Some Software for Symmetric Functions

and several applications

by

Dick van Leijenhorst

Abstract

Some Maple software is presented that was used in a study of the symmetric functions: in particular, MacMahon's method of counting Latin squares, and iterative behavior. Some background of various algorithms has been provided as well.

Keywords: symmetric functions, combinatorics, Latin squares, complexity.

1 Introduction.

1.1 The symmetric functions: an approach by computer algebra.

In this report I would like to record and discuss some Maple software developed and used in the course of an investigation in the theory of the symmetric functions. Some background will be provided as well.

The fascinating symmetric functions are useful in many places: classic effective algebraic number theory [22], transcendental number theory [3] and the construction and investigations of representations of the symmetric and other groups [25].

Also by themselves of course, the symmetric functions form a very interesting subject in algebra and discrete mathematics. For instance, MacMahon used them to count Latin squares, applying the theory of Hammond operators that was also studied later on by the prominent Dutch mathematician Van Der Corput ([7], [8].)

The theory of symmetric functions over the rational numbers forms an ideal playground for the computer algebraist, due to its beauty in combination with the long and complex polynomials involved.

In this respect, it is remarkable how many beautiful combinatorial and algebraic results have been obtained in the 19th century and the first half of the 20th by Hammond, Cayley, MacMahon, Weber and others [26], notwithstanding these intricacies and without any help of calculatory devices!

Modern computer algebra systems such as Maple turn out to be an excellent tool to investigate the theory. Moreover, combinatorial theory has advanced much, allowing one to state old results in a clearer way. Like many other authors, we shall mainly refer to MacDonald's celebrated book [25], where the theory has been put on a more secure footing.

During the rise of computer science and, more in particular, complexity theory, new questions were addressed to, such as deriving bounds for the time necessary to compute symmetric functions [34], and the construction of small formulas and circuits for the Boolean symmetric functions, e.g, the bits of the Hamming weight ([31], [32]) which are essential in the theory of error-correcting codes [27].

In this survey we shall start with our own Maple implementations of the basic algorithms involving the symmetric functions and their various bases. Our notations will closely follow the first chapters of Macdonalds textbook [25]. In this part there is some overlap with John R. Stembridge's library package SF (which is also based on MacDonald's book). We have used Maple V, vs. 3 and 4.

However, some new implementations belonging to more advanced topics will be addressed too (e.g, Weber's algorithm and an algorithmic form of the "Decomposition Lemma" [20]). In [20], we give a complete, rigourous and elementary proof of MacMahon's celebrated formula for the number of Latin squares of given order along the lines of "Scriptum 3" of Van Der Corput [7]. This treatise contains some of the software used for tests and verifications of MacMahon's theory of counting, Hammond's differential operators, Cayley reciprocity (MacMahon [26], Van Der Corput [7]).

Finally, we give some software used in the study of the iterative behaviour of symmetric functions [19]. A plaintext version of the software in this report can be downloaded as http:/www.cs.kun.nl/ bolke/Research/SFPACK.gz

Contents.
This report is divided into six sections:

> **1.** Introduction. Technical preliminaries: the symmetric functions; power series in countably many variables; subrings; lists; notations; other representations; unicity using sufficiently many variables. For more background we refer to [20].
>
> **2.** The symmetric function theorem and some generalities. Various ways of converting symmetric functions; e.g, implementation of Weber's algorithm and a discussion of its merits.

3. My own conversion package. Generators of and conversion procedures between the standard forms of the symmetric functions.

4. Hammond operators, the algebra of differential operators, and Latin squares.

5. Software for the k-fold symmetric functions (cf. [19]).

6. References.

1.2 Technical preliminaries.

The objects we shall study are power series in a countably infinite set $X = \{x_1, x_2, \cdots\}$ of variables. The underlying domain will be an infinite field K, occasionally a commutative ring R. Let F_∞ be the set of these series. Note that the set of terms $x_{i_1}^{j_1} x_{i_2}^{j_2} \cdots x_{i_m}^{j_m}$ is itself countable, so that the set F_∞ is in fact no more intimidating than the collection of power sets in one variable only!

The *symmetric* functions over X are "ideal" objects to be defined below. They form a set $S_\infty \subset F_\infty$. In practice of course, one works with the set S_n of symmetric functions over $X_n = \{x_1, x_2, \cdots x_n\}$ which are specializations defined by putting $x_i = 0,\ i > n$ in the obvious way. In a Platonic way, the symmetric function over X_n will be called a "shadow" (or, in this case, an "n-shadow"). In a notation similar to above, $S_n \subset F_n$.

Following [25], a *partition* P of *length* $k \geq 0$ is a nondecreasing sequence $P = (p_1, \cdots, p_k)$ of nonnegative integers, i.e. $0 \leq p_1 \leq p_2 \cdots \leq p_k$. In another notation, $P = (a_1^{i_1} \cdots a_m^{i_m})$ where $0 \leq a_1 < a_2 \cdots < a_m$ and each $i_j > 0$. The a_j are the different elements among the p_i, a_j occurring with multiplicity i_j. Hence, $\sum_{j=1}^{m} i_j = k$.

Given P, let T_P be the set of all terms of the form $t = x_{i_1}^{p_1} \cdots x_{i_k}^{p_k}$, with the i_j's from $\mathbb{N}$, all different. With a partition P we shall associate the following element of $\subset F_\infty$:

$$[P] = \sum_{t \in T_P} t.$$

$[P]$ is called a *partition function* or also a *listfunction* or, in short, a *list*.

The set S_∞ is defined as the linear subspace of F_∞ generated by all the $[P]$. The set S_n is obtained by the *projection* Π_n defined as the substitution $x_i = 0,\ i > n$. We let $S_0 =_{def} K$. The following facts are easy and well-known ([25]):

1. S_n is the set of symmetric functions in the variables $x_1, \cdots, x_n$.
2. $[P]$ maps to 0 by Π_n if $k > n$; however if $n \geq k$ the restriction of Π_n to S_k is injective.

Further definitions will be made below at the appropriate places. For more background we refer to [19], [20].

2 Implementing the Symmetric Function Theorem.

2.1 Introduction.

In this section we shall describe some Maple software implementing conversions between various representations of the symmetric functions. Though similar programs often are already available, e.g. in the Maple package SF, we shall discuss a few more unusual methods (e.g, Weber's algorithm and an algorithmic form of the "Decomposition Lemma" [20]). The references are those of the former section.

2.2 The Symmetric Function Theorem.

The Symmetric Function Theorem states that over any integral domain, any symmetric function in n variables can be written in a unique way as a polynomial in the elementary symmetric functions. Here by "function" we mean, in an old-fashioned way, "polynomial"; note that over an infinite field any polynomial function belongs to a unique polynomial, but this is not true over finite fields.

The Symmetric Function Theorem is one of the first results in invariant theory, which, indeed, can be seen as a generalization thereof [35]. The usual way to prove this is reminiscent of more modern reduction methods from Groebner basis theory in computer algebra.

In the old-fashioned, effective development of algebraic number theory [22] the Symmetric Function Theorem is indispensable.

In most computer algebra systems implementations of the symmetric function theorem are given and efficiency in calculating these is very important.

Some new and perhaps interesting variations on the Symmetric Function Theorem can be found in [14] and [19].

Most important is the implementation used for the symmetric function theorem (SFT), i.e. to transform symmetric functions into "ordinary" functions of the elementary symmetric functions. Indeed, the transformations between the other kinds of symmetric function bases are computationally trivial if one is given, but there always remains one "hard" transformation. So how to implement the SFT?

At least five approaches to this end are possible:

0. Consider the identity $f(x_1, \cdots, x_n) = g(a_1, \cdots, a_n)$ as a set of linear equations over the space generated by all the terms $x_1^{i_1} \cdots x_n^{i_n}$, with the coefficients of g as unknowns; and solve that equation.
1. The well-known classic reduction method as found in, e.g, of der Waerden;
2. Reduction using Groebner basis theory; in fact fitting in classic invariant theory and as such presented in Sturmfels's book [35] on

this topic;
3. Weber's method [27]; and
4. Using multidimensional interpolation.

A straightforward implementation of **1.** is given in our package as the Maple procedure xTOec. The slow algorithms **0.** and **2.** are mainly of theoretical interest and not implemented. We shall discuss methods **3.** and **4.** now.

2.3 Weber's Algorithm.

2.3.1 Introduction.

As to point **3.**, let us present in modern exact form, the old and not widely known idea of Weber. Maple code will be given which in many cases appears to be faster than the usual algorithms, e.g., as implemented by A.H.M. Levelt [21]. Some experimental results are also given.

2.3.2 Background.

Let f be a symmetric polynomial in $F_n = K[x_1, \cdots, x_n]$. According to the Symmetric Function Theorem there exists a unique polynomial f_s in $K[y_1, \cdots, y_n]$ such that $f_s(\alpha_1, \cdots, \alpha_n) = f$, where as before α_i is the i^{th} symmetric function in x_1 to x_n : $\alpha_i = \sum_{1 \le j_1 < j_2 \cdots < j_i \le n} x_{j_1} x_{j_2} \cdots x_{j_i}$. Here follows the algorithm and a proof of its correctness:

* If $n < 2$ then $f_s = f(y_1)$. Else,

* If $n \ge 2$ write $f = \sum f_i x_n^i$ where the f_i in F_{n-1} are symmetric in $x_1, \cdots, x_{n-1}$. Let $\alpha'_1, \cdots, \alpha'_{n-1}$ be the elementary symmetric functions in $x_1, \cdots, x_{n-1}$. Now *recursively apply Weber's algorithm and find polynomials* $f_{si}(y_1, \cdots, y_{n-1})$ *with* $f_{si}(\alpha'_1, \cdots, \alpha'_{n-1}) = f_i, for all i.$

* There exist polynomials $a'_{s\ i}(y_1, \cdots, y_{n-1}, x)$ in $K[y_1, \cdots, y_{n-1}, x]$ that satisfy $a'_{s\ i}(\alpha'_1, \cdots, \alpha'_{n-1}, x_n) = \alpha'_i$.

(Indeed, $\alpha'_i = \alpha_i - x_n\alpha_{i-1} + x_n^2\alpha_{i-2} - \cdots + (-1)^{i-1}x_n^{i-1}\alpha_1 + (-1)^i x_n^i$; $1 \le i \le n-1$. This follows from the identity $1 + \sum_{i=1}^{n-1} \alpha'_i x^i = \prod_{i=1}^{n-1}(1 + x_i x) = (1+x_n x)^{-1} \prod_{i=1}^{n}(1 + x_i x) = (1+x_n x)^{-1}(1 + \sum_{i=1}^{n} \alpha_i x^i)$ by expanding the right hand side as a power series in x.)

Now let $P(y_1, \cdots, y_{n-1}, x) = \sum f_{si}(a'_{s\ 1}, \cdots, a'_{s\ n-1})x_i$; note that by construction $P(\alpha_1, \cdots, \alpha_{n-1}, x_n) = f(x_1, \cdots, x_n)$. *Rewrite* P *in the form* $\sum e_i(y_1, \cdots, y_{n-1})x_i$.

* Let $Q(y_1, \cdots, y_n, x) =_{def} x^n + \sum_{i=0}^{n-1}(-1)_{n-i} y_{n-i} x^i$. Here y_n is a new indeterminate. Note that $Q(\alpha_1, \cdots, \alpha_n, x) = \prod_{i=1}^{n}(x - x_i)$. *Now consider* P *and* Q *as polynomials in* x *and divide* P *by* Q. This

can be done in $K[y_1, \cdots, y_n][x]$ since Q has highest coefficient 1. One obtains a relation $P(y_1, \cdots, y_{n-1}, x) = A(y_1, \cdots, y_n, x)Q(y_1, \cdots, y_n, x)+ R(y_1, \cdots, y_n, x)$. The remainder R has degree $< n$ in x.

* *Claim: x does not occur in R and $R = f_s$; the sought-for representation of f.*

Proof: Consider $R(y_1, \cdots, y_n, x) - f_s(y_1, \cdots, y_n)$; this equals $P(y_1, \cdots, y_{n-1}, x)$ $-A(y_1, \cdots, y_n, x)\, Q(y_1, \cdots, y_n, x) - f_s(y_1, \cdots, y_n)$. Substitution of $y_i = \alpha_i$ yields: $R(\alpha_1, \cdots, \alpha_n, x) - f_s(\alpha_1, \cdots, \alpha_n) = P(\alpha_1, \cdots, \alpha_{n-1}, x) - A(\alpha_1, \cdots, \alpha_n, x)Q(\alpha_1, \cdots, \alpha_n, x)$ $-f_s(\alpha_1, \cdots, \alpha_n)$.

Then $R(\alpha_1, \cdots, \alpha_n, x_n) - f_s(\alpha_1, \cdots, \alpha_n) = P(\alpha_1, \cdots, \alpha_{n-1}, x_n) - A(\alpha_1, \cdots, \alpha_n, x_n)$ $Q(\alpha_1, \cdots, \alpha_n, x_n) - f_s(\alpha_1, \cdots, \alpha_n) = f(x_1, \cdots, x_n) - A(\alpha_1, \cdots, \alpha_n, x_n) \prod_{i=1}^{n} (x_n - x_i)$ $-f(x_1, \cdots, \underline{x_n}) = f - A(\alpha_1, \cdots, \alpha_n, x_n).0 - f = 0.$

After the application of any coordinate permutation that changes x_n into x_j this becomes $R(\alpha_1, \cdots, \alpha_n, x_j) - f_s(\alpha_1, \cdots, \alpha_n) = f(x_1, \cdots, x_n) - A(\alpha_1, \cdots, \alpha_n, x_j)$ $Q(\alpha_1, \cdots, \alpha_n, x_j) - f(x_1, \cdots, x_n) = 0$. Let K_0 be the field $K(x_1, \cdots, x_n)$. We see that $R(\alpha_1, \cdots, \alpha_n, x_j) - f_s(\alpha_1, \cdots, \alpha_n)$, as an element of $K_0[x]$, has degree $< n$ in x whilst it possesses n zeroes in K_0, namely $x_1, \cdots, x_n$.

