Advanced Features in MATLAB

This is Chapter 7 of the textbook

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2014 ; distribution outside this group of students is NOT allowed.*

1. Introduction [file: d1.m]

`1.1` Sparse matrix representation and function `sparse`. In MATLAB when one issues the command `eye(n)` one creates an identity matrix that is zero except for its diagonal elements. Only n of the n^2 elements of the matrix are nonzeroes. This creates too much waste considering that each element requires 8 bytes when stored as a double. MATLAB allows for the representation of a matrix in `sparse format`. Only the nonzeroes are stored; however for each value stored, the `row,column` index of the element needs to be stored as well. Thus if one uses function `sparse(a)` on a **regular/full/dense** matrix a, one can obtain a sparse representation of a. Of course, if the matrix is small, one wonders whether the overhead of storing `row,column` indexes is worth the savings. The following example shows that even for small n one could achieve some savings!

```
>> a=eye(4);
a =  1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
>> b=sparse(a)
b =(1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
>> a(3,2)=20;
>> b=sparse(a)
b =(1,1)        1
   (2,2)        1
   (3,2)       20
   (3,3)        1
   (4,4)        1
>> whos
  Name      Size                    Bytes  Class
  a         4x4                       128  double array             %
  b         4x4                       120  double array (sparse)    % b is smaller in size than a
```

(c) Copyright A. Gerbessiotis. All rights reserved.

2.1 Sparse matrix generation with `speye`. In MATLAB instead of following the two step process of creating a matrix with **eye** and then making it sparse with `sparse`, one can combine the two steps into one by issuing the sparse equivalent `speye` to `eye`.

2.2 Dense matrix generation from a sparse matrix using `full`. In MATLAB one can also turn a `sparse` matrix into a full (dense) matrix by calling function **full**. Thus, `full(speye(n))` has the effect of `eye(n)`.

2.3 Referencing sparse matrices. One can still retrieve or change elements of a sparse matrix by using our familiar matrix row-column indexing notation. We have already done so in the previous page when we changed the third row and second column of a from 0 to 20 by issuing `a(3,2)=20`. Note that the result shows that even in sparse form, matrices are stored in column-major notation!

2.4 Efficiency considerations. Based on the discussion of the previous page and 2.1-2.3 we can achieve the same task (of generating a sparse matrix) in a variety of ways. One can observe that whereas a requires 128 bytes for storage, b requires only 120 (i.e. sparsity achieves savings even for a 4×4 matrix). Yet in Methods 2,3 below, c and d use up 264 bytes, almost twice as many. Why? In both these methods we extend the number of nonzeroes by one for a sparse matrix c or d that was to have 4 elements. In method 1, we generated a sparse matrix from a full/regular/dense matrix of 5 nonzero elements. Thus before you generate a sparse matrix make sure that it will not be extended. Or if you plan to extend it several times with nonzero elements, convert it back into a full/regular matrix using the **full** function and then convert the full matrix into sparse. For Method 3, this procedure is outlined in Method 4 below. One can observe that h uses 120 rathen than 264 bytes (i.e. the conversion into a dense matrix using `full` and back into a sparse matrix is equivalent to Method 1)!

```
% Method 1
>> a=eye(4)
>> a(3,2)=20
>> b=sparse(a)

>> whos
```

```
% Method 2
 >> c= speye(4)
 >> c(3,2)=20


 >> whos
```

```
% Method 3
>> d=sparse(eye(4))
>> d(3,2)=20


>> whos
```

```
% Method 4
>>d=sparse(eye(4));
>>d(3,2)=20;
>>g=full(d);
  h=sparse(g);
>>whos
```

K2.5 `spalloc`. To avoid such problems one can use `spalloc(m,n,nzmax)` to generate a sparse matrix that can accommodate an $m \times n$ full matrix with up to `nzmax` nonzero elements.

(c) Copyright A. Gerbessiotis. All rights reserved.

3. Other sparse matrix-related functions [file: d2.m]

3.1 Random number generators. Function **sprand** has the effect of `rand` for sparse matrices. But `sprand(m,n,d)` will create an $m \times n$ matrix with approximately $d * m * n$ nonzero values drawn from the uniform distribution in the $(0, 1)$ range. For normally distributed values one can use `sprandn`. Note that this discussion applies to non-trivial case of d, m, n (a d close to 1 and small values for m, n might not convince you of the proper operation of those two functions).

