MATLAB® digest

Matrix Indexing in MATLAB®

by Steve Eddins and Loren Shure
Send email to Steve Eddins and Loren Shure

Indexing into a matrix is a means of selecting a subset of elements from the matrix. MATLAB has several indexing styles that are not only powerful and flexible, but also readable and expressive. Indexing is a key to MATLAB's effectiveness at capturing matrix-oriented ideas in understandable computer programs.

Indexing is also closely related to another term MATLAB users often hear: vectorization. Vectorization means using MATLAB language constructs to eliminate program loops, usually resulting in programs that run faster and are more readable. Of the many possible vectorization techniques, many rely on MATLAB indexing methods, five of which are described in this article. To learn more about other similar methods, see the resources listed at the end of this article.

Indexing Vectors

Let's start with the simple case of a vector and a single subscript. The vector is

```
v = [16 5 9 4 2 11 7 14];
```

The subscript can be a single value.

```
v(3)     % Extract the third element
ans =
      9
```

Or the subscript can itself be another vector.

```
v([1 5 6])      % Extract the first, fifth, and sixth elements
ans =
      16    2    11
```

MATLAB's colon notation provides an easy way to extract a range of elements from `v`.

```
v(3:7)     % Extract the third through the seventh elements
ans =
      9    4    2    11    7
```

Swap the two halves of v to make a new vector.

```
v2 = v([5:8 1:4])     % Extract and swap the halves of v
v2 =
      2    11    7    14    16    5    9    4
```

The special end operator is an easy short-hand way to refer to the last element of `v`.

```
v(end)     % Extract the last element
ans =
```

```
      14
```

The end operator can be used in a range.

```
v(5:end)     % Extract the fifth through the last elements
ans =
      2   11    7   14
```

You can even do arithmetic using end.

```
v(2:end-1)     % Extract the second through the next-to-last elements
ans =
      5    9    4    2   11    7
```

Combine the colon operator and end to achieve a variety of effects, such as extracting every k-th element or flipping the entire vector.

```
v(1:2:end)   % Extract all the odd elements
ans =
    16    9    2    7
v(end:-1:1)   % Reverse the order of elements
ans =
    14    7   11    2    4    9    5   16
```

By using an indexing expression on the left side of the equal sign, you can *replace* certain elements of the vector.

```
v([2 3 4]) = [10 15 20]   % Replace some elements of v
v =
    16   10   15   20    2   11    7   14
```

Usually the number of elements on the right must be the same as the number of elements referred to by the indexing expression on the left. You can always, however, use a *scalar* on the right side.

```
v([2 3]) = 30   % Replace second and third elements by 30
v =
    16   30   30   20    2   11    7 1  4
```

This form of indexed assignment is called *scalar expansion*.

Indexing Matrices with Two Subscripts

Now consider indexing into a matrix. We'll use a magic square for our experiments.

```
A = magic(4)
A =
      16           2           3          13
       5          11          10           8
       9           7           6          12
       4          14          15           1
```

Most often, indexing in matrices is done using two subscripts - one for the rows and one for the columns. The simplest form just picks out a single element.

```
A(2,4)   % Extract the element in row 2, column 4
ans =
       8
```

More generally, one or both of the row and column subscripts can be vectors.