It follows that $R(\alpha_1, \cdots, \alpha_n, x_j) = f_s(\alpha_1, \cdots, \alpha_n) = f$.

Consider $R(y_1, \cdots, y_n, x) - f_s(y_1, \cdots, y_n)$ as a polynomial in x over $K[y_1, \cdots, y_n]$. The substitution $y_i \to \alpha_i$ (all i) makes the coefficients 0; however, the α_i's are algebraically independent since we always assume that K is infinite. Hence, the coefficients also disappear in $K[y_1, \cdots, y_n]$. So $R(y_1, \cdots, y_n, x) = f_s(y_1, \cdots, y_n)$ indeed. □

An elementary analysis readily shows that, theoretically, this is not a fast algorithm: the costs are of order $\Omega(n^n)$. However, in practice the usual algorithm often are even slower.

2.3.3 A historical note and some experiments.

Prof. Levelt (CSI/KUN) was so kind as to provide us with some earlier experiments by his hand (sent to the Maple User's Group in 1991), where the "classic" reduction procedure in the proof of the symmetric function theorem is compared with a Groebner basis variant.

Quoting him:

" ********Old fashioned algebra sometimes better than Groebner bases.
Long ago, before the advent of computer algebra, I was taught the Main Theorem on Symmetric Polynomials. The proof was presented as an example of a beautiful algorithm. Yes, some algebraist were interested in algorithms in those days. Look into of der Waarden's well-known book "Modern Algebra". It is interesting to see that the Main Theorem fits into Groebner basis theory. But for an efficient algorithm it is better to do it the classic way, which can be viewed as a normal

form algorithm 'avant la lettre'. Here follows a program using only low level Maple stuff and an example in order to compare the direct method with the Groebner basis method. "

One of Levelts typical results was: "

```
f:=x1^15+x2^15+x3^15+x4^15:
g:=sym2elsym(f,x,s);

# Computation (MapleV) takes about 47 secs on a'Spark station (IPC).
# Computation using normalf algorithm of Grobner packages takes more
# than 300 secs.
```

"

2.3.4 Some observations.

Using the above implementation ("weber"), some experiments were done as to assess the speed relative to the existing software mentioned in section 2. First, Maple 5.01 (US) for the Apple Macintosh was used on an old Macintosh IIsi. Unless stated otherwise however, all the following calculations were done under Unix on a Sun (the CSI "zeus" or the KUN Dept. of Math. wn3 computers).

As a first test, Levelt's example of section 2 was chosen:

```
f:=x1^15+x2^15+x3^15+x4^15.
```

"weber" took only a few seconds; checking the result took 24 seconds instead of 36 on a Sparc Workstation (Levelt). Hence, even allowing for a 33"weber" performs much better in this case. A summary account of this and some more sessions follows next.

Next, a procedure "average" was written that reduces a number of random symmetric polynomials in both ways, and computes the average running times. As a first experiment, a random polynomial "rapo" was taken and the elementary symmetric functions were substituted into it, yielding a complex symmetric polynomial "raposym". This polynomial was reduced to "rapo" using "weber" and "sym2elsym". Here, the classic method appeared to be 20 to 30 times faster than "weber"! This led us to conclusion 1. below.

The experiments suggest the following tentative remarks: 1. "weber" is probably faster than the classic method (e.g., "sym2elsym") when the output result is very complex.

(This leads to an interesting complexity problem: is there a simple relation or tradeoff between the complexity (e.g. string length, number of terms) of the symmetric function $f(x_1, \cdots, x_n)$ and that of its pendant $g(y_1, \cdots, y_n)$ (where $g(\alpha_1, \cdots, \alpha_n) = f$)? A possible approach might be, to consider f and g as vectors indexed by the terms and to investigate the matrix of the linear mapping $f \to g$.)

2. The running time of "weber" may be more "uniform" than the classic method (e.g., "sym2elsym").

3. During the "weber" procedure, Maple ran into trouble while working with very large polynomials. In "weber", the table gamma is a local variable. Somewhere during the recursion Maple forgot some of its stored values, whence the corresponding table entries evaluated as their names. Debugging showed that only gamma[0] existed from a given point. Obviously this is a memory problem; the same program performed well under Unix on a Sun system.

2.3.5 Maple procedures connected with Weber's algorithm.

For clarity's sake, an expanded version of our implementation of Weber's algorithm is presented first. A shorter version that, however, is more difficult to decypher can be found next; in the package it appears as "xTOe", our default conversion procedure.

```
# WEBER'S ALGORITHM FOR THE SYMMETRIC FUNCTION THEOREM:
# Input for the following procedure is a symmetric polynomial f(x1, x2,...);
# output is this polynomial written as g(y1,y2...) where yi denotes the ith
# elementary symmetric function ai; i.e. y1 = x1+x2+ . . . etc.

weber :=
   proc(f)
   local X,n,dgr,result,gamma,psi,eta,aacc,omega;
       n:=nops(indets(f));
                          dgr := degree(f,x.n);
       gamma := table();
       eta := table();
       aacc := table();
       if n < 2 then
           result := expand(subs(x.1 = y.1,f))
       else
      omega:=X^n; for i to n do omega:=omega+y.i*(-1)^i*X^(n-i) od;
           for i to n-1 do
           aacc[i]:=(-X)^i;
           for j to i do
           aacc[i]:=aacc[i]+z.j*(-X)^(i-j)
           od
           od;
           collect(f,x.n);
           for i from 0 to dgr do
               gamma[i] := weber(coeff(f,x.n,i))
           od;
           for i from 0 to dgr do
               for j to n-1 do
                   gamma[i] := subs(y.j = aacc[j],gamma[i])
               od
           od;
           for i from 0 to dgr do
           for j to n-1 do  gamma[i] := subs(z.j = y.j,gamma[i]) od
           od;
           psi:=0;for i from 0 to dgr do
           psi:=psi+gamma[i]*X^i;psi:=collect(psi,X)
           od;
           result := expand(rem(psi,omega,X))
       fi;
       result
   end;
```

```
#############################################################################################

# Converts symmetric function, given in x-vorm, to e-polynomial
# cf. Weber's method. Often faster than the classic method ``xTOec"
#  (see elsewhere). May be called as xTOe(f,n) (by Maple convention).
xTOe := proc(f) local X,n,dgr,gamma;
n:=nops(indets(f)); if n < 2 then RETURN(expand(subs(x.1 = e.1,f))) fi;
dgr := degree(f,x.n);

gamma:=subs(seq(z.j = e.j,j=1..n-1),eval(subs(seq(e.j = (-X)^j+
sum('z.k*(-X)^(j-k)','k'=1..j), j=1..n-1), eval(array(0..dgr,
[seq(xTOe(coeff(f, x.n,i)), i=0..dgr)])))));

expand(rem(collect(sum('gamma[i]*X^i','i'=0..dgr),X),X^n+
sum('(-1)^(n-i)*e.(n-i)*X^i','i'=0..n-1),X))
end:

#############################################################################################

# Since we used it in our initial experiments, we shall quote the following
# software by A. Levelt. Our own package contains a similar  program ``xTOec".

# Implementation of the Main Theorem on Symmetric Polynomials: For any
# commutative ring R let f be a symmetric polynomial in x1,...,xn and
# coefficients in R there exist one and only one polynomial g in s1,...,sn
# with coefficients in R such that f(x1,...,xn)=g(elsym1,...,elsymn). Here
# elsymi is the i-th elementary symmetric polynomial in x1,...,xn.
# (elsym1=x1+...+xn, elsym2=x1*x2+x1*x3+..., ...,xn=x1*x2*...*xn).
#
# Author: A.H.M. Levelt, Mathematisch Instituut, Toernooiveld, 6525 ED Nijmegen,
# The Netherlands. E-mail: ahml@sci.kun.nl
# (Lots of people must have programmed this algorithm as an exercise).

# For any list x=[x1,...,xn] of indeterminates (resp. expressions) and
# any integer i elsym(i,x) returns the i-th elementary symmetric polynomial.
# It returns 0 if i<1 or i>n. Notice that divisions are nowhere needed.

elsym:=proc(i,x)
local j;
if x=[] then RETURN(0) fi;
if i<1 or i>nops(x) then RETURN(0) fi;
if i=1 then RETURN(sum('x[j]','j'=1..nops(x))) fi;
if nops(x)=1 then RETURN(x[1]) fi;
expand(x[1]*elsym(i-1,rest(x))+elsym(i,rest(x)))
end:

# Let x:=[x1,...,xn], s=[s1,...,sn] be lists of indeterminates.
# Let f be a symmetric polynomial in x1,...,xn. Then
# sym2elsym(f,[x1,...,xn],[s1,...sn]) returns a polynomial g in s1,...,sn
# such that subs({s1=elsym(1,x),...,elsym(n,x)},g)=f.

sym2elsym:=proc(f,x,s)
local g,n,i,h,e,t,el;
g:=expand(f);
n:=nops(x);
if g=0 then RETURN(0) fi;
if n=0 then RETURN(f) fi;
 for i to n do el[i]:=elsym(i,x) od;
h:=highest(g,x);
e:=h[2];
t:=h[1]*product('el[i]^(e[i]-e[i+1])','i'=1..n-1)*el[n]^e[n];
expand(h[1]*product('s[i]^(e[i]-e[i+1])','i'=1..n-1)*s[n]^e[n]
 +sym2elsym(g-t,x,s))
end:

# Let g be a polynomial in x1,...,xn. Then highest(g,x) returns a list
# [c,[d1,...,dn]], where c*x1^d1*...*xn^dn is the leading monomial of g
# in the pure lexicographic ordering of g   (x1>x2>...>xn).
```

```
highest:=proc(g,x)
local d,c,h,restx;
if f=0 then RETURN(0) fi;
if x=[] then RETURN(g) fi;
d:=degree(g,x[1]);
c:=coeff(g,x[1],d);
if nops(x)=1 then RETURN([c,[d]]) fi;
h:=highest(c,rest(x));
[h[1],[d,op(h[2])]]
end:

rest:=proc(lijst)
local cnt,i;
cnt:=nops(lijst);
if cnt<2 then []
else [op(i,lijst)$i=2..cnt] fi
end:

##############################################################################

# Comparing the running times of ``sym2elsym" and ``weber" on a number of randomly
# chosen symmetric functions.


# average
# 1. Name/inputs:  average(nrgetest,nrvars,cfbound,grbound)
# 2. Functionality:  subjects a number of pseudorandom symmetric functions to
#  ``sym2elsym" and ``weber" and yields the average CPU-times.
# 3. Input format:  The input parameters of ``average" have the following meaning:
#  nrgetest is the number of random symmetric functions to be tested;
#  nrvars is the number of variables one works with;
#  cfbound is a bound for the absolute values of the coefficients; and
#  grbound is the total degree of the first chosen random polynomial in
#  the elementary symmetric functions.
# 4. Output format:  Maple screen output
# 5. Aux. proc's:  elsym
# 6. Description:  the procedure ``average" uses Levelts procedure ``elsym" to generate
# the elementary symmetric functions; then it generates a ``random"
# polynomial with integral coefficients in the elementary symmetric
# functions; expands it into a symmetric polynomial, and subjects it to
# ``sym2elsym" and ``weber". Having done this a number of times, the
# CPU time for both is averaged and displayed as ``levtime" and
# ``webtime", respectively.
# 7. Special remarks: -
# 8. Example(s): -

average:=proc(nrgetest,nrvars,cfbound,grbound)
local rapo, levtime,webtime, timbeg,x,s,c1,c2,i,j,n,raposym,LT,WT;
x:=[]; for i to nrvars do x:=[op(x),x.i] od;
s:=[]; for i to nrvars do s:=[op(s),s.i] od;
for i to nrgetest do
     rapo:=randpoly(s,coeffs=rand(-cfbound..cfbound),degree=grbound);
    print(`rapo is:`);print(rapo);raposym:=rapo;

     for j to nrvars do raposym:=subs(s.j=elsym(j,x),raposym) od;
raposym:=expand(raposym);

     levtime:=0;webtime:=0; timbeg:=time();

     c1:=weber(raposym);
webtime:=time()-timbeg+webtime;
print(`c1 is:`);print(c1);
c1:=expand(c1-rapo);if not(c1=0) then ERROR fi;

  timbeg:=time();
     c2:=sym2elsym(raposym,x,s);
```

```
levtime:=time()-timbeg+levtime;
     print(‘c2 is:‘);print(c2);
c2:=expand(c2-rapo);if not(c2=0) then ERROR fi
 od;
 LT:=levtime/nrgetest; WT:=webtime/nrgetest;
 print(‘levtime= ‘); print( LT); print(‘webtime= ‘); print(WT)
 end:
################################################################################
```

2.4 The interpolation approach.

2.4.1 Introduction.

As to point **4.**, consider any symmetric function $f(x_1, \cdots, x_n)$, written as $g(a_1, \cdots, a_n)$ where the a_i are the elementary symmetric functions. Now $f(x_1, \cdots, x_n)$ is determined completely by its values on a discrete $(d_1+1) \times \cdots \times (d_n+1)$-block of points, where $d_i \geq deg(f, x_i)$, all i.

Given such a block B however, take the set $B' = \{(a_1(\underline{x}), \cdots, (a_n(\underline{x}) | \, \underline{x} \in B\}$. B' may not be a block, but we know the values $g(\underline{y})$ for all $\underline{y} \in B'$ (note that $g(\underline{y})$ is the value of f in $\underline{x}$, where $(a_1(\underline{x}), \cdots, (a_n(\underline{x}) = \underline{y})$, and these determine g. We conclude that g can be recovered by use of interpolation, which can be done fast using FFT-related techniques.

However, no actual implementation of this idea is known to the author. Some Maple experiments will be presented now.

