

The function `power_maxel` has an additional parameter `*r` compared to the function `power` discussed earlier. In lines 41–46, the subscript r of the component x_r of the largest absolute value of the eigenvector $\mathbf{x}$ is calculated, in effect. As explained in the comment in lines 31–40, in actual fact, the component of the largest value is calculated instead. This allows us to be certain that `x[r]==1`, which allows us make this assumption in the calling program. The value `*r` of this subscript is returned to the calling program; as was pointed out in the theoretical discussion above, this value of r is used in the deflation. Aside from this, the function `power_maxel` is identical to the function `power` discussed earlier, on account of the power method. The deflation itself is performed by the functions in the file `wielandt.c`:

```
 1 #include "wielandt.h"
 2
 3 /* It is important to preserve the original row
 4    pointers for **a so the matrix could be freed */
 5
 6 void wielandt(double **a, double *x, int n, int r)
 7 {
 8   int i, j;
 9   for (i=1;i<=n;i++)
10     if ( i!=r )
11       for (j=1;j<=n;j++) a[i][j] -= x[i]*a[r][j];
```

```
12     for (i=r;i<n;i++) a[i]=a[i+1];
13     for (i=1;i<n;i++)
14       for (j=r;j<n;j++) a[i][j]=a[i][j+1];
15   return;
16 }
17
18 int wielandt_defl(double **a, double *x, int n,
19         double *lambda, float tol, int maxits)
20 {
21   int m, r, success, eigencount, itcount;
22   for (m=n;m>1;m--) {
23     eigencount=n-m;
24     itcount=power_maxel(a,x,m,&lambda[eigencount+1],&r,
25        tol,maxits,&success);
26     printf("%3u iterations used for eigenvalue number %u\n",
27           itcount, eigencount+1);
28     if ( !success ) break;
29     wielandt(a,x,m,r);
30   }
31   if ( success && m==1 ) {
32     eigencount=n;
33     lambda[n]=a[1][1];
34   }
35   return eigencount;
36 }
```

The function `wielandt` in lines 6–16 has as parameters the matrix `**a`, the vector `*x` containing the eigenvector found by the calling program, the size `n` of the matrix, and the subscript `r` of the row and column to be deleted. On line 11, formula (2) is used to calculate the components of the matrix B and stored at the same location where the matrix A was stored. Note that the eigenvector `*x` was obtained by the power method (through the function call on line 24 below); the power method ensures that the component `x[r]` of the largest absolute value of `x` is 1; this is why there is no need to divide by x_r on line 11, unlike in formula (2).[155] In lines 13–15, the rth row and rth column of the matrix B is deleted.

The function `wielandt` described in lines 18–36 returns the number of the eigenvalues that were successfully calculated. Its parameters are the matrix `**a`, the vector `*x` that will contain the various eigenvectors calculated, but no data will be returned in `*x` to the calling program. Its role is only to reserve memory only once, to be used to store various vectors in the course of the calculation. The remaining parameters are the size `n` of the matrix A, the vector `*lambda` that will contain the eigenvalues being calculated, the tolerance (or permissible error) `tol`, and the maximum number of iterations `maxit` to be used in calculating each eigenvalue. The integer `m` will contain the current size of the deflated matrix. The loop in lines 22–30 starts with $m = n$, and the value of m will be decreased by 1 with each deflation step. In lines 24–25, the function `power_maxel` is called to calculate the largest eigenvalue of the current matrix; as mentioned in the description of this function, the parameter `r` will contain the subscript of the largest component of the eigenvector `x`. On line 26, the number of iterations used to calculate the current eigenvalue is printed out; alternatively, this value could be returned as the component of a vector to the calling program, and the calling program could decide whether to print out this value of to discard it. If the calculation of

[155] This was the reason that in the file `power_maxel.c` we were looking for the largest x_r rather than the x_r with the largest absolute value in lines 41–43. If there is an $r' < r$ for which $x_{r'}$ is rounded to -1, r' would be returned instead of r, even though without rounding errors we should have $|x'_r| < x_r = 1$. With $x_r = -1$, the calculation on line 11 would not work.

the eigenvalue was unsuccessful in lines 24–25, the calculation is abandoned on line 28 by breaking out of the loop. Otherwise, the function `wielandt` is called to carry out one step of deflation, and the body of the loop is executed again unless `m==1`. If everything was successful, in line 33 the last eigenvalue is calculated. At this point, the matrix has order 1×1, and the eigenvalue of this matrix is the only matrix element. On line 35, the number is returned of the eigenvalues successfully calculated.

This program was used with a symmetric matrix, since it is guaranteed that all eigenvalues of a symmetric matrix is real; a complex eigenvalue might interfere with the power method used during the deflation process. The file `symm_coeffs` with the following content was used:

```
1 5e-14
2 5
3 1  2  3  5  8
4    3 -2  2  3
5       1  1 -1
6         -1  3
7             2
```

The first column of numbers are line numbers, not part of the file. The number on line 1 is the tolerance, that on line 2 is the size of the matrix. In lines 3–7, the entries in the upper triangle of the matrix are given; since the matrix is assumed to be symmetric, there is no need to store the other entries. The calling program is given in the file `main.c`:

```
 1 #include "wielandt.h"
 2
 3 main()
 4 {
 5   /* This program reads in the elements of in the
 6      upper triangle of a square matrix int the file
 7      named symm_coeffs, and fills in the lower triangle
 8      to make the matrix symmetric. The first entry in
 9      the file is the required tolerance, the
10      second entry is n, the size of the matrix. The
11      rest of the entries are the elements in the
12      upper triangle of the matrix. The first entry
13      must be of type float, the third entry must be
14      an unsigned integer, the other entries can be
15      integers or reals.                          */
16   double **a, *x, *lambda;
17   float s, tol;
18   int n, i, j, subs, readerror=0, success, eigencount;
19   char ss[25];
20   FILE *coefffile;
21   coefffile=fopen("symm_coeffs", "r");
22   fscanf(coefffile, "%f", &tol);
23   printf("Tolerance used: %g\n", tol);
24   fscanf(coefffile, "%u", &n);
25   a=allocmatrix(n);
26   for (i=1;i<=n;i++) {
27     if ( readerror ) break;
28     for (j=i;j<=n;j++) {
29       if (fscanf(coefffile, "%s", ss)==EOF) {
30         readerror=1;
```

```
31          printf("Not enough coefficients\n");
32          printf("i=%u j=%u\n", i, j);
33          break;
34        }
35        a[j][i]=a[i][j]=strtod(ss,NULL);
36      }
37    }
38    fclose(coefffile);
39    if ( readerror ) printf("Not enough data\n");
40    else {
41      x=allocvector(n);
42      lambda=allocvector(n);
43  eigencount=wielandt_defl(a,x,n,lambda,tol,150);
44      if ( eigencount==0 )
45        printf("No eigenvalues could be calculated\n");
46      else if ( eigencount==1 )
47        printf("Only one eigenvalue could be calculated. It is:\n");
48      else if ( eigencount<n )
49        printf("Only %u eigenvalues could be calculated. They are:\n",
50                eigencount);
51      else
52        printf("All eigenvalues were calculated. They are:\n");
53      for (i=1;i<=eigencount;i++) {
54        printf("%16.12f ", lambda[i]);
55        if ( i % 4 == 0 ) printf("\n");
56      }
57      printf("\n");
58      free(lambda);
59      free(x);
60    }
61    free(a[0]); free(a);
62  }
```

All coefficients of the matrix A are stored. This is necessary, since even if the starting matrix is symmetric, the matrices obtained during the deflation process will not be symmetric (but they will all have only real eigenvalues). The coefficients of the matrix are read in the loop in lines 26–37; much of what is being done here is similar to the way earlier matrices have been read, except that only the upper triangle of the matrix is read, and the missing elements of the matrix are filled in on line 35. As in earlier programs, if there is no reading error (that is, there were enough numbers in the input file), in lines 40–60, the calculation of eigenvalues is performed. In lines 41–42, memory for the vectors pointed to by `x` and `lambda` is reserved, and this memory is freed in lines 58–59, inside the same `else` statement. Wielandt deflation is called on line 43, with at most 150 iterations permitted for calculating each eigenvalue. The printout of the program was as follows:

```
1 Tolerance used: 5e-14
2  56 iterations used for eigenvalue number 1
3  48 iterations used for eigenvalue number 2
4 139 iterations used for eigenvalue number 3
5  20 iterations used for eigenvalue number 4
6 All eigenvalues were calculated. They are:
7  13.583574215424  -7.852544184194   4.256975565094  -3.363332212396
8  -0.624673383928
```

Problems

1. Consider the matrix

$$A = \begin{pmatrix} 14 & 8 & 10 \\ 8 & 22 & 14 \\ 4 & 8 & 8 \end{pmatrix}.$$

An eigenvalue of this matrix is 2 with eigenvector $\mathbf{x} = (-1, -1, 2)^T$. Do one step of Wielandt deflation.

Solution. We have $\lambda = 2$, the rth component with $r = 3$ has the largest absolutle value of the eigenvector $\mathbf{x}$, $x_r = x_3 = 2$, and the rth, i.e., third, row of A is $a_{3*} = (4, 8, 8)$. Hence

$$\mathbf{z} = \frac{1}{\lambda x_3} a_{3*}^T = \frac{1}{2 \cdot 2}(4, 8, 8)^T = \frac{1}{2}(2, 4, 4)^T;$$

here the row vector $(4, 8, 8)$ is the third row of the matrix A. The third row is used since the third component of the eigenvector $\mathbf{x}$ has the largest absolute value.

The purpose of choosing the component of the largest absolute value of the eigenvector is to make the roundoff error the smallest possible. The reason this results in the smallest roundoff error is that this is likely to make the entries of the subtracted matrix $\mathbf{x} \cdot \mathbf{z}^T$ the smallest possible (since we divide by x_r, the largest component of $\mathbf{x}$, in calculating $\mathbf{z}$). The larger the components of the matrix that we are subtracting from A, the more the original values of the entries of A will perturbed by roundoff errors. This is especially important if one performs repeated deflations, since the roundoff errors then accumulate. When doing exact calculations with integers, this point may be moot, since in this case there are no roundoff errors.

We have

$$B = A - \lambda \cdot \mathbf{x} \cdot \mathbf{z}^T = A - 2 \cdot \mathbf{x} \cdot \mathbf{z}^T = \begin{pmatrix} 14 & 8 & 10 \\ 8 & 22 & 14 \\ 4 & 8 & 8 \end{pmatrix} - \begin{pmatrix} -1 \\ -1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 & 4 & 4 \end{pmatrix}$$
$$= \begin{pmatrix} 14 & 8 & 10 \\ 8 & 22 & 14 \\ 4 & 8 & 8 \end{pmatrix} - \begin{pmatrix} -2 & -4 & -4 \\ -2 & -4 & -4 \\ 4 & 8 & 8 \end{pmatrix} = \begin{pmatrix} 16 & 12 & 14 \\ 10 & 26 & 18 \\ 0 & 0 & 0 \end{pmatrix}.$$

When looking for the eigenvalues of the matrix B (except for the eigenvalue 0, which is not an eigenvalue of the original matrix A), one can delete the third row and third column of the matrix B, and look for the eigenvalues of

$$B' = \begin{pmatrix} 16 & 12 \\ 10 & 26 \end{pmatrix}.$$

2. Consider the matrix

$$A = \begin{pmatrix} 36 & 54 & 12 \\ 30 & 48 & 36 \\ 36 & 72 & 36 \end{pmatrix}.$$

An eigenvalue of this matrix is 12 with eigenvector $\mathbf{x} = (6, -2, -3)^T$. Do one step of Wielandt deflation.

Solution. We have $\lambda = 12$, the rth component with $r = 1$ has the largest absolutle value of the eigenvector $\mathbf{x}$, $x_r = x_1 = 6$, and the rth, i.e., first, row of A is $a_{1*} = (36, 54, 12)$. Hence

$$\mathbf{z} = \frac{1}{\lambda x_1} a_{1*}^T = \frac{1}{12 \cdot 6}(36, 54, 12)^T = \frac{1}{12}(6, 9, 2)^T,$$

we have

$$B = A - \lambda \cdot \mathbf{x} \cdot \mathbf{z}^T = A - 12 \cdot \mathbf{x} \cdot \mathbf{z}^T = \begin{pmatrix} 36 & 54 & 12 \\ 30 & 48 & 36 \\ 36 & 72 & 36 \end{pmatrix} - \begin{pmatrix} 6 \\ -2 \\ -3 \end{pmatrix} \begin{pmatrix} 6 & 9 & 2 \end{pmatrix}$$
$$= \begin{pmatrix} 36 & 54 & 12 \\ 30 & 48 & 36 \\ 36 & 72 & 36 \end{pmatrix} - \begin{pmatrix} 36 & 54 & 12 \\ -12 & -18 & -4 \\ -18 & -27 & -6 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 42 & 66 & 40 \\ 54 & 99 & 42 \end{pmatrix}.$$

When looking for the eigenvalues of the matrix B (except for the eigenvalue 0, which is not an eigenvalue of the original matrix A), one can delete the first row and first column of the matrix B, and look for the eigenvalues of

$$B' = \begin{pmatrix} 66 & 40 \\ 99 & 42 \end{pmatrix}.$$

42. SIMILARITY TRANSFORMATIONS AND THE QR ALGORITHM

Let A and S be $n \times n$ matrices and assume that S is nonsingular. The matrix $S^{-1}AS$ is said to be *similar* to A, and the the transformation that takes A to $S^{-1}AS$ is called a *similarity transformation*.[156] The importance of similarity transformations for us is that A and $S^{-1}AS$ have the same eigenvalues. In fact, we have

$$A\mathbf{v} = \lambda\mathbf{v}$$

if and only if

$$S^{-1}AS(S^{-1}\mathbf{v}) = \lambda(S^{-1}\mathbf{v}),$$

as is easily seen by multiplying the first of these equations by S^{-1} on the left.

An *upper Hessenberg* matrix is a square matrix where all elements below the main diagonal, but not immediately adjacent to it, are zero; that is, $A = (a_{ij})$ is an upper Hessenberg matrix if $a_{ij} = 0$ whenever $i > j + 1$. There are methods to find the eigenvalues of a Hessenberg matrix. Hence, as a first step in finding the eigenvalues of a matrix A may be to transform it to a Hessenberg matrix. In fact, there are various methods to find a matrix $S^{-1}AS$ similar to A that is an upper Hessenberg matrix. One method performs Gaussian elimination. One can do this by performing the following steps.

One goes through the columns of the matrix A, beginning with the first column. When dealing with the kth column ($1 \le k \le n-2$), one makes the elements $a_{k+2\,k}, \ldots, a_{nk}$ in this column zero. If these elements are already zero then nothing is done, and one goes on to deal with column $k+1$. If not, then one finds the element of largest absolute value among the elements $a_{k+1\,k}, \ldots, a_{nk}$. If this is $a_{i'k}$ then one interchanges row $k+1$ and row i'. To make this a similarity transformation, one then interchanges column $k+1$ and column i'. (We will give some explanation of this point below.)

At this point, the element that became $a_{k+1,k}$ is nonzero. For $i = k+2, \ldots, n$, one then subtracts $m_i \stackrel{def}{=} a_{ik}/a_{k+1\,k}$ times row $k+1$ from row i. To make this a similarity transformation, one then adds m_i times column i to column $k+1$.

[156] A matrix can be thought of as the description of a linear transformation of a finite dimensional vector space in terms of how the basis vectors are transformed. Namely, if T is a linear transformation and $\mathbf{e}_1, \ldots, \mathbf{e}_n$ are basis vectors, then the matrix $A = (a_{ij})$ such that

$$T(\mathbf{e}_i) = \sum_{i=1}^{n} a_{ij}\mathbf{e}_i.$$

A similarity transformation corresponds to a change of basis.

