Construction of Recurrence Relations for the Jacobi Coefficients, Using Maple

S. Lewanowicz and J. Witkowski*

Institute of Computer Science, University of Wrocław, Wrocław, Poland

Last modified in February 2000

1 Introduction

In the field of numerical analysis, one frequently needs to determine the coefficients $a_k^{(\alpha,\beta)}[f]$ in the expansion of a given function f into a uniformly convergent series of Jacobi polynomials,

$$f(x) = \sum_{k=0}^{\infty} a_k^{(\alpha,\beta)}[f]\, P_k^{(\alpha,\beta)}(x) \qquad (-1 \le x \le 1), \tag{1}$$

where $P_k^{(\alpha,\beta)}(x)$ $(\alpha,\, \beta > -1)$ is the usual notation for the kth Jacobi polynomial (cf. [2], Vol. II, §10.8, or [5], Vol. I, Ch. 8). The particular case of $\alpha = \beta = -\frac{1}{2}$ is closely related to the Chebyshev series expansion,

$$f(x) = \frac{1}{2} c_0[f]\, T_0(x) + \sum_{k=1}^{\infty} c_k[f]\, T_k(x) \qquad (-1 \le x \le 1), \tag{2}$$

partial sums of which are known to be an extremely powerful approximation tool (see, e.g., [5], Ch. 17).

Sometimes, by a stroke of luck, closed forms for the *Jacobi coefficients* $a_k^{(\alpha,\beta)}[f]$ are available. More frequently, however, they are computed from a recurrence relation of the form

$$\Lambda a_k^{(\alpha,\beta)}[f] = \vartheta(k), \tag{3}$$

judiciously employed. Here Λ is a difference operator,

$$\Lambda := \sum_{j=u}^{u+r} \lambda_j(k)\, E^j, \tag{4}$$

where $\lambda_u, \lambda_{u+1}, \ldots, \lambda_{u+r}$ $(\lambda_u \not\equiv 0,\ \lambda_{u+r} \not\equiv 0)$ are known rational functions in k, $u \in \mathbf{Z}$, $r \in \mathbf{Z}_+$ is referred to as the *order* of Λ, and E^j $(j \in \mathbf{Z})$ is the jth shift operator, acting on the variable k:

$$E^j \mu(k) = \mu(k+j)$$

for any function $\mu : \mathbf{Z} \to \mathbf{Z}$. (We write I for E^0.) See, e.g., [5], Vol. II, §12.5, [6], Ch. 15 or [7], Ch. 11. Needless to say, it is important to have a recurrence relation (3) of the lowest possible order (i.e., to minimize the order of Λ).

*`Stanislaw.Lewanowicz@ii.uni.wroc.pl`

Recently (see [3]), an algorithm was proposed for constructing a relation of the form (3) under the assumption that the function f satisfies a linear differential equation

$$\mathbb{P}_n f \equiv \sum_{i=0}^{n} p_{ni}(x)\, \mathbb{D}^i f(x) = q(x) \tag{5}$$

of order n, where $\mathbb{D} := \dfrac{d}{dx}$ is the ordinary differentiation operator, the coefficients $p_{n0},\, p_{n1}, \ldots, p_{nn}$ are polynomials, and the coefficients $a_k^{(\alpha,\beta)}[q]$ exist and are known. Final result of the application of this algorithm is a pair P, L of difference operators such that the following identity holds:

$$L b_k^{(\alpha,\beta)}[f] \;=\; P b_k^{(\alpha,\beta)}[q]\,. \tag{6}$$

Here we use the notation

$$b_k^{(\alpha,\beta)}[f] \;:=\; \omega(k) a_k^{(\alpha,\beta)}[f]\,, \tag{7}$$

where

$$\omega(k) := \frac{\Gamma(k+\alpha+1)}{\Gamma(k+\lambda)(2k+\lambda-1)_3} \qquad (\lambda := \alpha+\beta+1). \tag{8}$$

The symbol $(c)_m$ denotes the shifted factorial: $(c)_0 = 1$, $(c)_m = c(c+1)\ldots(c+m-1)$ $(m>0)$. Now, the recurrence relation (3) holds for

$$\Lambda := L\,(\omega(k) I) \;\text{ and }\; \vartheta(k) := P\left(\omega(k) a_k^{(\alpha,\beta)}[q]\right). \tag{9}$$

A recurrence relation for the *Chebyshev coefficients* $c_k[f]$ can be obtained from (6) for $\alpha = -\frac{1}{2}$, $\beta = -\frac{1}{2}$, using $b_k^{(-\frac{1}{2},-\frac{1}{2})}[f] = (4k^2-1)^{-1} c_k[f]$.

It can be shown that the order of the resulting recurrence relation is

$$\operatorname{ord}(P) + 2 \max_{0\le i\le n,\, p_{ni}\not\equiv 0} (\deg p_{ni} - i), \tag{10}$$

where $\operatorname{ord}(P)$ denotes the order of the operator P. See, e.g., [3].

Briefly, the **algorithm can be summarized as follows**:

First, the differential expression $\mathbb{P}_n f$ in the left-hand side of the equation (5) is transformed to the *canonical form*

$$\mathbb{P}_n f = \sum_{i=0}^{n} \mathbb{Q}_i(q_i f), \tag{11}$$

where the special-form differential operators $\mathbb{Q}_0, \mathbb{Q}_1, \ldots, \mathbb{Q}_n$ and polynomials $q_0,\, q_1, \ldots, q_n$ are formed recursively.

Second, difference operators $Q_0,\, Q_1, \ldots, Q_n$ and $M_0,\, M_1, \ldots, M_n$ are constructed such that the following identities hold:

$$Q_i b_k^{(\alpha,\beta)}[\mathbb{Q}_i f] \;=\; M_i b_k^{(\alpha,\beta)}[f] \qquad (i = 0, 1, \ldots, n). \tag{12}$$

Third, the operator P (cf. (6)) is obtained as a common multiple of the operators $Q_0,\, Q_1, \ldots, Q_n$. Moreover, the operators $Z_0,\, Z_1, \ldots, Z_n$ are constructed such that

$$P = Z_i Q_i \qquad (i = 0, 1, \ldots, n).$$

Last, the operator L in the left-hand side of (6) is obtained in the form

$$L = \sum_{i=0}^{n} Z_i M_i q_i(X). \tag{13}$$

Here X is a known second-order difference operator (cf. (15). For details, see Appendix A.

