

# Multidimensional Arrays

## Overview

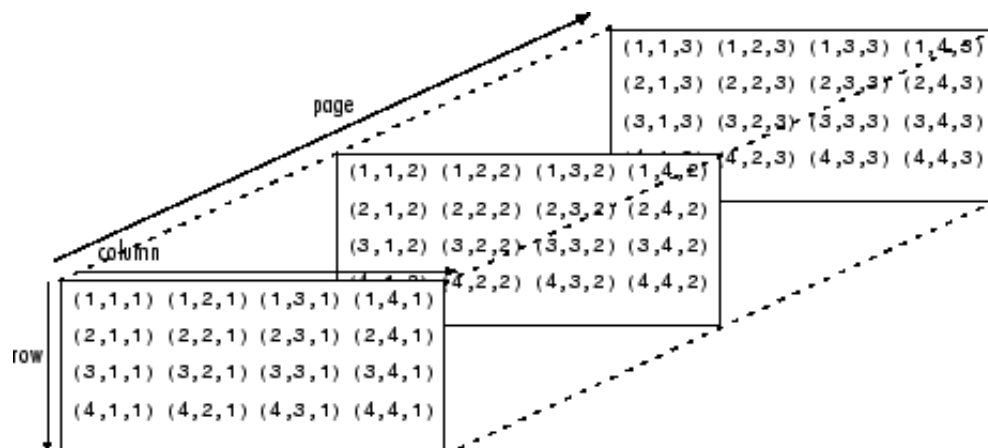
An array having more than two dimensions is called a multidimensional array in the MATLAB® application. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Matrices have two dimensions: the row dimension and the column dimension.

	column			
row	(1,1)	(1,2)	(1,3)	(1,4)
	(2,1)	(2,2)	(2,3)	(2,4)
	(3,1)	(3,2)	(3,3)	(3,4)
	(4,1)	(4,2)	(4,3)	(4,4)

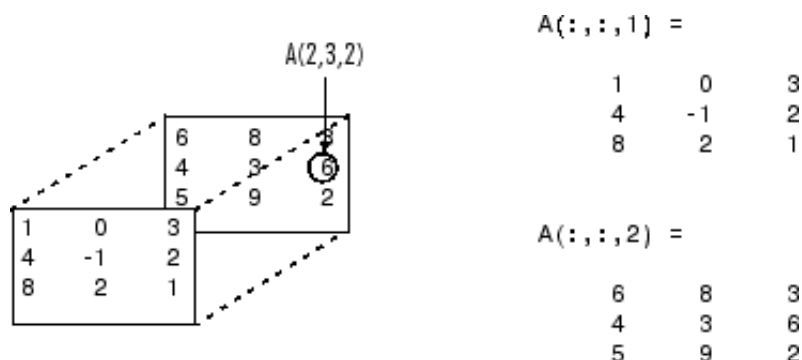
You can access a two-dimensional matrix element with two subscripts: the first representing the row index, and the second representing the column index.

Multidimensional arrays use additional subscripts for indexing. A three-dimensional array, for example, uses three subscripts:

- The first references array dimension 1, the row.
- The second references dimension 2, the column.
- The third references dimension 3. This illustration uses the concept of a *page* to represent dimensions 3 and higher.



To access the element in the second row, third column of page 2, for example, you use the subscripts (2,3,2).



As you add dimensions to an array, you also add subscripts. A four-dimensional array, for example, has four subscripts. The first two reference a row-column pair; the second two access the third and fourth dimensions of data.

Most of the operations that you can perform on matrices (i.e., two-dimensional arrays) can also be done on multidimensional arrays.

**Note** The general multidimensional array functions reside in the `datatypes` directory.

## Creating Multidimensional Arrays

You can use the same techniques to create multidimensional arrays that you use for two-dimensional matrices. In addition, MATLAB provides a special concatenation function that is useful for building multidimensional arrays.

