МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2006
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2006
Proceedings of the Thirty Fifth Spring Conference of
the Union of Bulgarian Mathematicians
Borovets, April 5–8, 2006

BAD AND GOOD COMPUTATIONAL PRACTICES (WITH MATLAB EXAMPLES)*

Mihail Konstantinov, Petko Petkov

We consider a number of bad and good computational practices. The bad practices may produce completely erroneous answers when the corresponding computational schemes are implemented in a finite arithmetic. The paper is intended to help teaching good computational practices in schools and universities.

1. Introduction.

In a recent paper [11] we discussed 13 myths and pitfalls in numerical analysis. Before that this subject had been treated in [17] and [5]. In this paper we consider a number of bad and good computational practices. The bad practices may produce completely erroneous answers when the corresponding computational schemes are implemented in a finite arithmetic. Unfortunately, bad computational algorithms are often taught in schools and even universities without any warning for possible consequences.

In particular, we consider the computational practices in (i) solving linear systems of algebraic equations, (ii) determination of the rank of a matrix (invertibility of a matrix in particular), (iii) solving nonlinear algebraic equations, (iv) computation of the eigenvalues of a matrix, and (v) differentiation and integration of a function. Our considerations are illustrated by MATLAB[1] examples. For more details about MATLAB see [1] and `http://www.mathworks.com/products/matlab`.

2. Floating–point computations. Consider the computational problem $Y = f(X)$, where X is the *data* and Y is the *result*. Usually X and Y are elements of finite dimensional normed spaces and the function f is at least continuous in the neighborhood of a given argument X. In many important cases this function is even differentiable. Further on we suppose that near X it is fulfilled that

$$\|f(X_1) - f(X_2)\| \leq L\|X_1 - X_2\|, \tag{1}$$

where $\|\cdot\|$ denotes the corresponding norm. The *Lipschitz constant* $L \geq 0$ is also known as the *absolute condition number* of the computational problem $Y = f(X)$. More precisely, the condition number $L = L(X, r)$ is defined as the supremum of $\|f(X+\Delta)-f(X)\|/\|\Delta\|$ over all $\Delta \neq 0$ such that $\|\Delta\| \leq r$.

First we recall some basic facts about finite precision computations. Suppose that the problem has to be solved in a machine arithmetic which operates with a finite set $\mathbb{M} \subset \mathbb{R}$

*2000 Mathematics Subject Classification: 65–01.

[1] MATLAB is a trademark of MathWorks, Inc.

of machine numbers. For definiteness we consider a floating–point arithmetic (FPA) with rounding unit $\mathbf{u} \in \mathbb{M}$ and standard range $R := [m, M]$, where $m \in \mathbb{M}$ is the minimum positive and $M \in \mathbb{M}$ is the maximum positive number that may be represented in FPA. According to the IEEE Standard [7] we have $\mathbf{u} \simeq 1.1 \times 10^{-16}$, $m \simeq 4.9 \times 10^{-324}$ and $M \simeq 1.8 \times 10^{308}$.

Real numbers $x \in (-m, m)$ are rounded to zero and this is called *underflow*. Numbers with $|x| \in R$ are rounded to the nearest machine number $\mathrm{fl}(x)$ (with a rule to break ties if x is in the middle between two machine numbers). Finally, numbers with $|x| > M$ cannot be represented in FPA. This phenomenon is called *overflow* and should be avoided. Moreover, for $|x| \in R$ we have

$$\mathrm{fl}(x) = x(1+\varepsilon),\ |\varepsilon| \le \mathbf{u}.$$

According to the *main hypothesis of FPA*, if $\circ$ denotes an arithmetic operation and $M^{-1} \le |x|, |y|, |x \circ y| \le M$ then the computed value $\mathrm{fl}(x \circ y)$ of $x \circ y$ satisfies

$$\mathrm{fl}(x \circ y) = (x \circ y)(1+\delta),\ |\delta| \le \mathbf{u}.$$

It must be stressed that *in FPA the arithmetic association rule as well as the distributive rule may be violated*. This may be a surprise for some of the students and here a number of well designed examples may be useful, see [9].

A very dangerous operation in FPA is the subtraction $x - y$ of close approximate numbers x and y. If $2y \ge x \ge y \ge 0$ then the computed result $\mathrm{fl}(x-y)$ is obtained without rounding errors. Nevertheless, even if the operands x and y are known with high accuracy, the computed result may be much less accurate due to the loss of significant most left digits. This phenomenon is known as *cancellation*, or even *catastrophic cancellation*. Let for example $x = 0.123456789\xi$ and $y = 0.123456789\eta$ be two (very accurate) approximate numbers with 9 true significant digits $1, 2, \ldots, 9$ and let their tenth digits ξ, η be uncertain. Then the difference $z = \pm 0.000000000|\xi - \eta|$ will be computed with no rounding errors in FPA but it will not contain a true significant digit!

