Contents

1 Lab 2: Static Response, Cantilevered Beam 3
  1.1 Objectives ....................................................... 3
  1.2 Scalars, Vectors and Matrices (Allen Downey) ................ 3
    1.2.1 Attribution ........................................... 3
    1.2.2 Vectors .................................................. 3
    1.2.3 Vector arithmetic .................................... 3
    1.2.4 Everything is a matrix ................................ 4
    1.2.5 Indices .................................................. 5
    1.2.6 Indexing errors ....................................... 6
    1.2.7 Vectors and Sequences ................................ 7
    1.2.8 Plotting .................................................. 8
    1.2.9 Plotting Vectors ....................................... 8
    1.2.10 Reduce ................................................... 9
    1.2.11 Apply ................................................... 9
    1.2.12 Spoiling the fun .................................... 10
    1.2.13 Glossary ............................................... 11
  1.3 Example of Apply: Function Evaluation ....................... 11
    1.3.1 Using a Loop .......................................... 11
    1.3.2 Using an Apply Operation .............................. 12
  1.4 Example of Reduce: Time-Average ............................ 12
  1.5 Representing Polynomials with Vectors ....................... 13
  1.6 Example: Fitting a Curve to Experimental Data ............. 14
Chapter 1

Lab 2: Static Response, Cantilevered Beam

1.1 Objectives

1.2 Scalars, Vectors and Matrices (Allen Downey)

1.2.1 Attribution

This section is an adaptation from Allen Downey’s excellent book “Physical Modeling in MATLAB” which is freely available from http://greenteapress.com/matlab and distributed under the terms of the Creative Commons Attribution-NonCommercial 3.0 Unported License, which is available at http://creativecommons.org/licenses/by-nc/3.0/.

1.2.2 Vectors

The values we have seen so far are all single numbers, which are called scalars to contrast them with vectors and matrices, which are collections of numbers.

Variables that contain vectors are often capital letters. That’s just a convention; MATLAB doesn’t require it, but for beginning programmers it is a useful way to remember what is a scalar and what is a vector.

The numbers that make up the vector are called elements.

1.2.3 Vector arithmetic

You can perform arithmetic with vectors, too. If you add a scalar to a vector, MATLAB increments each element of the vector:

```
>> Y = X+5
Y = 6 7 8 9 10
```
The result is a new vector; the original value of \( X \) is not changed.

If you add two vectors, MATLAB adds the corresponding elements of each vector and creates a new vector that contains the sums:

\[
\begin{align*}
\gg Z &= X+Y \\
Z &= 7 \quad 9 \quad 11 \quad 13 \quad 15
\end{align*}
\]

But adding vectors only works if the operands are the same size. Otherwise:

\[
\begin{align*}
\gg W &= 1:3 \\
W &= 1 \quad 2 \quad 3 \\
\gg X+W \\
??? \ Error \ using \Rightarrow \ plus \\
Matrix \ dimensions \ must \ agree.
\end{align*}
\]

The error message in this case is confusing, because we are thinking of these values as vectors, not matrices. The problem is a slight mismatch between math vocabulary and MATLAB vocabulary.

### 1.2.4 Everything is a matrix

In math (specifically in linear algebra) a vector is a one-dimensional sequence of values and a matrix is two-dimensional (and, if you want to think of it that way, a scalar is zero-dimensional). In MATLAB, everything is a matrix.

You can see this if you use the \texttt{whos} command to display the variables in the workspace. \texttt{whos} is similar to \texttt{who} except that it also displays the size and type of each variable.

First I’ll make one of each kind of value:

\[
\begin{align*}
\gg \texttt{scalar} &= 5 \\
\texttt{scalar} &= 5 \\
\gg \texttt{vector} &= 1:5 \\
\texttt{vector} &= 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
\gg \texttt{matrix} &= \texttt{ones}(2,3) \\
\texttt{matrix} &= \\
&\quad 1 \quad 1 \quad 1 \\
&\quad 1 \quad 1 \quad 1
\end{align*}
\]

\texttt{ones} is a function that builds a new matrix with the given number of rows and columns, and sets all the elements to 1. Now let’s see what we’ve got.

\[
\begin{align*}
\gg \texttt{whos}
\end{align*}
\]
According to MATLAB, everything is a double array: “double” is another name for double-precision floating-point numbers, and “array” is another name for a matrix.