This section discusses

- [Creating and Extending Multidimensional Arrays Using Indexed Assignment](#)
- [Generating Arrays Using MATLAB Functions](#)
- [Building Multidimensional Arrays with the `cat` Function](#)

### Creating and Extending Multidimensional Arrays Using Indexed Assignment

You can create a multidimensional array by creating a 2-D array and extending it. Create a 2-D array `A` and extend `A` to a 3-D array using indexed assignment:

```
A = [5 7 8; 0 1 9; 4 3 6];
A(:,:,2) = [1 0 4; 3 5 6; 9 8 7]

A(:,:,1) =
     5     7     8
     0     1     9
     4     3     6

A(:,:,2) =
     1     0     4
     3     5     6
     9     8     7
```

You can extend an array by assigning a single value to the new elements. MATLAB expands the scalar value to match the dimensions of the addressed elements. This expansion is called scalar expansion.

Extend `A` by a third page using scalar expansion.

```
A(:,:,3) = 5

A(:,:,1) =
     5     7     8
     0     1     9
     4     3     6

A(:,:,2) =
     1     0     4
     3     5     6
     9     8     7

A(:,:,3) =
     5     5     5
     5     5     5
     5     5     5
```

To extend the rows, columns, or pages of an array, use similar assignment statements. The dimensions of arrays on the right side and the left side of the assignment must be the same.

Extend A into a 3-by-3-by-3-by-2, four-dimensional array. In the first assignment, MATLAB pads A to fill the unassigned elements in the extended dimension with zeros. The next two assignments replace the zeros with the specified values.

```
A(:,:,1,2) = [1 2 3; 4 5 6; 7 8 9];
A(:,:,2,2) = [9 8 7; 6 5 4; 3 2 1];
A(:,:,3,2) = [1 0 1; 1 1 0; 0 1 1];
```

## Generating Arrays Using MATLAB Functions

You can use MATLAB functions such as [randn](#), [ones](#), and [zeros](#) to generate multidimensional arrays in the same way you use them for two-dimensional arrays. Each argument you supply represents the size of the corresponding dimension in the resulting array. For example, to create a 4-by-3-by-2 array of normally distributed random numbers:

```
B = randn(4,3,2)
```

To generate an array filled with a single constant value, use the `repmat` function. `repmat` replicates an array (in this case, a 1-by-1 array) through a vector of array dimensions.

```
B = repmat(5, [3 4 2])
```

```
B(:,:,1) =
     5     5     5     5
     5     5     5     5
     5     5     5     5
```

```
B(:,:,2) =
     5     5     5     5
     5     5     5     5
     5     5     5     5
```

**Note** Any dimension of an array can have size zero, making it a form of empty array. For example, 10-by-0-by-20 is a valid size for a multidimensional array.

## Building Multidimensional Arrays with the cat Function

The [cat](#) function is a simple way to build multidimensional arrays; it concatenates a list of arrays along a specified dimension:

```
B = cat(dim, A1, A2...)
```

where A1, A2, and so on are the arrays to concatenate, and `dim` is the dimension along which to concatenate the arrays.

For example, to create a new array with `cat`:

```
B = cat(3, [2 8; 0 5], [1 3; 7 9])
```

```
B(:,:,1) =
     2     8
     0     5
```

```
B(:,:,2) =
     1     3
     7     9
```

The `cat` function accepts any combination of existing and new data. In addition, you can nest calls to `cat`. The lines below, for example, create a four-dimensional array.

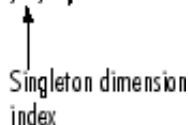
```
A = cat(3, [9 2; 6 5], [7 1; 8 4])
B = cat(3, [3 5; 0 1], [5 6; 2 1])
D = cat(4, A, B, cat(3, [1 2; 3 4], [4 3; 2 1]))
```

cat automatically adds subscripts of 1 between dimensions, if necessary. For example, to create a 2-by-2-by-1-by-2 array, enter

```
C = cat(4, [1 2; 4 5], [7 8; 3 2])
```

In the previous case, cat inserts as many singleton dimensions as needed to create a four-dimensional array whose last dimension is not a singleton dimension. If the dim argument had been 5, the previous statement would have produced a 2-by-2-by-1-by-1-by-2 array. This adds additional 1s to indexing expressions for the array. To access the value 8 in the four-dimensional case, use

```
C(1,2,1,2)
```



Singleton dimension  
index

## Accessing Multidimensional Array Properties

You can use the following MATLAB functions to get information about multidimensional arrays you have created.

