Writing Fast MATLAB Code

Pascal Getreuer

August 11, 2004

Contents

1	The	Profiler	2
2	Arr	ay Preallocation	3
3	Vec	torization	5
	3.1	Vectorized Computations	5
	3.2	Vectorized Logic	6
	3.3	Example 1: Removing elements	7
	3.4	Example 2: Piecewise functions	8
	3.5	Example 3: Drawing images with meshgrid	8
	3.6	Example 4: Polynomial interpolation	9
4	Ref	erencing Operations	11
	4.1	Subscripts vs. Indices	11
	4.2	Vectorized Subscripts	11
	4.3	Vector Indices	12
	4.4	Reference Wildcards, :	13
	4.5	Deleting Submatrices with []	13
5	Mis	cellaneous Tricks	13
	5.1	Convert any array into a column vector	13
	5.2	Get the number of elements in an array	14
	5.3	Bound a value without if statements	14
	5.4	Find the min/max of a matrix or N-d array	14
	5.5	Repeating/tiling vectors without repmat	15
	5.6	Vectorized use of set on GUI objects	15
6	Goi	ng Further	16

MATLAB is a popular programming language for its simplicity and broad range of tools for topics like signal processing, statistics, differential equations, and of course, matrices. Users can easily add their own m-file functions for specific applications, further extending MATLAB's usefulness.

However, the MATLAB programming language is parsed – code is interpreted and translated into computer operations in realtime – where faster languages like C/C++ are compiled ahead of time into the computer's native language. Some advantages to parsing in realtime are greater platform independence, robustness, and easier debugging. (Also note that MATLAB's "eval" function has no C++ equivalent.) The disadvantage of parsing in realtime is significant loss in speed, as well as increased overhead and less low-level control.

To compensate, MATLAB offers means to help speed up code. This article discusses these and other strategies to improving the speed of MATLAB code.

- The Profiler tool
- Array preallocation
- Vectorized computation and logic
- Vectorized referencing

Caution!

Before beginning, a few notes of caution in code optimization. This applies not only to MATLAB, but programming in general.

- **Remember to comment**: Optimized code especially vectorized code tends to be terse and cryptic. To help others and yourself, remember to comment optimized code.
- **Don't optimize code before its time**: Before ever optimizing code, consider if it will be worth the effort. If the code will soon be revised or extended, it will be rewritten anyway and the time spent optimizing the original is a waste.
- Only optimize where necessary: Make sure the code in consideration is really a speed bottleneck. If it isn't, optimization only obfuscates the code.

1 The Profiler

Since MATLAB 5.0, a tool called the "profiler" helps determine where the bottlenecks are in a program. As an example, consider the following function:

.....

```
function result = example1(Count)
for k = 1:Count
  result(k) = sin(k/50);
  if result(k) < -0.9
    result(k) = gammaln(k);
  end</pre>
```

end

To analyze the efficiency this function, first enable the profiler:

```
>> profile on
```

In case the profiler was on earlier, clear any old profiler data:

```
>> profile clear
```

Next, run the program (change the input argument higher or lower so that it takes about a second):

```
>> example1(5000);
```

Now enter:

```
>> profreport('example1')
```

The profiler will generate an HTML report on the function and launch a browser window. Note that depending on the system, profiler results may be a little different from this example.

MATLAB Profile Report: Summary

Report generated 30-Jul-2004 16:57:01

Total recorded time: 3.09 s Number of M-functions: 4 Clock precision: 0.016 s

Function List

Name	Time	,	Calls	Time/call	Self	time	Location
example1	3.09	100.0%	1	3.094000	2.36	76.3%	C:\MATLABR11\work\example1.m
<u>gammaln</u>	0.73	23.7%	3562	0.000206	0.73	23.7%	C:\MATLABR11\toolbox\matlab\specfun\gammaln.m
<u>profile</u>	0.00	0.0%	1	0.000000	0.00	0.0%	C:\MATLABR11\toolbox\matlab\general\profile.m
profreport	0.00	0.0%	1	0.000000	0.00	0.0%	C:\MATLABR11\toolbox\matlab\general\profreport.m

Clicking the blue "example1" link gives more details:

100% of the total time in this function was spent on the following lines:

```
3: for k = 1:Count
2.1168%4: result(k) = sin(k/50);
5:
0.14 5%6: if result(k) < -0.9
0.8427%7: result(k) = gammaln(k);
8: end</pre>
```

The most time-consuming lines are displayed, along with time, time percentage, and line number. Notice that the most costly lines are the computations on lines 4 and 7.