2.4.2 Experiments.

In order to test the interpolation idea, some preliminary experiments were made. Remember that we wish to interpolate the function $g(\underline{y})$ representing f as a polynomial in the elementary symmetric functions, on the set $B' = \{(a_1(\underline{x}), \cdots, (a_n(\underline{x}) | \, \underline{x} \in B\}$. Here B is a "block", but B' probably not. Interpolation is of course still possible using a multidimensional version of Lagrange's formulas. A naïve approach (not using an intelligent way to generate these formulas as, e.g., done in [1] for the one dimensional case), leads of course to an algorithm which is totally impractical except in the simplest applications. For completeness sake, some Maple procedures are presented below.

A better way to interpolate seems the following. One desires the set B' to be a block, such that it is easy to interpolate. Then, the original set B will in general not be a block; but it can be found using the numerical power of Maple. Indeed; prescribe the values of the elementary symmetric functions. The the elements of B are the roots of $X^n - a_1(\underline{\beta})X^{n-1} + a_2(\underline{\beta})X^{n-2} + \cdots + (-1)^{n-1}a_1(\underline{\beta}) + (-1)^n = 0$ for all the points $\underline{\beta} \in B$ which can be recovered by well-known algorithms ("interp" in Maple.)

Let us stress that, for a more-or-less practical implementation, it is advisible to make the calculation of these (complex) roots a precomputation, the results of which are stored in an array BIGARRAY.m. One may do this up to a certain dimension n and up to a degree d (the degree of the symmetric function f in any variable). Consequently, an algorithm like this will only be useful if one wishes to calculate g for a massive amount of symmetric functions f of restricted degree and number of variables. Also, the precision of the roots plays a role, restricting the size of the coefficients of the function f.

Having found the roots (forming a set B as above) one can, for given f, compute the values in these roots and than interpolate g on B'. In the appendix, ..., this idea was tested on the set $B' = \{1, \cdots, d+1\}^n$ for d and n as above. The results however, are disappointing (92 seconds CPU time for $f = x_1^8+x_2^8+x_3^8+x_4^8$, to be compared with only one second for "weber".)

Several refinements may be possible: a modular approach wherein one performs an analogous algorithm modulo several primes p and then uses Chinese remaindering to find the actual polynomial g; or, taking for B' the Cartesian product of n sets $\{1 = \xi^0, \xi, \xi^2, \cdots, \xi^d\}$ with ξ a primitive d^{th} root of unity. Then, the interpolation on B' can be done by FFT techniques.

However, our computational results indicate that a sequential approach will probably not be viable. A great advantage of interpolation methods above the others is, that they can easily be parallelized. In this way, these algorithms may be of some future use.