To explain why changing the matrix A in this way is a similarity transformation, consider interchanging row $k+1$ and row i'. This corresponds to multiplying A on the left by a permutation matrix $P = (p_{ij})$ where $p_{ii} = 1$ if $i \neq k+1$ and $i \neq i'$, and $p_{k+1\,i'} = 1$ and $p_{i'\,k+1} = 1$ and $p_{ij} = 0$ otherwise. The inverse of the matrix P is P itself.[157] Multiplying A by P on the right amounts to interchanging columns $k+1$ and i' of A. That is, first interchanging the rows and the the columns as stated amounts to forming the matrix $PAP = P^{-1}AP$.

Let k and l be arbitrary with $1 \leq k, l \leq n$ with $k \neq l$, and let c be a number. Subtracting ρ times row k from row l in the matrix A amounts to multiplying the matrix A on the left by the matrix $C = (c_{ij})$, where[158]

$$c_{ij} = \delta_{ij} - \rho\delta_{il}\delta_{kj}.$$

Adding ρ times column l to column k in the matrix A amounts to multiplying A on the right by $D = (d_{ij})$, where

$$d_{ij} = \delta_{ij} + \rho\delta_{kj}\delta_{il}.$$

It is easy to see that $C = D^{-1}$. Indeed,[159]

$$\begin{aligned}\sum_{r=1}^{n} c_{ir}d_{rj} &= \sum_{r=1}^{n}(\delta_{ir}\delta_{rj} + \rho\delta_{ir}\delta_{kj}\delta_{rl} - \rho\delta_{il}\delta_{kr}\delta_{rj} + \rho^2\delta_{il}\delta_{kr}\delta_{kj}\delta_{rl}) \\ &= \delta_{ij} + \rho\delta_{il}\delta_{kj} - \rho\delta_{il}\delta_{kj} + \rho^2\delta_{il}\delta_{kl}\delta_{kj} = \delta_{ij};\end{aligned}$$

the last equation holds because $\delta_{kl} = 0$ since we assumed $k \neq l$.

Plain rotations will be used to transform a Hessenberg matrix to an upper triangular matrix. Given p and q with $1 \leq p, q \leq n$ and $p \neq q$, a plain rotation by angle θ involving row p and q is a transformation such that

$$a'_{pj} = a_{pj}\cos\theta - a_{qj}\sin\theta,$$

$$a'_{qj} = a_{pj}\sin\theta + a_{qj}\cos\theta,$$

and

$$a'_{ij} = a_{ij} \qquad \text{if } i \neq p \text{ and } i \neq q.$$

The reason this is called a plain rotation is because if one describes the components of a vector with unit vectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$, and then one takes the plain of the vectors $\mathbf{e}_p$ and $\mathbf{e}_q$, rotates the unit vectors in this plain by an angle θ to obtain the new unit vectors $\mathbf{e}'_p$ and $\mathbf{e}'_q$, (while leaving all other unit vectors unchanged, the above formulas describe the vector (namely, the jth column vector of the matrix) in the new coordinate system. The way the matrix $A' = (a'_{ij})$ is obtained from the matrix $A = (a_{ij})$ amounts to multiplying A on the left by the matrix $S = s_{ij}$ where

$$s_{pp} = \cos\theta, \quad s_{pq} = -\sin\theta, \quad s_{qp} = \sin\theta, \quad s_{qq} = \cos\theta, \quad \text{and} \quad s_{ij} = \delta_{ij} \text{ otherwise.}$$

The matrix S is clearly an orthogonal matrix; hence its inverse is its transpose. The entries a''_{ij} of the matrix $A'' = AS^T$ can be obtained by the formulas

$$a''_{ip} = a'_{ip}\cos\theta - a'_{iq}\sin\theta,$$

$$a''_{iq} = a'_{ip}\sin\theta + a'_{iq}\cos\theta,$$

[157] In terms of permutations this means that if one interchanges row $k+1$ and row i', and then one again interchanges row $k+1$ and row i', then one gets back the original matrix.

[158] δ_{ij}, called Kronecker's delta, is defined to be 1 is $i = j$ and 0 if $i \neq j$.

[159] Instead of the following calculation, one might note that multiplying the matrix CA on the left by D amounts to adding ρ times row k to row l of the matrix CA; the resulting matrix is clearly the original matrix A; thus $DCA = A$; since this is true for an arbitrary matrix A, it follows that CD is the identity matrix.

and

$$a''_{ij} = a'_{ij} \qquad \text{if } j \neq p \text{ and } j \neq q.$$

The transformation SAS^T is a similarity transformation, since $S = (S^T)^{-1}$.

Assuming that A is an upper Hessenberg matrix, one uses plain rotations to make the elements under the main diagonal zero, so as to transform A into an upper triangular matrix. When wanting to change the element a_{qp} to zero (where $q = p + 1$ in our case), one uses a plain rotation S that changes the element a_{qp} to zero. Using a plain rotation with rows p and q with angle θ, we have

$$a'_{qp} = a_{pp} \sin\theta + a_{qp} \cos\theta.$$

In order to have $a'_{qp} = 0$, the angle θ needs to be determined so that, with

$$\alpha = \frac{1}{\sqrt{a_{pp}^2 + a_{qp}^2}},$$

we have

$$\sin\theta = -a_{qp}\alpha \quad \text{and} \quad \cos\theta = a_{pp}\alpha.$$

For $p = 1, \ldots, p = n - 1$, one multiplies A from the left by such a matrix S_p involving rows p and $p + 1$ to change the element $a_{p+1\,p}$ to zero. That is, one forms the matrix

$$S_{n-1} S_{n-2} \ldots S_1 A.$$

Since one wants to perform a similarity transformation so as not to change the eigenvalues, one then needs to multiply on the right with the inverses (or, what amounts to the same, transposes) of these matrices, to obtain the matrix

$$A_1 = S_{n-1} \ldots S_1 A S_1^T \ldots S_{n-1}^T. \tag{1}$$

This matrix will not be a triangular matrix, since multiplication from the right will change the zero elements $a_{p+1\,p}$ next to the main diagonal back to nonzero elements. Nevertheless, assuming that all eigenvalues of A are real, by repeatedly performing this transformation (i.e., again changing the elements just below the main diagonal to zero), one obtains a sequence of matrices that usually converges to a triangular matrix.

One can speed up this convergence by applying shifts. That is, instead of starting with the Hessenberg matrix A, one forms the matrix $A - sI$ with an appropriate value of s, then forms the matrix

$$A'_1 = S_{n-1} \ldots S_1 (A - sI) S_1^T \ldots S_{n-1}^T,$$

and then applies the shift $A_1 = A'_1 + sI$. Of course, equation (1) is then also valid, since the shifts $-sI$ and $+sI$ will cancel (since the matrices S_i and S_i^T commute with the identity matrix I), but the shift makes a difference in determining the appropriate matrices S_i. The shift that needs to be applied can be determined the lower right 2 by 2 minor of the matrix A. If λ_1 and λ_2 are the eigenvalues of this minor, then one chooses s as the eigenvalue λ_i for which $|\lambda_i - a_{nn}|$ is smaller in case λ_1 and λ_2 are real, and the real part of λ_1 (or λ_2) if these eigenvalues are complex (since they are complex conjugates of each other, assuming that we are dealing with a real matrix).

After obtaining the matrix A_1, one obtains the matrices $A_2, \ldots$, until in the matrix A_k, the element $a^{(k)}_{n-1\,n}$ is close enough to zero; in this case, an eigenvalue of A_k will be (close to) the element $a^{(k)}_{nn}$. Having determined this eigenvalue, the matrix A_k can be deflated by crossing out the last row and last column; then one can determine the eigenvalues of the deflated matrix.

When calculating A_1, one can form the matrices $B_1 = S_1 A$, $B_2 = S_2 B_1 S_1^T$, $B_3 = S_3 B_2 S_2^T$, ..., $B_{n-1} = S_{n-1} B_{n-2} S_{n-2}^T$, and $A_1 = B_{n-1} S_{n-1}^T$. This is because multiplying by S_{i-1} on the right does not change the entries of the matrix involved in determining the rotation S_{i+1} (since multiplying on the right by S_{i-1} affects columns $i-1$ and i of the matrix only, and to determine S_{i+1} one needs only the current values of the entries $a_{i+1\,i+1}$ and $a_{i+2\,i+1}$).[160]

To implement this method on computer, one used the header file `qr.h`:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
#define square(x) ((x)*(x))

void simgauss(double **a,int n);
double shiftamount(double **a, int n);
int protation(double **a, int p, int q, int n,
     double *sintheta, double *costheta);
int leftprotate(double **a, int p, int q, int n,
     double sintheta, double costheta);
int rightprotate(double **a, int p, int q, int n,
     double sintheta, double costheta);
double qr(double **a, int n, double tol, int maxits,
      int *itcount, int *success);
double **allocmatrix(int n);
double *allocvector(int n);
void printmatrix(double **a, int n);
void printmatrix_mathematica(double **a, int n);
```

The functions in this file will be explained below. The file `alloc.c` is the same as the one used before:

```
#include "qr.h"

double **allocmatrix(int n)
  /* Allocate matrix. The following code segment allocates
     a contiguous block of pointers for the whole matrix.
     There is no real advantage for this here, because with
     pivoting, rows will be interchanged. So memory for rows
     could be allocated separately. On the other hand, even
     the largest matrix for which Gaussian elimination
     is feasible occupies a relatively modest amount of
     memory, there does not seem to be any disadvantage
     in asking for a contiguous block. */
{
  int i;
  double **a;
  a=(double **) malloc((size_t)((n+1)*sizeof(double*)));
  if (!a) {
```

[160] That is, the matrix B_i still contains the original entries of A needed to determine S_{i+1}. The order of evaluation $C_1 = S_1 A S_1^{-1}$, $C_2 = S_1 A S_2^{-1}$, ..., $C_n = S_1 A S_n^{-1}$, and $A_1 = C_n$ would be less advantageous, since the matrix C_i no longer contains the entries needed to determine S_{i+1}, and so one would need to remember the entries of the original matrix A.

```
18     printf("Allocation failure 1 in matrix\n");
19     exit(1);
20   }
21   /* Allocate the whole matrix: */
22   a[1]=(double *) malloc((size_t)(n*(n+1)*sizeof(double)));
23   if (!a[1]) {
24     printf("Allocation failure 2 in matrix\n");
25     exit(1);
26   }
27   /* Initialize the pointers pointing to each row: */
28   for (i=2; i<=n; i++) a[i]=a[i-1]+n+1;
29   /* Save the beginning of the matrix in a[0]; the
30      rows will be interchanged, the values of a[i]
31      for 1<=i<=n may change:                      */
32   a[0]=a[1];
33   return a;
34 }
35
36 double *allocvector(int n)
37 {
38   double *b;
39   b=(double *) malloc((size_t)((n+1)*sizeof(double)));
40   if (!b) {
41     printf("Allocation failure 1 in vector\n");
42     exit(1);
43   }
44   return b;
45 }
```

This is identical to the file `alloc.c` first used for Gaussian elimination. The only change we made is line 1, where the file `qr.h` is included now. The file `hessenberg.c` contains the function using Gaussian elimination to produce the upper Hessenberg matrix:

```
 1 #include "qr.h"
 2
 3 void simgauss(double **a,int n)
 4 /* The matrices are size n times n */
 5 {
 6   const double near_zero=1e-20;
 7   int i,j,k,l,pivi,success;
 8   double max, temp, *rowptr, norm;
 9   success=1;
10   for(k=1;k<n-1;k++) {
11     pivi=k+1;
12     max=absval(a[k+1][k]);
13     for (i=k+2;i<=n;i++)
14       if( (temp=absval(a[i][k])) > max ) {
15         max=temp;
16         pivi=i;
17       }
18     /* interchange rows and column */
19     if ( pivi != k+1 ) {
```

```
20       /* interchange rows */
21       rowptr=a[k+1]; a[k+1]=a[pivi]; a[pivi]=rowptr;
22       /* interchange columns */
23       for (i=1;i<=n;i++) {
24         temp=a[i][k+1]; a[i][k+1]=a[i][pivi]; a[i][pivi]=temp;
25       }
26     }
27    /* If a[k+1][k] is nearly zero then do nothing, since
28       then column k of the matrix is almost zero already */
29    if (absval(a[k+1][k])>near_zero)
30      for (i=k+2;i<=n;i++) {
31        temp=a[i][k]/a[k+1][k];
32        for (j=k;j<=n;j++) a[i][j] -= temp*a[k+1][j];
33        for (j=1;j<=n;j++) a[j][k+1] += temp*a[j][i];
34      }
35    }
36 }
37
38 double shiftamount(double **a, int n)
39 {
40   double discr, theshift, realpart;
41   realpart=(a[n-1][n-1]+a[n][n])/2.;
42   discr=square(realpart)/4.+
43     a[n-1][n]*a[n][n-1]-a[n-1][n-1]*a[n][n];
44   if ( discr<0 ) return realpart;
45   if ( a[n-1][n-1]>=a[n][n] )
46     return realpart-sqrt(discr);
47   return realpart+sqrt(discr);
48 }
49
50 int protation(double **a, int p, int q, int n,
51      double *sintheta, double *costheta)
52 {
53   const near_zero=1e-20;
54   double alpha;
55   if (p==q) return 0;
56   if ( absval(a[q][p])<=near_zero ) return;
57   alpha=1./sqrt(square(a[p][p])+square(a[q][p]));
58   *sintheta=-alpha*a[q][p];
59   *costheta= alpha*a[p][p];
60   return 1;
61 }
62
63 int leftprotate(double **a, int p, int q, int n,
64      double sintheta, double costheta)
65 {
66   int j;
67   double temp;
68   /* error is p==q */
69   if (p==q) return 0;
70   for (j=1;j<=n;j++) {
```

```
71     temp=a[p][j];
72     a[p][j]=costheta*temp-sintheta*a[q][j];
73     a[q][j]=sintheta*temp+costheta*a[q][j];
74   }
75   return 1;
76 }
77
78 int rightprotate(double **a, int p, int q, int n,
79      double sintheta, double costheta)
80 {
81   int i;
82   double temp;
83   /* error is p==q */
84   if (p==q) return 0;
85   for (i=1;i<=n;i++) {
86     temp=a[i][p];
87     a[i][p]=temp*costheta-a[i][q]*sintheta;
88     a[i][q]=temp*sintheta+a[i][q]*costheta;
89   }
90   return 1;
91 }
```

The function `simgauss` in lines 3–36 has the matrix `**a` and its size `n` as parameters. Dealing with columns $k = 1$ through $k = n - 2$ in the loop in lines 10–35, the pivot element is found in lines 11–18, and then the rows and columns are interchanged in lines 19–26; as we mentioned, we need to interchange columns as well in order to make the transformation a similarity transformation. Then the $k + 1$st row is subtracted from later rows in line 32, and the appropriate column subtraction (necessary to make the transformation a similarity transformation) is performed in line 33.

The function `shiftamount` in lines 38–48 uses the quadratic formula to find the eigenvalues of the lower right 2 by 2 minor of the matrix `**a`, and calculates the appropriate shift to be applied to the matrix before using plane rotations. The function `protation in lines 50--61` calculates $\sin\theta$ and $\cos\theta$ for the plain rotation that would make the element a_{qp} zero. The angle θ itself does not need to be determined. This function returns the values of $\sin\theta$ and $\cos\theta$ as pointers `sintheta` and `costheta`. The function `leftprotate` in lines 63–76 applies this transformation as a matrix multiplication on the left, and the function `rightprotate` in lines 78–91 applies its transpose as a matrix multiplication on the right. The last three functions mentioned return 0 in case p and q are equal (when the transformation cannot be carried out), otherwise they return 1.

The file `qr.c` contains the function applying these plain rotations:

```
 1 #include "qr.h"
 2
 3 double qr(double **a, int n, double tol, int maxits,
 4       int *itcount, int *success)
 5 {
 6   int i, j, pk;
 7   double lambda, sint, cost, oldsint, oldcost;
 8   *success=0;
 9   for (*itcount=1;*itcount<=maxits;(*itcount)++) {
10     pk=shiftamount(a,n);
11     for (i=1;i<=n;i++) a[i][i] -= pk;
12     for (i=1;i<n;i++) {
13       protation(a,i,i+1,n,&sint,&cost);
```