A computational algorithm for the computation of $Y = f(X)$ is a finite sequence of arithmetic operations that produces the answer $\widehat{Y} = \widehat{f}(X)$. Here the expression $\widehat{f}(X)$ depends not only on X but also on the parameters $\mathbf{u}$, m and M of FPA. An algorithm is said to be *numerically stable* if the computed result $\widehat{Y}$ is close to the exact result of a near problem in the sense that

$$\|\widehat{Y} - f(\widehat{X})\| \le C_1 \mathbf{u} \|Y\|,\ \|\widehat{X} - X\| \le C_2 \mathbf{u} \|X\|, \tag{2}$$

where $C_1, C_2 \ge 0$ are constants depending only on the computational algorithm. In practice, the inequalities (2) are fulfilled within a tolerance of order $\mathbf{u}^2$.

Relations (1) and (2) yield an overall accuracy estimate for the computed result as follows. We have

$$\begin{aligned} \|\widehat{Y} - Y\| &= \|\widehat{Y} - f(X)\| = \|\widehat{Y} - f(\widehat{X}) + f(\widehat{X}) - f(X)\| \\ &\le \|\widehat{Y} - f(\widehat{X})\| + \|f(\widehat{X}) - f(X)\| \\ &\le C_1 \mathbf{u} \|Y\| + L\|\widehat{X} - X\| \le C_1 \mathbf{u}\|Y\| + LC_2 \mathbf{u}\|X\|. \end{aligned}$$

For $X \ne 0$ and $Y \ne 0$ we obtain a bound for the relative error in the computed solution

$$\frac{\|\widehat{Y} - Y\|}{\|Y\|} \le \mathbf{u}(C_1 + C_2 C_{\mathrm{rel}}), \tag{3}$$

where

$$C_{\rm rel} = L\frac{\|X\|}{\|Y\|}$$

is the *relative condition number* of the computational problem $Y = f(X)$. The estimate (3) reveals very clearly the three main factors determining the accuracy of the computed solution: (i) the machine arithmetic (through the constant $\mathbf{u}$ and implicitly through the avoidance of over– and underflows), (ii) the properties of the computational problem (through the constant L) and the properties of the computational algorithm (through the constants C_1 and C_2). Frequently, from heuristic reasons, it is assumed that $C_1 = 0$ and $C_2 = 1$. In this case the heuristic bound takes the form

$$\frac{\|\widehat{Y} - Y\|}{\|Y\|} \leq \mathbf{u}C_{\rm rel}. \tag{4}$$

The estimate (3) (and (4) in particular) correspond to a *good computational practice*. According to the heuristic *rule of thumb* if $\mathbf{u}C_{\rm rel} < 1$ then one may expect about $-\lg(\mathbf{u}C_{\rm rel})$ true decimal digits in the computed solution.

In cases when the actual relative error is much larger than $\mathbf{u}(C_1 + C_2C_{\rm rel})$ or $\mathbf{u}C_{\rm rel}$ we have an example of a *bad computational practice*. Bad computational practices often occur as an attempt to translate directly a classical computational scheme (usually invented before the computer era, i.e. before the year 1946) into a computational algorithm without taking into account the particularities of the machine arithmetic.

The fundamental constants of FPA may be found by the MATLAB commands `realmax` (finds M), `realmin` (finds $4/M$) and `eps` (finds $2\mathbf{u}$). The constant m may be calculated by the command `eps*realmin`.

3. Determinants. Let A be an $n \times n$ matrix over the field $\mathbb{K}$ (the reader may accept that $\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{C}$) with columns A_k and elements a_{ik}. We recall that the *determinant* $\det : \mathbb{K}^{n\times n} \to \mathbb{K}$ is a scalar function of a matrix argument which is linear in each column of the argument, changes its sign when two columns of the argument are interchanged, and satisfies $\det(I) = 1$, where I is the identity matrix.

Determinants provide an useful tool in matrix analysis but their computation may be a difficult task in FPA, especially if one tries to use the expression

$$\det(A) = \sum_{\mathbf{j}\in\Pi_n} \operatorname{sign}(\mathbf{j}) a_{1,j_1} a_{2,j_2} \cdots a_{n,j_n}, \tag{5}$$

where $\mathbf{j}$ is the permutation $k \mapsto j_k$, Π_n is the set of permutations of order n and $\operatorname{sign}(\mathbf{j}) = \pm 1$ is the sign of the permutation $\mathbf{j}$.