2.4.3 Software.

```
##############################################################################
# 1. ``PRIMITIVE" PROCEDURES USING LAGRANGE INTERPOLATION
formdelta:=proc(X,getallenlijst)
# X is a variable, ``getallenlijst" is a list without repetitions.
# formdelta constructs the list ``delta" of Lagrange-interpolation terms
# the j-th of this is 1 if X is the j-th element of ``getallenlijst"
# and 0 on each i-th element of ``getallenlijst" with i unequal to j.
local N,i,j,delta;
N:=nops(getallenlijst); delta:=list[1..N];
for j to N
do delta[j]:=1;
   for i to N
   do if not (i=j)
      then delta[j]:=expand(delta[j]*(X-getallenlijst[i])/
      (getallenlijst[j]-getallenlijst[i]))
      fi
   od
od;
eval(delta)
end:

# Example:
#
# formdelta(x,[1,2,3]);
#
# table([
#
```

```
#
#     1 = 1/2 x^2 - 5/2 x + 3
#
#          2
#     2 = -x  + 4 x - 3
#
#                2
#     3 = 1/2 x  - 3/2 x + 1
#
#     ])
#
# >

multidelta:=proc(pointlist)
# ‘‘pointlist" is a list of vectors (which in their turn
# are lists of, say, m components each).
# There will be constructed a list ‘‘delta" of Lagrange-interpolationpolynomials
# ‘‘delta" where the nu-th is 1 on the nu-th element of pointlist and 0 on
# the others. The variables in delta are x.1 to x.m.
local N,delta,nu,i,j,V,m,k;
N:=nops(pointlist);delta:=list[1..N];m:=nops(pointlist[1]);
for j to m
do V[j]:={}
od;
for j to m
do for nu to N
   do V[j]:=V[j] union {pointlist[nu][j]}
   od
od;V[j]:=convert(V[j],list);
for nu to N
do delta[nu]:=1;
   for j to m
   do for i to nops(V[j])
      do if V[j][i]=pointlist[nu][j]
         then k[j]:=i
         fi
      od; delta[nu]:=expand(delta[nu]*formdelta(x.j,V[j])[k[j]])
   od
od;
eval(delta)
end:

# trace(multidelta);

# h:=multidelta({[1,2,3],[4,5,6],[1,2,2],[8,4,8],[3,3,3]});
# S:={[1,2,3],[4,5,6],[1,2,2],[8,4,8],[3,3,3]};
# for i to 5
# do for j to 5
#    do print(‘i= ‘,i,‘j= ‘,j,‘h[i][j-de punt]=
# ‘,subs(x1=S[j][1],x2=S[j][2],x3=S[j][3],h[i]))
#    od
# od;

a:=proc(k,m) local h;
# a(k,m) is the k-th symm. function in m variables xi
if k=0 then h:=1 elif k>m then h:=0
else h:=a(k-1,m-1)*x.m+a(k,m-1)
fi; expand(h)
end:

interptable:=proc(pointlist)
# input: list of vectors of length m; output: another list of vectors
# nl. all [a.1(P),...,a.m(P)] with P from the pointlist and the a.i's
# the elementary symmetric functions in x.1,...,x.m
local i;
[seq(subs([seq(x.k=pointlist[i][k],k=1..nops(pointlist[1]))],
[seq(a(k,nops(pointlist[1])),k=1..nops(pointlist[1]))]),
i=1..nops(pointlist))]
end:
```

```
# P:=[[1,2,3],[3,4,5]];
# interptable(P);

syminterpol:=proc(f,pointlist)
# f is een symmetric function in m variables.
# input is further a pointlist of m-vectors.
# ‘‘syminterpol" constructs a new pointlist nwpl consisting of all [a.1(P),...,a.m(P)]
# with P from the pointlist and the a.i’s the elementary symmetric functions
# in x.1,...,x.m. Further the f(P) are being computed. Finally, the
# f(P)’s are interpolated on nwpl which yields a function g with g(a.1,...,a.m)=f,
# that is, if P is large enough.
local nwpl,i,wrdnlist,lagrs;
nwpl:=interptable(pointlist);
lagrs:=multidelta(nwpl);
wrdnlist:=[seq(subs([seq(x.k=pointlist[i][k],k=1..nops(pointlist[1]))],f),
i=1..nops(pointlist))];
expand(sum(lagrs[i]*wrdnlist[i],i=1..nops(pointlist)))
end:

trace(syminterpol);

f:=x1^2+x2^2+x3^2;

for i from 0 to 2
do for j from 0 to 2
   do for k from 0 to 2
      do PS[i+3*j+9*k+1]:=[2^i,3^j,4^k]
      od
   od
od;

PS:=convert(PS,list);

syminterpol(f,PS);

#########################################################################

# 2. ROOT INTERPOLATION IN THE GENERAL CASE, NOW USING ‘‘INTERP"
# ON POINTS OF CART. PROD. [1..d+1]^n

# Construct a multidimensional table of dimension n where
# the (j1,j2,...,jm)-th place (with each ji>0 and <= d+1)
# is filled with the list of roots of the polynomial having as its
# cofficinten (-1)^(k-1)*jk, k=1..m. This is the praecomputatie.
#
# A nice feature is, that this procedure, in a sense, writes itself.
# This (well-known) trick is necessary to provide a multidimensional array
# of (a priori) undetermined dimension.
# A similar way, using,‘‘pseudo-local"
# variables, is on pg 85 ff of the Maple V programming guide [29].
# Remark: the procedure regards AA namely as a kind of global variable.
# see pg 83 ff Maple V programming guide [29].
rootlist:=proc(n,d)
# so: for symm. functions up to n variables of deg <= d in each variable
local m,s,ss,sss,il,i,statem,hulpol,AA;
Digits:=12;
for m to n
do s:=‘‘; for i to m
  do s:=cat(s,‘for j.‘,i,‘ to ‘,d+1,‘ do ‘)
  od;

  il:=‘j.1‘;
  for i from 2 to m
  do il:=cat(il,‘,j.‘,i)
  od;

  ss:=cat(‘ hulpol:=(-X)^‘,m,‘;for k to ‘,m,‘ do hulpol:=hulpol+(-X)^(k-1)*j.k
  od;AA[‘,il,‘]:=[fsolve(hulpol,X,complex)]‘);

  sss:=‘ ‘;
```

```
  for i to m
  do sss:=cat(sss,‘od ‘)
  od; sss:=cat(sss,‘;‘);

  statem:=cat(s,ss,sss);
  parse(statem,statement);
od;Digits:=10;
AA
end:

# We shall now assume that the praecomputation has been performed and that we,
# thus, possess a global array BIGLIST(0..k,0..k,0..k)
# filled with root lists as described in the procedure ‘‘rootlist".
# Let there be given numbers m and d with:
# m <= n and: for all i <= m one has d < k. Let there furtehr be given a
# symmetric polynomial f in the variables x1,...,xm of deg d in each xi.
# The procedure below computes an array with the function values of
# f in the m roots that occur in BIGLIST[j1,...,jm]; this for all j1,...,jm
# with 1 <= ji <= d+1 (i=1,...,m).
flist:=proc(f) local m,s,ss,sss,il,ill,i,statem,hulpol,hf,xxx,fAR,d,n;
m:=nops(indets(f)); d:=degree(f,x1);hf:=convert(f,string);

s:=‘‘; for i to m
  do s:=cat(s,‘for j.‘,i,‘ to ‘,d+1,‘ do ‘)
  od;

il:=‘j.1‘;for i from 2 to m
  do il:=cat(il,‘,j.‘,i)
  od;

xxx:=‘round(subs(‘;for i to m
  do xxx:=cat(xxx,‘x.‘,i,‘=BIGLIST[‘,il,‘][‘,i,‘],‘)
  od;xxx:=cat(xxx,hf,‘));‘);

ss:=cat(‘fAR[‘,il,‘]:=‘,xxx);

sss:=‘ ‘;for i to m
  do sss:=cat(sss,‘od ‘)
  od; sss:=cat(sss,‘;‘);

statem:=cat(s,ss,sss);
parse(statem,statement);
eval(fAR)
end:

# The below procedure interpolates the values of f in the
# rootlist fAR. The result is the polynomial g(y1,...,ym)
# that expresses f in the elementary symmetric functions.
# n is the dimension of the big rootlist-array.
interpsfst:=proc(f)
local mm,m,i,il,s,ss,sss,xxx,statem,pl,t;global Loud,Lnew,d;
mm:=nops(indets(f));d:=degree(f,x1);flist(f);Loud:=fAR;

for m from mm-1 by -1 to 1
do
  s:=‘‘; for i to m
  do s:=cat(s,‘for j.‘,i,‘ to ‘,d+1,‘ do ‘)
  od;

  il:=‘j.1‘;for i from 2 to m
  do il:=cat(il,‘,j.‘,i)
  od;

  pl:=‘1‘;for i from 2 to d+1
  do pl:=cat(pl,‘,‘,i)
  od;
```

```
  ss:=cat(‘Lnew[‘,il,‘]:=expand(interp([‘,pl,‘],[‘);

  xxx:=cat(‘Loud[‘,il,‘,1]‘);
  for i from 2 to d+1
  do xxx:=cat(xxx,‘,Loud[‘,il,‘,‘,i,‘]‘)
  od;

  sss:=‘ ‘;for i to m
  do sss:=cat(sss,‘od ‘)
  od;sss:=cat(sss,‘;‘);

  statem:=cat(s,ss,xxx,‘],y.‘,mm-m,‘)) ‘,sss);
  parse(statem,statement);
  Loud:=eval(Lnew);Lnew:=evaln(Lnew);
od;
interp([seq(i,i=1..d+1)],[seq(Loud[i],i=1..d+1)],y.mm)
end:

# EXAMPLE:
rootlist(4,2):
# 4 is the max nr of variables, 2 de max deg
save(AA,BIGLIST.m);
# (take care!: this should be done first; it is essential since
# ‘‘flist" will be applied later on!)


read(BIGLIST.m): BIGLIST:=":

# interpsfst(x1^2+x2^2+x3^2);
# interpsfst(x1^2+x2^2+x3^2);
# interpsfst(x1^2+x2^2+x3^2+x4^2);
interpsfst(x1^2+x2^2+x3^2+x4^2-3*(x1+x2+x3+x4)*(x1+x2+x3+x4));

# on WN4: (old proc) rootlist(6,8): save(AA,BIGLIST.m);

################################################################################

# 2-APPENDIX: ROOT INTERPOLATION ON POINTS OF CART. PROD. [1..d+1]^n
# WRITTEN IN THE FORM OF A PROCEDURE:

fastsfst:=proc(f) local rootlist, flist,interpsfst,BIGLIST,Loud,Lnew,AA;

read(BIGLIST.m): BIGLIST:=":
# We shall now assume that the praecomputation has been performed and that we,
# thus, possess a global array BIGLIST(0..k,0..k,0..k)
# filled with root lists as described in the procedure ‘‘rootlist".
# Let there be given numbers m and d with:
# m <= n and: for all i <= m one has d < k. Let there further be given a
# symmetric polynomial f in the variables x1,...,xm of deg d in each xi.
# The procedure below computes an array with the function values of
# f in the m roots that occur in BIGLIST[j1,...,jm]; this for all j1,...,jm
# with 1 <= ji <= d+1 (i=1,...,m).
flist:=proc(f) local m,s,ss,sss,il,ill,i,statem,hulpol,hf,xxx,d,n;
m:=nops(indets(f)); d:=degree(f,x1);hf:=convert(f,string); n:=nops(BIGLIST);

s:=‘‘; for i to m
  do s:=cat(s,‘for j.‘,i,‘ to ‘,d+1,‘ do ‘)
  od;

il:=‘j.1‘;for i from 2 to m
  do il:=cat(il,‘,j.‘,i)
  od;

xxx:=‘round(subs(‘;for i to m
  do xxx:=cat(xxx,‘x.‘,i,‘=BIGLIST[‘,il,‘][‘,i,‘],‘)
  od;xxx:=cat(xxx,hf,‘));‘);

ss:=cat(‘fAR[‘,il,‘]:=‘,xxx);

sss:=‘ ‘;for i to m
  do sss:=cat(sss,‘od ‘)
  od; sss:=cat(sss,‘;‘);
```

```
statem:=cat(s,ss,sss);
parse(statem,statement);
eval(fAR)
end:

# print(f);
flist(f);


# The below procedure interpolates the values of f in the
# rootlist fAR. The result is the polynomial g(y1,...,ym)
# that expresses f in the elementary symmetric functions.
# n is the dimension of the big rootlist-array.
interpsfst:=proc(f)
local mm,m,i,il,s,ss,sss,xxx,statem,pl,t,d;global Loud,Lnew;
mm:=nops(indets(f));d:=degree(f,x1);Loud:=fAR;

for m from mm-1 by -1 to 1
do
  s:=``; for i to m
  do s:=cat(s,`for j.`,i,` to `,d+1,` do `)
  od;

  il:=`j.1`;for i from 2 to m
  do il:=cat(il,`,j.`,i)
  od;

  pl:=`1`;for i from 2 to d+1
  do pl:=cat(pl,`,`,i)
  od;

  ss:=cat(`Lnew[`,il,`]:=expand(interp([`,pl,`],[`);

  xxx:=cat(`Loud[`,il,`,1]`);
  for i from 2 to d+1
  do xxx:=cat(xxx,`,Loud[`,il,`,`,i,`]`)
  od;

  sss:=` `;for i to m
  do sss:=cat(sss,`od `)
  od;sss:=cat(sss,`;`);

  statem:=cat(s,ss,xxx,`],y.`,mm-m,`)) `,sss);
  parse(statem,statement);
  Loud:=eval(Lnew);Lnew:=evaln(Lnew);
od;
interp([seq(i,i=1..d+1)],[seq(Loud[i],i=1..d+1)],y.mm)
end:
interpsfst(f)
end:
# trace(fastsfst);
# fastsfst(x1^2+x2^2+x3^2+x4^2-3*(x1+x2+x3+x4)*(x1+x2+x3+x4));

##############################################################################

# 3. THE FFT AND ITS INVERSE

# Root interpolation on the points [1..d+1]^n (see below) turns out to be
# very slow. We try to acellerate this by interpolation
# on [1,ksi,ksi^2,..,ksi^d]^n with ksi a primitive d+1-th
# root of unity. It is possible to gain a log factor using the
# FFT. The Maple version of the FFT works exclusively
# with floats; therefore first we write our own version of the
# (recursive) FFT and its inverse. The iterative version
# may be faster (depending on the Maple compiler)
# but takes more programming; we postpone this.


# ROOT INTERPOLATION IN GENERAL CASE; on roots of unity
# ROUNDING THE COEFFICIENTS OF A POLYNOMIAL OR A LIST OF POLYNOMIALS
```

```
# In the auxiliary procedure below Lf is either a polynomial or a list
# of polynomials. Multiple variables and/or complex coefficients are
# allowed. The coefficients of (the elements of) Lf are rounded off towards
# the nearest integer. The output is a polynomial if Lf
# was, and a list otherwise.

allround:=proc(Lf) local rf,hf,L,i,hLf,bo,j;
bo:=type(Lf,polynom); if bo then hLf:=[Lf] else hLf:=Lf fi;
for i to nops(hLf)
do hf:=[evalc(Re(expand(hLf[i]))),evalc(Im(expand(hLf[i])))];
   for j in [1,2]
   do rf:=0;
      while not(hf[j]=0)
      do
         if   nops(indets(hf[j]))>0
         then L:=grobner[leadmon](hf[j],convert(indets(hf[j]),list))
         else L:=[hf[j],1]
         fi;
         hf[j]:=hf[j]-L[1]*L[2]; rf:=rf+round(L[1])*L[2]
      od;hf[j]:=rf
   od;
  hLf[i]:=hf[1]+I*hf[2]
od; if bo then hLf:=hLf[1] fi;
hLf
end:

# trace(allround);
# allround((3.3+0.17*I)*x^2*y-12.1*a^2*x+b*y);
# allround([(3.3+0.17*I)*x^2*y-12.1*a^2*x+b*y,3.14*I*y-23.00001,22.2]);


# BUILDING A LARGE ROOT LIST
# Call roottable(mmax,imax).
# For d=2^i-1, 1<=i<=imax and 1<= m <= mmax consider all index sets
# [j.1,...,j.m] of length m with 0<=j.k<=d for all k.
#
# For die indexset compute the list L of the m zeroes of the
# poluynomial (-1)^m*ksi^jm +(-1)^(m-1)*ksi^(j.(m-1))*X
# +...-(-X)^(m-1)*ksi^(j.1) +(-X)^m.
# Here, ksid is a prim. d-th root of 1
# i.e: ksi^j.i is the i-de elementary symm. function in the roots.
# L is stored in a table BIGTABLE at the location [j.1,...,j.m,d]
# This is the praecomputatie.
#
# A nice feature is, that in a sense this procedure writes itself.
# (this is necessary to provide a multidimensional array
# of (a priori) unknown dimension; a known trick.
# A similar method, using ``pseudo-local"
# variables, can be found on pp 85 ff of the Maple programmers guide [29].
# Remark: the procedure sees AA namely as a kind of global variable.
# see pg 83 ff Maple programming guide [29].

roottable:=proc(mmax,imax)
# i.e: for symm. functions up to mmax variables of degree <= 2^imax-1
# in each variable
local d,m,s,ss,sss,il,i,k,statem,hulpol,AA,ksi;global WW;

Digits:=12;
for i from 1 to imax
do d:=2^i-1;WW:=evaln(WW);
   for k from 0 to d do WW[k]:=evalf(cos(2*Pi*k/(d+1))+I*sin(2*Pi*k/(d+1))) od;

   for m to mmax
   do s:=``;
      for k to m
      do s:=cat(s,`for j.`,k,` from 0 to `,d,` do `)
      od;
```

```
      il:=‘j.1‘;
      for k from 2 to m
      do il:=cat(il,‘,j.‘,k)
      od; il:=cat(il,‘,‘,d);

      ss:=cat(‘ hulpol:=X^‘,m,‘;for k to ‘,m,‘
      do hulpol:=hulpol+(-1)^k*X^(‘,m,‘-k)*WW[j.k]
      od;AA[‘,il,‘]:=[fsolve(hulpol,X,complex)]‘);

      sss:=‘ ‘;
      for k to m
      do sss:=cat(sss,‘od ‘)
      od; sss:=cat(sss,‘;‘);

      statem:=cat(s,ss,sss);
      parse(statem,statement)
   od
od;
AA
end:

# trace(roottable);
# roottable(4,2);
# 4 is the max nr of variables, 2^2-1=3 the max degree.
# save(AA,BIGTABLE .m);
#
read(BIGTABLE.m): BIGTABLE :=AA:


# VALUES OF THE SYMM. FUNCTION f ON (RELEVANT PART OF) THE ROOT LIST
# We now assume having made a precomputation, providing us with a global
# array BIGTABLE filled with root lists as described in the procedure ‘‘rootlist".
#
# Let there be given the symm. function f in m variables of deg. d with:
# m <= mmax and d+1 <= dmax.
# Increase d until d+1 is a power of 2; suppose this has been done.
# Compute the value of f in the list of zeroes in
# BIGTABLE[j.1,...,j.m,-1,d]; this for all j.k's with 0<=j.k<=d.
# This function value is stored in the table fAR[j.1,...,j.m].
# It is the value of g in y1=ksid^j1,...,ym=ksid^jm, ksid een
# prim. d-th root of 1.

flist:=proc(f) local m,M,s,ss,sss,il,ill,i,statem,hulpol,hf,xxx,n,d;global fAR;
Digits:=12;fAR:=evaln(fAR);
m:=nops(indets(f)); hf:=convert(f,string);
d:=2^floor(log[2](degree(f,x1) + 1.001))-1;if not (d = degree(f,x1))
then d := 2*d+1 fi;

s:=‘‘; for i to m
  do s:=cat(s,‘for j.‘,i,‘ from 0 to ‘,d,‘ do ‘)
  od;

il:=‘j.1‘;for i from 2 to m
  do il:=cat(il,‘,j.‘,i)
  od;

xxx:=‘subs(‘;for i to m
do xxx:=cat(xxx,‘x.‘,i,‘=BIGTABLE [‘,il,‘,‘,d,‘][‘,i,‘],‘)
od;xxx:=cat(xxx,hf,‘);‘);

ss:=cat(‘fAR[‘,il,‘]:=‘,xxx);

sss:=‘ ‘;for i to m
  do sss:=cat(sss,‘od ‘)
  od; sss:=cat(sss,‘;‘);

statem:=cat(s,ss,sss);
parse(statem,statement);
eval(fAR)
end:
```

```
# THE FAST FOURIER TRANSFORM AND ITS INVERSE.
# The following procedure snelft is an implementation of the FFT.
# Let there be given a list a=[a0,a1,...] of Maple objects (floats, polynomials,...).
# If necessary, a will be padded with zeoroes first to a list [a0,...,aD] with D+1
# a power of 2. The output of snelft always has length D+1.
#
# snelft(a,`fft`) computes [F(ksi^0),F(ksi^1),...,F(ksi^D)]
# where F(X)=a0+a1*X+...+aD*X^D with ksi a primitive D+1-th root of 1.
# (N.b. a0 is L[1], if L = the list [a0,...,aD]!!!)
#
# snelft(b,`inv`) computes [a0,...,aD] where, for all i,
# F(ksi^i)=bi, again with F(X) equal to a0+a1*X+...+aD*X^D.
# Like a, b will first be padded with zeroes to [b0,...,bD].
#
# Later on we shall use ``snelft" instead of the standard Maple procedure ``interp"

snelft:=proc(a,s) local f,f1,f2,f3,i;global fff;
# ``global fff" is important, since fff will not be passed otherwise
# within her own recursion!!!
Digits:=15;
# n.b. this is valid only locally inside snelft; it is not necessary
# to set it back to 10
fff := proc(a,s) local D,Dhalf, i, W, ah,  A, B, C,L,b, c;
    for i from 0 to nops(a)-1 do ah[i]:=a[i+1] od;

    D := 2^floor(log[2](nops(a) + .001))-1;
    if not (D=nops(a)-1) then D:= 2*D+1 fi;

    if D = 0 then RETURN([[ah[0]],1]) else Dhalf := -1+(D+1)/2 fi;

    for i from nops(a) to D+1 do ah[i]:=0 od;

    W[0] := 1;
    if s=`fft` then W[1] := cos(2*Pi/(D+1)) + I*sin(2*Pi/(D+1))
    else W[1] := cos(2*Pi/(D+1)) - I*sin(2*Pi/(D+1)) fi;
    for i to D do W[i] := evalf(W[1]^i) od;

    b := []; for i from 0 by 2 to D-1 do b := [op(b), ah[i]] od;
    c := []; for i        by 2 to D   do c := [op(c), ah[i]] od;
    B := fff(b,s)[1];
    C := fff(c,s)[1];
    for i from 0 to Dhalf do
        A[    i    ] := evalf(expand(B[i+1] + W[i]*C[i+1]));
        A[i+Dhalf+1] := evalf(expand(B[i+1] - W[i]*C[i+1]))
    od;
    L:=[];for i from 0 to D do L:=[op(L),A[i]] od;
[L,D+1]
end:
f:=fff(a,s);f1:=f[1];f2:=f[2];
if s=`fft` then RETURN(f1) else RETURN([seq(f1[i]/f2,i=1..nops(f1))]) fi
end:

# INTERPOLATION TO A POLYNOMIAL WITH THE FFT
# Small change in snelft: interpolates only and yields a polynomial in
# a given variable X.
snelftpol:=proc(a,X) local h,i,L;
L:=snelft(a,`inv`); h:=0; for i to nops(L) do
h:=h+L[i]*evaln(X)^(i-1) od;
h
end:

# The procedure ipsf below interpolates the values of f(x1,...,xm) in the
# rootlist (table) fAR. The result is the polynomial g(y1,...,ym)
# that expresses f in the elementary symmetric functions.
```

```
ipsf:=proc(f)
local m,mloc,i,il,s,ss,sss,xxx,statem,pl,t,D;global Loud,Lnew;
m:=nops(indets(f));
flist(f);
Loud:=fAR;

D:=2^floor(log[2](degree(f,x1) + 1.001))-1;
if not (D = degree(f,x1)) then D := 2*D+1 fi;

for mloc from m-1 by -1 to 1
do
  s:=``; for i to mloc
  do s:=cat(s,`for j.`,i,` from 0 to `,D,` do `)
  od;

  il:=`j.1`;for i from 2 to mloc
  do il:=cat(il,`,j.`,i)
  od;

# pl:=`0`;for i from 1 to D
# do pl:=cat(pl,`,`,i)
# od;

  pl:=seq(i,i=0..D);

  ss:=cat(`Lnew[`,il,`]:=expand(snelftpol([`);

  xxx:=cat(`Loud[`,il,`,0]`);
  for i from 1 to D
  do xxx:=cat(xxx,`,Loud[`,il,`,`,i,`]`)
  od;

  sss:=` `;for i to mloc
  do sss:=cat(sss,`od `)
  od;sss:=cat(sss,`;`);

  statem:=cat(s,ss,xxx,`],y.`,mloc+1,`)) `,sss);
  parse(statem,statement);
  Loud:=eval(Lnew);Lnew:=evaln(Lnew);
od;
allround(snelftpol([seq(Loud[i],i=0..D)],y.1))
end:

# EXAMPLE:
# trace(ipsf);

# trace(roottable);
# roottable(4,2);
# 4 is het max aantal variabelen, 2^2-1=3 de max graad
# save(AA,BIGTABLE .m);
#
read(BIGTABLE.m): BIGTABLE :=AA:


# ipsf(x1^2+x2^2+x3^2);

# ipsf(x1^2+x2^2+x3^2+x4^2);

ipsf(x1^2+x2^2-3*(x1+x2)*(x1+x2));

# ipsf(x1^2+x2^2+x3^2+x4^2-3*(x1+x2+x3+x4)*(x1+x2+x3+x4));

# ipsf(x1^3+x2^3+x3^3+x4^3);

# fAR:=map(round,fAR);

################################################################################
```

3 The bases of the symmetric functions.

3.1 Introduction.

The notations are those of MacDonald [25] but in linear text.

