MDET – Exact Integer Determinants and Permanents

© 2019 Valentín Albillo

Abstract

MDET is a small subprogram written in 2005 for the HP-71B pocket computer to evaluate determinants and permanents for real or complex NxN matrices. Unlike other methods where floating-point divisions are involved, if the elements are moderately-sized integers the integer results will be exact, even for singular or very ill-conditioned matrices. Two worked examples are included.

Keywords: *determinant, permanent, exact, integer, real, complex, matrix, HP-71B, pocket computer*

1. Introduction

MDET is a short *(5 lines) BASIC* subprogram that I wrote in 2005 for the *HP-71B* pocket computer to evaluate *determinants* and *permanents* for real or complex *NxN* matrices from 2x2 upwards. Unlike other methods where floating-point divisions are involved, if the dimension *N* is small (say *N*<10) and elements are moderately-sized integers (so that no intermediate overflows are produced while doing the computation), then the also integer results will be *exact*, even for *singular* or *very ill-conditioned* matrices, as only integer multiplications and additions/subtractions are performed and no divisions of any type (integer or floating-point).

MDET uses a recursive *general expansion by minors* procedure and works for any dimensions from 2x2 upwards, though it's only reasonably efficient for low-order *N* because computation time grows like *N!*. For that reason, maximum recommended *N* is around 7x7, which already entails ~5000 additions and ~30000 multiplications. Its principal advantages over the usual *LU-decomposition* methods are:

- It produces *exact* integer results for integer matrices (provided there are no intermediate overflows)
- It can compute both *determinants* and *permanents* (which the *Math ROM* doesn't implement and also can't be computed using a classical *LU-decomposition* either)
- It can work with real or *complex* matrices (the *Math ROM* doesn't compute complex determinants)

The code can be greatly optimized, e.g.: it should avoid creating the minor and making the recursive call if the element is 0. Then, instead of always expanding minors by the 1st row, it should find the row/column which has the most *zero* values and do the minor expansion by it, which would save much time. Finally, it should use the exact formula for the 3x3 case to avoid recursive calls for extra speed. However, this version is intended as an academic example for *proof-of-concept* purposes rather than production, so that's left as an exercise to the reader.

2. Program Listing

```
100  SUB MDET(A(,),D)  @ N=UBND(A,1)  @ F=2*FLAG(0)-1
110  IF N=2 THEN D=A(1,1)*A(2,2)+F*A(1,2)*A(2,1) @ END
120  D=0 @  IF TYPE(A)<6 THEN  REAL E,B(N-1,N-1)  ELSE COMPLEX E,B(N-1,N-1)
130  FOR K=1 TO N @ FOR I=2 TO N @ C=1 @ FOR J=1 TO N @ IF J#K THEN B(I-1,C)=A(I,J) @ C=C+1
140  NEXT J @ NEXT I @ CALL MDET(B,E) @ D=D+F^(K+1)*A(1,K)*E @ NEXT K @ END SUB
```

- The line numbering is arbitrary, use whatever suits you as *MDET* doesn't use any line numbers internally.
- *MDET* requires that the matrix passed to it has been declared while `OPTION BASE 1` was (and still is) in effect.
- *MDET* accepts two parameters, both passed by reference: the *NxN* matrix and one scalar variable where the value of the *determinant* or *permanent* will be returned. Both must be of the same type, real or complex.
- *Flag 0*, which is global, specifies whether the *determinant (flag 0 clear)* or the *permanent (flag 0 set)* is returned.
- If you don't have the *Math ROM* you can't work with complex matrices but you *can* do real matrices as long as you delete the boxed parts in the listing above and further pass the dimension *N* to *MDET* as a 3rd parameter. i.e.: edit the declaration and the recursive call to **`SUB MDET(A(,),D,N)`** and **`CALL MDET(B,E,N-1)`**, respectively.

3. Usage Instructions

To use *MDET*:

- first of all *clear* or *set flag 0* to specify computation of the *determinant* or *permanent*, respectively.
- from the keyboard or your own program, call *MDET* passing as parameters (both by reference) the matrix and the scalar variable where the computed value will be returned (both must be of the same type, real or complex), like this:

```
          CALL MDET(M,D)
```

- Upon return from the subprogram, ***D*** will hold the computed value for the determinant or permanent. *The matrix itself remains undisturbed*.