3.2 Retrieving nonzeros elements/indexes of a sparse matrix with find Function `find` returns the indices and/or values of nonzeroes in a sparse matrix. What it returns depends on the type of the return value specified. Three ways of retrieving `find` results are shown below.

```
[ index ]   = find(b)
[ ii jj]    = find(b)
[ ii jj vv] = find(b)
```

In the first case, `find` returns the indexes of the nonzero values in column-major form in an array form. Thus the indexes returned for a sparse matrix originating from an $m \times n$ fullmatrix will be in the range from 1 to $m \cdot n$. In the second case, `find` return row, column indexes. Thus for an $m \times n$ matrix the range of values for ii, jj will be $1..m, 1..n$ respectively. In the third case, the nonzero values are also returned.

For the example `d2.m` a `find(b)` after invoking `d2` will return vector `1 7 8 10 13 19`. This might not make initially much sense; it references however in column-major form the nonzero elements of `b`. Thus the 19-th element of `b` is in the fourth row and fourth column of a 5×4 matrix, and it is the sixth nonzero element of it, i.e. it is element `b(4,4)` which is 1. (If you want to convince yourselves type `full(b)`.)

3.3 Other functions Function **nnz** returns the number of nonzero elements of a sparse matrix. (This is equivalent to `length(find(b))`.) And if we are not interested in indexes but values **nonzeros(b)** will return the nonzero values as a vector i.e. it will return `1 1 30 20 1 1` which is array `vv` in the third form of `find` above. Function **spones** replaces the nonzero values with ones. If one wants to check whether a matrix is sparse or not one can use a predicate function such as `issparse`. One can also apply a function to the nonzero elements of a matrix using `spfun`.
For example `spfun(@exp,b)` or `spfun(@(x) x .^2 , b)` would apply to the nonzeros of sparse b and raise them to the e-th power or square them.
A function `spy` will visualize the sparsity of a matrix by providing an interesting plot.

(c) Copyright A. Gerbessiotis. CS 101 : Fall 2014 . All rights reserved.

Table of sparse matrix functions

Sparse Matrix Function	Meaning
sparse(b)	Create sparse matrix from dense/regular matrix b
speye(n),speye(n,m)	Create a sparse identity matrix
full(b)	Convert a sparse matrix into a full/dense one
b(i,j)	Reference i-th row j-th column of sparse matrix b
sprand(m,n,d)	Create mxn matrix with approximately d*m*n uniformly at random distributed elements
sprandn(m,n,d)	Same as sprand for Normally distributed elements
find(b)	Returns vector of indexes of nonzero elements of sparse b in column-major based indexing from 1 to m*n if a is mxn [index] = find(b) [ii jj] = find(b) [ii jj vv] = find(b)
nnz(b)	Number of nonzeros of b
nonzeros(b)	Vector of nonzeros of b (also use find)
nzmax(b)	Amount of storage allocated for nozeros
issparse(b)	true/false if b is sparse or not
spfun(fun,b)	Apply function fun on the nonzeros of b
spy(b)	Plot the nonzeros of b.
spones	Replace nonzero values with 1
spalloc(m,n,nzmax)	Create sparse matrix mxn to accommodate up to nzmax nonzeros.

(c) Copyright A. Gerbessiotis. All rights reserved.

4. Introduction [file d3.m]

4.1 `cell` array. A cell array is a special MATLAB array whose elements are cells, i.e. containers that can hold other MATLAB arrays. For example one element may hold an array of real numbers, another an array of strings, etc. (Be reminded that a scalar value is an array or rather a 1×1 matrix.) For those of you familiar with C/C++ a cell array cell is a pointer to another data structure. Fof referencing cell arrays use braces {} instead of parentheses ().

4.2 Creation of cell array with `cell`. A function invocation `cell(n), cell(m,n), cell(m,n,p)` will create a $n \times n, m \times n, m \times n \times p$ cell array that is empty. If one does not create a cell array with the `cell` function, one can dynamically create it by just referencing elements of it. Thus `z {1,1} =a` would have created a 1×1 cell array. If we had omitted such a statement and also a `z=cell(2,2)` one, a first statement such as `z{ 2,2 } = b` would have created 2×2 cell array. Thus cell arrays can be dynamically created similarly to regular arrays.

4.3 Displaying with `celldisp`, plotting with `cellplot` and deleting cell arrays. A function invocation `celldisp(a)` will display in column-major form the contents of cell array a. We can delete multiple elements from a cell array or all of it using `clear`. Subsets of cells can be deleted by assigning the emptry array to them. Thus the statement `z(2,:)=[]` will delete the second row of z as shown in the example to follow. (**Note** that parentheses are used on the left side of a delete operation.)

(c) Copyright A. Gerbessiotis. All rights reserved.