2 Array Preallocation

One convenient characteristic of MATLAB's default matrix variables is the ability to dynamically augment rows and columns. For example,

MATLAB automatically resizes the matrix. Internally, the memory allocated to the matrix must be reallocated with larger size. If a matrix must be resized repeatedly – like within a for loop – the speed cost becomes noticeable. To avoid this problem, "preallocate" the matrix with the zeros command. Consider the following lines (generates approximate cosine and sine waves of period 100):

0

1

```
.....
```

The profiler timed this code to take 0.47 seconds to perform (system dependent). The big bottleneck is resizing a and b within the loop. After running this code, both arrays are row vectors of length 8000, thus to preallocate, both arrays should be created with 8000 elements.

.....

With this modification, the code takes only 0.14 seconds (over three times faster). Preallocation is often easy to do, in this case it was only necessary to determine the right preallocation size and add two lines.

What if the preallocation size cannot be determined? In some cases, the final array size cannot be determined ahead of time. One strategy is to use the upper bound on the array size and cut the excess after the loop:

```
a = zeros(1,10000); % preallocate
count = 0;
for k = 1:10000
v = exp(rand(1)*rand(1));
    if v > 0.5 % conditionally add to array
        count = count + 1;
        a(count) = v;
        end
end
a = a(1:count); % trim the result
```

Preallocation improved this particular program from taking on average 0.42 seconds to 0.18 seconds.

Preallocation can also be applied to cell arrays, using the cell command to create the desired size. Using preallocation on a frequently resizing cell array results in an even more significant speed up than on double arrays.

3 Vectorization

>> sqrt([1,4;9,16])

Vectorization optimization unlocks MATLAB's processing power and can result in significant speed gains. On the downside, there is no step by step list to vectorization. Good vectorization requires knowledge of MATLAB's available vector functions, strong understanding of the computation to be optimized, and creativity. It is a skill that must be developed.

3.1 Vectorized Computations

Most standard MATLAB functions are "vectorized," meaning they can operate on an array as if the function had been applied individually to every element.

```
ans =
    1    2
    3    4
>> abs([0,1,2,-5,-6,-7])
ans =
    0    1    2    5    6    7
```

When some computation must be performed for every element in an array, the computation is vectorized by performing operations on the array as a whole rather than per element. Consider the following function:

```
function d = minDistance(x,y,z)
% Given a set of 3D points specified by column vectors x,y,z, this
% function computes the minimum distance to the origin
nPoints = length(x);
d = zeros(nPoints,1); % preallocate
for k = 1:nPoints % compute distance for every point
    d(k) = sqrt(x(k)^2 + y(k)^2 + z(k)^2);
end
d = min(d); % get minimum distance
```

For every point, the distance is computed and stored in d. Then, the minimum distance is found using the min function. To vectorize the distance computation, the for loop is replaced with vector operations:

```
function d = minDistance(x,y,z)
% Given a set of 3D points specified by column vectors x,y,z, this
% function computes the minimum distance to the origin
d = sqrt(x.^2 + y.^2 + z.^2); % compute distance for every point
d = min(d); % get minimum distance
```

The modified code performs the distance computation with vector operations. The x, y and z arrays are first squared using the per-element power operator, .[^]. There are also similar per-element multiplication and division operators, .* and ./, respectively. The squared components are added with vector addition. Finally, the square root of the vector sum is computed per element, yielding an array of distances.

.....

The un-vectorized version of the minDistance program takes 0.73 seconds on 50000 points. The vectorized version takes less than 0.04 seconds, more than 18 times faster. Vectorization is usually a much more significant optimization than preallocation. Especially for per element computations on large data sets, vectorization is very effective.

