Effective Programming and Data Types in Matlab

Center for Interdisciplinary Research and Consulting

Department of Mathematics and Statistics

University of Maryland, Baltimore County

www.umbc.edu/circ

Fall 2015

Mission and Goals: The Center for Interdisciplinary Research and Consulting (CIRC) is a consulting service on mathematics and statistics provided by the Department of Mathematics and Statistics at UMBC. Established in 2003, CIRC is dedicated to support interdisciplinary research for the UMBC campus community and the public at large. We provide a full range of consulting services from free initial consulting to long term support for research programs.

CIRC offers mathematical and statistical expertise in broad areas of applications. including biological sciences, engineering, and the social sciences. On the mathematics side, particular strengths include techniques of parallel computing and assistance with software packages such as MATLAB and COMSOL Multiphysics (formerly known as FEMLAB). On the statistics side, areas of particular strength include Toxicology. Industrial Hygiene, Bioequivalence, Biomechanical Engineering, Environmental Science, Finance, Information Theory, and packages such as SAS, SPSS, and S-Plus.

Acknowledgements: We gratefully acknowledge the efforts of the CIRC research assistants and students in Math/Stat 750 Introduction to Interdisciplinary Consulting in developing this tutorial series.

1 Introduction

The purpose of this tutorial is to introduce some of the advanced features of Matlab programming. The material presented in this tutorial is of great importance if one would likes to utilize the full power of Matlab programming. The first thing we discuss is the idea of logical subscripting which is an efficient and at the same time elegant technique unique to Matlab. Next, we introduce the preallocation technique to optimize M-files. Then we talk about optimizing M-files which is where we discuss the very important concept of vectorization of code; this is where a solid knowledge of linear algebra will be important. We will also mention Matlab `structs` and cell arrays which allow the programmers to define composite data structures.

The following list captures the major topics discussed in this tutorial.

1.1 Objectives of This Tutorial

- Logical Subscripting
- Optimizing M-files: Memory Preallocation
- Optimizing M-files: Vectorization
- Optimizing M-files: Sparse Matrices
- Composite data types: structs and cell arrays

2 Logical Subscripting

2.1 Logical Data Type

The logical data type represents a value of true or false which are represented by the numbers 1 and 0, respectively. For instance, consider the following:

```
>> x = 5
x =
     5
>> x == 5
ans =
     1
>>
```

Note that when we compare `x == 5`, Matlab returns the value of 1 because the given logical statement is true in this example. We can also have a vector or a matrix of logicals where each entry will have a value of 0 or 1. For instance, we can compare two vectors (of same size) to get a vector of logical values. The following illustrates the idea.

```
>> x = [4 2 0 -2]
x =
     4     2     0    -2
>> y = [1 2 3 4]
y =
     1     2     3     4
>> x == y
ans =
     0     1     0     0
>> x > y
ans =
     1     0     0     0
>> x <= y
ans =
     0     1     1     1
>>
```

Note that the above results reflect element by element comparison of the vectors x and y.

We can also compare a vector or matrix by a single scalar; this is a technique we will employ shortly in our discussion of logical subscripting. Suppose we are given a vector x and we want to single out elements that are larger than zero. We can do as the following example suggests:

```
>> x = [-5 : 5]
x =
    -5    -4    -3    -2    -1     0     1     2     3     4     5
>> I = x>=0
I =
     0     0     0     0     0     1     1     1     1     1     1
```

Here I is a vector of logicals; elements of I are determined as follows: if an element of x is greater than or equal to zero then the corresponding element of I will be 1, otherwise the corresponding element of I will be zero. The following shows how to do a similar operation with a matrix in which case we end up with a matrix of logical values.