4. Introduction continued [file d3.m]

```
>> z=cell(2,2);
>> z{1,1} =a;
z =
    [3x3 double]     []
            []     []
>> z{2,2} = b;
z =
    [3x3 double]            []
            []    [3x3 double]
>> z{2,2}(3,2)
ans =
    20
>> find(z{2,2})
ans =
     1
     5
     6
     9
```

```
>>celldisp(z)
 z{1,1} =   1   0   0
            0   1   0
            0  20   1
 z{2,1} = []
 z{1,2} = []
 z{2,2} = (1,1) 1
          (2,2) 1
          (3,2) 20
          (3,3) 1
```

```
>>z(2,:)= []
      [3x3 double] []
>>celldisp(z)
z{1}   =  1   0   0
          0   1   0
          0  20   1
z{2}   =   []
```

(c) Copyright A. Gerbessiotis. All rights reserved.

5. Cell Arrays of strings [file: d4.m]

5.1 Cell array of strings `cellstr` and `char`. We can store strings in cell arrays instead of using regular arrays. With regular arrays strings need to be padded with spaces so that they all have the same length. Unless of course we use the `char` function that does the conversion for us. We can apply `char` on a cell array of strings as well. Of course we can convert a regular array of strings into a cell array of strings using the function `cellstr` also. This is highlighted in the examples below.

```
>> ab{1}='Computer'                                    % Create a first-column string
ab = 'Computer'
>> ab{2}='Science'                                     % Create a second-column string
ab = 'Computer'    'Science'
>> whos
  Name      Size                    Bytes  Class       % Cell 1x2 array  of 254 bytes
  ab        1x2                       254  cell
>> cd=char(ab)                                         % Convert it into a 2x'1string' regular array
cd =                                                  % with strings padded to 8 bytes
Computer                                              % i.e. create a 2x8 character array
Science
>> whos
  Name      Size                    Bytes  Class
  ab        1x2                       254  cell
  cd        2x8                        32  char        % 2x8x2bytes/char = 32 bytes
>> ef =cellstr(cd);
>> whos
  Name      Size                    Bytes  Class
  ab        1x2                       254  cell
  cd        2x8                        32  char
  ef        1x2                       254  cell
```

(c) Copyright A. Gerbessiotis. All rights reserved.

5. Why cell arrays [file: d4test.m]

5.2 Why cell arrays? Cell arrays can accommodate a variable number of arguments if used as a parameter in a function. If we define function `d4test` so that `varargin` is set to a cell array, different possibilities arise.

```
function d4test(varargin)
 disp(['Number of arguments = ' int2str(nargin)  ]);
 disp('They are as follows');
 disp(varargin);
 disp('End of printing');
end

>> d4test();
>> d4test([1 2  3]);
>> d4test([1;2; 3]);
>> d4test(1, 'test', [1;2;3] , [1,2,3]);
```

(c) Copyright A. Gerbessiotis. All rights reserved.

6. Structure arrays [file: d5.m]

There is yet a third type of an array in MATLAB: the **structure array**.

6.1 Arrays vs Cell arrays vs Structure arrays. An array in MATLAB uses a name that identifies all its elements. All the elements of an array are of the same type. For cell arrays, the elements of a cell array can be of different data types. However for a structure array each individual element of the structure has its own name called a field and each field can have its own data type. An array of structures forms a structure array.

6.2 Example of a structure. We can create a structure called `a` that identifies a student with `fields` that includes `firstname, lastname, qz, ex, mp`. The `qz` is an array itself of 6 grades, `ex` an array of 2 grades, and `mp` an array of 3 grades. Thus below on the left column we create structure a and populate all of its fields with corresponding values. Note that in `a.` we use a period to define and also assign a value to the field of the structure variable `a`. In the second column we decide to turn `a` in a `structure array`. We can extend structure arrays dynamically the same way we extend regular arrays. Thus `a(2).` accesses the second element of a structure array. Then we can print all the information by using `a(1)` to print `John Smith`'s information and `a(2)` to print `Smith Jones`' information.

Sometimes we might preallocate space for a structure array. A call `z(120)=struct('firstname', '2mith', 'lastname','2ones','qz1', 220);` can do that. This creates a structure array of 120 elements with three fields: `firstname,lastname, qz1`. A `struct` function call has as an odd-indexed argument the name of a field in quotes, and the argument immediately following it is the value of that field. Thus `z(120).qz1` is assigned value 220.

```
a.firstname = 'John';       a(2).firstname = 'Smith';
a.lastname  = 'Smith';      a(2).lastname  = 'Jones';
a.qz = [50 40 50 40 50 40]; a(2).qz =[ 40 40 40 40 40 40 ];
a.ex = [300 300]          ; a(2).ex =[ 200 200];
a.mp = [50 50 50]         ; a(2).mp =[50 50 50];
```

(c) Copyright A. Gerbessiotis. All rights reserved.

