UDC 517.5

J. Gilewicz (Centre Phys. Théor., Marseille, France),
M. Pindor* (Inst. Theor. Phys., Warsaw Univ., Poland)

GENERAL ALGORITHM OF COMPUTATION OF c-TABLE AND DETECTION OF VALLEYS

ЗАГАЛЬНИЙ АЛГОРИТМ ОБЧИСЛЕННЯ c-ТАБЛИЦЬ ТА ВИЗНАЧЕННЯ ТОЧОК МІНІМУМУ

We present a review of all interesting results concerning the c-table obtained by the authors during two last decades, which are not widely known, because they were presented in publications of a limited circulation. We discuss different computational aspects of softwares producing the c-tables with presence of blocs and their evolution following the evolution of computer environment:

effects of use of the 32-bit arithmetic (≈ 8 digits), 64-bit arithmetic (double precision, ≈ 16 digits) and of Bailey's Fortran multiprecision package (32 or 64 digits),

concurrence between the ascending and descending algorithms,

relation between complexity of computation and precision, overflow and underflow problems,

concurrence between different formulas allowing to overcome the blocs in the c-table,

practical simple criterion of detecting numerical zeros in the c-table allowing to identify the blocs,

automatic detection of valleys.

Наведено огляд усіх цікавих результатів щодо c-таблиць, одержаних авторами протягом двох останніх десятиліть, які маловідомі з причини публікації у виданнях обмеженого поширення. Розглянуто різні обчислювальні аспекти програм, що продукують c-таблиці з наявністю блоків, а також їх еволюцію, обумовлену еволюцією комп'ютерного середовища, а саме:

наслідки використання 32-бітової арифметики ($\approx$ 8 розрядів), 64-бітової арифметики (подвійна точність, ≈ 16 розрядів) та високоточного пакету Фортрана Бейлі (32 або 64 розряди),

порівняння зростаючих та спадних алгоритмів,

зв'язок між складністю обчислень і точністю, проблеми надпотоків та недостатніх потоків,

порівняння різних формул, шо дозволяють уникнути блоків у c-таблицях,

практичний простий критерій для визначення числових нулів у c-таблицях, що дозволяють ідентифікувати блоки,

автоматичне визначення точок мінімуму.

1. Introduction. The application of Padé approximants in computational problems starts frequently by the computation of the auxiliary table, so called c-table [1]. The entries of c-table are the Toeplitz determinants of matrices of linear systems defining the denominators of Padé approximants. The square blocs of zeros in the c-table indicate the existence of corresponding blocs in the table of Padé approximants. The so called valleys in the c-table, it is the lines of minimal absolute values of entries located on each antidiagonal, indicate the lines of best Padé approximants in the Padé table [2]. Notice that each element of an antidiagonal in c-table or Padé table is computed using the same coefficients, same information about a considered function. Because the computation of the c-table is simpler, it is recommended to begin by this computation to obtain global preliminary information about the interesting Padé approximants before of their own calculations. Let us recall some definitions.

Let $[m/n]$ be a Padé approximant P_m/Q_n to the formal power series $f(z) = \sum_{j=1}^{\infty} c_j z^j$ defined by $f(z)-P_m(z)/Q_n(z) = O(z^{m+n+1})$. The normalized denominator $Q_n(z) = 1 + q_1 z + \ldots + q_n z^n$, if it exists, is defined by the linear system $\sum_{j=0}^{n} c_{k-j} q_j = -c_k$, $k = m+1, \ldots, m+n$. The determinant of the matrix of this system is the Toeplitz determinant

*M. Pindor passed away in 2003.

$$m \geq 0, \quad n \geq 1: \quad C_n^m = \begin{vmatrix} c_m & c_{m-1} & \dots & c_{m-n+1} \\ c_{m+1} & . & \dots & . \\ . & . & \dots & . \\ . & . & \dots & . \\ c_{m+n-1} & . & \dots & c_m \end{vmatrix}, \tag{1}$$

where we put $c_k \equiv 0$ if $k < 0$. Defining $C_0^m := 1$ we can build an infinite table of C_n^m's, called c-table:

m	n				
	0	1	2	3	...
0	1	c_0	$(c_0)^2$	$(c_0)^3$	...
1	1	c_1	...	...	...
2	1	c_2	...	...	...
⋮	...	...	...	...	...

