AEROSPACE ENGINEERING

UNIVERSITY of MICHIGAN ■ COLLEGE of ENGINEERING

Vectorization in MATLAB

And other minor tips and tricks

Derek J. Dalle

University of Michigan, Ann Arbor, MI 48109

March 28, 2012

Introduction

For most of the examples, you will have to open MATLAB and enter in the sample code to see what the results will be.

Vectorization

The main reason for this tutorial is to explain the basics of using vectorizing syntax in MATLAB. In particular there are some examples of things that are particularly hard to find in help files and on the internet.

Variable types

More importantly, the slides lay out a basic strategy for eliminating unnecessary loops from your code. Sometimes you may want to keep unnecessary loops in your code to make the code easier to read or to make easier to generalize at a later time, but in many, many cases, removing a loop would make the code better. These slides will introduce several techniques that can be used to vectorize trickier sections of code.

Creating vectors
Direct input

Simple cases

This is the easiest way to create an array.

```
A = [1, 3; 2, 4]
```

Commas are not required.

```
A = [1 3; 2 4]
```

But be careful with operators!

```
A = [1 -2]
```

is very different from

```
A = [1 - 2]
```

This is a rare case where spaces matter.

Appending arrays

Add elements to vectors.

```
x = [1 2 3];
x = [x, 4]
```

Combine arrays.

```
A = [1, 2];
B = [4, -1; 2, 5];
C = [A; B];
```

MATLAB will check for compatible sizes.

Indexing new entries

Avoid doing this!

```
x = 1; x(2) = 3
```

Initializing large vectors

- Creating large vectors by appending is very inefficient
- Very difficult to make a higher-dimension array

Initialization

The commands `ones`, `zeros`, and `nan` are particularly useful for creating vectors.

```
A = zeros(20, 10)
```

This makes a 20-by-10 matrix with zero in all entries. The following makes a row vector in which all entries are 4.

```
x = 4 * ones(1, 10)
```

For some reason, using only one input gives you a square matrix.

```
>> A = zeros(2)
A =
     0     0
     0     0
```

Multidimensional arrays

This is the preferred way to make an array with more than two dimensions.

```
A = zeros(3, 2, 4)
```

This creates a 3-by-2-by-4 matrix, which is basically 4 3-by-2 matrices stacked in pages.

Getting the dimensions

For the array defined above, `size(A)` will return `[3 2 4]`, and `numel(A)` will return `24`. The popular command `length` will return `4`. Avoid using the `length` command!

Other initialization functions

Not a number: `NaN`

You can initialize a matrix of `NaN`s.

```
A = nan(20, 20)
```

A `NaN` can be used as more than just an error placeholder. Usually they come from a calculation like `0/0`, but they have other uses. If you have a vector whose size isn't known in advance, it's a good way to keep track of how many of the values have been used.

Random numbers

The `rand` function can also make arbitrary-size arrays.

```
A = rand(20, 15)
```

This creates a 20-by-15 matrix with pseudo-random numbers between 0 and 1.

Diagonal matrices

Also check how to use the commands `eye` and `diag`.

```
I = eye(4)
```

This creates a 4-by-4 identity matrix.

Vectorization
MATLAB code performance

This is the key to writing fast code in MATLAB.

Compare two versions that calculate the following formula for each element of two $m \times n$ matrices

$$z = x^2 \sin y$$

Bad version

```
for i = 1:m
  for j = 1:n
    z(i,j) = x(i,j)^2*sin(y(i,j))
  end
end
```

Good version

```
z = x.^2 .* sin(y)
```

Built-in functions
Purely numeric functions

- Most MATLAB functions, like `sin`, `cos`, `log`, and many others, work regardless of the size of the input.
- If `A` is a numeric array of any size, then `B=sin(A)` is another array such that `B(i,j) == sin(A(i,j))`

Example

This creates a 20-by-4-by-2 3D array called `A` and then creates another array with the same dimensions using the exponential function.

```
A = rand(20, 4, 2);
B = exp(A)
```

Most numeric functions have this behavior. The exceptions are matrix operators and functions like `max` and `sum` which obviously output a smaller array than the input.

Built-in functions
Collapsing functions

- Some function act on a vector and output a scalar.
- When given an input with more than one non-singleton dimension MATLAB only collapses one dimension at a time.

