A Maple program for computing $\hat{\theta}^{2\lambda}_{2\mu}$

Murali K. Srinivasan

Abstract

We present a simple recursive Maple implementation of the algorithm described in Section 5 of [**3**] for computing the eigenvalues $\hat{\theta}^{2\lambda}_{2\mu},\ \lambda, \mu \vdash n$ of the Bose-Mesner algebra $\mathcal{B}_{2n}$ of the perfect matching association scheme. This algorithm does not depend on knowing or computing the irreducible characters of the symmetric group.

The program computes, resonably efficiently, any given eigenvalue upto $\mathcal{B}_{40}$. We were able to determine the entire spectrum of the perfect matching derangement matrices in $\mathcal{B}_{2n}$, up to $2n = 40$.

This article describes the Maple program EigenMatch, whose binary file `EigenMatch.m` is available at

`http://www.math.iitb.ac.in/~mks/papers/EigenMatch.m`

To use this file, open a new Maple worksheet, download `EigenMatch.m` into the current working directory `currentdir()` and enter
```
> read "EigenMatch.m"
```
Then enter
```
> hello()
```
to see some examples of its use.

We shall assume familiarity with and use the notation of [**3**]. For recursive Maple programming see Wilf's lecture notes [**4**].

Let $\lambda \in \mathcal{Y}_n,\ \mu \in \mathcal{P}_n$. Consider the three integers

$$\chi^{\lambda}_{\mu},\ \hat{\phi}^{\lambda}_{\mu},\ \hat{\theta}^{2\lambda}_{2\mu},$$

that are, respectively, an irreducible character of S_n, a central character of S_n, and an eigenvalue of $\mathcal{B}_{2n}$. As noted in [**3**], we think of $\hat{\phi}^{\lambda}_{\mu}$ and $\hat{\theta}^{2\lambda}_{2\mu}$ as analogs of each other. They are both eigenvalues of (distinguished basis elements of) commutative $*$-algebras of matrices of dimension $p(n)$, the sizes of the matrices being $n! \times n!$ in the first case and

$(2n-1)!! \times (2n-1)!!$ in the second case. We are concerned with algorithms for computing these quantities.

From (1) of [**3**] we have

$$\hat{\phi}^{\lambda}_{\mu} = \frac{|C_\mu|\chi^{\lambda}_{\mu}}{\dim(V^{\lambda})}.$$

Since there are well known explicit formulas for $|C_\mu|$ and $\dim(V^\lambda)$ we may regard the irreducible and central characters as being equivalent from the point of view of computing them. For instance, it is known that, given λ, μ as inputs, it is #-P hard to determine χ^λ_μ ([**1**]) and it is NP-hard to decide if $\chi^\lambda_\mu = 0$ (see [**2**] which gives precise statements) and thus the same results hold for $\hat{\phi}^\lambda_\mu$. On the other hand, there is a very efficient practical algorithm available in Maple to compute χ^λ_μ for fairly large values of n and this algorithm can be used to calculate $\hat{\phi}^\lambda_\mu$.

How about algorithms for computing $\hat{\theta}^{2\lambda}_{2\mu}$? The author is not conversant with computational complexity but one can expect that, given λ, μ as inputs, it is #-P hard to compute $\hat{\theta}^{2\lambda}_{2\mu}$ and it is NP-hard to decide if $\hat{\theta}^{2\lambda}_{2\mu} = 0$. We consider the problem of giving a practical algorithm that computes $\hat{\theta}^{2\lambda}_{2\mu}$ for reasonable values of n.

There is a classical formula (see Theorem 3.1 in [**3**]) expressing (a multiple of) $\hat{\theta}^{2\lambda}_{2\mu}$ as the sum of the values of the irreducible character $\chi^{2\lambda}$ over a coset of the hyperoctahedral subgroup of S_{2n}. This formula is not combinatorial. Assuming that we can calculate χ^λ_μ, or equivalently $\hat{\phi}^\lambda_\mu$, a combinatorial algorithm to calculate $\hat{\theta}^{2\lambda}_{2\mu}$ was given in Section 3 of [**3**]. But that algorithm is quite involved as it needs inversion of several integer matrices. In Section 5 of [**3**] we gave a very different algorithm, based on eigenvectors (the so called first Gelfand-Tsetlin vectors) and not depending on symmetric group characters, to calculate $\hat{\theta}^{2\lambda}_{2\mu}$. This algorithm is much easier to implement. We present a straightforward recursive Maple implementation. The program computes, resonably quickly, any given eigenvalue upto $\mathcal{B}_{40}$. We were also able to determine the entire spectrum of the perfect matching derangement matrices in $\mathcal{B}_{2n}$, up to $2n = 40$.