The use of the sum (5) is a catastrophic way to find the determinant. Indeed, this sum contains in general $n!$ terms and the calculation of each terms requires $n - 1$ multiplications. Hence we have a total of $m_n := (n - 1)n!$ multiplications. Using the approximate Stirling formula for $n!$, namely

$$m_n \simeq (n - 1)\left(\frac{n}{e}\right)^n \sqrt{2\pi n},$$

or a computer system we may estimate the time necessary for the computation of the determinant of a general matrix by the formula (5). Let for example we have a (relatively fast) computer which can perform 10^9 (one billion) floating point operations (FLOPS) per second, or 3.1536×10^{17} FLOPS per year. Then the computation of the determinant of

a general 25×25 matrix will require about 10^{10} (ten billion) years which is approximately the age of the Universe.

Of course, there are much more reliable ways to compute the determinant of a matrix. For instance, using the LU decomposition $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with diagonal elements, equal to 1, and $U = [u_{ik}]$ is an upper triangular matrix, we obtain $\det(A) = \varepsilon u_{11} u_{22} \cdots u_{nn}$, where $\varepsilon = 1$ if the matrix P corresponds to an even permutation and $\varepsilon = -1$ if P corresponds to an odd permutation. Another decompositions, e.g. the QR decompositions $A = QRP$ with partial pivoting, may also be used for the computation of $\det(A)$, where Q is an orthonormed matrix, R is an upper triangular matrix and P is a permutation matrix.

Determinants are calculated in MATLAB by the command `det`.

4. Linear algebraic equations. Consider the linear vector algebraic equation

$$AY = B, \ \ A = [A_1, A_2, \ldots, A_n] = [a_{ik}] = [A(i,k)], \tag{6}$$

where A is an $n \times n$ invertible matrix with columns A_k and entries a_{ik}, and B is an n–vector with entries b_k. Equation (6) may also be written as

$$B = \sum_{i=1}^{n} y_i A_i.$$

The $n^2 + n$ elements of A and B form the vector of the data, while Y is the solution n–vector with elements y_i. Formally, the unique solution of (6) is given by $Y = A^{-1}B$. Of course, the computation of Y in this way is not a good idea.

Another bad computational practice here is to use the so called *Cramer formulae*. The latter are easily derived as follows. Denote by $A(B\colon k)$ the matrix A with its k–th column replaced by the vector B and let E_k be the k–th column of the identity $n \times n$ matrix I. Since $AE_k = A_k$ we may rewrite equation (6) as $AI(Y\colon k) = A(B\colon k)$. Taking determinants from both sides of this equality we get the Cramer formulae $y_k \det(A) = \det(A(B\colon k))$ which yield (for $\det(A) \neq 0$) the expressions

$$y_k = \frac{\det(A(B\colon k))}{\det(A)}, \ \ k = 1, 2, \ldots, n.$$

Another simple way to derive these formulae is to observe that

$$\begin{aligned} \det(A(B\colon k)) &= \det\left(A\left(\sum_{i=1}^{n} y_i A_i \colon k\right)\right) = \sum_{i=1}^{n} y_i \det(A(A_i\colon k)) \\ &= y_k \det(A(A_k\colon k)) = y_k \det(A). \end{aligned}$$

Here we have used the fact that $\det(A(A_i\colon k)) = \delta_{ik} \det(A)$, where δ_{ik} is the Kronecker delta.

As it follows from the results presented in Section 3, the use of Cramer formulae is a very bad way to solve the linear algebraic equation (6) in the general case. Instead, one may use a decomposition of A, say the QR decomposition $A = QR$, where Q is an $n \times n$ matrix with orthonormed columns ($Q^{\mathrm{H}}Q = I$) and $R = [r_{ij}]$ is an upper triangular matrix with nonzero diagonal elements r_{ii}. The QR decomposition is obtained via a finite numerically stable algorithm. After that the vector $Y = [y_i]$ is computed by back substitution

$$y_n = \frac{c_n}{r_{nn}},\ y_{n-1} = \frac{c_{n-1} - r_{n-1,n} y_n}{r_{n-1,n-1}}, \ldots, y_i = \frac{c_i - \sum_{j=i+1}^{n} r_{ij} y_j}{r_{ii}},\ i = n-1, n-2, \ldots, 1.$$

Another popular way to solve (6) is via the Gauss elimination method (or LU decomposition) with partial pivoting. Both QR and LU decompositions are not very expensive and require about $O(n^3)$ FLOPS. Singular value decomposition (see Section 5) may also be used but this is an expensive way to solve a linear algebraic equation requiring about $O(n^4)$ FLOPS.