Vector behavior

Works on either a row or column vector.

```
x = [1, 4, 8, -1, 2];
max(x) == 8
y = [2; 7; -2; -3; 2];
max(y) == 7
```

Array behavior

The function collapses the first dimension that is larger than 1.

```
A = [1, 3, 2; -4, 5, -1];
max(A) == [1  5  2]
```

Collapsing arrays

Suppose you have an array like

```
A = rand(3, 3, 3)
```

and you want to add up all of the 27 entries. You can use `sum(sum(sum(A)))`, but the command changes based on the size of `A`. A better option is to convert `A` to a column vector and use `sum` once.

```
sum(A(:))
```

The syntax `A(:)` converts any array into a column vector.

Operators
Entry-by-entry arithmetic

- Some basic operators, like `*` and `^`, do not operate on elements.
- They default to matrix multiplication and other matrix math.
- Use operators like `.*` and `.^` instead.
- The operators `+` and `-` are already element-by-element.
- It's best to use the dots unless you explicitly want matrix math.

Matrix multiplication

```
A = [1, 2; 0, 4];
B = [2, -1; 3, 4];
A * B == [8, 7; 12, 16]
```

Entry-wise multiplication

```
A = [1, 2; 0, 4];
B = [2, -1; 3, 4];
A .* B == [2, -2; 0, 16]
```

- Note that some combinations of matrix size will work with one type of operator and not the other.
- Matrix math doesn't work at all with multidimensional arrays.
- It's always possible to combine scalars with arrays, for example `4 + A`

Manipulation of numeric arrays

Here we discuss several topics critical to using arrays effectively. There are a host of issues that come up when vectorizing a code, and any one of them can force you to write a loop. This section gives you an idea of what kind of hurdles can be overcome.

- Comparing and testing arrays
- Reshaping and repeating arrays
- Logical indexing and searching
- Vectors of vectors
- Advanced examples

Comparing arrays
Equality of arbitrary arrays

Suppose you have two variables `A` and `B`. You suspect that they might be equal, but you aren't even sure if both of them are numeric. The normal thing to try would be simply `A == B`, but this has multiple problems. If both `A` and `B` are not scalars but have different sizes, this will throw an error. Even if `A` and `B` are the same array, this will create an array equivalent to `true(size(A))` rather than a single `true` value.

Example

Initialize two arrays that are equal.

```
A = rand(4,4,3); B = A;
```

Try this and see what you get.

```
A == B
```

Not exactly what we wanted... Now try this example.

```
A = rand(4); B = rand(2,3);
A == B
```

Comparing arrays
Equality of arbitrary arrays *(continued)*

The full solution is, unfortunately, more complicated.

Create two arrays.

```
A = rand(4,5,2); B = A;
```

Get the sizes of the arrays to make sure they can be compared.

```
s_A = size(A); s_B = size(B);
```

Before we can compare `s_A` and `s_B` directly, we also need to know that *they* have the same size, but at least we know that they are both row vectors. So now we can test if `A` and `B` are the same using a long test.

```
q = (numel(s_A) == numel(s_B)) && all(s_A == s_B) && all(A(:) == B(:))
```

Way easier way...

MATLAB has a command to do all of these checks for you.

```
q = isequal(A, B)
```

Comparing arrays
Comparing real numbers

Two ways to compare

Array with a scalar

This method compares each entry in the array with the scalar individually.

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

Array with an array

This method compares each entry with the corresponding entry from the other array

```
[1, 3, 4, -1] > [-1, 4, 3, 0]
ans =
     1   0   1   0
```

Be careful with ==

Close but not equal

Try the following.

```
sin(380*pi/180) == sind(380)
```

Clearly these should be equal, but MATLAB is saying that they are not. The problem comes from roundoff error when computations are done in different orders. The following is a more reliable test for "equality" of real numbers.

```
abs(x - y) <= tol
```

where `tol` is a small number like `1e-14`.

Combining tests

We've seen how comparing two arrays with the same dimensions creates a third array of ones and zeros with the same dimensions. We can also use this technique to perform multiple tests.

