Norwegian University of Science and Technology
Department of Mathematics

MA2501 Numerical Methods Spring 2015

Solutions to exercise set 1

1 Consider the following two segments of pseudocode:

Program A:

Data: a vector $a = [a_0, a_1, \dots, a_n]$ of real numbers, a real number x;
Output: a real number y;
Initialization: $y \leftarrow a_0$;
for $k = 1$ **to** n **do**
  $y \leftarrow y + a_k x^k$;
end

Program B:

Data: a vector $a = [a_0, a_1, \dots, a_n]$ of real numbers, a real number x;
Output: a real number y;
Initialization: $y \leftarrow a_n$;
for $k = n-1$ **to** 0 **by** -1 **do**
  $y \leftarrow a_k + xy$;
end

a) What do these programs actually do?

b) In theory, both programs should yield the same result. Can they be expected to do so also numerically?

c) Which of the programs is usually preferable?

Possible solution:

The loop in the first program (obviously) gives the result

$$y = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n.$$

In the second program we obtain

$$y = a_0 + x\big(a_1 + x\big(a_2 + x\big(a_3 + \dots + x\big(a_{n-1} + x a_n\big)\cdots\big)\big)\big),$$

which can, again, be simplified to

$$y = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n.$$

Thus both programs provide the same result, if all operations are performed without rounding errors. Because rounding errors usually cannot be avoided, the *numerical* results can be expected to be different, though.

In order to answer the question, which of the programs is preferable, we first have to clarify what "preferable" actually means—and there are several possible interpretations:

1. Can one of the two programs be expected to yield more accurate results?
2. Is one of the two programs most probably faster?
3. Are there noticeable differences in memory usage?

Next we look at these points separately:

1. From the viewpoint of accuracy of the result, at first glance none of the two methods is *obviously* better. Rounding errors in the form of cancellation may occur in both programs in their main steps (either the calculation of $y + a_k y^k$ or $a_k + xy$).
2. The situation is different, however, if one counts the number of operations:

   Program B requires in each iteration one multiplication and one addition, totalling in n multiplications and n additions.

   For Program A, the total number of additions is again n, but the number of multiplications is larger. At first glance, the computation of x^k seems to require $k-1$ multiplications. This would amount to $0 + 1 + 2 + \ldots + n - 1 = n(n-1)/2$ multiplications. Exploiting the fact that, for instance, $x^4 = (x^2)^2$, this number can be decreased quite a bit. Even more, it would be possible (and very sensible) to keep the value x^k in the memory and to compute x^{k+1} in the next step by multiplying the stored value with x. The main code line would then be replaced by something like

   $$\begin{aligned} z &\leftarrow xz, \\ y &\leftarrow y + a_k z. \end{aligned}$$

   Still, this requires two multiplications in each step, leading to a total of $2n$ (or $2n-1$ if one discounts the unnecessary first one).
3. Apparently, memory usage is no real issue in both programs.

Roughly spoken, these considerations imply that the second program is almost twice as fast as the (optimal implementation of the) first program without sacrificing any accuracy. Thus it should in general be preferred.

Program B is usually known as *Horner's Algorithm* (see also Cheney and Kincaid, pp. 8 sqq.).

$\boxed{2}$ Solve the two linear systems

$$\begin{aligned} 11x_1 + 10x_2 + 14x_3 &= 1, \\ 12x_1 + 11x_2 - 13x_3 &= 1, \\ 14x_1 + 13x_2 - 66x_3 &= 1, \end{aligned} \qquad \text{and} \qquad \begin{aligned} 11x_1 + \quad 10x_2 + 14x_3 &= 1, \\ 12x_1 + 11.01x_2 - 13x_3 &= 1, \\ 14x_1 + \quad 13x_2 - 66x_3 &= 1. \end{aligned}$$

Also test what happens if the right hand side of the first equation is replaced by 1.001. Try to explain the results.

Possible solution:

The first system has the solution $(x_1, x_2, x_3) = (1, -1, 0)$; the second one the approximate solution $(x_1, x_2, x_3) \approx (-0.243, 0.1217, 0.0036)$. Thus, changing only one coefficient of the equations by less than one tenth of a percent completely changes the solution—note that even the sign pattern is different. Similarly, if we choose the right hand side to be $(1.001, 1, 1)$ then we obtain results of approximately $(0.4430, -0.3900, 0.0020)$ and $(0.0435, 0.0474, 0.0034)$. Again, the solution is *extremely* sensitive with respect to changes in the data. This shows that the linear systems are *ill-conditioned*.