Some functions to keep in mind for vectorizing computations:

min, max, repmat, meshgrid, sum, cumsum, diff, prod, cumprod, filter

3.2 Vectorized Logic

The previous section only shows how to vectorize pure computation. Quite often, a real world computation involves conditionals. Like computations, MATLAB's logic operators are vectorized:

```
>> [1,5,3] < [2,2,4]
ans =
1 0 1
```

Two three-element arrays are compared per-element. The output from logic operations are "logical"-type arrays of ones and zeros.

How is this useful? MATLAB has a few powerful functions for operating on logical arrays:

- find: Find indices of nonzero elements.
- any: True if any element of a vector is nonzero.
- all: True if all elements of a vector are nonzero.

The find function gives back the indices where the vector logic operations return true. In the first command, 1 < 2 is true, 5 < 2 is false, and 3 < 4 is true, so find returns that the first and third comparisons are true. In the second command, find returns the indices where the identity matrix is equal to one. The indices 1, 5, and 9 correspond to the diagonal of a 3 by 3 matrix – see section 5.1 on index references for details.

The any and all functions are simple but can be very useful. If any element is true, any returns true. If all elements are true, all returns true. They are extensions to the standard binary "and" and "or" operations.

3.3 Example 1: Removing elements

The situation often arises where array elements must be removed on some per-element condition. For example, this code removes all NaN and infinity elements from an array \mathbf{x} :

```
i = find(isnan(x) | isinf(x)); % find elements in x that are NaN or infinity
x(i) = []; % delete the elements
```

The first line uses vector logic and find to find the indices of \mathbf{x} elements which are NaN or infinity. The second line deletes the unwanted elements. These two lines can be written equivalently as:

i = find(~isnan(x) & ~isinf(x)); % find elements that are not NaN and not infinity
x = x(i); % keep those elements

This time, the elements of x that are not NaN and not infinity are found with find – it finds the elements that should be retained. The second line replaces x with x(i), keeping only those elements.

3.4 Example 2: Piecewise functions

It is frequently necessary to compute functions with piecewise definitions (for example, spline interpolants, the Heaviside step function, and Chebyshev polynomials). The sinc function is particularly interesting because it is defined everywhere but has a removable singularity at zero. To numerically compute the sinc function, the zero case must be handled separately. The sinc function is defined as

$$\operatorname{sinc}(x) = \begin{cases} \sin(x)/x, & x \neq 0\\ 1, & x = 0 \end{cases}$$

For $x \neq 0$, sinc can be computed with $y = \sin(x) . /x$. This program uses the find function in combination with vectorized computation to handle the two cases separately:

.....

```
function y = sinc(x)
% Computes the sinc function per-element for a set of x values.
```

y = ones(size(x));	% set y to all ones, sinc(0) = 1
i = find(x = 0);	% find nonzero x values
y(i) = sin(x(i)) . / x(i);	% compute sinc where x $\sim = 0$

3.5 Example 3: Drawing images with meshgrid

The **meshgrid** function takes two input vectors and converts them to matrices by replicating the first over the rows and the second over the columns.

```
x =
             2
                     3
                                   5
      1
                            4
             2
                     3
                                   5
      1
                            4
      1
             2
                     3
                            4
                                   5
y =
      1
             1
                     1
                            1
                                   1
      2
             2
                     2
                            2
                                   2
      3
             3
                     3
                            3
                                   3
```

>> [x,y] = meshgrid(1:5,1:3)

The matrices above work like a map for a width 5, height 3 image. For given pixel, the x-location can be read from x and the y-location from y. This is a little wasteful as x and y simply record the column

and row positions, but turns out to be very useful. To draw an ellipse, meshgrid with vector logic can concisely construct the image.

```
% create x and y for a width 150, height 100 image
[x,y] = meshgrid(1:150,1:100);
```

```
% ellipse with origin (60,50) of size 15 x 40
Img = sqrt(((x-60).^2 / 15^2) + ((y-50).^2 / 40^2)) > 1;
```

```
% plot the image
imagesc(Img); colormap(copper);
axis equal; axis off;
```

