Student's Guide to Physical Modeling with MATLAB

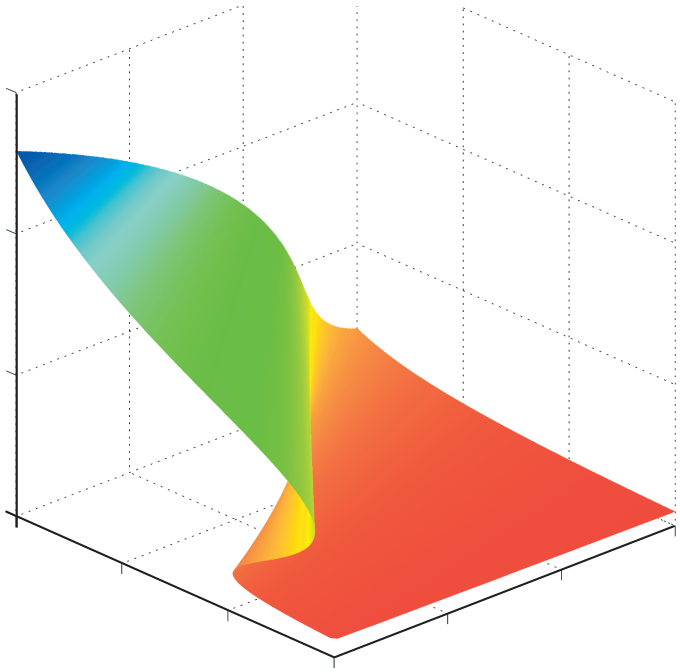
Philip Nelson with Tom Dodson

Version 1.0, Updated on November 25, 2014

This tutorial aims to help you teach yourself enough of the MATLAB® programming language to get started on physical modeling, and particularly the problems appearing in *Physical Models of Living Systems* (Nelson, 2015). This is not an official publication of The MathWorks, Inc. We attempt to maintain it, but no claim is made that every suggestion made here will work properly with future versions of MATLAB.

This is a free online document. If you'd prefer a nice hard copy, you can get one from `lulu.com`. Code listings that appear in this document, errata, and more can be found online via `http://www.physics.upenn.edu/biophys/PMLS/Student`.

A companion to this tutorial covers similar techniques, but with the Python programming language (Kinder & Nelson, 2015).



Contents

CHAPTER 1

Getting Started with MATLAB

The Analytical Engine weaves algebraical patterns, just as the Jacquard loom weaves flowers and leaves.

— Ada, Countess of Lovelace, 1815–1853

The goal of this tutorial is to get you started with the computer math package MATLAB. Many excellent introductions exist, and more are written every year. *This* one is distinguished mainly by the fact that it tries to stick with skills specifically useful to solving problems arising in physical modeling, and specifically those that appear in the book *Physical Models of Living Systems* (Nelson, 2015).

A few sections are flagged with this "Track 2" symbol: T2. These are more advanced and can be skipped if you don't need that topic.

1.1 ALGORITHMS AND ALGORITHMIC THINKING

1.1.1 Algorithmic thinking

Suppose that you need to instruct a friend how to back your car out of your driveway. Your friend has never driven a car, but it's an emergency, and your only communication channel is a phone conversation before the operation begins.

You need to break the required task down into small, explicit steps that your friend understands and can execute in sequence. For example, you might say

```
1 Put the key in the ignition.
2 Turn the key until the car starts, then let go.
3 Push the button on the shift lever and move it to "Reverse."
4 ...
```

Unfortunately, for many cars this "code" won't work, even if your friend understands each instruction: It contains a **bug**. Before step 2, many cars require that the driver

```
Place right foot on left pedal and push down.
```

Also, the shift is usually marked `R`, not `Reverse`, and so on. It is difficult at first to get used to the very high degree of explicitness needed when composing instructions like these.

Also, since you are giving the instructions in advance (your friend has no mobile phone), it's wise to allow for contingencies:

```
If a crunching sound is heard, immediately place right food on left pedal and push down...
```

Breaking the steps of a long operation down into small, ultra-explicit substeps, and anticipating contingencies, are the beginning of learning *algorithmic thinking.*

If your friend has had a lot of experience watching people drive cars, then the instructions above may be sufficient. But a friend from Mars, say, or a robot, would need much more detail. For example, the first two steps may need to be expanded to something like

```
Insert the pointed end of the key into the slot on lower right side of the
        steering column.
Rotate the key about its long axis in the clockwise direction (when
        viewed from the wide end toward the pointed end)...
```

A "low-level" computer programming system requires instructions analogous to these last ones.[1] A "high-level" system comes preprogrammed to understand many everyday operations, and therefore can be instructed in a more condensed style, as in the first example. MATLAB is a high-level language, because it knows about many operations commonly needed when performing mathematical calculations, processing text, manipulating files, making graphs, and image processing.

1.1.2 States

You may be familiar with multistep mathematical proofs. The goal of such a narrative is to establish the truth of a desired, unobvious conclusion by sequentially appealing to given information and a formal system. Thus, each statement's truth, although not evident in isolation, is supposed to be straightforward in light of the preceding statements. The reader's "state" (list of propositions known to be true) grows while reading through the proof.

An algorithm has a very different goal. A chain of instructions is to be followed, each of which performs an easily specified operation, with the goal of accomplishing a not-easily performed task. The chain may involve a lot of repetition, so you don't want to supervise the execution of every step. Instead, you wish to specify all the steps in advance, then stand back while your electronic assistant performs them rapidly. There may also be contingencies that cannot be precisely known in advance (`If a crunching sound is heard, ...`).

In an algorithm, the *computer* has a state that is constantly being modified. For example, it has many memory cells, whose contents change during the course of any operation. Your goal may be to arrange for one or more of them to contain the result of some complex calculation, once the algorithm has finished running. (You may also want a particular graphical image to appear.)

1.1.3 What does `a = a + 1` mean?

To put the preceding distinction into sharp relief, note that most computer math systems, including MATLAB, accept lines of input such as these:

```
a = 1
a = a + 1
```

In mathematics, this makes no sense. More precisely, the second line is an assertion that is always false; equivalently, it is an equation with no solution. To MATLAB, however, these lines have the following meaning:

1. Determine whether any memory cell is currently assigned the name `a`. If not, assign this name to a currently unused memory cell. Either way, *discard* whatever value was previously stored in that cell, and store the value 1 in it.
2. Extract the value stored in the memory call named `a`. Calculate the sum of that value plus 1. Store the resulting value in the memory cell named `a`, *discarding* its previous value.

In other words, the equals sign is a verb instructing MATLAB to change its *state.* In contrast, mathematical notation uses the equals sign to create a proposition, which may be true or false. Note, too, that MATLAB treats the left and right sides of the command `x = y` very differently, whereas in math the equals sign is symmetric. For example, MATLAB will give an error message if you say something like `b + 1 = a`; the object on the left side of an assignment must be the name of a memory cell, able to accept a value.

Actually, we often do wish to evaluate whether a named memory cell (or "variable") has a particular value or not. To avoid ambiguity, MATLAB uses a different symbol for this operation, the double equals sign:

[1] "Assembly language" is an example of a low-level system. "Firmware" is lower still.

```
a = 1;
b = (a==1);
```

The above code again sets up a variable `a` and assigns it a numerical value. Then it sets up a second variable `b`, and assigns it a logical value: When the code has completed, the value of `b` is also 1, which is MATLAB's representation of TRUE. That value can be used in contingent code, as we'll see later.

Mistakenly using = when == is desired is one of the top coding errors!

You can get very mysterious results if you make this error, because both = and == are legitimate MATLAB syntax. In any particular situation, however, one of them is not what you want.

1.1.4 Symbolic versus numerical

In math, it's perfectly reasonable to start a derivation by saying "Let $b = a^2 - a$," even if the reader doesn't yet know the value of a. It's understood that this is a generic statement of a relation defining b in terms of a, whatever the value of a turns out to be.

If we launch MATLAB and immediately give the same statement, `b = a^2 - a`, however, the result is an error message. Every time you hit `<Return/Enter>` (or later, the (RUN) button), MATLAB tries to calculate values for everything in every assignment. Because the variable `a` has not been assigned a value yet, evaluation fails and MATLAB complains. Other computer math packages can accept such input, remember it in the symbolic form given, and manipulate it later, but basic MATLAB does not.[2]

In math, it's also understood that a definition like "Let $b = a^2 - a$" will persist unchanged throughout the discussion. If we then say "In the case $a = 1, \ldots$" then the reader knows that b equals zero; if later we say "In the case $a = 2, \ldots$" then we need not reiterate the definition of b for the reader to know that this symbol now represents the value $2^2 - 2 = 2$.

In contrast, a numerical system like MATLAB *forgets* the relation between `b` and `a` immediately after executing the assignment `b = a^2 - a`. All that it remembers is the *value* now stored in `b`. If we later change the value of `a`, the value of `b` will *not* change.

Similarly, if `a` has a value and we say `b = a^2 - a`, then nothing stops us from later saying `b = exp(a)`. The second assignment updates MATLAB's state by discarding the value calculated in the first one and substituting the newly computed value.

1.2 LAUNCH MATLAB

1.2.1 Command Window

From now on, you must actually have MATLAB running as you read, try every snippet of code given here, and observe what MATLAB does in response.