Let's create three arrays.

```
A = [1, 3, 4, -1];
B = [-1, 4, 3, 0];
C = rand(1, 4);
```

The following tests which entries of `B` are strictly between the corresponding entries of `A` and `C`.

```
A < B & B < C
```

Note the use of a single `&`. With scalars, use `&&` instead. The following example finds which entries of `C` are greater than the corresponding entry of `B` or `A` (or both).

```
C > B | C > A
```

Example: testing bounds

Let's do an example in which we have to find which entries of an array `A` are between a lower bound `a` and `b`.

```
A = rand(28, 8);
a = 0.4; b = 0.6;
```

the entries of `A` greater than or equal to `a` are given by

```
A >= a
```

Similarly, we have `A <= b` for the other test. So the correct solution is

```
i = A >= a & A <= b
```

We'll return to this example later.

Logical indexing
Extracting elements

The story of this slide is that the `find` command is often not needed.

Suppose we have the following array.

```
A = [-4, 2, 0; 1, -2, 3];
```

The goal is to extract the positive entries of `A`. The following test kind of locates them.

```
i = A > 0
i =
     0     1     0
     1     0     1
```

We can do `A(find(A>0))`, but the use of `find` is unnecessary.

```
A(A>0)'
ans =
     2     1     3
```

The jist of it...

When you use a logical array (that is, an array that results from a test) as an index, MATLAB applies whatever command you do only to the entries where the logical array has a `1`.

General description

If you use a command `A(i)` to reference parts of an array `A`, the variable `i` can be one of two types. The first is an array of integers like `i=[1 3 5 6]`, but the other way is a logical array like `i=[1 0 1 0 1 1]`.

Logical indexing
Assigning entries

We can also use the syntax `A(i) = B` to change the values of only part of `A`. However, `B` must be either a single value or have the same dimensions as `A(i)`.

Example: testing bounds

Let's return to a previous example

```
A = rand(28, 8);
a = 0.4; b = 0.6;
```

Now we want to make the entries of `A` all be between `a` and `b`.

```
A(A<a) = a;
A(A>b) = b;
```

In this case there is another method.

```
A = max(a, min(b, A));
```

Let's create two random arrays.

```
A = rand(6, 4);
B = rand(6, 4);
```

This command makes the entries of `A` negative for each element where `B` is greater than `A`.

```
A(B>A) = -A(B>A);
```

This adds the corresponding entry of `B` whenever `A` is less than one half.

```
i = A < 0.5;
A(i) = A(i) + B(i);
```

Logical indexing
Conclusions and reminders

Using logical indexing when possible is much faster than `find`. The reason for this is that `find` is more powerful, and it wastes time to use that power when it's not needed!

Using logical indexes is a little counterintuitive at first, but if you are writing a function in the MATLAB editor, it will often help you out. If you use `find` in an unnecessary way, MATLAB will often suggest a small change to your code that's faster. Listening to this is how I learned logical indexing!

As a basic review, logical indexing works in the following way. With the syntax `A(i)`, it ignores any entry of `A` where the corresponding entry of `i` is `0`.

Initializing arrays
Cleanup of unkown-size vectors

Sometimes we end up making arrays whose dimensions cannot be known beforehand. In these cases, it is much better to initialize a larger array than needed and then trim off the extra entries later.

Example

Create a vector that has `i_1` entries of `1`, `i_2` entries of 2, ... , `i_n` entries of `n` where each `i` value is a random integer between `1` and `m`.

Easier way

Initialize an empty vector.

```
m = 5; n = 4;
x = [];
```

Now loop through, appending to `x` each time.

```
for i = 1:n
    m_i = randi(m);
    x = [x, i*ones(1, m_i)];
end
```

Better

We know the maximum size `x` can have.

```
m = 5; n = 4; M = randi(m, 1, n);
x = nan(1, m*n);
```

Now make entries as we go. The last command deletes the extra entries of `x`.

```
j = 0;
for i = 1:n
    x(j+(1:M(i))) = i;
    j = j + M(i);
end
x(isnan(x)) = [];
```

Basic extraction
Slicing

Extracting a column

Suppose `A = eye(7)`. Then `A(:,4)` is the 4th column, often denoted $\hat{\mathbf{e}}_4$.