$$\equiv \begin{array}{|c|c|c|c|} \hline C_0^0 & C_1^0 & C_2^0 & \dots \\ \hline C_0^1 & C_1^1 & C_2^1 & \dots \\ \hline C_0^2 & C_1^2 & C_2^2 & \dots \\ \hline \dots & \dots & \dots & \dots \\ \hline \end{array}. \tag{2}$$

The second column of this table contains the coefficients of the power series $C_1^m = c_m$. The first row contains the powers of c_0, $C_n^0 = (c_0)^n$ and the second row can be computed recursively by expanding C_n^1:

$$C_n^1 = -\sum_{j=1}^{n} (-1)^j (c_0)^{j-1} c_j C_{n-j}^1. \tag{3}$$

The c-table was first introduced by Gragg [1]. Baker [3] used an alternative definition permuting rows in (1), calling the resulting determinants $C(m/n)$ and the corresponding table C-table. An obvious relation between $C(m/n)$ and C_n^m is

$$C_n^m = (-1)^{n(n-1)/2} C\left(\frac{m}{n}\right), \qquad C\left(\frac{m}{n}\right) = \begin{vmatrix} c_{m-n+1} & . & \dots & c_m \\ c_{m-n+2} & . & \dots & c_{m+1} \\ . & . & \dots & . \\ . & . & \dots & . \\ c_m & . & \dots & c_{m+n-1} \end{vmatrix}. \tag{4}$$

An interest in the c-table is due to its direct relation to the Padé table [2] and to a particular easiness of its computation. The c-table contains an information about a block structure of the Padé table. Each square block of zeros in the c-table defines a corresponding block in the Padé table. Moreover, a dominant term of an error of Padé approximant is given by a ratio of two Toeplitz determinants:

$$f(z) - \frac{P_m(z)}{Q_n(z)} = (-1)^n \frac{C_{n+1}^{m+1}}{C_m^n} z^{m+n+1} + \dots. \tag{5}$$

A theory of valleys in the c-table [4] leading to a numerical algorithm of choice of the best Padé approximant [5] is based on the property (5).

2. Computation of the *c*-table in the normal case. We say that the Padé table, respectively c-table, is normal, if it has no blocks. The determinants C_n^m can be computed directly by (1), but it is easier to compute the c-table recursively by means of the Sylvester formula relating five neighborhood entries

$$NS + EW = C^2, \qquad \begin{matrix} & N & \\ W & C & E \\ & S & \end{matrix}. \tag{6}$$

Starting from the two first columns one can compute the *East* elements by **ascending algorithm** [6]

$$E = (C^2 - NS)/W \quad \text{or} \quad E = C/W * C - S/W * N. \tag{7}$$

Alternatively, starting from the two first rows one can compute the *South* elements by the **descending algorithm**

$$S = (C^2 - EW)/N \quad \text{or} \quad S = C/N * C - E/N * W. \tag{8}$$

The two last forms in (7) and (8) was used thirty years ago in [6] in order to diminish a risk of overflows or underflows in 32-bit arithmetic computation. Numerical experiments show, that C_m^n's decrease or increase quite rapidly. In contrast to [6] we are not concerned today with the problem of overflow, because the double precision IEEE standard allows for a range of exponents sufficiently large for most applications (approximately between -308 and $+308$). Moreover the case of overflow is signaled by Inf or Nan conditions, without aborting a program. Therefore, we can use the first forms of formulas (7) and (8) which are faster and somewhat precise because of their smaller complexity.

To calculate the whole c-table, we can use either the ascending, or the descending algorithm, or a combination of the two. In [6] authors have analyzed a complexity of computation of elements in the c-table when using both algorithms (counting step by step all multiplications and divisions necessary to obtain the C_m^n). It was shown that there exists a line separating the c-table in two parts: the lower one where the ascending algorithm needs a lower computational cost, and the upper one where the descending algorithm is more advantageous. This line is very close to the diagonal. The following two elements are situated on the both sides of this line. Each element is followed by a number of operations needed by the descending algorithm and next by the ascending one [7] :

$$C_{11}^8 : 105\ 266; 159\ \ 995 \qquad C_{11}^9 : 349\ 092; 160\ 070.$$

On the other side, if we consider problems of precision, situation seems to be different. Naively thinking, we could expect, that a precision loss will be approximately monotonously dependent on a complexity, i.e., on a number of arithmetic operations necessary to calculate a given element and therefore the line mentioned, would divide c-table in two parts which should be calculated using different algorithms to minimize the complexity. However, our numerical experiments seem to show that the ascending algorithm is, generally, more precise. One could attribute this observation to the fact, that formula (3) for calculating C_n^1 contains many additions (subtractions), what is very dangerous for precision. On the contrary, each calculation using one of the algorithms