2 Remarks on the Maple implementation

The algorithm was implemented in Maple V, version for DOS operating system (see Appendix B for the full listing). The main program named `recrel` accepts the following variable-length list of arguments:

- type of the recurrence relation: 0 for the general (Jacobi) case, 1 for the Chebyshev case;
- function of integer variable, returning k-th Jacobi coefficient of the right-hand side of the differential equation;
- single scalar variable;
- sequence of the coefficients of the left-hand side of the differential equation, given in the ascending order, as expressions depending on the preceding variable.

The program simply prepares some global data and then performs four function calls. First of them is `canonical_form`, dealing with differential operators. It roughly covers Step 1 and part of Step 3 of the algorithm (see Appendix A). Next, the functions `P_operator` and `L_operator` compute the main components of the right- and left-hand side, respectively, of the equation (6); the results are stored in the global variables `wP` and `wL`. The last call, `scalar_form`, evaluates the components of the scalar form of the recurrence relation.

During the development of the program, some problems were encountered, which had to be solved with nonstandard methods. Although most of these problems were caused by the limitations of the DOS operating system, some of them were probably bugs of the used Maple version .

2.1 Differential operators

The algorithm uses only linear ordinary differential operators with polynomial coefficients.

In the Maple language one can define function acting as a differential operator (for instance, the built-in operator `D` is defined in such a way) and then perform all calculations with generic Maple functions. However this approach has several drawbacks. Here are the most important of them:

- there is no simple way to define differential operator $\mathbb{Q}$ from an expression Q depending on function f and variable x so that $\mathbb{Q}(f)(x) = Q$;
- evaluating the product of operators sometimes leads to a wrong result, e.g. `(A@@n)@(B@@m)` simplifies to `A@@(n+m)`;
- the calculations are very time- and memory-consuming; what is more, on the DOS machines, more complicated problems could result in hanging the operating system.

For these reasons the differential operators are represented as Maple tables. Coefficients of powers of $\mathbb{D}$ operator are stored in table elements numbered from `0` to the operator order. The order itself is stored in an element with the number `-1`. Every coefficient is a polynomial expression depending on a variable, the name of which is stored in an element numbered `-2`. For example, table `rT` represents operator $\sum_{i=0}^{[-\ ]} \mathtt{rT}[\mathtt{i}]\mathbb{D}^i$, where `rT[i]` are polynomials of variable `rT[-2]`. For the sake of readability, the names of operators start with lowercase `r` letter.

Of course, all the built-in Maple functions acting on operators had to be replaced by the ones defined anew. For instance,

`oper_create` creates an operator from the name of scalar variable and the list of coefficients given as expressions of this variable;
`&@` computes composition of two operators;
`&@@` computes integer power of an operator;
`&-` computes difference of two operators;
`Q_app_qf` given operator $\mathbb{Q}$ and polynomial q, returns the operator $\mathbb{S}$ such that $\mathbb{S}(f)(x) = \mathbb{Q}(q \cdot f)(x)$.

The step 1, as described in Appendix A, is performed by the function `canonical_form`. It uses global variables `rP.i`, `rQ.i`, `q.i`, `delta.i`, `deltap.i`, `eps.i`, `t.i`, `s.i`, `r.i` and `v.i` for storing the computed values of $\mathbb{P}_i$, $\mathbb{Q}_i$, q_i, δ_i, δ'_i, ε_i, t_i, s_i, r_i and v_i, respectively. Given a polynomial, the function `polynomial_decomposition` computes all the elements of its special representation, i.e. ϱ, σ, ε and $w_i(x)$. The basic operators $\mathbb{U}$, $\mathbb{J}$, $\mathbb{V}_\varepsilon$, $\mathbb{K}_\varepsilon$ are given as the constants `rU`, `rJ`, `rV[eps]` and `rK[eps]`, respectively. Finally, the constants needed in the Step 3 of the algorithm (namely ε, e_1, e_{-1}, e, d_1, d_{-1} and d) are also computed here (as `eps`, `e[1]`, `e[-1]`, `e[0]`, `d[1]`, `d[-1]`, `d[0]`, respectively).

2.2 Difference operators

The algorithm uses only linear difference operators of the form (4). Such operators are represented as a power series of global variable `wE`. Their names in the program start with the lowercase `w`, while `wE` itself stands for the shift operator E. These operators must also have specially defined functions to act on them:

> `&O` computes composition of two operators;
> `&OO` computes integer power of an operator;
> `q_app_wX` evaluates polynomial of an operator variable.

The difference operators, defined in the Step 3 of the algorithm, are given as the global constants or functions `wD`, `wX`, `wR`, `wT`, `wU`, `wA`, `wS`, `wW`, `wY`, `wZ`, `wM`. For computing the operators `wP` and `wL`, two special functions are provided: `P_operator` and `L_operator`, respectively. Finally, function `scalar_form` is provided for simplifying the results. It has two input parameters: case number `case` and function `rhs_coeff`, returning kth Jacobi (or Chebyshev) coefficient of the function q constituting the right-hand side of the differential equation (5). The results are printed on the screen and stored in the global variables `lam` (table of coefficients of the operator Λ) and `theta` (right-hand side of the recurrence relation (3)).

2.3 Examples

We give two worked examples solved with Maple.

Example 1 We are looking for a recurrence relation for the coefficients $a_{n-1,k}$ in

$$W_{n-1}^{(\alpha,\beta)} = \sum_{k=0}^{n-1} a_{n-1,k} P_k^{(\alpha,\beta)},$$

where the $(n-1)$*th associated Jacobi polynomial* $W_{n-1}^{(\alpha,\beta)}$, given by

$$W_{n-1}^{(\alpha,\beta)} := \int_{-1}^{1} (x-t)^{-1}(1-x)^\alpha(1+x)^\beta[P_k^{(\alpha,\beta)}(x) - P_k^{(\alpha,\beta)}(t)]\,dt,$$

is known to satisfy the fourth-order differential equation

$$\mathbb{P}_4 f \equiv \sum_{i=0}^{4} p_{4i}(x)\mathbb{D}^i f(x) = 0, \tag{14}$$

where

$$\begin{aligned}
&p_{44}(x) = (x^2-1)^2, \qquad p_{43}(x) = 10x(x^2-1),\\
&p_{42}(x) = A\,(x^2-1) - [(\lambda+1)x+\delta][(\lambda-3)x+\delta] + 12,\\
&p_{41}(x) = 3[5-\lambda^2-2n(n+\lambda)]x - 3\delta(\lambda-1), \qquad p_{40}(x) = (n^2-1)[(n+\lambda)^2-1],
\end{aligned}$$

and $A = 20 - 2(n+1)(n+\lambda-1)$, $\delta = \alpha-\beta$. See, e.g., [4].