```
14       leftprotate(a,i,i+1,n,sint,cost);
15       if (i>1) rightprotate(a,i-1,i,n,oldsint,oldcost);
16       oldsint=sint; oldcost=cost;
17     }
18     rightprotate(a,n-1,n,n,oldsint,oldcost);
19     for (i=1;i<=n;i++) a[i][i] += pk;
20     if ( absval(a[n][n-1])<=tol ) {
21       *success=1;
22       return a[n][n];
23     }
24   }
25   /* We get to this point if we are unsuccessful */
26   return 0.;
27 }
28
29 void printmatrix(double **a, int n)
30 {
31     int i,j;
32     for (i=1;i<=n;i++) {
33       printf("\n");
34       for (j=1;j<=n;j++)
35         printf("%8.5f ",a[i][j]);
36     }
37     printf("\n");
38 }
39
40 void printmatrix_mathematica(double **a, int n)
41 {
42     int i,j;
43     printf("\nm={");
44     for (i=1;i<=n;i++) {
45       if ( i>1 ) printf("},");
46       printf("\n{");
47       for (j=1;j<n;j++)
48         printf("%8.5f,",a[i][j]);
49       printf("%8.5f",a[i][n]);
50     }
51     printf("}}\n");
52     printf("N[Eigenvalues[m]]\n");
53 }
```

The function `qr` uses plain rotations involving rows i and $i+1$ on the Hessenberg matrix to make the element $a_{i+1\,i}$ zero. One needs to multiply with these rotation matrices also on the right with one step delay as was explained above. For this reason, only the last matrix used needs to be remembered. This is why one stores the current values of `sint` and `cost` in `oldsint` and `oldcost` on line 16, to use it for the next cycle of the loop on lines 9–16. The parameters of the function `qr` are the matrix `**a`, the size `n` of the matrix, the tolerance `tol` to stop the iteration when the test on line 20 shows that the element `a[n][n-1]` is small enough, the maximum number of iterations `maxits` allowed, the number of iterations `*itcount` (as a pointer, so the calling program can read it), and the integer `*success` showing whether the function was successful.

The function `printmatrix` can be used to print out the entries of the matrix `**a`. A similar function `printmatrix_mathematica` prints out the matrix in the form usable by the software

Mathematica. This is useful for debugging purposes, since Mathematica can be used to calculate the eigenvalues of the matrix, to compare it with the result produced by the present program. This program was used with the following input file `symm_coeffs`:

```
1 5e-14
2 5
3 1  2  3  5  8
4    3 -2  2  3
5       1  1 -1
6         -1  3
7             2
```

The numbers on the left are line numbers. The first number is the tolerance, the second number is the size of the matrix; rows 3–7 then contain the upper triangle of a symmetric matrix. The present program does not require that the matrix be symmetric, but it works only if the eigenvalues are real. For a symmetric matrix, it is guaranteed that the eigenvalues are real. The calling program is contained in the file `main.c`:

```
 1 #include "qr.h"
 2
 3 main()
 4 {
 5   /* This program reads in the elements of in the
 6      upper triangle of a square matrix int the file
 7      named symm_coeffs, and fills in the lower triangle
 8      to make the matrix symmetric. The first entry in
 9      the file is the required tolerance, the
10      second entry is n, the size of the matrix. The
11      rest of the entries are the elements in the
12      upper triangle of the matrix. The first entry
13      must be of type float, the third entry must be
14      an unsigned integer, the other entries can be
15      integers or reals.                          */
16   double **a, lambda;
17   float tol;
18   int n, i, j, subs, readerror=0, success, itcount;
19   char ss[25];
20   FILE *coefffile;
21   coefffile=fopen("symm_coeffs", "r");
22   fscanf(coefffile, "%f", &tol);
23   printf("Tolerance used: %g\n", tol);
24   fscanf(coefffile, "%u", &n);
25   a=allocmatrix(n);
26   for (i=1;i<=n;i++) {
27     if ( readerror ) break;
28     for (j=i;j<=n;j++) {
29       if (fscanf(coefffile, "%s", ss)==EOF) {
30         readerror=1;
31         printf("Not enough coefficients\n");
32         printf("i=%u j=%u\n", i, j);
33         break;
34       }
35       a[j][i]=a[i][j]=strtod(ss,NULL);
```

```
36     }
37   }
38   fclose(coefffile);
39   if ( readerror ) printf("Not enough data\n");
40   else {
41     simgauss(a,n);
42     lambda=qr(a,n,tol,100,&itcount,&success);
43     printf("The number of iterations was %u\n", itcount);
44     if ( success ) printf("An eigenvalue is %16.12f\n", lambda);
45     else printf("No eigenvalue could be calculated\n");
46     printmatrix(a,n);
47   }
48   free(a[0]); free(a);
49 }
```

In lines 26–37 the matrix is read; if there is no error in reading the input (no end of file is reached before the required number of data are read), in line 41 the function `simgauss` is invoked to perform a similarity transformation based on Gaussian elimination. Then, in line 42, the eigenvalue is calculated by using plane rotations. In lines 42–45 the result is printed out, and in line 46 the final entries of the matrix `**a` are printed out. The printout of the program is as follows:

```
1 Tolerance used: 5e-14
2 The number of iterations was 20
3 An eigenvalue is  -0.624673383928
4
5 13.58357  3.79959 -0.88053  4.24524  6.29943
6  0.00001 -7.85017  2.74484 -0.32985 -3.76364
7 -0.00000  0.01046  4.25460  1.40126 -1.52720
8 -0.00000  0.00000 -0.00000 -3.36333 -1.18487
9 -0.00000 -0.00000 -0.00000 -0.00000 -0.62467
```

The lower right entry of the matrix contains the calculated eigenvalue. It is interesting to note that the other elements in the main diagonal of the matrix are also fairly close to the other eigenvalues. In fact, the eigenvalues, as printed out by Wielandt's deflation are as follows:

```
1 Tolerance used: 5e-14
2  56 iterations used for eigenvalue number 1
3  48 iterations used for eigenvalue number 2
4 139 iterations used for eigenvalue number 3
5  20 iterations used for eigenvalue number 4
6 All eigenvalues were calculated. They are:
7  13.583574215424  -7.852544184194   4.256975565094  -3.363332212396
8  -0.624673383928
```

43. SPECTRAL RADIUS AND GAUSS-SEIDEL ITERATION

Spectral radius. In this section we will discuss $N \times N$ matrices with complex numbers as entries. An *eigenvalue* of such a matrix A is a complex number for which there is a nonzero n-dimensional column vector $\mathbf{x} = (x_1, x_2, \dots, x_N)^T$, called the *eigenvector* corresponding to λ, for which $A\mathbf{x} = \lambda x$; in other word, $(A - \lambda I)\mathbf{x} = 0$, where I is the identity matrix. This equation can also be read as saying that the linear combination of the column vectors of $A - \lambda I$) formed by the coefficients x_1,

$\ldots$, x_N; that is, these column vectors are linearly dependent. This is equivalent to saying that $A - \lambda I$ is not invertible; in what follows, it will be simpler to write $\lambda - A$ instead of $\lambda I - A$. The *spectrum* of A is the set of its eigenvalues. In other words, one can define the spectrum of A as the set of complex numbers for which the matrix $\lambda - A$ is not invertible.[161]

The *spectral radius* $\rho(A)$ of a matrix A is defined as the maximum of the absolute values of the elements of its spectrum. If $\|\cdot\|$ is a matrix norm *compatible* with a vector norm,[162] for any matrix A we have

$$\rho(A) \leq \|A^n\|^{1/n} \qquad \text{for any positive integer } n \geq 1. \tag{1}$$

Furthermore,

$$\rho(A) = \lim_{n\to\infty} \|A^n\|^{1/n}. \tag{2}$$

The proof of the first of these relations is easy to prove. In fact, if λ is an eigenvalue of A with eigenvector $\mathbf{x}$, then $A\mathbf{x} = \lambda\mathbf{x}$, $A^2\mathbf{x} = A(A\mathbf{x}) = A(\lambda\mathbf{x}) = \lambda A\mathbf{x} = \lambda^2\mathbf{x}$, and, in fact, using a similar argument combined with induction on n, for any positive integer n, we have $A^n\mathbf{x} = \lambda^n\mathbf{x}$. Taking λ to be the eigenvalue with the largest absolute value of A, and $\mathbf{x}$ to be a corresponding eigenvector with $\|\mathbf{x}\| = 1$, we have

$$\rho(A) = \lambda = (\lambda^n)^{1/n} = (|\lambda^n|\,\|\mathbf{x}\|)^{1/n} = (|\lambda|^n|\mathbf{x}\|)^{1/n} = \|A^n\mathbf{x}\|^{1/n} \leq (\|A^n\|\|\mathbf{x}\|)^{1/n} \leq \|A^n\|^{1/n}.$$

The proof of (2) is more complicated. For matrices, it can be carried out using the *Jordan normal form*,[163] but the most elegant proof, generalizable way beyond the scope of matrices, relies on the concept of the *resolvent* of a matrix, discussed in the next subsection. This discussion, however, relies on the theory of complex functions, and should be skipped by those unfamiliar with the basics of this theory.

The resolvent of a matrix. The *resolvent* (of *resolvent function*) $R(\zeta)$ of the matrix A is defined as

$$R(\zeta) = R_A(\zeta) \stackrel{def}{=} (\zeta - A)^{-1},$$

where ζ is a complex variable. $R(\zeta)$ is defined at every point where $\zeta - A$ is invertible, that is, at every point ζ that does not belong to the spectrum of A. The complement of the spectrum of A, that is, the domain of R_A, is called the *resolvent set* of A.

If ζ_0 is in the resolvent set of A and ζ is close to ζ_0, one might try to calculate the inverse of

$$\zeta - A = (\zeta_0 - A) - (\zeta_0 - \zeta) = \left(I - (\zeta_0 - \zeta)(\zeta_0 - A)^{-1}\right)(\zeta_0 - A);$$

note that the inverse occurring in the second factor on the right-hand side exists according to the assumption that ζ_0 belongs to the resolvent set of A. We want to calculate this inverse of the left-hand side of A; to this end, we need the inverse of the first factor on the right-hand side. In analogy for the expansion

$$(1 - x)^{-1} = 1 + x + x^2 + \ldots = \sum_{n=0}^{\infty} x^n,$$

[161] This second definition of the spectrum is much better than the first definition, since it can be generalized to entities much more general than matrices, and in the generalized setting not all elements of the spectrum will be eigenvalues.

[162] A matrix norm $\|\cdot\|$ is called compatible, or *consistent*, with a vector norm, also denoted as $\|\cdot\|$, if for any matrix A and any column vector $\mathbf{x}$ we have

$$\|A\mathbf{x}\| \leq \|A\|\|\mathbf{x}\|.$$

In the discussion that follows, $\|\cdot\|$ will denote a fixed vector norm or a fixed matrix norm compatible with this vector norm.

[163] For the Jordan normal form of matrices, see, for example Serge Lang, Linear Algebra, Second edition, Addison-Wesley Publishing Company, Reading, Mass, 1971.

convergent for $|x| < 1$, we might try

$$\big(I - (\zeta_0 - \zeta)(\zeta_0 - A)^{-1}\big)^{-1} = \sum_{n=0}^{\infty} \big((\zeta_0 - \zeta)(\zeta_0 - A)^{-1}\big)^n = \sum_{n=0}^{\infty} (\zeta_0 - \zeta)^n (\zeta_0 - A)^{-n}.$$

In fact, it is easy to show that if

$$|\zeta_0 - \zeta| \, \|(\zeta_0 - A)^{-1}\| < 1,$$

then this series converges and equals the inverse on the left-hand side. This shows that the resolvent set is open, since the small circle represented by the numbers ζ belongs to the resolvent set; the expansion can be termwise differentiated[164] in this small circle, showing that the resolvent is holomorphic[165] on its domain (since ζ_0 was an arbitrary element of the domain).

Take any circle with center at the origin that includes the spectrum of A in its interior. Note that any circle of radius greater than the spectral radius $\rho(A)$ of A will do. For any positive integer n, the value of the integral

$$\oint_{|\zeta|=r} \zeta^n R_A(\zeta) \, d\zeta$$

taken along the circle $\{\zeta : |\zeta| = r\}$ of radius r with center at the origin, is independent of r as long as r is large enough for this circle to include the spectrum of A in its interior, since a well-known theorem of complex function theory says that path integrals on simply connected domains[166] only depend on the starting point and the endpoint, and not the path connecting them. For any such integral with large enough r, we can write

$$\begin{aligned}\oint_{|\zeta|=r} \zeta^n R_A(\zeta) \, d\zeta &= \oint_{|\zeta|=r} \zeta^n (\zeta - A)^{-1} \, d\zeta = \oint_{|\zeta|=r} \zeta^{n-1} \left(I - \zeta^{-1} A\right)^{-1} \, d\zeta \\ &= \oint_{|\zeta|=r} \zeta^{n-1} \sum_{k=0}^{\infty} \zeta^{-k} A^k \, d\zeta = \oint_{|\zeta|=r} \sum_{k=0}^{\infty} \zeta^{n-1-k} A^k \, d\zeta = \sum_{k=0}^{\infty} \oint_{|\zeta|=r} \zeta^{n-k-k} A^k \, d\zeta = 2\pi i A^n;\end{aligned}$$

the expansion after the third equality is analogous to the expansion of $(1 - x)^{-1}$ described above, and it is convergent for large r, e.g. if $r^{-1}\|A\| = |\zeta|^{-1}\|A\| < 1$; in this case the integral of the series can be evaluated termwise, as the forth equality shows. The last equality follows since the integral of ζ^{-n} for integer $n \geq 0$ is zero unless $n - 1 - k = -1$, i.e., unless $k = n$.

However, as we remarked above, the value of the integral on the left-hand side, is the does not depend on r as long as $r > \rho(A)$. So the left-hand side is still equal to the right-hand side for any such r, even though the intervening infinite series need not make sense for all such r. Hence, for any $r > \rho(A)$ and $n > ge1$ we have

$$\begin{aligned}\|A^n\| = \left\| \frac{1}{2\pi i} \oint_{|\zeta|=r} \zeta^n R_A(\zeta) \, d\zeta \right\| &\leq \frac{1}{2\pi} \oint_{|\zeta|=r} |\zeta^n| \max_{|\zeta|=r} \|R_A(\zeta)\| \, d|\zeta| \\ &\leq \frac{1}{2\pi} 2\pi r \cdot r^n \max_{|\zeta|=r} \|R_A(\zeta)\| = r^{n+1} \max_{|\zeta|=r} \|R_A(\zeta)\|;\end{aligned}$$

[164] That is, if one differentiates each term of the series and then adds up these terms, the result will be the derivative of the sum of this series.

[165] *Holomorphic* is the term used in complex function theory for functions that are differentiable (differentiability in complex function theory behaves very differently in differentiability on the real line, so much so that the use of a different term is justified).

[166] A *domain* is a connected open set, i.e., an open set such that from any point of the set one can reach any other point by walking along a number of straight line segments. A domain is called simply connected if it has no holes inside; a somewhat more precise way of describing this is that any path consisting of line line segments entirely inside the domain can be continuously deformed into any other similar path with the same endpoints – when doing the deformation, one may need to break up a line segment into a path consisting of a number of line segments.

the second inequality holds since $|\zeta| = r$ along the path of integration, and the length of the path of integration is $2\pi r$. Making $n \to \infty$, we obtain

$$\limsup_{n\to\infty} \|A^n\|^{1/n} \le \lim_{n\to\infty} r^{(n+1)/n} \big(\max_{|\zeta|=r} \|R_A(\zeta)\|\big)^{1/n} = r;$$

on the left-hand side, we need to take lim sup, since the existence of the limit is not guaranteed. Since this is true for any $r > \rho(A)$, it follows that

$$\limsup_{n\to\infty} \|A^n\|^{1/n} \le \rho(A).$$

This, together with (1), establishes (2).

Hermitian matrices. Write $\bar{z}$ for the complex conjugate of the number z; that is, if $z = x + iy$ with real x and y, then $\bar{z} = x - iy$. Given an $m \times n$ matrix $A = (a_{jk})_{j,k:1\le j\le m,\ 1\le k\le n}$, the *conjugate transpose* A^* of A is the $n \times m$ matrix $A^* = (\overline{a_{jk}})_{k,j:1\le j\le m,\ 1\le k\le n}$; to understand the notation, note that the first subscript listed outside the parentheses before the column identifies rows, and the second one identifies column. The order of listing the ranges of the subscripts is immaterial. Thus the matrix A has m rows and n columns, and the entry in the intersection of the jth row and the kth column is a_{jk}. On the other hand, the matrix $A*$ has n rows and m columns, and the element in the intersection of the kth row and jth column is $\overline{a_{jk}}$, the complex conjugate of a_{jk}. We could equally well have written $A^* = (\overline{a_{kj}})_{j,k:1\le j\le n,\ 1\le k\le m}$. If one can multiply the matrices A and B (in this order),[167] then it is easy to check that one can also multiply the matrices B^* and A^*, and

$$(AB)^* = B^*A^*.$$

This is true even in case AB is a number,[168] with the understanding that for a number c we write $c^* = \bar{c}$.

In what follows, N will be a fixed positive integer, and we will only be interested in the conjugate transpose of an $N \times N$ square matrix, or an N-dimensional column vector. The conjugate transpose of the latter is an N-dimensional row vector. For an N-dimensional column vector $\mathbf{x} = (x_1, \dots, x_n)$ with complex entries, the l^2 norm of $\mathbf{x}$ is defined as

$$\|\mathbf{x}\| = \|\mathbf{x}\|_2 \stackrel{def}{=} \mathbf{x}^*\mathbf{x} = \left(\sum_{n=1}^{N} \overline{x_n} x_n\right)^{1/2} = \left(\sum_{n=1}^{N} |x_n|^2\right)^{1/2}.$$

The norm for vectors $\|\cdot\|$ will from now on denote the l^2 norm. The matrix norm, also called l^2 norm, *induced* by this vector norm is defined for $N \times N$ matrices is defined as

$$\|A\| \stackrel{def}{=} \max_{\mathbf{x}:\|\mathbf{x}\|=1} \|A\mathbf{x}\|;$$

it is easily seen that this matrix norm is compatible with the vector norm $\|\cdot\|$, that is,

$$\|A\mathbf{x}\| \le \|\mathbf{x}\|.$$

We will be interested in the expression, called a *quadratic form*, $\mathbf{x}^*A\mathbf{x}$, associated with an $N \times N$ matrix $A = (a_{ij})$. It is easy to see that, with $x = (x_1, \dots x_N)^T$, we have

$$\mathbf{x}^*A\mathbf{x} = \sum_{j,k=1}^{N} \overline{x_j} a_{jk} x_k.$$

[167] That is, if the number of columns of A is the same as the number of rows of B.

[168] I.e., if A is a column vector, and B is a row vector (of the same dimension, so they can be multiplied).

The $N \times N$ matrix $A = (a_{ij})$ is called *Hermitian*[169] if $A^* = A$; this is equivalent to saying that $a_{jk} = \overline{a_{kj}}$ for any j, k with $1 \le j, k \le N$. For a Hermitian matrix A and for any column vector $\mathbf{x}$, the quadratic form $\mathbf{x}^* A\mathbf{x}$ is a real number. Indeed,

$$(\mathbf{x}^* A\mathbf{x})^* = \mathbf{x}^* A^* (x^*)^* = \mathbf{x}^* A\mathbf{x}.$$

A Hermitian matrix A is called *positive definite* if for any nonzero column vector $\mathbf{x}$ we have

$$\mathbf{x}^* A\mathbf{x} > 0.$$

As in the real case, this implies that the diagonal elements are positive, since if in the vector $\mathbf{x}$ we have x_j all but the kth component is zero, and the kth component is 1, then we have

$$\mathbf{x}^* A\mathbf{x} = a_{kk}.$$

If the entries of A are real, then it is sufficient to require this inequality for column vectors with real entries. In fact, any column vector $\mathbf{z}$ can be written as $\mathbf{x}+i\mathbf{y}$ with $\mathbf{x}$ and $\mathbf{y}$ being column vectors with real entries. Then

$$\mathbf{z}^* = (\mathbf{x} + i\mathbf{y})^* = \mathbf{x}^T - i\mathbf{y}^T,$$

and so

$$\mathbf{z}^* A\mathbf{z} = (\mathbf{x}^T - i\mathbf{y}^T)A(\mathbf{x} + i\mathbf{y}) = \mathbf{x}^T A\mathbf{x}^T + i\mathbf{x}^T A\mathbf{y}^T - i\mathbf{y}^T A\mathbf{x}^T + \mathbf{y}^T A\mathbf{y}^T.$$

Since we assumed for this calculation that the entries of A are real, we have $A^* = A^T$, that is A is a real symmetric matrix. Then $\mathbf{x}^T A\mathbf{y}^T$ is real, so

$$\mathbf{x}^T A\mathbf{y} = (\mathbf{x}^T A\mathbf{y})T = \mathbf{y}^T A^T \mathbf{x} = \mathbf{y}^T A\mathbf{x},$$

and so

$$\mathbf{z}^* A\mathbf{z} = \mathbf{x}^T A\mathbf{x}^T + \mathbf{y}^T A\mathbf{y}^T.$$

That is, for symmetric matrices with real entries, the present definition of the term "positive definite" given for Hermitian matrices reverts to the definition given earlier.

Gauss-Seidel iteration. Consider the system of linear equations $A\mathbf{x} = \mathbf{b}$, where $A = (a_{ij})$ is an $N \times N$ matrix with complex entries, $\mathbf{x} = (x_1, \dots, x_N)^T$ is the column vector of the unknowns, and $\mathbf{b} = (b_1, \dots, b_N)^T$ is the right-hand side of the equation. Write A in the form

$$A = L + D + U,$$

where L is a lower triangular matrix with zeros in the diagonal, D is a diagonal matrix, and U is an upper diagonal matrix with with zeros in the diagonal.[170] Write $\mathbf{x}_n = (x_1^{(n)}, \dots, x_N^{(n)})^T$ for the nth approximation to the solution. Then Gauss-Seidel iteration can be described as follows: We start with an arbitrary vector $\mathbf{x}_0$, and for $n \ge 0$ we determine the vector $\mathbf{x}_{n+1}$ by using the equation

$$(L + D)\mathbf{x}_{n+1} + U\mathbf{x}_n = \mathbf{b}.$$

Indeed, this can be written as a system of equations

$$\sum_{k=1}^{j} a_{jk} x_k^{(n+1)} + \sum_{k=j+1}^{N} a_{jk} x_k^{(n)} = b_j \qquad (1 \le j \le N).$$

[169] Named after the French mathematician Charles Hermite, 1822-1901.

[170] Clearly, L and U here is not the same as the matrices L and U obtained in LU factorization.

By solving the jth equation for $x_j^{(n+1)}$, we obtain the usual equations used in Gauss-Seidel iteration. When solving these equations, it is necessary to require that $a_{jj} \neq 0$ for $1 \leq j \leq N$, i.e., that the diagonal elements of the matrix A (which is the same as the diagonal elements of the matrix D) are nonzero. The matrix form of these equations can also be solved for $\mathbf{x}_{n+1}$:

$$\mathbf{x}_{n+1} = (L + D)^{-1}(\mathbf{b} - U\mathbf{x}_n). \tag{3}$$

The condition for solving these equations is that the matrix $L + D$ should be invertible. As $L + D$ is a lower triangular matrix, it is invertible if and only if its diagonal entries, i.e., the diagonal entries of D, are nonzero (this is the same as the condition mentioned above necessary in order to be able to solve the jth equation for $x_j^{(n+1)}$ for each j in the above system of equations). The following is an important result concerning the convergence of Gauss-Seidel iteration.

THEOREM. *Assume A is a positive definite Hermitian matrix (with complex entries). Then the equation $A\mathbf{x} = \mathbf{b}$ is solvable by Gauss-Seidel iteration.*

PROOF. If $\mathbf{x}_n$ is the nth vector of Gauss-Seidel iteration, then for $n \geq 0$ we have

$$\mathbf{x}_{n+2} - x_{n+1} = -(L + D)^{-1}U(\mathbf{x}_{n+1} - \mathbf{x}_n).$$

Indeed, this equation can be obtained by taking equation (3) with $n+1$ replacing n, and subtracting from it equation (3) as given. Iterating this, we can see that

$$\mathbf{x}_{n+1} - \mathbf{x}_n = (-1)^n((L + D)^{-1}U)^n(\mathbf{x}_1 - \mathbf{x}_0),$$

that is,

$$\|\mathbf{x}_{n+1} - \mathbf{x}_n\| \leq \|(L + D)^{-1}U)^n\| \, \|\mathbf{x}_1 - \mathbf{x}_0\|,$$

where we used l^2 norms. In order to show that the iteration converges, it will be sufficient to show that the series $\sum_{n=0}^{\infty} \|\mathbf{x}_{n+1} - \mathbf{x}_n\|$ is convergent, and for this it will be sufficient that there be a positive number $q < 1$ such that

$$\|(L + D)^{-1}U)^n\| < q^n,$$

and this will follow according to (2) if the spectral radius $\rho((L+D)^{-1}U)$ of $(L+D)^{-1}U$ is less than 1, that is, if all eigenvalues of $(L+D)^{-1}U$ lie in the unit circle (of the complex plane). This is what we are going to show.

To this end, let λ be an eigenvalue of $(L + D)^{-1}U$ with the associated eigenvector $\mathbf{x}$; that is,

$$(L + D)^{-1}U\mathbf{x} = \lambda\mathbf{x},$$

where λ is a complex number and $\mathbf{x}$ is a nonzero vector. We need to show that $|\lambda| < 1$. Recall that the matrix $A = L + D + U$ is assumed to be Hermitian; hence the entries of D are real and $U = L^*$. Hence, the last equation can be written as

$$(L + D)^{-1}L^*\mathbf{x} = \lambda\mathbf{x},$$

or, multiplying both sides by $L + D$ on the left and switching the sides, as

$$\lambda(L + D)\mathbf{x} = L^*\mathbf{x}.$$

Adding $\lambda L^*\mathbf{x}$ to both sides, noting that $L + D + L^* = A$, and multiplying both sides by x^* on the left, we obtain

$$\lambda\mathbf{x}^*A\mathbf{x} = (1 + \lambda)\mathbf{x}^*L^*\mathbf{x}. \tag{4}$$

The conjugate transpose of this equation is

$$\bar{\lambda}\mathbf{x}^* A\mathbf{x} = (1+\bar{\lambda})\mathbf{x}^* L\mathbf{x}.$$

Multiplying the first equation by $1+\bar{\lambda}$ and the second one by $1+\lambda$ and adding the equations, we obtain

$$(\lambda+\bar{\lambda}+2\lambda\bar{\lambda})\bar{x}^* A\bar{x} = (1+\lambda)(1+\bar{\lambda})\mathbf{x}^*(L+L^*)\mathbf{x}.$$

Adding $(1+\lambda)(1+\bar{\lambda})\mathbf{x}^* D\mathbf{x}$ to both sides and noting on the right-hand side that $L+D+L^* = A$, we obtain

$$(1+\lambda)(1+\bar{\lambda})\mathbf{x}^* D\mathbf{x} + (\lambda+\bar{\lambda}+2\lambda\bar{\lambda})\bar{x}^* A\bar{x} = (1+\lambda+\bar{\lambda}+\lambda\bar{\lambda})\mathbf{x}^* A\mathbf{x}.$$

Transferring the term involving A on the left-hand side to the right-hand side, noting that $\lambda\bar{\lambda} = |\lambda|^2$ and $(1+\lambda)(1+\bar{\lambda}) = (1+\lambda)\overline{(1+\lambda)}$, we obtain that

$$|1+\lambda|^2\mathbf{x}^* D\mathbf{x} = (1-|\lambda|^2)\mathbf{x}^* A\mathbf{x}.$$

Note that $\lambda \neq -1$; indeed, equation (4) with $\lambda = -1$ implies that $\mathbf{x}^* A\mathbf{x} = 0$, whereas $\mathbf{x}^* A\mathbf{x} > 0$ since A is positive definite. A being positive definite also implies that all the diagonal elements of A (which are the same as the diagonal elements of D) are positive; hence we also have $\mathbf{x}^* D\mathbf{x} > 0$. Thus, the last displayed equation can hold only if $1-|\lambda|^2 > 0$. This shows that $|\lambda| < 1$. □

44. ORTHOGONAL POLYNOMIALS

Let (a,b) be an interval and let w be a positive integrable function on (a,b).[171] The polynomials

$$p_n(x) = \gamma_n x^n + \dots \qquad (n = 0,1,2\dots)$$

such that

$$\int_a^b p_m(x)p_n(x)w(x)\,dx = \delta_{mn} \qquad (m,n = 0,1,2,\dots) \tag{1}$$

are called *orthonormal* on the interval (a,b) with respect to the weight function w.[172] Here we assume that $\gamma_n > 0$, so that p_n is a polynomial of degree n.[173] Sometimes one drops to condition of normality, and one wants to consider the *monic*[174] polynomials $\frac{1}{\gamma_n}p_n$.

One can construct the polynomials p_n by using what is called the *Gram-Schmidt orthogonalization*: Let

$$\gamma_0 = \left(\int_a^b w(x)\,dx\right)^{-1/2};$$

[171] We will assume that the interval (a,b) is finite. However, the arguments can be extended to the case when a or b or both are infinite if one assumes that the function $|x|^n w(x)$ is integrable on (a,b) for all positive integers n.

[172] The word *orthonormal* is constructed by combining the words *orthogonal* (referring to the above relation when $m \neq n$) and *normal* (referring to the case when $m = n$.

[173] The condition that

$$\int_a^b p_n^2(x)w(x)\,dx = 1$$

determines only $|\gamma_n|$. One then is free to choose γ to be positive or negative. We make the choice of γ being positive.

[174] Monic polynomials are polynomials with leading coefficient one.

the integral here is positive since we assumed that $w(x) > 0$; hence raising it to the power $-1/2$ is meaningful. Put $p_0(x) \stackrel{def}{=} \gamma_0$. Then

$$\int_a^b p_0(x) = 1.$$

Assuming that the polynomials $p_k(x)$ of degree k have been defined for k with $0 \le k < n$ in such a way that

$$\int_a^b p_k(x)p_l(x)\,dx = \delta_{kl}$$

whenever $1 \le k, l < n$. Writing

$$\eta_k = \int_a^b x^n p_k(x)\,dx,$$

put

$$P_n(x) = x^n - \sum_{k=0}^{n-1} \eta_k p_k(x).$$

For $l < n$ we have

$$\int_a^b P_n(x)p_l(x)w(x)\,dx = \int_a^b x^n p_l(x)w(x)\,dx - \sum_{k=0}^{n-1} \int_a^b \eta_k p_k(x)p_l(x)\,dx = \eta_l - \eta_l = 0.$$

Write

$$\gamma_n = \left(\int_a^b P_n^2(x)w(x)\,dx\right)^{-1/2};$$

note that the integral here is positive since $w(x)$ is positive and the polynomial $P_n(x)$ is not identically zero (namely, its leading coefficient is 1). Put $p_n(x) = \gamma_n P_n(x)$. Then

$$\int_a^b p_n^2(x)w(x)\,dx = 1.$$

Thus, the system of polynomials p_n constructed this way is orthonormal on (a, b) with respect to the weight function w.

It is clear that every polynomial $p(x)$ of degree $m \ge 0$ can be written as a linear combination

$$p(x) = \sum_{i=0}^{m} \lambda_i p_i(x).$$

Hence, (1) implies

$$\int_a^b p_n(x)p(x)w(x)\,dx = 0 \tag{2}$$