In certain cases the computations of the inverse matrix $Z := A^{-1} = [Z_1, Z_2, \ldots, Z_n]$ is necessary. Here the column Z_k of Z may be found as the solution of the linear equation $AZ_k = E_k$, where E_k is the k–th column of the identity $n \times n$ matrix. In turn, the equations $AZ_k = E_k$ are solved by LU or QR decomposition or even by singular value decomposition.

Equation (6) is solved in MATLAB by the command `Y = A\B`. The same command solves the linear least squares problem (LLSP) $AY \simeq B$ which consists in the minimization of $\|AY - B\|$ (in the 2–norm) under the additional requirement that $\|Y\|$ is also minimum. Note that here it is *not recommended* to form the normal system $A^{\mathrm{H}}AY = A^{\mathrm{H}}B$ due to possible roundings and cancellations in the computation of $A^{\mathrm{H}}A$ and $A^{\mathrm{H}}B$. Instead, the LLSP is solved by QR or singular value decompositions where these effects are minimized.

5. Rank determination. Consider a general matrix $A \in \mathbb{K}^{m \times n}$. There are several ways to define the rank of A. First of all we assume that $\mathrm{rank}(A) = 0$ if and only if A is the zero matrix. Then we may define the $\mathrm{rank}(A)$ as the maximum size of the nonzero minors of A. This leads to a bad computational procedure since the computation of determinants in finite precision arithmetic should be avoided. In particular, the check for invertibility of the matrix $A \in \mathbb{K}^{n \times n}$ using the inequality $\det(A) \neq 0$ must also be avoided.

Next, $\mathrm{rank}(A)$ may be defined as the number of linearly independent columns (or rows) of A. This leads to a better way to compute the rank since the number of linearly independent rows may be detected using the LU or QR decompositions of A. In particular, if $r := \mathrm{rank}(A) < m$ then the QR decomposition of A is

$$A = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} = Q_1 R_1,$$

where $R_1 \in \mathbb{K}^{r \times n}$ is an upper triangular matrix with $\mathrm{rank}(R_1) = r$. Thus the rank of A can be revealed as the number r of the first nonzero rows (forming the matrix R_1) of the factor R in the QR decomposition $A = QR$. For improving the accuracy of floating point computations a QR decomposition $A = QRP$ with column pivoting is usually used, where P is a permutation matrix.

Another way to define $\mathrm{rank}(A)$ is as the number of nonzero singular values of A. The singular values may be retrieved by the *singular value decomposition (SVD)*

$$A = USV^{\mathrm{H}},\ S = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r, 0) \in \mathbb{K}^{m \times n},$$

of A, where $U \in \mathbb{K}^{m \times m}$ and $V \in \mathbb{K}^{n \times n}$ are orthonormed matrices (orthogonal in the real case and unitary in the complex case) and $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$. The quantities $\sigma_k = \sigma_k(A)$ are the *singular values* of A. They are also the positive eigenvalues of the

matrices $A^{\mathrm{H}}A$ and AA^{H}.

Using the SVD we may define the *Moore–Penrose pseudoinverse* $A^{\dagger} \in \mathbb{K}^{n\times m}$ of $A \in \mathbb{K}^{m\times n}$ as $A^{\dagger} := 0_{n\times m}$ if $A = 0_{m\times n}$, and

$$A^{\dagger} := VS^{\dagger}U^{\mathrm{H}},\ S^{\dagger} := \operatorname{diag}\left(\sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_r^{-1}, 0\right) \in \mathbb{K}^{n\times m},$$

if $A \neq 0$. Note that $VS^{\dagger}U^{\mathrm{H}}$ is *not* necessarily the SVD of the matrix $A^{\dagger}$ (explain why!).

The determination of $\operatorname{rank}(A)$ as the number r of (positive) singular values of A is the most reliable computational procedure for this purpose. However, there is a principal difficulty in the rank determination by finite precision arithmetic. The reason is that the rank is a very sensitive characteristic of a matrix in the following sense. Suppose first that $r < p := \min\{m, n\}$. Then an arbitrary small perturbation δA in A may increase the rank to p in the sense that $\operatorname{rank}(A + \delta A) = p$. On the other hand perturbations with $\|\delta A\|_2 < \sigma_r(A) = 1/\|A^{-1}\|_2$ cannot decrease the rank.