stemming from Sylvester formula, contains only one subtraction. This explanation can be verified by calculating C_n^1 with higher precision, but a conclusion is strongly perturbed by the fact that often C_n^1 and C_1^n differ completely in their behavior as functions of n. In some cases C_n^1 rise sharply, when C_1^n drop very fast. Consequently it is impossible to say, in advance, which algorithm will give more accurate values, even if C_n^1's are calculated with extra precision. On the other hand, Bailey's [8] Fortran package for multiprecision makes calculations with extra digits as easy that we think, one does not need to care about more precise algorithm, as even for fifty terms in power series and 64 digits of precision the whole c-table could be calculated in about 5 seconds. Interestingly, we remarked that running calculation with 32 digits was only marginally faster, what probably means that with those numbers of digits, function calls, and not actual arithmetic operations, were the main burden for the processor.

Table 1 illustrates some situations encountered during numerical experimenting. We are going to compare first three columns with the two remaining ones considered as being exact. This two columns have first four digits identical, although they have been calculated with different algorithms, and therefore we consider these digits as exact.

Some elements of c-table calculated using either ascending or descending algorithms with different precision for $f(z) = -1/z * \log(1-z)$:

a — double precision, ascending alg.;

b — double precision, descending alg.;

c — double precision, descending alg. starting from C_n^1 computed with 32 digits;

d — full 32 digits, ascending alg.;

e — full 32 digits, descending alg.

Table 1

row	a	b	c	d	e
			9 th column		
11	.7086d-54	.7085d-54	.7087d-54	.7087d-54	.7087d-54
12	.5307d-57	.5308d-57	.5304d-57	.5304d-57	.5304d-57
13	.7619d-60	.7607d-60	.7625d-60	.7625d-60	.7625d-60
14	.1867d-62	.1875d-62	.1868d-62	.1868d-62	.1868d-62
15	.7236d-65	.7163d-65	.7148d-65	.7150d-65	.7150d-65
16	.3761d-67	.3834d-67	.4007d-67	.4004d-67	.4004d-67
17	.3768d-69	.3645d-69	.3110d-69	.3115d-69	.3115d-69
18	.1437d-71	.1894d-71	.3242d-71	.3232d-71	.3232d-71
19	.9312d-73	.6939d-73	.4291d-73	.4322d-73	.4322d-73
20	-.6779d-75	.5790d-75	.7369d-75	.7249d-75	.7349d-75
21	.5458d-76	-.1103d-76	.1435d-76	.1489d-76	.1489d-76

row	a	b	c	d	e
13 th column					
8	.1345d-54	.1345d-54	.1344d-54	.1344d-54	.1344d-54
9	.3457d-64	.3445d-64	.3465d-64	.3465d-64	.3465d-64
10	-.1041d-73	-.1073d-73	-.1035d-73	-.1035d-73	-.1035d-73
11	-.6476d-83	-.5203d-83	-.6039d-83	-.6054d-83	-.6054d-83
12	-.4552d-92	.1140d-91	.1475d-91	.1443d-91	.1443d-91
13	.1653d-97	.9137d-98	.1197d-98	.1387d-98	.1387d-98
14	-.9270d-103	-.7552d-103	.1614d-104	.9683d-105	.9683d-105
15	-.2095d-108	-.6141d-108	-.5007d-110	.2715d-110	.2715d-110
16	-.2044d-111	-.1911d-111	.2723d-114	.2241d-115	.2241d-115
17	-.4047d-115	-.4269d-115	-.1742d-118	.4453d-120	.4453d-120

In Table 1 above we present elements lying below the line discussed above (except C_{13}^{8} and C_{13}^{9}). Therefore, calculation of these elements using ascending algorithm requires less multiplications/divisions than using descending algorithm. We could then expect that using the former one would give more precise results. For 9-th column it is, in fact, the case for rows 11 through 14. Surprisingly, for next columns, descending algorithm, even with double precision only, gives better results, even if for rows 20 and 21 both numbers are very bad. For 13-th column elements in rows 8, 9 and 10 lie above the line and descending algorithm should be better for them, and, generally it is, if we calculate second row with 32 digits of precision. For 11-th row however ascending algorithm gives more accurate value, then the descending one, even in case c and again for 12-th row, descending algorithm is better while further down double precision gives completely erroneous numbers independently of an algorithm used. Below 13-th row, even calculation of the second row with 32 digits does not help, and we miss correct numbers by orders of magnitude.

