МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2012
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2012
Proceedings of the Forty First Spring Conference
of the Union of Bulgarian Mathematicians
Borovetz, April 9–12, 2012

NUMERICAL ISSUES IN USING MATLAB*

Mihail Konstantinov, Vesela Pasheva, Petko Petkov

In this tutorial paper we consider some numerical problems arising in using the computer system for doing mathematics MATLAB: evaluation of trigonometric functions, computing matrix powers, spectral analysis of integer matrices and computing roots of algebraic polynomials. Some of the reasons for these numerical difficulties may be explained by the properties of the underlying binary floating-point arithmetic.

Introduction. MATLAB[1] is a powerful interactive computer system for doing mathematics. Its numerical part is based on a number of reliable matrix algorithms. However, working in purely numerical mode (using standard floating-point machine arithmetic) MATLAB may produce results which differ unexpectedly strong from the true answers. Such cases must be understood and detected in order to avoid the introduction of large errors in the solution without a warning for the user. We consider several such cases and explain the corresponding numerical behavior of the computer system.

Some of the problems arising in the standard use of finite machine arithmetic may be avoided applying interval methods, algorithms and computer codes. Such codes are freely available in the MATLAB toolbox INTLAB based on reliable and self-validating algorithms [3].

For simplicity we have considered only computations in real arithmetic. Also, we have used two versions of MATLAB: 6.5.0 (R13), 2002 and 7.11.0 (R2010b), 2010. The two versions produce slightly different results for some problems.

The results presented reflect the experience of the authors in teaching the basic mathematical courses in the University of Architecture, Civil Engineering and Geodesy, the Technical University of Sofia and the European Polytechnical University.

Floating-point computations. Here we recall the basic facts about binary floating-point computations, see e.g. [2]. They are characterized by a finite set of machine numbers $\mathbb{M} \subset \mathbb{R}$ together with the rules for performing operations in $\mathbb{M}$, including rules for rounding. The set $\mathbb{M}$ contains 0 and is symmetric relative to $\mathbb{R}$.

A number $x \in \mathbb{R}$ is rounded to the nearest machine number $x^* \in \mathbb{M}$ (denoted also as $\mathrm{fl}(x)$) and $x^* = x$ if and only if $x \in \mathbb{M}$. If x is in the middle between two consecutive machine numbers then it is rounded to the machine number with zero least significant digit.

*2000 Mathematics Subject Classification: Primary 97N20.
Key words: numerical computations, numerical problems, computer systems.
[1] MATLAB© is a trademark of MathWorks, Inc.

The binary floating point double-precision arithmetic in MATLAB obeys the IEEE standard [1]. It is characterized by three important positive numbers, namely $\mathbf{r}_{\max} = 2^{1024} \simeq 1.7977 \times 10^{308}$, $\mathbf{r}_{\min} = 2^{-1022} \simeq 2.2251 \times 10^{-308}$, $\mathbf{u}_{\rm rnd} = 2^{-53} \simeq 1.1102 \times 10^{-16}$. They may be received in MATLAB by the commands `realmax`, `realmin` and `eps/2`. The number $\mathbf{u}_{\rm rnd}$ is said to be the *rounding unit*. In the single precision version of the arithmetic the rounding unit is $2^{-24} \simeq 5.9605 \times 10^{-8}$.

A number $x \in \mathbb{R}$ is in the *standard range* if either $x = 0$, or $|x| \in [\mathbf{r}_{\min}, \mathbf{r}_{\max}]$. In this case x is rounded to the nearest machine number $x^* \in \mathbb{M}$ (with the rule to break ties described above) so that $0^* = 0$ and $|x^* - x|/|x| \le \mathbf{u}_{\rm rnd}$, $x \ne 0$. Thus numbers from the standard range are rounded with small relative error of order 10^{-16}, i.e. with 15-16 true decimal digits.

Let $\diamond$ be an arithmetic operation and let the non-zero quantities x, y and $x \diamond y$ be in the standard range. Let $(x \diamond y)^* \in \mathbb{M}$ be the result of the machine computation of $x \diamond y$. Then we have $(x \diamond y)^* = (x \diamond y)(1 + \alpha)$, where $|\alpha|$ is a small multiple of $\mathbf{u}_{\rm rnd}$.