- size — Returns the size of each array dimension.

```
size(C)
ans =
     2     2     1     2
   rows columns dim3 dim4
```

- ndims — Returns the number of dimensions in the array.

```
ndims(C)
ans =
     4
```

- whos — Provides information on the format and storage of the array.

```
whos
Name      Size      Bytes   Class
A         2x2x2        64   double array
B         2x2x2        64   double array
C         4-D          64   double array
D         4-D       192   double array
```

Grand total is 48 elements using 384 bytes

## Indexing Multidimensional Arrays

Many of the concepts that apply to two-dimensional matrices extend to multidimensional arrays as well.

To access a single element of a multidimensional array, use integer subscripts. Each subscript indexes a dimension—the first indexes the row dimension, the second indexes the column dimension, the third indexes the first page dimension, and so on.

Consider a 10-by-5-by-3 array `nddata` of random integers:

```
nddata = fix(8 * randn(10,5,3));
```

To access element (3,2) on page 2 of `nddata`, for example, use `nddata(3,2,2)`.

You can use vectors as array subscripts. In this case, each vector element must be a valid subscript, that is, within the bounds defined by the dimensions of the array. To access elements (2,1), (2,3), and (2,4) on page 3 of `nddata`, use

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

### The Colon and Multidimensional Array Indexing

The MATLAB colon indexing extends to multidimensional arrays. For example, to access the entire third column on page 2 of `nddata`, use `nddata(:,3,2)`.

The colon operator is also useful for accessing other subsets of data. For example, `nddata(2:3,2:3,1)` results in a 2-by-2 array, a subset of the data on page 1 of `nddata`. This matrix consists of the data in rows 2 and 3, columns 2 and 3, on the first page of the array.

The colon operator can appear as an array subscript on both sides of an assignment statement. For example, to create a 4-by-4 array of zeros:

```
C = zeros(4, 4)
```

Now assign a 2-by-2 subset of array `nddata` to the four elements in the center of `C`.

```
C(2:3,2:3) = nddata(2:3,1:2,2)
```

### Linear Indexing with Multidimensional Arrays

MATLAB linear indexing also extends to multidimensional arrays. In this case, MATLAB operates on a page-by-page basis to create the storage column, again appending elements columnwise. See [Linear Indexing](#) for an introduction to this topic.

For example, consider a 5-by-4-by-3-by-2 array `C`.

MATLAB displays C as

MATLAB stores C as

page(1,1) =

1	4	3	5
2	1	7	9
5	6	3	2
0	1	5	9
3	2	7	5

page(2,1) =

6	2	4	2
7	1	4	9
0	0	1	5
9	4	4	0
1	8	2	5

page(3,1) =

2	2	8	3
2	5	1	8
5	1	5	2
0	9	0	9
9	4	5	3

page(1,2) =

9	8	2	3
0	0	3	3
6	4	9	6
1	9	2	3
0	2	8	7

page(2,2) =

7	0	1	3
2	4	8	1
7	5	8	6
6	8	8	4
9	4	1	2

page(3,2) =

1	6	6	5
2	9	1	3
7	1	1	1
8	0	1	5
3	2	7	6

1
2
5
0
3
4
1
6
1
2
3
7
3
5
7
5
9
2
9
5
6
7
0
9
1
2
1
0
4
8
4
4
1
4
2
2
9
5
2
5
2
2
5
0
9
2
5
1
9
4
:

Again, a single subscript indexes directly into this column. For example, C(4) produces the result

```
ans =
     0
```

If you specify two subscripts (i,j) indicating row-column indices, MATLAB calculates the offset as described above. Two subscripts always access the first page of a multidimensional array, provided they are within the range of the original array dimensions.

If more than one subscript is present, all subscripts must conform to the original array dimensions. For example, C(6,2) is invalid because all pages of C have only five rows.

If you specify more than two subscripts, MATLAB extends its indexing scheme accordingly. For example, consider four subscripts (i,j,k,l) into a four-dimensional array with size [d1 d2 d3 d4]. MATLAB calculates the offset into the storage column by

$$(l-1)(d_3)(d_2)(d_1) + (k-1)(d_2)(d_1) + (j-1)(d_1) + i$$

For example, if you index the array C using subscripts (3, 4, 2, 1), MATLAB returns the value 5 (index 38 in the storage column).