3. Blocks. Separate question concerns detection of blocks in c-table. Evidently, we cannot expect to see exact zeros, and much discussion has been devoted to the problem of *computational zero*. We think that the method of Vignes [9] is computationally too expensive to be of practical use in our case. A simple and practical criterion of detecting zeros in c-table has been proposed by Guzinski [10]. It consists of observing changes in absolute values of entries in c-table computed in subsequent steps. If there is a rapid drop in absolute value of a current entry with respect to a preceding one (we have to select in advance a corresponding factor), we decide that this current value should be zero. We experimented with this criterion with, generally, positive results, though we have found that, again, high number of significant digits was absolutely necessary in some instances. The reason for this was that sometimes loss of precision was so severe that when working with double precision only, there was no significant drop in absolute values of elements at a border of a block. This is well illustrated in examples below.

First we show a fragment of c-table for a series produced by a rational function

$$f(x) = 1/(1 - x/3)^2 - 9x^3/(1 - x/9)^2$$

and the ascending algorithm (Table 2).

Table 2

	3	4	5	6	7	8	9	10
3	-7.04D + 02	6.28D + 03	-5.60D + 04	4.99D + 05	-4.45D + 06	3.97D + 07	-3.54D + 08	3.15D + 09
4	1.98D-01	6.61D-01	2.19D + 00	7.18D + 00	2.34D + 01	7.59D + 01	2.45D + 02	7.88D + 02
5	9.07D-04	6.61D-07	-6.61D-07	6.61D-07	-6.61D-07	6.61D-07	-6.61D-07	6.61D-07
6	4.09D-06	9.06D-10	6.55D-26	-2.26D-26	7.48D-27	-5.19D-27	1.68D-27	8.83D-30
7	1.78D-08	1.24D-12	-3.11D-29	3.48D-47	9.31D-47	2.17D-47	-4.33D-48	4.90D-49
8	6.92D-11	1.71D-15	1.41D-32	-1.28D-49	1.06D-66	-1.69D-67	4.82D-69	1.28D-70
9	1.29D-13	2.34D-18	-1.34D-35	4.09D-53	-2.31D-70	1.07D-87	-1.03D-89	1.45D-91
10	-2.24D-15	3.21D-21	5.94D-39	1.12D-56	9.06D-75	1.42D-92	-1.01-110	-1.59-113
11	-5.22D-17	4.40D-24	4.29D-44	1.73D-60	3.30D-79	2.73D-97	-2.18-116	-1.03-134
12	-8.47D-19	6.04D-27	-1.29D-45	2.68D-64	-4.03D-83	5.77-102	-3.23-121	2.02-140
13	-1.25D-20	8.28D-30	7.78D-49	1.16D-68	2.25D-88	7.42-107	5.52-127	
14	-1.78D-22	1.14D-32	-3.96D-52	-1.50D-73	2.01D-92	9.33-112		
15	-2.47D-24	1.56D-35	2.04D-55	6.88D-76	2.42D-96			
16	-3.40D-26	2.14D-38	-7.76D-59	1.30D-79				
17	-4.61D-28	2.93D-41	1.59D-62					

It should contain zeros in an infinite block starting at element C_5^6. If we have decided that a recently calculated element of c-table would be considered to be zero, when its absolute value was at least 10^{13} times smaller than that of a preceding element, we would have correctly detected the block. However if we calculate the same c-table using the descending algorithm, we get Table 3.