The space of symmetric functions in x.1,...,x.n possesses various bases. We have written a Maple package to find the representations of symmetric functions with respect to this bases, and to transform these into each other.

There is some overlap between this set and the Maple standard library package SF.

A symmetric function f may be written in the following different forms:

0. x-form: f as a polynomial f(x.1,...,x.n).

1. e-form: f as a polynomial g(e.1,...,e.n) where e.r is the r-th elementary symmetric function (e.r =Σx.i1*x.i2*...*x.ir; the sum taken over all unordered r-tuples (sets) i1,...,ir from 1,2,...,n. Let us also define e.0=1 and e.k=0 for k ¿ n.

2. f-form: f written as a linear combination of the "forgotten" symmetric functions f.lambda; zie [25], p. 15. These are obtained via a canonical involution "omega" (also implemented).

3. h-form: f written as a polynomial in the functions h.r where h.r=Σm.lambda; the sum taken over all multisets lambda = (lambda.1,...,lambda.k) that satisfy |lambda|=lambda.1+...+lambda.k = r.

4. m-form: f written as a linear combination of the partition functions of the vorm m.lambda=Σx.1^lambda.1*x.2*lambda.2*...*x.k^lambda.k; the sum taken over all combinatorially different permutations of the multiset ("partition", "list") lambda=(lambda.1,...,lambda.k).

5. p-form: f written as a polynomial in the power functions (Newton polynomials) p.r=Σx.i^r. This goes not uniquely if n=1: e.g, x.1^5=p.1^5=p.5. However, in the overlying ring of power series uniqueness does hold. ([25] p. 16, 2.12)

6. s-form: f written as a linear combination of the functions s.lambda.

These are defined as follows ([25], pp. 23 ff): consider a monomial mu.alpha= x.1^alpha.1*...*x.n^alpha.n. Form a.alpha=Σ sgn(w)*w(mu) where w runs through all permutations in S.n and w acts on mu werkt by index permutation of the exponents.

W.l.o.g. one may assume that the alpha.i's differ, lest the result be 0. Let them be in descending order. Let delta be the partition (n-1,n-2,...,1,0) and write alpha=(alpha.1,...,alpha.n) =lambda+delta (componentwise). It can easily be shown that mu.alpha/mu.delta is a symmetric function s.lambda.

f can then be written as a linear combination of all s.lambda's having lambda's of ¡= n parts.

Thus we have implemented:

```
LIST OF GENERATORS:

    e       f       h       m       p       s       x

Forms: e(r,n); f(lambda,n); h(r,n); m(lambda,n); p(r,n); s(lambda,n).
```

3.2 Maple procedures connected with generators.

```
# e(r,n) is the r-th elementary symmetric function in n variables.
e:=proc(r,n);
if r=0 then RETURN(1) fi; if (r>n) or (r<0) then RETURN(0) fi;
expand(e(r-1,n-1)*x.n+e(r,n-1))
end:

# elab(lambda,n) is the product of the e.lambda.i with the lambda.i from
# the partition lambda. A call is made to the procedure e.
elab:=(lambda,n)->expand(product('e(lambda[i],n)','i'=1..nops(lambda))):
################################################################################

# The ``forgotten" symmetric functions f
# A call is made to the procedure omega.
f:=(lambda,n)->omega(m(lambda,n),n):
################################################################################

# h.r is the r-th complete symmetric function. It is the sum of all
# m.lambda's of degree r. One has h(0,n)=1 if n >= 0.
# Remark: this can also be done usingof the recursion (2.6') of Macdonald,
# p.14. However, then one must apply the symmetric function theorem
# first, and the question remains whether this is as efficient.
# This procedure calls the procedure m.
h:=proc(r,n);
if r<0 then RETURN(0) fi;combinat[partition](r);
expand(sum('m("[i],n)','i'=1..nops(")))
end:

# h.lambda is the product of the h.lambda.i's with the lambda.i from lambda.
# This procedure calls the procedure h.
hlab:=(lambda,n)-> expand(product('h(lambda[i],n)','i'=1..nops(lambda))):
################################################################################

# The involutie ``omega". This acts by applying the symmetric
# function theorem first and next substitution of h.r for ``e.r".
# ``omega" calls the procedures h en xTOe.
omega:=(f,n)->expand(subs(seq(e.i=h(i,n),i=1..n),xTOe(f,n))):
################################################################################

# m(lambda,n) computes for a partition
# lambda=[lambda.1,...,lambda.r] of  positive
# integers the associated complete symmetric function. The
# procedure also works well if the lambda.i's are unordered (in [25] they
# are non-increasing). E.g, m([1,2,2,3],5); yields
#    3   2   2          3   2   2
#  x1  x2  x4  x3 + x1  x2  x4  x5 + ...(+ many more terms)
m:=proc(lambda,n);
```

```
if nops(lambda)>n then RETURN(0) fi;
combinat[permute]([op(lambda),seq(0,i=nops(lambda)+1..n)]);
sum('product('(x.i)^"[j][i]','i'=1..n)','j'=1..nops("))
end:

# m is twice as fast as the following, old m2:
# m2:= proc(lambda,n) local explist,aantalexp, r,varlist,hulp;
# if lambda=[] then RETURN(1) fi; r:= nops(lambda);if n<r then RETURN(0)
# fi;
# with(combinat): explist := permute(lambda); varlist := choose(n,r);
# sum('sum('product('cat(`x`,varlist[k][j])^explist[i][j]','j'=1..r)',
# 'i'=1..nops(explist))','k'=1..nops(varlist))
# end:
```

```
################################################################################
```

```
# p.r is the r-th powersum (Newton polynomial). For n>=0 p.0=n.
p:=(r,n) ->sum('x.i^r','i'=1..n):

# plab is the product of the p.lambda.i with the lambda.i from the partition
# lambda. The lambda.i's have to be >=1.
plab:=(lambda,n)->expand(product('p(lambda[i],n)','i'=1..nops(lambda))):
```

```
################################################################################
```

```
# Computes the s-function of the list lambda, for n variables.
# lambda must be in standard descending order. Slow.
# calls the procedure ``h".
s:=proc(lambda,n) local i;
if n<nops(lambda) then RETURN(0) fi;
[op(lambda),seq(0,i=nops(lambda)+1..n)];
expand(linalg[det](array(1..n,1..n,[seq([seq(h("[i]-i+j,n),j=1..n)], i=1..n)])))
end:
```

```
################################################################################
```

```
# computes the antisymmetric function a.(lambda+delta) cf. pg 23/4
# MacDonald. lambda is a (non-increasing) partition of lengte <=n.
alpd:=proc(lambda,n) local i;
[seq(n-i+lambda[i],i=1..nops(lambda)),seq(n-i,i=nops(lambda)+1..n)];
expand(linalg[det](array(1..n,1..n,[seq([seq((x.i)^"[j],j=1..n)], i=1..n)])))
end:
```

3.3 Conversion procedures.

In order to rewrite a symmetric function f(x.1,...,x.n) from the x-vorm (0.) to the various bases it somehow seems necessary to apply the symmetric function theorem in one form or another. This is the hard, time consuming work.

Next, whenever f is given w.r.t. one of the above bases, transformations to another basis usually takes less time, since there are only simple substitutions involved. Transformations between bases (1.) - (6.) may be represented as a directed graph with vertices the five representations and as directed edges the transformations.

Hence one may strive to implement just as many (cheap) edges, that in the resulting subgraph all vertices are connected between them.

Below are listed the comments belonging to the conversion procedures. The code itself can be found in the appropriate section of the appendix.

Implemented were the transformations between the representations in terms of the various bases, and also from the x-form to the e-form. The procedures have the form $sigma_1TOsigma_2$ and are given in pairs $< sigma_1TOsigma_2, sigma_2TOsigma_1 >$ in a natural lexicographically induced ordering.

Many of these have been implemented, as said above, by linking other transformation procedures. Some others are rather unconventional (e.g., the procedure "mTOp" that is based upon the "Decomposition Lemma" of [20]) - but let us refer to the comments above the procedures.

Remark: xTOe uses Weber's method; an alternative is given as xTOec = the "classic" method.

```
CONVERSION PROCEDURES.

LIST OF GENERATORS:

    e       f       h       m       p       s       x


TABLE OF CONVERSION PROCEDURES:

    e       f       h       m       p       s       x
-------------------------------------------------------------
e  #       eTOf    eTOh    eTOm    eTOp    eTOs    eTOx

f   fTOe   #       fTOh    fTOm    fTOp    fTOs    fTOx

h   hTOe    hTOf   #       hTOm    hTOp    hTOs    hTOx

m   mTOe    mTOf    mTOh   #       mTOp    mTOs    mTOx

p   pTOe    pTOf    pTOh    pTOm   #       pTOs    pTOx

s   sTOe    sTOf    sTOh    sTOm    sTOp   #       sTOx

x   xTOe    xTOf    xTOh    xTOm    xTOp    xTOs   #

-------------------------------------------------------------
```