In general, the offset formula for an array with dimensions [d1 d2 d3 ... dn] using any subscripts (s1 s2 s3 ... sn) is

$$(s_n-1)(d_{n-1})(d_{n-2}) \dots (d_1) + (s_{n-1}-1)(d_{n-2}) \dots (d_1) + \dots + (s_2-1)(d_1) + s_1$$

Because of this scheme, you can index an array using any number of subscripts. You can append any number of 1s to the subscript list because these terms become zero. For example,

```
C(3,2,1,1,1,1,1,1)
```

is equivalent to

```
C(3,2)
```

### Avoiding Ambiguity in Multidimensional Indexing

Some assignment statements, such as

```
A(:, :, 2) = 1:10
```

are ambiguous because they do not provide enough information about the shape of the dimension to receive the data. In the case above, the statement tries to assign a one-dimensional vector to a two-dimensional destination. MATLAB produces an error for such cases. To resolve the ambiguity, be sure you provide enough information about the destination for the assigned data, and that both data and destination have the same shape. For example:

```
A(1, :, 2) = 1:10;
```

### Reshaping Multidimensional Arrays

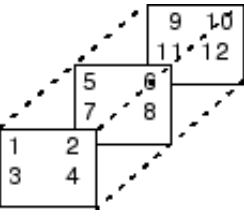
Unless you change its shape or size, a MATLAB array retains the dimensions specified at its creation. You change array size by adding or deleting elements. You change array shape by re-specifying the array's row, column, or page dimensions while retaining the same elements. The `reshape` function performs the latter operation. For multidimensional arrays, its form is

```
B = reshape(A, [s1 s2 s3 ...])
```

`s1`, `s2`, and so on represent the desired size for each dimension of the reshaped matrix. Note that a reshaped array must have the same number of elements as the original array (that is, the product of the dimension sizes is constant).

M	reshape(M, [6 5])
<div><div><div>12345 90637 81502</div><div>97852 35851 69433</div></div></div>	<div><div>13575 96755 85293 24982 03381 10643</div></div>

The `reshape` function operates in a columnwise manner. It creates the reshaped matrix by taking consecutive elements down each column of the original data construct.

C	reshape(C, [6 2])												
	<table border="1"> <tbody> <tr><td>1</td><td>6</td></tr> <tr><td>3</td><td>8</td></tr> <tr><td>2</td><td>9</td></tr> <tr><td>4</td><td>11</td></tr> <tr><td>5</td><td>10</td></tr> <tr><td>7</td><td>12</td></tr> </tbody> </table>	1	6	3	8	2	9	4	11	5	10	7	12
1	6												
3	8												
2	9												
4	11												
5	10												
7	12												

Here are several new arrays from reshaping `nddata`:

```
B = reshape(nddata, [6 25])
C = reshape(nddata, [5 3 10])
D = reshape(nddata, [5 3 2 5])
```

MATLAB automatically removes trailing singleton dimensions (dimensions whose sizes are equal to 1) from all multidimensional arrays. For example, if you attempt to create an array of size 3-by-2-by-1-by-1, perhaps with the command `rand(3,2,1,1)`, then MATLAB strips away the singleton dimensions at the end of the array and creates a 3-by-2 matrix. This is because every array technically has an *infinite* number of trailing singleton dimensions. A 3-by-2 array is the same as an array with size 3-by-2-by-1-by-1-by-1-by-1-by-...

```
A = eye(2)
```

$$A =$$
$$\begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$$

Find the size of the fourth dimension of A.

```
size(A,4)
```

ans =

1

The first two dimensions of an array are never stripped away, since they are always relevant.

```
size(3)
```

ans =

1 1

MATLAB creates singleton dimensions if you explicitly specify them when you create or reshape an array, or if you perform a calculation that results in an array dimension of one:

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

size(B)

ans =

2            3            1            4

```
C = squeeze(B);
```

```
size(C)
```

ans =

2                      3                      4

## Permuting Array Dimensions

The `permute` function reorders the dimensions of an array:



```
B = permute(A, dims);
```