> *Reading this tutorial won't teach you* MATLAB. *You can teach* yourself MATLAB *by working through all the examples and exercises here, and then using what you've learned on your own problems.*

If you are reading an electronic version of this tutorial, you may be tempted to just cut and paste the code into MATLAB. But then you may find that extraneous blank spaces, bad characters, and other such things appear. It's safer to retype the examples yourself, and anyway, you'll learn faster that way.

This tutorial also won't show you any graphics. You will create them yourself as you work through the examples.

[2] MATLAB does have a symbolic manipulation subsystem, but we won't be using it.

Consult your instructor about ways to obtain MATLAB at your institution, and how to install and launch it.[3] MATLAB has many extensions called "toolboxes"; none of these will be needed while working through this tutorial, nor when solving problems that appear in *Physical Models*.

Upon launch, MATLAB opens a complicated, multipanel display. Things you type will show up in the main panel ("Command Window") after the >> symbol (the "Command Window promptCommand Window!prompt"). Try typing the lines of code given above, hitting `<Return/Enter>` after each line. MATLAB responds immediately after each `<Return/Enter>`, attempting to perform whatever command you entered.[4]

MATLAB *code consists entirely of unformatted ("plain") text.*

Any formatting shown in this tutorial, including fonts and coloring, was added for readability, and is not something you need to worry about while entering code. Similarly, the tiny line numbers shown on the left are just there to let us refer to particular lines; you don't type them. MATLAB will assign and show line numbers for you when you work in the Editor Window, and will use them to tell you where it thinks you have made errors, but they are not part of the code itself.

In particular, this tutorial will show predefined names in bold type, to distinguish them from user-defined variables and functions, but you don't need to type that.

Now notice the window on the upper right (the "Workspace Window"). Each time you enter a command and hit `<Return/Enter>`, this window may change, reflecting any changes in MATLAB's state: Initially empty, it later shows a list of all named memory cells (variables), and a summary of their values. Later, when we study variables with many values (arrays), you'll see that you can double-click any entry in this list; a spreadsheet opens that lets you inspect all the values. You can even copy from this spreadsheet and paste into other applications.

At any time, you can reset MATLAB's state by quitting and relaunching it, or more conveniently by issuing the command

```
clear all
```

Example: Try that. Then try the following commands at the prompt, and explain everything you see happen:

```
q
q == 2
q = 2
q == 2
q == 3
```

Solution: MATLAB complains about the first two lines, because even though it automatically creates a symbol named `q` for you, and assigns it a memory cell, still that cell contains no value, and so expressions involving it cannot yet be evaluated. Changing MATLAB's state in the third line above changes this situation, so the next two lines do not generate errors.

[3] MATLAB is a registered trademark of The MathWorks, Inc. You can also obtain it, or the Student version, directly from `http://www.mathworks.com`.

[4] This tutorial uses the word "command" in a generic sense, to mean any lump of code that could stand on its own. Assignments like `a = 1`, isolated function calls like **`plot`**`(x,y)`, and directives like **`clear all`** are all examples of commands.

Example: Now clear MATLAB's state again:

```
clear all
```

Try the following at the prompt, and explain everything you see happen (it may be useful to refer to Section 1.1.4):

```
a = 1
b = a^2 - a
a = 2
b
b = a^2 - a
```

Solution: The results from the first two lines should be clear: We assign values to the variables `a` and `b`. In the third line, we change the value of `a`, but recall that because MATLAB only remembers the *value* of `b`, that value is unchanged in the fourth line until we update it with the last command.

These examples illustrate another important element of MATLAB syntax: When MATLAB encounters a command that consists of just an expression, and that ends with no punctuation (or with a comma), it will print out the expression's value to the Command Window. When the command is an assignment and ends with no punctuation (or with a comma), MATLAB makes the assignment and also prints out the assigned value. When either of these commands ends with a semicolon, however, MATLAB suppresses automatic printout.

> *Often you will forget a semicolon somewhere in the middle of a long code, sometimes resulting in an enormous amount of text appearing on your screen.*

You can end a command just by starting a new line, if you want the automatic printout feature. If not, you can end it with a semicolon and then start a new line. Or, if you wish you can end a command with a semicolon or comma and *no* new line, cramming another command on the same line. It's best not to make too much use of that ability, however: Your code may take up fewer lines, and MATLAB won't care, but human readability will suffer.

The opposite situation can be important: You may wish to use a very long command that doesn't fit on one line. For such cases, you can end a line with an ellipsis (type three periods just prior to starting a new line). MATLAB will then continue reading the next line as part of the same command. Try this:

```
q = 1 + ...
2
```

There are some restrictions on where you may break a command, so you may need to experiment a bit with this.

1.2.2 Error messages

You should already have encountered an error message by now. They appear in red in the Command Window, accompanied by a beep. See Section B.1 for some hints about interpreting such messages, and some examples often encountered by beginners.

1.2.3 Sources of help

Suppose that you wish to evaluate the natural logarithm of the number 2. You think there's probably a function called **`log`** in MATLAB, but is it the natural or common log? Get `Help>Documentation` from the MATLAB menu, and type **`log`** into the search box.[5] The first entry states that **`log`** is the one you want, so typing

[5] Equivalently, you can type **`doc log`** at the Command Window prompt.

```
log(2)
```

at the command prompt gives your desired result. To learn more, you could also have clicked the entry `log` in the Help; try it.

Unfortunately, MATLAB is not as friendly if you don't know the name of the command you need. Suppose you really did want the common logarithm. It's not obvious in the Help listing for **`log`** how to get this. So it's useful to know that you can also type

```
help log
```

(or the name of any other command) in the Command Window and this will give you a different, briefer, and sometimes more helpful entry. At the bottom of the resulting text, MATLAB says `"See also log1p, log2, log10..."`; that third entry looks promising, and clicking shows it's what you want. Much, much more help is available online; from any search engine, searching `matlab common logarithm` quickly gives you the command even if you don't know its name in advance.

> *At least in the beginning, a lot of your coding time will be spent using a search engine to get help.*

In addition to searching the whole Web, a more targeted approach is to visit Matlab Central (on the MathWorks website).

1.2.4 Good practice: Keep a diary

As you work through this tutorial, you will hit many little and big roadblocks. How do you label graph axes? What if you want a subscript in a graph axis label? The list is endless. Every time you resolve such a puzzle (or a friend helps you), *make a note of how you did it* in a diary file somewhere on your computer. Later, looking through that diary will be much easier than scanning through all the code you wrote months ago (and less irritating than asking your friend over and over).

1.3 MATLAB EXPRESSIONS

1.3.1 Numbers

You can enter explicit numerical values in various ways:

- `123` and `1.23` mean what you might expect. When entering a very big number, however, don't separate groups of digits by commas (don't say `1,000,000` if you mean a million).
- `2.3e5` is convenient shorthand for $2.3 \cdot 10^5$.
- `2+3i` refers to the complex number $2 + 3\mathrm{i}$.

MATLAB stores numbers internally in several different formats. However, it knows when it needs to convert from one type to another; beginners generally don't need to think about this. Just notice that for some uses, a number must be an integer (whole number), for example, when indicating which entry in a list you want. (If you need to force a number to be an integer, you can use the functions **`round`** or **`floor`**.)

1.3.2 Arithmetic and predefined functions

MATLAB has all the basic arithmetic operators you might expect, for example, `+`, `-`, `*` (multiplication), `/`, and `^` (exponentiation). Note that, unlike standard mathematics notation, you may not omit multiplication signs. Try typing

```
(2)(3)
a = 2; a(3)
3a
3 a
```

Each of these produces a different error message. None, however, generate a message like `You forgot a *`. MATLAB used its evaluation rules, and these expressions didn't make sense, but it doesn't know what you were trying to express, so it can't tell you what exactly is wrong. *Study these four error messages;* you'll probably see them again.

Arithmetic operations have the usual precedence (ordering), which can be overridden by using parentheses. For example, you may want to express the number $\frac{1}{2\pi}$, so you may type `1.0 / 2.0 * pi`. Try it. What goes wrong, and why? Later we'll meet other kinds of operators, for example relations like `<` and logical operations. They, too, have a precedence ordering, which you may not wish to memorize. So use parentheses liberally to avoid unintended meanings.

To get used to MATLAB arithmetic operations, figure out what problem these lines solve, and check that MATLAB got it right:

```
a = 1; b = 2; c = 3;
(-b + sqrt(b^2 - 4*a*c))/(2*a)
```

Notice that **`sqrt`** is the name of a *function* that MATLAB already knows when it launches.[6] When MATLAB encounters this name in the expression above, it

1. Evaluates the "argument," that is, everything inside the following pair of round parentheses (which may itself contain functions);
2. Interrupts handling your evaluation and begins a piece of internal code named **`sqrt`**, handing that code the value found in **`1`**;
3. Substitutes the value returned by **`sqrt`**`(...)` into your expression; and
4. Finishes evaluating your expression as usual.

How do you know what functions are available predefined for you? See Section 1.2.3 above, get the menu item `Help>MATLAB>Mathematics`, or type **`help elfun`** at the Command Window prompt.

A few symbols in MATLAB are predefined but do not require any argument in parentheses. Try **`pi`** (the constant π) and **`i`** (the constant $\sqrt{-1}$).