Table 3

	3	4	5	6	7	8	9	10
3	7.04D + 02	6.28D + 03	-5.60D + 04	4.99D + 05	-4.45D + 06	3.97D + 07	-3.54D + 08	3.15D + 09
4	1.98D-01	6.61D-01	2.19D + 00	7.18D + 00	2.34D + 01	7.59D + 01	2.45D + 02	7.88D + 02
5	9.07D-04	6.61D-07	-6.61D-07	6.61D-07	-6.61D-07	6.61D-07	-6.61D-07	6.71D-07
6	4.09D-06	9.06D-10	-3.10D-19	2.64D-19	1.61D-18	1.84D-18	-2.88D-17	4.40D-17
7	1.78D-08	1.24D-12	3.63D-22	8.61D-31	-3.19D-30	7.53D-29	-1.13D-27	2.00D-27
8	6.92D-11	1.71D-15	3.03D-24	4.38D-33	-3.39D-41	1.12D-39	-3.93D-38	1.40D-37
9	1.29D-13	2.34D-18	4.75D-27	1.42D-34	1.54D-42	-1.03D-51	-1.22D-48	1.13D-47
10	-2.24D-15	3.21D-21	-1.02D-28	2.92D-36	-7.39D-44	1.68D-51	-3.84D-59	1.30D-57
11	-5.22D-17	4.30D-24	2.13D-31	7.10D-39	3.62D-46	2.13D-53	1.79D-60	8.41D-68
12	-8.54D-19	9.24D-27	-1.48D-34	-9.18D-42	2.79D-49	-1.14D-55	-3.64D-62	-4.62D-69

Now, due to extra loss of accuracy when calculating elements C_n^1, elements of c-table drop only about 10 orders of magnitude between rows 5 and 6 in columns 5 and next

ones. At the same time, when we proceed from element C_3^{17} to C_4^{16} (see Table 2), none of them lying in the block, there is also a drop in absolute magnitude of about 10 orders of magnitude. This illustrates that to use Guzinski criterion we have to resort to multidigit precision. In fact when we calculate c-table, for the same function, with 32 digits, we get Table 4.

Table 4

	3	4	5	6	7	8	9	10
3	-7.04D + 02	6.28D + 03	-5.60D + 04	4.99D + 05	-4.45D + 06	3.97D + 07	-3.54D + 08	3.15D + 09
4	1.98D-01	6.61D-01	2.19D + 00	7.18D + 00	2.34D + 01	7.59D + 01	2.45D + 02	7.88D + 02
5	9.06D-04	6.61D-07	-6.61D-07	6.61D-07	-6.61D-07	6.61D-07	-6.61D-07	6.61D-07
6	4.09D-06	9.06D-10	-3.68D-39	-2.83D-40	1.71D-40	-3.00D-41	3.09D-42	-3.99D-43
7	1.78D-08	1.24D-12	-3.88D-43	1.07D-72	-3.15D-74	5.66D-76	3.70D-78	1.15D-78
8	6.92D-11	1.71D-15	3.22D-46	4.32D-77	2.25-108	-1.45-110	-2.06-112	-4.88-114
9	1.29D-13	2.34D-18	-7.76D-50	1.06D-81	-1.99-113	1.19-144	-7.78-147	8.63-149
10	-2.24D-15	3.21D-21	1.09D-53	-9.55D-87	-3.87-118	1.07-149	2.05-181	-1.42-183
11	-5.22D-17	4.40D-24	-1.94D-57	4.06D-90	-1.26-122	1.62-154	-1.95-186	-1.57-218
12	-8.47D-19	6.04D-27	-1.29D-60	8.34D-94	1.29-126	1.74-160	3.09-191	1.48-223
13	-1.25D-20	8.28D-30	1.74D-63	5.83D-97	-1.21-130	-2.46-163	-4.78-196	
14	-1.78D-22	1.14D-32	1.39D-66	6.60-100	1.22-133	1.48-167		
15	-2.47D-24	1.56D-35	-3.19D-69	4.55-103	-4.24-135			
16	-3.40D-26	2.14D-38	2.23D-72	1.09-106				
17	-4.61D-28	2.93D-41	-8.23D-76					

With this precision, both algorithms, the ascending and the descending one, give the same values for elements of c-table (at least within 4 most significant digits). We easily see that the smallest drop in absolute values of elements between 5 th and 6th columns and 4 th ant 5 th row, is 26 orders of magnitude (C_4^7 and C_5^6). This drop can be made, of course, arbitrarily large if we resort to calculations with even more digits. E.g. when we make calculations with 64 digits, we get this drop of absolute values to be around 10^{-60}.