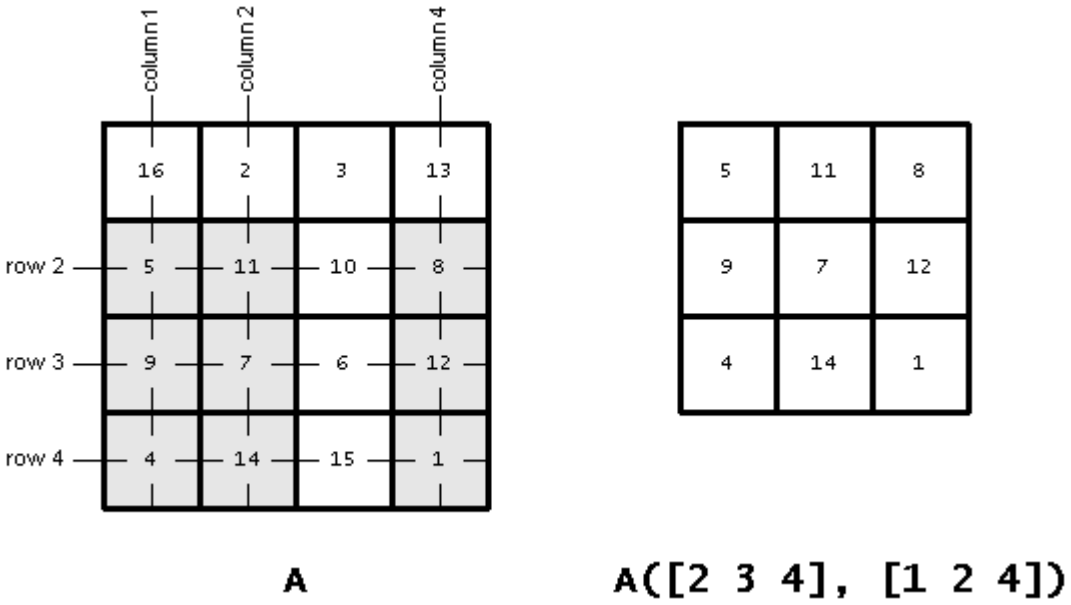
```
A(2:4,1:2)
ans =
```

```
     5    11
     9     7
     4    14
```

A single ":" in a subscript position is short-hand notation for "1:end" and is often used to select entire rows or columns.

```
A(3,:)    % Extract third row
ans =
    9    7    6    12
A(:,end)    % Extract last column
ans =
     13
      8
     12
      1
```

There is often confusion over how to select scattered elements from a matrix. For example, suppose you want to extract the `(2,1)`, `(3,2)`, and `(4,4)` elements from `A`? The expression `A([2 3 4], [1 2 4])` won't do what you want. This diagram illustrates how two-subscript indexing works.



Extracting scattered elements from a matrix requires a different style of indexing, and that brings us to our next topic.

Linear Indexing

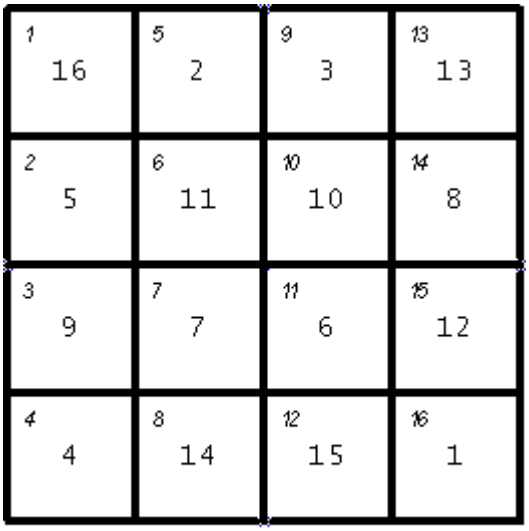
What does this expression `A(14)` do?

When you index into the matrix `A` using only one subscript, MATLAB treats `A` as if its elements were strung out in a long column vector, by going down the columns consecutively, as in:

```
     16
      5
      9
      -
      8
     12
      1
```

The expression `A(14)` simply extracts the 14th element of the implicit column vector. Indexing into a matrix with a single subscript in this way is often called *linear indexing*.

Here are the elements of the matrix `A` along with their linear indices.



A

The linear index of each element is shown in the upper left.

From the diagram you can see that `A(14)` is the same as `A(2,4)`.

The single subscript can be a vector containing more than one linear index, as in:

```
A([6 12 15])
ans =
    11   15   12
```

Consider again the problem of extracting just the `(2,1)`, `(3,2)`, and `(4,4)` elements of `A`. You can use linear indexing to extract those elements:

```
A([2 7 16])
ans =
    5   7   1
```

That's easy to see for this example, but how do you compute linear indices in general? MATLAB provides a function called sub2ind that converts from row and column subscripts to linear indices. You can use it to extract the desired elements this way.

```
idx = sub2ind(size(A), [2 3 4], [1 2 4])
ans =
    2   7   16
A(idx)
ans =
    5   7   1
```

Advanced Examples using Linear Indexing

Example 1: Shifting the rows of a matrix

A MATLAB user recently posed this question in the `comp.soft-sys.matlab` newsgroup.