MATLAB also provides standard trigonometric functions, but notice:

> *The trig functions **`sin`**, **`cos`**, and so on, all expect their arguments to be angles expressed in radians.*

1.3.3 Good practice: Variable names

Note that MATLAB offers you no protection against accidentally changing the value of a symbol like **`pi`** (or indeed, of any variable that you set up): If you say **`pi`** `= 22/7`, then until you change it or reset MATLAB, **`pi`** will have that value.[7]

> *It's especially easy to redefine **`i`**, because we so often use it as an index. If you need the built-in constant equal to $\sqrt{-1}$, use the syntax **`1i`** instead of **`i`**, because **`1i`** cannot be redefined.*

When your code gets long, it also becomes very likely that, if you assign a variable with a generic name like `a` in the beginning, you'll later choose the same name for some completely different purpose. Later still,

[6] We have already met **`log`** and **`log10`**.
[7] It is even possible to create a variable whose name overshadows a built-in function, for example, **`hist`** `= 1`.

you may want the original `a`, having forgotten about the new one. But MATLAB has overwritten the value you wanted. Puzzling behavior may ensue. You have a "name collision."

It's good practice to use longer, more meaningful names for variables. They take longer to type, but they can also improve your code's readability. Perhaps the variable you were planning to call `a` could instead be called `whichItem`, because it indexes a list. Later, the other variable that caused the name collision could logically be called `partialSum` or something such. Later still, when you ask for `whichItem`, there will be no problem.[8]

Blank spaces and periods are not allowed in variable names. Some coders use capitalization in the middle of a variable name to denote word boundaries. Others prefer to use the underscore (`which_item`). Variable names may contain digits (`myCount2`), but they must start with a letter.

1.3.4 Good practice: Units

Most physical quantities carry units, for example, 3 cm. MATLAB doesn't know about units; all its values are pure numbers. If you are trying to code a problem involving a quantity L with dimensions of length, you'll need to represent it by a variable **`length`** whose value equals L *divided by some unit.* That's fine, as long as you are consistent everywhere about what unit to use.

You can make things easier on yourself if you include a block of comments at the head of each code declaring (to yourself and other human readers) the variables you'll use, and the units chosen to make them pure numbers:[9]

```
%% Variables:
%   length = length of the microtubule [um]
%   vel = velocity of motor [um/s]
%   rcon = rate constant [1/days]
...
```

As you work on the code, you can refer back to this section to keep yourself consistent. (The notation `um` is a standard, easy-to-type version of μm.) Such fussy hygiene will save you a lot of confusion someday.

Later on, when you start to get data files, you need to learn from whoever gave a file to you what units are being used to express quantities. Ideally this will be spelled out in a text file (perhaps called `README.txt`) accompanying the data file, or in the opening lines of a spreadsheet.

1.3.5 More about functions

You are accustomed from math to thinking of a function as a machine that eats exactly one number (its "argument") and spits out exactly one number (its result). Some MATLAB functions, like **`sqrt`**, have this character.[10] But MATLAB has a much broader notion of function. Here are some illustrations (some involve functions that we have not formally met yet):

- A function may change MATLAB's state in other ways than by returning a result. For example, **`plot`** opens a window and creates a plot. Other possible "side effects" include writing text into the Command Window, for example, via **`disp`**`('hello')`.
- A function may instead have *two* (or more) arguments, separated by commas. Or it may have none; try just typing **`rand`**`()` at the command prompt.

[8] Variable names are case-sensitive, and MATLAB's predefined names are all lower-case. So you can avoid name collisions with predefined things simply by including one or more capital letters in any user-defined variable or function name.
[9] Section 2.4.3 will give more details about commenting.
[10] Instead of a single number, the argument of **`sqrt`** can be a *list*, but MATLAB still regards that as a single argument. See Section 2.1.

- A function may allow a *variable* number of arguments, and behave differently depending on how many you supply. For example, we will see examples of functions that allow you to specify any number of options as arguments. Each function's Help text will list each allowed way of using it.
- A function may also *return* more than one value. You can capture the returned values by using a special kind of assignment (see Section 5.3). Their number can even be variable; the function may behave differently depending on how many results you requested.

If a function accepts two or more arguments, how does it know which is which? In mathematics notation, the *order* of arguments conveys this information. For example, if we define $f(x, y) = x\mathrm{e}^{-y}$, then later $f(2, 6)$ means $2\mathrm{e}^{-6}$: The first given value (2) gets substituted for the first named variable in the definition (x), and so on. MATLAB uses the same scheme.

CHAPTER 2

Structure and control

Much of the power of computation comes from the machine's ability to do repetitive, or nearly repetitive tasks. You need to understand how to formulate instructions for this sort of task, so that your electronic assistant can do all the steps without your supervision. This chapter describes two key elements:

- Grouping of data into "data structures," such as lists and arrays, and
- Grouping of code into repeated blocks (loops) and contingent blocks (conditional code).

2.1 LISTS AND ARRAYS

2.1.1 Setting up a list

Much of the power of computer programming comes when we handle numbers in batches. A batch of numbers can be a single mathematical object, like the components of a force vector, or a list representing many objects of the same type, like the values at which you wish to evaluate some function. MATLAB doesn't care about such distinctions. It regards every such object as a rectangular grid (an "array"), like the cells of a spreadsheet.[1] You can create one by using the command

```
a = zeros(3,5)
```

The function **zeros** accepts two arguments and creates an array with, in this case, three rows and five columns. Then it gives all 15 entries the same value (zero). Find `a` in the Workspace Window and see how it appears. Double-click it to look inside it. Repeat with the function **ones** in place of **zeros** and see what you get. Finally, try **eye**`(3)`.

A list, or "row vector," is the special case of an array with just one row: **zeros**`(1,5)`. Similarly, a "column vector" is an array with just one column: **zeros**`(3,1)`.

> *Another common hang-up: If you want a row of N zeros, you need* ***zeros****`(1,N)`, not* ***zeros****`(N)`. Try the second of these to see what it does give. Some other array-valued functions, such as* **ones** *and* **rand***, have similar behavior.*

MATLAB can report how big an array is: After setting up `a`, use the command **size**`(a)` and see what you get. Also try **size**`(a,1)` and **size**`(a,2)` to find out separately the number of rows and columns, respectively, in `a`. This information is also displayed in the Workspace Window.

Perhaps you'd like to set up an array with more interesting values. The syntax `a = [2.71, 3.14, 3000]` creates an array with one row and three columns. Changing commas to semicolons gives an array with three rows and one column.

Your Turn 2A

Try
```
a = [2, 3, 5; 7, 11, 13]
```
and explain the result.

[1] MATLAB also uses the word "matrix" to mean an array whose entries are numbers.

Sometimes you'd rather not specify each entry explicitly. The notation `(N:M)` creates a row vector with the value of `N` as its first entry, `N + 1` as the second, and so on, until it reaches a value that exceeds `M`. Try `a = (2.1:5.4)` and explain what you get. The notation `a = (2.1:0.1:5.4)` also creates a series, but in this case each entry is incremented by 0.1, not the default 1, from its predecessor.

> *The colon construction's syntax is `startvalue:increment:endvalue`.*
> *The middle entry can be omitted if it equals 1.*

Type **`help colon`** at the prompt to learn more.

The colon construction has a behavior that may surprise you: Type `1:3 + 20` and see what you get. If you're surprised, add some parentheses until you get what you expect, and remember:

> *Extra parentheses do no harm, so when in doubt, either check or just put them in to force the operator ordering you want.*

The function **`linspace`**`(A, B, C)` does a similar job, but slightly differently. It always creates a row vector with exactly `C` entries, evenly spaced. The first entry equals `A`. Unlike the colon notation, however, the last entry exactly equals `C`. But you don't get to specify the exact spacing; MATLAB chooses that to be whatever is needed to get the required starting and ending values.

Your Turn 2B

Try

```
a = 0:2:10
b = linspace(0,10,6)
```

and explain the results. You should see that `a` and `b` are equivalent. Now try

```
a = 0:(1.5):10
b = linspace(0,10,7)
```

and note that `a` and `b` are not equivalent. Explain why not, and decide which form you should use if you want to evaluate a function over the range 0 to 10.

When creating a series to compute the values of a function over a range, **`linspace`** is the most appropriate function, as it allows you to explicitly choose the number of points in the series and the start and end of the range. However, when the exact spacing between points is important, the colon operator is more appropriate (see, for example, Section 3.2.3).

2.1.1.1 Concatenation

The square bracket construction can also build up an array from smaller arrays. Try these examples:

```
a = [1, 10:15]
b = [a; 100:106]
```

2.1.2 Slicing

Once you have set up an array, each of its entries can be accessed individually. Try

```
a = [2, 3, 5; 7, 11, 13];
a(2,3) = 1000
```

The second command changes just one entry in the array. That entry is specified in the same convention as the one mathematicians use for matrices:[2] `a(k, j)` is the entry at the intersection of row `k` and column `j`. (A mathematics text would call it a_{kj}.) If `a` has only one row, you can say `a(k)` as an abbreviation for `a(1, k)`. If `a` has only one column, you can say `a(j)` as an abbreviation for `a(j, 1)`.

It can get confusing that MATLAB uses the same notation (round parentheses) for two very different purposes:

- When following a function name, they enclose the arguments(s).
- When following an array name, they enclose the index (or multiple indices) that specify an individual element.