The only difference is the size, which is specified by the number of rows and columns. The thing we called scalar is, according to MATLAB, a matrix with one row and one column. Our vector is really a matrix with one row and 5 columns. And, of course, matrix is a matrix.

The point of all this is that you can think of your values as scalars, vectors, and matrices, and I think you should, as long as you remember that MATLAB thinks everything is a matrix.

Here’s another example where the error message only makes sense if you know what is happening under the hood:

```
>> X = 1:5
X = 1 2 3 4 5
>> Y = 1:5
Y = 1 2 3 4 5
>> Z = X*Y
??? Error using ==
mtimes
Inner matrix dimensions must agree.
```

First of all, mtimes is the MATLAB function that performs matrix multiplication. The reason the “inner matrix dimensions must agree” is that the way matrix multiplication is defined in linear algebra, the number of rows in \( X \) has to equal the number of columns in \( Y \) (those are the inner dimensions).

If you don’t know linear algebra, this doesn’t make much sense. When you saw \( X*Y \) you probably expected it to multiply each the the elements of \( X \) by the corresponding element of \( Y \) and put the results into a new vector. That operation is called elementwise multiplication, and the operator that performs it is \( .* \):

```
>> X .* Y
ans = 1 4 9 16 25
```

We’ll get back to the elementwise operators later; you can forget about them for now.

### 1.2.5 Indices

You can select elements of a vector with parentheses:
This means that the first element of \( Y \) is 6 and the fifth element is 10. The number in parentheses is called the index because it indicates which element of the vector you want.

The index can be any kind of expression.

Loops and vectors go together like the storm and rain. For example, this loop displays the elements of \( Y \).

```
for i = 1:5
  Y(i)
end
```

Each time through the loop we use a different value of \( i \) as an index into \( Y \).

A limitation of this example is that we had to know the number of elements in \( Y \). We can make it more general by using the \texttt{length} function, which returns the number of elements in a vector:

```
for i = 1:length(Y)
  Y(i)
end
```

There. Now that will work for a vector of any length.

### 1.2.6 Indexing errors

An index can be any kind of expression, but the value of the expression has to be a positive integer, and it has to be less than or equal to the length of the vector. If it’s zero or negative, you get this:

```
>> Y(0)
??? Subscript indices must either be real positive integers or logicals.
```

“Subscript indices” is MATLAB’s longfangled way to say “indices.” “Real positive integers” means that complex numbers are out. And you can forget about “logicals” for now.
If the index is too big, you get this:

```
>> Y(6)
??? Index exceeds matrix dimensions.
```

There’s the “m” word again, but other than that, this message is pretty clear.

Finally, don’t forget that the index has to be an integer:

```
>> Y(1.5)
??? Subscript indices must either be real positive integers or
    logicals.
```

### 1.2.7 Vectors and Sequences

Vectors and sequences go together like ice cream and apple pie. For example, you can evaluate the Fibonacci sequence by storing successive values in a vector. The definition of the Fibonacci sequence is $F_1 = 1$, $F_2 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 3$. In MATLAB, that looks like

```
F(1) = 1
F(2) = 1
for i=3:n
    F(i) = F(i-1) + F(i-2)
end
ans = F(n)
```

Notice that I am using a capital letter for the vector $F$ and lower-case letters for the scalars $i$ and $n$. At the end, the script extracts the final element of $F$ and stores it in `ans`, since the result of this script is supposed to be the $n$th Fibonacci number, not the whole sequence. The only drawbacks are

- You have to be careful with the range of the loop. In this version, the loop runs from 3 to $n$, and each time we assign a value to the $i$th element. It would also work to “shift” the index over by two, running the loop from 1 to $n-2$:

```
F(1) = 1
F(2) = 1
for i=1:n-2
    F(i+2) = F(i+1) + F(i)
end
ans = F(n)
```

Either version is fine, but you have to choose one approach and be consistent. If you combine elements of both, you will get confused. I prefer the version that has $F(i)$ on the left side of the assignment, so that each time through the loop it assigns the $i$th element.