6.3 More on Structure arrays. One could write `a` and retrieve the fields of a structure array. To retrieve the values of a specific element `a(1)` or `a(2)` is needed for multi-element structure arrays. To retrieve the value of a specific field `a(1).lastname` suffices. Or if the field is an array, `a(1).qz(3)`.

6.4 More on Structure arrays. One can query a specific field or set a value to a specific field by also using the `getfield , setfield` function calls. And one can also remove a field with `rmfield`.

```
>> qz2 = getfield(a,{1},'qz',{2});          %equivalent to qz2grade=  a(1).qz(2);
>> a= setfield(a,{1},'qz',{2},111);         %equivalent to a(1).qz(2)=111; Note that the left
                                            %side is a= !
>> qz2 = getfield(a,{1},'qz',{2});          % is now 111 !
>> a= rmfield(a,'ex');                      % removes the ex field!
>> a(1)
>> a
```

6.5 MATLAB `whos`. If we perform in MATLAB `q= whos('a')` the output of whos is stored in a structure array. Its fields include `name, size, bytes, class, global, sparse, coplex, nesting, persistent.` Moreover `q=whos` sends to q information about all variables defined in the user's workspace.

```
>> q= whos;
>> q(:).name       % the names of all variables
>> eval(q(5).name) % Gives the value of the 5-th variable in the user's workspace
>> getfield(whos,{5},'name') % ans = t
```

(c) Copyright A. Gerbessiotis. All rights reserved.

7. Images in MATLAB [file: d6.m]

7.1 Images in MATLAB and `imread`. In MATLAB one can read an image by using the function call `imread(image,type)`. The `image` is an image-file such as `d6.jpg` enclosed in single quotes, and a type is an image-type such as `jpg` also enclosed in quotes. For example we could have used the statement `pict=imread('d6.jpg','jpg')` to perform this task of opening an image-file.

7.2 Displaying an image with `image`. Having opened an image file, displaying it is straightforward. Just issue `image(pict)` function call to display `d6.jpg`.

7.3 RGB images. An image-handle such as `pict` is a three-dimensional-array. The third dimension has only three planes. Thus pict is of dimension `407x660x3`. The first plance of the third-dimension corresponds to the `Red` component of the image, the second to the `Green` component of the image and the third to the `Blue` component of the image. Thus the image is a `407x660` matrix, with each pixel (short for `pic`ture `el`ement) having three color components corresponding to R, G, and B respectively. Each component has a value from `0` to `255` (i.e. it is one byte long), to indicate how Red (0 means no red, 255 is red) a pixel is. A pixel is completely red if and only if `R=255, G=0, B=0`. Varying the `R, G, B` components one can generate all the colors of the spectrum. One can retrieve then in separate two dimensional matrices the red,green and blue components. For example `g=pict(:,:,2)` retrieves the green component, and then we could diplay its pixel value by plotting them with `plot(g,'g')` in the corresponding color!

```
pict=imread('d6.jpg','jpg');     % Open image
image(pict);                     % Display image
r=pict(:,:,1);                   % Retrieve red
g=pict(:,:,2);                   % Retrieve green component
b=pict(:,:,3);                   % Retrieve blue
plot(g,'g');                     % Plot a histogram of the green values
whos
newpict=pict;
newpict(:,:,3)=0                 % Blue is set to 0!
image(newpict);                  % A yellowish tint results!
```

(c) Copyright A. Gerbessiotis. All rights reserved.

7. Images in MATLAB [file: d6b.m]

7.4 More on images. The following functions manipulate another image available in MATLAB. The second image is an averaged version of the original. Its pixel values however are interpreted differently from the default `colormap` that tells MATLAB how to interpret in color the RGB values. A `colormap(Jet)` is the default one and displays images in a predictable form from blue to yellow to orange to red. A `colormap(HSV)` begins with red passes through yellow,green,cyan,blue,magent to return to red to display periodic functions. Other colormaps are given below.

```
pict=imread('d6a.jpg','jpg');
image(pict);
newpict=mean(pict,3);
image(newpict);
z(1)=subplot(1,2,1);
image(pict); title('RGB original');
z(2)=subplot(1,2,2);
image(newpict); title('New one');
colormap(Hot);
%colormap(Pink);
%colormap(Jet);
%colormap(HSV);
%colormap(Hot);
%colormap(Cool);
%colormap(Spring);
%colormap(Summer);
%colormap(Autumn);
%colormap(Winter);
%colormap(Gray);
%colormap(Bone);
%colormap(Copper);
%colormap(Pink);
%colormap(Lines);
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.1 MATLAB functions on matrices/arrays. Several MATLAB functions are defined in terms of other more primitive MATLAB function. Thus one can call directly the latter rather than the former.