The computation of the rank of a general matrix $A \in \mathbb{K}^{m\times n}$ of even moderate size cannot be done "by hand". But when the matrix is written in the computer memory it is rounded to the nearest machine matrix $\mathrm{fl}(A)$, where $\|\mathrm{fl}(A) - A\|$ is of order $\mathbf{u}\|A\|$. If the initial matrix A is not of a full rank, then the rounded matrix may well be of full rank p. In addition, if the initial matrix arises in a mathematical model of a real system or a process, then it may be subject to measurement and/or parametric uncertainties. In both cases the "theoretical rank" of the matrix is a unknowable quantity. The above considerations show that the rank of a general matrix of moderate size may not be revealed in a reasonable way. This leads to the concept of numerical rank of a matrix.

Let $\widehat{\sigma}_1 \geq \widehat{\sigma}_2 \geq \cdots \geq \widehat{\sigma}_r > 0$ be the singular values of $A \in \mathbb{K}^{m\times n}$ computed in FPA with rounding unit $\mathbf{u}$. If among these eigenvalues there are some very small (compared with $\|A\|$) then it may be expected that they are due to rounding errors and not to the rank of the initial matrix A. When the singular values are grouped in two well defined clusters: large (of the size of $\|A\|$) and small (of the size of $\mathbf{u}\|A\|$ or less) the problem seems easy and one may decide that the rank is the number of large singular values.

The problem of rank determination is much more difficult when the singular values form a sequence with uniformly decreasing members, e.g. $\sigma_k \simeq \|A\|10^{1-k}$. Here we must eventually delete the singular values which are not inherent for the problem and are due to the effects of rounding during the decomposition. For this purpose we define the concept of numerical rank of a matrix.

Let $\tau > 0$ be a (small) numerical threshold. Then the matrix A is of *numerical rank* $\rho = \rho(A, \tau)$ within a threshold τ when it has exactly ρ singular values larger than τ, i.e. when $\sigma_\rho > \tau$ and $\sigma_{\rho+1} \leq \tau$ (if $\rho \leq p - 1$). If $\sigma_1 \leq \tau$ the numerical rank is assumed as zero.

Usually the threshold is taken as $\tau = C\mathbf{u}\|A\|$, where $C = C(p)$. In practice we have only the computed singular values $\widehat{\sigma}_k$. That is why the numerical rank in machine arithmetic is defined as the number of quantities $\widehat{\sigma}_k$ which are larger than $C\mathbf{u}\widehat{\sigma}_1$ (we recall that $\|A\|_2 = \sigma_1$). Here usually the constant C is taken as 1 or p. There are also some more conservative rank estimators in which it is assumed that $C = p^2$.

In accordance with the concept of numerical rank we may define the numerical pseudoinverse of a matrix $A \in \mathbb{K}^{m\times n}$ with SVD $A = USV^{\mathrm{H}}$. Namely, *the numerical pseudoinverse* $A_\tau^{\dagger} \in \mathbb{K}^{n\times m}$ *of* A *within a threshold* $\tau > 0$ is defined as $A_\tau^{\dagger} = 0_{n\times m}$ if

$\sigma_1 \leq \tau$, and

$$A_\tau^\dagger := V S_\tau^\dagger U^{\mathrm{H}},\ S_\tau^\dagger := \operatorname{diag}\left(\sigma_1^{-1}, \sigma_2^{-1}, \ldots, \sigma_\rho^{-1}, 0\right) \in \mathbb{K}^{n\times m},$$

if $\sigma_1 > \tau$. Here $\rho = \rho(A, \tau)$ is the numerical rank of A. In practice we choose τ as $C\mathbf{u}\widehat{\sigma}_1$, where $C = p^s$, $s = 0, 1, 2$.

The rank determination and the computation of pseudoinverse matrices based on SVD and on the concepts of numerical rank and numerical pseudoinverse are examples of good computational practices. The rank determination using QR decomposition with column pivoting is also a good computational algorithm.

The rank of the matrix A is found in MATLAB by the command `rank(A)`. The command `rank(A,tol)` computes the numerical rang of A within a user defined tolerance `tol`. This is the number of computed singular values of A that are larger than `tol`.

The vector of singular values of A is calculated by `svd(A)`, while the command `[U,S,V] = svd(A)` computes the matrices in the singular value decomposition $A = USV^{\mathrm{H}}$ of A.

The pseudoinverse $A^\dagger$ and the numerical pseudoinverse within a tolerance `tol` are computed by the commands `pinv(A)` and `pinv(A,tol)` respectively.

6. Algebraic equations. As it is well known algebraic equations

$$p(x) := a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = 0,\ a_n \neq 0, \tag{7}$$

over $\mathbb{R}$ or $\mathbb{C}$ may be solved in general by explicit formulae for $n \leq 4$. However, the direct use of explicit formulae for $n = 2, 3, 4$ may lead to a severe loss of accuracy in the computed result due to cancellation errors. In the most simple case $n = 2$ the solution is given by

$$x_{1,2} = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0a_2}}{2a_2}.$$

Paradoxically, this bad computational approach is sometimes taught as the only way to solve the quadratic equation. Moreover, the calculation of $x_{1,2}$ in this way is sometimes recommended as an exercise in informatics (e.g. when the concept of an algorithm is presented to the students). Thus a bad computational practice is firmly fixed in the students heads.

