Drea's Desk: Vectorization Tips

Ask any crusty MATLAB programmer how to speed up your code and they'll tell you "Vectorize!". OK, you say. How? This is a hard question to answer generally because:

- There are different techniques for different problems.
- There are different techniques for the same problem.
- Different techniques are better or worse depending on the matrix size.
- There is no end to the clever and obscure ways to vectorize in MATLAB.

One nice thing about MATLAB is that you can program in an entirely scalar way and your program will work fine, it just isn't optimal. It is like programming in C without using pointers (directly). You can do it, but you'll probably end up with sub-optimal code.

There is an enjoyable puzzle aspect to optimizing M-code. I've known people who spend hours devising intricate indexing tricks to avoid a for loop. The tricks are so complex the result is even slower than having the for loop. Still, you get a certain satisfaction from the exercise. If you want to cause a flurry on comp.soft-sys.matlab, just ask a question like "how do I vectorize this?". There is a group of dedicated "speed freaks" among MATLAB users. All of the winners of the contests in this digest probably qualify.

This Desk is primarily targeted toward MATLAB users that have written between 10 and 100 M-files, but I hope more experienced MATLAB programmers will also find a useful tip or two.

I've identified three generic programming problems and will describe ways to vectorize them.

1. Find a subset of a matrix that satisfies a condition, and do something to it.

2. Look for patterns in a vector and do something to them.

3. Do something different to each column of a matrix without a loop.

----

1) Find a subset of a matrix that satisfies a condition, and do something to it.

Say you have a matrix,

```
  M = magic(3)
  M =
      8     1     6
      3     5     7
      4     9     2
```

and you want to negate every element greater than 4. The
way you'd do this in conventional (scalar) languages is,

```
for i=1:3,
  for j=1:3,
    if (M(i,j) > 4),
      M(i,j) = -M(i,j);
    end
  end
end
```

In MATLAB,

```
ind = find(M > 4);
M(ind)=-M(ind);
```

MATLAB has a FIND command that is extensively used. It returns
the indices of non-zero elements of a matrix, and it is FAST.
I timed the above two methods on a 300x300 matrix and the results
were,

```
for loop method: elapsed_time = 23.4956
FIND method:     elapsed_time =  2.1153
```

A factor of 10 increase in speed! This is how the FIND method works:

```
ind = find(M > 4)

  ind =
       1
       5
       6
       7
       8
```

(M > 4) produces a matrix of 1's and 0's; 1 if the inequality is
true for that element and 0 if it isn't. FIND then returns the
indices of all the 1's.

It might seem a little odd that the indices go up to 8 given that

the matrix is 3x3. This is because MATLAB lets you index into
matrices in two ways, M(i,j) or M(k). When you use only one index,
MATLAB effectively strings out the matrix into a vector, column
by column, and then indexes into that vector.

  ----
2) Look for patterns in a vector and do something to them.

A common thing you might want to do is remove extra whitespace
from a string. For instance,

```
V = 'I  really    hate    extra white    space'
```

A conventional algorithm would look something like,

```
  len = length(V);
  i=1;
  while (i<len),
    if (V(i) == ' ' & V(i+1) == ' '),
      for j = i:(len-1),
        V(j) = V(j+1);
      end
      V(len)=0;       % Or something that will shorten the vector
      len=len-1;
    else,
      i=i+1;
    end
  end
  V = setstr(V);
```

In MATLAB,

```
  ind = find(filter([1 1],2,V==' ')==1);
  V(ind)= [];
```

or,

```
  ind = findstr(V,'  ');
  V(ind)= [];
```

Every MATLAB programmer has his or her favorite built-in command.
Mine is FILTER, so I'll talk about the first method.

(V==' ') produces a vector of 1's and 0's. FILTER makes
a window two elements wide and slides it down the vector. It
multiplies each element in the window by a weighting factor

(in our case [1 1]), sums them, then divides the result by another
factor (in our case the constant 2). Thus, filter([1 1],2,X)
produces a running average of two points of the vector X. When filter
encounters two 1's in a row (indicating multiple spaces), the average
value is 1, and the FIND gets the index of that location.

The syntax V(ind) = []; is a deletion operation. It removes all
of the selected elements from the vector V and shrinks the matrix
accordingly. This is very useful because removing several elements
from a vector and copying the remaining elements to the right place
is moderately tricky. This operation can also be used to delete
entire rows or columns from a matrix. And, as you might have guessed,
this is faster than other ways of reducing a matrix.

The FINDSTR (find string) method is elegant, but there is
a for loop hidden in there and in this case it is about half as fast.

  ----
3) Do something different to each column of a matrix without a loop.

In this case, let's remove the mean from each column of the matrix
M,

```
  M = rand(10,5);
  V = mean(M);
```

A partially vectorized solution would look like,

```
  for i=1:5,
    M(i,:)=M(i,:)-V(i);
  end
```

A better solution is,

```
  M=M-V(ones(10,1),:);
```

That is, V is turned into a matrix the size of M and the matrices
are subtracted in one operation. There is one subtlety. It makes
use of "Tony's Trick",

```
  a = [17 29 31];
  V = a([1 1 1],:)

    V =
        17    29    31
        17    29    31
    17    29    31
```

The astute reader might notice that you could have accomplished the same result with,

```
  V = a*ones(1,3);
```

So, why use Tony's trick? It is usually faster. The second method performs unnecessary matrix multiplication. This technique is most useful when you have large matrices and the operation is inside your inner-most loop (where speed is critical). Let's test this hypothesis.

```
  a = rand(1,100);
  I = ones(100,1);

  tic;V=I*a;toc

  elapsed_time = 0.0315

  tic;V=a(I,:);toc

  elapsed_time = 0.0115
```

In this case, the indexing method is more than twice as fast as the matrix multiply.

One other little tidbit: Avoid using []'s when they aren't necessary. MATLAB has to create temporary storage for the results of what it believes to be concatenation.

```
  tic;for i=1:1000,a=[17];end;toc

  elapsed_time = 0.1987

  tic;for i=1:1000,a=17;end;toc

  elapsed_time = 0.0435
```

I hope some of these these tips have been helpful to many of you. If not, you probably stopped reading before here anyway :-).

If you have a topic that you would like discussed on Drea's Desk, let me know. Just send it to digest@mathworks.com.

Drea Thomas
drea@mathworks.com