whenever $p(x)$ has degree less than n. Now, it is also clear that $p_{n+1}(x)$, can be written as a linear combination

$$p_{n+1}(x) = \lambda_{n+1} x p_n(x) + \sum_{i=0}^{n} \lambda_i p_i(x);$$

in fact, any polynomial of degree $n+1$ can be written in such a way for appropriate choices of the coefficients λ_i, $0 \le \lambda \le n+1$. Multiplying both sides by $p_k(x)w(x)$ for some integer k and then integrating on (a, b), we obtain

$$\int_a^b p_{n+1}(x)p_k(x)w(x)\,dx = \lambda_{n+1} \int_a^b p_n(x)\,xp_k(x)\,w(x)\,dx + \sum_{n=0}^{n} \lambda_i \int_a^b p_i(x)p_k(x)w(x)\,dx.$$

For k with $0 \le k < n-1$, every integral is zero except the coefficient of λ_i for $i = k$ on the right-hand side; in fact, we have

$$\int_a^b p_n(x)\, xp_k(x)\, w(x)\, dx = 0$$

in particular, since the polynomial $xp_k(x)$ has degree less than n. Hence we have[175]

$$0 = \lambda_k \int_a^b p_k^2(x) w(x)\, dx = \lambda_k.$$

Therefore, we have

$$p_{n+1}(x) = \lambda_{n+1} x p_n(x) + \lambda_n p_n(x) + \lambda_{n-1} p_{n-1}(x) \tag{3}$$

This is the *three-term recurrence formula* for orthogonal polynomials. For $n = 0$, (3) should be replaced with

$$p_1(x) = \lambda_1 x p_0(x) + \lambda_n p_n(x),$$

as is easily seen by following the above calculation for $n = 0$. However, instead of considering the case $n = 0$ separately, it is more convenient to stipulate that $p_{-1}(x) \equiv 0$, in which case (3) becomes valid even for $n = 0$.

We will take a closer look at the coefficients λ_{n-1}, λ_n, and λ_{n+1} in (3). First, note that the coefficient of x^{n+1} on the left-hand side is γ_{n+1}, and that on the right-hand side is $\gamma_n \lambda_{n+1}$. Thus

$$\lambda_{n+1} = \frac{\gamma_{n+1}}{\gamma_n}; \tag{4}$$

note that the denominator on the right-hand side is not zero, since we assumed that $\gamma_n \neq 0$. Furthermore, $\lambda_{n+1} \neq 0$ since $\gamma_{n+1} \neq 0$. Multiplying both sides in (3) by $p_{n-1} w(x)$ and integrating, we obtain

$$\begin{aligned}\int_a^b & p_{n+1}(x) p_{n-1}(x) w(x)\, dx \\ &= \lambda_{n+1} \int_a^b x p_n(x) p_{n-1}(x) w(x)\, dx + \lambda_n \int_a^b p_n(x) p_{n-1} w(x)\, dx + \lambda_{n-1} \int_a^b p_{n-1}^2(x) w(x)\, dx.\end{aligned}$$

The integral on the left and the second integral on the right are zero, and the third integral on the right-hand side is 1. Thus,

$$0 = \lambda_{n+1} \int_a^b x p_n(x) p_{n-1}(x) w(x)\, dx + \lambda_{n-1}.$$

So, using the expression for λ_{n+1} given by (4), we obtain

$$\lambda_{n-1} = -\frac{\gamma_{n+1}}{\gamma_n} \int_a^b x p_n(x) p_{n-1}(x) w(x)\, dx.$$

This expression can be further simplified. In fact, we have

$$x p_{n-1}(x) = \gamma_{n-1} x^n + \text{lower order terms} = \frac{\gamma_{n-1}}{\gamma_n} p_n(x) + q(x),$$

[175] $p_k^2(x)$ is short for $(p_k(x))^2$.

where $q(x)$ is some polynomial of degree less than n. Thus, we have

$$\lambda_{n-1} = -\frac{\gamma_{n+1}}{\gamma_n}\int_a^b p_n(x)\, xp_{n-1}(x)w(x)\,dx = -\frac{\gamma_{n+1}}{\gamma_n}\int_a^b p_n(x)\left(\frac{\gamma_{n-1}}{\gamma_n}p_n(x)+q(x)\right)w(x)\,dx$$
$$= -\frac{\gamma_{n+1}\gamma_{n-1}}{\gamma_n^2}\int_a^b p_n^2(x)w(x)\,dx - \frac{\gamma_{n+1}}{\gamma_n}\int_a^b p_n(x)q(x)w(x)\,dx.$$

The first integral on the right-hand side is one and the second one is zero (because the degree of $q(x)$ is less than n – cf. (2)). Therefore,

$$\lambda_{n-1} = -\frac{\gamma_{n+1}\gamma_{n-1}}{\gamma_n^2}. \tag{5}$$

Multiplying both sides of (3) by $p_n(x)w(x)$ and integrating

$$\int_a^b p_{n+1}(x)p_n(x)w(x)\,dx$$
$$= \lambda_{n+1}\int_a^b xp_n^2(x)w(x)\,dx + \lambda_n\int_a^b p_n^2(x)w(x)\,dx + \lambda_{n-1}\int_a^b p_{n-1}(x)p_n(x)w(x)\,dx.$$

Again, the integral on the left and the third integral on the right is zero, while the second integral on the right-hand side is 1. Hence,

$$0 = \lambda_{n+1}\int_a^b xp_n^2(x)w(x)\,dx + \lambda_n.$$

Therefore, again using the above expression for λ_{n+1}, we obtain

$$\lambda_n = -\frac{\gamma_{n+1}}{\gamma_n}\int_a^b xp_n^2(x)w(x)\,dx. \tag{6}$$

Write

$$a_k = \frac{\gamma_{k-1}}{\gamma_k} \quad \text{for} \quad k \geq 1,$$

which makes $a_n = \gamma_{n-1}/\gamma_n$ and $a_{n+1} = \gamma_n/\gamma_{n+1}$; note that $a_k > 0$ since $\gamma_i > 0$. Further, put

$$b_n = \int_a^b xp_n^2(x)w(x)\,dx.$$

Multiplying (3) by $\frac{\gamma_n}{\gamma_{n+1}}$, (4)–(6) imply

$$xp_n(x) = a_{n+1}p_{n+1}(x) + b_np_n(x) + a_np_{n-1}(x). \tag{7}$$

This is the usual form of the three-term recurrence formula. This formula is valid for any $n \geq 0$, provided that we stipulate that $p_{-1}(x) \equiv 0$ (since this choice made (3) valid for $n = 0$).[176]

Given a system of orthonormal polynomials p_n, it can be shown that any "nice"[177] function f can be written as an infinite series

$$f(x) = \sum_{n=0}^{\infty} c_np_n(x).$$

[176] In this case, one can define a_0 arbitrarily, since it is multiplied by zero; e.g., one can take $a_0 = 1$.

[177] We do not define here what is meant by "nice." The theory of Riemann integration is not adequate to discuss this topic in an elegant framework; one needs the theory of *Lebesgue integration*, developed by Henry Lebesgue at the beginning of the twentieth century, for this.

This property of orthogonal polynomials is called *completeness*; that is, a system of orthogonal polynomials is *complete*. Multiplying the above equation by $p_k(x)w(x)$ and integration on the interval (a, b), we obtain[178]

$$c_k = \int_a^b f(x)p_k(x)w(x)\,dx.$$

An important theorem about the zeros of orthogonal polynomials is the following

THEOREM. *Let (a, b) be an interval, let $w(x)$ be positive on (a, b), and let $\{p_n\}_{n=0}^{\infty}$ be a system of orthonormal polynomials on (a, b) with weight w. Then for each n, the zeros of the polynomial p_n are single, real, and they are located in the interval (a, b).*

PROOF. Let $\lambda_1, \lambda_2, \ldots, \lambda_k$ be the zeros of p_n of odd multiplicity that are located in the interval (a, b). Since λ_i for i with $1 \le i \le k$ are the exactly places where the polynomial $p_n(x)$ changes its sign, Therefore, the polynomial

$$p_n(x)\prod_{j=1}^{k}(x - \lambda_j)$$

has constant sign (i.e., it is always positive, or always negative) on the interval (a, b), with the exception of finitely many places (the λ_i's and the zeros of even multiplicity of $p_n(x)$ in the interval (a, b)). Hence we have

$$\int_a^b p_n(x)\Big(\prod_{j=1}^{k}(x - \lambda_j)\Big)w(x)\,dx \neq 0;$$

thus, we must have $k = n$ in view of (2). Thus, the numbers $\lambda_1, \ldots, \lambda_n$ constitute all the zeros of the polynomial p_n (since p_n has degree n, so it cannot have more zeros).

Another important observation about the zeros is the following

LEMMA. *Let (a, b) and $\{p_n\}_{n=0}^{\infty}$ be as in the Theorem above. Then p_n and p_{n+1} have no common zeros.*

PROOF. Assume $p_{n+1}(\lambda) = p_n(\lambda) = 0$ for some λ. Then $p_{n-1}(\lambda) = 0$ according the the recurrence formula (7). Repeating this argument, we can conclude that $p_{n-2}(\lambda) = \ldots = p_0(\lambda) = 0$. However, p_0 must be a constant different from zero. This contradiction shows that the assumption $p_{n+1}(\lambda) = p_n(\lambda) = 0$ cannot hold.

Much more than this is true; namely, the zeros of p_{n+1} and p_n are interlaced in the sense that every zero of p_n is located between adjacent zeros of p_{n+1}, and exactly one zero of p_n is located between two adjacent zeros of p_{n+1}. This can be established by induction on n with the aid of (7) by analyzing how the sign changes of $p_n(x)$ and $p_{n+1}(x)$ are related. We omit the details.

The Christoffel-Darboux formula. Let $k \ge 0$. We have

$$\begin{aligned} a_{k+1}\big(p_{k+1}(x)p_k(y) - p_k(x)p_{k+1}(y)\big) &= \big(a_{k+1}(p_{k+1}(x)\big)p_k(y) - p_k(x)\big(a_{k+1}p_{k+1}(y)\big) \\ &= \big((x - b_k)p_k(x) - a_k p_{k-1}(x)\big)p_k(y) - p_k(x)\big((y - b_k)p_k(y) - a_k p_{k-1}(y)\big) \\ &= (x - y)p_k(x)p_k(y) + a_k\big(p_k(x)p_{k-1}(y) - p_{k-1}(x)p_k(y)\big); \end{aligned}$$

the second equality here was obtained by using the recurrence equation (7) to express $a_{k+1}p_{k+1}(x)$ and $a_{k+1}p_{k+1}(y)$; note that the above calculation is true even for $k = 0$ if we take $p_{-1} \equiv 0$ (this

[178] In order to verify the following formula, one needs to be able to justify the legitimacy of the termwise integration of the infinite sum on on the right-hand side. This can be done fairly easily under very general circumstances using the theory of the Lebesgue integral. The theory of the Riemann integral is much less suitable for this purpose.

choice was needed to make (7) valid for $n = 0$). Dividing this equation by $x - y$ and rearranging it, we obtain

$$p_k(x)p_k(y) = a_{k+1}\frac{p_{k+1}(x)p_k(y) - p_k(x)p_{k+1}(y)}{x-y} - a_k\frac{p_k(x)p_{k-1}(y) - p_{k-1}(x)p_k(y)}{x-y}.$$

Summing this for $k = 0, 1, \ldots, n$, the sum on the right-hand side telescopes. Taking into account that $p_{-1} \equiv 0$, we can see that the second fraction on the right-hand side for $k = 0$ is zero. Hence we get

$$\sum_{k=0}^{n} p_k(x)p_k(y) = a_{n+1}\frac{p_{n+1}(x)p_n(y) - p_n(x)p_{n+1}(y)}{x-y}. \tag{8}$$

This is called the Christoffel-Darboux formula. Making $y \to x$, we obtain

$$\begin{aligned}\sum_{k=0}^{n} p_k^2(x) &= a_{n+1}\left(-p_{n+1}(x)\lim_{y\to x}\frac{p_n(y)-p_n(x)}{y-x} + p_n(x)\lim_{y\to x}\frac{p_{n+1}(y)-p_{n+1}(x)}{y-x}\right)\\ &= a_{n+1}\big(p_n(x)p'_{n+1}(x) - p_{n+1}(x)p'_n(x)\big)\end{aligned}$$

C. W. Clenshaw's summation method. Using the recurrence formula (7), the sum

$$f_n(x) = \sum_{k=0}^{n-1} c_k p_k(x)$$

can be efficiently evaluated as follows. Write $y_{n+1} = y_n = 0$, and put

$$y_k = \frac{x - b_k}{a_{k+1}}y_{k+1} - \frac{a_{k+1}}{a_{k+2}}y_{k+2} + c_k$$

for $k = n-1, n-2, \ldots, 0$. Solving this for c_k and substituting this into the above sum, we obtain

$$f_n(x) = \sum_{k=0}^{n-1} c_k p_k(x) = \sum_{k=0}^{n-1}\left(y_k - \frac{x-b_k}{a_{k+1}}y_{k+1} + \frac{a_{k+1}}{a_{k+2}}y_{k+2}\right)p_k(x).$$

Given m with $2 \le m \le n-1$, the quantity y_m occurs on the right-hand side in the terms corresponding to $k = m$, $k = m-1$, and $k = m-2$. Collecting these terms, we can see that the sum of the terms involving y_m on the right-hand side is

$$y_m\left(p_m(x) - \frac{x-b_{m-1}}{a_m}p_{m-1}(x) + \frac{a_{m-1}}{a_m}p_{m-2}(x)\right).$$

According to (7), the quantity in the parentheses is zero. Therefore, all terms involving y_k for $k \ge 2$ add up to zero in the sum expressing $f_n(x)$. Hence

$$f_n(x) = \left(y_0 + \frac{x-b_0}{a_1}y_1\right)p_0(x) + y_1p_1(x).$$

45. GAUSS'S QUADRATURE FORMULA

Given an interval (a, b), let $w(x)$ be positive on (a, b), and let $\{p_k\}_{k=0}^{\infty}$ be a system of orthonormal polynomials on (a, b) with respect to the weight function $w(x)$. Fix n, and let $x_1, x_2, \ldots, x_n$ be the zeros of p_n.[179] For i with $1 \le i \le n$, let l_i be the Lagrange fundamental polynomials

$$l_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} = \frac{p_n(x)}{p_n'(x_i)(x - x_i)} \tag{1}$$

considered on account of the Lagrange interpolation.[180] The numbers

$$\Lambda_i = \int_a^b l_i(x) w(x)\, dx$$

are called *Christoffel numbers*; they play an important role in the *Gauss Quadrature Formula*:[181] Λ_i of course also depends on n; if one wants to indicate this dependence, one may write Λ_{in} instead of Λ_i.

THEOREM (GAUSS). *For any polynomial $p(x)$ of degree less than $2n$ we have*

$$\int_a^b p(x) w(x)\, dx = \sum_{i=1}^{n} p(x_i) \Lambda_i,$$

PROOF. According to the polynomial division algorithm, we have

$$p(x) = q(x) p_n(x) + r(x),$$

where $q(x)$ and $r(x)$ are a polynomials of degree at most $n - 1$; one can call this equation a *division equation*. We have

$$\int_a^b q(x) p_n(x) w(x)\, dx = 0$$

in view of the orthogonality relation; cf. formula (2) in Section 44 on Orthogonal Polynomials. Furthermore,

$$r(x) = \sum_{i=1}^{n} r(x_i) l_i(x) = \sum_{i=1}^{n} p(x_i) l_i(x)$$

according to the Lagrange interpolation formula; the second equation hold because $p_n(x_i) = 0$, and so $r(x_i) = p(x_i)$ according to the division equation above. Multiplying this equation by $w(x)$ and then integrating on (a, b), we obtain

$$\int_a^b p(x) w(x)\, dx = \int_a^b r(x) w(x)\, dx = \int_a^b \sum_{i=1}^{n} p(x_i) l_i(x) w(x)\, dx = \sum_{i=1}^{n} p(x_i) \Lambda_i.$$

---

[179] As we proved above, the x_i are all distinct, real, and they lie in the interval (a, b).

[180] In Section 5 on Lagrange interpolation, we defined l_i by the first equality. The second equality was proved in the form

$$l_i(x) = \frac{p(x)}{p'(x_i)(x - x_i)},$$

where $p(x) = \prod_{j=1}^{n} (x - x_j)$. It is easy to see that we have $p_n(x) = \gamma_n p(x)$ for this $p(x)$ (as $p_n(x)$ and $p(x)$ have the same zeros, they must agree up to a constant factor), the second equation above for l_i follows.

[181] This formula is also called the *Gauss-Jacobi Quadrature Formula*. The term *quadrature* in a synonym for integration, in that in quadrature one seeks a square having the area equal to a certain area (in the case of integration, the area under a curve). The term is somewhat old fashioned, but it survives in certain contexts.

The first equation here holds in view of the integral of $q(x)p_n(x)w(x)$ being zero, as shown above, and the third equation holds in view of the definition of the Christoffel numbers.

It is worth pointing out that $\Lambda_i > 0$. This is because the Gauss Quadrature Formula is applicable with the polynomial $l_k^2(x)$ $(1 \le k \le n)$, since this polynomial has degree $2n-2$. We have

$$\int_a^b l_k^2(x)w(x)\,dx = \sum_{i=1}^{n} l_k^2(x_i)\Lambda_i = \Lambda_k.$$

The second equation holds because $l_k(x_i) = \delta_{ki}$. The left-hand side here is of course positive, showing that $\Lambda_i > 0$ as claimed.

The Christoffel-Darboux formula can be used to evaluate Λ_i. Namely, we have

$$\begin{aligned} a_{n+1}p_n'(x_i)p_{n+1}(x_i)l_i(x) &= a_{n+1}\frac{p_n(x)p_{n+1}(x_i)}{(x-x_i)} \\ &= -a_{n+1}\frac{p_{n+1}(x)p_n(x_i) - p_n(x)p_{n+1}(x_i)}{(x-x_i)} = -\sum_{k=0}^{n} p_k(x)p_k(x_i); \end{aligned}$$

the first equation here holds in view of (1), the second equation because x_i is a zero of p_n (i.e., $p_n(x_i) = 0$), and the third one because of the Christoffel-Darboux formula. Multiplying both sides by $w(x)$ and integrating on the interval (a, b), we obtain

$$\begin{aligned} a_{n+1}p_n'(x_i)p_{n+1}(x_i)\Lambda_i &= a_{n+1}p_n'(x_i)p_{n+1}(x_i)\int_a^b l_i(x)w(x)\,dx \\ &= -\sum_{k=0}^{n} p_k(x_i)\int_a^b p_k(x)w(x)\,dx = -1. \end{aligned}$$

The first equality here holds in view of the definition of Λ_i. To see why the last one holds, note that $p_0(x)$ is a nonzero constant; hence we have $p_0(x) = p_0(x_i) \ne 0$, say. So,

$$\int_a^b p_k(x)w(x)\,dx = \frac{1}{p_0(x_i)}\int_a^b p_k(x)p_0(x)w(x)\,dx = \frac{1}{p_0(x_i)}\delta_{k0}.$$

Thus,

$$\Lambda_i = -\frac{1}{a_{n+1}p'(x_i)p_{n+1}(x_i)}. \tag{2}$$

46. THE CHEBYSHEV POLYNOMIALS

The Chebyshev[182] polynomials are defined as

$$T_n(\cos\theta) = \cos n\theta \qquad (n = 0, 1, 2\dots).$$

For example, we have

$$T_0(x) = 1, \quad T_1(x) = x, \quad \text{and} \quad T_2(x) = 2x^2 - 1; \tag{1}$$

[182] P. L. Chebyshev, a Russian mathematician of the 19th century. His name is often transliterated into French as Tchebichef; this is why these polynomials are usually denoted as T_n.

the last equation corresponds to the identity $\cos 2\theta = 2\cos^2\theta - 1$. The above formula indeed defines T_n as a polynomial, since we have

$$\cos n\theta + i\sin n\theta = (\cos\theta + i\sin\theta)^n = \sum_{k=0}^{n} \binom{n}{k} i^k \cos^{n-k}\theta \sin^k\theta.$$

where $i = \sqrt{-1}$; the first equation here holds according to de Moivre's formula, and the second one holds in view of the Binomial Theorem. Taking real parts on both sides, we have an equation for $\cos n\theta$. The real part on the right-hand side will only involve even values of k, i.e., even powers of $\sin\theta$; that is, integral[183] powers of $\sin^2\theta = 1 - \cos^2\theta$. This shows that $\cos n\theta$ is indeed a polynomial of $\cos\theta$.

Given arbitrary nonnegative integers m and n,

$$\int_0^\pi \cos m\theta \cos n\theta \, d\theta = \frac{1}{2}\int_0^\pi \big(\cos(m+n)\theta + \cos(m-n)\theta\big)\, d\theta.$$

The integral of both cosine terms is zero unless $m+n$ or $m-n$ is zero, in which case the integral of the corresponding term is π. Thus we obtain that

$$\int_0^\pi \cos m\theta \cos n\theta \, d\theta = \begin{cases} \frac{\pi}{2} & \text{if } m = n > 0, \\ \pi & \text{if } m = n = 0, \\ 0 & \text{if } m \neq n. \end{cases}$$

Noting that the substitution $\theta = \arccos x$ gives

$$d\theta = -\frac{dx}{\sqrt{1-x^2}},$$

i.e.,

$$\int_0^\pi \cos m\theta \cos n\theta \, d\theta = -\int_1^{-1} T_m(x)T_n(x)\frac{dx}{\sqrt{1-x^2}} = \int_{-1}^{1} T_m(x)T_n(x)\frac{dx}{\sqrt{1-x^2}}.$$

Therefore,

$$\int_{-1}^{1} T_m(x)T_n(x)\frac{dx}{\sqrt{1-x^2}} = \begin{cases} \frac{\pi}{2} & \text{if } m = n > 0, \\ \pi & \text{if } m = n = 0, \\ 0 & \text{if } m \neq n. \end{cases} \tag{2}$$

Thus, the polynomials T_n are orthogonal (but not orthonormal) on the interval (-1,1) with respect to the weight function $\frac{1}{\sqrt{1-x^2}}$.

The recurrence equations for the polynomials T_n can be written as

$$xT_0(x) = T_1(x)$$

and

$$xT_n(x) = \frac{1}{2}\big(T_{n+1}(x) + T_{n-1}(x)\big) \tag{2}$$

[183]The word *integral* here is the adjectival form of the word *integer*, and has nothing to do with the work *integration* (except in an etymological sense).

for $n \geq 1$. These equations are just translations of the equations

$$\cos\theta \cos 0\theta = \cos 1\theta,$$

and

$$\cos\theta \cos n\theta = \frac{1}{2}\big(\cos(n+1)\theta + \cos(n-1)\theta\big).$$

It is simpler to work out the Clenshaw's recurrence equations for calculating the sum[184]

$$f_n(x) = \frac{c_0}{2} + \sum_{k=1}^{n-1} c_k T_k(x)$$

directly than to use the worked-out form in the section on Orthogonal Polynomials: Put $y_{n+1} = y_n = 0$ and write

$$y_k = 2xy_{k+1} - y_{k+2} + c_k$$

for $k = n-1, n-2, \ldots, 1$. Expressing c_k from this equation, we have

$$f_n(x) = \frac{c_0}{2} + \sum_{k=1}^{n-1} (y_k - 2xy_{k+1} + y_{k+2}) T_k(x).$$

For m with $3 \leq m \leq n-1$ the sum of the terms involving y_m on the right-hand side is

$$y_m(T_m(x) - 2xT_{m-1}(x) + T_{m-2}(x)) = 0,$$

where the second equality holds in view of the recurrent equation (2). Thus, only the terms involving y_1 and y_2 make a contribution to the sum describing $f_n(x)$. Hence,

$$\begin{aligned} f_n(x) &= \frac{c_0}{2} + y_1 T_1(x) - 2xy_2 T_1(x) + y_2 T_2(x) = \frac{c_0}{2} + y_1 x - 2xy_2 x + y_2(2x^2 - 1) \\ &= \frac{c_0}{2} + xy_1 - y_2; \end{aligned}$$

for the second equation, cf. (1).

Assume $f(x)$ is a polynomial of degree at most n. Then we have

$$f_n(x) = \frac{c_0}{2} + \sum_{k=1}^{n-1} c_k T_k(x)$$

with

$$c_k = \frac{2}{\pi} \int_{-1}^{1} f(x) T_k(x) \frac{dx}{\sqrt{1-x^2}} \quad (0 \leq k < n); \tag{3}$$

the factor $2/\pi$ comes about as the reciprocal of the integral of T_k^2 (with respect to the weight function).[185] Since the degree of $f(x)T_k(x)$ is at most $2n-1$ for $k < n$, we can use the Gauss Quadrature Formula with the zeros of T_n to evaluate this integral.

[184] It is customary to take $\frac{c_0}{2}$. The reason is that the the integral of T_0^2 is twice the integral of T_k^2 for $k > 0$; cf. (2). This sum is called the *Dirichlet kernel*, and is usually denoted as $D_n(\theta)$.

[185] In case $k = 0$ this integral is π. We still have $2/\pi$ in (3) for $k = 0$, since we took $c_0/2$ in the expansion of $f_n(x)$.

We are going to use formula (2) of Section 45 on the Gauss Quadrature formula to evaluate Λ_i. For this, we first need the orthonormal Chebyshev polynomials t_n:

$$t_0(x) = \frac{1}{\sqrt{\pi}} T_0(x) \quad \text{and} \quad t_n(x) = \sqrt{\frac{2}{\pi}} T_n \qquad (n > 0).$$

The recurrence formula for the orthonormal Chebyshev polynomials can be written as follows:

$$\begin{aligned} x t_0(x) &= \frac{1}{\sqrt{2}} t_1(x), \\ x t_1(x) &= \frac{1}{2} t_2(x) + \frac{1}{\sqrt{2}} t_0(x), \\ x t_n(x) &= \frac{1}{2} t_{n+1}(x) + \frac{1}{2} t_{n-1}(x) \qquad (n \geq 2). \end{aligned} \tag{4}$$

Thus, formula (2) in Section 45 on the Gauss Quadrature Formula, we have

$$\frac{1}{\Lambda_i} = -\frac{1}{2} t_n'(x_i) t_{n+1}(x_i) = -\frac{1}{2} \cdot \frac{2}{\pi} T_n'(x_i) T_{n+1}(x_i) = -\frac{1}{\pi} T_n'(x_i) T_{n+1}(x_i) \qquad (n \geq 1),$$

where x_i is a zero of $T_n(x)$. For the first equation we have $a_{n+1} = 1/2$ for $n \geq 1$ according to (4) for the coefficient in the recurrence equation for the Chebyshev polynomial.[186] Writing $x = \cos\theta$ and $x_i = \cos\theta_i$, we have

$$T_n'(x) = \frac{\frac{d \cos n\theta}{d\theta}}{\frac{d \cos\theta}{d\theta}} = \frac{n \sin n\theta}{\sin\theta}.$$

Further, noting that $0 = T_n(x_i) = \cos n\theta_i$, we have

$$T_{n+1}(x_i) = \cos(n+1)\theta_i = \cos(n\theta_i + \theta_i) = \cos n\theta_i \cos\theta_i - \sin n\theta_i \theta_i = -\sin n\theta_i \sin\theta_i.$$

Hence

$$T_n'(x_i) T_{n+1}(x_i) = -n \sin^2 n\theta_i = -n(1 - \cos^2 n\theta_i) = -n.$$

Thus we have

$$\Lambda_i = \frac{\pi}{n}.$$

Hence, (3) becomes

$$c_k = \frac{2}{n} \sum_{i=1}^{n} f(x_i) T_k(x_i).$$

Noting that $x_i = \cos\theta_i$, where θ_i are the zeros of $\cos n\theta$ with $-1 < \theta < 1$,[187] we have $\theta_i = \pi \frac{i - 1/2}{n}$ for $i = 1, 2, \ldots, n$. Hence,

$$c_k = \frac{2}{n} \sum_{i=1}^{n} f\left(\cos \pi \frac{i - 1/2}{n}\right) \cos k\pi \frac{i - 1/2}{n}. \tag{5}$$

This formula can be used for numerical calculations.

[186] As shown by (4), we have $a_1 = 1/\sqrt{2}$; this is why we need to require $n \geq 1$ in the last displayed formula.

[187] As we saw in the discussion of orthogonal polynomials, the zeros of $T_n(x)$ all lie in the interval $(-1, 1)$; this interval corresponds to the interval $(0, \pi)$ for θ. Of course, we could chose another interval for θ that the mapping $\theta \mapsto \cos\theta$ maps to the interval $[-1, 1]$ in a one-to-one way.

Next we discuss a program incorporating formula (5). The header file `chebyshev.h` contains the functions to be described below:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define absval(x) ((x) >= 0.0 ? (x) : (-(x)))
#define square(x) ((x)*(x))

double *allocvector(int n);
void chebcoeffs(double (*funct)(double), double *c, int n);
double chebyshev(double x, double *c, int n);
double funct(double x);
```