3.3.1 Maple procedures connected with conversions.

```
# Converts general symmetric function f from e-form to f-form.
# Calls the procedures eTOx and xTOf.
eTOf:=(f,n)->xTOf(eTOx(f,n),n):
```

```
# Converts general symmetric function f from f-form to e-form.
# Calls the procedures fTOx and xTOe
fTOe:=(f,n)->xTOe(fTOx(f,n),n):
#########################################################################

# Converts general symmetric function f from e-form to h-form.
# this cf Macdonald [25], pg 14.
eTOh:=(f,n)->
expand(subs(seq( e.(n-j+1)=(-1)^(n-j)*( h.(n-j+1)+
sum('(-1)^i*e.i*h.(n-j+1-i)','i'=1..n-j) ),j=1..n),f)):

# Converts general symmetric function f from h-form to e-form.
# this cf Macdonald [25], pg 14.
hTOe:=(f,n)->expand(subs(seq( h.(n-j+1)=(-1)^(n-j)*( e.(n-j+1)+
sum('(-1)^i*h.i*e.(n-j+1-i)','i'=1..n-j) ),j=1..n),f)):
#########################################################################

# Converts general symmetric function f from e-form to m-form.
# This procedure calls the procedures eTOx and xTOm
eTOm:=(f,n)->xTOm(eTOx(f,n),n):

# Converts general symmetric function f from m-form to e-form.
# This procedure calls the procedures mTOx and xTOe.
mTOe:=(f,n)->xTOe(mTOx(f,n),n):
#########################################################################

# Converts general symmetric function f from e-form om to p-form. (over Q).
# CAVEAT. eTOp sometimes gives possibly  unexpected results! All is correct, but
# the result of, e.g. eTOp(pTOe(p6,5),5) is not p6,
# but another p-expression equal to it. Reason: there are
# relations betwixt the p.i's in finite dimension; one finds
# only an expression in p.i's with i <= n. The above computation
# does yield p6 op if one takes not 5, but 6 variables
# (so n=6).

eTOp:=proc(f,n);
array(1..n,[p1,seq((-p.i*(-1)^i-sum('p.j*e.(i-j)*(-1)^j','j'=1..i-1))/i,i=2..n)]);
expand(subs(seq(e.(n-i)="[n-i],i=0..n-1),f))
end:

# Converts general symmetric function f from p-form to e-form.
pTOe:=proc(f,n) local  p,i,k,hf,L;
if nops(indets(f))=0 then RETURN(f) fi;
k:=sort(map((L->parse(substring(L,2..-1))),convert(indets(f), list)))[-1];hf:=f;
L:=[e1,seq(-sum('e.j*p.(i-j)*(-1)^j','j'=1..(i-1))-i*e.i*(-1)^i,i=2..n),
seq(-sum('e.j*p.(i-j)*(-1)^j','j'=1..n),i=n+1..k)];
# expand(subs(seq(p.(k-i)=L[k-i],i=0..k-1),f)) sometimes gives memory problems! so:
for i from 0 to k-1 do hf:=expand(subs(p.(k-i)=L[k-i],hf)) od
end:

# Converts p.k, with k > n, to p.j's with j < n. In fact this yields the
# relations between the p.i's occurring in the case of finitely many variables.
# Integrated and shortened form of pTOp:=(k,n)->eTOp(pTOe(p.k,n),n):
ponlyTOp:=proc(k,n) local p,i,hf,L;
hf:=p.k; if k<=n then RETURN(hf) fi;
L:=[seq(-sum('e.j*p.(i-j)*(-1)^j','j'=1..n),i=n+1..k)];
```

```
for i from k by -1 to n+1 do hf:=expand(subs(p.i=L[i-n],hf)) od;
[p1,seq((-p.i*(-1)^i-sum('p.j*e.(i-j)*(-1)^j','j'=1..i-1))/i,i=2..n)];
expand(subs(seq(e.(n-i)="[n-i],i=0..n-1),hf))
end:

# Converts f, given in p-form, to p-vorm with all p.j's of index j < n.
# Integrated and shortened form of pTOp:=(k,n)->eTOp(pTOe(p.k,n),n):
pTOp:=proc(f,n) local p,k,i,hf,L; if nops(indets(f))=0 then RETURN(f) fi;
k:=sort(map((L->parse(substring(L,2..-1))),convert(indets(f),list)))[-1];
hf:=f; if k<=n then RETURN(hf) fi;
L:=[seq(-sum('e.j*p.(i-j)*(-1)^j','j'=1..n),i=n+1..k)];
for i from k by -1 to n+1 do hf:=expand(subs(p.i=L[i-n],hf)) od;
[p1,seq((-p.i*(-1)^i-sum('p.j*e.(i-j)*(-1)^j','j'=1..i-1))/i,i=2..n)];
expand(subs(seq(e.(n-i)="[n-i],i=0..n-1),hf))
end:

# Yields the explicit relation over the integers between
# p.k and the p.j's with j <= n. The relation is non-trivial if k> n.
# The result is of type equation. Call this E; the l.h.s. then
# can be found as op(1,E) and the r.h.s. asop(2,E).
prel:=proc(k,n) local f,g,c;f:=pTOp(p.k,n);c:=icontent(f);f:=f/c;
sum('x.i^k','i'=1..n)/c=subs(seq(s=sum('x.i^parse(substring(s,2..-1))' ,
'i'=1..n),s=indets(f)),f)
end:
##############################################################################

# Converts general symmetric function f from e-form to s-form.
# Calls the procedures eTOx and xTOs.
eTOs:=(f,n)->xTOs(eTOx(f,n),n):

# Converts general symmetric function f from s-form to e-form.
# Calls the procedures xTOe and sTOx.
sTOe:=(f,n)->xTOe(sTOx(f,n),n):
##############################################################################

# Converts general symmetric function f from e-form to x-form.
# this procedure calls the procedure e.
eTOx:= (f,n)->expand(subs(seq(e.r = e(r,n),r=1..n),f)):

# Converts symmetric function, given in x-vorm, to e-polynomial
# cf. Weber's method. Often faster than the
# classic method ``xTOec" (see elsewhere). May be called
# as xTOe(f,n) (by Maple convention).
xTOe := proc(f) local X,n,dgr,gamma;
n:=nops(indets(f)); if n < 2 then RETURN(expand(subs(x.1 = e.1,f))) fi;
dgr := degree(f,x.n);
gamma:=subs(seq(z.j = e.j,j=1..n-1),eval(subs(seq(e.j = (-X)^j+
sum('z.k*(-X)^(j-k)','k'=1..j), j=1..n-1), eval(array(0..dgr,
[seq(xTOe(coeff(f, x.n,i)), i=0..dgr)])))));
expand(rem(collect(sum('gamma[i]*X^i','i'=0..dgr),X),X^n+
sum('(-1)^(n-i)*e.(n-i)*X^i','i'=0..n-1),X))
end:

# Converts symmetric function, given in x-form, to e-polynomial
# by the classic method (cf vd. Waerden [36]).
# This procedure calls the procedure e.
xTOec := proc(f)
        local xlist,i,n,floc,T,cf,L,LL,trekaf,trekafsymb,A;
```

```
        A:=`A`;floc := expand(f);n:=nops(indets(floc));
            if n=0 then RETURN(floc) fi;
            xlist := [A];
            for i to n do  xlist := [op(xlist),x.i] od;
floc := sort(floc+A,xlist,plex);
T := op(2,floc);floc:=floc-A;
            cf := coeffs(T);
            T := sort(T/cf,xlist,plex);
            if nops(indets(T)) = 1 then L := [T,0]
            else L := [op(convert(T,list)),0]
            fi;
            LL := [];
            for i to nops(L)-1 do
                LL := [op(LL),ldegree(L[i])-ldegree(L[i+1])]
            od;
            trekaf := cf;
            trekafsymb := cf;
            for i to nops(LL) do
                trekaf := trekaf*e(i,n)^LL[i];
                trekafsymb := trekafsymb*e.i^LL[i]
            od;
            trekafsymb;
            floc := floc-trekaf;
            expand(trekafsymb+xTOec(floc))
        end:
##############################################################################

# Converts general symmetric function f from f-form to h-form.
# Calls the procedures fTOx and xTOh
fTOh:=(f,n)->xTOh(fTOx(f,n),n):

# Converts general symmetric function f from h-form to f-form.
# Calls the procedures hTOx and xTOf
hTOf:=(f,n)->xTOf(hTOx(f,n),n):
##############################################################################

# Converts general symmetric function f from f-form to p-form.
# Calls the procedures fTOx and xTOp
fTOp:=(f,n)->xTOp(fTOx(f,n),n):

# Converts general symmetric function f from p-form to f-form.
# Calls the procedures pTOx and xTOf
pTOf:=(f,n)->xTOf(pTOx(f,n),n):
##############################################################################

# Converts general symmetric function f from f-form to m-form.
# Calls the procedures fTOx and xTOm
fTOm:=(f,n)->xTOm(fTOx(f,n),n):

# Converts general symmetric function f from m-form to f-form.
# Calls the procedures mTOx and xTOf
mTOf:=(f,n)->xTOf(mTOx(f,n),n):
##############################################################################

# Converts general symmetric function f from f-form to s-form.
# Calls the procedures fTOx and xTOs
fTOs:=(f,n)->xTOs(fTOx(f,n),n):
```

```
# Converts general symmetric function f from s-form to f-form.
# Calls the procedures sTOx and xTOs
sTOf:=(f,n)->xTOs(sTOx(f,n),n):
```

```
##############################################################################
```

```
# Converts general symmetric function f from f-form to x-form.
# Typical input: 3*‘f[1, 2, 3]‘ + 7*‘f[1, 1, 2]‘ or output of xTOf.
fTOx:=(f,n)->omega(mTOx(subs(seq(v=cat(‘m‘,substring(v,2..length(v))),
v=indets(f)),f),n),n):

# Converts general symmetric function f from x-form to f-form.
# (i.e. expressed in the ‘‘forgotten" symmetric functions f.lambda.
# Slow. Typical output: 3 f[1, 2, 3] + 7 f[1, 1, 2] where
# the f[...] are names (i.e. represent variables).
# for f[] the constant 1 is substituted.
xTOf :=proc(g,n);
xTOm(omega(g,n),n);
subs(seq(v=cat(‘f‘,substring(v,2..length(v))),v=indets(")),")
end:
```

```
##############################################################################
```

```
# Converts general symmetric function f from h-form to m-form.
# this procedure calls the procedures hTOx and xTOm.
hTOm:=(f,n)->xTOm(hTOx(f,n),n):

# Converts general symmetric function f from m-form to h-form.
# this procedure calls the procedures mTOx and xTOh.
mTOh:=(f,n)->xTOh(mTOx(f,n),n):
```

```
##############################################################################
```

```
# Converts general symmetric function f from h-form to p-form cf. pg. 16 Macdonald:
hTOp:=(f,n)->expand(subs(seq( h.(n-j+1)=( p.(n-j+1)+sum('h.i*p.(n-j+1-i)',
'i'=1..n-j) )/(n-j+1),j=1..n),f)):

# Converts general symmetric function f from p-form to h-form cf. pg. 16 Macdonald:
pTOh:=(f,n)->expand(subs(seq( p.(n-j+1)=(n-j+1)*h.(n-j+1)-sum('p.i*h.(n-j+1-i)',
'i'=1..n-j) ,j=1..n),f)):
```

```
##############################################################################
```

```
# Converts general symmetric function f from h-form to s-form.
# Calls the procedures hTOx and xTOs.
hTOs:=(f,n)->xTOs(hTOx(f,n),n):

# Converts general symmetric function f from s-form to h-form.
# Calls the procedures xTOh and sTOx.
sTOh:=(f,n)->xTOh(sTOx(f,n),n):
```

```
##############################################################################
```

```
# Converts general symmetric function f from h-form to x-form.
# this procedure calls the procedures pTOx and hTOp.
hTOx:=(f,n)->pTOx(hTOp(f,n),n):

# Converts general symmetric function f from x-form to h-form.
# This procedure calls the procedures xTOe and eTOh.
xTOh:=(f,n)->eTOh(xTOe(f,n),n):
```

```
##############################################################################
```

```
# Converts general symmetric function f from m-form to p-form.  The presence
# of ‘‘n" is not necessary but allowed, in
# Maple. Faster than the obvious mTOp:=(f,n)->xTOp(mTOx(f,n),n):
# It is based upon the the ‘‘m-decomposition-lemma" from the theory of Hammond
# operators. As with eTOm, p.i’s with i>n
# can occur. In order to exclusively get p.i’s with i < = n one should apply
# pTOp(",n) on the result.
# Warning! this is necessary for pTOh to function correctly!
# (otherwise, e.g.:  7 m[2, 1, 1] + 3 m[3, 2, 1]
# > mTOp(%,4);
#                                                    2
#   7/2 p2 p1  - 7 p3 p1 + 7 p4 - 7/2 p2  + 3 p3 p1 p2 - 3 p1 p5
#                                                    2
#          - 3 p4 p2 + 6 p6 - 3 p3
#
# > pTOh(%,4);
#                                                  4
#   -39 h3 h1  - 75 h1  h2  + 51 h1  h2 - 27 h3  - 28 h2  + 6 p6
#                                                  2                  4
#          + 70 h1  h2 - 3 h1 p5 + 12 h1  h4 - 24 h4 h2 - 21 h1
#
#                                                3       6
#          + 28 h4 + 96 h1 h3 h2 + 12 h2  - 9 h1  - 49 h1 h3
#

mTOp:=proc(f) local i,c,V,hulpf,huidigevar,LUa,hulp,a,L,mL,t,Lta,mLta;
c:=L->combinat[numbperm](L)/nops(L)!;V:=indets(f); hulpf:=f;
while not (V={})
do huidigevar:=V[1]; V:=V minus {huidigevar};
   LUa:=parse(substring(huidigevar,2..length(huidigevar)));
   if nops(LUa)=1 then hulp:=cat(‘p‘,LUa[1]) else
      a:=LUa[-1];L:=subsop(-1=NULL,LUa);
      mL:=cat(‘m‘,convert(L,string));V:=V union {mL};
      hulp:= mL*p.a/c(L); for t to nops(L)
      do Lta:=
         subsop(t=L[t]+a,L);mLta:=cat(‘m‘,convert([seq(sort(Lta)[
         nops(Lta)-i+1],i=1..nops(Lta))],string));
         V:=V union {mLta};hulp:=hulp-mLta/c(Lta)
      od; hulp:=hulp*c(LUa);
   fi; hulpf:=subs(huidigevar=hulp, hulpf)
od; expand(hulpf)
# ;pTOp(",n)
end:

# Converts general symmetric function f from p-form to m-form.
# This procedure calls the procedures pTOx and xTOm.
pTOm:=(f,n)->xTOm(pTOx(f,n),n):
##############################################################################

# Converts general symmetric function f from m-form to s-form.
# Calls the procedures mTOx and xTOs.
mTOs:=(f,n)->xTOs(mTOx(f,n),n):

# Converts general symmetric function f from s-form to m-form.
# Calls the procedures xTOm and sTOx.
sTOm:=(f,n)->xTOm(sTOx(f,n),n):
##############################################################################
# Converts general symmetric function f from m-form to x-form.
# Typical input: 3*‘m[1, 2, 3]‘ + 7*‘m[1, 1, 2]‘ or output of xTOm.
mTOx:=(f,n)->expand(subs(seq(s=m(parse(substring(s,2..length(s))),n),
```

```
s=indets(f)),f)):

# Converts general symmetric function f from x-form to m-form.
# Slow. Typical output: 3 m[1, 2, 3] + 7 m[1, 1, 2] where
# the m[...] are names (i.e. representing variables.)
# for m[] the constante 1 is substituted.
# Regrettably, the weird construction with A appears to be
# necessary with one term functions, such as -8*x1
xTOm :=proc(f,n) local i,floc,T,cf,L,LL,A;
floc := expand(f);if nops(indets(floc)) = 0 then RETURN(floc) fi;
A:=evaln(A);T:=op(1,floc+A);if T=A then T:=op(2,floc+A) fi;cf :=coeffs(T);
T := T/cf; if nops(indets(T))=1 then L:=[T] else L:=convert(T,list) fi;
LL:=sort([seq(ldegree(L[i]),i=1..nops(L))]);
LL:=[seq(LL[nops(LL)-i+1],i=1..nops(LL))];
for i from nops(LL) by -1 to 1 while LL[i]=0 do LL:=subsop(i=NULL,LL) od;
floc := floc-cf*m(LL,n);
subs(`m[]`=1,cf*cat(`m`,convert(LL,string))+xTOm(floc,n))
end:

# alternative term extraction with grobner; somewhat slower.
# n:=nops(indets(f)); with(grobner,leadmon): s:=[f]: L:=[];
# for i from n by -1 to 0 do s:=leadmon(s[1],[x.i]);
# if not(s[2]=1) then L:=[degree(s[2]),op(L)] fi od;
##############################################################################

# Converts general symmetric function f from p-form to s-form.
# Calls the procedures pTOx and xTOs.
pTOs:=(f,n)->xTOs(pTOx(f,n),n):

# Converts general symmetric function f from s-form to p-form.
# Calls the procedures xTOp and sTOx.
sTOp:=(f,n)->xTOp(sTOx(f,n),n):
##############################################################################
# Converts general symmetric function f from p-form to x-form.
# This procedure calls the procedures pTOe and eTOx.
# Is faster tan pTOx:=(f,n)->eTOx(pTOe(f,n),n):
pTOx:=(f,n)->expand(subs(seq(s=sum('x.i^parse(substring(s,2..-1))' ,
'i'=1..n),s=indets(f)),f)):

# Converts general symmetric function f from x-form to p-form.
# This procedure calls the procedures eTOp and xTOe.
xTOp:=(f,n)->eTOp(xTOe(f,n),n):
##############################################################################

# Converts general symmetric function f from s-form to x-form.
# Typicalinput: 3*`s[5]` + 7*`s[1, 1, 1,1,1]` or output of xTOs.
sTOx:=(f,n)->expand(subs(seq(v=s(parse(substring(v,2..-1)),n), v=indets(f)),f)):

# xTOs strategy: cf pg 23/4 MacDonald [25].
# The given symm. function f is first multiplied by
# a.delta, to fas, say. Then fas is antisymmetric and can be factorized into the
# functions a.(lambda+delta). This can be done in the way of of xTOm! Afterwards
# the variable a.(lambda +delta) is replaced via substitution by
# s.lambda. a.(lambda+delta) is computed as a determinant (since there seems to be
# no signum-function in Maple, I think).
# For s[] the constante 1 is substituted.
xTOs :=proc(f,n) local floc,reduceer; if expand(f) = 0 then RETURN(0) fi;
```

```
with(linalg,det,vandermonde):
floc := expand(det(vandermonde([seq(x.i,i=1..n)]))*f*(-1)^(n*(n-1)/2));

reduceer:= proc(ff,p::procedure) local i,ffloc,fflocplus,fflocmin,T,cf,L,LL,A,n;
ffloc:=expand(ff);n:=nops(indets(ffloc)); if n =0 then RETURN(ffloc) fi;

T:=op(1,ffloc);if nops(indets(T))=0 then T:=op(2,ffloc) fi;cf :=coeffs(T); T := T/cf;
if nops(indets(T))=1 then L:=[T] else L:=convert(T,list) fi;

LL:=sort([seq(ldegree(L[i]),i=1..nops(L))]);
LL:=[seq(LL[nops(LL)-i+1]-n+i,i=1..nops(LL))];A:=cf*alpd(LL,n);
fflocplus:=expand(ffloc+A);fflocmin:=expand(ffloc-A);
if nops(fflocplus)>nops(fflocmin) then ffloc := fflocmin
else ffloc := fflocplus;cf:=-cf fi;
for i from nops(LL) by -1 to 1 while LL[i]=0 do LL:=subsop(i=NULL,LL) od;
subs(`s[]`=1,cf*cat(`s`,convert(LL,string))+p(ffloc,p))
end:
reduceer(floc,reduceer)
end:

##############################################################################
```

