MATLAB$^{®}$ Help for

Difference Sets: Connecting algebra, combinatorics, and geometry

by Emily Moore and Harriet Pollatsek

AMS Student Mathematical Library No. 67

May 2013

Some of the exercises in our book require—or at least suggest—the use of the computer. We have chosen to use MATLAB$^{®}$ to write computer programs to investigate potential difference sets. MATLAB allows us to write functions that can be called by other functions. It uses the array as a fundamental data structure that works well for our purposes. And it has built-in functions that are useful.

This document provides a very brief introduction to MATLAB. It includes

- general guidance on using MATLAB,
- a list of useful built-in functions and special features,
- some techniques for assigning and manipulating vectors and matrices,
- some functions of particular help in searching for difference sets (two functions for checking whether a specified subset of an abelian group is a difference set, two functions for applying the BRC test, and one for creating unions of orbits under a multiplier),
- an illustrative case study.

More MATLAB programs and suggestions for their use appear in the Complete Solutions to Exercises, available to instructors who use our text. (Email `textbooks ams.org` to learn more.) Additional information about using MATLAB is available on the web. If you have suggestions for additions to this document, we would appreciate hearing from you.

Running MATLAB. On many systems, to run MATLAB you click (or double-click) on the MATLAB icon. On other systems, to run MATLAB, type `matlab` in your terminal window. Either opens a command window in which you can type mathematical expressions and can invoke functions.

<u>MATLAB is an interactive facility</u>. It can be used as a fancy calculator. To calculate the value of a MATLAB expression, simply type it at the prompt in the command window. For instance, if you type `3+4`, the response is `ans = 7`. If you type `factor(12)` you invoke the built-in function `factor` with argument `12`, assigning the

resulting vector `[2 2 3]` to the default variable `ans`. Since `ans` can be overwritten by the next command, you may wish to save the results of a calculation in a variable for use later. The command `fac = factor(12)` assigns the vector to `fac`, a variable with name of your own choosing.

In interactive mode, MATLAB stores values in variables that can be used in succeeding calculations. To see the value of any variable, simply type its name. To get a list of the variables saved, type `who`. This workspace disappears when you sign off of MATLAB. You can use the `save` command to save the workspace. Type `help save` for details. Later you can `load` the workspace and start where you left off.

<u>MATLAB is an advanced computer language</u>. You can write a function in MATLAB, and can run the function by giving its name at the command window prompt and supplying any needed input values. One way to enter a function is to type it directly into the command window. You can also cut a function from this file (or from another file) and paste it into the command window. MATLAB stores this function definition in your current workspace. Alternatively, you can write a function and store it in a file with extension `.m`. This file must be stored in your directory where MATLAB can find it. This could be in the directory from which you invoked MATLAB. Or it could be another directory that is on the search path that is stored in the variable `MATLABPATH`. Type this variable name at the prompt to see its value.

Use the `help` command to get more information about a command or function. For instance, type `help eig` to get information about the `eig` function.

The following table lists some built-in functions, and other features of the language. In the examples `X` is a row (or column) matrix.

`all(X)`	true if all entries in `X` are nonzero
`any(X)`	true if any entry in `X` is nonzero
`fac = factor(m)`	returns prime factors, ascending order
`pos = find(X)`	returns positions with nonzero entries
`pos = find(s==X)`	returns positions where an entry of `X` equals `s`
`val = mod(a,b)`	returns the value of `a` mod `b`
`val = prod(X)`	returns the product of values in `X`
`val = abs(num)`	absolute value
`val = round(num)`	`num` is rounded to the nearest integer
`val = sqrt(num)`	square root of `num`
`len = length(X)`	returns the length of vector `X`
`[ln,wd] = size(M)`	returns length and width of matrix `M`
`M2 = reshape(M,1,10)`	requires that matrix `M` have 10 values returns 1×10 vector of these values
`M = zeros(ln,wd)`	returns an `ln` $\times$ `wd` matrix of zeros
`D = eig(M)`	compute eigenvalues
`[V,D] = eig(M)`	compute eigenvalues and eigenvectors
`M = kron(A,B)`	returns the Kronecker product
`help(any)`	get help message for `any`

Since many of our variables are vectors or matrices, it is important to know how to assign and manipulate these structures. The following table shows some useful techniques.