There are several reliable approaches to solve equation (7). One of them is through the eigenvalues of the corresponding associated matrix. First we may assume that $a_n = 1$ (otherwise we may divide the left hand side of the equation by a_n). Next we form the matrix

$$A_p := \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ -a_0 & -a_1 & -a_2 & \ldots & -a_{n-1} \end{bmatrix},$$

associated with the polynomial $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$. Finally we apply the QR algorithm or some other sophisticated scheme (as the Kagström–Ruhe algorithm [8]) to compute the eigenvalues of A_p which are exactly the roots of the polynomial p.

In MATLAB the roots of the polynomial p in (7) are found by the command `roots(v)`, where $v = [a_n, a_{n-1}, \ldots, a_0]$ is the $(n+1)$–vector of the coefficients of p.

7. Eigenvalues of a matrix. The eigenvalues of the $n \times n$ matrix $A = [a_{ij}]$ are the roots of its characteristic equation

$$h(\lambda) := \det(\lambda I - A) = \lambda^n - c_1\lambda^{n-1} + \cdots + (-1)^n c_n = 0, \tag{8}$$

where c_k is the sum of principal k–th order minors of A. In particular $c_1 = \mathrm{tr}(A)$ and $c_n = \det(A)$. Due to this definition sometimes the solution of the characteristic equation is presented as a way to find the spectrum (the set of eigenvalues) $\Lambda := \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ of A. This is in general a bad computational algorithm for the following two reasons.

First, the computation of the coefficients c_k for a general matrix A may be a difficult task and hence the computed values $\widehat{c}_k$ may be contaminated with large errors. Of course, the correct way to compute c_k is not the use of determinants but even more sophisticated algorithms for this purpose may not produce an acceptable result.

Second, the roots λ_k of the characteristic polynomial $h(\lambda)$ may be very sensitive to changes in its coefficients c_k even if the eigenvalues λ_k of the matrix A are not sensitive relative to changes in its entries a_{ik}.

Due to the above reasons *the numerical computation of the eigenstructure (eigenvalues in particular) of a matrix is not done via its characteristic equation* (8). An exception of this rule for integer matrices is considered below.

Modern numerical algorithms for the computation of Λ are based on the transformation of A into Schur condensed form $S = U^{\mathrm{H}}AU$, where U is an unitary matrix. The matrix S is upper triangular with the eigenvalues of A on its diagonal. It is well known that the roots of an n–th degree polynomial for $n \geq 5$ can not in general be expressed by algebraic expressions involving its coefficients. Hence there is no finite computational procedure for finding the spectrum of general large order matrices. As a result the algorithms to compute S are iterative and the computed result contains both truncation and rounding errors.

For reduction of A into Schur form S usually the QR algorithm of Fransis–Kublanovskaya is used. The idea of the algorithm is to construct a sequence $\{A_k\}$ which converges to S. Let for example $A := A_1 =: Q_1R_1$ be the QR decomposition of A_1, where the matrix Q_1 is unitary and the matrix R_1 is upper triangular. Define the matrix $A_2 := R_1Q_1$ and let $A_2 =: Q_2R_2$ be the QR decomposition of A_2. At the the k–th step we use the already available matrices $Q_1, Q_2, \ldots, Q_{k-1}$ and $R_1, R_2, \ldots, R_{k-1}$. We form the matrix $A_k := R_{k-1}Q_{k-1}$ and compute its QR decomposition $A_k =: Q_kR_k$. Of course, the actual algorithm is much more sophisticated and includes a number of mechanisms for improving the accuracy and efficiency of computations.

The QR algorithm for the computation of Λ works relatively well when the eigenvalues of the matrix A are not very sensitive to perturbations and in particular when this matrix has only linear elementary divisors (or only 1×1 blocks in its Jordan canonical form). When the matrix has very sensitive eigenvalues (nonlinear elementary divisors in particular) the corresponding eigenvalues may be computed with large errors. Consider for example the 3×3 matrix

$$A = \begin{bmatrix} -90001 & 769000 & -690000 \\ -810000 & 6909999 & -6200000 \\ -891000 & 7601000 & -6820001 \end{bmatrix}$$

which has a single Jordan block with an eigenvalue -1 and characteristic polynomial (c.p.) $\lambda^3 + 3\lambda^2 + 3\lambda + 1$. The command `eig(A)` from MATLAB, ver. 6.0 produces the