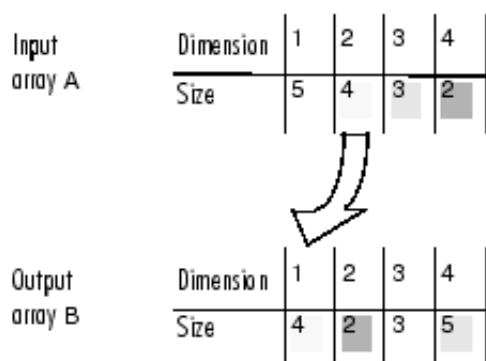
`dims` is a vector specifying the new order for the dimensions of `A`, where 1 corresponds to the first dimension (rows), 2 corresponds to the second dimension (columns), 3 corresponds to pages, and so on.

<b>A</b>	<b>B = permute(A, [2 1 3])</b>	<b>C = permute(A, [3 2 1])</b>																								
<b>A(:, :, 1) =</b> <table> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td></tr> <tr><td>7</td><td>8</td><td>9</td></tr> </table>	1	2	3	4	5	6	7	8	9	<b>B(:, :, 1) =</b> <table> <tr><td>1</td><td>4</td><td>7</td></tr> <tr><td>2</td><td>5</td><td>8</td></tr> <tr><td>3</td><td>6</td><td>9</td></tr> </table>	1	4	7	2	5	8	3	6	9	<b>C(:, :, 1) =</b> <table> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>0</td><td>5</td><td>4</td></tr> </table>	1	2	3	0	5	4
1	2	3																								
4	5	6																								
7	8	9																								
1	4	7																								
2	5	8																								
3	6	9																								
1	2	3																								
0	5	4																								
<b>A(:, :, 2) =</b> <table> <tr><td>0</td><td>5</td><td>4</td></tr> <tr><td>2</td><td>7</td><td>6</td></tr> <tr><td>9</td><td>3</td><td>1</td></tr> </table>	0	5	4	2	7	6	9	3	1	<b>B(:, :, 2) =</b> <table> <tr><td>0</td><td>2</td><td>9</td></tr> <tr><td>5</td><td>7</td><td>3</td></tr> <tr><td>4</td><td>6</td><td>1</td></tr> </table>	0	2	9	5	7	3	4	6	1	<b>C(:, :, 2) =</b> <table> <tr><td>4</td><td>5</td><td>6</td></tr> <tr><td>2</td><td>7</td><td>6</td></tr> </table>	4	5	6	2	7	6
0	5	4																								
2	7	6																								
9	3	1																								
0	2	9																								
5	7	3																								
4	6	1																								
4	5	6																								
2	7	6																								
		<b>C(:, :, 3) =</b> <table> <tr><td>7</td><td>8</td><td>9</td></tr> <tr><td>9</td><td>3</td><td>1</td></tr> </table>	7	8	9	9	3	1																		
7	8	9																								
9	3	1																								

For a more detailed look at the `permute` function, consider a four-dimensional array `A` of size 5-by-4-by-3-by-2. Rearrange the dimensions, placing the column dimension first, followed by the second page dimension, the first page dimension, then the row dimension. The result is a 4-by-2-by-3-by-5 array.

```
B = permute(A, [2 4 3 1])
```

Move dimension 2 of `A` to first subscript position of `B`, dimension 4 to second subscript position, and so on.



The order of dimensions in `permute`'s argument list determines the size and shape of the output array. In this example, the second dimension moves to the first position. Because the second dimension of the original array had size 4, the output array's first dimension also has size 4.

You can think of `permute`'s operation as an extension of the `transpose` function, which switches the row and column dimensions of a matrix. For `permute`, the order of the input dimension list determines the reordering of the subscripts. In the example above, element (4,2,1,2) of `A` becomes element (2,2,1,4) of `B`, element (5,4,3,2) of `A` becomes element (4,2,3,5) of `B`, and so on.

### Inverse Permutation

The `ipermute` function is the inverse of `permute`. Given an input array `A` and a vector of dimensions `v`, `ipermute` produces an array `B` such that `permute(B,v)` returns `A`.