This can become a problem if the right hand side (or the coefficients of the system) represents some real world data including measurement errors. In this case, even if the errors can be guaranteed to be less than 0.1%, the solution of the system is basically worthless.

We can gain some additional insight if we compute the condition numbers (in for example the Euclidean norm), using `cond` in MATLAB. The first system then has condition number 1.1×10^5, while the second has 1.4×10^4, so clearly the systems are very ill-conditioned. We would already suspect this from the observation that approximately 2/3 times the first row of the coefficient matrix and 1/3 times the third equals the second. It turns out the coefficient matrices for these systems becomes singular for $a_{22} \approx 11.0010846$.

$\boxed{3}$ Consider the floating point system with 3 significant digits and 2 decimal exponents, i.e. numbers have the form $\pm d_1.d_2d_3 \times 10^{d_4d_5-49}$ with $d_i \in \{0, 1, 2, \ldots, 9\}$ for $i = 1, 2, 3, 4, 5$ and $d_1 \neq 0$. We assume no tricks so we can not represent zero.

a) Prove that two different set of digits lead to two different numbers, i.e. that each machine number has a unique representation.

b) What is

- the smallest positive machine number?
- the smallest machine number strictly greater than one?
- the unit roundoff error/machine epsilon?
- the biggest possible number?

Possible solution:

a) Suppose the statement is false. Two different sets of digits leads to the same number. It's obvious that both representations must have the same sign, so assume without loss of generality they are both positive: $d_1.d_2d_3 \times 10^{d_4d_5-49}$ and $d_1^*.d_2^*d_3^* \times 10^{d_4^*d_5^*-49}$. If they are to represent the same number, we must have:

$$\frac{d_1.d_2d_3}{d_1^*.d_2^*d_3^*} = 10^{d_4^*d_5^*-d_4d_5}. \tag{1}$$

Suppose first that $d_4d_5 = d_4^*d_5^*$. Then clearly $d_4 = d_4^*$ and $d_5 = d_5^*$, and the right hand side of (1) equals 1. Then we must have $d_1.d_2d_3 = d_1^*.d_2^*d_3^*$, which can only be the case if $d_1 = d_1^*$, $d_2 = d_2^*$ and $d_3 = d_3^*$. This cannot be the case if the sets of digits are to be different.

Suppose now that $d_4d_5 \neq d_4^*d_5^*$. Then for the right hand side of (1) we have $10^{d_4^*d_5^*-d_4d_5} \leq 0.1$ or $10^{d_4^*d_5^*-d_4d_5} \geq 10$. However because $1 \leq d_1.d_2d_3 \leq 9.99$ and $1 \leq d_1.d_2d_3 \leq 9.99$ it follows that the left hand side of (1) satisfies the inequality

$$0.1 < \frac{1}{9.99} \leq \frac{d_1.d_2d_3}{d_1^*.d_2^*d_3^*} \leq 9.99 < 10$$

Thus it is not possible for the two numbers to be equal in this case as well.

We conclude that it is impossible for two numbers for two different sets of digits to lead to the same number.

What is

b)

the smallest positive machine number? Solution: 1.00×10^{-49}.

the smallest machine number strictly greater than one? Solution: 1.01.

the unit roundoff error/machine epsilon? Solution: 0.005.

the biggest possible number? Solution: 9.99×10^{50}.

$\boxed{4}$ **Cf. Cheney & Kincaid, Exercise 1.1.54.**

It is known that

$$\pi = 4 - 8\sum_{k=1}^{\infty} \frac{1}{16k^2 - 1}.$$

Thus, replacing the infinite sum by the finite sum

$$K_n = 4 - 8\sum_{k=1}^{n} \frac{1}{16k^2 - 1}$$

can be expected to give some approximation of π.

a) Estimate the size of the approximation error $E_n := |\pi - K_n|$ in dependence of the number of terms in the sum (assuming exact calculations).[1]

[1] Note that the approximation error can be very well estimated by a certain integral.

b) Assuming you compute K_n by the iteration $K_0 := 4$, $K_{k+1} := K_k - 8/(16k^2-1)$, provide an estimate of the quality of the best possible approximation of π when using double precision. Is it possible to improve the results with a different implementation of the same formula?

c) Verify your results using MATLAB.