Obviously, $a_{n-1,k} = a_k^{(\alpha,\beta)}[W_{n-1}^{(\alpha,\beta)}]$ $\quad (k = 0, 1, \ldots, n-1)$.

The Maple session would be as follows:

```
> A := 2*(10-(n+1)*(n+lambda-1)):
> delta := alpha-beta:
> p44 := (x^2-1)^2:  p43 := 10*x*(x^2-1):
> p42 := A*(x^2-1)-((lambda+1)*x+delta)*((lambda-3)*x+delta)+12:
> p41 := 3*(5-lambda^2-2*n*(n+lambda))*x-3*delta*(lambda-1):
> p40 := ((n+lambda)^2-1)*(n^2-1):
> recrel(0, k->0, 'x', p40, p41, p42, p43, p44);
```

```
   lam[-1] = - (lambda + n - k) (k + n) (2 k + lambda + 1) (k + lambda)

       (k - 1 + lambda) (2 k + 2 + lambda)

     lam[0] = (lambda - 2 alpha - 1)

                                           2         2
         (3 k lambda + n lambda + lambda + 3 k  - 1 + n ) (k + lambda)

         (2 k - 2 + lambda) (2 k + 2 + lambda)

    lam[1] = - (k + alpha + 1) (k + lambda - alpha) (2 lambda + k + n)

        (k + lambda - n) (2 k + lambda - 1) (2 k - 2 + lambda)

                              theta = 0
```

Thus, the recurrence relation for $a_{n-1,k}$ is homogeneous and can be written as

$$
\begin{aligned}
&(n-k+\lambda)(n+k)(k+\lambda-1)_2(2\,k+\lambda+1)_2 a_{n-1,k-1}\\
&\quad -(\lambda-2\,\alpha-1)(k+\lambda)(2\,k+\lambda-2)(2\,k+\lambda+2)[(n+1)(n+\lambda-1)+3\,k(k+\lambda)]a_{n-1,k}\\
&\quad +(k+\alpha+1)(k+\lambda-\alpha)(k-n+\lambda)(k+n+2\,\lambda)(2\,k+\lambda-2)_2 a_{n-1,\,k+1}=0\\
&\hfill (k=1,2,\ldots,n-1;\, a_{n-1,\,n}=0).
\end{aligned}
$$

Example 2 Let us construct the recurrence relation for the Chebyshev coefficients $c_k[f]$ of the function $f(x)=x\exp x$. This function satisfies the first-order equation $xf'-(x+1)f=0$, however, a better result is obtained using the third-order equation (see [3])

$$\mathbb{P}_3 f \equiv (x^2-1)f'''(x)-(x^2-3x-1)f''(x)-(4x-1)f'(x)-3f(x)=0.$$

```
> alpha := -1/2: beta := -1/2: lambda := 0:
> recrel(1, <0>, 'x', -3, 1-4*x, 1+3*x-x^2, x^2-1):
```

```
                           2
             lam[-1] = - k  - k - 1

                              3
                 lam[0] = 2 k

                         2
               lam[1] = k  - k + 1

                   theta = 0
```

Thus, the needed equation has the form

$$-(k^2+k+1)c_{k-1}[f]+2\,k^3 c_k[f]\;+(k^2-k+1)c_{k+1}[f]=0.$$

Appendix A: Detailed description of the algorithm

Definition 3 (Basic differential operators) We shall use the following first- and second-order differential operators:

$$\mathbb{U} := (x^2-1)\mathbb{D} + [(\lambda+1)x+\alpha-\beta]\,\mathbb{I}, \qquad \mathbb{J} := \mathbb{U}\mathbb{D}\,,$$
$$\mathbb{V}_\varepsilon := (x+\varepsilon)\mathbb{D} + \nu_\varepsilon\mathbb{I}, \qquad \mathbb{K}_\varepsilon := \mathbb{V}_\varepsilon\mathbb{D}\,,$$

where $\varepsilon = \pm 1$, $\nu_\varepsilon := \frac{1}{2}[\lambda+1-\varepsilon(\alpha-\beta)]$ and $\lambda := \alpha+\beta+1$.

Definition 4 (Basic difference operators) Let us define the difference operators D and X by

$$\begin{aligned} D &:= \delta_0(k)E^{-1} + \delta_1(k)I + \delta_2(k)E, \\ X &:= \xi_0(k)E^{-1} + \xi_1(k)I + \xi_2(k)E, \end{aligned} \tag{15}$$

respectively, where

$$\begin{aligned} \delta_0(k) &:= 2(k+\alpha)(2k+\lambda-3)\gamma(k), & \xi_0(k) &:= k\delta_0(k), \\ \delta_1(k) &:= 2(\alpha-\beta)(2k+\lambda)\gamma(k), & \xi_1(k) &:= \tfrac{1}{2}(1-\lambda)\delta_1(k), \\ \delta_2(k) &:= -2(k+\beta+1)(2k+\lambda+3)\gamma(k), & \xi_2(k) &:= -(k+\lambda)\delta_2(k), \end{aligned}$$

and $\gamma(k) := 1/(2k+\lambda-1)_3$.