4. Examples

The following examples can be useful to check that the program is correctly entered and to understand its usage. (*both require the Math ROM to be plugged-in as they use some of its keywords for convenience and small size.)*

4.1 Example 1

Compute the exact determinants for the following 7x7 real matrix ***A*** and 3x3 complex matrix ***B***:

$$A = \begin{pmatrix} 58 & 71 & 67 & 36 & 35 & 19 & 60 \\ 50 & 71 & 71 & 56 & 45 & 20 & 52 \\ 64 & 40 & 84 & 50 & 51 & 43 & 69 \\ 31 & 28 & 41 & 54 & 31 & 18 & 33 \\ 45 & 23 & 46 & 38 & 50 & 43 & 50 \\ 41 & 10 & 28 & 17 & 33 & 41 & 46 \\ 66 & 72 & 71 & 38 & 40 & 27 & 69 \end{pmatrix} \quad , \quad B = \begin{pmatrix} 1+2i & 2+3i & 3+i \\ -1+2i & 2-i & -1-i \\ 3i & -2 & 2+2i \end{pmatrix}$$

Instead of doing it right from the keyboard (perfectly possible) we use the following *"driver"* program, which declares and populates the matrices, calls *MDET* to compute their determinants and outputs the results:

```
10  DESTROY ALL @ OPTION BASE 1 @ CFLAG 0
20  DATA 58,71,67,36,35,19,60,50,71,71,56,45,20,52,64,40,84,50,51,43,69,31,28
30  DATA 41,54,31,18,33,45,23,46,38,50,43,50,41,10,28,17,33,41,46,66,72,71,38
40  DATA 40,27,69,(1,2),(2,3),(3,1),(-1,2),(2,-1),(-1,-1),(0,3),(-2,0),(2,2)
50  REAL    A(7,7),D1 @ READ A @ CALL MDET(A,D1) @ DISP "Det(A) =";D1
60  COMPLEX B(3,3),D2 @ READ B @ CALL MDET(B,D2) @ DISP "Det(B) =";D2

[RUN]    Det(A) = 1
         Det(B) = (44,-6)  i.e.:  44 - 6i
```

For comparison purposes, let's compute again the determinant of ***A***, this time using the assembly-language `DET` keyword from the *Math ROM*, which internally uses 15-digits for extra accuracy:

`DET(A)    [ENTER]`   ***.97095056196***   *(not good, the exact result is **1**, as computed above by MDET)*

Thus, despite the 15-digit internal precision the determinant returned by `DET` is correct to *only ~2 digits* and it's a *floating-point* value instead of an *integer* as it should, so this demonstrates an important advantage of *MDET:* it produces *exact* integer values for integer matrices (as long as no intermediate products or sums exceed 12 digits), which `DET` might not, and in this example actually *does not*. This can happen with any *LU*-based algorithm, not just `DET`, and with matrices of any dimension, even 2x2. We can't check the *LU* complex result for matrix ***B*** because `DET` doesn't work with complex matrices, yet another limitation.

The 7x7 particular matrix used in this example is my 7x7 integer-element ***"Albillo Matrix #1"***, as discussed in my ***"HP Article VA016 - Mean Matrices"***. See the article for more details on this and other hugely more difficult ***Albillo Matrices***, some of which require in excess of 20-30 digits or more to be computed correctly using *LU-decomposition* algorithms, despite being of low dimensions and with rather small integer elements.

4.2 Example 2

A teacher gives a test to ***N*** students and wants to let them grade each other's tests, but obviously no students can be allowed to grade *their own* tests. Write a program which accepts as input the number of students and outputs in how many ways the teacher may distribute the tests among the students for grading, subject to said restriction.

Without the restriction, the number of ways would be the *permutations* of ***N*** tests, i.e.: *N!*, but with that restriction in place this is called a *derangement*, and the number of such derangements of a set of size ***N*** is given by the *permanent* of this *NxN* matrix:

$$\textit{Number of ways} = \textit{Permanent of} \begin{pmatrix} 0 & 1 & 1 & .. & 1 \\ 1 & 0 & 1 & .. & 1 \\ 1 & 1 & 0 & .. & 1 \\ .. & .. & .. & .. & .. \\ 1 & 1 & 1 & .. & 0 \end{pmatrix}$$

This program inputs ***N***, creates the associated matrix, calls *MDET* to compute the permanent and outputs it:

```
10  DESTROY ALL @ OPTION BASE 1 @ INPUT "N=";N @ DIM A(N,N),B(N,N)
20  MAT A=CON @ MAT B=IDN @ MAT A=A-B @ SFLAG 0 @ CALL MDET(A,D) @ DISP "# ways =";D
```

Let's run it with *5*, *6* and *7* students:

```
[RUN] N=    5   [R/S]     # ways = 44

[RUN] N=    6   [R/S]     # ways = 265

[RUN] N=    7   [R/S]     # ways = 1854
```

Notes

1. MDET doesn't work for 1x1 matrices, calling it with such a matrix will result in a runtime error.

2. For dimensions greater than 7x7 or so the running time might be excesive so it would be preferable to perform instead the classical *LU decomposition* for speed. However, the accuracy for integer matrices would degrade and for *singular* or very *ill-conditioned* ones it could be unacceptable. Also, the *LU* approach doesn't work at all for *permanents,* though there are faster ways to compute them than *MDET*'s expansion by minors (but still *not* polynomial-time in general.)

References

Francis Scheid (1988). *Schaum's Outline of Theory and Problems of Numerical Analysis, 2nd Edition*.
Wikipedia. *Permanent* and *Derangement* entries.
Valentin Albillo (2005). *HP Article VA016 - Mean Matrices*.

Copyrights