We represent partitions by weakly decreasing lists of positive integers. For example, $\lambda = [6, 4, 2, 1]$ is a partition of 13. We shall write three Maple procedures `thetahat,` `eigder,` and `specder` such that, given partitions λ, μ of n, we have

- `thetahat(`2λ`, `2μ`)` is the eigenvalue $\hat{\theta}^{2\lambda}_{2\mu}$ of $\mathcal{B}_{2n}$.
- `eigder(`2λ`)` is the eigenvalue of the perfect matching derangement matrix on the eigenspace $V^{2\lambda}$ of $\mathbb{C}[\mathcal{M}_{2n}]$ (under left action of $\mathcal{B}_{2n}$).
- `specder(`$2n$`)` outputs the entire spectrum of the perfect matching derangement matrix in $\mathcal{B}_{2n}$.

For instance, the following values are computed:

```
thetahat([10,8,6,4,2],[8,6,6,6,4]) = -179200
eigder([12,10,8,6,4,2]) = -31620
```

`specder(14)` outputs the following:

```
[2,2,2,2,2,2,2] :  6
[4,2,2,2,2,2] :  -2
[4,4,2,2,2] :  -8
[4,4,4,2] :  -12
[6,2,2,2,2] :  16
[6,4,2,2] :  16
[6,4,4] :  16
[6,6,2] :  34
[8,2,2,2] :  -84
[8,4,2] :  -92
[8,6] :  -102
[10,2,2] :  664
[10,4] :  688
[12,2] :  -6584
[14] :  79008
```

We now present 14 Maple procedures in numbered items below. The first nine are simple book keeping procedures. The tenth procedure, `updatematch`, is the main workhorse of the eigenvalue computation. This is repeatedly used by the next procedure `thetarow` to recursively calculate an entire row of the eigenvalue table. Given `thetarow`, it is a simple matter to write the last three procedures `thetahat`, `eigder`, and `specder`.

For each procedure we specify the range of validity of the input but do not check for it.

1. We shall need to count, rank, and unrank partitions and pointed partitions. We begin with partitions. Let $p(n,k)$ denote the number of partitions of n with largest part k. We have the recurrence (see [**4**])

$$p(n,k) = p(n-1,k-1) + p(n-k,k),$$

with the initial values $p(n,k) = 0$ if $n \leq 0$ or $k \leq 0$ or $k > n$, and $p(1,1) = 1$.

The following procedure computes $p(n,k)$, for positive integers n,k.

```
park := proc(n,k)
options remember;
if n ≤ 0 or k ≤ 0 or k > n then RETURN(0) fi;
if n = 1 then RETURN(1) fi;
RETURN(park(n-1,k-1) + park(n-k,k));
end:
```

2. We compute the number of partitions of a positive integer n as follows.

```
par := proc(n)
local k;
options remember;
RETURN(add(park(n,k),k=1..n));
end:
```

3. We list partitions of n in lexicographic order. For instance, the ordering of the partitions of 5 is

`[1,1,1,1,1], [2,1,1,1], [2,2,1], [3,1,1], [3,2], [4,1], [5].`

The rank of a partition of n is its position in the above ordering. For example, the rank of `[2,2,1]` above is 3.

The following procedure computes the rank of a partition of a positive integer.

```
rankpar:= proc(L)
local n, i; options remember;
n := add(L[i],i=1..nops(L));
if L[1] = 1 then RETURN(1) fi;
if nops(L) = 1 then RETURN(par(n)) fi;
RETURN(add(park(n,i),i=1..L[1]-1) + rankpar(subsop(1=NULL,L)));
end:
```

4. Given positive integers r, n, with $r \leq$ `par(n)`, the following procedure returns the partition of n with rank r.

```
unrankpar:= proc(r,n)
local j, t; options remember;
if r = 1 then RETURN([seq(1,j=1..n)]) fi;
j := 1; t := park(n,j);
while (t + park(n,j+1)) < r do
  j := j+1; t := t+park(n,j)
od;
RETURN([j+1,op(unrankpar(r-t,n-j-1))]);
end:
```

5. A *pointed partition* of n is a pair (μ, i), where μ is a partition of n and i is a part of μ. Clearly, for a fixed i, the number of pointed partitions (μ, i) of n is the number of partitions of $n - i$.