Further, given $\varepsilon \in \{-1, 1\}$, we define the following sequences of difference operators:

$$A_m^{(\varepsilon)} := I - 2\varepsilon \frac{k+\nu_\varepsilon}{\delta_0(k+1)(2k+\lambda+m+1)_2} E \qquad (m = 0, 1, \ldots),$$
$$S_{ij}^{(\varepsilon)} := \begin{cases} A_i^{(\varepsilon)} A_{i-1}^{(\varepsilon)} \cdots A_j^{(\varepsilon)} & (i \geq j), \\ I & (i < j), \end{cases}$$
$$P_h^{(\varepsilon)} := S_{h-1,0}^{(\varepsilon)} \qquad (h = 0, 1, \ldots),$$
$$R_m^{(\varepsilon)} := \delta_0(k)E^{-1} + 2\varepsilon \frac{k+m+\nu_{-\varepsilon}}{(2k+\lambda+m)_2} I \qquad (m = 0, 1, \ldots),$$
$$T_{ij}^{(\varepsilon)} := \begin{cases} R_i^{(\varepsilon)} R_{i+1}^{(\varepsilon)} \cdots R_j^{(\varepsilon)} & (i \leq j), \\ I & (i > j), \end{cases}$$
$$U_h^{(\varepsilon)} := T_{h-1,0}^{(\varepsilon)} \qquad (h = 0, 1, \ldots).$$

Also, we shall use the notation

$$\mu_{2j+\omega}^{(\varepsilon)}(k) := (k+\nu_\varepsilon)_{j+\omega} \Big/ \prod_{h=0}^{j} \delta_0(k+h) \qquad (\varepsilon = \pm 1;\ j = 0, 1, \ldots;\ \omega = 0, 1).$$

Algorithm (Construction of the recurrence relation for the Jacobi coefficients)
Let the function f satisfy the differential equation

$$\mathbb{P}_n f \equiv \sum_{i=0}^{n} p_{ni}(x)\,\mathbb{D}^i f(x) = q(x)$$

of order n, where the coefficients $p_{n0}, p_{n1}, \ldots, p_{nn}$ are polynomials, and the coefficients $a_k^{(\alpha,\beta)}[q]$ exist and are known.

Step 1. *Define the differential operators $\mathbb{P}_i$ $(i = 0, 1, \ldots, n-1)$ and $\mathbb{Q}_j$ $(j = 1, 2, \ldots, n)$, and the polynomials $q_0, q_1, \ldots, q_n$ in the following recursive way. For $i = n, n-1, \ldots, 1$, given the operator $\mathbb{P}_i = \sum\limits_{j=0}^{i} p_{ij}\mathbb{D}^j$,*

- *represent the coefficient* p_{ii} *in the form* $p_{ii}(x) = (x^2-1)^{\varrho}(x+\varepsilon)^{\sigma} w_i(x)$ *with* $\varrho,\ \sigma \in \mathbf{Z}_+$, $\varepsilon \in \{-1,\ 1\}$ *and a polynomial* w_i *such that* $w_i(-1) \neq 0$, $w_i(1) \neq 0$, *then find out which of the following cases holds:*
  - *case A:* $\quad \varrho \geq m$,
  - *case B:* $\quad \varrho + \sigma \geq m > \varrho$,
  - *case C:* $\quad m > \varrho + \sigma$,

  where $m := \lfloor (i+1)/2 \rfloor$;
- *define*

$$q_i(x) := w_i(x) \cdot \begin{cases} (x^2-1)^{\varrho - m}(x+\varepsilon)^{\sigma} & \textit{(case A)}, \\ (x+\varepsilon)^{\varrho+\sigma-m} & \textit{(case B)}, \\ 1 & \textit{(case C)}; \end{cases}$$

- *define*

$$\delta_i := \begin{cases} i \bmod 2 & \textit{(case B)}, \\ 0 & \textit{(case A or C)}, \end{cases} \qquad \delta_i' := \begin{cases} i \bmod 2 & \textit{(case A)}, \\ 0 & \textit{(case B or C)}, \end{cases}$$

$$\varepsilon_i := \begin{cases} \varepsilon & (\sigma > 0), \\ 1 & (\sigma = 0), \end{cases} \qquad t_i := \min\{\varrho,\ \lfloor i/2 \rfloor\},$$

$$s_i := \min\{\sigma,\ \lfloor i/2 \rfloor - t_i\}, \quad r_i := i - 2s_i - 2t_i - \delta_i - \delta_i';$$

- *define the operator* $\mathbb{Q}_i$ *by*

$$\mathbb{Q}_i := \mathbb{D}^{r_i} \mathbb{K}_{\varepsilon_i}{}^{s_i} \mathbb{V}_{\varepsilon_i}{}^{\delta_i} \mathbb{J}^{t_i} \mathbb{U}^{\delta_i'};$$

- *define the operator* $\mathbb{P}_{i-1}$ *of degree* $i-1$ *as*

$$\mathbb{P}_{i-1} f := \mathbb{P}_i f - \mathbb{Q}_i(q_i f).$$

Step 2. *Define the difference operators*

$$M_0 := I, \qquad M_i := \mu_{v_i}^{(\varepsilon_i)}(k) E^{s_i} P_{\delta_i}^{(-\varepsilon_i)} \kappa^{t_i + \delta_i'}(k) D^{\delta_i'} \qquad (i \in \Omega;\ \kappa(k) := k(k+\lambda)).$$

Step 3. *Define*

$$\Omega := \{1, 2, \ldots, n\}, \qquad \Omega_\eta := \{i \in \Omega :\ \varepsilon_i = \eta\} \qquad (\eta = \pm 1),$$

$$v_i := 2s_i + \delta_i \quad (i \in \Omega),$$

$$e_\eta := \max_{i \in \Omega_\eta} r_i, \qquad d_\eta := \max_{i \in \Omega_\eta}(v_i + r_i) - e_\eta \qquad (\eta = \pm 1),$$

$$\varepsilon := \begin{cases} 1 & (d_1 + e_1 \geq d_{-1} + e_{-1}), \\ -1 & (d_1 + e_1 < d_{-1} + e_{-1}), \end{cases}$$

$$e := \max\{e_\varepsilon,\ d_{-\varepsilon} + e_{-\varepsilon}\}, \qquad d := d_\varepsilon + e_\varepsilon - e,$$

$$\gamma_i := d_\eta + e_\eta - v_i - r_i \qquad (i \in \Omega_\eta;\ \eta = \pm 1),$$

$$W^{(\eta)} := \begin{cases} T_{d, d_\varepsilon - 1}^{(\varepsilon)} & (\eta = \varepsilon), \\ P_d^{(\varepsilon)} D^{e - e_{-\varepsilon} - d_{-\varepsilon}} U_{d_{-\varepsilon}}^{(-\varepsilon)} & (\eta = -\varepsilon), \end{cases}$$

$$Y_i := T_{d_\eta,\, v_i + \gamma_i - 1}^{(\eta)} S_{v_i + \gamma_i - 1,\, v_i}^{(\eta)} \quad (i \in \Omega_\eta;\ \eta = \pm 1),$$

and

$$P := P_d^{(\varepsilon)} D^e,$$

$$Z_0 := P, \qquad Z_i := W^{(\eta)} Y_i \quad (i \in \Omega_\eta;\ \eta = \pm 1).$$

Step 4. *Define*

$$L = \sum_{i=0}^{n} Z_i M_i q_i(X).$$

Then the Jacobi coefficients $a_k^{(\alpha,\beta)}[f]$ *satisfy the recurrence relation*

$$\Lambda a_k^{(\alpha,\beta)}[f] = \vartheta(k),$$

where

$$\Lambda := L\left(\omega(k)I\right), \qquad \vartheta(k) := P\left(\omega(k)a_k^{(\alpha,\beta)}[q]\right),$$

with the above-obtained difference operators L and P, and ω defined in (8).