What happens with numbers $x \in \mathbb{R}$ that are outside the standard range is explained in [2].

Evaluation of trigonometric functions. The approximate value `pi` for π in MATLAB is important in evaluating trigonometric functions.

The command `>> pi` displays the 15–digit answer `pi = 3.14159265358979` in long format. To find exactly `pi` one may use the symbolic command `>> sym(pi,'e')` with error estimate which gives `pi - 198*eps/359`. This means that pi $= \pi - s$, where $s = 99 \times 2^{-51}/359 \simeq 1.2246 \times 10^{-16}$. Thus the relative error s/π in `pi` is about $0.3511\,\mathbf{u}_{\rm rnd}$.

Consider now the evaluation of trigonometric functions. The command `>> s1 = sin(pi)` gives `s1 = 1.224646799147353e-016`, or $s1 \simeq s$, and illustrates the fact that $\sin(\text{pi}) = s + \mathrm{O}(s^3)$.

A problem with the approximation of π is that trigonometric functions with arguments $N\pi$ for large N of order 10^{15} or more may be computed with very large errors. For example, the commands

```
>> a = sin(10^15*pi), b = cos((10^15+0.5)*pi)
```

give `a = -0.2362` and `b = 0.3044` while the exact answer is $a = b = 0$.

We recall that the command `vpa(S,N)` produces the value of a symbolic quantity S in the form of a string with N decimal digits. Attempts to improve the accuracy using a 100-digit approximation `vpa(pi,100)` for π are not very successful since

```
>> sin(vpa(pi,100)*10^16)
```

gives `ans = -.28841971693993751058209749445923e-016`

Some of these problems may be avoided defining a symbolic value for π from `>> PI = sym('pi')`. With this value the command

```
>> sin(10^16*PI)
```

already gives the correct answer 0.

Thus approach may not be suitable for use in complex computational algorithms. That is one may define inline trigonometric functions instead of the standard trigonometric functions, e.g. `>> Sin = inline('sin(rem(x,2*pi))')`, where `rem(x,y)` computes the remainder of the quotient x/y. Now we have the correct result

```
>> Sin(10^16*pi) = 0
```

instead of the wrong answer

```
>> sin(10^16*pi) = -0.3752
```

Matrix powers. The computation of matrix powers is one of the dangerous operation in floating point arithmetic. Consider the 6×6 matrix

$$A = \begin{bmatrix} -0.8790 & 1.8440 & -9.1200 & -3.6960 & 24.5380 & -54.7360 \\ 1.7420 & -3.6300 & 17.9860 & 7.2780 & -48.4100 & 107.9380 \\ -1.6570 & 3.4480 & -17.0880 & -6.9120 & 46.0000 & -102.5510 \\ -0.4800 & 0.9640 & -4.8390 & -1.9430 & 13.0410 & -29.0110 \\ 0.2510 & -0.5000 & 2.5160 & 1.0080 & -6.7840 & 15.0820 \\ 0.4940 & -1.0160 & 5.0560 & 2.0400 & -13.6160 & 30.3330 \end{bmatrix}.$$

The exact 6-th power of this matrix is (to four digits)

$$A^6 = 10^{-7} \times \begin{bmatrix} -0.0010 & 0.0020 & -0.0102 & -0.0041 & 0.0275 & -0.0612 \\ 0.0020 & -0.0041 & 0.0204 & 0.0082 & -0.0551 & 0.1224 \\ -0.0019 & 0.0039 & -0.0195 & -0.0078 & 0.0526 & -0.1170 \\ -0.0006 & 0.0012 & -0.0062 & -0.0025 & 0.0167 & -0.0372 \\ 0.0003 & -0.0007 & 0.0033 & 0.0013 & -0.0089 & 0.0198 \\ 0.0006 & -0.0012 & 0.0060 & 0.0024 & -0.0162 & 0.0360 \end{bmatrix}.$$