For example, these statements create an array `E` that is equal to the input array `C`:

```
D = ipermute(C, [1 4 2 3]);
E = permute(D, [1 4 2 3])
```

You can obtain the original array after permuting it by calling `ipermute` with the same vector of dimensions.

### Computing with Multidimensional Arrays

Many of the MATLAB computational and mathematical functions accept multidimensional arrays as arguments. These functions operate on specific dimensions of multidimensional arrays; that is, they operate on individual elements, on vectors, or on matrices.

## Operating on Vectors

Functions that operate on vectors, like `sum`, `mean`, and so on, by default typically work on the first nonsingleton dimension of a multidimensional array. Most of these functions optionally let you specify a particular dimension on which to operate. There are exceptions, however. For example, the `cross` function, which finds the cross product of two vectors, works on the first nonsingleton dimension having length 3.

**Note** In many cases, these functions have other restrictions on the input arguments — for example, some functions that accept multiple arrays require that the arrays be the same size. Refer to the online help for details on function arguments.

## Operating Element-by-Element

MATLAB functions that operate element-by-element on two-dimensional arrays, like the trigonometric and exponential functions in the `elfun` directory, work in exactly the same way for multidimensional cases. For example, the `sin` function returns an array the same size as the function's input argument. Each element of the output array is the sine of the corresponding element of the input array.

Similarly, the arithmetic, logical, and relational operators all work with corresponding elements of multidimensional arrays that are the same size in every dimension. If one operand is a scalar and one an array, the operator applies the scalar to each element of the array.

## Operating on Planes and Matrices

Functions that operate on planes or matrices, such as the linear algebra and matrix functions in the `matfun` directory, do not accept multidimensional arrays as arguments. That is, you cannot use the functions in the `matfun` directory, or the array operators `*`, `^`, `\`, or `/`, with multidimensional arguments. Supplying multidimensional arguments or operands in these cases results in an error.

You can use indexing to apply a matrix function or operator to matrices within a multidimensional array. For example, create a three-dimensional array `A`:

```
A = cat(3, [1 2 3; 9 8 7; 4 6 5], [0 3 2; 8 8 4; 5 3 5], ...
    [6 4 7; 6 8 5; 5 4 3]);
```

Applying the `eig` function to the entire multidimensional array results in an error:

```
eig(A)
??? Undefined function or method 'eig' for input
arguments of type 'double' and attributes 'full 3d real'.
```

You can, however, apply `eig` to planes within the array. For example, use colon notation to index just one page (in this case, the second) of the array:

```
eig(A(:,:,2))
ans =
    12.9129
    -2.6260
     2.7131
```

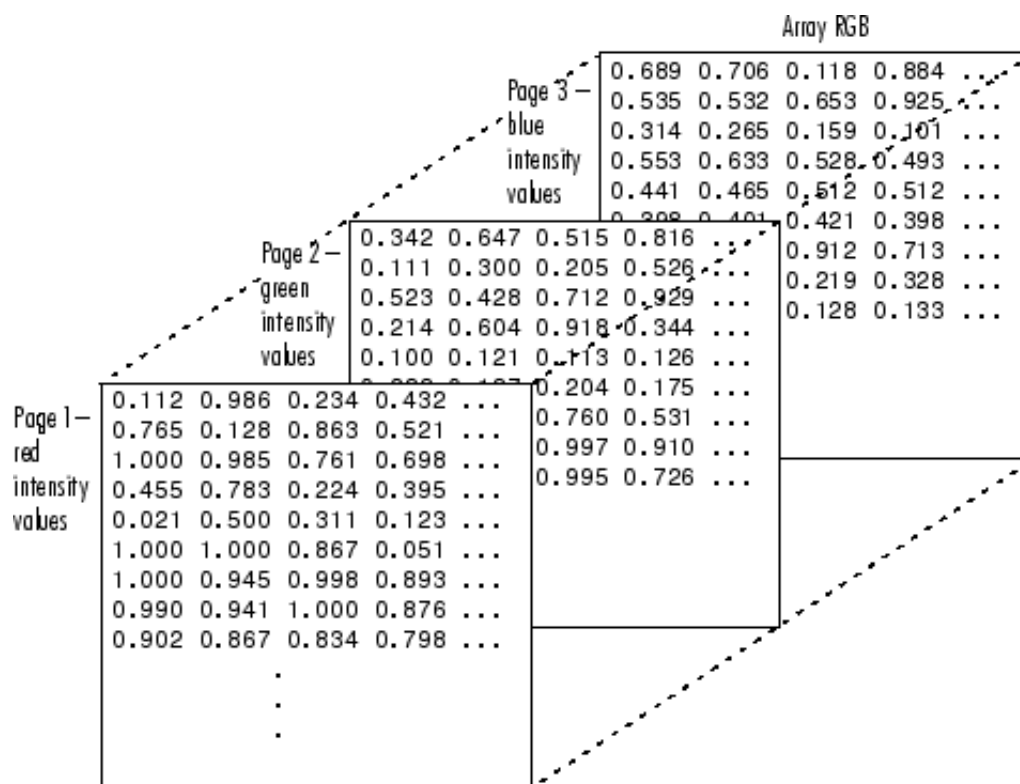
**Note** In the first case, subscripts are not colons; you must use `squeeze` to avoid an error. For example, `eig(A(2,:,:))` results in an error because the size of the input is `[1 3 3]`. The expression `eig(squeeze(A(2,:,:)))`, however, passes a valid two-dimensional matrix to `eig`.