Appendix B: The Maple code

```
# Constructing recurrence equation for Jacobi series coefficients
# of a fuction which satisfies a linear differential equation,
# see S.Lewanowicz, Applic. Math. (Zastos. Mat.) 21 (1991), 303-326
#
# - Version 3.0 (System Maple V release 3 or 4),
# - Jerzy Witkowski, 04/04/1995; modified by Pawel Keller, 03/12/1999
# - all modifications commented with '# - '

# Naming conventions:
#    rX - differential operator X, represented as a Maple table:
#       -2     name of the scalar variable;
#       -1     order n;
#        0..n  expressions depending on above scalar variable,
#                representing operator coefficients;
#    wX - difference operator X, represented as a power series
#           of the global scalar variable wE.
#
# Calling sequence:
#    canonical_form() - operators rQ.i, polynomials q.i and
#       some global constants;
#    wP := rhside() - right-hand side operator wP;
#    wL := lhside() - left-hand side operator wL;
#    scalar_form() - scalar form of the recurrence relation for the
#       Jacobi coefficients:
#    recrel() - above statements wrapped together; parameters:
#       1 - type of the recurrence relation; 0 - general (Jacobi) case,
#                                            1 - Chebyshev case,
# -                                          2 - user defined case
# -                                             (using user 'alpha' & 'beta');
#       2 - function of integer variable, returning k-th Jacobi coefficient
#           of the right-hand side of the given differential equation;
#       3 - single scalar variable;
#       4..N+4 - coefficients of the left-hand side of the given
#           differential equation as a expressions depending on the
#           preceding variable.
#
# - The final result is stored in the global variable 'recurrence',
# - and has the following structure:
# -   array ( -2..1, [ u,
# -                    r,
# -                    array( u..u+r, [ lam[u], ... , lam[u+r] ] ),
# -                    theta
# -                  ] ),
# - where 'u' is the index of the first item, 'r' is the order
# - of the following resulting recurrence realtion:
# -    sum_{k=u}^{u+r} lam[k]*a(k) = theta
# - with unknowns a(k)'s,
# - where (if appears) lambda = alpha + beta + 1.
#
# - If the variable reccurence='noprint' when calling 'recrel'
# - procedure, the result is not printed - but it is
# - still stored in the 'recurrence' variable.
#
```

```
# Creation of a differential operator.
# Input:
#    1st argument - name of the scalar variable;
#    2nd .. (n+2)th - polynomials constituting operator coefficients,
#                     starting from the one of the 0th derivative;
# Output: differential operator.
oper_create := proc ()
   local i, rT;
   rT[-2] := args[1]; rT[-1] := nargs-2;
   for i from 2 to nargs do rT[i-2] := args[i] od;
   rT
end: # oper_create

# Unification of scalar variable of two operators.
oper_var := proc (rA, rB)
   local va, vb, i;
   va := rA[-2]; vb := rB[-2];
   if va<>vb then
      for i from 0 to rB[-1] do rB[i] := subs(vb=va,rB[i]) od;
      rB[-2] := va
   fi;
   va
end: # oper_var

# Composition of two differential operators
`&@` := proc (rA, rB)
   local i, j, k, m, rT, v, na, nb, n;
   v := oper_var(rB,rA);
   na := rA[-1]; nb := rB[-1]; n := na+nb;
   for i from 0 to n do rT[i] := 0 od;
   for i from 0 to na do
      for j from 0 to nb do
         m := i+j; rT[m] := rT[m] + rA[i]*rB[j];
         for k from 1 to i do
            m := m-1;
            rT[m] := rT[m] + rA[i]*binomial(i,k)*diff(rB[j],v$k)
   od od od;
   rT[-1] := n; rT[-2] := v;
   oper_simp(rT)
end: # `&@`

# Power of a differential operator
`&@@` := proc (rA, n)
   local i, rT;
   if n<0 then 'procname(args)'
   elif n=0 then oper_create(rA[-2],1)
   else rT := rA; for i from 2 to n do rT := rA &@ rT od; rT
   fi
end: # `&@@`

# Subtraction of differential operators
`&-` := proc (rA, rB)
   local na, nb, n, rT, i, v;
   v := oper_var(rA,rB);
   na := rA[-1]; nb := rB[-1];
   if na>nb then
      n := nb; rT[-1] := na;
      for i from n+1 to na do rT[i] := rA[i] od
   else
      n := na; rT[-1] := nb;
      for i from n+1 to nb do rT[i] := -rB[i] od
   fi;
   for i from 0 to n do rT[i] := rA[i]-rB[i] od;
   rT[-2] := v;
   oper_simp(rT)
end: # `&-`
```

```
# Simplifying a differential operator
oper_simp := proc (rA)
   local i, rT, a, n;
   rT := rA;
# - "n:=-1;" could lead to order -1 if rA is 0 operator
   n := 0;
##
   for i from 0 to rT[-1] do
      a := simplify(rT[i]);
      if a<>0 then n := i fi;
      rT[i] := a
   od;
   rT[-1] := n;
   rT
end: # oper_simp

# Input:
#    rQ - differential operator,
#    q - function.
# Output:
#    rQ(q*f), where f denotes functional parameter of rQ.
Q_app_qf := proc (rQ, q)
   local rT, i, j, n, v;
   n := rQ[-1]; v := rQ[-2]; rT[0] := rQ[0]*q(v);
   for i from 1 to n do rT[i] := 0 od;
   for i from 1 to n do
      for j from 0 to i-1 do