The following procedure returns the number of pointed partitions of a positive integer.

```
ppar:= proc(n)
local i; options remember;
if n=1 then RETURN(1) fi;
RETURN(1 + add(par(n-i),i=1..n-1));
end:
```

6. We represent a pointed partition (μ, i) by a list L of positive integers defined as follows: the last element of L is i, the list is weakly decreasing from 1 to `nops(L)-1`, and if we sort the elements of L we get the partition μ. For example, here is the list of all pointed partitions of 4:

```
[1,1,1,1], [2,1,1], [1,1,2], [2,2], [3,1], [1,3], [4]
```

We list all the pointed partitions of n as follows: first list all the partitions of $n - 1$ as in item 3 above. Append 1 as the last elements of all these lists (representing partitions of $n - 1$). Then list all the partitions of $n - 2$ as in item 3 above and append a 2 as the last elements of each of these lists. And so on till we append $n - 1$ as the last element. Finally, we add the partition `[n]`. This procedure when applied to $n = 4$ produces the following list:

```
[1,1,1,1], [2,1,1], [3,1], [1,1,2], [2,2], [1,3], [4]
```

The following procedure computes the rank of a pointed partition of a positive integer in the above listing.

```
rankppar:= proc(L)
local n,i; options remember;
n := add(L[i],i=1..nops(L));
if nops(L)=1 then RETURN(ppar(n)) fi;
RETURN(add(par(n-i), i=1..L[-1]-1) + rankpar(subsop(nops(L)=NULL,L)));
end:
```

7. Given positive integers r, n, with $r \leq \texttt{ppar(n)}$, the following procedure returns the pointed partition of n with rank r.

```
unrankppar:= proc(r,n)
local j, t; options remember;
if r = 1 then RETURN([seq(1,j=1..n)]) fi;
if r = ppar(n) then RETURN([n]) fi;
j := 1; t := 0;
while (t + par(n-j)) < r do
  t := t + par(n-j); j:=j+1;
od;
RETURN([op(unrankpar(r-t,n-j)),j]);
end:
```

8. Given a weakly decreasing list of positive integers representing the row lengths of a nonempty Young diagram the following procedure returns the contents of the outer boxes of the Young diagram as a strictly decreasing list of integers.

```
cob:= proc(L)
local i, S; options remember;
if nops(L) = 1 then RETURN([L[1],-1]) fi;
S:= map(x->x-1,cob(subsop(1=NULL,L)));
if L[1] = L[2] then RETURN([L[1], op(subsop(1=NULL,S))]) fi;
RETURN([L[1],op(S)]);
end:
```

9. Given a possibly empty list L of weakly decreasing integers and an integer x the following procedure returns the list obtained by inserting x into L and maintaining weak decrease.

```
insert:= proc(L,x)
if nops(L)=0 then RETURN([x]) fi;
if x ≥ L[1] then RETURN([x,op(L)]) fi;
RETURN([L[1], op(insert(subsop(1=NULL,L),x))]);
end:
```

10. Given a vector $v \in \mathbb{C}[\mathcal{M}_{2n}]$ we store its relative orbital coefficients in an array L of size $pp(n)$. We have $v(2\mu, 2i) = L[j]$, where j is the rank of the pointed partition (μ, i) of n.

Given an array L of size $pp(n)$ containing the relative orbital coefficients of $v \in \mathbb{C}[\mathcal{M}_{2n}]$, $n \geq 1$ and given an integer a the following procedure computes the relative orbital coefficients of $(X_{2n-1} - aI) \cdot v$. It implements Algorithm 1 in Section 5 of [**3**].

```
updatematch:= proc(L::Array,n,a)
local S, A, i, j, u, k;
A:= Array(1..ppar(n),i->0);
for i from 1 to ppar(n) do
  S:=unrankppar(i,n);
  for j from 1 to nops(S)-1 do
    u:= rankppar([op(subsop(j=NULL, subsop(nops(S)=NULL,S))), S[j]+S[-1]]);
    A[u]:=2*S[j]*L[i] + A[u]
  od;
  k:= S[-1];
  for j from 1 to k-1 do
    u:= rankppar([op(insert(subsop(nops(S)=NULL, S), k-j)), j]);
    A[u]:=L[i] + A[u];
    A[i]:=L[i] + A[i];
  od;
od;
for i from 1 to ppar(n) do A[i]:= A[i] - a*L[i] od;
RETURN(A);
end:
```