If I want to shift the rows of an m-by-n matrix `A` by `k` places, I use `A(:,[n-k+1:n 1:n-k])`. But what if `k` is a function of the row number? That is, what if `k` is a vector of length `m`? Is there a quick

and easy way to do this?

Regular newsgroup contributor, Peter Acklam, posted this solution that uses `sub2ind` and linear indexing.

```
% index vectors for rows and columns
p = 1:m;
q = 1:n;
% index matrices for rows and columns
[P, Q] = ndgrid(p, q);
% create a matrix with the shift values
K = repmat(k(:), [1 n]);
% update the matrix with the column indexes
Q = 1 + mod(Q+K-1, n);
% create matrix of linear indexes
ind = sub2ind([m n], P, Q);
% finally, create the output matrix
B = A(ind);
```

Example 2: Setting some matrix elements to zero

Another MATLAB user posted this question.

I want to get the maximum of each row, which isn't really a problem, but afterwards I want to set all the other elements to zero. For example, this matrix:

```
     1     2     3     4
     5     5     6     5
     7     9     8     3
```

should become:

```
     0     0     0     4
     0     0     6     0
     0     9     0     0
```

Another regular newsgroup contributor, Brett Shoelson, provided this compact solution.

```
[Y,I] = max(A, [], 2);
B = zeros(size(A));
B(sub2ind(size(A), 1:length(I), I')) = Y;
```

Logical Indexing

Another indexing variation, *logical indexing*, has proven to be both useful and expressive. In logical indexing, you use a single, logical array for the matrix subscript. MATLAB extracts the matrix elements corresponding to the nonzero values of the logical array. The output is always in the form of a column vector. For example, `A(A > 12)` extracts all the elements of `A` that are greater than 12.

```
A(A > 12)
ans =
      16
      14
      15
      13
```

Many MATLAB functions that start with "is" return logical arrays and are very useful for logical indexing. For example, you could replace all the NaNs in an array with another value by using a combination of `isnan`, logical indexing, and scalar expansion. To replace all `NaN` elements of the matrix `B` with zero, use

```
B(isnan(B)) = 0
```

Or you could replace all the spaces in a string matrix `str` with underscores.

```
str(isspace(str)) = '_'
```

Logical indexing is closely related to the `find` function. The expression `A(A > 5)` is equivalent to `A(find(A > 5))`. Which form you use is mostly a matter of style and your sense of the readability of your code, but it also depends on whether or not you need the actual index values for something else in the computation. For example, suppose you want to temporarily replace `NaN` values with zeros, perform some computation, and then put the `NaN` values back in their original locations. In this example, the computation is two-dimensional filtering using `filter2`. You do it like this.

```
nan_locations = find(isnan(A));
A(nan_locations) = 0;
A = filter2(ones(3,3), A);
A(nan_locations) = NaN;
```

We hope that the MATLAB indexing variants illustrated in this article give you a feel for ways you can express algorithms compactly and efficiently. Including these techniques and related functions in your MATLAB programming repertoire expands your ability to create masterful, readable, and vectorized code.

Resources

Here are a few places to go for further information on MATLAB indexing styles and related topics.

- The book *Using MATLAB* from the MATLAB documentation set.The sections *Subscripting and Indexing and Optimizing MATLAB Code* are particularly relevant. http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/matlab_prog.shtml
- The technical note "How Do I Vectorize My Code?"
- The Usenet newsgroup *comp.soft-sys.matlab*
  Quite a few very knowledgeable MATLAB users offer their help in this newsgroup with indexing and vectorization techniques. There is also a FAQ (Frequently Asked Questions list), maintained by MATLAB user Pete Boettcher.
- The article "MATLAB Array Manipulation Tips and Tricks," written by MATLAB user, Peter Acklam.

related topics:

News & Notes | Using MathWorks Products For... | MATLAB Based Books | Third-Party Products