4. Overview of the program: subroutine[*] *c*-table(a, c, nm, ll, koptim, eps, ier). Program first initializes each element of the c-table to 1. Then it calculates C_n^1 using formula (3). Next it checks the second column and the second row for zeros, and puts zeros into other entries of the c-table, so that they form square blocks of zeros with those found in the second row and the second column. Coordinates of left upper corners of the blocks found this way, if any, are stored in order of appearance in array nb, a first entry of which, being a number of the block. Finally, the program calculates systematically all antidiagonals from the upper left corner C_2^2, first entry unknown at this moment, down. It starts, each time, from the second column, or the second row, depending on a choice of a user (ascending or descending algorithm, respectively). Before any of elements is calculated, it is checked whether it was not already put zero. If so, it means that the element belongs to one of square blocks already partially, or completely discovered. In this case program checks whether a previously considered element (on the same

[*]Two subroutines can be obtained writing to J. Gilewicz: gilewicz@cpt.univ-mrs.fr.

antidiagonal) and an element two columns to the left (two rows up for descending algorithm) are both different from zero. If any of them is zero we proceed to a next element on the same antidiagonal. In the opposite case, we are in the last row, second column (last column, second row) of a completely discovered block and we have to calculate a multiplier, that will be necessary to calculate elements just behind (below of) the block. We also need a number of the block to know where to store the multiplier. We identify the block by finding its upper left corner, calculate the multiplier using (7) or (8), store it in a position of an array hh defined by a number of the block, and proceed to a next element on the antidiagonal.

If the element is not zero (it was initialized to 1), it means that it has not yet been calculated. Then, at first, the program tries to apply the Sylvester formula, and to this end it checks whether an element of the c-table, which appears in a denominator of the formula is not zero. If it is not, the formula (6) is applied, and the element is calculated. If it is, however, other formulae — Gilewicz formula [2, p. 374] (formula (85)) or Paszkowski formula [11] are to be applied. To decide which one of them is appropriate, the program checks an element to the left, for the ascending algorithm, or up, for the descending one. If it is zero — Gilewicz formula is to be applied, if it is not — Paszkowski formula is the right one. In both cases the program has to identify the block. In the first case we need the corresponding multiplier. In the second one we need to know the block position and size, to find elements entering Paszkowski formula. The block is identified by finding its left upper corner and then finding its number in the array nb. The number, necessary if we apply Gilewicz formula, gives us a position of the multiplier needed in this case, in an array hh. After calculating the current element of the c-table, we have to check whether it should not be considered as being zero. A procedure to decide it, even if precision errors prevent the element to vanish, was described, in detail, above (cf. Guzinski criterion). We remark here that the check is unnecessary for elements calculated using Gilewicz formula, because to use it, we have to know that we are just behind (or below of) a block. If we decided that the element is zero, we are discovering a new block. We check an element right above (or to the left) to see if it is also zero. If it is not - we have just hit an upper left corner of a new block. In this case, we open a new entry in the array nb, putting coordinates of the corner there and proceed to the next element on the current antidiagonal. In the opposite case, we have just discovered a new part of a block encountered already before. Therefore, we fill with zeros a new "layer", as explained on Fig. 1. Then, we proceed to the next element on the current antidiagonal. In any other case we also proceed to the next element on the antidiagonal. This way, we systematically calculate all elements of the c-table, defined by coefficients of the series that we have.

5. Valleys: subroutine *c*-vallay(c, np, n, nv, nnv). To design a routine detecting valleys in c-table, we have to take into account the fact that if not many enough coefficients of the series, being studied, are known, a part of c-table which we can calculate, may not have well formed valleys. Therefore, we decided that detecting the smallest element in the last antidiagonal calculated, is not a viable procedure and some more information is necessary for a user to decide whether the program actually detected a valley, or he should inspect the whole c-table himself. First, we take into account that the true valley (that would be detected, if more coefficients were known) may not manifest itself as the deepest of possible spurious valleys seen at that order. Therefore, we

						X	X	X	$\swarrow$
	X	X	X	X	X	X	X	1	1
	X	X	0	0	0	①	1	1	
	X	X	0	0	0	①	1	1	
	X	X	0	0	0	①	1	1	
	X	X	ⓧ	①	①	①	1	1	
	X	X	1	1	1	1	1	1	
X	X	1	1	1	1	1	1	1	
$\nearrow$	1	1							
	1	1							
	1								

Fig. 1. A fragment of c-table with a block of zeros being discovered. X — an entry already calculated; 1 — an entry unknown as yet; 0 — an entry found to be 0 earlier; ⓧ— an entry just calculated and found to be zero; ①— entries put to 0 to form a square block with ⓧ and with 0's; $\nearrow$ and $\swarrow$ indicate a paradiagonal under consideration.

