1109 How Do I Vectorize My Code?

Revison: 1.3 *Last Date Modified: 31-August-2000*

This technical note provides an introduction to vectorization techniques. In order to understand some of the tricks available, an introduction to MATLAB indexing is provided. Then several vectorization techniques are discussed, in order of simplest to most complicated.

There are two aspects to this presentation. Knowledge of all the techniques available is only half of the battle. The other half is knowing when to use them -- recognizing situations where this approach or that one is likely to yield a better (quicker, cleaner) algorithm. Each section provides an example, which proceeds from a description of the problem to a final solution. We hope that illustrating the process by example achieves both of these goals.

1. MATLAB Subscripting

A powerful feature of MATLAB is the ability to select subsets of an array or matrix. There are two types of subscripting available in MATLAB.

In *indexed subscripting*, the values of the subscript are the indices of the matrix where the matrix's values are desired. Thus, if A = 1:5, then A([3,5]) denotes the third and fifth elements of matrix A:

```
» A = 1:5;
» A([3,5])
ans =

  3 5
```

The second type of subscripting is *logical subscripting*. With this type of subscripting, the subscript is a matrix the same size as A containing only 0's

and 1's.

The elements of A that are selected have a '1' in the corresponding position of the subscripting matrix. For example, if A = 1:5, then A(logical([0 0 1 0 1])) denotes the third and fifth elements of A:

```
» A = 1:5;
» A(logical([0 0 1 0 1]))
ans =

  3 5
```

This second type of subscripting is very powerful, and we use it frequently in the following sections. For more information on subscripting, see the *Using MATLAB* manual.

2. Basic Array Operations: y(i) = fcn(x1(i), x2(i), ...)

The simplest type of vector operations in MATLAB can be thought of as *bulk processing*. In this approach, the same operation is performed for each corresponding element in a data set (which may include more than one matrix).

Suppose you have some data from an experiment. The data measurements are the length L, width W, height H, and mass M of an object, and you want to find the density D of the object. If you had run the experiment once, you would just have one value for each of the four observables (i.e., L, W, H, and M are scalars). Here is the calculation you want to run:

```
D = M/(L*W*H)
```

Now, suppose that you actually run the experiment 20 times. Now L, W, H, and M are vectors of length 20, and you want to calculate the corresponding vector D, which represents the density for each run.

In most programming languages, you would set up a loop, the equivalent of the following MATLAB code:

```
» for i = 1:20
D(i) = M(i)/(L(i)*W(i)*H(i));
```

```
end
```

The beauty of MATLAB is that it allows you to ignore the fact that you actually ran 20 experiments. You can perform the calculation element-by-element over each vector, with (almost) the same syntax as the first equation, above:

```
» D = M./(L.*W.*H);
```

The only difference is the use of the .* and ./ operators. These differentiate element-by-element operators, called array operators, from the (quite different) linear algebra operators, called matrix operators. For example, * denotes matrix multiplication, ^ denotes matrix exponentiation, etc. (See Strang, *Linear Algebra with Applications*, for an introduction to linear algebra operators.) In this context you should use the array operators, although the matrix operators will become useful later on.

3. Boolean Array Operations: y = bool(x1, x2, ...)

A logical extension of the *bulk processing* idea is to vectorize comparisons and decision making. MATLAB comparison operators, like the array operators above, accept vector inputs and produce vector outputs.

Suppose that, after running the above experiment, you find that negative numbers have been measured for the mass. As these values are clearly erroneous, you decide that those runs for the experiment are invalid.

You can find out which runs are valid using the >= operator on the vector M:

```
» M = [-.2 1.0 1.5 3 -1 4.2 3.14];
» M>=0
ans =

  0 1 1 1 0 1 1
```

Now you can exploit the indexing power of MATLAB to remove the erroneous values:

```
» D = D(M>=0);
```

This selects the subset of D for which the corresponding elements of M are non-negative.