```
length(x)    :   It is   max(size(x))
ndims(x)     :   It is   length(size(x))
numel(x)     :   It is   prod(size(x))

isempty(x)   :   numels(x) == 0
isinf(x)     ;   abs(x) == Inf
isfinite(x)  :   abs(x) ~= Inf
```

8.2 MATLAB functions on vectors. How can one reverse a vector i.e. `x=[1 2 3]` to generate `x= [3 2 1]` or to just shift it left or right by one position (i.e. rotate it) to get `x=[2 3 1]` or `x=[3 1 2]`.

```
x([end 1:end-1])         : shift right/rotate one position
x([end-k+1:end 1:end-k]) : shift right/rotate k   positions

x([2:end 1])
x([k+1:end  1:k])
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.3 Eucilidean distance. In MiniProject 1 we saw how one can find the Euclidean distance of two points (x_1, y_1) and (x_2, y_2) in the two-dimensional place. What if we have two vectors $X = (x_1, x_2, x_3, \ldots, x_n)$, and $Y = (y_1, y_2, y_3, \ldots, y_n)$? How dow we find their distance? We can consider them as points in the n-th dimensional space and work similarly so that $d(X, Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$. In MATLAB we could also use the `norm` function.

```
>> X = [  1 1 1 1 ]
>> Y = [  0 0 0 0 ]
>> norm(X-Y)
   ans = 2
>> sqrt(sum(abs(X-Y) .^2 ))
   ans = 2
>> sqrt(sum((X-Y) .^2 ))
   ans = 2
```

8.4 Overloaded functions. In MATLAB several operators are oveloaded. Thus $+, -$ operate on scalar and matrices indiscriminantly. They also operate on real and complex numbers. But how about some other functions. For example `abs(x)` is not just the absolute value of x. Or it is the absolute value of x if x is a real number.

```
>> clear
>> abs(i)    % i is 1+1i
ans = 1
>>  i*i
ans = -1
>> abs(-2)
ans =2
>> abs(1+i)
ans = 1.4142   % this is the square root of 2
>> abs(3+4i)
ans = 5        % 3^2 + 4^2 = 25
>> abs(3+4*i)
ans = 5        %  4i and 4*i mean the same thing
```

(c) Copyright A. Gerbessiotis. All rights reserved.

8.5 How to generate random vectors. Suppose we want to generate a vector x of $n = 5$ elements, where each element is an arbitrary integer in the range $1..r$, where $r = 10$. We could write this operation as shown below. If we want to generate a vector y of random numbers in an arbitrary range $[a, b]$ we could write as follows as well. This is becauze if z is a random real number $0 \leq z < 1$ then $0 \leq (b - a + 1)z < (b - a + 1)$ and $0 \leq \lfloor (b - a + 1)z \rfloor < b - a + 1$. Since $\lfloor (b - a + 1)z \rfloor$ is an integer less than $b - a + 1$ it can be at most $b - a$. That is $0 \leq \lfloor (b - a + 1)z \rfloor \leq b - a$. Now adding a everywhere we get That is $a \leq a + \lfloor (b - a + 1)z \rfloor \leq b - a + a$. Thus $a \leq Z = (a + \lfloor (b - a + 1)z \rfloor) \leq b$, and Z is an element of vector y below since $a \leq Z \leq b$.

```
>> n=5
>> r=10
>> x= ceil(r*rand(1,n))
>> y= a+ floor((b-a+1)*rand(1,n))
```

8.6 Index to subscript conversion (and the other way around). Suppose we have a 5×5 array A defined as $eye(5)$. What is $A(2, 2)$, what is $A(5)$? $A(2, 2)$ is a subscript description of the element of row 2 and column 2 i.e. a diagonal element that is 1. $A(5)$ is an index description of the 5-th element of A in column-major order i.e. the element of row-5 column-1, i.e. it describes $A(5, 1)$ which is a 0.

```
Let A be an m x n matrix

A(i,j)   is A(i+(j-1)*m)

A(ind)   is A( rem(index-1,m)+1, floor((ind-1)/m)+1 )  % rem(x,y) is x-floor(x/y)*y
```