`A = [1,2,3]`	assign row matrix
`B = [4;5;6]`	assign column matrix
`M = [1,2;3,4]`	assign 2 x 2 matrix
`A(3)`	refer to 3rd value in `A`
`B(2)`	refer to 2nd value in `B`
`M(2,1)`	refer to value in row 2 column 1
`C = A*B`	matrix multiplication. `C` is 1x1
`D = B*A`	matrix multiplication. `D` is 3x3
`E = B’`	transpose matrix
`F = [A, B’ ]`	concatenate row matrices
`A = [A, [4]]`	append new value to `A`
`index = [2,3]`	
`G = A(index)`	`G` is a row matrix consisting of entries 2 and 3 of `A`
`H = D(:,2)`	`H` is the second column of `D`

A semicolon at the end of an assignment statement suppresses output. In inner loops of long searches you should use these to save time. Omit the semicolons when you want to see the values.

The boolean-valued function `IsDiffset` on the following page checks whether a subset of a cyclic group is a difference set. Here is an example of its use.

```
>>  OK = IsDiffset( [1 2 4], 7 )
OK = 1                              % 1 indicates true

>>  OK = IsDiffset( [1,2,5], 7 )
OK = 0                              % 0 indicates false
```

The following program tests whether a subset of a cyclic group is a difference set.

```
function[OK] = IsDiffset( D, modi );
%
%  This function accepts a vector D and checks whether it
%  is a difference set in Z_{modi}.  It calculates and
%  tallies all  k(k-1)  nonzero differences.  Then it
%  checks that all counts equal lambda.

v      = modi;
k      = length( D );
lambda = k*(k-1)/(v-1);
if lambda ~= round( lambda )     % lambda must be integer
   OK = false;
   return
end

counts = zeros( 1,v-1 );         % initialize counts
for i = 1:k-1
   for j = i+1:k
      val1 = mod( D(i) - D(j), modi );
      val2 = modi - val1;
      counts( val1 ) = counts( val1 ) + 1;
      counts( val2 ) = counts( val2 ) + 1;
   end
end
OK = all( counts == lambda );
```

Note: In large searches for difference sets in cyclic groups, this function may be called thousands of times. To make the search more efficient, we could pass the values of `v`, `k`, and `lambda` into the function instead of calculating these values. In that case we would also remove the check that `lambda` is an integer.

The next program tests whether a subset of an abelian group is a difference set. It is a generalization of the program `IsDiffset`. Some subtleties are explained on the next page.

```
function[OK] = IsDS2( v, k, lambda, D, modi );
%
%  This function accepts D (an  r x k  array of r-tuples)
%  and  modi (a  1xr  array with moduli).  Each r-tuple
%  in D is an element of the abelian group
%      Z_m1 x Z_m2 x ... x Z_mr
%  where the  m1, m2, ..., mr  are the values in  modi.
%  The function first checks for input errors and then
%  checks to see if  D  is a difference set in the group.

diffs = zeros( r, k*(k-1) );         % preallocate space
                                     %   for differences
col = 0;                             % column for diffs
for i = 1:k-1
   for j = i+1:k
      val1 = mod( D(:,i) - D(:,j), modi' );
      val2 = mod( -val1, modi' );
      diffs(:, col+1) = val1;
      diffs(:, col+2) = val2;
      col = col + 2;
   end
end

counts = zeros( 1,v-1);              % initialize counts
for i = 1:col
   item = diffs(:,i);
   code = item(1);
   for j = 2:r
      code = code*modi(j) + item(j);      % encode item
   end
   counts( code ) = counts( code ) + 1;
end
OK = all( counts == lambda );
```

Notes on function IsDS2:

In $G = \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_r}$ each element is an r-tuple. From an r-tuple we calculate a value `code` that is in the range $1 \ldots v-1$ and is unique for every nonzero r-tuple. For example, if $r = 3$ then the code for the element (a, b, c) is: `code` $= (a \cdot m_2 + b) \cdot m_3 + c$. This is used as the subscript for the element in `counts`.

This is a technical computer science note, but one that can save much time. When a value is appended to a matrix (as in the statement `A = [A, [4]]`), the storage space for the original value cannot accommodate the longer matrix. The computer must allocate extra space. This "dynamic allocation" can be costly both of time and of space. In `IsDS2` we preallocate the space for the list of $k(k-1)$ differences rather than forming the array by appending one difference at a time, making the program more efficient.