This double meaning doesn't confuse MATLAB, because it knows from context what to do.

Often you will wish to extract more than one element from an array. Suppose that you have been given experimental data in an array `a` with two columns and 20 rows. In order to make a graph, you need an array with just the first column of `a`, and another one with just the second:

```
b = a(1:20, 1);
c = a(1:20, 2);
```

The colon construction sets up a list; using a list in place of an array index forms a new array by running through the members of that list. If you don't know the number of entries in `a`, you could start out by finding it:

```
len = size(a, 1)
b = a(1:len, 1);
c = a(1:len, 2);
```

But MATLAB has a shortcut for this very common operation: A colon all by itself represents every allowed value of an index, for example, `b = a(:, 1)`. You can also say `b = a(2:end, 1)` to extract the entire first column except for the first entry; the keyword **`end`** refers to the last allowed value.

Your Turn 2C

Try

```
a = [1:20; 101:120]'
b = a(1:(end-1), 2)
```

and explain the output you get from each line. In this code, the apostrophe operator (transpose) appended to an array generates its matrix transpose (it exchanges rows and columns).[3]

2.1.2.1 T2 Lists as indices

As mentioned earlier, you can use a list where an index is expected, to address a set of array entries given by the elements of that list. Try

```
a = -10:10;
b = [2, 4, 5];
a(b)
```

[2] If this seems like the obvious convention, be aware that in some other computer languages the first entry of an array has index *zero*.

[3] More precisely, `a'` requests the Hermitian conjugate of `a`, which equals the transpose if the entries of `a` are all real numbers. If `a` is complex, and you really want the ordinary transpose, use the function `transpose(a)`.

This method can be extremely useful if the list `b` was itself generated automatically. For example, you could use the **`find`** function to get the locations of all the nonnegative entries in a list `a`, then use those locations as an index to generate a new list, with the negative entries deleted (see **`help find`**):

```
aNonNeg = a(find(a >= 0))
```

An alternative approach is even more concise: You can also use an array of true/false entries as an index for an array, to generate a new array with just the entries corresponding to TRUE entries in the index. (The list must be the same size and shape as the matrix being indexed.) With this addressing, the preceding example becomes:

```
aNonNeg = a(a >= 0)
```

2.1.3 Flattening an array

An array is a list with more than one dimension. You may wish to repackage all its values as an ordinary list. Try the following syntax, and note how it accomplishes that goal ("flattening"):

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

For example, you can find the average of all the elements of `A` with the command **`mean`**`(A(:))`.

2.2 STRINGS

MATLAB can manipulate other types of information besides numbers. The second most important type is the "string." A string variable contains any number of keyboard characters. You can create one with

```
a = 'Hello world.'
```

Notice how `a` looks in the Workspace Window. The expression on the right side of the equals sign above is an explicit string value (that is, a **string literal**). The equals sign assigns this short sentence (without the single quotes) as the value of `a`.

*A string literal starts and ends with a single **right quote**.*

Note that "single right quote" is the same key on your keyboard as "apostrophe."[4] If you actually need a single right quote (or apostrophe) inside the string, type it twice: `'It''s about time'`.

A string may contain a collection of digits that looks to us like a number, for example, `a = '123'`. It's important to note that MATLAB *still considers such a value to be a string,* and therefore quite different from the number 123. Try typing

```
a = '123'
b = a + 1
```

MATLAB offers you no protection against this sort of error, because the operation of adding a string to a number *is* defined; it's just not very likely to be what you actually want.

One useful operation that you can do with strings is to join ("concatenate") them:

```
a = 'Hello world.'
b = 'I am MATLAB.'
c = [a, b]
```

[4] Elsewhere on your keyboard there's a different key, the "single left quote" (or "grave accent"). You may be tempted to use it, for example, writing `` `string' ``. MATLAB won't understand that.

The last line above shows that MATLAB regards a string similarly to an array: Applying this same syntax to two row vectors creates a new row vector by appending the second to the end of the first, and similarly with strings as above.

Your Turn 2D

Something isn't completely nice about the result. Replace the last line with `c = [a, ' ', b]` and explain the point.

Some MATLAB functions require arguments that are strings, for example, graph titles (see Section 3.2). But you may wish to include the current value of some numerical variable in a graph title. To accomplish this, you need to *convert* a numeric value to its corresponding string, *as if* you were printing it out. The handy function **num2str** ("number to string") accomplishes this:

```
title(['Poisson distribution for \mu = ', num2str(a)])
```

- The command **title** takes a single string argument and places it as the title of the current graph (see Section 3.2.4).
- The brackets join two strings as above.
- The first of these is a literal. It contains a special character, the Greek letter μ, expressed as `\mu`. The **title** function knows how to interpret such codes.[5]
- The second string is obtained from the current value of a variable that we suppose you have called `a`.

You can give **num2str** an optional second argument, the number of significant digits you wish to show, if you don't like the default choice: **num2str**`(a,2)` will look nicer in a graph title than **num2str**`(a)`.

2.3 LOOPS

2.3.1 `for` and `while`

So far, we have described MATLAB as a glorified calculator. Yes, it can evaluate the solutions to a quadratic equation. But what if you want to graph the solution, say as a function of `a` for fixed `b` and `c`? The first step is to calculate a lot of solutions. One way is with a *loop*. Try typing

```
b = 2; c = 3;
for a = -1:0.5:2, (-b + sqrt(b^2 - 4*a*c))/(2*a), end;
```

The keyword **for** instructs MATLAB to perform a block of code repeatedly:

1. The notation `-1:0.5:2` indicates a series of values, starting with the first entry (here -1), increasing by steps equal to the second entry (-1, -0.5, 0, 0.5...), and not to exceed the last value (here 2) (see Section 2.1.1).
2. Thus, MATLAB initially assigns `a` the value -1. Then it evaluates the following expression and prints the result (because the expression is terminated without a semicolon).
3. The keyword **end** tells MATLAB to move back to the nearest incomplete **for** command, update the value of `a`, and determine whether it exceeds the upper limit (here, 2). If not, then the body of the loop is evaluated *again,* and so on. Eventually, however, `a` exceeds 2; then MATLAB jumps to whatever comes after the **end** (in this case, nothing). We say it "exits the loop."

Note that the counter `a` is an ordinary variable, whose value can be accessed in calculations. However, it's not a good idea to modify its value inside the loop; leave that to the **for** command.

Unfortunately, MATLAB won't complain at all if you type something like

[5] T2 For more complex math notation in a label, MATLAB can recognize a subset of the LaTeX typesetting language.

```
for x = -1, 2
x^2, end;
```

It will just do something that is probably not what you intended!

> *Forgetting to use the colon syntax in a* **for** *loop is another top coding error.*

A more general form of loop is often useful. You may wish to repeat a block of code as long as some general condition holds, but terminate it the first time that condition fails to be true. For example, you may only want solutions to the quadratic equation when they are real. Try

```
b = 2; c = 3; a = -1
while (b^2-4*a*c >= 0), (-b + sqrt(b^2-4*a*c))/(2*a), a = a + 0.5, end; disp('done!')
```

Notice that this time, we instructed our code to tell us the value of `a` for each solution, because we don't know in advance which values will be used. Note, too, that we now need to increment `a` explicitly before ending the loop, because **while** doesn't do that for us.

Each **for** or **while** keyword must be terminated by an **end**. The **end** tells MATLAB when to go back; it also tells MATLAB at what point to pick up again after the loop has completed. For example, the last code fragment above prints out a little message after finishing the loop. (The **disp** command takes a string argument and prints, or "displays," it in the Command Window.)

2.3.2 Vectorizing math

One reason to use arrays is that MATLAB has a very concise syntax for handling repetitive operations on them. For example, the preceding code can be replaced by

```
b = 2; c = 3;
a = (-1:0.5:2);
(-b + sqrt(b^2 - 4*a*c))./(2*a)
```

The second line creates an array `a` and fills it with values at which we'd like to evaluate the expression. In the third line, MATLAB carries out the following steps:

1. It starts with the innermost subexpression, first calculating b^2 and saving the result. In the second term, it notices that the array `a` is to be multiplied by 4. MATLAB interprets this request as meaning that you want to multiply *each element separately* by 4. This is the standard interpretation in math: When you multiply a force vector, say, by 4, each of its components gets that operation done separately.
2. It finishes evaluating `4*a*c`. Now it sees that you've asked it to combine this result with the single quantity `b^2`. *Unlike* standard math notation, MATLAB interprets this as a request to set up a new vector, whose kth entry equals `b^2 - 4*a(k)*c`. That is, subtraction of an array from a single value is done item-by-item, as in step **1**.
3. It then feeds the array generated in step **2** to the **sqrt** function as an argument. **sqrt** normally accepts a single number, but like many MATLAB functions (**sin**, **cos**, **exp**, ...), it has a standard behavior when you instead feed it an array: It again acts item-by-item (that is, performs its usual behavior with each entry individually), and creates a new list with the results.[6]
4. Subtracting `b` follows the rule already given in **2**.
5. The final division by `2*a` will be discussed in a moment.

We say that the code just given is a "vectorized" form of the earlier one. One advantage of using vectorized code is that it often runs much faster than equivalent code written with explicit loops. This is because

[6] If you ever actually want to find the square root of a matrix, in the sense of linear algebra, that's not MATLAB's default behavior. Look in the Help.