         rT[j] := rT[j] + rQ[i]*binomial(i,j)*diff(q(v),v$(i-j))
      od;
      rT[i] := rT[i] + rQ[i]*q(v)
   od;
   rT[-1] := n; rT[-2] := v;
   rT
end: # Q_app_qf

# Input:
#    q - polynomial,
#    wX - differential operator
# Output:
#    q(wX) - result is computed with operator composition used
#            instead of multiplication.
q_app_wX := proc (q, wX)
   local x, qx, deg, i, wS;
   qx := collect(simplify(q(x)),x);
   deg := degree(qx,x); wS := coeff(qx,x,deg);
   for i from deg-1 by -1 to 0 do
      wS := (wS &O wX) + coeff(qx,x,i)
   od;
   simplify(wS)
end: # q_app_wX

# Representing a polynomial in the form
#    p(x) = (x^2-1)^ro * (x+eps)^sig * w(x),
#    where eps=1 or eps=-1, and w(-1)<>0 and w(1)<>0.
# Input:
#    p - polynomial.
# Output:
#    ro, sig, eps - constants,
#    w - polynomial.
polynomial_decomposition := proc (p, x, ro, eps, sig, w)
   local gm, gp, gpm, dgp, dgm, deg, px;
   px := collect(simplify(p),x);
   deg := degree(px,x);
   gpm := gcd(px,collect((x^2-1)^deg,x));
   gm := gcd(gpm,(x-1)^deg); gp := gcd(gpm,(x+1)^deg);
   dgm := degree(gm,x); dgp := degree(gp,x);
   if dgp>=dgm then ro := dgm; sig := dgp-dgm; eps := 1
```

```
   else ro := dgp; sig := dgm-dgp; eps := -1
   fi;
   w := unapply(px/gm/gp,x);
   NULL
end: # polynomial_decomposition

# Coefficients of a finite power series.
# Input:
#    q - power series;
#    x - scalar variable.
# Output:
#    the following list:
#       1st element - the lowest degree of x;
#       2nd element - the highest degree of x;
#       3rd element - common denominator of coefficients;
#       4th and next ones - numerators of coefficients.
series_coeffs := proc (q, x)
   local ncoeff, dcoeff, ddeg, ndeg, c, res, i;
   ncoeff := collect(numer(q),x); dcoeff := collect(denom(q),x);
   ddeg := degree(dcoeff,x); c := coeff(dcoeff,x,ddeg);
   if simplify(c*x^ddeg-dcoeff)<>0 then
      ERROR(`Denominator must be a monomial.`)
   fi;
   ndeg := degree(ncoeff,x);
   res := -ddeg, ndeg-ddeg, c;
   for i from 0 to ndeg do res := res, coeff(ncoeff,x,i) od;
   [res]
end: # series_coeffs

# Composition of two difference operators (given as power series
# of global variable wE; every coefficient is a function of global variable k).
`&O` := proc (wA, wB)
   local A, deg, ifrom, ito, dcoeff, res, i;
   global wE, k;
   A := series_coeffs(wA,wE);
   deg := A[1]; ifrom := 4; ito := 4+A[2]-deg;
   dcoeff := A[3]; res := 0;
   for i from ifrom to ito do
      res := simplify(res + A[i]*wE^deg*subs(k=k+deg,wB));
      deg := deg+1
   od;
   res/dcoeff
end: # `&O`

# Power of a difference operator
`&OO` := proc (wA, n)
   local res, i;
   if n>=0 then
      res := 1;
      for i from 1 to n do res := res &O wA od;
      res
   else 'procname(args)'
   fi
end: # `&OO`

# Factorizing series coefficients
coeffs_factor := proc (wT)
   local n, m, i, wW, wR;
   global wE;
   wW := collect(wT,wE); n := ldegree(wW,wE); m := degree(wW,wE);
   wR := 0;
   for i from n to m do  wR := wR + factor(coeff(wW,wE,i))*wE^i od;
   wR
end: # coeffs_factor

# Global formulae
beta := lambda-alpha-1: delta := alpha-beta: sigma := alpha+beta+2:
```

```
rD := oper_create('x',0,1):
rU := oper_create('x',(lambda+1)*x+alpha-beta,x^2-1):
rJ := rU &@ rD: ni := eps -> (lambda+1+eps*(beta-alpha))/2:
rV[1] := oper_create('x',ni(1),x+1): rV[-1] := oper_create('x',ni(-1),x-1):
rK[1] := rV[1] &@ rD: rK[-1] := rV[-1] &@ rD:
Fact := proc (c, m) # c*(c+1)*(c+2)*...*(c+m-1)
   local s, i;
   if not type(m,'numeric') then 'procname(args)'
   else
      s := 1;  for i from 0 to m-1 do s := s*(i+c) od;  s
   fi
end: # Fact
fact2 := k -> k*(k+1):  fact3 := k -> k*(k+1)*(k+2):
gam := k -> fact3(2*k+lambda-1):
Del0 := k -> 2*(k+alpha)*(2*k+lambda-3)/gam(k): del0 := Del0(k):
del1 := 2*(alpha-beta)*(2*k+lambda)/gam(k):
del2 := -2*(k+beta+1)*(2*k+lambda+3)/gam(k):
ksi0 := k*del0: ksi1 := (1-lambda)*del1/2: ksi2 := -(k+lambda)*del2:
wD := del0/wE + del1 + del2*wE:   wX := ksi0/wE + ksi1 + ksi2*wE:
Ro := (eps, m) -> 2*eps*(k+m+ni(-eps))/fact2(2*k+lambda+m):
wR := (eps, m) -> simplify(del0/wE + Ro(eps,m)):
wT := proc (eps, i, j)
   if i>j then 1
   else simplify(wR(eps,i) &O wT(eps,i+1,j))
   fi
end: # wT
wU := (eps, h) -> simplify(wT(eps,0,h-1)):
mi := proc (eps, i)
   local j, s, h, j2;
   global k;
   j := iquo(i,2); j2 := j+j;
   if i=j2 then
      s := 1;
      for h from 1 to j do s := s * Del0(k+h) od;
      Fact(k+ni(eps),j)/s
   else (k+j+ni(eps))*mi(eps,j2)
   fi
end: # mi
tau := (eps, m) -> -2*eps*(k+ni(eps))/(Del0(k+1)*fact2(2*k+lambda+m+1)):
wA := (eps, m) -> simplify(1 + tau(eps,m)*wE):
wS := proc (eps, i, j)
   if i<j then 1 else simplify(wA(eps,i) &O wS(eps,i-1,j)) fi
end: # wS
wPP := (eps, h) -> simplify(wS(eps,h-1,0)):
wW := proc (eta)
   if eta=eps then wT(eps,d[0],d[eps]-1)
   else wPP(eps,d[0]) &O (wD &OO (e[0]-e[-eps]-d[-eps])) &O wU(-eps,d[-eps])
   fi
end: # wW
Gam := i -> d[eps.i]+e[eps.i]-v.i-r.i:
wY := i -> simplify(wT(eps.i,d[eps.i],v.i+Gam(i)-1) &O wS(eps.i,v.i+Gam(i)-1,v.i)):
wZ := proc (i)
   if i=0 then simplify(wPP(eps,d[0]) &O (wD &OO e[0]))
   else simplify(wW(eps.i) &O wY(i))
   fi
end: # wZ
kappa := k*(k+lambda):
wM := i -> simplify((mi(eps.i,v.i)*(wE^s.i)) &O
          wPP(-eps.i,delta.i) &O ((kappa^(t.i+deltap.i))*(wD &OO deltap.i))):

########################  Main part of the algorithm  #######################

# Step 1: rQ.i, q.i and part of Step 2: global constants
canonical_form := proc ()
   local i, m, omega, case_number, ro, sig, x, maxdeg, tmp, deg;
   global N, d, e, eps, r, rD, rJ, rK, rU, rV;
   d[1] := 0; d[-1] := 0; e[1] := 0; e[-1] := 0;
```

```
# - rP.N[0] may be 0 so 'maxdeg := degree(rP.N[0],rP.N[-2]);' is now
   maxdeg := degree(rP.N[N],rP.N[-2])-N;
##
   for i from N by -1 to 0 do
      m := trunc((i+1)/2); omega := i mod 2;
# - The requested item rP.i[i] may not exist (if rP.i[-1]<i),
# - so 'if rP.i[i]=0 then' was replaced by...
      if rP.i[i]=0 or rP.i[-1]<i then
##
         delta.i := 0;  deltap.i := 0;  eps.i := 1;  t.i := 0;  r.i := 0;
         s.i := 0;  v.i := 0;  rQ.i := oper_create('x',1);
# - The old 'q.i:=<0>;' doesn't work under release 4
         q.i := 0;
##
      else
         polynomial_decomposition(rP.i[i],rP.i[-2],'ro','eps','sig','w.i');
         if m<=ro then
            case_number := 1; delta.i := 0; deltap.i := omega;
         elif m<=ro+sig then
            case_number := 2; delta.i := omega; deltap.i := 0;
         else
            case_number := 3; delta.i := 0; deltap.i := 0;
         fi;
         if sig>0 then eps.i := eps else eps.i := 1 fi;
         t.i := min(ro,m-omega);  s.i := min(sig,m-t.i-omega);
         r.i := i-2*(s.i+t.i)-delta.i-deltap.i;  v.i := 2*s.i+delta.i;
         if case_number=1 then
            rQ.i := (rJ &@@ t.i) &@ (rU &@@ omega);
            q.i := unapply(w.i(x)*(x^2-1)^(ro-m)*(x+eps)^sig,x)
         elif case_number=2 then
            rQ.i := (rK[eps.i] &@@ s.i) &@ (rV[eps.i] &@@ omega) &@
                    (rJ &@@ ro);
            q.i := unapply(w.i(x)*(x+eps.i)^(ro+sig-m),x)
         else
            rQ.i := (rD &@@ r.i) &@ (rK[eps.i] &@@ sig) &@ (rJ &@@ ro);
            q.i := unapply(w.i(x),x)
         fi;
      fi;
      if i>0 then
         rP.(i-1) := rP.i &- Q_app_qf(rQ.i,q.i);
         if r.i>e[eps.i] then e[eps.i] := r.i fi;
         if v.i+r.i>d[eps.i] then d[eps.i] := v.i+r.i fi;
# -      (...)
      fi;
# - The below condition was excluded from (...) above
# - to cover the case of 'i=0'
      if rP.N[i]<>0 then
         deg := degree(rP.N[i],rP.N[-2])-i;
         if deg > maxdeg then maxdeg := deg fi;
      fi;
##
   od;
   d[1] := d[1]-e[1]; d[-1] := d[-1]-e[-1];
   if d[1]+e[1] >= d[-1]+e[-1] then eps := 1 else eps := -1 fi;
   tmp := d[-eps]+e[-eps];
   if e[eps] > tmp then e[0] := e[eps] else e[0] := tmp fi;
   d[0] := d[eps]+e[eps]-e[0]; r := d[0]+2*e[0]+2*maxdeg;
   NULL
end: # canonical_form

# Step 3: constructing the P operator
P_operator := proc ()
   coeffs_factor(wPP(eps,d[0]) &O (wD &OO e[0]))
end: # P_operator

# Step 4: constructing the L operator
L_operator := proc ()
```

```
   local i, s;
   global N, wX;
   s := 0;
   for i from 0 to N do
      s := simplify(s + (wZ(i) &O wM(i) &O q_app_wX(q.i,wX)))
   od;
   coeffs_factor(simplify(s))
end: # L_operator

# Scalar form of the recurrence relation
# Input:
#    case - 0 = general (Jacobi) case, 1 = Chebyshev case;
#    rhs_coeff - function of integer variable, returning k-th
#       Jacobi coefficient of the right-hand side of the equation;
# Output:
#    lam - table of coefficients of the operator Lambda,
#    theta - right-hand side of the recurrence relation.
scalar_form := proc (case, rhs_coeff)
   local lam, ff, u, j, nwdl, nwwm, theta, fact, low, up;
   global alpha, beta, d, e, k, lambda, r, wE, wL, wLc, wP;
   wL := collect(wL,wE); wP := collect(wP,wE);
   u := ldegree(numer(wL),wE)-degree(denom(wL),wE);
# - Recalculation of the value of r. The original formula (10) from the
# - corresponding paper doesn't work or is incorrectly implemented.
# - - Original recalculation of 'r' has been fixed.
# - - r := degree(wL,wE)-ldegree(wL,wE);
##
   lam := array(u..u+r);  # r - global variable
   for j from u to u+r do lam[j] := factor(coeff(wL,wE,j)) od;
# - 'if case = 0' changed to 'if case <> 1' as more cases are now available
   if case <> 1 then  # not Chebyshev case
##
      low := min(u,-e[0]);  up := max(e[0]+d[0],r+u);
      ff := array(low..up);  ff[low] := 1;
      for j from low+1 to up do ff[j] := (k+alpha+j)*ff[j-1]/(k+lambda+j-1) od;
      nwwm := 1;  nwdl := numer(lam[u]);
      for j from u to u+r do
         lam[j] := lam[j]*ff[j]/((2*k+2*j+lambda-1)*(2*k+2*j+lambda)*(2*k+2*j+lambda+1));
         nwwm := lcm(nwwm,denom(lam[j]));
         if j>u then nwdl := gcd(nwdl,numer(lam[j])) fi
      od;
      fact := nwwm/nwdl; theta := 0;
      for j from -e[0] to d[0]+e[0] do
         theta := theta+factor(coeff(wP,wE,j)*ff[j]