## Organizing Data in Multidimensional Arrays

You can use multidimensional arrays to represent data in two ways:

- As planes or pages of two-dimensional data. You can then treat these pages as matrices.
- As multivariate or multidimensional data. For example, you might have a four-dimensional array where each element corresponds to either a temperature or air pressure measurement taken at one of a set of equally spaced points in a room.

For example, consider an RGB image. For a single image, a multidimensional array is probably the easiest way to store and access data.



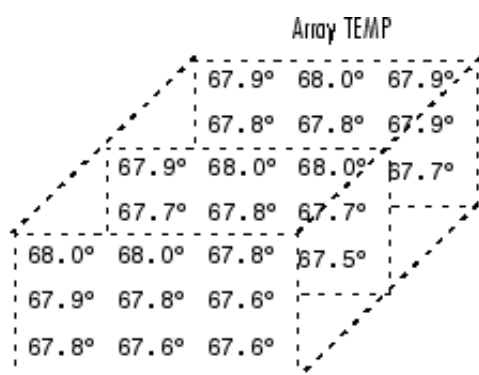
To access an entire plane of the image, use

```
redPlane = RGB(:,:,1);
```

To access a subimage, use

```
subimage = RGB(20:40,50:85,:);
```

The RGB image is a good example of data that needs to be accessed in planes for operations like display or filtering. In other instances, however, the data itself might be multidimensional. For example, consider a set of temperature measurements taken at equally spaced points in a room. Here the location of each value is an integral part of the data set—the physical placement in three-space of each element is an aspect of the information. Such data also lends itself to representation as a multidimensional array.



Now to find the average of all the measurements, use

```
mean(mean(mean(TEMP)));
```

To obtain a vector of the "middle" values (element (2,2)) in the room on each page, use

```
B = TEMP(2,2,:);
```

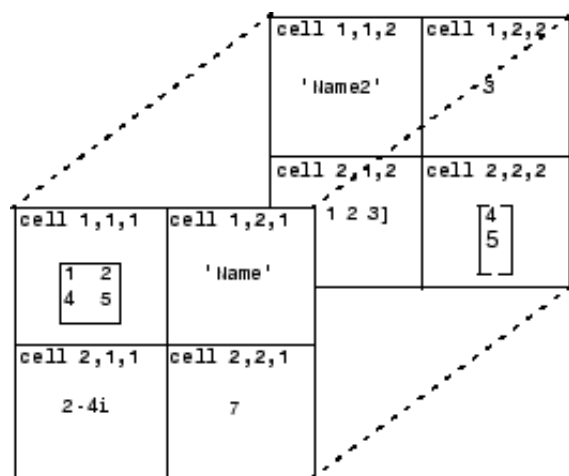
## Multidimensional Cell Arrays

Like numeric arrays, the framework for multidimensional cell arrays in MATLAB is an extension of the two-dimensional cell array model. You can use the `cat` function to build multidimensional cell arrays, just as you use it for numeric arrays.

