Lesson 03: Indexing

At first, indexing in MATLAB seems like a trivial topic, and for the most part, the basics will suffice for most *basic* operations. However, once you start trying to vectorize more complicated operations, especially ones that rely on conditions, you'll quickly find yourself limited in what you can calculate.

Subscript Indexing

For the remainder of this lesson, we'll be working with the following:

$$\boldsymbol{x} = [1, 2, 3, 4, 5, 6, 7, 8] \tag{1}$$

$$\boldsymbol{A} = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix} \tag{2}$$

We've already seen that we can access multiple elements of a vector by indexing with another vector:

```
a([3, 1, 2])
```

```
ans =

     3     1     2
```

But we can also write to these elements:

```
y = x;
y([3, 1, 2]) = -1
```

```
y =

    -1    -1    -1     4     5     6     7     8
```

Matrices behave much the same as this behavior extends to each dimension:

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

```
ans =

     5     9
     8    12
```


Say we want to select all elements of a dimension; we can use a bare `:` in place of `1:end` to save us some typing:

```
A(:, 3)
```

```
ans =

     9
    10
    11
    12
```

Linear Indexing

Last lesson, we flirted with the idea of linear indexing without getting too much into how it works. Essentially, MATLAB stores its arrays in column-major order, and linear indexing is nothing more than taking advantage of this design decision.

Take for example:

```
A([4, end])  % [A(1, 4), A(4, 4)]
```

```
ans =

     4    16
```

You might find yourself having to convert between subscripts and linear indices depending on what you're trying to achieve, and for that MATLAB has the `sub2ind()` and `ind2sub()` functions.

As the names may suggest, they return return one indexing regime given the other. The only requirement for these functions is the size of the array you're working with:

```
[row, col] = ind2sub(size(A), 6)
```

```
row =

     2

col =

     2
```



```
ind = sub2ind(size(A), row, col)
```

```
ind =

     6
```

Logical Arrays

We've finally reached the point where we can introduce a new datatype: the *logical* datatype. MATLAB logical values operate over $\mathbb{Z}_2$ where `0` denotes `false`, and `1` denotes `true`.

What makes logical arrays so powerful is that they obey the rules of Boolean Algebra and allow for us to extract elements of arrays.

Operator	Purpose
`&`	Logical AND
`&&`	Logical AND with short-circuiting
`~`	Logical NOT
`|`	Logical OR
`||`	Logical OR with short-circuiting
`xor`	Logical exclusive-OR
`all`	Determine if all array elements are nonzero or true
`any`	Determine if any array elements are nonzero

Table 1: Logical Operations

Say we wanted to extract the corners of $\boldsymbol{A}$. We could create a logical array with `logical()` with 1's in the location of the elements we'd like to extract:

```
corners = logical([
                   1, 0, 0, 1
                   0, 0, 0, 0
                   0, 0, 0, 0
                   1, 0, 0, 1
                  ]);
```


We can then use this logical array to index into $\boldsymbol{A}$:

```
A(corners)
```

```
ans =

     1
     4
    13
    16
```

Relational Operations

Now that we know about logical values, we can combine this power with relational operators to select elements satisfying a condition.

Operator	Purpose
==	Equality
>=	Greater than or equal to
>	Greater than
<=	Less than or equal to
<	Less than
~=	Inequality
isequal	Array equality
isequaln	Array equality, treating NaN values as equal

Table 2: Relational Operations

For example:

```
A < 7
```

```
ans =

  4x4 logical array

   1   1   0   0
   1   1   0   0
   1   0   0   0
   1   0   0   0
```


Notice how the result of this operation is a logical array, which might tempt you to utilize our previously learned logical operators!

Say we wanted to get all values in the domain $(3, 7)$:

```
A < 7 & A > 3
```

```
ans =

  4x4 logical array

   0   1   0   0
   0   1   0   0
   0   0   0   0
   1   0   0   0
```

As another example, we can use the `find()` function to get the linear index of the element(s) that equal 5:

```
find(A == 5)
```

```
ans =

     5
```

As a final example, let's select all the columns of $\boldsymbol{A}$ that sum to a value greater than 30:

```
satisfy = sum(A) > 30
oops = A(satisfy)
```

```
satisfy =

  1x4 logical array

   0   0   1   1

oops =

     3     4
```

Oops, it looks like we didn't get what we wanted! Unfortunately, MATLAB doesn't broadcast when indexing, so we'll need to use `repmat()` to generate the logical matrix we desire and a `reshape()` to get the final output in the correct format.


```
gucci = repmat(satisfy, 4, 1)
slay = A(gucci)
cols = reshape(slay, 4, [])
```

```
gucci =

  4x4 logical array

   0   0   1   1
   0   0   1   1
   0   0   1   1
   0   0   1   1

slay =

     9
    10
    11
    12
    13
    14
    15
    16

cols =

     9    13
    10    14
    11    15
    12    16
```

As you can see, these operations are *powerful* but may require you to massage the outputs to get it into the correct format. At first, they may seem unintuitive, but like all things in life, you'll get the hang of it with persistence and patience.