                        /((2*k+2*j+lambda-1)*(2*k+2*j+lambda)*(2*k+2*j+lambda+1))*rhs_coeff(k+j))
      od
   else
# - moved to 'recrel': alpha := -1/2; beta := -1/2; lambda := 0;
      wLc := collect(wL,wE); fact := 1;
      for j from u to u+r do
         lam[j] := factor(lam[j]*(2*k-1)*(2*k+1)/((2*k+2*j-1)*(2*k+2*j+1)));
         fact := lcm(fact,denom(lam[j]))
      od;
      theta := 0;
      for j from -e[0] to d[0]+e[0] do
         theta := theta+factor(coeff(wP,wE,j)*rhs_coeff(k+j)/((2*k+2*j-1)*(2*k+2*j+1)))
      od;
      theta:=theta*(2*k-1)*(2*k+1)
   fi;
   for j from u to u+r do
      lam[j] := factor(simplify(fact*lam[j]));
# - moved to 'symrel': print(`lam[`.j.`]` = lam[j]);
   od;
   theta := factor(fact*theta);
# - moved to 'symrel': print('theta' = theta);
# - added output (used in 'symrel')
   array(-2..1, [ u, r,
```

```
          array( u..u+r, [ seq(lam[j],j=u..u+r) ] ),
          theta ] )
##
end: # scalar_form

# Main program
recrel := proc (case, rhs_coeff, var, operator_coefficients_sequence)
# - some locals and globals added (see 5+ and 39+ lines below)
   local i, expr, ok, ok2;
   global N, wL, wP, alpha, beta, lambda,
          rD, rU, rJ, rV, rK, wE, k;
##
# - symbol 'k' is restricted, '?/?' constants cause error in 'gcd',
# - checking that; also testing the type of input arguments...
   k:='k'; wE:='wE';  # just in case k<>'k' or wE<>'wE'
   ok:=0; ok2:=0;  # 0 = all ok
   for i from 4 to nargs do
      expr:=collect(op(i,[args]),k);
      if lcoeff(collect(expr-coeff(expr,k,0),k),k)<>0 then ok:=i fi;
      if not type(expr,polynom) then ok2:=i fi
   od;
   if not member(case,{0,1,2}) or nargs<4 or
        not type(var,string) then
      print();
      print(`Invalid argument(s)!`)
   elif ok<>0 and var<>k then
      print();
      print(`Differential equation coefficients may not depend on 'k'!`)
   elif ok2<>0 then
      print();
      print(`All differential equation coefficients have to be polynomials!`);
      print(`Rational constants should be enclosed in backquotes!`)
   else
##
# - more careful treating of different cases...
      if case = 0 then
         alpha := 'alpha'; lambda := 'lambda'; beta := lambda-alpha-1
      elif case = 1 then
         alpha := -1/2; lambda := 0; beta := -1/2
      else
         if not type(beta,numeric) then beta := 'beta' fi;
         lambda := alpha+beta+1
      fi;
##
      N := nargs-4; rP.N := oper_create(op(3..nargs,[args]));
# - changing variable in rD, rU, rJ, rV[.], rK[.]
# - in order to allow 'x' constant on input
      oper_var(rP.N,rD);
      oper_var(rP.N,rU);
      oper_var(rP.N,rJ);
      oper_var(rP.N,rV[1]);
      oper_var(rP.N,rV[-1]);
      oper_var(rP.N,rK[1]);
      oper_var(rP.N,rK[-1]);
##
      canonical_form();
      wP := P_operator();
      wL := L_operator();
# - Reindexing result to obtain index-symetric relation
      symrel(scalar_form(case, rhs_coeff))
##
# - see corresponding 'if' above
   fi;
##
   NULL
end: # recrel
```

```
# - New procedure 'symrel':
# - Tries to make the result index-symetric (also prints the result)
# - input:
# -    result returned by 'scalar_form'
# - output (stored in the global variable 'recurrence')
# -    is the "almost" index-symetric relation;
# -    the same array structure as returned by 'scalar_form'
symrel := proc(res)
   local new_u, u, r, du, j, temp;
   global recurrence;
   u := res[-2];
   r := res[-1];
   new_u := -iquo(r,2);
   du := new_u-u;
   temp:=subs( k=k+du,
               array(-2..1, [ new_u, r,
                     array( new_u..new_u+r, [ seq(res[0][j],j=u..u+r) ] ),
                     res[1] ] ) );
   temp[0] := map(factor,temp[0]);
   if recurrence<>'noprint' then
      print();
      for j from new_u to new_u+r do
         print(`lam[`.j.`]` = temp[0][j])
      od;
      print('theta' = temp[1])
   fi;
   recurrence:=eval(temp);
   NULL
end: # symrel
##
```

Acknowledgement

This work was supported by Komitet Badań Naukowych (Poland) under grant 2 1029 91 1.

The authors are grateful to Paweł Keller who has introduced many corrections and improvents to the Maple program.

References

[1] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, S. M. Watt, *Maple V Language Reference Manual*, Springer-Verlag, New York, (1991).

[2] A. Erdélyi *et al.*, *Higher Transcendental Functions*, McGraw-Hill, New York, (1953).

[3] S. Lewanowicz, A new approach to the problem of constructing recurrence relations for the Jacobi coefficients, *Zastos. Mat.*, **21**, 303–326, (1991).

[4] S. Lewanowicz, Quick construction of recurrence relations for the Jacobi coefficients, *J. Comput. Appl. Math.*, **43**, 355–372, (1992).

[5] Y. L. Luke, *The Special Functions and their Approximations*, Academic Press, New York, (1969).

[6] S. Paszkowski, *Numerical Applications of Chebyshev Polynomials and Series*, PWN, Warsaw, (1975) (in Polish).

[7] J. Wimp, *Computation with Recurrence Relations*, Pitman, Boston, (1984).