Vectorization

Using Vectorization

MATLAB® is optimized for operations involving matrices and vectors. The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations is called *vectorization*. Vectorizing your code is worthwhile for several reasons:

- *Appearance*: Vectorized mathematical code appears more like the mathematical expressions found in textbooks, making the code easier to understand.
- *Less Error Prone*: Without loops, vectorized code is often shorter. Fewer lines of code mean fewer opportunities to introduce programming errors.
- *Performance*: Vectorized code often runs much faster than the corresponding code containing loops.

Vectorizing Code for General Computing

This code computes the sine of 1,001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

This is a vectorized version of the same code:

```
t = 0:.01:10;
y = sin(t);
```

The second code sample usually executes faster than the first and is a more efficient use of MATLAB. Test execution speed on your system by creating scripts that contain the code shown, and then use the `tic` and `toc` functions to measure their execution time.

Vectorizing Code for Specific Tasks

This code computes the cumulative sum of a vector at every fifth element:

```
x = 1:10000;
ylength = (length(x) - mod(length(x),5))/5;
y(1:ylength) = 0;
for n= 5:5:length(x)
    y(n/5) = sum(x(1:n));
end
```

Using vectorization, you can write a much more concise MATLAB process. This code shows one way to accomplish the task:

```
x = 1:10000;
xsums = cumsum(x);
y = xsums(5:5:length(x));
```

Array Operations

Array operators perform the same operation for all elements in the data set. These types of operations are useful for repetitive calculations. For example, suppose you collect the volume (`V`) of various cones by recording their diameter (`D`) and height (`H`). If you collect the information for just one cone, you can calculate the volume for that single cone:

```
V = 1/12*pi*(D^2)*H;
```

Now, collect information on 10,000 cones. The vectors `D` and `H` each contain 10,000 elements, and you want to calculate 10,000 volumes. In most programming languages, you need to set up a loop similar to this MATLAB code:

```
for n = 1:10000
   V(n) = 1/12*pi*(D(n)^2)*H(n));
end
```

With MATLAB, you can perform the calculation for each element of a vector with similar syntax as the scalar case:

```
% Vectorized Calculation
V = 1/12*pi*(D.^2).*H;
```

Note: Placing a period (`.`) before the operators *, /, and ^, transforms them into array operators.

Logical Array Operations

A logical extension of the bulk processing of arrays is to vectorize comparisons and decision making. MATLAB comparison operators accept vector inputs and return vector outputs.

For example, suppose while collecting data from 10,000 cones, you record several negative values for the diameter. You can determine which values in a vector are valid with the >= operator:

```
D = [-0.2 1.0 1.5 3.0 -1.0 4.2 3.14];
D >= 0
```

```
ans =

     0     1     1     1     0     1     1
```

You can directly exploit the logical indexing power of MATLAB to select the valid cone volumes, `Vgood`, for which the corresponding elements of `D` are nonnegative:

```
Vgood = V(D >= 0);
```

MATLAB allows you to perform a logical AND or OR on the elements of an entire vector with the functions `all` and `any`, respectively. You can throw a warning if all values of `D` are below zero:

```
if all(D < 0)
   warning('All values of diameter are negative.')
   return
end
```

MATLAB can compare two vectors of the same size, allowing you to impose further restrictions. This code finds all the values where V is nonnegative and `D` is greater than `H`:

```
V((V >= 0) & (D > H))
```

The resulting vector is the same size as the inputs.

To aid comparison, MATLAB contains special values to denote overflow, underflow, and undefined operators, such as `inf` and `nan`. Logical operators `isinf` and `isnan` exist to help perform logical tests for these special values. For example, it is often useful to exclude `NaN` values from computations:

```
x = [2 -1 0 3 NaN 2 NaN 11 4 Inf];
xvalid = x(~isnan(x))
```

```
xvalid =

     2    -1     0     3     2    11     4   Inf
```

Note: `Inf == Inf` returns true; however, `NaN == NaN` always returns false.

Matrix Operations

Matrix operations act according to the rules of linear algebra. These operations are most useful in vectorization if you are working with multidimensional data.

Suppose you want to evaluate a function, `F`, of two variables, `x` and `y`.

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

To evaluate this function at every combination of points in the `x` and `y`, you need to define a grid of values:

```
x = -2:0.2:2;
y = -1.5:0.2:1.5;
[X,Y] = meshgrid(x,y);
F = X.*exp(-X.^2-Y.^2);
```

Without `meshgrid`, you might need to write two `for` loops to iterate through vector combinations. The function `ndgrid` also creates number grids from vectors, but can construct grids beyond three dimensions. `meshgrid` can only construct 2-D and 3-D grids.