Drawing lines is almost the same, just a change in the formula.

```
[x,y] = meshgrid(1:150,1:100);
```

% the line y = x*0.8 + 20 Img = (abs((x*0.8 + 20) - y) > 1);

```
imagesc(Img); colormap(copper);
axis equal; axis off;
```

Polar functions can be drawn by first converting x and y variables with the cart2pol function.

```
[x,y] = meshgrid(1:150,1:100);
[th,r] = cart2pol(x - 75,y - 50); % convert to polar
```

```
% spiral centered at (75,50)
Img = sin(r/3 + th);
```

imagesc(Img); colormap(hot); axis equal; axis off;



Note that one can also use the imread function to read an image from a file.

3.6 Example 4: Polynomial interpolation

Sometimes it is necessary to perform an operation not just between every pair of elements of two vectors, but between every combination of elements. This example will show how to do this as part of a polynomial fitting algorithm. Given n points $x_1, x_2, x_3, \ldots x_n$ and n corresponding function values $y_1, y_2, y_3, \ldots y_n$, the coefficients $c_0, c_1, c_2, \ldots c_{n-1}$ of the interpolating polynomial of degree n-1 can be found by solving the matrix equation

$$\begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \cdots & x_1^2 & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \cdots & x_2^2 & x_2 & 1 \\ \vdots & & & & \vdots \\ x_n^{n-1} & x_n^{n-2} & \cdots & x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} c_{n-1} \\ c_{n-2} \\ \vdots \\ c_0 \end{bmatrix} = \begin{bmatrix} y_{n-1} \\ y_{n-2} \\ \vdots \\ y_0 \end{bmatrix}$$





MATLAB can solve such a matrix equation no problem, but the difficult part is setting it up.

```
.....
function c = polyint(x,y)
% Given a set of points and function values x and y,
% computes the interpolating polynomial.
x = x(:);
                                   % make sure x and y are both column vectors
y = y(:);
n = length(x);
                                   % n = number of points
%%% construct the left-hand side matrix %%%
xMatrix = repmat(x, 1, n);
                                  % make an n by n matrix with x on every column
powMatrix = repmat(n-1:-1:0, n, 1);
                                  % make another n by n matrix of exponents
A = xMatrix .^ powMatrix;
                                  % compute the powers
                                   % solve matrix equation for coefficients
c = A \setminus y;
```

The strategy to construct the left-hand side matrix is to first make two n by n matrices of bases and exponents and then put them together with the per element power operator, .[^]. The repmat function ("replicate matrix") is used to make the base matrix xMatrix and the exponent matrix powMatrix.

	x(1)	x(1)	 x(1)		n-1	n-2	 0]
	x(2)	$\mathbf{x}(2)$	 x(2)	powMatrix =	n - 1	n-2	 0
xMatrix =			:		÷		÷
	x(n)	$\mathtt{x}(\mathtt{n})$	 x(n)		n-1	n-2	 0

The xMatrix is made by repeating the column vector x over the columns n times. Similarly, powMatrix is a row vector with elements n - 1, n - 2, n - 3, ..., 0 repeated down the rows n times. MATLAB 4.0 and earlier do not have the repmat function – see section 6.5 for an alternative. The two matrices could also have been created using meshgrid:

[powMatrix, xMatrix] = meshgrid(n-1:-1:0, x);

This function is only an example. Use the standard polyfit function for polynomial interpolation. It has the generality for performing polynomial least-squares fitting in addition to exact interpolation. It is also algorithmically more efficient (though also more complicated), using a Horner-like method to construct the left-hand side matrix (see polyfit.m lines 45-48).

4 Referencing Operations

Although referencing is not often considered an individual operation, referencing in MATLAB is varied and powerful enough to deserve a section of discussion. Good understanding of referencing enables the vectorization of a broader range of programming situations. This section assumes you have experience with the MATLAB language.

4.1 Subscripts vs. Indices

Subscripts are the most common way to address elements in a matrix. For example, consider a 10 by 10 matrix A. Internally, MATLAB stores the matrix linearly down the columns as a one-dimensional, 100-element array of data.