4 Maple procedures connected with Hammond's differential operators and MacMahon's counting method of Latin squares.

4.1 Introduction: Hammond operators.

The Hammond operator of a partition lambda is an element of a certain commutative algebra of differential operators acting on the symmetric functions. This algebra is isomorphic with that of the symmetric functions [7], [20].

Here one has to work with infinitely many variables. However, one may restrict oneself to finitely many (a homomorphic image), in which the associated Hammond operators become finite.

In the above operator-algebra are operators d.i; d.i(f) is differentiation of the symmetric function f, written in e-form, with respect to e.i. In the case of n variables we put d.i = 0 for i > n.

Let there be given a partition lambda (in descending order as in [25], and if necessary considered to possess an infinite tail of zeroes.

e.lambda is as usual the product of all e(lambda[i]) (Maple procedure: e.lambda(lambda,n), in x1,...,xn).

Furthermore let c.lambda =j1!j2!...jk! with ji the multiplicity of the i-the part (>0) of lambda.

Let us say that a list a list mu lies *above lambda,* if for all i mu[i]≥lambda[i]. Notation: mu ≥lambda (in MacDonald this is denoted by a inverse inclusion sign (pg. 4)). mu-lambda is defined componentwise ("skew diagram" in MacDonald).

The Hammond operator H(lambda) (or H(lambda,n) if we have n variables) now is defined as (1/c.lambda)*e(mu-lambda)d(mu), the sum over all lists mu $\geq$lambda. This sum is finite if we have n variables, since e.i is zero for i > n. H(lambda) has coefficients in $\mathbb{Z}$.

H(lambda,n) is zero if max lambda.i > n. When lambda = [] (empty) is, the operator is the identity.

For more details we refer to [20].

4.2 Maple code.

```
# 1. Some code useful to experiment with Hammond operators.

# Combinatorial version of a Hammond cancellation operator for functions in x-form.
# HAM is the Hammond operator that cancels a part p from the lists occurring in
# the symmetric function f, (given in x-form).
# HAM calls the procedures ``sch" (see below), xTOm and mTOx.
# a constant term C equals C.m([],n) so is made zero
# at the start (for p does not occur); in the result m[]
# of course equals 1
HAM:=proc(f,p,n);
xTOm(f-subs(seq(x.i=0,i=1..n),f),n);
mTOx(subs(`m0`=0,subs(seq(v=cat(`m`,convert(sch(parse(substring(v,
2..length(v))),p,n),string)),v=indets(")),")),n)
end:

#ALTERNATIVE VERSION: xTOm and mTOx are not being used here (as in HAM)
HAMm:=proc(f,p,n);
f-subs(seq(x=0,x=indets(f)),f);
subs(`m[]`=1,`m0`=0,subs(seq(v=cat(`m`,convert(sch(parse(substring(v,
2..length(v))),p,n),string)),v=indets(")),"))
end:

##############################################################################

# sch cancels, if possible, a part p from the partition lambda; otherwise it yields 0.
# If the number of variables n is smaller than nops(lambda) the result also is 0.
# Assumption: the list lambda is in descending order.
sch:=proc(lambda,p,n) local i,nl;
nl:=nops(lambda);if n<nl then RETURN(0) fi;
i:=1;while (i<=nl) and not (lambda[i]=p) do i:=i+1 od;
if i=nl+1 then RETURN(0) fi;subsop(i=NULL,lambda)
end:

# EXAMPLE: sch([7,7,7],7,3);

##############################################################################

# The Hammond operator for the partition lambda and n variables, applied to
# the symmetric function f in x-form. lambda may be unordered and contain zeroes.
HAMMOND:=proc(lambda,f,n) local i; global L,ham;
L:=evaln(L);L:=sort(lambda);L:=[seq(L[nops(L)-i+1],i=1..nops(L))];
for i from nops(L) by -1 to 1 while L[i]=0 do L:=subsop(i=NULL,L) od;
if L=[] then RETURN(expand(f)) fi;

ham:=evaln(ham);ham:=0;

parse(cat(seq(` for mu`.i.` from L[`.i.`] to `.n.` do`,i=1..nops(L)),
` ham:=subs(e0=1,expand(ham+product('e.(mu.i-L[i])','i'=1..`,nops(L),`)*`,
seq(`diff(`,i=1..nops(L)),convert(xTOe(f,n),string),seq(`,e.mu`.i.`)`,
```

```
i=1..nops(L)),`))`,seq(` od`,i=1..nops(L)),`;`),statement);
eTOx(ham*combinat[numbperm](L)/nops(L)!,n)
end:
#############################################################################

# the Hammond operator for the partition lambda and n variables, applied to
# the symmetric function f given in e-form. lambda may be unordered
# and contain zeroes.
HAMMONDe:=proc(lambda,f,n) local i; global L,ham;
L:=evaln(L);L:=sort(lambda);L:=[seq(L[nops(L)-i+1],i=1..nops(L))];
for i from nops(L) by -1 to 1 while L[i]=0 do L:=subsop(i=NULL,L) od;
if L=[] then RETURN(expand(f)) fi;
ham:=evaln(ham);ham:=0;
parse(cat(seq(` for mu`.i.` from L[`.i.`] to `.n.` do`,i=1..nops(L)),
` ham:=subs(e0=1,expand(ham+product('e.(mu.i-L[i])','i'=1..`,nops(L),`)*`,
seq(`diff(`,i=1..nops(L)),convert(expand(f),string),seq(`,e.mu`.i.`)`,
i=1..nops(L)),`))`,seq(` od`,i=1..nops(L)),`;`),statement);
ham*combinat[numbperm](L)/nops(L)!
end:
#############################################################################
#############################################################################

#2. EXAMPLE:
# (SFPACK is our software package containing generators, conversions etc.;
# downloadable from http:/www.cs.kun.nl/~bolke/Research/SFPACK.gz)
# read SFPACK;
# Fx:=expand(3*m([1,2,3],5)+7*m([1,2,2],5)-12):
# Fe:=xTOe(Fx,5):
# Ff:=xTOf(Fx,5):
# Fh:=eTOh(Fe,5):
# Fp:=hTOp(Fh,5):
# Fm:=xTOm(Fx,5):
# Fs:=xTOs(Fx,5):
# trace(HAMMONDe);
# HAMMOND([1,2,2],Fx,5);
# eTOx(HAMMONDe([1,2,2],Fe,5),5);
# ----------------------------------------------------------------------------
# read SFPACK;
# trace(HAMMOND);
# HAMMOND([1,1],Fx,5);
# xTOm('',5);
# G:=elab([1,2,3],5):
# HAMMOND([1,2,3],G,5);
#
# > HAMMOND([1,2,3],G,5);
#                                 1
# > HAMMOND([1,5,0],G,5);
#                                 0
# > HAMMOND([1,1,1],G,5):
#
# > xTOm('',5);
#                          m[2, 1] + 3 m[1, 1, 1]
# > xTOm(G,5);
# 3 m[2, 2, 2]+m[3, 2, 1]+8 m[2, 2, 1, 1]+3 m[3, 1, 1, 1]+22 m[2, 1, 1, 1, 1]
#
# Fx:=expand(3*m([1,2,3],5)+7*m([1,2,2],5)-12):
# HAMMOND([1,2,4],Fx,5);
#                                 0
# HAMMOND([1,2,2],Fx,5);
```

```
#                    3 x5 + 3 x4 + 3 x3 + 3 x2 + 3 x1
# HAMMOND([3,3,1],m([1,2,4],4),4);
# HAMMOND([1,2,4],m([3,3,1],4),4);

###############################################################################
###############################################################################


# 3. The isomorphism mapping the symmetric functions onto the space
# of the Hammond operators.
# MacMahon's ([ ], pg. 27/8/9) operators d.i and D.i:


# DMM.i is the cancellation of part i from a partition m(lambda,n) and
# corresponds with e(i,n)=m([seq(1,j=1..i)],n). DMM.i is called A.i in
vd Corput's ``Scriptum 3".
# It calls the procedure HAMMOND.
DMM:=(k,f,n)->HAMMOND([seq(1,i=1..k)],f,n):

# dMM.i is defined if  a.(i+j)*d/da.j, the sum taken over all j >= 0.
# dMM.i corresponds with p(i,n), the i-the Newton- or power function, so with
# m([i],n). It calls the procedure HAMMOND.
dMM:=(k,f,n)->HAMMOND([k],f,n):

###############################################################################


# projection operator, necessary to apply Hammond operators to Latin
# rectangles. proj(n,m,F) makes all variables x.i zero from and including
# the n+1-th, and all variables 1 to and incl. the m-th.
# F is in x-form; take n >= m.
proj:=(n,m,F)->subs(seq(x.i=1,i=1..m),seq(x.i=0,i=n+1..nops(indets(F))),F):

###############################################################################
###############################################################################


# 4. EXAMPLE:
# Verification of MacMahons counting method for Latin squares.
# (SFPACK is our software package containing generators, conversions etc.;
# downloadable from http:/www.cs.kun.nl/~bolke/Research/SFPACK.gz)

read SFPACK;

n:=3;
lambda:=[seq(2^i,i=0..n-1)];
alist:=[seq(1,i=1..2^n-1)];
mmacht:=expand(xTOe(m(lambda,n),n)^n);
for i to n do mmacht:=HAMMONDe(alist,mmacht,n) od;


# For n=2 this yields 2; for n=3 12, as it should.
# For n=4  all this takes much too much time.


# A better idea therefore is:
# First compute the power m(lambda,n)^n;
# convert this to m-form;
# next, perform ***combinatorial cancellations in the m-components!!!!***

mmacht:=m(lambda,n)^n;
mmacht:=xTOm(mmacht,n):
for i to n do mmacht:=HAM(mmacht,2^n-1,n) od;

# yields very quickly: 576=(4!)^2; in accordance with MacMahon pg. 251.


# HAMMOND itself gives the general isomorphism: given a symmetric function
# f, form xTOm(f,n) and linearly combine the operators of each of the
# components that occur.
```

5 Maple procedures connected with the k-fold symmetric functions.

5.1 Introduction.

Let K be a field of characteristic 0, and R the ring $K[x_1, \ldots, x_n]$ where n is > 0.

The elementary symmetric functions are denoted by $a_0 = 1$, $a_i = 0$ ($i < 0$ or $i > n$), $a_i = \sum_{1 \leq j_1 < j_2 < \ldots < j_i \leq n} x_{j_1} x_{j_2} \ldots x_{j_i}$ ($1 \leq i \leq n$).

By the *Symmetric Function Theorem*, any symmetric function f can be uniquely written as $g(a_1, \ldots, a_n)$ for some $g = g(x_1, \ldots, x_n)$ from R, called the *symmetric representation* of f.

In [], the question was addressed of what happens when this g is symmetric again. This is of course perfectly possible and if it occurs $k-1$ times, f is called *k-fold symmetric*. The number k was called the *symmetric complexity* of f. It is an interesting complexity problem to find a bound on k expressed in the coefficients and exponents of f. Such a result is given in Theorem 1 of [], using a method based on term ordering and the like, familiar from Groebner basis theory [2].

In the course of the investigation, some software was written to calculate the leading terms of the so-called k-fold elementary symmetric functions, defined by a k-fold iteration of the iteration $(x_1, \ldots, x_n) \to (a_1, \ldots, a_n)$.

Another interesting question that arises in a natural way in this context is: how can we describe the behavior (e.g., fixpoints) of this iteration ? We shall give the Maple code for a numerical example for $n = 4$.

Finally, some software was written (and referred to in the paper) to perform some checks on the rather intimidating formulas for the eigenvalues used there. This can also be found below.