very wrong result $\lambda_{1,2} = -1.5994 \pm 1.0441\,\imath$, $\lambda_3 = 0.1989$ (the results are rounded to 4 significant decimal digits). Here even a positive eigenvalue has been computed. The command `poly(A)` computes the wrong c.p. $\lambda^3+3.0000\lambda^2+3.0120\lambda-0.7255$. The reason is the high sensitivity of the eigenvalues with nonlinear elementary divisors although the result -0.7255 instead of 1 for $-\det(A)$ is difficult to explain. In MATLAB, ver. 7.0 the computed results are slightly better, namely $\lambda_{1,2} = -1.4506 \pm 0.7837\,\imath$, $\lambda_3 = -0.0988$. Correspondingly, the computed c.p. now is $\lambda^3 + 3.0000\lambda^2 + 3.0051\lambda + 0.2686$. Here again the free term is computed with large error of about 73 percent. However, using the command `jordan(A)` from MATLAB, ver. 7.0 which implements symbolic computations we get the correct answer for the Jordan structure of A.

This example has the interesting feature that the computation of the spectrum of A via its c.p. gives the correct answer in MATLAB environment! For example we have `trace(A) = -3`, `det(A) = -1` and the sum of the minors $A(i,i)*A(j,j)-A(i,j)*A(j,i)$ for $i > j$ gives the true value 3 for the coefficient before λ in the c.p. of A. The reason here is that the multiplication of two elements of A may still be done in integer arithmetic with no rounding errors.

The best numerical way to find the eigenvalues of A is to apply directly the QR algorithm or the algorithm of Kagström and Ruhe [8, 13] (which includes the QR algorithm as a preliminary step) for the computation of the eigenstructure of A. A modification of this algorithm is implemented in the function `jord` from the interactive system SYSLAB [12, 10]. Applying this function to the matrix A defined above we obtain the Jordan form and the modal matrix of A within 9 true significant decimal digits.

8. Differentiation and integration. Suppose that the function $f : \mathbb{R} \to \mathbb{R}$, differentiable at the point x, is defined by an explicit expression containing a number of elementary functions, e.g. by the decomposition $f = f_1 \circ f_2 \circ \cdots \circ f_m$. Then it is considered (more or less correctly) that finding the derivative $f'(x)$ of f at the point x by successively applying the chain rule is a routine task and should not present any significant difficulties for the student.

On the contrary, finding the definite integral $J := \int_a^b f(x)\mathrm{d}x = F(b) - F(a)$ using the primitive function F of f (such that $F' = f$) is considered as a generally difficult task even if $f(x)$ is a simply looking expression defined by elementary functions. Of course, the idea to compute J using the primitive F is usually not a good idea. On the other hand the definition of the integral by primitives when teaching mathematics in technical universities is preferable [15, 16].

The integration is a bounded and stable operation in the sense that the definite integral of a bounded function on a finite interval is bounded, and small perturbations in the integrand lead to small perturbations in the value of the integral. And although the numerical integration of even continuous functions with large derivatives and/or fast oscillation may be a problem, there exist reliable adaptive and non–adaptive reliable integration schemes, see e.g. [14, 17]. The computer system MATLAB also contains reliable algorithms for the computation definite integrals, see the commands `quad`, `quadl` and `polyint`. Double integrals may be computed by the command `dblquad`. Details about the implementation of the commands may be found by e.g. `help quad`, etc.

Thus many students remain with the impression that to find derivatives is easy while to compute definite integrals is difficult. When differentiation and integration are

performed in a FPA the situation is usually the opposite, i.e. differentiation is difficult and potentially contaminated with large errors while integration may be realized by a reliable computational procedure.

There are several reasons for this phenomenon. The differentiation may be very difficult because this is an unbounded operation. Moreover, when a finite difference approximation for the derivative is applied, e.g.

$$f'(x) \approx D(x,h) := \frac{f(x+h) - f(x-h)}{2h}$$

then the truncation error $|f'(x) - D(x,h)|$ is of order $O(h^2)$ while the rounding errors may be of order $O(h^{-1})$. In addition the subtraction of close quantities $f(x+h)$ and $f(x-h)$ (for small $h > 0$) may lead to catastrophic cancellation and a significant loos of accuracy. As a result the increment h cannot be decreased beyond the order of $\mathbf{u}^{1/3}$ (when f and its derivatives at the point x are of order 1) and this is about 10^{-5} to 10^{-6}. More generally, the increment h should be of order $\mathbf{u}^{1/(q+1)}$ when the truncation error in the approximate formula for the derivative is of order q. For the purposes of differentiation specialized adaptive algorithms may be applied [17].