Computing A^6 in MATLAB with single precision gives

$$\mathrm{fl}(A^6) = 10^{-7} \times \begin{bmatrix} 0.0005 & -0.0010 & 0.0050 & 0.0020 & -0.0135 & 0.0298 \\ -0.0012 & 0.0024 & -0.0118 & -0.0047 & 0.0317 & -0.0706 \\ 0.0042 & -0.0086 & 0.0427 & 0.0172 & -0.1149 & 0.2560 \\ 0.0004 & -0.0007 & 0.0036 & 0.0014 & -0.0097 & 0.0217 \\ -0.0005 & 0.0011 & -0.0054 & -0.0022 & 0.0145 & -0.0322 \\ -0.0002 & 0.0004 & -0.0019 & -0.0008 & 0.0051 & -0.0112 \end{bmatrix}.$$

Even the signs of the elements of the computed matrix power are wrong! The reason for this catastrophe are the large errors during the cancelations in computing the elements of A^6. This matrix has very small elements in comparison to the elements of the original matrix and the only way to compute these small elements is through cancelation.

In Figure 1 we show the relative errors in the computed result. For all matrix elements the relative errors are larger than 1 which explains the wrong signs of all computed elements.

Eigenstructure of small integer matrices. Computing numerically the eigenstructure and the inverse of even small integer matrices may be a severe problem. Consider the matrix

$$A = \begin{bmatrix} -90\,001 & 769\,000 & -690\,000 \\ -810\,000 & 6\,909\,999 & -6\,200\,000 \\ -891\,000 & 7\,601\,000 & -6\,820\,001 \end{bmatrix}$$

which has a triple eigenvalue $\lambda = -1$ and Jordan form $J = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & -1 \end{bmatrix}$.

The command `eig(A)` performed in MATLAB, ver 6.5 and ver 7.11, gives the completely wrong answer

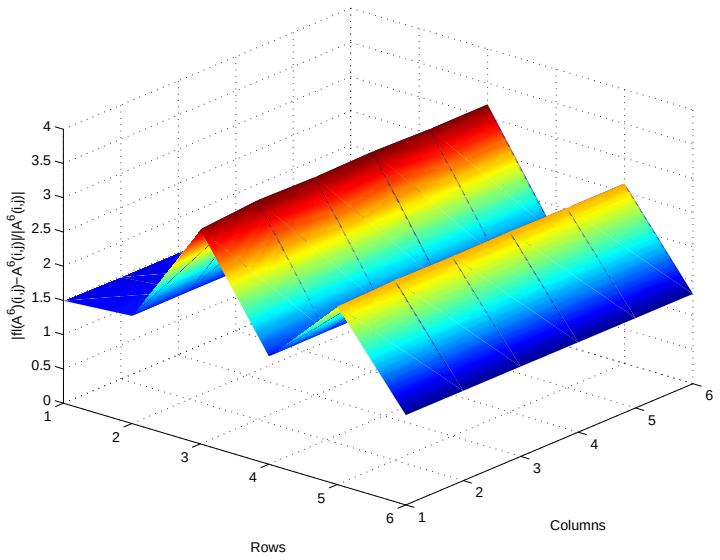

Fig. 1. Relative errors in the elements of computed matrix power

```
 -1.59942995872280 + 1.044058803139 56i
 -1.59942995872280 - 1.044058803139 56i
  0.19885991695974
```

Here the average relative error

```
>> norm(eig(A)+ones(3,1))/sqrt(3) = 1.20222227841444
```

is more than 120 percent! Moreover, even the conclusion about the stability of the matrix A is wrong because one of the eigenvalues has been computed as a positive quantity (0.19886 instead of -1)! An additional problem in this numerical computation is that there has been no warning for the wrong performance of the command **eig**.

But things are even worse. An experienced user will check the computed result using the eigenvalue condition numbers of the matrix A which are the reciprocals of the cosines of the angles between the left and right eigenvectors of A (in our case there is one eigenvector and two principal vectors). These quantities are obtained from

```
>> condeig(A)
ans =
  1.0e+008 *
   4.61280842056556
   4.61280848930177
   4.57446786644502
```

Since the quantity `eps*norm(condeig(A))` is 1.769×10^{-7} we may expect up to 6-7 true digits in the computed eigenvalues (although we have none).