MATLAB knows how to perform multiple operations in parallel on your computer's multiple processors, if you express your calculation in vectorized form.

Another reason to use vectorized code is that, because it is so concise, it can help you to write better code. Long, rambling code can be hard to read, making it hard to spot bugs. Of course, inscrutably dense code is also hard to read. Your coding style will evolve as you get more experienced. Certainly if a code runs fast enough with explicit loops, then there's no need to go back and vectorize it.

Not every math operation behaves item-by-item. Suppose that you wanted to graph the function $y = x^2$. You could set up an array of x values by `x = (1:20)`. But when you try `y = x^2`, you get an error (and `y = x*x` doesn't work either). By default, MATLAB's multiplication, division, and power operators operate on arrays using the rules of matrix math, which isn't defined for the operation you (unwittingly) requested. But you can override this default behavior by prefixing these operations with a period, for example, `x.*y`, `x./y`, or `x.^y`. Thus, to square each element in `x` you could write `y = x.^2`, or equivalently `y = x.*x`.

> *Forgetting the period prefix is another common* MATLAB *coding error, made by amateurs and experts alike.*

A particularly puzzling gotcha arises when you forget the dot in an expression of the form `list1/list2`. MATLAB will not complain about this expression; instead, it will compute something that probably isn't what you wanted.

We can now finish explaining the example at the start of this section. The variable `a` is a list; therefore `-b + sqrt(b^2 - 4*a*c)` and `2*a` are lists too, as explained earlier. To divide each item in the first of these lists by the corresponding item in the second one, we need the operation `./`, as shown in the code at the start of this section.

Your Turn 2E

a. We often wish to evaluate the function $y = \mathrm{e}^{-(x^2)}$ over a range of x values. Figure out how to code this in vectorized notation.
b. We often wish to evaluate the function $\mathrm{e}^{-\mu}\mu^j/(j!)$ over the integer values $j = 0,\ 1, \ldots, N$. Here the exclamation point denotes the factorial function. Figure out how to code this in vectorized notation.

The item-by-item operations work equally well with two-dimensional arrays. In every case, however, the arrays to be combined must have exactly the same size: For example, the expressions `a + b` and `a.*b` generate error messages unless **`size`**`(a)` matches **`size`**`(b)`. (An exception has already been noted: If `a` is a single number, then operations like `a + b` are allowed regardless of the size of `b`. MATLAB also understands `a.^b` and `b.^a`, if `a` is a single number.)

Most often, you'll want item-by-item operations. Here is one useful exception: The "dot product" of two column vectors can be coded concisely as follows:

```
a = [1, 2, 3]; b = [1; 0.1; 0.01];
a*b
```

The number of columns in `a` equals the number of rows in `b`, so the matrix product is defined; in this case it is a 1×1 array (the single number $\boldsymbol{a} \cdot \boldsymbol{b} = a_1b_1 + a_2b_2 + a_3b_3$). More general matrix products are also handled by the `*` operator (not `.*`).

2.3.3 More vector operations

In addition to ordinary functions like **`sin`**, which act item-by-item when given an array, some functions are specifically about arrays. Try

```
a = [1:20; 101:120]
c = sum(a,1)
d = sum(a,2)
b = sum(a)
```

The function **sum**(a,b) takes an array a and an integer b and creates a new array. Each entry in the new array is the sum of the values in a with all allowed values of the bth index, holding fixed the other index (or indices). The notation **sum**(a) is a convenient notation for **sum**(a,1) : It sums the columns of a, creating a vector with a single row, that is, with an entry for each column of a.

You can explore the Help for the useful related functions **mean**, **std**, **min**, and **max**.

2.3.4 Higher dimensions

MATLAB can also handle arrays with more than two dimensions. Try

```
zeros(2,3,4)
```

and see what happens.

2.4 SCRIPTS

2.4.1 The Editor Window

MATLAB can do some useful things from the Command Window prompt, which is how we have been using it so far. But most tasks require code more complex than the examples above. Your code will go through many versions as you get it right, and it's tiresome to keep retyping things (making new errors each time). You'll want to work on it, take a break, then return to it, perhaps closing MATLAB or even moving to a different computer in the meantime. And you will wish to share code with other people in a "pure" form, free from all the typos, missteps, and output made along the way. For all these reasons, you will want to do most of your coding in the form of *scripts* (sometimes called "M-files").

A script is simply a chain of commands that you edit in a separate window, then execute with a single mouse click. To create a script, switch to MATLAB's "Editor Window" and create a new document from the New menu.[7] (If there's no such window open, type **edit** at the Command Window prompt.) Type in any of the codes discussed above. This time, when you hit <Return/Enter>, MATLAB does nothing. Instead, it lets you finish entering your code. When you click the (RUN) button, MATLAB executes all the code in the Editor Window, placing typed output in the Command Window. Before it does that, however, it always saves the code to a file, prompting you if needed for where to save it. It's customary to end MATLAB code file names with the extension .m.

> *It's good practice to start any script with* ***clear all****, so that you know exactly what state you're in when it begins.*

2.4.2 First steps to debugging

If you make an error in your code, in some cases MATLAB will catch it before you even click Run: A discreet little red bar appears in the far right of the Editor Window at the offending line. Try typing an expression with unbalanced parentheses to see this. Hover the mouse cursor over the red bar to get a tooltip with a brief

[7] If your heart is already pledged to another plain text editor, MATLAB understands. Just type a file name at the Command Window prompt and, if the current folder contains a plain text file with that name and extension .m, MATLAB will run it as a script. However, formatted text files, such as .rtf, .doc, or .docx files, won't work.

explanation. If you attempt to run the code without fixing the error, a more detailed message will appear in the Command Window, citing the line number where MATLAB first noticed something wrong. There may be other errors as well, but MATLAB quits scanning when it finds the first one. Fix it and try again.

If no such syntactic errors are found, there may nevertheless be other errors, such as attempting to evaluate an expression containing undefined variables.

> *The most common origin for undefined-variable errors is misspelling. Scanning the Workspace Window can help you find such errors. Variable names are case-sensitive, so inconsistent case is a form of spelling mistake.*

For example, if you define `myVelocity = 1` and later attempt to use `myvelocity`, MATLAB treats the second instance as a totally new, and undefined, variable.

Runtime errors also generate messages in the Command Window when you run the code. Most cause MATLAB to halt immediately. A few, however, are "nonfatal": For example, try[8]

```
myRange = -1:8; plot(myRange, log(myRange))
```

MATLAB finds nothing mathematically wrong here. But **`plot`** doesn't know how to handle the first two entries in the list of values you've given it, so it omits them and issues a warning message.

Still other situations generate no message at all—you just get puzzling results. For example, if you change the example just given to say `myRange = 0:8`, then there's no warning. The first point of your plot is just missing. It seems trivial, but you will encounter many situations of this sort that are too complicated to understand at once. Then you'll need to "debug" your code.

Whole books are written about the art of debugging. When a code generates a message that you don't understand, or no message but output that you don't understand, you need to look for clues. Here are some ideas:

- Read your code carefully. It's tedious; it requires intense concentration; but it's often the quickest route to insight.
- Build your code gradually. Most scripts are designed to execute a complicated task by breaking it into a series of simple tasks. Make sure each step does exactly what you expect.
- Start with an easier case. You're using a computer because you can't do the problem by hand, but maybe there's another case that you *can* do by hand. Adapt your code for that case (perhaps just a matter of changing the parameters) and compare to the output to the answer you know to be correct.
- Probe your variables. After a code finishes (or terminates with an error), all of its variables retain their most recent values; check them to see if anything seems amiss. In the example above, you can look in the Workspace Window and discover that `myRange` contains a negative value.
- Insert diagnostics. Somewhere prior to where you suspect there's an error, you can add a line or two that cause MATLAB to print out the value of some variables *at that moment* in the code's execution. Then rerun and check whether they're what you expect. Sometimes just removing a semicolon is enough.
- Be proactive as you write. If you think there's some possibility that `myRange` (defined somewhere else in your code) could be bad, you could insert a line that issues your own diagnostic:[9]

  ```
  if sum(myRange <= 0), disp('bad myRange!'); end
  ```

  Of course, you can't foresee every possible exceptional case. But you will develop a sense of what are the likely bugs in any situation.
- Learn about "breakpoints." This topic goes beyond the scope of this tutorial, and probably isn't helpful for most exercises in *Physical models,* but there may come a day when your code is complex enough to need it.

[8] The **`plot`** command will be discussed in Section 3.2.
[9] See Section 2.4.5 for the **`if`** command.

- Explain the code line-by-line *out loud* to another person or inanimate object. The later practice is often called "Rubber Duck Debugging." Professional developers commonly force themselves to explain malfunctioning code to a rubber duck or similar totem to avoid having to involve another developer. The act of describing what the code is *supposed* to do while examining what it actually does quickly reveals discrepancies.
- Ask a more experienced friend. This may be embarrassing, because almost all coding errors appear "stupid" once you've found them. But it's not as embarrassing as asking your instructor! And either one is better than endlessly banging your head on the temple wall. You need to become self-reliant, eventually, but it doesn't happen all at once.
- Ask an online forum. An amazing, unexpected development in human civilization was the spontaneous appearance of sites like `http://stackoverflow.com` (and others), where people pose queries at every level, and total strangers freely help them. The turnaround time can be rather slow; however, your question may already be asked, answered, archived, and available.