"Indices" and "subscripts"

Suppose `A = rand(3)`. Then `A(2,1)` is the same as `A(2)`, and `A(1,2)` is the same as `A(4)`. When using indices (for example `A(2)`), MATLAB goes top-to-bottom and then left-to-right. Use `ind2sub` and `sub2ind` to convert between the two types.

A 'page' of a larger array

Suppose `A = rand(3, 4, 5)`. This can be viewed as 5 'pages' of 3×4 matrices. To get the kth page, use `A(:,:,k)`.

Submatrices

If you use the example command `A([1 3], [2 4])`, the result is not

```
[A(1,2),    A(3,4)]
```

but a whole submatrix.

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

Example 1: Dot products

If we have vectors like `u = [1; 2; -1]` and `v = [3; 0; 2]`, the dot product is easy in MATLAB. We just do `u' * v`.

Challenge: vectorize

Suppose instead we have two large matrices of 3-vectors, for instance

```
U = 2*rand(3, 40) - 1;
V = 2*rand(3, 40) - 1;
```

We can do `U' * V` again, but this will actually create a 40×40 matrix in which we only want the diagonal entries.

The second solution is to do a `for` loop, but we don't want that.

Instead, we have to find a way to do the dot product somewhat manually. The following command works perfectly; it multiplies $u_{ij}v_{ij}$, and then adds up each column, giving a row vector of $u_{1j}v_{1j} + u_{2j}v_{2j} + u_{3j}v_{3j}$ for $j = 1, \ldots, 40$.

```
sum(U .* V)
```

Example 1: Dot products
(continued)

Actually we can take this method even further. Suppose we have two vectors for each point in a 3D mesh. We might have

```
U = 2*rand(20, 50, 30, 3) - 1;
V = 2*rand(20, 50, 30, 3) - 1;
```

for a $20 \times 50 \times 30$ mesh. All we have to do to calculate all the dot products is dot-multiply the arrays again and tell MATLAB which direction to add up.

```
D = sum(U .* V, 4);
```

This gives a $20 \times 50 \times 30$ array for `D` where

```
D(i,j,k) == U(i,j,k,1)*V(i,j,k,1)
     + U(i,j,k,2)*V(i,j,k,2) + U(i,j,k,3)*V(i,j,k,3)
```

Example 2: Norms

If we have a vector like `u = [1; 2; -1]`, calculating the norm is easy in MATLAB. We just do `norm(u)`.

Challenge: vectorize

Suppose instead we have a large matrix of 3-vectors, for instance

```
U = 2*rand(3, 40) - 1;
```

We cannot do `norm(U)` to calculate the norm of each column, because MATLAB will try to calculate a matrix norm.

The easy solution is to do a `for` loop, but we don't want that.

Instead, we have to find a way to calculate the norms somewhat manually. The following command works perfectly; it computes u_{ij}^2, then adds up each column, and takes the square root of the resulting sums. The result is a row vector of $\sqrt{u_{1j}^2 + u_{2j}^2 + u_{3j}^2}$ for $j = 1, \ldots, 40$.

```
L = sqrt(sum(U.^2))
```

Example 3: Dividing by a vector of scalars

If we have a vector like `u = [1; 2; -1]` and a scalar `c = 3`, dividing by the vector is easy. We just do `u / c` or `u ./ c`.

Challenge: vectorize

Suppose instead we have a large matrix of 3-vectors and a row of scaling constants, for instance

```
U = 2*rand(3, 40) - 1;
C = rand(1, 40);
```

We cannot do `U ./ C` to scale each vector because `C` does not have the same dimensions as `U`.

The easy solution is to do a `for` loop and use `U(:,i) / c(i)`.

Instead we need to make `C` bigger. The easiest way is `U ./ [C; C; C]`, but this does not generalize to longer vectors. Here are two solutions.

```
V = U ./ repmat(C, 3, 1);
V = U ./ (ones(3,1) * C);
```

In the general case, replace `3` with `size(U,1)`.

Function handles
Introduction: anonymous functions

Suppose you have a simple calculation that you have to perform several times, but you don't want to write the operation each time. A way to do this is to create an **anonymous function**.

```
f = @(x) x^2 + sin(x);
a = f(3); b = f(-1); c = f(4);
```

This allows you to make certain functions without creating a new `.m` file. But this example isn't very good because it won't work on a vector. The following version is better.

```
f = @(x) x.^2 + sin(x);
a = f([3, -1, 4]);
```

The `@` symbol tells MATLAB that an anonymous function declaration is coming. What follows is a list of input arguments between parentheses. Then a single line of code, whose value is treated as the output of the function, follows.