The file `alloc.c` now only needs to define vectors; matrices are not needed:

```
#include "chebyshev.h"

double *allocvector(int n)
{
  double *b;
  b=(double *) malloc((size_t)((n+1)*sizeof(double)));
  if (!b) {
    printf("Allocation failure 1 in vector\n");
    exit(1);
  }
  return b;
}
```

The definition of the function `allocvector` is the same as before. The main calculations take place in the file `chebyshev.c`:

```
#include "chebyshev.h"
#define F(x) ((*fnct)(x))

void chebcoeffs(double (*fnct)(double), double *c, int n)
{
  const double pi=3.1415926535897931;
  int i,j;
  long double sum;
  double two_over_n, *fvector;
  fvector=allocvector(n);
  two_over_n=2.0/n;
  for (j=1;j<=n;j++) {
    fvector[j]=F(cos(pi*(j-0.5)/n));
  }
  for (i=0;i<n;i++) {
    sum=0;
    for (j=1;j<=n;j++)
      sum += fvector[j]*cos(pi*i*(j-0.5)/n);
    c[i]=two_over_n*sum;
  }
  free(fvector);
```

```
22 }
23
24 double chebyshev(double x, double *c, int n)
25 {
26   int i;
27   double fx, oldfx, newfx;
28   fx=0.; oldfx=0.;
29   for (i=n-1;i>=1;i--) {
30     newfx=2.*x*fx-oldfx+c[i];
31     oldfx=fx;
32     fx=newfx;
33   }
34   return x*fx-oldfx+0.5*c[0];
35 }
```

The function `chebcoeff` has a pointer to the function `*fnct`; this function in the program is referred to as `F` (cf. the `#define` on line 2). The second parameter, the vector `*c` will contain the coefficients c_k, and the last parameter, the integer `n`, corresponds to n in formula (5). The calculation in lines 5–22 is a straightforward implementation of formula (5). One point worth mentioning is that the inner product calculated in line 18 is accumulated in the variable `sum` of type `long double`. Once the coefficients c_k have been calculated, the value of $f_n(x)$ can be calculated with the aid of the Clenshaw recurrence equations. This is accomplished by the function `chebyshev` in lines 24–35. The file `funct.c` defines the function that is evaluated with the aid of Chebyshev polynomials:

```
1 #include "chebyshev.h"
2
3 double funct(double x)
4 {
5   double value;
6   value = 1.0/(1.0+x*x);
7   return(value);
8 }
```

This file defines the function `funct` as

$$f(x) = \frac{1}{1+x^2}.$$

This example is somewhat pointless, the real role of Chebyshev polynomials is to evaluate functions repeatedly when these functions are expensive to evaluate. That is, by calculating the coefficients in the Chebyshev expansion of $f(x)$ we can replace further evaluations with evaluations of the Chebyshev expansion. These programs were used with the calling program in the file `main.c`:

```
1 #include "chebyshev.h"
2
3 main()
4 {
5   int n=40;
6   double *c, x=.34753, y;
7   c=allocvector(n-1);
8   chebcoeffs(&funct,c,n);
9   y=chebyshev(x,c,n);
10  printf("     x          Chebyshev approx"
11    "        function value\n\n");
12  printf("%10.8f   %20.16f   %20.16f\n", x, y, funct(x));
```

```
13   free(c);
14 }
```

On line 8, the program `chebyshev` is called to calculate forty Chebyshev coefficients. Then the function is evaluated at a single location on line 9, and the result is compared to the actual value of the function. The printout of this program is as follows:

```
1      x          Chebyshev approx        function value
2
3 0.34753000     0.8922380723133866     0.8922380723133848
```

This shows good agreement between the actual function value and the one obtained by Chebyshev expansion. Chebyshev polynomials can be used to approximate a function with uniform accuracy on an interval, as opposed to Taylor polynomials, which approximate a function very well near the base point, less well away from this point. For example, if one finds that, in evaluating a power series, one needs to calculate too many terms near the endpoints of an interval, one can reduce the amount of calculation by using Chebyshev polynomials.