look, in the last antidiagonal calculated, for two minima — the deepest one and the next one with respect to depth. Next, we look for two analogous minima in an antidiagonal calculated using two orders of the series less — two, because only there can elements of c-table correspond to the same paradiagonal. If the two minima found there (if there is more than one), lie on the same paradiagonals as the corresponding minima in the last antidaigonal, the first of our criteria (for finding a true valley) is satisfied. Then we look for an antidiagonal before the last one and perform the same search there. If it is equally successful, and moreover if paradiagonals along valleys found this way in the last and before the last, antidaigonals, are neighboring ones, we decide that , actually, valleys (or one valley) have been found. If any of this criteria is violated, coordinates of elements at two deepest minima (in absolute values) in the last antidiagonal, returned by the routine, are given negative signs (they are returned via an array nv; $nv(1,i)$, $i = 1,2$, contain coordinates of a shallower minimum, and $nv(2,i)$, $i = 1,2$, contain coordinates of absolute minimum). It is a signal for a user that the program cannot decide itself where the true valley is (if it can be found at all) and a human inspection of c-table is, probably, necessary. To facilitate a decision, program returns also coordinates of elements lying in supposed valleys on an antidiagonal before the last one (nnv). If only one valley is found, coordinates of the other one are put to zeros. We have applied our routine to c-table for $-1/z * \log(1-z)$ with obvious, excellent, results. For thirty coefficients we got $nv(1,i) = (0,0)$ — only one valley found; $nv(2,i) = (15,15)$ — in this, 29 th, order we can calculate determinants for Padé approximants of the next order and for this (Stieltjes) function the best is one $[15/15]$. One order back, the best would be $[14/15]$ and this is in prefect agreement with values of nnv: $(0,0)$ — again only one valley — and $(14,15)$. We have also applied our routine to the rational function used above for a discussion of precision problems. An interesting observation we made here is, that the valley pronounces itself already before an order, corresponding to a sum of

degrees of its numerator and denominators, is reached and also that the valley continues into the area of c-table, which should be a block if we could work with infinite precision. This last fact can easily be attributed to an appearance of Froissart pairs [2] which, at each next order, introduce one more zero to the numerator and to the denominator, perturbing only slightly the rational function being the origin of the series. It could be, therefore, expected in advance that such perturbed original function would be the best Padé approximant at any order. Nevertheless it is reassuring that it, actually, is the case. To demonstrate both these facts clearly we present a part of the c-table calculated for a rational function of the type $(7, 4)$ using the ascending algorithm (Table 5).

Table 5

	1	2	3	4	5	6	7	8
1	6.67D-01	1.11D-01	3.33D-01	8.89D-01	7.04D-01	3.39D-01	5.46D-01	9.01D-01
2	3.33D-01	-2.10D-01	1.23D-02	5.56D-01	1.94D-01	-2.69D-01	-7.62D-03	3.86D-01
3	4.81D-01	3.59D-01	3.50D-01	3.44D-01	2.66D-01	2.18D-01	1.90D-01	1.66D-01
4	-3.83D-01	1.88D-01	-9.28D-02	4.60D-02	-2.28D-02	1.13D-02	-5.61D-03	2.78D-03
5	-8.64D-02	-4.06D-04	-8.38D-05	-9.95D-06	-9.89D-07	-7.85D-08	-2.48D-09	8.97D-10
6	-2.06D-02	-3.76D-05	-3.21D-08	3.51D-10	-8.59D-12	3.29D-13	-1.36D-14	5.68D-16
7	-5.33D-03	-1.86D-06	-1.70D-10	1.53D-14	4.21D-17	1.15D-19	3.17D-22	8.70D-25
8	-1.47D-03	-6.77D-08	-1.79D-12	2.11D-17	4.73D-34	1.19D-37	-4.28D-42	-1.75D-43
9	-4.20D-04	-1.05D-09	-2.72D-14	2.89D-20	-5.93D-38	1.39D-55	6.56D-59	3.35D-62
10	-1.20D-04	1.52D-10	-4.30D-16	3.96D-23	-1.07D-42	3.28D-59	8.80D-77	3.29D-81
11	-3.41D-05	2.72D-11	-6.59D-18	5.43D-26	2.19D-44	8.39D-63	-1.65D-81	3.39D-99