- If you really only want the $n$th Fibonacci number, then storing the whole sequence wastes some storage space. But if wasting space makes your code easier to write and debug, that’s probably ok.
Exercise 1.1 Write a loop that computes the first \( n \) elements of the geometric sequence \( A_{i+1} = A_i/2 \) with \( A_1 = 1 \). Notice that the math notation puts \( A_{i+1} \) on the left side of the equality. When you translate to MATLAB, you may want to shift the index.

1.2.8 Plotting

`plot` is a versatile function for plotting points and lines on a two-dimensional graph. Unfortunately, it is so versatile that it can be hard to use (and hard to read the documentation!). We will start simple and work our way up.

To plot a single point, type

```matlab
>> plot(1, 2)
```

A Figure Window should appear with a graph and a single, blue dot at \( x \) position 1 and \( y \) position 2. To make the dot more visible, you can specify a different shape:

```matlab
>> plot(1, 2, 'o')
```

The letter in single quotes is a string that specifies how the point should be plotted. You can also specify the color:

```matlab
>> plot(1, 2, 'ro')
```

\( r \) stands for red; the other colors include \texttt{green}, \texttt{blue}, \texttt{cyan}, \texttt{magenta}, \texttt{yellow} and \texttt{black}. Other shapes include \(+\), \(*\), \texttt{x}, \texttt{s} (for square), \texttt{d} (for diamond), and \( ^\) (for a triangle).

When you use `plot` this way, it can only plot one point at a time. If you run `plot` again, it clears the figure before making the new plot. The `hold` command lets you override that behavior. \texttt{hold on} tells MATLAB not to clear the figure when it makes a new plot; \texttt{hold off} returns to the default behavior.

Try this:

```matlab
>> hold on
>> plot(1, 1, 'o')
>> plot(2, 2, 'o')
```

You should see a figure with two points. MATLAB scales the plot automatically so that the axes run from the lowest value in the plot to the highest. So in this example the points are plotted in the corners.

1.2.9 Plotting Vectors

Plotting and vectors go together like the moon and June, whatever that means. If you call `plot` with a single vector as an argument, MATLAB plots the indices on the \( x \)-axis and the elements on the \( y \)-axis. To plot the Fibonacci numbers we computed in the previous section:

```matlab
plot(F)
```
1.2 Scalars, Vectors and Matrices (Allen Downey) 9

This display is often useful for debugging, especially if your vectors are big enough that displaying the elements on the screen is unwieldy.

If you call `plot` with two vectors as arguments, MATLAB plots the second one as a function of the first; that is, it treats the first vector as a sequence of \( x \) values and the second as corresponding \( y \) value and plots a sequence of \((x, y)\) points.

```matlab
X = 1:5
Y = 6:10
plot(X, Y)
```

By default, MATLAB draws a blue line, but you can override that setting with the same kind of string we saw in Section 1.2.8. For example, the string `'ro-'` tells MATLAB to plot a red circle at each data point; the hyphen means the points should be connected with a line.

In this example, I stuck with the convention of naming the first argument \( X \) (since it is plotted on the \( x \)-axis) and the second \( Y \). There is nothing special about these names; you could just as well plot \( X \) as a function of \( Y \). MATLAB always treats the first vector as the “independent” variable, and the second as the “dependent” variable (if those terms are familiar to you).

1.2.10 Reduce

A frequent use of loops is to run through the elements of an array and add them up, or multiply them together, or compute the sum of their squares, etc. This kind of operation is called `reduce`, because it reduces a vector with multiple elements down to a single scalar.

For example, this loop adds up the elements of a vector named \( X \) (which we assume has been defined).

```matlab
total = 0
for i=1:length(X)
    total = total + X(i)
end
ans = total
```

We use the `length` function to find the upper bound of the range, so this loop will work regardless of the length of \( X \). Each time through the loop, we add in the \( i \)th element of \( X \), so at the end of the loop \( \text{total} \) contains the sum of the elements.

**Exercise 1.2** Write a similar loop that multiplies all the elements of a vector together. You might want to call the accumulator `product`, and you might want to think about the initial value you give it before the loop.

1.2.11 Apply

Another common use of a loop is to run through the elements of a vector, perform some operation on the elements, and create a new vector with the results. This kind of operation is called `apply`, because you apply the operation to each element in the vector.
For example, the following loop computes a vector \( Y \) that contains the squares of the elements of \( X \) (assuming, again, that \( X \) is already defined).

```matlab
for i = 1:length(X)
    Y(i) = X(i)^2
end
```

**Exercise 1.3** Write a loop that computes a vector \( Y \) that contains the sines of the elements of \( X \). To test your loop, write a script that

1. Uses `linspace` (see the documentation) to assign to \( X \) a vector with 100 elements running from 0 to \( 2\pi \).
2. Uses your loop to store the sines in \( Y \).
3. Plots the elements of \( Y \) as a function of the elements of \( X \).