47. BOUNDARY VALUE PROBLEMS AND THE FINITE ELEMENT METHOD

In the discussion of a boundary value problem, it will be helpful to reformulate the problem in terms of the *calculus of variations*. As a background, we included a problem from classical mechanics; while this background can be skipped and the discussion of the differential equation below can be followed, some of the motivation would be lost.

Background from Lagrangian mechanics. The Lagrangian of a mechanical system is given as $L = T - V$, where T is the total kinetic energy, and V is the potential energy. These quantities are described in terms of the position variables $q_1, q_2, \ldots, q_f$, and in terms of their time derivatives, called generalized velocities, $\dot{q}_1, \dot{q}_2, \ldots, \dot{q}_f$. Here f is a positive integer, called the degree of freedom of the system. Explicit dependence on time t is also allowed. Assume that the system start at the point $A = (a_0, \ldots, a_f)$ at time $t = t_0$ and ends up at the point $A = (b_0, \ldots, b_f)$ at time $t = t_1$. In order to describe the evolution of the system, one wants to find the the position coordinates as a function of time, i.e., one wants to find functions $q_1(t), q_2(t), \ldots, q_f(t)$, with $q_1(t_0) = a_1$, $q_2(t_0) = a_2$, $\ldots$, $q_f(t_0) = a_f$, and $q_1(t_1) = b_1$, $q_2(t_1) = b_2$, $\ldots$, $q_f(t_1) = b_f$, According to Hamilton's principle, the evolution of of the system takes place in a way for which the integral

$$I = \int_{t_0}^{t_1} L(q_i(t), \dot{q}_i(t), t)\, dt$$

is minimal (sort of – more about this later); here $\dot{q}_i(t) = dq_i(t)/dt$. The above integral is called the *action* of the mechanical system.

As an example, in describing the motion of a planet of mass m around the sun of mass M, one may assume that the sun is always at the center of the coordinate system. This is, of course, not really true, but since the sun's mass is so much larger than that of the planet, the sun's movement is neglected. The planet is assumed to move in the xy plane, but instead of the usual Cartesian coordinates, its position will be described with polar coordinates ρ (the polar radius) and θ (the polar angle. Then the potential energy of the planet is

$$V = -\gamma \frac{Mm}{\rho},$$

and the kinetic energy is

$$T = \frac{m(\dot{\rho}^2 + \rho^2 \dot{\theta}^2)}{2},$$

so the Lagrangian is

$$L = \frac{m(\dot\rho^2 + \rho^2\dot\theta^2)}{2} + \gamma\frac{Mm}{\rho}.$$

This system has a degree of freedom $f = 2$, and one can take $q_1 = \rho$ and $q_2 = \theta$. In general, if one considers the motion of n point particles is three-dimensional space, we will have $n = 3f$ with three Cartesian coordinates for each particle, but, depending on the nature of the system, one may find it convenient to use some coordinates other than the Cartesian coordinates to describe the system.

In order to find the minimum of the above integral, one replaces the paths $q_i(t)$ by nearby paths $q_i(t) + \delta_i(t)$, where $\delta_i(t)$ is assumed to be small, and $\delta_i(t_0) = \delta_i(t_1) = 0$ (i.e., the two endpoints of the paths do not change. Then

$$\begin{aligned} L(q_i(t) + \delta_i(t), \dot q_i(t) + \dot\delta_i(t), t) &\approx L(q_i(t), \dot q_i(t), t) \\ &+ \sum_{k=1}^{f}\left(\frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial q_k}\delta_i(t) + \frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial \dot q_k}\dot\delta_i(t)\right), \end{aligned}$$

where the change in L is approximated by the total differential (this approximation implies that not only $\delta_i(t)$, but also $\dot\delta_i(t)$ is assumed to be small). So the change in the above integral can be described approximately as

$$\delta I = \delta\int_{t_0}^{t_1} L(q_i(t), \dot q_i(t), t)\,dt = \int_{t_0}^{t_1}\sum_{k=1}^{f}\left(\frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial q_k}\delta_i(t) + \frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial \dot q_k}\dot\delta_i(t)\right)\,dt,$$

In order for the integral I to be minimal, this first order approximation, or *variation*, δI to this integral must be zero, the same way as the derivative of a function at a place of minimum must be zero. In fact, Hamilton's principle states that δI is zero, not that I is minimum, and this is described by saying that the system evolves along a *stationary* path, i.e., evolves in a way that if this path is slightly changed then the above integral does not change in the first order (i.e., its change is much smaller than the change in the path).

Hamilton's principle gives a kind of teleological[188] formulation of Newtonian mechanics. In Newtonian mechanics, the particles move along paths determined by forces acting upon them. In the formulation given by Hamilton's principle, the particles have a goal of getting from point A to point B, and they choose a path which minimizes the action integral.

Hamilton's principle goes earlier teleological principles, such as Fermat't principle that light between two points travels in a way that takes the least amount of time. Fermat used this to explain the law of refraction. For example, light travels more slowly in water than in air. So if light starts out at a point in air to go to another point under water, then going along a straight line is not the fastest way for the light to get there; it is faster to travel a longer path in air, and a shorter path in water.

Pierre de Fermat (1601–1665) stated his principle in 1662. René Descartes (1596–1650) thought that the speed of light was infinite. The speed of light was first measured by the Danish astronomer Ole Rømer in 1676 by studying the orbits of the satellite Io of Jupiter. The first measurement of the speed of light on earth was carried out by the French physicist Hippolite Fizeau in 1849. So Fermat must have relied on some assumptions unverifiable at the time he stated his principle.

Using integration by parts, we have

$$\begin{aligned} \int_{t_0}^{t_1}\frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial \dot q_k}\dot\delta_i(t)\,dt &= \frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial \dot q_k}\delta_i(t)\Big|_{t=t_0}^{t=t_1} - \int_{t_0}^{t_1}\left(\frac{d}{dt}\frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial \dot q_k}\right)\delta_i(t)\,dt \\ &= -\int_{t_0}^{t_1}\left(\frac{d}{dt}\frac{\partial L(q_i(t), \dot q_i(t), t)}{\partial \dot q_k}\right)\delta_i(t)\,dt; \end{aligned}$$

[188]Teleology is the doctrine that final causes (i.e, goals or purposes) exist. In other words, the mechanical system, knowing in advance where it wants to go, finds a way to get there in such a manner that involves the least action.

the second equality holds since we have $\delta_i(t_0) = \delta_i(t_1) = 0$, and so the integrated-out part is zero. Taking this into account, the equation $\delta I = 0$ can be written as

$$\int_{t_0}^{t_1} \sum_{k=1}^{f} \left(\frac{\partial L(q_i(t), \dot{q}_i(t), t)}{\partial q_k} - \frac{d}{dt} \frac{\partial L(q_i(t), \dot{q}_i(t), t)}{\partial \dot{q}_k} \right) \delta_i(t)\, dt = 0.$$

This integral must be zero for all choices of small $\delta_k(t)$ for which $\delta_k(t_0) = \delta_k(t_1) = 0$, and this is possible only if

$$\frac{\partial L(q_i(t), \dot{q}_i(t), t)}{\partial q_k} - \frac{d}{dt} \frac{\partial L(q_i(t), \dot{q}_i(t), t)}{\partial \dot{q}_k} = 0 \qquad \text{for } k = 1, 2, \ldots, f. \tag{1}$$

To see this, choose $\delta_k(t)$ to be the of the same sign as the left-hand side of this equation. For example, one can choose

$$\delta_k(t) = \epsilon(t - t_0)(t_1 - t) \cdot \frac{\partial L(q_i(t), \dot{q}_i(t), t)}{\partial q_k} - \frac{d}{dt} \frac{\partial L(q_i(t), \dot{q}_i(t), t)}{\partial \dot{q}_k};$$

here, a small positive ϵ ensures that $\delta_k(t)$ is small, and the factor $(t-t_0)(t_1-t)$ (where $t_0 < t_1$) ensures that $\delta_k(t_0) = \delta_k(t_1) = 0$, Equations (1) are the Euler-Lagrange differential equations describing the mechanical system. They originate in a letter that Lagrange sent to Euler in 1755, when he was only 19 years old. The area of mathematics that seeks to minimize integrals of the above type is called the *calculus of variations*. The Lagrangian formulation of mechanics plays a very important part in modern physics, in that it led to a reformulation of quantum mechanics by Richard Feynman.

Problem

1. Write the Euler-Lagrange equations for the planetary motion example above.

Solution. The position variables are ρ and θ. For the variable ρ we have

$$\frac{\partial L}{\partial \rho} = m\rho\dot{\theta}^2 - \gamma \frac{Mm}{\rho^2},$$

and

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{\rho}} = \frac{d}{dt} m\dot{\rho} = m\ddot{\rho},$$

and the corresponding Euler-Lagrange equation is

$$\frac{\partial L}{\partial \rho} - \frac{d}{dt} \frac{\partial L}{\partial \dot{\rho}} = 0 : \qquad m\rho\dot{\theta}^2 - \gamma \frac{Mm}{\rho^2} - m\ddot{\rho} = 0.$$

For the variable θ we have

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{\theta}} = \frac{d}{dt} m\rho^2\dot{\theta} = 2m\rho\dot{\rho}\dot{\theta} + m\rho^2\ddot{\theta},$$

and

$$\frac{\partial L}{\partial \theta} = 0,$$

and the corresponding Euler-Lagrange equation is

$$\frac{\partial L}{\partial \theta} - \frac{d}{dt} \frac{\partial L}{\partial \dot{\theta}} = 0 : \qquad -2m\rho\dot{\rho}\dot{\theta} - m\rho^2\ddot{\theta} = 0.$$

A boundary value problem. Consider the differential equation

$$y'' + f(x) = 0 \tag{2}$$

on the interval $[0, 1]$, that is, one wants to find a function $y = y(x)$ satisfying the above equation under the conditions that $y(0) = a$ and $y'(1) = b$. This type of problem is called a boundary-value problem, since constraints on y are at the two end points (the boundary) of the domain on which we want to find y. It is easy to solve this problem explicitly; this is, however, a side issue here, since we want to discuss a numerical method applicable to handling a group of problems, some of which are not explicitly solvable.

In order to approach this problem, we will restate it with the aid of the calculus of variations. First, consider an arbitrary nice (continuous, differentiable, etc.) function $v(x)$ on $[0, 1]$ such that $v(0) = 0$. Then the above equation implies that

$$\int_0^1 (y''(x) + f(x))v(x)\,dx = 0.$$

Using integration by parts, we can see that

$$\int_0^1 y''(x)v(x)\,dx = y'(x)v(x)\Big|_{x=0}^{x=1} - \int_0^1 y'(x)v'(x)\,dx = bv(1) - \int_0^1 y'(x)v'(x)\,dx;$$

the second equation here holds because $v(0) = 0$ and $y'(1) = b$. Hence the above equation becomes

$$\int_0^1 y'(x)v'(x)\,dx = \int_0^1 f(x)v(x)\,dx + bv(1). \tag{3}$$

This equation needs to be satisfied for every nice function $v(x)$. (2) is called the strong statement of the problem, and (3) is the weak statement (or variational statement) of the same problem. Equations (2) and (3) are equivalent as long as we require that y'' be continuous at every point in the interval $[0, 1]$, since the argument used to obtain (3) from (2) can be reversed.[189]

We will not discuss here what kind of functions is $v(x)$ can be in equation (3), but one important point is that $v'(x)$ does not need to exist at every point. The argument leading from (2) to (3) will work for functions $v(x)$ that are continuous on $[0, 1]$ and differentiable everywhere with the exception of finitely many points, and at these points $v(x)$ is assumed only to have left and right derivatives, but these need not be the same; this is because one can break up the interval $[0, 1]$ into finitely many parts, do the integration by parts argument on each subinterval. When adding up the integrated-out terms, these will all cancel except those corresponding to the endpoints 0 and 1 of the interval $[0, 1]$.[190]

In order to obtain a numerical approximation of the solution of equation (3), in Galerkin's approximation method, we will restrict the functions y' and v to certain subspaces of functions. Namely, consider a partition

$$P : 0 = x_0 < x_1 < x_2 < \ldots < x_N = 1$$

[189] There are some subtle points in reversing the argument that will not be discussed here. An obvious difference is that equation (3) does not seem to require the existence of the second derivative of y. This is true, but if one relies on modern integration theory, then the existence of the second derivative is not required for (2), either. What is needed for (2) is that y' be *absolutely continuous*, – we will not define here what meant by that; in any case, this assumption allows for y'' not to exist at many points.

Solutions of equation (2) are called *strong solutions*, and solutions of equation (3) are called *weak solutions*. There are important situations in which certain differential equations originating in physics are only known to have weak solutions, and no strong solutions.

[190] Using modern integration theory, this argument can be carried out more elegantly.

of the interval $[0,1]$, and consider the functions $G_i(x)$ for i with $1 \le i \le N-1$ such that $G(x) = 0$ for x outside the interval (x_{i-1}, x_{i+1}), $G_i(x_i) = 1$, and $G(x)$ is linear on each of the intervals $[x_{i-1}, x_i]$ and $[x_i, x_{i+1}]$. That is,

$$G_i(x) = \begin{cases} 0 & \text{if } x < x_{i-1}, \\ \frac{x - x_{i-1}}{x_i - x_{i-1}} & \text{if } x_{i-1} \le x < x_i, \\ \frac{x - x_{i+1}}{x_i - x_{i+1}} & \text{if } x_i \le x < x_{i+1}, \\ 0 & \text{if } x_{i+1} < x. \end{cases}$$

In addition, we define $G_0(x)$ to be 1 at 0, linear on the interval $[0, x_1]$, and 0 for $x \ge x_1$; that is,

$$G_0(x) = \begin{cases} 1 & \text{if } x < 0, \\ \frac{x - x_1}{-x_1} & \text{if } 0 \le x < x_1, \\ 0 & \text{if } x_i \le x; \end{cases}$$

we defined $G_0(x)$ for all x, although $G_0 x)$ outside the interval $[0,1]$ is of no interest. Finally, we define $G_N(x)$ to be 1 at 1, linear on the interval $[x_{N-1}, 1]$, and 0 for $x < x_{N-1}$; that is,

$$G_N(x) = \begin{cases} 0 & \text{if } x < x_{N-1}, \\ \frac{x - x_{N-1}}{1 - x_{N-1}} & \text{if } x_{N-1} \le x < 1, \\ 1 & \text{if } 1 \le x; \end{cases}$$

we defined $G_N(x)$ for all x, although $G_N(x)$ outside the interval $[0,1]$ is of no interest. We will seek $y(x)$ in the form

$$y(x) = aG_0(x) + \sum_{j=1}^{N} c_j G_j(x) = \sum_{x=0}^{N} c_j G_j(x),$$

where $c_0 = a$ and the c_j for $j \ge 1$ are constants to be determined in such a way that equation (3) be satisfied for all $v(x)$ such that

$$v(x) = \sum_{k=1}^{N} \lambda_k G_k(x),$$

These equations ensure that $y(0) = a$ and $v(0) = 0$, as required. If we substitute these choices of $y(x)$ and $v(x)$, equation (3) becomes

$$\sum_{k=1}^{N} \lambda_k \sum_{j=0}^{N} c_j \int_0^1 G_k'(x) G_j'(x)\, dx = \sum_{k=1}^{N} \lambda_k \int_0^1 f(x) G_k(x)\, dx + b\lambda_N.$$

This equation must be satisfied for any choice of the λ_k; hence, the coefficient of λ_k on each side of the equation must agree. To say this another way, for each $i = 1, 2, \ldots, N$, pick $\lambda_k = \delta_{ik}$ to see that the coefficient of λ_k for $k = i$ on each side of the equation agrees. That is, writing

$$a_{ij} = \int_0^1 G_i'(x) G_j'(x)\, dx$$

and

$$b_j = \int_0^1 f(x) G_j(x)\, dx + b\delta_{jN},$$

we have the equations

$$\sum_{j=1}^{N} a_{ij} c_j = b_j - a a_{i0} \qquad (1 \le i \le N); \tag{4}$$

the term $-aa_{i0}$ on the right-hand side comes from the term for $j = 0$ in the sum on the left-hand side of the above equation, since we know that $c_0 = a$. These equations represent N linear equations for the unknowns $c_1, \ldots, c_N$, and by solving them we will obtain an approximate solution

$$y(x) = \sum_{j=0}^{N} c_j G_j(x)$$

of equation (3).