5.2 Maple software.

```
# 1. Procedure which computes the leading terms of the k-fold
# elementary symmetric functions.

s:=proc(k,m) local h;
if k=0 then h:=1 elif k>m then h:=0
else h:=s(k-1,m-1)*x.m+s(k,m-1)
fi; expand(h)
end:

ksym :=

  proc(n,k)
  local i,j,m,ksymlist,varlist;
      ksymlist := array(1 .. n);
      varlist := [];
      for i to n do  ksymlist[i] := y.i; varlist := [op(varlist),x.i] od;
      for i to k do
          for j to n do
              for m to n do
```

```
                ksymlist[j] := expand(subs(y.m = s(m,n),ksymlist[j]))
            od
        od;
        for j to n do
            for m to n do  ksymlist[j] := subs(x.m = y.m,ksymlist[j]) od
        od
    od;
    for j to n do
        for m to n do  ksymlist[j] := subs(y.m = x.m,ksymlist[j]) od
    od;
    with(grobner);
    for j to n do  ksymlist[j] := leadmon(ksymlist[j],varlist,plex)[2] od;
    print(ksymlist);
    ksymlist
  end;
ksym(4,4);
```

#2. Finding a fixed polynomial of the elem. symm. f. iteration.

```
with(grobner):
x:=gbasis({x1+x2+x3+x4-x1,x1*x2+x1*x3+x2*x3+x1*x4+x2*x4+x3*x4-x2,x1*x2*x3+
x1*x2*x4+x1*x3*x4+x2*x3*x4-x3,x1*x2*x3*x4-x4},[x1,x2,x3,x4],plex);
```

```
  x := [7 x1 x3 + 7 x3 + 2 x4^8 - 7 x4^7 + 11 x4^6 - 15 x4^5 + 10 x4^4
        - 4 x4^3 + x4^2, 7 x1 x4 - x4^8 + 7 x4^7 - 9 x4^6 + 11 x4^5
        - 5 x4^4 - 5 x4^3 + 3 x4^2 + 7 x4, x2 + x3 + x4,
       7 x3^2 - 7 x3 + x4^8 + 2 x4^6 - 4 x4^5 + 5 x4^4 - 2 x4^3 + 4 x4^2,
       7 x3 x4 - x4^8 - 2 x4^6 + 4 x4^5 - 5 x4^4 + 2 x4^3 + 3 x4^2 - 7 x4,
       -x4^5 - x4^2 + 2 x4^4 + x4^3 + 2 x4^7 + x4^9 - 2 x4^8 - x4^6 ]
```

```
> evalf(solve(x[6]));
  0, 0, .5698402912, .2150798545 + 1.307141280 I,
        .2150798545 - 1.307141280 I, -.6924404010 - .3181479578 I

> x:=expand(subs(x4=%[3],x));
                                                    -9
  x := [7 x1 x3 + 7 x3 - .1 10^   , 3.988882038 x1 + 3.988882038,
                                  2
        x2 + x3 + .5698402912, 7 x3  - 7 x3 + 1.295261628,
                                                  -10
        3.988882038 x3 - 3.011117964, .7 10^    ]
> evalf(solve(x[5]));
                                .7548776663
x:=expand(subs(x3=%,x)); etcetera! Resulting in: with
                         x1 := -1
                         x2 := -1.324717958
                         x3 := .7548776663
                         x4 := .5698402912 we have:

  1 -1.000000001T -1.324717958T2  + .7548776672T3  + .5698402918T4 =

   (-1.T+1)  .   (-1.324717958T+1).(.7548776663T+1).(.5698402912T+1)
```

```
# 3. Some code to check the terrible formulas involving eigenvalues in []
with(linalg):
n:=4;
w:=array(1..n);
alpha:=array(1..n); V:=array(1..n);
lambda:=array(1..n); mu:=array(1..n);
x:=array(1..n,1..n);
```

```
for p to n do hoekp:=-2*p*Pi/(2*n+1);
w[p]:=-evalf(cos(hoekp)+I*sin(hoekp));
alpha[p]:=w[p]+w[p]^(-1);
V[p]:=w[p]-w[p]^(-1);
```

```
lambda[p]:=evalf(4*cos(p*Pi/(2*n+1))^2);
mu[p]:=evalf((-1)^n/(2*cos(2*n*p*Pi/(2*n+1))));
for m to n do x[p,m]:=evalf(2*((-1)^(m+1))*sin(2*p*m*Pi/(2*n+1))/sqrt(2*n+1)) od od;

for p to n do a:=lambda[p]-2+alpha[p] od;
for p to n do a:=V[p]^2-alpha[p]^2+4 od;
for p to n do a:=w[p]^(2*n+1)+1 od;
for p to n do a:=mu[p]*(w[p]^n+w[p]^(-n))-1 od;
for p to n do a:=mu[p]^2*lambda[p]-1 od;
aa:=array(1..n,1..n);for p to n do for m to n do
aa[p,m]:=x[p,m]-(w[p]^m-w[p]^(-m))/(I*sqrt(2*n+1)) od od;evalm(aa);
for p to n do a:=2*w[p]-alpha[p]-V[p] od;
for p to n do a:=2*w[p]^(-1)-alpha[p]+V[p] od;
for p to n do a:=w[p]^2-alpha[p]*w[p]+1 od;
for p to n do a:=w[p]^(-2)-alpha[p]*w[p]^(-1)+1 od;

with(linalg):
n:=4;D1:=array(1..n,1..n);
for i to n do for j to n do
if j>n-i then D1[i,j]:=1 else D1[i,j]:=0 fi od od;
evalm(D1);Y:=D1;Y:=evalm(D1&*Y);

E1:=array(1..n,1..n);
for i to n do for j to n do
if j>=i then E1[i,j]:=1 else E1[i,j]:=0 fi od od;
evalm(E1);Y:=E1;Y:=evalm(Y&*D1);
Y:=evalm(Y&*D1);

Dinv:=array(1..n,1..n);
for i to n do for j to n do
if j=n-i+1 then Dinv[i,j]:=1 elif j=n-i
then Dinv[i,j]:=-1 else Dinv[i,j]:=0 fi od od;
evalm(Dinv);
evalm(Dinv&*D1);
evalm(Dinv&*Dinv);

S:=array(1..n,1..n); D1S:=array(1..n,1..n); aim:=array(1..n,1..n);
for p to n do for m to n do S[m,p]:=x[p,m]; aim[m,p]:=mu[p]*S[m,p] od od;
evalm(D1&*S-aim);

evalm(x&*S);

for p to n do print(abs(mu[p])) od;
for p to n do print(sign(mu[p])*(-1)^(n+p)) od;

k:=5; D1k:=D1;
for i from 1 to k-1 do D1k:=D1k&*D1 od;
D1k:=evalm(D1k);

aim:=array(1..n,1..n);
for i to n do for j to n do
aim[i,j]:=-D1k[i,j];
for p to n do aim[i,j]:=aim[i,j]+evalf(((-1)^(i+j+(n+p)*k))*
sin(2*p*i*Pi/(2*n+1))*sin(2*p*j*Pi/(2*n+1))/
((2*n+1)*(2^(k-2))*cos(p*Pi/(2*n+1))^k)) od od od;
evalm(aim);

Delta:=array(1..n,1..n);
for i to n do for j to n do Delta[i,j]:=0 od od;
for i to n do Delta[i,i]:=mu[i] od;

ST:=array(1..n,1..n);
for i to n do for j to n do ST[i,j]:=S[j,i] od od;
evalm(S&*ST);
evalm(D1-(S&*Delta)&*ST);

aim:=array(1..n,1..n);
```

```
for i to n do for j to n do
aim[i,j]:=-D1k[i,j];for p to n do aim[i,j]:=
aim[i,j]+evalf(mu[p]^k*S[i,p]*S[j,p]) od od od;
evalm(aim);

xp:=2*p*Pi/(2*n+1);
alpha:=cos(xp)+I*sin(xp);

eerste:=sum('(-1)^q*q*sin(q*xp)','q'=1..n);
tweede:=(-1)^n*alpha*(alpha^n-alpha^(n+1))/(2*I*(alpha+1)^2);
derde:=sum('(-1)^q*q*(alpha^q-alpha^(-q))/(2*I)',q=1..n);

evalf(subs(n=5,p=3,eerste));
evalf(subs(n=5,p=3,tweede));
evalf(subs(n=5,p=3,derde));

n:=5; t:=3;

Dtsom1:=0;
for q to n do for p to n do xp:=2*p*Pi/(2*n+1);
Dtsom1:=evalf(Dtsom1+q*((-1)^(n+q+(n+p)*t))*sin(n*xp)*sin(q*xp)/
((2*n+1)*(2^(t-2))*cos(xp/2)^t)) od od;

Dtsom2:=0;
for p to n do xp:=2*p*Pi/(2*n+1);
Dtsom2:=evalf(Dtsom2+((-1)^(p*t))*(sin(n*xp)^2)/(cos(xp/2)^(t+2))) od;
Dtsom2:=evalf(((-1)^(n*t))*Dtsom2/((2*n+1)*(2^t)));

Dtsom1;
Dtsom2;

################################################################################
```

6 REFERENCES.

The list below mainly consists of a compilation of references used in earlier work.

1. A.V. AHO, J.E. HOPCROFT and J.D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.

2. G. ALBERTS, "Jaren van berekening", Thesis, Amsterdam Univ., 1998, Amsterdam Univ. Press, 1998.

3. A. BAKER, "Transcendental Number Theory", Cambridge Univ. Press, 1975.

4. R.E. BLAHUT, "Principles and Practice of Information Theory", Addison-Wesley, 1987.

5. T.S. CHIHARA, "An Introduction to Orthogonal Polynomials", Gordon and Breach, 1978.

6. L. COMTET, "Advanced Combinatorics", Reidel, 1974.

7. J.G. VAN DER CORPUT, "Sur les fonctions symétriques", Scriptum 3, Mathematisch Centrum, Amsterdam, undated.

8. J.G. VAN DER CORPUT, "Symmetrische functies", Christiaan Huygens 18(1940), 251-277.

9. D. COX, J. LITTLE and D. O'SHEA, "Ideals, Varieties and Algorithms", Springer, 1992.

10. F.N. DAVID, BARTON, "Combinatorial Chance", Ch. 17.

11. F.N. DAVID, KENDALL, BARTON, "Symmetric Functions and Allied Tables", Cambridge Univ. Press, 1966.

12. PH.J. DAVIS, "Circulant Matrices", Wiley, 1979.

13. L.E. DICKSON, "Linear Groups with an Exposition of the Galois Field Theory", Dover, 1958.

14. R. FACTH, "Symmetric functions, II: Symmetric functions, the finite case"; Rep. [CSI-R9805, jan. 1998], K.U. Nijmegen; or via http://www.cs.kun.nl/~bolke/Research.html

15. K.O. GEDDES, S.R. CZAPOR, and G. LABAHN, "Algorithms for Computer Algebra", Kluwer, Dordrecht, 1992.

16. C. GODSIL and B. MCKAY, "Asymptotic Enumeration of Latin Rectangles", Journal of Combinatorial Theory, Series B 48(1990), 19-44.

17. R.K. GUY, "Unsolved Problems in Number Theory", Springer, 1981.

18. HARDY and WRIGHT, "An Introduction to the Theory of Numbers", Oxford University Press, 1960.

19. D.C. VAN LEIJENHORST, "How Symmetric Can a Function Be?", Linear Algebra and its Applications 281(1998), 1-10.

20. D.C. VAN LEIJENHORST, "Symmetric Functions, Latin Squares, and Van Der Corputs "Scriptum 3" "; Report Dept. of Comp. Science K.U. of Nijmegen nr. [], to be published.

21. A.H.M. LEVELT, private communication.

22. W.J. LeVEQUE, "Topics in Number Theory" Vols 1, 2; Addison-Wesley, 1958.

23. M. LI and P. VITANYI, "An Introduction to Koplmogorow Complexity and its Applications", Springer, 1993.

24. J.E. LITTLEWOOD and A. WALFISZ, "The Lattice Points of a Circle", Proc. Royal Society A Vol 106 (1924) pp. 479 ff; also in: J.E. Littlewood, Collected Papers, Clarendon Press, Oxford, 1982, pp 939-950.

25. I.G. MACDONALD "Symmetric Functions and Hall Polynomials", Oxford Univ. Press, 1979.

26. P.A. MACMAHON, "Combinatory Analysis Vol. 1 and 2", Chelsea Publishing Company, 1960.

27. F.J. MACWILLIAMS and N.J.A. SLOANE, "The Theory of Error-correcting Codes, North-Holland, 2nd reprint, 1983.

28. B. MCKAY and E. ROGOYSKI, "Latin Squares of Order 10", Electronic Journal of Combinatorics. 2(3): 1-4, 1995.

29. M.B. MONOGAN et al, "Maple V Programming Guide", 2nd printing, Springer, 1996.

30. L.J. MORDELL, "Diophantine Equations", Academic Press, 1969.

31. M.S. PATERSON, N. PIPPENGER and U. ZWICK, "Faster Circuits and Shorter Formulae for Multiple Addition, Multiplication and Symmetric Boolean Functions", in 31st Ann. Symp. on Found. of Comp. Science, IEEE, St. Louis, Missouri, 22-24 Oct. 1990, Vol II, 642-650.

32. N. PIPPENGER, "Short formulas for symmetric functions", IBM Res. Report RC-5143, Yorktown Heights, NY, 1974.

33. J.P.M. SCHALKWIJK, "On a Quantitive Definition of Information and its Inmpact on the Field of Communications", Rep. Prog. Phys. 45 (1982) pp. 1213-1260.

34. V. STRASSEN, "Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoefficienten", Numer. Math. 20 (1973) pp 238-251.

35. B. STURMFELS, "Algorithms in Invariant Theory", Springer, 1993.

36. B.L. of DER WAERDEN, "Algebra" 1 & 2; Springer, 1936/1967/1971.

37. H. WEBER, "Lehrbuch der Algebra", 2e Aufl. Vieweg, Braunschweig, 1899-1912.

38. D. WELLS, "The Penguin Dictionary of Curious and Interesting Numbers", Penguin Books, 1988.