Γ	1	11	21	• • •	81	91
	2	12	22		82	92
	3	13	23		83	93
l	÷					:
L	10	20	30		90	100

An index refers to an element's position in this one-dimensional array. For example, A(83) references the element on row 3, column 9. Using subscripts is much more common, breaking up the reference into a row and column. The syntax A(3,9) also references the element on row 3, column 9.

Conversion between subscripts and indices can be done with the sub2ind and ind2sub functions. However, because these are m-file functions rather than fast built-in operations, it is much more efficient to compute conversions directly. For a two-dimensional matrix A of size M by N, the conversion between subscript (i,j) and index (index):

 $A(i,j) \rightleftharpoons A(i + (j-1)*M)$

A(index) \rightleftharpoons A(rem(index-1,M)+1, floor(index/M)+1)

Indexing extends to N-D matrices as well, with indices increasing first through the columns, then through the rows, through the third dimension, and so on. Subscript notation extends intuitively,

A(..., dim4, dim3, row, col).

4.2 Vectorized Subscripts

More often it is useful to work with submatrices rather than individual elements. This is done by using a vector of indices or subscripts. If **A** is a two-dimensional matrix, a vector subscript reference has the syntax

A(rowv, colv)

where rowv is a vector of rows with M elements and colv is a vector of columns with N elements. Both may be of any length (including the empty matrix) and their elements may be in any order. If either is a matrix, it is reshaped to a column vector. There is no difference between using row vectors or column vectors in vector subscripts.

On the right-hand side (rhs), a vector subscript reference returns a submatrix of elements of size M by N:

$$\begin{array}{cccc} A(\operatorname{rowv}(1),\operatorname{colv}(1)) & A(\operatorname{rowv}(1),\operatorname{colv}(2)) & A(\operatorname{rowv}(1),\operatorname{colv}(3)) & \cdots & A(\operatorname{rowv}(1),\operatorname{colv}(N)) \\ A(\operatorname{rowv}(2),\operatorname{colv}(1)) & A(\operatorname{rowv}(2),\operatorname{colv}(2)) & A(\operatorname{rowv}(2),\operatorname{colv}(3)) & \cdots & A(\operatorname{rowv}(2),\operatorname{colv}(N)) \\ & \vdots & & \vdots \\ A(\operatorname{rowv}(M),\operatorname{colv}(1)) & A(\operatorname{rowv}(M),\operatorname{colv}(2)) & A(\operatorname{rowv}(M),\operatorname{colv}(3)) & \cdots & A(\operatorname{rowv}(M),\operatorname{colv}(N)) \end{array}$$

On the left-hand side (lhs), if the vector subscripted matrix is used as a destination, the rhs result must be of size M by N or scalar. If any elements in the destination reference are repeated, for example, this ambiguous assignment of A(1,2) and A(2,2),

A([1,2],[2,2]) = [1,2;3,4]

A(1,2)	A(1,2)	1	2
A(2, 2)	A(2,2)	3	4

the latter value assigned is the resulting value in the repeated elements.

```
>> A = zeros(2); A([1,2],[2,2]) = [1,2;3,4]
```

```
A =
```

0 2 0 4

Vector subscript references extend intuitively in higher dimensions.

4.3 Vector Indices

Multiple elements can also be referenced with vector indices.

A(indexv)

where indexv is an array of indices. On the rhs, a vector index reference returns a matrix the same size as indexv. If indexv is a row vector, A(indexv) returns a row vector. If indexv is a 3 by 4 matrix, A(indexv) is a 3 by 4 matrix.

```
\mathbb{A}(\texttt{indexv}) = \left[ \begin{array}{ccc} \mathbb{A}(\texttt{indexv}(1,1)) & \mathbb{A}(\texttt{indexv}(1,2)) & \mathbb{A}(\texttt{indexv}(1,3)) & \mathbb{A}(\texttt{indexv}(1,4)) \\ \mathbb{A}(\texttt{indexv}(2,1)) & \mathbb{A}(\texttt{indexv}(2,2)) & \mathbb{A}(\texttt{indexv}(2,3)) & \mathbb{A}(\texttt{indexv}(2,4)) \\ \mathbb{A}(\texttt{indexv}(3,1)) & \mathbb{A}(\texttt{indexv}(3,2)) & \mathbb{A}(\texttt{indexv}(3,3)) & \mathbb{A}(\texttt{indexv}(3,4)) \end{array} \right]
```

Where vector subscripts are limited to referring to block-shaped submatrices, vector indices refer to individual elements and can refer to any shape.