Uses of function handles

Quickly plotting a function or family of functions

Suppose you want to plot a few functions quickly. This can be somewhat obnoxious if it requires making a range of x-coordinates and then using a `for` loop to calculate each entry. This quickly plots the function $ax^2 + e^{-x}$ for several values of a.

```
x = linspace(0, 1, 101);
f = @(x, a) a.*x.^2 + exp(-x);
plot(x, f(x,1), x, f(x,2), x, f(x,3))
```

If you happen to forget to use `.*`, `./`, etc., the MATLAB function `vectorize` can be of use.

```
vectorize( @(x, y) x^2 * y )
ans =
      @(x,y)x.^2.*y
```

This may not always be what you want if your function actually contains a matrix operation.

Advanced function handles
Special considerations

When an anonymous function is created, any variable that is not an input to the function is evaluated. This is hard to explain, but consider this example.

```
>> a = 4;
>> f = @(x) a .* x;
>> f(3)
ans =
      12
>> a = 5; f(3)
ans =
      12
```

Changing `a` after `f` was created had no effect on `f`.

More complex functions

You can also reference functions that you have created in separate `.m` files. Suppose you have two functions `funa` and `funb`.

```
f = @(x,a,b) ...
   funa(x,a) .* funb(x,b)
```

This creates a function that combines the two two-argument functions into a single three-argument function.

Using `ode45` with parameters

How to avoid using `global`

The MATLAB solvers for ordinary differential equations require a function handle for the first input. But these handles must be for functions with two inputs and one output. Almost all of the time, though, the function depends on some sort of other parameters. Suppose your differential equation is

$$\dot{x} = x - \cos(at) \qquad x(0) = 0 \qquad 0 \le t \le \pi \tag{1}$$

so you create a function using an `.m` file with the following contents.

```
function dx = myfun(t, x, a)

dx = sin(a.*t) - x;
```

But how do you use this function with `ode45` since it has three inputs? The answer is to create a handle to `myfun` in the command.

```
a = 4;
[t, x] = ode45(@(t,x)myfun(t,x,a), [0, pi], 0);
```

The `@(t,x)` syntax creates a function of two inputs, and `a` is set to `4`.

Cell arrays
Introduction and basic usage

Cell arrays allow you to combine objects that are not of the same type into a single array-like variable. The following creates a cell array with row vectors of different sizes.

```
S = {[1, 2, 3]; [-1, 5]; 4; [2, 3, 6]};
```

They can contain all sorts of different variables.

```
S = {@(x) x.^2, 'a string'; rand(2,2), pi};
```

You can use normal subscripts to get *part of the cell array*.

```
S(1,:)
ans =
    @(x)x.^2        'a string'
```

To get *an element of the cell array*, you have to use different syntax.

```
S{1,2}
ans =
a string
```

```
S(1,2)
ans =
    'a string'
```

These are different because `S(1,2)` is still a cell array.

Cell arrays
Two little things

Cell arrays can be initialized similar to the way numeric arrays are.

```
S = cell(3, 4)
```

This creates a 3×4 cell array in which each entry is an empty matrix.

Now try to figure out what is happening when you run the following command.

```
S{2:4}
```

Cell arrays of strings

Since strings tend to not all have the same length, cell arrays are an extremely convenient way to store them. Here's an example.

```
S = {'Mercury', 'Venus', 'Earth', 'Mars', ...
    'Jupiter', 'Saturn', 'Uranus', 'Neptune'};
```

Now, to get the name of the `i`th planet, do not try `S(i)`. MATLAB is not too helpful about what is going on here, and I often see beginners trying to do crazy things like use `cellstr` to get the string. The real solution, though is that the `i`th planet is `S{i}`.

