Lesson 02: Vectorization

So far, what we've learned hasn't been all that useful. Sure, we can instantiate matrices, but we're here to make MATLAB work like a horse. One of the fundamental principles behind *clean & efficient* code is the vectorization of operations. By vectorizing, our code appears more like mathematical expressions found in textbooks, introduces fewer opportunities for errors, and runs much faster as it can utilize hardware vector operations![1]

Vector Generation

Say we want to create a vector $\boldsymbol{x} = [i : i \in \mathbb{Z}_8]$. We could do it like so:

```
x = 0:7
```

```
x =

     0     1     2     3     4     5     6     7
```

We achieved the following result by utilizing the `:` operator. This operator is useful for generating vectors, which we can use either for calculations or indexing into arrays.

The operator also allows us to specify a step increment other than one:

```
y = 2:-2:-12
```

```
y =

     2     0    -2    -4    -6    -8   -10   -12
```

But what if we want to create a vector over some range with a fixed amount of points (i.e. four evenly spaced values for $t \in [0, 1]$)? We could instead utilize `linspace()`:

```
t = linspace(0, 1, 4)
```

```
t =

         0    0.3333    0.6667    1.0000
```

[1] `https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html`

Indexing

Now, to *ruin* your day. MATLAB starts indexing at one...

Let's reuse $\boldsymbol{x}$ from the previous section. We can access the third element of $\boldsymbol{x}$ like so:

```
x(3)
```

```
ans =

     2
```

We can also access multiple elements at once:

```
x([1, 3, 6])
```

```
ans =

     0     2     5
```

Or if we're so inclined, the penultimate element:

```
x(end - 1)
```

```
ans =

     6
```

Matrices logically follow by adding another index:

```
A = [
     1, 5
     6, 2
    ];

A(2, 2)
```

```
ans =

     2
```


We can also index *linearly:*

```
A(4)
```

```
ans =

     2
```

But that's an adventure for another time `;)`

Size & Shape

When dealing with N-dimensional arrays, it's important to know how to get basic information related to the size and shape. Lucky for us, MATLAB offers a few convenient functions just for this!

- `length()` — the length of the largest array dimension
- `size()` — the size of the array
- `ndims()` — the number of array dimensions
- `numel()` — the number of array elements

Reshaping Arrays

Say we wanted to create the following matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \tag{1}$$

We can agree that it's nothing more than

$$\boldsymbol{a} = [1, 2, 3, 4, 5, 6, 7, 8, 9], \tag{2}$$

where each column of $\boldsymbol{A}$ is every three elements of $\boldsymbol{a}$.

We can achieve this pragmatically with `reshape()`:

```
A = reshape(1:9, [3, 3])
```

```
A =

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

What's important to note is that for a reshape to be possible, the resultant array must have the same number of elements. Given this, we can let MATLAB do the hard work of determining a dimension for us:

```
A = reshape(1:9, 3, [])
```

```
A =

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

We could also *flatten* a matrix by reshaping, but you'll rarely see a MATLAB programmer do it this way. Instead, most programmers will linearly index all the elements of the matrix:

```
a = A(1:end)
```

```
a =

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

Broadcasting

Probably the most trippy thing when you first start using MATLAB is the concept of ***broadcasting***. In essence, when working with array operations, MATLAB will take the liberty of *implicitly* doing a `repmat()` to make the sizes of operations work out.

Take for example:

```
u = [1, 2];
v = [3; 4; 5];

u + v
```

```
ans =

     4     5
     5     6
     6     7
```

Strange, but it's as if we had performed the following operation manually:

```
[u; u; u] + [v, v]
```

Although broadcasting is *very* convenient, it is always waiting for you to drop your guard to stab you in the back when you least expect it. Having been a victim of MATLAB's senseless violence, you can do nothing besides being cautious around broadcasting operations to avoid introducing some *nasty* bugs.[2]

Function Calls

Depending on which function you call, its behavior on matrices will be different. Let's reuse matrix $\boldsymbol{A}$ from the previous section:

```
exp(A)
```

```
ans =

   1.0e+03 *

    0.0027    0.0546    1.0966
    0.0074    0.1484    2.9810
    0.0201    0.4034    8.1031
```

As you can see, `exp()` returns the exponential evaluated on each element of an array. However, this is notalways the case; take, for example:

[2] Worst of all, you might even introduce a broadcasting operation unintentionally; that's when the real head-banging can begin!


```
sum(A)
```

```
ans =

     6    15    24
```

Notice that `sum()` only evaluated the sum of each column of $\boldsymbol{A}$. This example illustrates why it's important to read a function's documentation *before* you use it, as it may not behave as you'd expect!

Now, we turn to an interesting experiment where we sum all the elements of a matrix. Let's generate a "large" matrix with uniformly distributed pseudorandom integers:

```
rng(0xc1c3eddb);  % sets a constant seed for reproducibility

B = randi(10, 10e3, 10e3);
```

Now, let's compare the performance of summing all the elements of matrix $\boldsymbol{B}$ using two methods:

```
tic
sum(B(1:end))
toc
```

```
ans =

   549929576

Elapsed time is 1.052319 seconds.
```

```
tic
sum(B, 'all')
toc
```

```
ans =

   549929576

Elapsed time is 0.104034 seconds.
```

Wow, I don't know about you, but flattening is *significantly* slower than just calling `sum()` correctly. The reason for this dramatic difference is that we need to make a flattened *copy* of $\boldsymbol{B}$ in memory before we can sum all the values.

The moral of the story is that a bit of reading can save you a lot of time!