If a vector index reference is on the lhs, the rhs must return a matrix of the same size as indexv or a scalar. As with vector subscripts, ambiguous duplicate assignments use the later value assigned.

>> A = zeros(2); A([3,4,3,4]) = [1,2,3,4]

A = 0 0

3

4

4.4 Reference Wildcards, :

Using the wildcard, :, in a subscript refers to an entire row or column. For example, A(:,1) refers to every row in column one – the entire first column. This can be combined with vector subscripts, A([2,4],:) refers to the second and fourth rows.

When the wildcard is used in a vector index, the entire matrix is referenced. On the rhs, this always returns a column vector.

A(:) = column vector of length prod(size(A))

This is frequently useful: For example, if a function input must be a row vector, the user's input can be quickly reshaped into row vector with A(:)' (make a column vector and transpose to a row vector).

Because A(:) refers to all elements, A(:) on the lhs assigns all the elements of A, but does not change its size. For example, A(:) = 8 changes all elements of matrix A to 8.

4.5 Deleting Submatrices with []

Elements in a matrix can be deleted by assigning the empty matrix. For example, A([3,5]) = [] deletes the third and fifth element from A. After a deletion using index references, the matrix is reshaped into a row vector.

It is also possible to use deletion with subscripts, but only when at most one subscript is not the wildcard. A(2,:) = [] deletes the second row, resulting in a matrix with one less row. Deletions like A(2,1) = [] or even A(2,1:end) = [] are not allowed.

5 Miscellaneous Tricks

5.1 Convert any array into a column vector

It is often useful to force an array to be a column vector, for example, when writing a function expecting a column vector as an input. This simple trick will convert the input array to a column vector if the array is a row vector, a matrix, an N-d array, or already a column vector.

x = x(:); % convert x to a column vector

Note that by following this operation with a transpose, it is also possible to force an array to be a row vector.

5.2 Get the number of elements in an array

The **size** function gives back a vector of an array's dimensions. Along with **prod**, the number of elements in an array is concisely computed with

nElements = prod(size(x)); % compute the number of elements in x

Use ndims(x) or length(size(x)) to get the number of dimensions in an array.

5.3 Bound a value without if statements

If the problem is "value x must be at least a and at most b," the most straightforward way to code this is with if and elseif statements. Unfortunately, MATLAB if statements are slow. Furthermore, if it is necessary to bound every value in a set, this must be coded within a for loop or vectorized using the find function. This method uses the min and max functions to bound x between lowerBound and upperBound:

x	=	<pre>max(x,lowerBound);</pre>	%	make	x	>=	lowerBound
x	=	<pre>min(x,upperBound);</pre>	%	make	x	<=	upperBound

This will work if x is a scalar or a matrix of any size.

5.4 Find the min/max of a matrix or N-d array

Given a matrix input (with no other inputs), the min and max functions operate along the columns, finding the extreme element in each column. Usually it is more useful to find the extreme element of the entire matrix. It is possible to repeat the min or max function on the column extremes; min(min(A)) to find the minimum of a two-dimensional matrix A. However, if the location of the extreme element is also necessary, analyzing the indices output (second output) from multiple min/max calls is cumbersome, especially for N-d arrays. This method uses only one call to min/max (using the convert to column vector trick) and determines the extreme element's location:

```
[MinValue, MinIndex] = min(A(:));  % find minimum element in A
  % the minimum value is MinValue, the index is MinIndex
MinSub = ind2sub(size(A), MinIndex); % convert MinIndex to subscripts
```

The minimum element is A(MinIndex) (index reference) or alternatively A(MinSub(1), MinSub(2), ...) (subscript reference). See section 5.1 to learn more about the difference between index and subscript referencing. To find the maximum value, replace the call to min with max.