As can be seen, starting from 6th order the smallest elements in c-table are: $(5, 2)$, $(6, 2)$, $(6, 3)$, $(7, 3)$, $(7, 4)$, $(8, 4)$, $(8, 5)$ - left upper corner of the block, $(9, 5)$, $(9, 6)$, $(10, 6)$, $(10, 7)$ etc. It must also be remarked that $(n + 4, n)$ and $(n + 3, n + 1)$ differ by a factor of 2, while $(n + 3, n)$ is smaller at least two orders of magnitude from $(n+4, n-1)$ and $(n+2, n+1)$, when we are outside of the block, and many orders of magnitude, inside it. Our routine finds this valley without any problem. To demonstrate how it behaves in an ambiguous situation we show results for a series studied by Van Dyke and discussed also in [2, p. 414]. c-Table for this series is given in Table 6 (we skipped 0 th row and 0 th column).

Table 6

	1	2	3	4	5	6	7	8
1	-7.50D-01	0.00D + 00	1.25D-01	-2.34D-01	5.86D-02	-1.95D-03	-1.13D-01	5.57D-02
2	5.62D-01	9.37D-02	1.56D-02	4.76D-02	2.97D-03	6.66D-03	1.30D-02	3.96D-04
3	-2.97D-01	-1.16D-01	-3.37D-02	-9.47D-03	-5.26D-03	-2.88D-03	-1.46D-03	
4	3.63D-01	3.72D-02	2.46D-03	-1.84D-03	1.16D-04	9.03D-05		
5	-3.19D-01	-4.19D-03	-2.21D-03	-3.28D-04	-3.42D-05			
6	2.92D-01	-1.85D-02	1.42D-03	-1.75D-05				
7	-3.25D-01	1.75D-02	-7.69D-04					
8	3.02D-01	-3.00D-03						
9	-2.90D-01							

There are two minima on the last diagonal: $(2, 8)$ and $(6, 4)$, but none of them seems to lie in a valley going back, neither there are two minima in the earlier order — only one at $(4, 5)$, again having no continuation back. Our routine gives for $nv(1, i)$ $(-2, -8)$ and for $nv(2, i)$ $(-6, -4)$. For nnv it gives $(0, 0)$ and $(4, 5)$. This values indicate clearly, even without looking at c-table, that the situation is highly obscure and without using higher order coefficients, one can make only guesses — it is exactly, what we would like that the routine of this type could say in such a situation.

Conclusion. This paper analyzes the effects of number representations in the computer and shows how we can to control the numerical errors due to the stability and to the precision. Unfortunately using certain modern packages we have no information about the number representation and then we can not say something about the results.

1. *Gragg W. B.* The Padé table and its relation to certain algorithms of numerical analysis // SIAM Rev. – 1972. – **14**. – P. 1 – 62.
2. *Gilewicz J.* Padé Approximants // Lect. Notes Math. – 1978. – **667**.
3. *Baker G. A. (Jr.).* Essentials of Padé Approximants. – Acad. Press, 1975.
4. *Gilewicz J., Magnus A.* Valleys in c-table // Padé Approxim. and Appl. Proc. (Antwerp 1979) (Lect. Notes Math.) – 1979. – **765**. – P. 135 – 149.
5. *Gilewicz J.* From a numerical technique to a method in the Best Padé Approximation // Orthogonal Polynomials and their Appl. / Ed. J. Vinuesa (Lect. Notes in Pure and Appl. Math.). – 1989. – **117**. – P. 35 – 51.
6. *Gilewicz J., Leopold E.* Subroutine CTABLE for Best Padé Approximant detection. – Marseille: Centre Phys. Théor., CNRS, 1981. – Rep. CPT-81/P.1324.
7. *Gilewicz J.* Computation of the c-table related to the Padé approximation. – Marseille: Centre Phys. Théor., CNRS, 1991. – Rep. CPT-91/P. 2601.
8. *Bailey's* Fortran multiprecision package, http://crd.lbl.gov/∼dhbailey/mpdist/
9. *Vignes J.* Review of stochastic approach to round-off error analysis and its applications // Math. and Comput. Simulat. – 1988. – **30**. – P. 481 – 491.
10. *Guziński W.* PADELIB: program library for Padé approximation (in Polish) // Inst. Nucl. Res. – Warszawa, 1978.
11. *Paszkowski S.* Evaluation of C-table // J. Comput. and Appl. Math. – 1992. – **44**. – P. 219 – 233.

Received 14.12.09