### 1.2.12 Spoiling the fun

Experienced MATLAB programmers would never write the kind of loops in this chapter, because MATLAB provides simpler and faster ways to perform many reduce, filter and search operations. For example, the `sum` function computes the sum of the elements in a vector and `prod` computes the product.

Many apply operations can be done with elementwise operators. The following statement is more concise than the loop in Section 1.2.11

\[
Y = X .^ 2
\]

Also, most built-in MATLAB functions work with vectors:

```matlab
X = linspace(0, 2*pi)
Y = sin(X)
plot(X, Y)
```

Finally, the `find` function can perform search operations, but understanding it requires a couple of concepts we haven’t got to, so for now you are better off on your own.

I started with simple loops because I wanted to demonstrate the basic concepts and give you a chance to practice. At some point you will probably have to write a loop for which there is no MATLAB shortcut, but you have to work your way up from somewhere.

If you understand loops and you are are comfortable with the shortcuts, feel free to use them! Otherwise, you can always write out the loop.

**Exercise 1.4** Write an expression that computes the sum of the squares of the elements of a vector.
1.3 Example of Apply: Function Evaluation

Allen Downey showed examples of an **apply** operation on a vector and how MATLAB will often work directly on vectors in this way. One handy way to use this functionality is to evaluate a mathematical function over a range of values. For example, say that you have a model for the relationship between the input force \( F \) on a cantilevered beam and the output strain \( \epsilon \) as expressed by the equation

\[
\epsilon = \frac{F(x - l)(h/2)}{EI}
\]

where \( x, l, h, E, \) and \( I \) are known constants. Suppose we want to evaluate the strain for a number of input force values, for example we want to know the strain for force values from 0–10 Newtons.

### 1.3.1 Using a Loop

One way to accomplish this would be to setup a loop using a **for** statement. Each time through the loop the function is evaluated, the values are plotted and the input force is incremented.

```
% Function Evaluation Example: Loop
x = 0.05; % Location of strain gage [m]
l = 0.25; % Length of beam [m]
E = 68.9e9; % Modulus of elasticity of aluminum [Pa]
I = 8.3e−12; % Second moment of area [m^4]
h = 0.0016; % Height of beam [m]
```
1.3.2 Using an Apply Operation

If we understand the MATLAB considers everything to be a vector and that we can use MATLAB to naturally and efficiently work with vectors, it makes it much easier to evaluate our function. We can eliminate the loop completely and simply evaluate the function were the force variable is a vector.

Listing 1.2: Evaluating a function using vectors.

```
% Function Evaluation Example: Loop
x = 0.05;  % Location of strain gage [m]
l = 0.25;  % Length of beam [m]
E = 68.9e9;  % Modulus of elasticity of aluminum [Pa]
I = 8.3e-12;  % Second moment of area [m^4]
h = 0.0016;  % Height of beam [m]

FF = linspace(0, 10, 11);  % Vector force values [N]
% Evaluate the function for over the vector of input forces.
EPS = (FF*(x-l)*(h/2))/(E*I);

% Setup a figure window
figure(2);
clf;
plot(FF, EPS, 'ro')
```

You should note that the FF variable is a 1x11 vector and the calculated output variable EPS is also a 1x11 vector.

1.4 Example of Reduce: Time-Average

One thing we’ll need to do in the second lab exercise is to evaluate the average of a time-series. This is a good example of a reduce operation because we will take a vector and reduce it to a scalar.