5.5 Repeating/tiling vectors without repmat

As of MATLAB 5.3 (R11), the repmat function is implemented as an m-file rather than as a built-in function. Although it is a very fast function, it still incurs the overhead costs that any m-file carries. Thus it is of no surprise that most standard MATLAB m-file functions perform repmat inline rather than calling it (see meshgrid.m lines 41, 42, 55, ...; interp1.m lines 225, 240, 322, ...; interp2.m lines 232, 233, 240, ...; and blanks.m line 13). While repmat has the generality to operate on matrices, it is often only necessary to tile a vector or just a scalar. To repeat a column vector y over the columns n times,

A = y(:,ones(1,n)); % equivalent to A = repmat(y,1,n);

To repeat a row vector **x** over the rows **m** times,

A = x(ones(1,m),:); % equivalent to A = repmat(x,m,1);

To repeat a scalar **s** into an **m** by **n** matrix,

A = s(ones(m,n)); % equivalent to A = repmat(s,m,n);

or alternatively

A = zeros(m,n);	%	make A	aı	ı m	by 1	n n	natrix
A(:) = s;	%	assign	s	to	eve	ry	element

See section 5.2 on vectorized subscript references for details on how this works. Notice that this method only uses the built-in **ones** function – this is basically how **repmat.m** performs the tiling itself (see lines 46, 52, 53).

5.6 Vectorized use of set on GUI objects

A real-world graphical user interface (GUI) can have dozens of objects for text labels, buttons, sliders, and so forth. These objects must all be meticulously initialized with the uicontrol function with lengthy string arguments. For example, to define three edit boxes with white text background (default is gray) and left text alignment (default is center):

```
uicontrol('Units', 'normalized', 'Position', [0.1,0.9,0.7,0.05], ...
 'HorizontalAlignment', 'left', 'Style', 'edit', 'BackgroundColor', [1,1,1]);
uicontrol('Units', 'normalized', 'Position', [0.1,0.8,0.7,0.05], ...
 'HorizontalAlignment', 'left', 'Style', 'edit', 'BackgroundColor', [1,1,1]);
uicontrol('Units', 'normalized', 'Position', [0.1,0.7,0.7,0.05], ...
 'HorizontalAlignment', 'left', 'Style', 'edit', 'BackgroundColor', [1,1,1]);
```

This code is a lot of text for three edit boxes – notice that the only change between the three objects is the position parameter. To take advantage of the repeated parameters, a vectorized call to **set** can reduce the wordiness:

```
h(1) = uicontrol('Units', 'normalized', 'Position', [0.1,0.9,0.7,0.05]);
h(2) = uicontrol('Units', 'normalized', 'Position', [0.1,0.8,0.7,0.05]);
h(3) = uicontrol('Units', 'normalized', 'Position', [0.1,0.7,0.7,0.05]);
set(h, 'HorizontalAlignment', 'left', 'Style', 'edit', 'BackgroundColor', [1,1,1]);
```

This method has also has the advantage that changing common parameters for multiple objects can be done with one change rather than changing every object.

6 Going Further

If you have a coding situation which cannot be optimized any further, keep in mind MATLAB's natural limitations as a parsed language. If the code is either very **for** loop oriented or requires the top possible speed, consider MEX-files. With an external C or Fortran compiler, it is possible write and compile C or Fortran functions that interface with MATLAB. See the MathWorks MEX-files Guide

http://www.mathworks.com/support/tech-notes/1600/1605.html to get started. Writing MEX-files is quite different from writing m-files, and takes some effort and patience to learn. However, with a well written MEX-file, the speed gains over the equivalent m-file program can easily be 10 to 20 times faster.

Hopefully you have found some suggestions relevant to your specific projects or at least learned something new in this article. To be a good programmer, never assume your method is best but use the best that you know – stay open to other programmers' ideas. Along those lines, this article is only one humble addition to the collection of MATLAB programming techniques.