Suppose further that, if all values for masses are negative, you want to display a warning message and exit the routine. You need a way to condense the vector of Boolean values into a single value. There are two vectorized Boolean operators, any and all, which perform Boolean AND and OR functions over a vector. Thus you can perform the following test:

```
if all(M<0)
disp('Warning: all values of mass are negative.');
return;
end
```

You can also compare two vectors of the same size using the Boolean operators, resulting in expressions such as:

```
» (M>=0) & (L>H)
```

The result is a vector of the same size as the inputs.

Here are two notes on the use of the Boolean functions. First, since MATLAB uses IEEE arithmetic, there are special values to denote overflow, underflow, and undefined operations: Inf, -Inf, and NaN, respectively. Inf and -Inf can be tested for normally (i.e., Inf == Inf returns true), but, by the IEEE standard, NaN is never equal to anything (even other NaN's). Therefore, there are special Boolean operators, isnan and isinf, to help test for these values.

Second, if you try to compare two matrices of different sizes, an error results. You need to check sizes explicitly if the operands might not be the same size.

4. Constructing Matrices from Vectors: y(:,i) = f(x(:))

Creating some simple matrices, like constant matrices, is easy in MATLAB. The following code creates a matrix of all 10s:

```
» A = ones(5,5)*10
```

It turns out that the multiplication here is not necessary; we can get around it using an indexing trick. We'll see this in a moment.

Another common application involves selecting specified elements of a vector to create a new matrix. This is often a simple application of the indexing power of MATLAB. For example, to create a matrix corresponding to elements [2, 3; 5, 6; 8, 9; 11, 12; ...] of a vector x:

```
» x = 1:51;<
» x = reshape(x, 3, length(x)/3);
» A = x(2:3,:)';
```

We are using indexing to exploit the symmetry of the desired matrix. There are several tools available to exploit different types of symmetry.

There is a famous trick for duplicating a vector of size M by 1, n times, to create a matrix of size M by N . In this method, known as Tony's Trick, the first column of the vector is indexed (or referenced, or selected) n times:

```
» v = (1:5)';
» M = v(:,ones(3,1))
M =

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

Now we realize how our first matrix, size 5 by 5 constant matrix of all 10s, can be created without matrix multiplication:

```
» A = 10;
» A = A(ones(5))
A =

  10 10 10 10 10
  10 10 10 10 10
  10 10 10 10 10
  10 10 10 10 10
  10 10 10 10 10
```

The same trick can be applied to row vectors, by switching the subscripts. It can also duplicate specific rows or columns of a matrix, *provided* that you are not selecting the first row or column, or that the resulting matrix is not the same size as the original matrix. If these conditions both hold, then there is an ambiguity. The subscripting vector, which is all ones, could represent *logical subscripting* where all columns or rows are chosen once, or it could represent *indexed subscripting* where just the first column or row is chosen several times. In the case of ambiguity, MATLAB chooses to view the intent as *logical subscripting.* For example,

```
» A = magic(3)
A =

  8 1 6
  3 5 7
  4 9 2

» A(:,ones(1,3))
ans =

  8 8 8
  3 3 3
  4 4 4

» A(:,ones(1,2))
ans =

  8 8
  3 3
  4 4
```

There are also two MATLAB functions which can help create special matrices. If the desired matrix has a constant diagonal or anti-diagonal, you can sometimes use the toeplitz or hankel function to create your matrix. Help sections are shown below:

```
TOEPLITZ Toeplitz matrix.

TOEPLITZ(C,R) is a non-symmetric Toeplitz matrix
having C as its first column and R as its first row.
TOEPLITZ(C) is a symmetric (or Hermitian) Toeplitz
```

```
matrix.

HANKEL Hankel matrix.