However, we can make a cell array that contains the 1st, 2nd, 4th, and 7th planets like this.

```
T = T([1, 2, 4, 7]);
```

String comparisons

strcmp and strcmpi

Let's get our list of planets again.

```
S = {'Mercury', 'Venus', 'Earth', 'Mars', ...
    'Jupiter', 'Saturn', 'Uranus', 'Neptune'};
```

Comparing strings poses a particular challenge because they don't always have the same length. Our usual trick of trying `all(S{1} == S{2})` will not work. Fortunately MATLAB has a command for this purpose.

```
strcmp('Mars', S{4})
ans =
     1
```

```
strcmp('Mars', S{1})
ans =
     0
```

Furthermore, the command `strcmpi` works almost exactly the same except that it ignores case.

```
strcmpi('mars', S{4})
ans =
     1
```

```
strcmp('mars', S{4})
ans =
     0
```

String comparisons using cell arrays

`strcmp`, `strcmpi`, `any`, and `all`

Let's get our list of planets again.

```
S = {'Mercury', 'Venus', 'Earth', 'Mars', ...
    'Jupiter', 'Saturn', 'Uranus', 'Neptune'};
```

Testing for inclusion

This tests if a string `s` matches any strings in `S`.

```
s = 'Mercury';
q = any(strcmp(s, S));
```

The case-insensitive version also works.

```
s = 'MERCURY';
q = any(strcmpi(S, s));
```

Testing for matches

Suppose we have two cell arrays that represent two different sets of values for four things. The following tests which events match in both cell arrays.

```
S={'on' ,'low'   ,'new','hi'};
T={'off','square','new','hi'};
strcmpi(S, T)
ans =
     0    0    1    1
```

Listing inputs

Cell arrays can also be used for some pretty confusing-looking commands. The first one is to make a list of inputs to some function where the list can have any length. Suppose you want to make a list of options to your `plot` command but you want to use the same list of options repeatedly.

```
opts = {'Color', [1 0.5 0.2], 'LineWidth', 1.7, 'Marker', 'x'};
plot(x1, y1, opts{:})
plot(x2, y2, opts{:})
plot(x3, y3, opts{:})
plot(x4, y4, opts{:})
```

This is the same as typing `plot(x1, x2, 'Color', ...)`, but more convenient if you want to change something later.

The syntax `opts{:}` evaluates each element of the cell array in sequence, which creates the same effect as a list of inputs. Syntax like `opts{3:end}` is also allowed and works the same way.

Arbitray outputs

Possibly the most confusing syntax in MATLAB

This is kind of the converse of the previous slide. Suppose you have a function that normally gives multiple outputs, but you'd like to put them into one cell array. It's kind of hard to come up with a good example where you'd want this, but it has happened. This example illustrates how to get each of the dimensions of an array into a cell array.

```
S = cell(1,ndims(A));
[S{:}] = size(A);
```

For this to work, the cell array on the left-hand side of the assignment must have the correct dimensions, which is the reason that `S` is initialized using `cell` first.

There aren't really any built-in functions for which this is useful, but it is possible to make functions where you would want this.

You should learn about `varargin` and `varargout` before you consider this too much.

Structs
Fields and values

`struct`s are a very powerful way to store information with names for each piece of information. A `struct` contains several 'fields', which each contain a value.

```
s.a = 14; s.b = ones(1,4); s.c = 'Earth'
s =
    a: 14
    b: [1 1 1 1]
    c: 'Earth'
```

It's possible to make a `struct` with multiple fields using a special command.

```
s = struct('a', 14, 'b', ones(1,4), 'c', 'Earth');
```

Extracting information from a struct is very simple.

```
s.b + 2
ans =
    3    3    3    3
```

Each field can also be a `struct` if you want.

```
s.d.name = 'umich'; s.d.type = 'awesome';
```

Structs
Advanced referencing

A useful command is `fieldnames`, which takes a `struct` as input and returns a cell array of strings. Each string represents the name of a field of the struct.

```
s = struct('a', 14, 'b', 'New York', 'c', eye(2));
fieldnames(s)
ans =
    'a'
    'b'
    'c'
```

The complementary command is `struct2cell`, which creates a cell array with all of the values of the fields.

Suppose you have the name of a field as a string, and you want to get the value. Here is the syntax for that.

```
f = 'a';
s.(f)
ans =
    14
```

Suppose you're not sure if the field actually exists.

```
if isfield(s, 'd')
    d = s.d;
else
    d = [];
end
```