Maybe the single most important point to appreciate about debugging is that *it always takes longer than you expected.* The inevitable corollary is

Don't wait till the day before an assignment is due.

Some puzzles don't resolve until your subconscious has had time to unravel them. If you need help from a friend, lab partner, or instructor, that takes time too. Respect the subtlety of coding, and give yourself enough time.

2.4.3 Good practice: Commenting

Another key advantage of writing scripts is that you are free to include as many remarks to your reader as you like. Here is an example, starting with the code in Section 2.3.1:

```
% N Charles, 6/2014  quadsolutions.m  MATLAB R2014a: Tabulate some
% quadratic function solutions
clear all
%% parameters:
coef_b = 2; coef_c = 3;   % parameter values for b, c
step_a = 0.5;% stepping value for a
%% initialize
coef_a = -1;
%% find solutions
while (coef_b^2 - 4*coef_a*coef_c > 0),  % check whether solution will be real
    (-coef_b + sqrt(coef_b^2 - 4*coef_a*coef_c))/(2*coef_a),
    coef_a = coef_a + 0.5,
end;
disp('done!')
```

Compare this code to what was written in Section 2.3.1:

- The opening line now tells us who wrote the code, when, why, and in what version of the software.
- Variable names are more meaningful.
- Line 2 places MATLAB in a known initial state.
- Line 3 introduces a block of code, where parameter values are set up (with meaningful names). A "parameter" is simply a variable whose value is not going to change throughout the execution of the code.
- The first part of line 4 is code, but everything after the percent sign is commentary, here explaining the meaning of the first part to human readers.

- Lines 6 and 8 introduce other code blocks.

The script is certainly longer than the bare code introduced in Section 2.3.1, but which would your grader rather read? Which would *you* rather read, if you wanted to reuse this code after a month or two of not working on it?

> *Every coder eventually learns that documenting your steps saves time in the long run... as long as the documentation is accurately updated to reflect any changes made to the code.*

Notice that some lines begin with double percent signs. These mean nothing at all to MATLAB, but they help your reader by dividing the code into logical blocks. Besides looking good on the screen (try it), this structuring allows you to run individual blocks separately. To see how this works, click anywhere inside a block, then click `Run and advance` (also try the alternatives `Run section` and plain `Advance`).

Another key aspect of the code above is *indentation:* All of the steps to be repeated in the loop are indented relative to the main code. MATLAB doesn't require indentation, but it can be incredibly helpful in making your code more readable. You can select all your code and click `Indent>Smart` to see how MATLAB thinks its structure works. Often you can tell at a glance that an **`end`** is missing or misplaced, just by taking this step and scanning the resulting indentation.

2.4.4 Good practice: Use named parameters

Why did we clutter the code above with the variables named `coef_b, coef_c`, and `step_a`? We could have everywhere "hardcoded" them, replacing their cumbersome names by 2, 3, and 0.5, respectively. There are at least three reasons:

- We may later wish to use our code for some other purpose. That purpose may involve different values for fixed parameters. We may even wish to embed a snippet of our code into a *loop* that tries *many* values of a parameter. Keeping them symbolic helps.
- Using meaningful names for quantities improves your code's readability, even if those quantities stay fixed. What if there are two parameters, with different physical meanings, that just happen both to be equal to 5? If you give them different, meaningful names, then you won't get confused by this coincidence.
- Using named parameters to control the *size* of a calculation is extremely useful. You may intend to set up an array with, say, five million entries, and perform, say, a thousand calculations on each one. Such a code will take some time to execute. It would waste a lot of your time to wait for it every time you fix some small bug, add a feature, and so on. It's better to set up some parameters controlling how big the calculation is going to be at the top of the code, than to hardcode them. That way, you can choose small values for the development phase, and switch to the larger desired values only when you near the final version. Moreover, this way you can be certain that every array in the calculation will have the same number of elements.

These points are part of a bigger theme of coding practice:

> *Don't duplicate: define things once and then reuse the definitions.*

That is, if a parameter appears twice, invent a symbol for it and set its value *once.* If you don't implement this principle, then when you wish to change a value, inevitably you'll find and change all but one instance.[10] The one you missed will cause problems, and it will be hard to find.

Later we'll see how "don't duplicate" can be applied not just to parameters but also to code itself (Section 5.1).

[10] Or you may accidentally change something else!

2.4.5 Contingent behavior: Branching

The **while** loop above modified its behavior on the fly, depending on the result of an intermediate calculation. MATLAB has a much more general mechanism for specifying contingent behavior, called "branching." Try entering this code in the Editor Window and running it:

```
% S Spade, 6/2014:  branchExample.m  MATLAB R2014a: Illustrate branching
for trial = 1:5,
    userNumber = input('Pick a number: ');
    if userNumber < 0,
        disp('Square root is not real')
    else
        sqrt(userNumber)
    end
    userAgain = input('Another [y/n]?','s');
    if userAgain ~= 'y',
        break;
    end
end
if trial == 5,
    disp('Sorry, only 5 per customer')
elseif userAgain == 'n'
    disp('Bye!')
else
    disp('Sorry, I did not understand that.')
end
```

The code illustrates several new ideas:

- The **input** function causes MATLAB to type a prompt in the Command Window, then wait for the user to type something followed by <Return/Enter>.
- By default, **input** interprets the response as a number, and returns that value to the expression containing it.
- The keyword **if** is followed by a logical expression. If MATLAB evaluates the expression and finds it's TRUE, then it executes the next block of code, up to the next **end**, **else**, or **elseif** (in this case, line 5). If the expression is FALSE, MATLAB skips the block of code.
- In the example above, the first conditional block ends with an **else** in line 6. If the condition was TRUE, then MATLAB skips the following block (lines 7–8). If it was FALSE, then MATLAB executes the second block, up to the **end**.
- An alternative form of **input** appears in line 9: If an optional second argument is present, and it equals the string literal 's', then **input** treats user input as a string. The notation ~= in line 10 means "is not equal to."
- Sometimes a loop should terminate prematurely (before the condition in its **for** or **while** is satisfied). In line 11, the keyword **break** instructs MATLAB to go directly to the line following the **end** of the current loop (in this case line 14), and proceed. Line 14 then decides whether the loop terminated normally, and issues the appropriate message.
- Lines 14–20 set up a three-way branch: Only one of lines 15, 17, or 19 will be executed.

In short,

> *A branch construction starts with **if** and ends with a corresponding **end**. In between, there can be a single block of code, or multiple blocks separated by **else** or **elseif**.*

As with loops, indentation can be very useful to clarify the scope of a branching construction. You can see the conventional form by asking MATLAB to indent your code automatically (`Indent > Smart`).

The code listed above used the relation operators `<`, `==`, and `~=` to generate true/false values. Other such operators include `>`, `<=`, and `>=`. You can also generate more complex conditionals with the Boolean operators `&` (**and**) and `|` (**or**).

2.4.6 Nesting

When dealing with probability, we sometimes wish to create an array `a` whose `k,j` entry is a specified function of `k` and `j`. We can do this with a code like the following:

```
Nrows = 3; Ncols = 4;
a = zeros(Nrows, Ncols);
for theRow = 1:Nrows,
    for theCol = 1:Ncols,
        a(theRow, theCol) = theRow^2 + theCol^3;
    end % terminate for theCol
end     % terminate for theRow
```

Notice that

- Line 5 gets executed $3 \times 4 = 12$ times, because it sits inside two "nested loops."[11]
- Also for this reason, it is indented twice as much as the surrounding lines.
- It can be useful to add comments to each **end** indicating what it ends, even though indentation may make that unnecessary.
- Why did we start by calling **zeros**? Strictly speaking, it's not necessary; when you attempt to assign a value to a nonexistent element of an array, MATLAB quietly enlarges the array to accommodate your request. But if you know in advance how big an array will be, then it's good practice to declare its size in this way up front. Codes that perform this "preallocation" sometimes run much faster than those that don't.

[11] **if**/**end** pairs can also be nested.

CHAPTER 3

Data in, results out

Most data sets are too big to enter by hand, often because they were themselves generated by automated instruments. You need to know how to bring such a data set into your computing session ("import" it). You will also want to save your own work to files ("export" it) so you do not have to repeat complex calculations. MATLAB offers simple and efficient tools for reading and writing files.

Also, most results are too complex to grasp if presented as tables of numbers. You need to know how to present results in a graphical form that other humans (and you) can understand. MATLAB provides an excellent collection of resources for visualizing data sets.

3.1 IMPORTING AND SAVING

3.1.1 Importing data

3.1.1.1 Getting data

Much scientific work involves collections of experimental data, or "data sets." In order to crunch a data set, MATLAB must first import it. This is easiest if the data are in MATLAB's own proprietary format, generally indicated by a filename that ends with the extension `.mat`, but MATLAB can handle other formats as well. For example, "comma-separated value" (`.csv`) files are plain text; each line represents a row of an array, with entries in that row separated by commas. Also, the generic extensions `.txt` and `.dat` are often used on data files with comma or whitespace separation between items.