The function `IsEven` is small but useful. Like `IsDiffset`, it is a Boolean-valued function. It returns the value 1 (true) if the integer input is even, and 0 (false) if it is not.

```
function[ answer ] = IsEven(n);
%
%  This function checks whether  n  is even.

answer = ( mod(n,2)==0 );
```

The following functions are useful in checking whether parameters pass the BRC test. The function `squarefree` is used in the function `legendre`.

```
function[ newa ] = squarefree( a );
%
%  This function accepts an integer  a  and returns the
%  square-free part of  a.  For example:
%       a =  12     newa =  3
%       a =   9     newa =  1
%       a = -12     newa = -3

list = factor(abs(a));    % prime factors (in order)
len  = length(list);
i = 1;

while i < len
   if list(i) == list(i+1)
      list(i)   = 1;
      list(i+1) = 1;      % change pairs of primes to 1s
      i = i+2;
   else
      i = i+1;
   end
end

newa = prod( list );
if a < 0
   newa = -newa;
end
```

This function uses Legendre's Theorem to check whether a diophantine equation has a nonzero solution.

```
function[ OK ] = legendre( a, b );
%
%  This function accepts integers  a  and  b  and
%  checks whether the equation  x^2 = a y^2 + b z^2
%  has a nonzero solution.  To meet the hypotheses of
%  Legendre's Theorem, for each of integers  a  and  b
%  we find the square-free part.  We also check that
%  at least one of  a  and  b  is positive.

if a <= 0 & b <= 0
   OK = 0;
   return
end

a   = squarefree( a );
b   = squarefree( b );
d   = gcd( a,b );
OK1 = checksquare( a,b );
OK2 = checksquare( b,a );
OK3 = checksquare( -a*b/(d^2), d );
OK  = OK1 & OK2 & OK3;
```

Note that `checksquare(a,b)` checks to see if `a` is a square mod `|b|`. This function is the solution to Exercise 5.6(a).

Case study: Search for a $(63, 31, 15)$-difference set in Z_{63}.

The Second Multiplier Theorem tells us that $t = 2$ is a multiplier for any such difference set. The orbits mod 2 include 1 of length 1, 1 of length 2, 2 of length 3, and 9 of length 6. Any difference set must be equivalent to one fixed by this multiplier, so must be a union of some of these orbits.

$$\begin{array}{ll} (0) & (21, 42) \\ (9, 18, 36) & (27, 54, 45) \\ (1, 2, 4, 8, 16, 32) & (3, 6, 12, 24, 48, 33) \ \ldots \end{array}$$

To get a subset of 31 elements you need to use 0 together with either (a) 5 of the orbits of length 6, or (b) 4 of the orbits of length 6 and the two orbits of length 3.

The following function tries all combinations of both types. Notice that `O1` is a combination of the two orbits of length 3. The sets of values `[s1, s2, ..., s5]` for each successful difference set are stored in the output variable `good`. The display below shows the result when the function in invoked. Note that all rows contain value `s1 = 1`, indicating that all difference sets contain the two orbits of length 3.

```
>> good63 = try63                              % invoke try63

good63 =
     1     2     3     4     9                 % output
     1     2     3     5     7
     1     2     3     6    10
     1     2     4     5     8
     1     2     7     8    10
     1     3     4     5     6
     1     3     5     9    10
     1     3     6     7     9
     1     4     6     7     8
     1     4     8     9    10
     1     5     6     8    10
     1     5     7     8     9
```

```
function[ good ] = try63;
%
%  This function tries combinations of orbits mod 2 to
%  find (63,31,15)-difference sets.
%  O1  holds 2 length-3 orbits; others are length 6.

O1 = [ 9 18 36 27 54 45];   O2  = [ 1  2  4  8 16 32];
O3 = [ 3  6 12 24 48 33];   O4  = [ 5 10 20 40 17 34];
O5 = [ 7 14 28 56 49 35];   O6  = [11 22 44 25 50 37];
O7 = [13 26 52 41 19 38];   O8  = [15 30 60 57 51 39];
O9 = [23 46 29 58 53 43];   O10 = [31 62 61 59 55 47];

orbs = [O1,O2,O3,O4,O5,O6,O7,O8,O9,O10];
good = [];                          % Initially empty

for s1 = 1:6                        % s1 < 7 to allow
   ind1 = [(6*s1-5):(6*s1)];        %   s1 < s2 < ...
   for s2 = s1+1:7
      ind2 = [(6*s2-5):(6*s2)];
      for s3 = s2+1:8
         ind3 = [(6*s3-5):(6*s3)];
         for s4 = s3+1:9
           ind4 = [(6*s4-5):(6*s4)];
           for s5 = s4+1:10
              ind5 = [(6*s5-5):(6*s5)];
              index = [ind1,ind2,ind3,ind4,ind5];
              DD = [0, orbs(index)];
              if IsDiffSet(DD,63)
                 good = [good; [s1,s2,s3,s4,s5]];
              end
            end
         end
      end
   end
end
```