Possible solution:

a) The n-th approximation error is

$$E_n = |\pi - K_n| = \left| 8 \sum_{k=n+1}^{\infty} \frac{1}{16k^2-1} \right| = 8 \sum_{k=n+1}^{\infty} \frac{1}{16k^2-1}.$$

Define now

$$f(x) := \frac{8}{16x^2-1}.$$

Then

$$\int_{n+1}^{\infty} f(x)\,dx < E_n < \int_{n+2}^{\infty} f(x)\,dx.$$

Thus a good estimate of the error is

$$E_n \approx \int_{n+1}^{\infty} f(x)\,dx.$$

Now we compute

$$\int \frac{8}{16x^2-1}\,dx = \int \frac{4}{4x-1} - \frac{4}{4x+1}\,dx = \log(4x-1) - \log(4x+1)$$

and therefore

$$E_n \approx \int_{n+1}^{\infty} \frac{8}{16x^2-1}\,dx = \log(4n+5) - \log(4n+3).$$

Now note that for large x we have

$$\log(x+2) - \log(x) \approx \frac{2}{x}$$

(which follows from a Taylor expansion of log). Thus

$$E_n \approx \frac{2}{4n+3} \approx \frac{1}{2n}.$$

b) Denote by ϵ the machine precision. Then the iterates won't change as soon as

$$\frac{8}{16k^2-1} \approx 2\epsilon$$

(the factor 2 on the right hand side comes from the fact that the limit is somewhere between 2 and 4), or

$$k \approx \frac{1}{2\sqrt{\epsilon}}.$$

At that point the approximation error will be about

$$E_k \approx \sqrt{\epsilon}$$

For double precision we obtain a predicted smallest possible error of around 10^{-8} with an iteration number $n \approx 5 \cdot 10^7$. Numerical experiments with MATLAB (cf. the program `pi_approx.m`) indicate that our predictions both concerning the iteration at which the smallest possible error occurs and also about that error are indeed very good.

It is possible to obtain better results by reversing the order of addition. That is, one defines the sequence $s_1 := 8/(16n^2 - 1)$ and $s_{j+1} := s_j + \frac{8}{16(n-j)^2-1}$ and then $K_n := 4 - s_n$. Doing so, one avoids the problem of adding numbers of extremely different size, which is the cause of the failure of the method proposed in the exercise. Numerical experiments with this method (cf. the program `pi_approx2.m`) confirm this assertion. Still, this is by no means an efficient method for approximating π.

5 Solve the following linear systems using Gaussian elimination without pivoting or report where the algorithm fails:

a)

$$\begin{aligned} x_1 - 5x_2 + x_3 &= 7, \\ 10x_1 + 20x_3 &= 6, \\ 5x_1 - x_3 &= 4. \end{aligned}$$

b)

$$\begin{aligned} x_1 + x_2 - x_3 &= 1, \\ x_1 + x_2 + 4x_3 &= 2, \\ 2x_1 - x_2 + 2x_3 &= 3. \end{aligned}$$

c)

$$\begin{aligned} 2x_1 - 3x_2 + 2x_3 &= 5, \\ -4x_1 + 2x_2 - 6x_3 &= 14, \\ 2x_1 + 2x_2 + 4x_3 &= 8. \end{aligned}$$

d)

$$\begin{aligned} x_2 + x_3 &= 6, \\ x_1 - 2x_2 - x_3 &= 4, \\ x_1 - x_2 + x_3 &= 5. \end{aligned}$$

Possible solution:

a) The solution is $(x_1, x_2, x_3) = (43/55, -347/275, -1/11)$.

b) The method breaks down at the second step.

c) The solution is $(x_1, x_2, x_3) = (109, 27, -66)$.

d) The method breaks down at the first step.

6 **Cf. Cheney and Kincaid, Computer Exercise 2.2.4.**

The *Hilbert matrix* of order n is the $n \times n$ matrix with entries

$$a_{ij} = \frac{1}{i+j-1} \qquad \text{for } 1 \leq i, j \leq n.$$

It is a classical example of an invertible but ill-conditioned matrix.

a) Write a MATLAB program that constructs, for given $n \in \mathbb{N}$, the Hilbert matrix of order n.

b) Define a vector $b \in \mathbb{R}^n$ setting $b_i = \sum_j a_{ij}$. Then the solution of the linear system $Ax = b$ is the vector x with entries $x_i = 1$. Does this also hold numerically in the case where A is the Hilbert matrix of some moderate order (say $2 \leq n \leq 15$)?

Possible solution:

There is some code to play with on the webpage concerning different possibilities for the construction of A. The vector b in the second part of the exercise can in MATLAB be easily defined as `b=sum(A,2)` (the second argument of this command means that one sums the elements of the matrix A along its second dimension). For $n \leq 12$ the numerical results of the calculation `A\b` are somehow close to the true solution; for $n \geq 13$ they are not.