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> A >= 10
ans =
```

```
1     0     0     1
0     1     1     0
0     0     0     1
0     1     1     0
```

2.2 Logical Subscripting

Suppose we are given a vector and we would like to access a specific subset of entries of that vector which satisfy a given condition. We saw in the previous subsection that one can obtain a logical vector which can provide such information; the next step would be access the intended elements using the vector of logicals. For example to obtain the non-negative elements of a given vector `x` one can proceed as follows:

```
>> x = [-5 : 5]
x =
    -5    -4    -3    -2    -1     0     1     2     3     4     5
>> I = x>=0
I =
     0     0     0     0     0     1     1     1     1     1     1
>> p = x(I)
p =
     0     1     2     3     4     5
```

The use of the logical vector `I` to access elements of `x` in the statement `p = x(I)` is what we call logical subscripting (or logical indexing). We give three examples of using logical subscripting in this section.

Our first example is defining piecewise functions. Suppose we are given the function f defined in the interval $[-1, 1]$ as below.

$$f(x) = \begin{cases} -(x-1)(x+1) & \text{if } -1 \le x \le 1, \\ 0 & \text{otherwise.} \end{cases}$$

Using logical subscripting, we can define the above function in Matlab as below

```
function y = f(x)
y = zeros(size(x));
I = (-1 <= x) & (x <= 1);
y(I) = -(x(I)-1).*(x(I)+1);
```

Note that there are other ways of defining the function f in Matlab, but the above piece of code is the one of the most efficient methods to do so. Moreover, we see that the code is rather elegant at the same time.

In the second example, we show how one can plot a curve in such a way that a given portion of the plot is emphasized by the choice of line-style (we could also use different colors). For example, say we want to plot $f(x) = \sin(x)$ in the interval $[0, 2\pi]$, but we want to emphasize the positive part of the curve. The following statements show such an example (see the resulting plot in Figure 1).

```
x = [0 : 0.01 : 2*pi];
y = sin(x);
I1 = x <= pi;
I2 = x > pi;
plot(x(I1), y(I1), '-', x(I2), y(I2),'--');
```

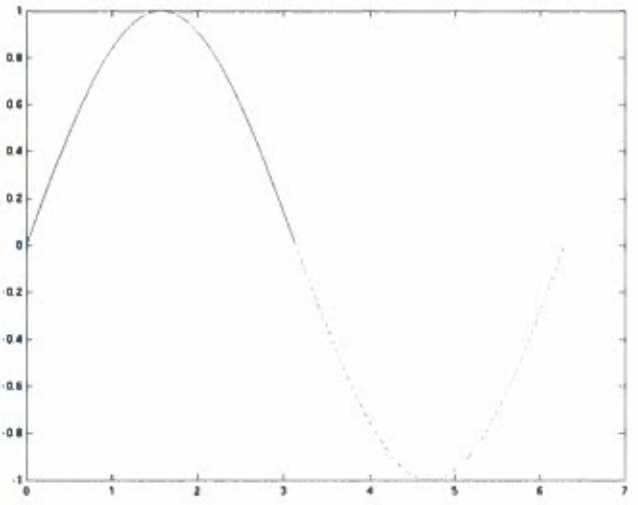

Figure 1: Use of logical subscripting in emphasizing a portion of plot

Our last example of use of logical subscripting is the following. Suppose we are given a matrix A which contains some measurement data. Since every measurement has some error tolerance say τ, we know that if an entry of A is below that tolerance level, then it is practically zero. Thus, we want to zero out those entries of A that fall below a given tolerance, τ. The following piece of code shows how one might do such an operation. Here for the purpose of our example, we generate A using Matlab's random number generator, and let $\tau = 0.005$.

```
>> A = 0.01*rand(5)
A =
    0.0045    0.0038    0.0061    0.0006    0.0008
    0.0004    0.0068    0.0002    0.0037    0.0045
    0.0003    0.0009    0.0002    0.0063    0.0044
    0.0031    0.0004    0.0019    0.0072    0.0035
    0.0001    0.0061    0.0059    0.0069    0.0015
```

```
>> A(A<=0.005) = 0
A =
         0         0    0.0061         0         0
         0    0.0068         0         0         0
         0         0         0    0.0063         0
         0         0         0    0.0072         0
         0    0.0061    0.0059    0.0069         0
```

3 Optimizing M-files: Memory Preallocation

Without preallocation MATLAB will have to resize an array every time it is enlarged in the code. While there are other options available, the main function used for preallocation is `zeros`. Note that due to improvements to automatic array growth in MATLAB 2011a memory preallocation results in a less drastic increase in efficiency for more recent releases of MATLAB as it did for older releases. However, for large enough problems this is still a useful way to optimize code.

The following example shows how memory preallocation can increase the efficiency of code. Suppose we want to compute the first n terms of the following recurrence relation:

$$\begin{cases} x_0 = 0, \\ x_n = 2x_{n-1} + 3 \quad n \geq 1. \end{cases}$$

The following Matlab functions solve the problem:

```
function x=recur1(n)
x = zeros(1,n);         % pre-allocate
for k = 2 : n
    x(k) = x(k-1)*2 + 3;
end

function x=recur2(n)
x = 0;
for k = 2 : n
    x(k) = x(k-1)*2 + 3;
end
```

We note that the only difference between the above function definitions is memory preallocation done right before the `for` loop. For small n the results are not too impressive, but for say $n = 10,000,000$, the first piece of code ran in 0.50 seconds while the second took 2.75 seconds (on the same machine). You can run them in command window, and use `tic` and `toc` to record their work time sperately.

Function	Description
`zeros`	Zeros array
`ones`	Ones array
`eye`	Identity matrix
`rand`	Uniformly distributed random numbers
`randn`	Normally distributed random numbers
`size`	Array dimensions
`length`	Length of array (size of longest dimension)
`end`	Last index in an indexing expression

Table 1: Elementary and special matrices and array information functions

```
>>tic; recur1(10000000); toc
>>tic; recur2(10000000); toc
```

Besides `zeros`, some other elementary and special vectors and matrices are also useful. Table 1 gives basic information about part of them.

4 Optimizing M-files: Vectorization

In this section, we discuss vectorization – the main technique used in optimizing Matlab code. In general, vectorization entails carrying out computations on data stored as vectors (or matrices) using either linear algebra capabilities of Matlab or Matlab's built-in functions that operate on vectors. As a general rule, loops are slow and one should instead utilize Matlab's extensive matrix/vector capabilities whenever possible; the reason is that Matlab's matrix/vector operations are fully optimized. Moreover, readers can be refereed to MathWorks' website for a more detailed guide for code vectorization. Its web page is on

`http://www.mathworks.com/support/tech-notes/1100/1109.html`

We have already seen examples of vectorization in our discussion of logical subscripting. For example, the problem in which we wanted to zero out the elements of a given matrix which fell below a given tolerance could have also been programmed using a double `for` loop, which would also work but would be slower; the use of logical subscripting in that example was an example vectorization. Here we discuss further options to vectorize Matlab code.

Example 1. Our first example of vectorizing a piece of code uses again the idea of logical subscripting. Say we are given two $n \times n$ matrices A and B and we want to form a matrix C which is defined by $C_{ij} = \max(A_{ij}, B_{ij})$; one way to solve this problem is to proceed as follows:

```
C = zeros(n);
for i = 1 : n
```

```
    for j = 1 : n
        C(i,j) = max(A(i,j), B(i,j));
    end
end
```

However, the following (vectorized) version is both shorter and more efficient:

```
C = B;
I = A>B;
C(I)=A(I);
```

Example 2. Another example that shows exactly the same idea of logic subscripting is as follow. Given a 10000×10000 matrix, we want to find out all entries that are smaller than zero, and set them to be zero. You can prepare such matrix A by typing `A=rand(10000)-0.5;`. The non-vectorized code takes much time to finish the problem:

```
tic
[m,n]=size(A)
 for i=1:m
   for j=1:n
     if A(i,j)<0
        A(i,j)=0;
     end
   end
end
toc
```

This is what C language programming does. The vectorized code does it concisely and efficiently.

```
tic
A(A<0)=0;
toc
```

The elapsed time are 7.7306 and 0.1674, respectively (on the same machine). Isn't it shocking? The above examples show the spirit of vectorization in general; we would like to replace loops by operations that utilize Matlab's matrix/vector capabilities. The next example is more mathematical in nature.

Example 3. Given two vectors $\mathbf{a}$ and $\mathbf{b}$, one defines their tensor product $\mathbf{a} \otimes \mathbf{b}$ (a matrix) by the following

$$(\mathbf{a} \otimes \mathbf{b})_{ij} = \mathbf{a}_i \mathbf{b}_j.$$

The following Matlab function returns the tensor product of the vectors `a` and `b`:

Function	Description
`min`	Find the smallest component
`max`	Find the largest component
`sum`	Find the sum of array elements
`cumsum`	Find cumulative sum
`find`	Find indices and values of nonzero elements
`all`	Test to determine if all elements are nonzero
`any`	Test for any non-zeros
`prod`	Find product of array elements
`cumprod`	Find the cumulative product of array elements
`repmat`	Replicate and tile an array
`reshape`	Change the shape of an array
`sort`	Sort array elements in ascending or descending order
`unique`	Find unique elements of a set

Table 2: Some of the Matlab's built-in functions used in vectorization