First you must obtain the data set of interest, and place it in a location where MATLAB can find it. Go to `http://www.macmillanhighered.com/physicalmodels1e` for a list of data sets.[1] Find the entry called `01HIVseries`, which contains a file called `HIVseries.mat`. Depending on your browser, you may need to do one of the following:

- Right-click the link[2] and choose `Save link as...` or `Download linked file as...`.
- Alternatively, you can enter the URL manually in your browser, or copy and paste it.
- If your browser downloads the file when you click on it, then you may need to figure out where your it actually placed the file; try looking in the browser's Downloads folder. Then move the data file to wherever you want it.
- If your browser displays the file contents as human-readable text when you click on it, you can copy and paste the text into a new file, or just use `File>Save` in the browser. Place it in your working directory as `HIVseries.csv`.

Where should you place your data file? Probably the most convenient place is the same folder that contains (or will contain) the script you'll write to analyze this particular data set.

You do not have to limit yourself to the data sets described here. Someday, you'll use data from some other source. You may have an instrument in the lab that creates it, or you may get it from a public repository or a coworker. Also, any scientific publication contains quantitative data, often in the form of graphs. A

[1] Resources mentioned here are also maintained at a mirror site: `http://www.physics.upenn.edu/biophys/PMLS/Student`.
[2] Right-click can also be accomplished on a Mac by holding the `<Ctrl>` key while clicking, or by using a trackpad gesture that you can customize in System Preferences.

graph is a representation of a set of numbers, and can be converted back to numbers by using a special purpose application such as Datathief.[3]

3.1.1.2 Bringing data into MATLAB

You are now ready to launch MATLAB and load the data set into it. To do this, do one of the following:

1. Double-click the file on your computer's file finder; or
2. Set MATLAB's default folder, and import from within MATLAB. To set the folder, look at the upper left of the Command Window, above the words `Current Folder`, click the open folder icon, and navigate to the folder you need.[4] Then either
   - Type **`load`** `HIVseries.mat` at the command prompt, or
   - Find the file in MATLAB's Current Folder Window and double-click it, or
   - Drag the file from your file finder and drop it onto the MATLAB window.

The data file will set up some variables with predetermined names. To find out what variables have been loaded, check MATLAB's Workspace Window. In this case, you'll discover a variable named `a` has appeared. Find out what kind of variable it is by using the command **`size`**; also find that same information in the Workspace Window.

To import a `.csv` file (perhaps exported from a spreadsheet application), navigate to its folder as in option **2** above. Find the file in MATLAB's Current Folder Window. Double-click it, or right-click and choose `Import Data`. A complicated Import Window opens, but just click the checkmark ("Import selection"). This time MATLAB doesn't know what variable names to give your data, so it makes up names (examine the Workspace Window to see them). Then close the Import Window. Try these steps with the file `HIVseries.csv`.

3.1.2 Saving

Saving your work is always important when working on a computer. It is even more important when you are working with the Command Window.

Data on the screen is fleeting; files on the hard drive are permanent.

If you have been working for hours and finally finish crunching numbers and making plots, great! But the moment you quit MATLAB, all that data is gone. Only data you have saved to files will outlast your session.

3.1.2.1 Code

Each time you run a script, MATLAB first saves it. The default choice of folder is probably not where you want it; change it as described in Section 3.1.1.2 above, or use `Save>Save As...` in the Editor Window.

You can also easily copy code and paste it into a word-processing application as part of an assignment (or part of your code diary), e-mail it, and so on, because code is plain text.

3.1.2.2 Data

In contrast to code, MATLAB does *not* automatically save its state; you must do this manually. It's an especially good idea if you're working on a big project, because then you can stop work and later pick up where you left off. (You can also return quickly to where you were previously, in case MATLAB crashes.)

To save MATLAB's entire current state, including all the variables that are currently defined, type

```
save('myData.mat')
```

[3] `http://www.softpedia.com/get/Science-CAD/DataThief.shtml`

[4] You can instead use **`cd`** at the Command Window prompt. Or you can automate this step by including the **`path`** command into your script (see its Help).

at the Command Window prompt. You can then take the resulting file `myData.mat` to another computer, double-click it in the file finder, and magically be back in the state you were in when you saved. Equivalently you can launch MATLAB, navigate to the appropriate folder, and type **`load`** `myData.mat`, or simply double-click the `.mat` file in the Current Folder Window.

You can also save data selectively, for example, like this:

```
save('myData.mat', 'x', 'y')
```

Here the name of the file is followed by names of the variables that you wish to save. Note that each must be enclosed in single right quotes. Dataset `.mat` files accompanying *Physical Models* were created in this way.

> MATLAB *doesn't allow filenames that contain blank spaces, so don't use them. If you wish to import a file with such a name, rename it first. Certain other punctuation symbols are also forbidden, even if your computer's operating system allows them.*

3.2 GRAPHS AND OTHER GRAPHICS

3.2.1 The `plot` command and its relatives

At last we are ready for some pictures. MATLAB will make an ordinary, 2D graph for you if you supply it a set of xy pairs. It's up to you to space those points appropriately. Try

```
num_points = 5;
x_list = linspace(0,4,num_points);
y_list = x_list.^2;
plot(x_list, y_list)
```

Notice that:

1. A new window has appeared. This is now the "current figure," and it's called "Figure 1" if you previously had no others.
2. Your graph is a bit lumpy, but you can improve that easily by specifying a larger value for the parameter `num_points`.
3. Inside the current figure, MATLAB has drawn and labeled some axes. These are now the "current axes."
4. MATLAB has automatically chosen to draw a region of the xy plane that contains all the points you supplied.
5. By default, the **`plot`** function takes the items in its first argument one by one, combines them with the corresponding items in the second argument to make xy pairs, and joins those points by a solid blue line.
6. Each figure window remains "live" until you close it. That means that you can modify its contents by issuing further commands (see below). However, changing the values in `y_list` will not automatically update your plot; you must issue another **`plot`** command to see such changes.
7. You can close a plot window manually in the usual way, or you can close them all with the command **`close all`**.

The two lists that you supply to **`plot`** must be of exactly the same length. That may sound obvious, but it's surprising in practice how often you'll get just one extraneous element in one of them, and MATLAB complains. If you're not sure exactly how many elements you'll get from `0:(.13):26`, just take a few seconds out from coding to type **`size`**`(0:(.13):26)` at the Command Window prompt and find out.

More generally, "off by one" errors dog programmers at every level:

> *Get in the habit of checking exact lengths of your lists.*

For the above example, you could do the following:

```
if size(x_list) == size(y_list)
    plot(x_list,y_list)
else
    disp('Error: x_list and y_list are not the same size')
end
```

Unless you only want to look at a rough plot of some data once and then discard the graph, you should write a script to generate your plot. Otherwise, you will find yourself retyping a lot of statements at the command prompt, just to change a label or the color of a single curve in the plot. Here are some of the more common adjustments you can make to a plot:

Every visual feature of a graph can be changed from its default, either at the time of creation or retroactively later. Unless you only want to look at a rough plot of some data once and then discard the graph, you should write a script to generate your plot. Otherwise, you will find yourself retyping a lot of statements at the command prompt, just to change a label or the color of a single curve in the plot. Here are some of the more common adjustments you can make to a plot:

- You can change your curve color at the time of plotting by adding another, optional argument: `plot(x_list, y_list, 'r')` gives a solid red line. Other line styles include the following:
  - `'b'` (blue), `'k'` (black), and other colors.
  - `':'` (dotted line), `'--'` (dashed line): When preparing an assignment, remember that red, blue, and other colors all look similar when printed on a grayscale printer. Other line style choices, such as dotted and dashed, are much easier to distinguish from a regular line.
  - `'.'` (don't join the points; instead use a small dot for each one), `'o'` (don't join the points; instead use a circle for each one), and other symbol choices.

  To some extent, options can be combined: `plot(x_list, y_list, 'or')` plots red circles, and so on.
- You can change MATLAB's choice of plot region by first creating the plot, then giving the command `xlim([1, 6])` to display the region $1 < x < 6$ (and stay with the default for y). The syntax is tricky: `xlim` expects a single argument, a list containing the lowest and highest values to display on the horizontal axis.[5] Similarly, `ylim` adjusts the vertical region.
- The command `axis('tight')` effectively issues `xlim` and `ylim` commands to make the axes just fit the range of the data, without the customary extra space around it.
- MATLAB chooses for you the height and width of the graph, then scales the x and y values by different amounts to make everything fit into a frame with standard height and width. If these variables are actually points in an xy plane, you may not like the resulting distortion of your figure. The command `axis('equal')` forces each axis to have the same scaling.

Some of the commands just mentioned point out a common theme in MATLAB: Attributes to be changed are often passed to a function as string literals, which must be typed exactly as shown. Some stand alone (like `'r'`). Others come in pairs, with a string literal keyword (like `'FontSize'`) followed by another argument that specifies the desired value (see Section 3.2.4 below).

If you don't want to join your plotted points, in addition to the options listed above there is a separate function called `scatter`. Using this function instead of `plot` gives you more control over the size, style, and color of every symbol.

Log axes

If you'd prefer a logarithmic vertical axis, use `semilogy` in place of `plot`. For a logarithmic horizontal axis, use `semilogx`; for a log-log plot, use `loglog`. In the two latter cases, you may want to evaluate your function

[5] These reasonable-sounding alternatives won't work: `xlim(1,6)` or `xlim(1:6)` .

at a set of values that appear uniformly spaced on a logarithmic scale; see the Help for **logspace** to set up such a list.