11. Given a weakly decreasing list L of positive length of even positive integers representing the row lengths of an even Young diagram 2λ with $2n$ boxes, the following procedure returns an array `thetarow(L)` of length $p(n)$ such that, for $1 \le i \le p(n)$, `thetarow(L)[i]` is equal to $\hat{\theta}^{2\lambda}_{2\mu}$, where μ `= unrankpar(i,n)`. It implements Algorithm 2 in Section 5 of [**3**].

```
thetarow:=proc(L)
local n,m,S,R,i,J,F,u,T,v;
options remember;
n:=add(L[i],i=1..nops(L))/2;
if n=1 then return(Array(1..1,1)) fi;
if L[-1]=2 then J:= subsop(nops(L)=NULL,L)
           else J:= [op(subsop(nops(L)=NULL,L)), L[-1]-2]
fi;
S:= thetarow(J);
R:= Array(1..ppar(n),i->0);
for i from 1 to par(n-1) do R[i]:= S[i] od;
F:= cob(J);
if L[-1]=2 then F:= subsop(nops(F)=NULL,F)
           else F:= subsop(nops(F)-1=NULL,F)
fi;
for i from 1 to nops(F) do R:= updatematch(R,n,F[i]) od;
T:= Array(1..par(n),i->0);
for i from 1 to ppar(n) do
  u:= unrankppar(i,n);
  v:= insert(subsop(nops(u)=NULL,u),u[-1]);
  T[rankpar(v)]:= R[i] + T[rankpar(v)];
od;
m:= T[1];
for i from 1 to par(n) do T[i]:= T[i]/m od;
RETURN(T);
end:
```

12. Given two even partitions of $2n$, $n \geq 1$ the following procedure returns the corresponding eigenvalue.

```
thetahat:=proc(lambda,mu)
RETURN(thetarow(lambda)[rankpar(map(x->x/2,mu))]);
end:
```

13. Given an even partition of $2n$, $n \geq 1$ the following procedure returns the corresponding eigenvalue of the perfect matching derangement matrix in $\mathcal{B}_{2n}$.

```
eigder:= proc(lambda)
local n,i,e,T,mu;
n:= add(lambda[i],i=1..nops(lambda))/2;
e:= 0;
T:= thetarow(lambda);
for i from 1 to par(n) do
  mu:= unrankpar(i,n);
  if mu[-1]>1 then e:= e + T[i] fi;
od;
RETURN(e);
end:
```

14. Given an even integer $m \geq 2$ the following procedure outputs the entire spectrum of the perfect matching derangement matrix.

```
specder:= proc(m)
local n, i, L;
n:= m/2;
for i from 1 to par(n) do
  L:= unrankpar(i,n);
  L:= map(x->2*x,L);
printf("          %a : %a\n\n",L,eigder(L));
od;
end:
```

In Section 5 of [**3**] we gave a very similar algorithm for calculating the central characters of S_n without first calculating the irreducible characters of S_n. We have also implemented this algorithm in three further procedures `updateperm`, `phirow`, and `phihat` that are very similar to `updatematch`, `thetarow`, and `thetahat`. The code for these can be viewed by entering

```
> interface(verboseproc = 2):
> print(updateperm); print(phirow); print(phihat)
```

For instance, the following value is computed

```
phihat([6,5,4,3,2,1],[3,3,3,3,3,3,3]) = 2358125
```

Although this algorithm to calculate the central characters is not as efficient as the one based on irreducible characters we have written this program to bring out the essential analogy between $\hat{\phi}^{\lambda}_{\mu}$ and $\hat{\theta}^{2\lambda}_{2\mu}$.

Acknowledgement

I am grateful to Bharat Adsul and Srikanth Srinivasan for helpful discussions and pointers to references about computational complexity.

References

[1] Hepler, C. T., On the complexity of computing characters of finite groups, *Master's thesis, University of Cagary*, (1994), Available at
`https://dspace.ucalgary.ca/handle/1880/45530`

[2] Pak, I., Panova, G., On the complexity of computing Kronecker coefficients, *Comput. Complexity* **26**, 1–36 (2017).

[3] Srinivasan, M. K., The perfect matching association scheme, available at
`http://www.math.iitb.ac.in/~mks/papers/pm.pdf`

[4] Wilf, H. S., East Side, West Side ... an introduction to combinatorial families-with Maple programming, available at
`http://www.math.upenn.edu/~wilf/eastwest.pdf`