HANKEL(C) is a square Hankel matrix whose first column
is C and whose elements are zero below the first anti-
diagonal. HANKEL(C,R) is a Hankel matrix whose first
column is C and whose last row is R. Hankel matrices
are symmetric, constant across the antidiagonals, and
have elements H(i,j) = R(i+j-1).
```

For example, to create a matrix A, where each column is a rotation of the vector v above:

```
» v = (1:5)';
» M = toeplitz(v, v([1,length(v):-1:2]))
M =

  1 5 4 3 2
  2 1 5 4 3
  3 2 1 5 4
  4 3 2 1 5
  5 4 3 2 1

» M = hankel(v, v([length(v), 1:length(v)-1]))
M =

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

The advantage of being able to construct matrices efficiently is that they can then be used in further processing. Let's take a look at this aspect next.

5. Matrix Functions of Two Vectors: y(i,j) = f(x1(i), x2(j))

Suppose we want to evaluate a function F of two variables:

$$F = x*\exp(-x^2 - y^2)$$

We want to evaluate the function at every point in vector x, and for each point in x, at every point in vector y. In other words, we want to define a grid of values given vectors x and y.

We can duplicate x and y to create an output vector of the desired size using Tony's Trick. This allows us to use the techniques from section 1 to actually compute the function.

```
» x = (-2:.2:2);
» y = (-1.5:.2:1.5)';
» X = x(ones(size(y)),:);
» Y = y(:,ones(size(x)));
» F = X.*exp(-X.^2-Y.^2);
```

You may notice that this is exactly the function examined in the help entry for the meshgrid function. The meshgrid function can be used as a shorthand for the above calculation (meshgrid just uses Tony's Trick internally). This can save some typing and make the code more compact:

```
» [X,Y] = meshgrid(-2:.2:2, -1.5:.2:1.5);
» F = X .* exp(-X.^2 - Y.^2);
```

In some cases, you can use the matrix multiplication operator in order to avoid creating the intermediate matrices. For example, if

```
F = x*y
```

you can simply take the outer product of the vectors x and y:

```
» x = (-2:2);
» y = (-1.5:.5:1.5);
» x'*y
ans =

  3.0000 2.0000 1.0000 0 -1.0000 -2.0000 -3.0000
  1.5000 1.0000 0.5000 0 -0.5000 -1.0000 -1.5000
  0      0      0      0  0       0       0
  -1.5000 -1.0000 -0.5000 0 0.5000 1.0000 1.5000
  -3.0000 -2.0000 -1.0000 0 1.0000 2.0000 3.0000
```

There are also cases where sparse matrices allow more efficient use of storage space, and also allow very efficient algorithms. We discuss this in more detail in section 6.

6. Ordering, Set, and Counting Operations

Thus far, any calculations done on one element of a vector have been pretty much independent of other elements in the same vector. However, in many applications, the calculation you are trying to do depends heavily on these other values. For example, suppose you are working with a vector x which represents a set. You do not know, without looking at the rest of the vector, whether a particular element is redundant and should be removed. It is not obvious at first how to remove these redundant values without resorting to loops.

This area of vectorization requires a fair amount of ingenuity, and can be a lot of fun. There are a number of functions available as tools:

max	Largest component.
min	Smallest component.
sort	Sort in ascending order.
diff	Difference operator: diff(X), for a vector X, is: [X(2)-X(1) X(3)-X(2) ... X(n)-X(n-1)].
find	Find indices of the non-zero, non-NaN elements.

Let's proceed with our example, eliminating redundant elements of a vector. Note that, once a vector is sorted, any redundant elements are adjacent. Further, any equal adjacent elements in a vector create a zero entry in the diff of that vector.

This suggests the following implementation for the operation. We are attempting to select the elements of the sorted vector, which correspond to non-zero differences.

```
% First try. NOT QUITE RIGHT!!
» x = sort(x(:));
» difference = diff(x);
» x = x(difference~=0);
```

This is almost correct, but we have forgotten to take into account the fact that the diff function returns a vector that has one fewer element than the input vector. In our first try algorithm, the last unique element is not accounted for. We can straighten this out by adding one element to the vector x before taking the difference. We need to make sure that the element we add is always different than the previous last element. One way to do this is to add a NaN.

```
% Final version.
» x = sort(x(:));
» difference = diff([x;NaN]);
» y = x(difference~=0);
```

Two quick notes on the above algorithm. First, note that we use the (:) operation to ensure that x is a vector. Second, you may be wondering why we didn't use the find function, rather than ~=0 -- especially since this function is specifically mentioned above. This is because the find function does not return indices for NaN elements, and the last element of difference as we defined it is a NaN.