Even if the experienced user decides that the matrix A has a triple eigenvalue, he may still try to estimate the absolute error in the computed result from

```
>> (eps*norm(A))^(1/3) = 0.00145570517581
```

In this case the user may expect about 3 true digits (although there are none).

The attempt to compute the characteristic polynomial $\lambda^3 + 3\lambda^2 + 3\lambda + 1$ of A is also not successful. We have (in `format short`)

```
>> poly(A)
ans =
   1.0000   3.0000   3.0121  -0.7255
```

There is a 3-percent error in the coefficient before λ and a 173-percent error in the free term which should be equal to $-\det(A) = 1$. The reason is that the coefficients of `poly(A)` are obtained as symmetric functions of the wrongly computed eigenvalues. It is interesting that we may compute $\det(A)$ correctly from `>> det(A) = -1`.

The above considerations clearly show that the numerical spectral analysis of even low order integer matrices may be contaminated with very large errors.

At the same time the command `jordan` (in symbolic mode) gives the correct Jordan form J of A together with the transformation matrix V such that $AV = JV$:

```
>> [V,J] = jordan(A)
V =
  1.0e+008 *
   1.00000000000000  -0.00001111111111  -0.00000007654321
   9.00000000000000                   0                   0
   9.90000000000000                   0   0.00000001000000
J =
    -1     1     0
     0    -1     1
     0     0    -1
```

The attempt to compute numerically A^{-1} by the command `inv(A)` is also not successful. We have

```
inv(A)*A Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 3.286207e-017.
ans =
   1.00000000000000                   0                   0
  -0.12500000000000   1.00000000000000                   0
                   0   1.00000000000000                   0
```

Thus the matrix A^{-1} is computed by `inv(A)` with large errors and it even does not commute with A,

```
>> A*inv(A)-inv(A)*A
ans =
   0.00012207031250                   0  -0.12500000000000
   0.12597656250000                   0                   0
   0.00097656250000  -1.00000000000000                   0
```

However, the computations with `inv` in this case release a warning for very small values of the reciprocal `RCOND` of the condition number $\|A\|\,\|A^{-1}\|$ of A. With `RCOND` of order 10^{-17} we may expect no true significant digit in the computed solution.

The correct inverse $B = A^{-1}$ may be computed using the exact Jordan form. Since $A = VJV^{-1}$ we have $A^{-1} = VJ^{-1}V^{-1}$ and

```
>> B = V/J/V
B =
  1.0e+009 *
   0.00008999900000   0.10923100000000  -0.09931000000000
   0.00081000000000   0.98308999900000  -0.89380000000000
   0.00089100000000   1.08139900000000  -0.98318000100000
```

or

$$A^{-1} = \begin{bmatrix} 89\,999 & 109\,231\,000 & -99\,310\,000 \\ 810\,000 & 983\,089\,999 & -893\,800\,000 \\ 891\,000 & 1\,081\,399\,000 & -983\,180\,001 \end{bmatrix}.$$

Finally the inverse of A is computed exactly. But in matrix operations with general matrices we cannot rely on the use of the symbolic function `jordan`.

Solving algebraic equations with the program 'roots'. In MATLAB the polynomial $p(\lambda) = \lambda^n + c_1\lambda^{n-1} + \cdots + c_n$ is represented with the $(n+1)$–vector $q = [1, c]$, $c = [c_1, c_2, \ldots, c_n]$. All zeros of the polynomial p may be found by the symbolic function `solve` or by the numerical function `roots`. Both functions produce the n–vector $[\lambda_1, \lambda_2, \ldots, \lambda_n]$T of the roots. The use of `roots`, however, may lead to large errors when $n > 2$ and the polynomial has multiple zeros. The problem is that such large errors are not accompanied by warning messages.