For our example we have a time-series of voltage measurements representing 1 second of measurements at 100 Hz. These measurements are stored in a vector called Voltage which is of size 100x1.

```
% Function Evaluation Example: Loop
x = 0.05;  % Location of strain gage [m]
l = 0.25;  % Length of beam [m]
E = 68.9e9;  % Modulus of elasticity of aluminum [Pa]
I = 8.3e-12;  % Second moment of area [m^4]
h = 0.0016;  % Height of beam [m]

FF = linspace(0, 10, 11);  % Vector force values [N]
% Evaluate the function for over the vector of input forces.
EPS = (FF*(x-l)*(h/2))/(E*I);

% Setup a figure window
figure(2);
clf;
plot(FF, EPS, 'ro')
```
You can determine this size and shape of a matrix, vector or scalar by using the `size` function, e.g.,

\[
\text{size(Voltage)} \\
\text{ans} = \\
100 \quad 1
\]

Now if we want to calculate the average of the vector we can simply use the `mean` function which will return a scalar value that is the time-averaged voltage.

\[
\text{avgVoltage} = \text{mean(Voltage)};
\]

Similar operators that you might be interested in are `median`, `std`, `min`, `max`.

### 1.5 Representing Polynomials with Vectors

MATLAB even uses vectors to represent polynomials. For example, if we want to represent a linear relationship between \(x\) and \(y\) we might use the equation for a line

\[
y = mx + b
\]

where \(m\) and \(b\) are constants. To represent this first-order polynomial in MATLAB we use a vector containing the two coefficients—the scalars \(m\) and \(b\).

\[
P = [m \ b];
\]

Of course this only makes sense if \(m\) and \(b\) are variables with numeric values.

As a more concrete example, suppose we know the slope of the line \((m = 3.14)\) and the \(y\)-intercept \((b = 1.0)\). Now we can represent the polynomial with the two-element vector \(P=[3.14 \ 1.0]\); To put this representation to work, we can use the `polyval` function to evaluate the polynomial for any value of \(x\).

\[
P = [3.14 \ 1.0]; \\
x = 1; \\
y = \text{polyval}(P, x)
\]

\[
y = 4.14
\]

We can combine this with MATLAB’s ability to work with vectors so that we can evaluate our polynomial over all the elements in a vector \(X\)

\[
P = [3.14 \ 1.0]; \\
X = 0:2; \\
Y = \text{polyval}(P, X)
\]

\[
Y = \\
1.0000 \quad 4.1400 \quad 7.2800
\]

Notice that again we put in a vector \(X\) and we get out a vector of the same size \(Y\).
1.6 Example: Fitting a Curve to Experimental Data

Now we will use the polynomial notation of MATLAB along with the function \texttt{polyfit} to fit a line to a set of measurements. The \texttt{polyfit} function takes in two vectors—one for the X data (force inputs in this case) and one for the Y data (strain measurements in this case). The function returns a polynomial that best fits the data. We can specify the degree of the polynomial. For this example we will tell MATLAB to fit a line to the data; a line is a polynomial of degree one \((N = 1)\). The line (AKA polynomial) is represented as a two-element vector as described in the previous section.

\begin{verbatim}
\% Linear Regression Example
\% The "data" below is the fictitious measurements we
\% are going to fit with a line.
\% Forces = [1, 2, 4.5, 7.2]; \% Tip force vector [N]
\% Strains = [0.0012, 0.0019, 0.0048, 0.0070]; \% Strain vector [m/m]

\% Plot the measurements
figure(1); clf();
plot(Forces, Strains, 'o')
xlabel('Force [N]')
ylabel('Strain [m/m]

\% Fit a line (order 1) to the data
P = polyfit(Forces, Strains, 1);
\% Now evaluate the model at the known x points (this is our model
\% prediction)
ForceVec = linspace(0,10,11);
StrainLinear = polyval(P, ForceVec);

\% Add the model to our plot
hold on
plot(ForceVec, StrainLinear, 'r--')

\% Quantify the "goodness of fit" using the R-squared value
Rmatrix = corrcoef(Forces, Strains);
Rsquared = Rmatrix(2,1)^2;

\% Add these evaluations of the model to the plot
title('Linear Fit Example');
legend('Measurements', sprintf('Model: \textbackslash R^2=\%.2f', Rsquared), 'location', 'northwest');
\end{verbatim}

The example above puts this all together. The vector \(P\) resulting from this curve is

\[
P =
\begin{bmatrix}
1.0e-003 & \\
0.9642 & 0.1814
\end{bmatrix}
\]

which is equivalent to the line

\[y = 0.0009642x + 0.0001814\]
or

\[ \epsilon = 0.0009642F + 0.0001814 \]

where $\epsilon$ is the strain and $F$ is the tip force.