Now, suppose that we do not just want to return the set x; we want to know how many occurrences of each element in the set occurred in the original matrix. Once we have the sorted x, we can use find to determine the indices where the distribution changes. The difference between subsequent indices will be the number of occurrences for that element. This is the "diff of find of diff" trick. Building on the above, we have:

```
% Find the redundancy in a vector x
» x = sort(x(:));
» difference = diff([x;max(x)+1]);
» count = diff(find([1;difference]));
» y = x(find(difference));
» plot(y,count)
```

This plots the number of occurrences of each element of x. Note that we avoided using NaN here, because find does not return indices for NaN elements. However, the number of occurrences of NaNs and Infs are easily computed as special cases:

```
» count_nans = sum(isnan(x(:)));
» count_infs = sum(isinf(x(:)));
```

7. Sparse Matrix Structures

Finally, in some cases you can use sparse matrices to increase efficiency. Often, vectorization is easier if you construct a large intermediate matrix. In some cases you can take advantage of a sparse matrix structure to vectorize code without requiring large amounts of storage space for this intermediate matrix. Let's proceed with an example.

Suppose that, in the last example, you know beforehand that the domain of the set y is a subset of the integers, {k+1, k+2, ..., k+n}; that is,

```
y = 1:n + k
```

For example, these might represent indices into a colormap. You can take advantage of this fact in counting the occurrences of each element of the set. This is an alternative method to the 'diff of find of diff' trick from the last section.

Let's construct a large m by n matrix A, where m is the number of elements in the original vector x, and n is the number of elements in the set y.

```
A(i,j) = 1 if x(i) = y(j)
0 otherwise
```

Looking back at sections 3 and 4, you might suspect that we need to construct matrices from x and y using Tony's Trick. This would work, but it would cost a lot of storage space. We can do better by exploiting the fact that most of the matrix A consists of 0's, with only one 1 value per element of x.

Here is how to construct the matrix (note that y(j) = k+j, from the above formula):

```
» x = sort(x(:));
» A = sparse(1:length(x), x(:)+k, 1, length(x), n);
```

Now we can perform a sum over the columns of A to get the number of occurrences.

```
» count = sum(A);
```

Note that in this case we do not have to form the sorted vector y explicitly, since we know beforehand that y = 1:n + k.

The key here is to use the data, i.e. x, to control the structure of the matrix A. The fact that x takes on integer values in a known range allows efficient construction of this matrix.

As a final example, suppose you want to multiply each column of a very large matrix by the same vector. There is a way to do this using sparse matrices that saves space and can also be faster than matrix construction using Tony's Trick. Here's how it works:

```
» F = rand(1024,1024);
» x = rand(1024,1);
% Point-wise multiplication of each row of F.
» Y = F * diag(sparse(x));
% Point-wise multiplication of each column of F.
» Y = diag(sparse(x)) * F;
```

We take advantage of the matrix multiplication operator to do the bulk of the work, and use sparse matrices to prevent the temporary variable from being too large.

Conclusion

The above discussion is by no means an exhaustive list of all the useful tricks available in MATLAB. For example, if you were doing 2-D geometric calculations, it might be useful to represent each point in space as a complex pair, with the real part corresponding to the x-axis value, and the imaginary part the y-axis value. As you become experienced, you will no doubt accumulate your own set of programming idioms. Hopefully, this technical note has helped you down that road.

Is the level of technical detail appropriate? Yes ○ Too Much ○ Not Enough ○

What did you expect to find on this page, that you want us to consider adding?

Additional Comments:

SUBMIT

related topics:

Demos | Search | Contact Support | Consulting | News & Notes | Usability