For $c_k = n!/k!(n-k)!$ the polynomial p has a root $\lambda = -1$ with multiplicity n. In this case the relative error `norm(roots(c) + ones(n,1))/sqrt(n)` is of order $\mathrm{eps}^{1/n}$. The reason is that `roots(c)` is the same as `eig(compan(c))`, where `compan(c)` is the companion matrix $C_p = \begin{bmatrix} -c \\ I_{n-1}\ 0 \end{bmatrix}$ of the polynomial p. When p has a root λ_1 of multiplicity n_1 the Jordan canonical form of C_p has a Jordan $n_1 \times n_1$ block with eigenvalue λ_1. The eigenvalues of such blocks are very sensitive to perturbations: the perturbation in the eigenvalue may be of order δ^{1/n_1}, where δ is the perturbation in the matrix. At the same time the matrix A may have diagonal Jordan form and, hence, be quite insensitive to perturbations in its elements.

For example, the command

```
>> roots([1,4,6,4,1])
ans =
 -1.00022566526762
 -0.99999997084627 + 0.00022563610654i
 -0.99999997084627 - 0.00022563610654i
 -0.99977439303985
```

shows that the root -1 with multiplicity 4 of the equation $(\lambda+1)^4 = \lambda^4 + 4\lambda^3 + 6\lambda^2 + 4\lambda + 1 = 0$ is computed with only 4 true decimal digits as predicted by the eigenvalue sensitivity analysis.

The program `roots` replaces the zero finding problem for a polynomial with the eigenvalue problem for the companion matrix: `roots(c) = eig(compan(c))`. The latter problem is potentially very sensitive and the solution may be contaminated with large errors. On the other hand the program `roots` is fast and works well for non-defective matrices of order up to several thousand.

Conclusions. In this tutorial paper we have discussed some important computational problems (evaluation of basic trigonometric functions, computing matrix powers, spectral analysis of low order integer matrices and solving low degree algebraic equations) which may be solved with large errors in the standard numerical mode of MATLAB. Such cases are very instructive when teaching the students how to solve practical problems with the most popular computer systems for doing mathematics. The cases when these systems function properly are typical but they are not so interesting. The interesting cases are when they produce large errors without issuing warning messages! In addition, meeting such situations the students become really interested in the particularities of the machine arithmetic, in some specific computational problems and in the performance of computational schemes which sometimes do not produce reliable results.

Acknowledgement. The authors would like to thank the anonymous referee for his valuable suggestions.

REFERENCES

[1] IEEE Standard for Floating-Point Arithmetic 754-2008, IEEE, New York, 2008, ISBN 978-0-7381-5753-5.

[2] M. Konstantinov, P. Petkov. Loss of accuracy in numerical computations. In: *Mathematics and Education in mathematics*, Proc. of the 40 Spring Conference of the Union of Bulgarian mathematicians, 2011, 293–299.

[3] S. Rump. INTLAB – INTerval LABoratory. Developments in Reliable Computing (Ed. T. Csendes). Kluwer Acad. Publ., Dordrecht, 1999, pp. 77–104.

Mihail Konstantinov
Department of Mathematics
University of Architecture,
Civil Engineering and Geodesy
1046 Sofia, Bulgaria
e-mail: mmk_fte@uacg.bg

Vesela Pasheva
Faculty of Applied Mathematics
and Informatics
Technical University of Sofia
1756 Sofia, Bulgaria
e-mail: vvp@tu-sofia.bg

Petko Petkov
Department of Automatics
Technical University of Sofia
1756 Sofia, Bulgaria
e-mail: php@tu-sofia.bg

ЧИСЛЕНИ АСПЕКТИ ПРИ ИЗПОЛЗВАНЕ НА MATLAB

Михаил Константинов, Весела Пашева, Петко Петков

Разгледани са някои числени проблеми при използването на компютърната система MATLAB в учебната дейност: пресмятане на тригонометрични функции, повдигане на матрица на степен, спектрален анализ на целочислени матрици от нисък ред и пресмятане на корените на алгебрични уравнения. Причините за възникналите числени трудности могат да се обяснят с особеностите на използваната двоичната аритметика с плаваща точка.