In some cases, using matrix multiplication eliminates intermediate steps needed to create number grids:

```
x = -2:2;
y = -1:0.5:1;
x'*y
```

```
ans =

    2.0000    1.0000         0   -1.0000   -2.0000
    1.0000    0.5000         0   -0.5000   -1.0000
         0         0         0         0         0
   -1.0000   -0.5000         0    0.5000    1.0000
   -2.0000   -1.0000         0    1.0000    2.0000
```

Constructing Matrices

When vectorizing code, you often need to construct a matrix with a particular size or structure. Techniques exist for creating uniform matrices. For instance, you might need a 5-by-5 matrix of equal elements:

```
A = ones(5,5)*10;
```

Or, you might need a matrix of repeating values:

```
v = 1:5;
A = repmat(v,3,1)
```

```
A =

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

The function `repmat` possesses flexibility in building matrices from smaller matrices or vectors. `repmat` creates matrices by repeating an input matrix:

```
A = repmat(1:3,5,2)
B = repmat([1 2; 3 4],2,2)
```

```
A =

     1     2     3     1     2     3
     1     2     3     1     2     3
     1     2     3     1     2     3
     1     2     3     1     2     3
     1     2     3     1     2     3


B =

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

The `bsxfun` function provides a way of combining matrices of different dimensions. Suppose that matrix `A` represents test scores, the rows of which denote different classes. You want to calculate the difference between the average score and individual scores for each class. Your first thought might be to compute the simple difference, `A - mean(A)`. However, MATLAB throws an error if you try this code because the matrices are not the same size. Instead, `bsxfun` performs the operation without explicitly reconstructing the input matrices so that they are the same size.

```
A = [97 89 84; 95 82 92; 64 80 99;76 77 67;...
 88 59 74; 78 66 87; 55 93 85];
dev = bsxfun(@minus,A,mean(A))
```

```
dev =

    18    11     0
    16     4     8
   -15     2    15
    -3    -1   -17
     9   -19   -10
    -1   -12     3
   -24    15     1
```

Ordering, Setting, and Counting Operations

In many applications, calculations done on an element of a vector depend on other elements in the same vector. For example, a vector, *x*, might represent a set. How to iterate through a set without a `for` or `while` loop is not obvious. The process becomes much clearer and the syntax less cumbersome when you use vectorized code.

Eliminating Redundant Elements

A number of different ways exist for finding the redundant elements of a vector. One way involves the function `diff`. After sorting the vector elements, equal adjacent elements produce a zero entry when you use the `diff` function on that vector. Because `diff(x)` produces a vector that has one fewer element than `x`, you must add an element that is not equal to any other element in the set. `NaN` always satisfies this condition. Finally, you can use logical indexing to choose the unique elements in the set:

```
x = [2 1 2 2 3 1 3 2 1 3];
x = sort(x);
difference  = diff([x,NaN]);
y = x(difference~=0)
```

```
y =

     1     2     3
```

Alternatively, you could accomplish the same operation by using the `unique` function:

```
y=unique(x);
```

However, the `unique` function might provide more functionality than is needed and slow down the execution of your code. Use the `tic` and `toc` functions if you want to measure the performance of each code snippet.

Counting Elements in a Vector

Rather than merely returning the set, or subset, of `x`, you can count the occurrences of an element in a vector. After the vector sorts, you can use the `find` function to determine the indices of zero values in `diff(x)` and to show where the elements change value. The difference between subsequent indices from the `find` function indicates the number of occurrences for a particular element:

```
x = [2 1 2 2 3 1 3 2 1 3];
x = sort(x);
difference  = diff([x,max(x)+1]);
count = diff(find([1,difference]))
y = x(find(difference))
```

```
count =

     3     4     3


y =

     1     2     3
```

The `find` function does not return indices for `NaN` elements. You can count the number of `NaN` and `Inf` values using the `isnan` and `isinf` functions.

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

Functions Commonly Used in Vectorization

Function	Description
`all`	Determine if all array elements are nonzero or true
`any`	Determine if any array elements are nonzero
`cumsum`	Cumulative sum
`diff`	Differences and Approximate Derivatives
`find`	Find indices and values of nonzero elements
`ind2sub`	Subscripts from linear index
`ipermute`	Inverse permute dimensions of N-D array
`logical`	Convert numeric values to logicals

meshgrid	Rectangular grid in 2-D and 3-D space
ndgrid	Rectangular grid in N-D space
permute	Rearrange dimensions of N-D array
prod	Product of array elements
repmat	Repeat copies of array
reshape	Reshape array
shiftdim	Shift dimensions
sort	Sort array elements
squeeze	Remove singleton dimensions
sub2ind	Convert subscripts to linear indices
sum	Sum of array elements

More About

- Matrix Indexing
- Techniques to Improve Performance

External Websites

- MathWorks Newsletter: Matrix Indexing in MATLAB