```
function C=tensor1(a,b)
n = length(a);
C = zeros(n);
for i = 1 : n
    for j = 1 : n
        C(i,j) = a(i)*b(j);
    end
end
```

The following vectorized code does the same thing much more efficiently (at the same time a much shorter code too):

```
function C=tensor2(a,b)
C = a(:)*b(:).'; % column * row
```

Here, `a(:)` converts any row or column vector into a column. `b(:).'` is the non-conjugate transpose. For more information, type `help transpose`, and `help ctranspose`. You can test these two codes like below.

```
>>a=[1:10000];
>>b=[10000:-1:1];
>>tic; tensor1(a,b); toc
>>tic; tensor2(a,b); toc
```

The elapsed time are 3.9380 and 0.2941, respectively (on the same machine). More sophisticated vectorization uses Matlab's built-in function which operate on vectors. Table 2 lists some of the most commonly used Matlab functions in vectorization.

Example 4. Consider the operation of computing the scalar product of two $n \times n$ matrices with real entries. We define,

$$A \cdot B = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} B_{ij}.$$

The Matlab function `testmatrixdot` in the next page provides two implementations of the the above operation in the subfunction `scalar_for` and `scalar_vec`. The function `testmatrixdot` receives n as an input parameter, generates two $n \times n$ random matrices A and B and computes the wall clock run time of both vectorized and non-vectorized version for purposes of comparison.

For example, with $n = 10000$ the following result was obtained:

```
Experiment with n = 10000
For the non-vectorized version t =  1.65
For the vectorized version t =  0.25
```

Note that timing results will vary depending on the machine on which the code is run.

```
function testmatrixdot(n)
A = rand(n);
B = rand(n);

tic;
scalar_for(A,B);
t1 = toc;

tic;
scalar_vec(A,B);
t2 = toc;

fprintf('Experiment with n = %i\n', n);
fprintf('For the non-vectorized version t = %5.2f\n', t1);
fprintf('For the vectorized version t = %5.2f\n', t2);

function c = scalar_vec(A,B)
c = sum(A(:).*B(:));

function c = scalar_for(A,B)
c = 0;
n = size(A,1);
for i = 1 : n
    for j = 1 : n
```

```
        c = c + A(i,j)*B(i,j);
    end
end
```

The benefit of vectorization can also be illustrated by the example of matrix function of two vectors. In some problems, we have two variables in the form of vectors. Each point in one vector need calculating with the other vector through the function. This gives us a grid, and we need to evaluate at every point on the grid. For non-vectorized code, it results in double loops. However, Matlab can perform it with matrix operations.

Example 5. Consider a saddle function $f(x, y)$ of two variables

$$f(x, y) = x^2 - 2y^2,$$

where x and y are vectors. Suppose that x and y are in the interval $[-30, 30]$. We can prepare data with double loops as below:

```
x=-30:.01:30;
y=x;
Z = zeros(length(y),length(x));
count_i=0;
count_j=0;
for i = x
   count_i = count_i+1;
   for j = y
      count_j = count_j + 1;
      Z(count_j,count_i) = i^2-2*j^2;
   end
   count_j = 0;
end
mesh(x,y,Z)
```

A more efficient way to prepare the data is to use the function `meshgrid` as below:

```
x=-30:.3:30;
y=x;
[X,Y]=meshgrid(x,y);
Z=X.^2-2*Y.^2;
mesh(x,y,Z)
title('saddle')
```

This shows you how to evaluate the function of two vectors at every point. In the second line `meshgrid` replicates vector `x` and `y` into arrays `X` and `Y`, where the rows of the output array `X` are copies of the vector `x` and the columns of the output array `Y` are copies of the vector `y`. After this, we can get a grid of values of `Z` by matrix operations. `mesh` is a function that produces 3-D mesh surface plots. The elapsed times are 50.7029 and 1.1642, respectively (on the same machine). This example's graph is shown in Figure 2.

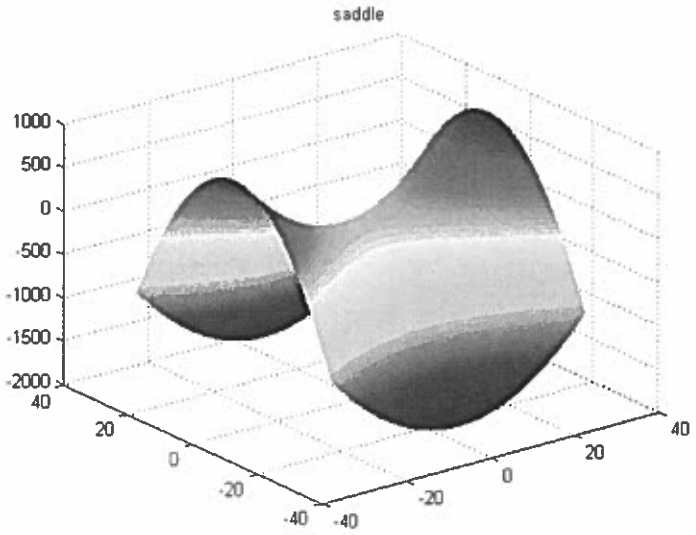

Figure 2: Figure of saddle function

5 Optimizing M-files: Sparse Matrices

A sparse matrix is a matrix with a large percentage of zero elements. Let A be an array of type `double`. Then `nnz(A)` displays the number of nonzero elements of A and `numel(A)` displays the number of elements in A. So, if `nnz(A)/numel(A)` is small then A is a sparse matrix.

MATLAB can take advantage of the sparsity of a matrix by storing and operating only on the nonzeros. MATLAB `double` arrays of dimension up to 2 can have a `sparse` attribute. Such arrays are stored as the nonzero entries together with the row and column indices.

There are a number of ways to generate sparse matrices. The simplest way is to use the `sparse` command which creates a sparse matrix and can take a variable number of inputs. Suppose we define the matrix `A_full` by

```
>> A_full=[0 0 1 0; 2 0 0 3; 0 0 0 0; 0 4 0 5]

A_full =

     0     0     1     0
     2     0     0     3
     0     0     0     0
     0     4     0     5
```

The full matrix `A_full` is converted to sparse storage format by providing the full matrix as a single input to the `sparse` command:

```
>> A=sparse(A_full)

A =

   (2,1)        2
   (4,2)        4
   (1,3)        1
   (2,4)        3
   (4,4)        5
```

MATLAB displays a sparse matrix by listing the nonzero entries preceded by their indices, sorted by columns. We can check that A is indeed a sparse matrix by using the `whos` command:

```
>> whos A_full A
  Name         Size              Bytes  Class     Attributes

  A            4x4                 120  double    sparse
  A_full       4x4                 128  double
```

Notice that the sparse matrix is stored in 120 bytes while the full matrix is stored in 128 bytes. The full matrix is comprised of 16 double precision numbers of 8 bytes each, making a total of 128 bytes. The storage for a sparse m-by-n matrix with `nnz` nonzeros is `16*nnz+8*(n+1)`, making a total of 120 bytes for the matrix `A`.

The command `A = sparse(i,j,s)` defines sparse matrix `A` of dimension `max(i)`-by-`max(j)` with `A(i(k),j(k)) = s(k)`, for `k=1:t` and all other elements zero. So, we can create the matrix A by:

```
i = [1 2 2 4 4];
j = [3 1 4 2 4];
A = sparse(i,j,1:5);
```

The sparse command also accepts three additional arguments. The command `A = sparse(i,j,s,m,n,nzmax)` constructs an m-by-n sparse matrix; the last argument allocates space for `nzmax` nonzeros, which is useful if extra nonzeros, not in `s` are to be introduced later.

Below we will demsontrate the improvement in efficiency by storing appropriate matrices in sparse storage. We create the Wathen matrix A by using MATLAB's `gallery` command in both sparse and full storage and multiply it by a vector x:

```
>>A=gallery('wathen',100,100);
>>B=full(A);
>>x=randn(length(A),1);
>>tic; A*x; toc
>>tic; B*x; toc
```

The elapsed times are 0.1014 and 1.2915, respectively (on the same machine).

6 Composite Data Types

6.1 Struct

The Matlab `struct` data type can store different types of data into a single variable. It is similar to the records in a database, which store a sequence of associated data.

There are two ways to define a `struct` type of data. First, it can be defined by assigned values directly. For example,

```
>>A.a1='abcd';
>>A.a2=100;
>>A.a3=[1 2 3 4];
>>A
 A =

    a1: 'abcd'
    a2: 100
    a3: [1 2 3 4]
```

In this example, before the "." operator in `A.a1`, `A` gives the structure's name. After the "."operator are three fields `a1`, `a2` and `a3`. You can access a certain fields by using "." operation, like `A.a2`, which gives you 100.

Another way to define a structure is by using function `struct`. See the same example.

```
>>A=struct ('a1', 'abcd', 'a2', 100, 'a3', [1 2 3 4])
A =

    a1: 'abcd'
    a2: 100
    a3: [1 2 3 4]
```

In the function `struct`, parameters are field name and field value in alternative. Table 3 lists some functions for structure type of variables.

6.2 Cell Arrays

Cell array is similar to structure in that they collect different types and sizes of data into a single array. You can view a cell arry as a special matrix, where each entry can be of different data types and sizes. You can access individual entry by using the same matrix index as they are in ordinary matrices. For example, we define a 2×2 cell array `cellA` as below. Entries in `cellA` are of different types and sizes.

```
>>A=[1 2; 3 4];
>>B='abcd';
```

Function	Description
`struct`	Create or convert to structure array
`fieldname`	Get structure field names
`getfield`	Get structure field contents
`setfield`	Set structure field contents
`rmfield`	Remove fields from a structure array
`isfield`	True if field is in structure array
`isstruct`	True if a variable is a structures

Table 3: Some functions for structre type of variables.

```
>>C=1:5;
>>D=ones(3);
>>cellA={A B; C D}

cellA =

    [2x2 double]    'abcd'
    [1x5 double]    [3x3 double]
```

You can access an individual entry by using its index, like

```
>>cellA{1,1}

ans =

     1     2
     3     4
```

Notice that we use '{' and '}' to enclose the index to obtain the entry. If we use '(' and ')', Matlab returns the compressed form of this entry.

```
>>cellA(1,1)

ans =

    [2x2 double]
```

Table 4 lists some functions for cell type of variables. For example, we can display the structure of a cell array as nested colored boxes. Type in command window `cellplot(cellA)`. We can see the result in Figure 3

Function	Description
`cell`	Create cell array
`cellfun`	Functions on cell array contents
`celldisp`	Display cell array contents
`cellplot`	Display graphical depiction of cell array
`num2cell`	Convert numeric array into cell array
`cell2struct`	Convert cell array to structure array
`struct2cell`	Convert structure array to cell array
`iscell`	True for cell array

Table 4: Some functions for cell type of variables.

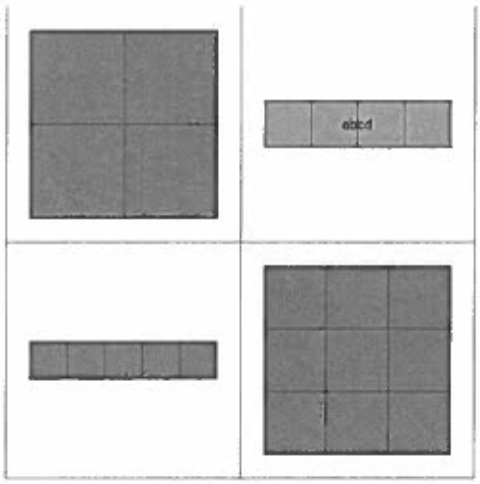

Figure 3: cellplot(cellA).