9. Conclusions and references. In this tutorial paper we have considered a number of computational practices, good and bad, that are taught in Bulgarian schools and universities. It may be observed that good practices are an exception rather than a rule. The reasons are the lack of appropriate literature in Bulgarian as well as the inadequate numerical background of school teachers and even of some university lecturers. Modern computer methods for mathematical calculations have been considered in [2, 3] and more recently in [4, 13, 17, 5]. A popular study of computational problems in control theory is presented in [6].

REFERENCES

[1] D. Duffy. Advanced Engineering Mathematics with MATLAB. Chapman and Hall/CRC Press, 2003.

[2] G. Forsythe, C. Moler. Computer Solution of Linear Algebraic Systems. Prentice Hall, Englewood Cliffs, N.J., 1967.

[3] G. Forsythe, M. Malcolm, C. Moler. Computer Methods for Mathematical Computations. Prentice Hall, Englewood Cliffs, N.J., 1977.

[4] G. Golub, C. Van Loan. Matrix Computations. The John Hopkins Univ. Press, Baltimore, 3rd ed., 1996.

[5] N. Higham. Accuracy and Stability of Numerical Algorithms. SIAM, Philadelphia, 1st ed. 1996, 2nd ed. 2002.

[6] N. Higham, M. Konstantinov, F. Mehrmann, P. Petkov. The sensitivity of computational control problems. *IEEE Control Systems Magazine*, **24** (2004), 28–43 (PDF text available at `http://ieeexplore.ieee.org`).

[7] IEEE Standard for Binary Floating–Point Arithmetic, ANSI/IEEE Standard 754–1985. IEEE, New York, 1985. Reprinted in *SIGPLAN Notices*, **22** (1987), No 2, 9–25.

[8] B. Kagström, A. Ruhe. An algorithm for numerical computation of the Jordan normal form of a complex matrix. *ACM Trans. Math. Software*, **6** (1980), 398–419; Algorithm 560 JNF, *Ibid*, 510–523.

[9] M. Konstantinov. Foundations of Numerical Analysis (with MATLAB Examples). Publ. House UACEG, Sofia, 2005 (PDF text available at `http://uacg.bg/books/math/ns.pdf`).

[10] M. Konstantinov, P. Petkov, N. Christov. Matrix Computations with the Dialog System SYSLAB. Publ. House Higher Inst. Arch. Civil Engr., Sofia, 1992 (in Bulgarian).
[11] M. Konstantinov, P. Petkov, Z. Gancheva. Thirteen myth in numerical analysis. *Proc. 34 Spring Conf. UBM*, Borovetz, 2005, 237–242.
[12] P. Petkov, N. Christov. Analysis and Design of Linear Control Systems Using SYSLAB. Tehnika, Sofia, 1993(in Bulgarian).
[13] P. Petkov, N. Christov, M. Konstantinov. Computational Methods for Linear Control Systems. Prentice Hall, Hemmel Hempstead, 1991.
[14] R. Piessens, E. de Doncker–Kapenga, C. Uberhuber, D. Kahane. QUADPACK: A Subroutine Package for Automatic Integration. Springer–Verlag, New York, 1983.
[15] V. Todorov, P. Stoev, M. Konstantinov. Newton integrals. *Proc. 31 Spring Conf. UBM*, Borovetz, 2002, 290–294 (in Bulgarian).
[16] V. Todorov, P. Stoev, M. Konstantinov. Primitives of certain classes of functions. *Proc. 33 Spring Conf. UBM*, Borovetz, 2004, 292–296.
[17] N. Vulchanov, M. Konstantinov. Modern Mathematical Methods for Computer Calculations. Part I: Foundations of Computer Calculations. Numerical Differentiation and Integration. Studies in Math. Sci., Bulgarian Inst. Anal. Res., vol. 1, Sofia, 1996 (in Bulgarian).

Mihail Konstantinov
UACEG
1046 Sofia, Bulgaria
e-mail: mmk_fte@uacg.bg

Petko Petkov
TU-Sofia
1756 Sofia, Bulgaria
php@tu-sofia.bg

ЛОШИ И ДОБРИ ИЗЧИСЛИТЕЛНИ ПРАКТИКИ (С ПРИМЕРИ ОТ MATLAB)

Михаил Константинов, Петко Петков

Разгледани са няколко лоши, както и съответните им добри, изчислителни практики. Лошите практики могат да произведат напълно погрешни резултати, когато съответните изчислителни схеми са реализирани в крайна аритметика. Цел на работата е да се подпомогне преподаването на добри изчислителни практики в училищата и университетите.