For example, create a simple three-dimensional cell array `C`:

```
A{1,1} = [1 2;4 5];
A{1,2} = 'Name';
A{2,1} = 2-4i;
A{2,2} = 7;
B{1,1} = 'Name2';
B{1,2} = 3;
B{2,1} = 0:1:3;
B{2,2} = [4 5]';
C = cat(3, A, B);
```

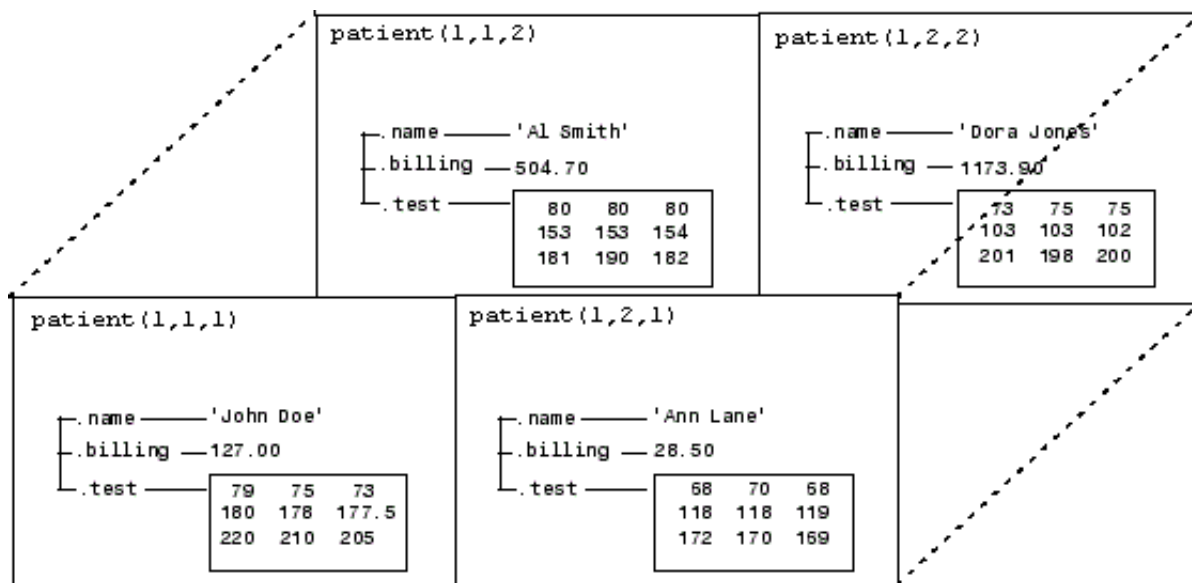
The subscripts for the cells of `C` look like



## Multidimensional Structure Arrays

Multidimensional structure arrays are extensions of rectangular structure arrays. Like other types of multidimensional arrays, you can build them using direct assignment or the `cat` function:

```
patient(1,1,1).name = 'John Doe';
patient(1,1,1).billing = 127.00;
patient(1,1,1).test = [79 75 73; 180 178 177.5; 220 210 205];
patient(1,2,1).name = 'Ann Lane';
patient(1,2,1).billing = 28.50;
patient(1,2,1).test = [68 70 68; 118 118 119; 172 170 169];
patient(1,1,2).name = 'Al Smith';
patient(1,1,2).billing = 504.70;
patient(1,1,2).test = [80 80 80; 153 153 154; 181 190 182];
patient(1,2,2).name = 'Dora Jones';
patient(1,2,2).billing = 1173.90;
patient(1,2,2).test = [73 73 75; 103 103 102; 201 198 200];
```



### Applying Functions to Multidimensional Structure Arrays

To apply functions to multidimensional structure arrays, operate on fields and field elements using indexing. For example, find the sum of the columns of the test array in `patient(1,1,2)`:

```
sum((patient(1,1,2).test));
```

Similarly, add all the billing fields in the `patient` array:

```
total = sum([patient.billing]);
```