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

Winter 2008

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.

Copyright © 2003–2008 by the Center for Interdisciplinary Research and Consulting, Department of Mathematics and Statistics, University of Maryland, Baltimore County. All Rights Reserved.

This tutorial is provided as a service of CIRC to the community for personal uses only. Any use beyond this is only acceptable with prior permission from CIRC.

This document is under constant development and will periodically be updated. Standard disclaimers apply.

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.

MATLAB is a registered trademark of The MathWorks, Inc., `www.mathworks.com`.

1 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 200×400 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(200,400)-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 0.1600 and 0.0500, 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 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=[1:1000];
>>b=[1000:-1:1];
>>tic; tensor1(a,b); toc
>>tic; tensor2(a,b); toc
```

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

The elapsed time are 0.1300 and 0.0300, respectively (on the same machine). More sophisticated vectorization uses Matlab's built-in function which operate on vectors. Table **??** 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 $n = 6000$ the following result was obtained:

```
Experiment with n = 6000
For the non-vectorized version t =  5.31
For the vectorized version t =  0.47
```

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 unused] = size(A);
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 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 points. 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. This example's graph is shown in Figure **??**.

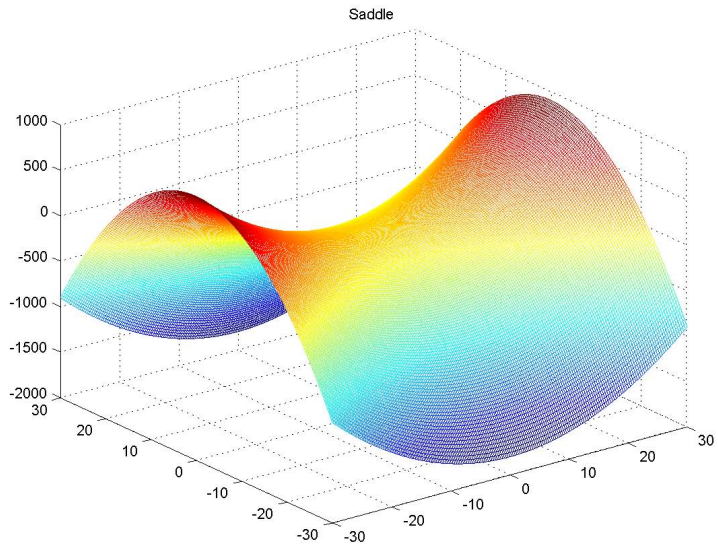

Figure 1: Figure of saddle function

2 Composite Data Types

2.1 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: 2
```

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 2: Some functions for structre type of variables.

```
    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 **??** lists some functions for structure type of variables.

2.2 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';
>>C=1:5;
>>D=ones(3);
>>cellA={A B; C D}

cellA =
```

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 3: Some functions for cell type of variables.

```
    [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 **??** 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 **??**

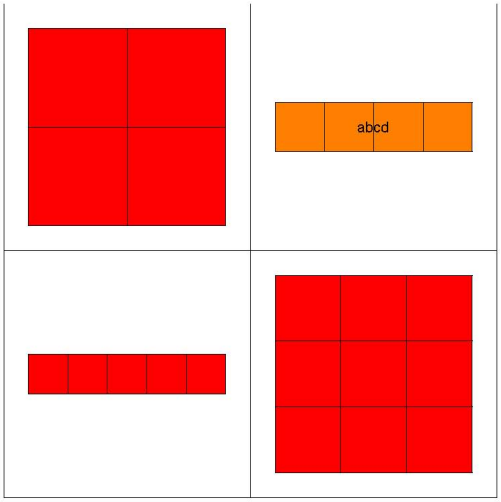

Figure 2: cellplot(cellA).