When considering methods to solve this equation, one might be guided by the following considerations. First, most coefficients on the left-hand side of (4) are zero. Indeed, $a_{ij} = 0$ unless $|i-j| \le 1$ in view of the definitions of the functions G_i. A matrix $(a_{ij})_{i,j=1}^{N}$ with this property is called a tridiagonal matrix (since all elements outside the main diagonal and the diagonals adjacent to it are zero).

Further, the matrix $(a_{ij})_{i,j=1}^{N}$ is positive definite. Here, a matrix A is called *positive definite* if for any nonzero column vector $\mathbf{z}$ the number $z^T Az$ is positive. To see that the matrix $A = (a_{ij})_{i,j=1}^{N}$ does indeed have this property, observe that, for $\mathbf{z} = (z_1, \ldots, z_n)^T$, we have

$$\mathbf{x}^T A\mathbf{x} = \sum_{i=1}^{N}\sum_{j=1}^{N} z_i a_{ij} z_j = \int_0^1 \sum_{i=1}^{N}\sum_{j=1}^{N} G_i'(x) z_i G_j'(x) z_j \, dx = \int_0^1 \Big(\sum_{i=1}^{N} G_i'(x) z_i\Big)^2 dx > 0;$$

strict inequality holds on the right-hand side, since not all z_i are zero (because $\mathbf{z}$ is not the zero vector). It is also immediately seen that the matrix $(a_{ij})_{i,j=1}^{N}$ is symmetric, i.e., that $a_{ij} = a_{ji}$.

The fact that this matrix is positive definite and symmetric is important, since it is known that a system of linear equations with a positive definite symmetric coefficient matrix can be solved by Gaussian elimination without pivoting.[191] Further, a such system of equations is also solvable by Gauss-Seidel iteration.[192] If one chooses to use Gaussian elimination, it is important to avoid pivoting, since pivoting would destroy the tridiagonal nature of the coefficient matrix; the fact that this matrix is tridiagonal allows one to store the matrix efficiently without storing the zero elements.

The method outlined above is an example of the *finite element method*. The finite element method can be used to handle many boundary values problems involving partial differential equations. In two dimensions, the domain over which the equation is considered is usually partitioned into triangles, and the function element considered is usually linear on these triangles.

48. THE HEAT EQUATION

Chebyshev polynomials of the second kind. With $x = \cos\theta$, the Chebyshev polynomials of the second kind are defined as

$$U_n(x) = U_n(\cos\theta) = \frac{\sin(n+1)\theta}{\sin\theta} \qquad (n \ge -1);$$

for $n = -1$, this equation gives $U_{-1} = \frac{\sin 0\theta}{\sin\theta} = 0$, and for $n = 0$ it gives $U_0 = \frac{\sin 1\theta}{\sin\theta} = 1$.

Using the de Moivre formula combined with the Binomial Theorem,

$$\cos(n+1)\theta + i\sin(n+1)\theta = (\cos\theta + i\sin\theta)^{n+1} = \sum_{k=0}^{n+1} \binom{n+1}{k} i^k \cos^{n+1-k}\theta \sin^k\theta.$$

[191] See the Corollary in Section 35, p. 162.)

[192] See the Theorem in Section 43, p. 226.)

where $i = \sqrt{-1}$. Taking imaginary parts of both sides, we obtain an equation for $\sin(n+1)\theta$. The imaginary part of the right-hand side will only involve odd values of k, i.e., only odd powers of $\sin\theta$. After dividing by $\sin\theta$, we can see that $\frac{\sin(n+1)\theta}{\sin\theta}$ involves only even powers of $\sin\theta$, that is only integral powers of $\sin^2\theta = 1 - \cos^2\theta$. This shows that $\frac{\sin(n+1)\theta}{\sin\theta}$ is indeed a polynomial of $\cos\theta$. Using the identity

$$\sin\alpha\cos\beta = \frac{1}{2}(sin(\alpha+\beta) + \sin(\alpha-\beta))$$

With $\alpha = (n+1)\theta$ and $\beta = \theta$, we can conclude that

$$\cos\theta\sin(n+1)\theta = \frac{1}{2}(sin(n+2)\theta + \sin n\theta)$$

for $n \geq 0$. This can also be written as

$$xU_n(x) = \frac{1}{2}(U_{n+1}(x) + U_{n-1}(x))$$

for $n \geq 0$; in addition, we have $U_{-1}(x) = 0$ and $U_0(x) = 1$. These are the recurrence equations for the Chebyshev polynomials of the second kind, It is worth recalling that these are the same recurrence equations that define the Chebyshev polynomials (sometimes also called Chebyshev polynomials of the first kind), with the exception that the latter satisfied a different equation for $n = 1$. It is not hard to show that these polynomials are orthogonal on the interval $[0.\pi]$ with respect to the weight function $\sqrt{1-x^2}$.

The heat equation. Heat conduction in a rod, with appropriate choice of the physical units, can be described by the equation

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, \tag{1}$$

subject boundary conditions $u(x,0) = f(x)$, $u(0,t) = u(1,t) = 0$. The function $u(x,y)$ for x, t with $0 \leq x \leq 1$ and $t \geq 0$ describes the temperature of a rod represented by the interval $[0,1]$ at point x and time t. The boundary value conditions indicate that at time zero the temperature of the rod at point x is $f(x)$, and the endpoints of the rod are kept at temperature 0 at all times. For consistency, we must of course have $f(0) = f(1) = 0$. In order to solve this equation numerically, the partial derivatives may be approximated by finite differences: choosing step sizes h and k for x and t, respectively, one may write

$$\frac{\partial u(x,t)}{\partial t} \approx \frac{u(x,t+k) - u(x,t)}{k} \tag{2}$$

and

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{u(x+h,t) - 2u(x,t) + u(x-h,t)}{h^2}; \tag{3}$$

the latter formula is related to the formula $f''(x) \approx \Delta^2 f(x)/(2h^2)$ with one-dimensional finite differences. Taking a $h = 1/N$, and taking a grid $x_m = mh$ and $t_n = nk$, one can approximate the differential equation above with the difference equation

$$\frac{u((m+1)h,nk) - 2u(mh,nk) + u((m-1)h,nk)}{h^2} = \frac{u(mh,(n+1)k) - u(mh,nk)}{k} \tag{4}$$

with the boundary conditions $u(mh,0) = f(mh)$ and $u(0,nk) = u(Nh,nk) = 0$ for m and n with $1 \le m \le N-1$ and $n \ge 0$. This equation can be solved explicitly for $u(m+1)h, nk)$:

$$u(mh,(n+1)k) = \left(1 - \frac{2k}{h^2}\right) u(mh,nk) + \frac{k}{h^2}(u((m+1)h,nk) + u((m-1)h,nk)).$$

Using this equation, $u(mh,(n+1)k)$ can be calculated for $n = 0$, $n = 1$, $n = 2$, ... and all m step by step.

Unfortunately, this calculation is numerically unstable unless $k \le h^2/2$. To see this, writing $\mathbf{v}_n = (u(h,nk), u(2h,nk), \dots, u((N-1)h,nk))^T$, the above equation can be written as

$$\mathbf{v}_{n+1} = A\mathbf{v}_n, \tag{5}$$

where, writing $s = k/h^2$, $A = (a_{ij})$ is the $(N-1) \times (N-1)$ matrix for which $a_{ii} = 1 - 2s$ where $1 \le i \le N-1$, $a_{i\,i-1} = a_{i-1\,i} = s$ where $2 \le i \le N-1$. With $\mathbf{v} = (v_0, \dots, v_{N-2})^T$, and writing $v_{-1} = v_{N-1} = 0$, the equation $Av = \lambda v$ can be written as

$$\frac{\lambda - 1 + 2s}{2s} v_k = \frac{1}{2}(v_{k-1} + v_{n+1}) \qquad (0 \le k \le N-2)$$

If we add the equation $v_0 = 1$ and temporary drop the equation $v_{N-1} = 0$, these equations uniquely determine v_k for $k = 1, 2, \dots, N-1$. In fact, these equations are the same as those defining the Chebyshev polynomial of the second kind at $(\lambda - 1 + 2s)/(2s)$, and so we must have

$$v_k = U_k\left(\frac{\lambda - 1 + 2s}{2s}\right) \qquad \text{for } k \text{ with } 0 \le k \le N-1.$$

Adding back the equation v_{N-1} now, this means that $U_{N-1}((\lambda - 1 + 2s)/(2s)) = 0$. With $\theta = \arccos x$, $T_{N-1}(x) = \sin N\theta / \sin\theta$, so $T_N(x) = 0$ if $\theta = k\pi/N$ for $k = 1, 2, \dots N-1$ (these are the values of k that place θ in the interval $[0,\pi]$, the range of the function arccos), i.e., we must have

$$\frac{\lambda - 1 + 2s}{2s} = \cos\frac{k\pi}{N},$$

that is,

$$\lambda = 1 - 2s + 2s\cos\frac{k\pi}{N} \qquad (k = 1, 2, \dots, N-1).$$

The calculation using equation (5) will accumulate errors if any of these eigenvalues has absolute value greater than one. Indeed, assume λ is the largest eigenvalue of A, associated with eigenvector $\mathbf{v}$. equation (5) can be written as $\mathbf{v}_n = A^n\mathbf{v}_0$; because of roundoff errors, at the kth step, a small component $c\mathbf{v}$ will appear as a part of the error in calculating $\mathbf{v}_k$. At the nth step, this error will be increased to $\lambda^{n-k}c\mathbf{v}$. As $\lambda^{n-k} \to \pm\infty$ in case of $|\lambda| > 1$, the error in the calculation will tend to infinity. As

$$\lim_{N\to\infty} \cos\frac{(N-1)\pi}{N} = -1,$$

we need $1 - 4s > -1$, i.e., $s < 1/2$ to guarantee that $|\lambda| < 1$. That is, this method of calculating $u(x,y)$ is unstable unless $s < 1/2$, i.e., $k < h^2/2$.

The requirement $k < h^2/2$ severely limits the usefulness of the above method, since h must be chosen small in order that the differences should closely approximate the derivatives, but then k must be too small for practical calculations. Instead of the forward difference in (2), one can use the backward difference

$$\frac{\partial u(x,y)}{\partial t} \approx \frac{u(x,t) - u(x,t-k)}{k}$$

to approximate the right-hand side of (1). By doing this, one obtains a method of solving equation (1) that is useful in practice. With this approximation, equation (1) becomes

$$\frac{u((m+1)h,nk)-2u(mh,nk)+u((m-1)h,nk)}{h^2}=\frac{u(mh,nk)-u(mh,(n-1)k)}{k}, \tag{6}$$

or else

$$-u((m+1)h,nk)+\left(\frac{h^2}{k}+2\right)u(mh,nk)-u((m-1)h,nk)=\frac{h^2}{k}u(mh,(n-1)k).$$

With a fixed $n>0$, for $m=1,2,\dots,N-1$ this represent a system of equations for the quantities $u(m,nk)$, assuming that $u(mh,(n-1)k)$ is already known (recall that $u(0,nk)=u(Nh,nk)=0$). One starts out with the boundary conditions $u(mh,0)=f(mh)$ and $u(0,nk)=u(Nh,nk)=0$, and then uses the above equation to determine $u(mh,nk)$ for $n=1,2,3,\dots.$ for m and n with $1\le m\le N-1$ and $n\ge 0$. This method is stable for all values of h and k; we will omit the proof of this.

The Crank-Nicolson Method. One can combine the forward difference method and the backward difference method by taking the average of equation (4) *with n replaced by $n-1$* and equation (6):

$$\frac{u((m+1)h,(n-1)k)-2u(mh,(n-1)k)+u((m-1)h,(n-1)k)}{2h^2}$$
$$+\frac{u((m+1)h,nk)-2u(mh,nk)+u((m-1)h,nk)}{2h^2}=\frac{u(mh,nk)-u(mh,(n-1)k)}{k}.$$

Similarly to the backward difference method, this can be rearranged as a system of equations for the quantities $u(\cdot,nk)$ when quantities $u(\cdot,(n-1)k$ are already known. One again obtains a method that is stable for all values of h and k; this method was invented by John Crank and Phyllis Nicolson.

BIBLIOGRAPHY

[AH] L. V. Atkinson–P. J. Harley, *An Introduction to numerical analysis with Pascal*, Addison-Wesley, Reading, Massachusetts, 1983.

[B] P. R. Beesack, *A general form of the remainder in Taylor's theorem*, Amer. Math. Monthly **73** (1966), 64–67.

[CB] S. D. Conte–Carl de Boor, *Elementary numerical analysis*, third edition, McGraw-Hill, New York–London-Paris, 1980.

[H] Thomas J. R. Hughes, *The finite element method. Linear static and dynamic finite element analysis*, Dover Puublications, Inc., Mineola, New York, 2000.

[KR] Brian W. Kernighan–Dennis M. Ritchie, *The C programming language*, second edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[M] Alfred J. Maria, *Taylor's formula with derivative remainder*, Amer. Math. Monthly **73** (1966), 67–68.

[Ma] Attila Máté, *The Cayley–Hamilton Theorem*, `http://www.sci.brooklyn.cuny.edu/~mate/misc/cayley_hamilton.pdf`.

[PTVF] W. H. Press–S. A. Teukolsky–W. T. Vetterling–B. P. Flannery, *Numerical recipes in C*, second edition, Cambridge University Press, Cambridge, England–New York, NY, USA–Oakleigh, Victoria, Australia, 1992.

[Ral] Anthony Ralston, *On differentiating error terms, jour Amer. Math. Monthly* **70** (1963), 187–189..

[RR] Anthony Ralston–Philip Rabinowitz, *A first course in numerical analysis*, second edition, Dover Publications, Mineola, New York, 2001.

[Ros] Maxwell Rosenlicht, *Introduction to Analysis*, Dover Publications, Inc., New York, 1968.

[Wiki] Wikipedia entry `http://en.wikipedia.org/wiki/Taylor%27s_theorem` as given on July 10, 2007.

LIST OF SYMBOLS

Page numbers usually refer to the place where the symbol in question was introduced. In certain cases, where important discussion concerning the symbol occurs at several places, the number of the page containing the main source of information is printed in italics.

SUBJECT INDEX

Page numbers in italics refer to the main source of information about whatever is being indexed. Often, they refer to the statement of a theorem, while the other page numbers listed refer to various applications of the same.