3.2.2 T2 Error bars

To make a graph with error bars, use

```
errorbar(x_list, y_list, e_list)
```

This function doesn't add error bars to an existing plot; rather, it creates the whole plot. It works like **plot** except with an additional list argument: Point `n` will be supplemented by an error bar that extends a coordinate distance `e_list(n)` above and below the point. (For asymmetric error bars and other options, see the Help.)

3.2.3 3D graphs

Sometimes a graph consists of points or a curve in a three-dimensional space. Try

```
t = 0:(pi/50):(10*pi);
plot3(sin(t),cos(t),t);
```

The function **plot3** accepts *three* arrays, whose corresponding entries are interpreted as the x, y, and z coordinates of the points to be drawn.

3.2.4 Manipulate and embellish

There is no limit to the time you can spend making your graphs pretty. Here are a few of the most useful tweaks. Many others can be quickly found via your favorite search engine (try searching the Web for `matlab graph triangle symbol`).

- You can change the font size for the numbers labeling the axes by first creating the plot, then giving the command **set**`(`**gca**`, 'FontSize', 14)`. In this command, **gca** ("get current axes") retrieves a number representing the plot you just made; then **set** modifies an attribute of those axes.
- You can add a title with **title**`('My first plot')`. If you don't like the default font size, instead use something like **title**`('My first plot', 'FontSize', 16)`.
- You can (and should) label the axes with **xlabel**`('speed')` and **ylabel**`('kinetic energy')`. No, wait! You can (and should) include units in label axes: **xlabel**`('speed [um/s]')`.[6] Even if you are reporting a relative quantity, like concentration relative to time zero, help your reader by stating explicitly `c/c(0) [unitless]` or `concentration [a.u.]`.
- Your screen is two-dimensional. If you ask for a "3D" plot, what you get must be a 2D projection, imagined as what a camera would see looking at your plot from some outside "viewpoint." Switching to a different viewpoint may make it easier to see what's going on. At the top of the plot window is a tiny icon called `Rotate 3D` (cube circled by an arrow). Click it. Now you can click and drag within your plot to change the viewpoint.

Your Turn 3A

Start with the code at the beginning of Section 3.2.1 and improve it. Make a smooth graph, with a thick red line, appropriate labels, and a title. Make the text big enough to read easily.

[6] Here are some other common formats: `speed, um/s`, or even `speed/(um/s)`. Use whichever you (or your instructor) prefer.

3.2.5 Multiple graphs on a single set of axes

You probably noticed in the previous examples that, if you already have a graph open, issuing a new graphing command wipes it out.

> *By default, a new graph replaces any previous graph in the currently active figure window.*

Sometimes this behavior is desirable, but not always. If you simply want two different plots open, issue the command **figure**(2), which opens a new figure window named Figure 2 (or switches to it if such a window already exists).[7] The next **plot** command will put its output there, leaving the other figure window alone.

Often, however, you'd like more than one graph superimposed on the same figure window. One way to accomplish this is to say **hold**('on') any time after creating a figure window; subsequent graphs created in that window pile on top of each other until you close the window or give the command **hold**('off'). MATLAB will ensure that all such graphs share the same numerical scale, and in fact the same axes.

The "hold" attribute is great for showing multiple graphs together, but it can get confusing if something is left over from some previous run of your script. It's good practice to start any graphing script with **close all**, so you know that all graphs that appear came from the current run. Don't attempt to superimpose log axes with regular axes by using **hold**('on'). The two styles are inconsistent; one of them will supersede the other.

Two other methods can be used to get multiple plots superimposed. Try

```
x = linspace(0, 1, 50);
y1 = exp(x); y2 = x.^2;
plot(x, y1, x, y2);
```

This example shows that you can give **plot** more than one set of xy list pairs. MATLAB will choose different colors for each curve, or you can specify them manually: **plot**(x, y1, 'r', x, y2, 'ko').

The third method is to give **plot** an ordinary vector of x values, but a 2D array of y values. Each row of y gives a separate curve. This method can be useful when you wish to look at a function for several parameter values.

Your Turn 3B

Try

```
num_curves = 3;
x = linspace(0, 1, 50);
y = zeros(num_curves, size(x,2));
for whichCurve = 1:num_curves,
    y(whichCurve,:) = sin(whichCurve*x*2*pi);
end
plot(x,y);
```

As soon as you start having multiple curves, you may need a "legend" explaining which is which. So look up the Help for **legend** and embellish the example just given.

3.2.5.1 T2 *Multiple axes in a figure window* You may wish to place more than one graph side by side in a single window for comparison. The function **subplot**(N, M, p) divides the current figure window into a grid of N rows and M columns, then makes cell number p the current one, where p is an integer between 1 and N*M. Any graphing command issued now will go into cell number p. Try this example:

```
close all; figure
```

[7] If you don't supply any argument, **figure** chooses a figure number that's not in use yet.

```
subplot(2, 2, 1); bar(rand(10, 2))
mylist1 = 0:.1:10*pi;
subplot(2, 2, 2); plot3(sin(mylist1), cos(mylist1), mylist1)
mylist2 = linspace(0, 3, 10)
subplot(2, 2, 4); plot(mylist2, mylist2.^2)
```

Make sure all your **subplot** commands for a particular figure window use the same values for `N` and `M`. If you want more freedom to place subplots, see the Help.

3.2.6 Saving figures

You can copy a graph from the menu in its plot window (`Edit>Copy figure`), then paste it into a word processor.

For higher-quality results, however, you must export the figure. One way is to use `File>Save As...` from its window menu, and choose the EPS file format. The `.eps` file generated in this way can be opened and modified in a "vector-graphics" application such as Inkscape or xfig (both freeware[8]), or a commercial alternative. Or you can save a graph in a "raster" (also called "bitmap") format such as `.jpg` or `.tif`, for inclusion in a presentation slide. Another way to export, which can be written into a script, is to use the **print** function (see its Help).

[8] `http://www.inkscape.org/en/`; `http://www.xfig.org`

CHAPTER 4

First Computer Lab

These exercises will use many ideas from the previous chapter. Our goals are to

- Develop basic graphing skills.
- Bring in a data set.
- Perform a simple fit of data to a model.

4.1 BASIC GRAPHING SKILLS: HIV EXAMPLE

4.1.1 Family of functions

Make a list by typing

```
time = 0:0.1:1
```

at the prompt and pressing <Return/Enter>. You should see a list of 11 numbers. Now assign a list of 101 numbers ranging from 0 to 10 to the variable `time` by typing `time =` followed by the appropriate expression.

Next, evaluate an expression involving your array. Use the solution to the differential equation solved in *Physical Models* (Nelson, 2015, Chapter 1) when modeling the concentration, V, of HIV in the blood at time t after the start of treatment:

$$V(t) = X\exp(-\alpha t) + Y\exp(-\beta t), \tag{4.1}$$

where the four parameters X, α, Y, and β are constants that control the behavior of the model; we don't know their values yet.

First, rename the constants as things you can type, for example, `aRate` and `bRate`. Give them some values. Also, it's wise to give longer, more descriptive names even to the ones you can type, for example, `time` for t. Next, try the code

```
viralLoad = X*exp(-aRate*time) + Y*exp(-bRate*time)
```

Let's first set $Y = 0$. Choose some interesting values for X and α, and evaluate $V(t)$ on some interesting range of values for t. You should now have two lists of numbers of the same length, called `time` and `viralLoad`, so try

```
plot(time, viralLoad)
```

4.1.2 Fit

Now let's have a look at some experimental data.

Go to the Data Sets section of `http://www.macmillanhighered.com/physicalmodels1e`.[1] Find the entry called `01HIVseries`, and get the file `HIVseries.mat`.[2] This file contains time series data about the

[1] Resources mentioned here are also maintained at a mirror site: `http://www.physics.upenn.edu/biophys/PMLS/Student`.
[2] See Section 3.1.1.

concentration of a virus in a patient's blood versus time after drug treatment. After you import it, you'll find a variable called `a` in the Workspace Window. It is an array (like a spreadsheet) with two columns of data. The first column is the time in days since administration of a treatment to an HIV-positive patient; the second contains the concentration of virus in that patient's blood in arbitrary units.

Before you can plot the viral load as a function of time, you need to do a simple manipulation to extract one vector from the first column of `a` and another vector from the second column. Do that and plot the data points now. Don't join the points by line segments; make each point a symbol, for example, a small circle or plus sign. Label the axes of your plot. Give it a title, too.

Assignment:

a. *Figure out how to superimpose the experimental data points on a continuous-curve plot of the function in Equation 4.1 above. Select reasonable values for the four parameters for the model and see what you get.*

The goal is now to tweak the four parameters of Equation 4.1 until the model agrees well with the data. But it's hard to find the right needle in a four-parameter haystack! We need a more systematic approach than just guessing. So think: How does our trial solution behave at long times? If the data also behave that way, can we use the long-time behavior to determine two of the four unknown constants, then hold them fixed while adjusting the other two?

b. *Even two constants is a lot to twiddle, so think some more: How does the initial value V_0 depend on the four constant parameters? Can you vary these constants in a way that always maintains fixed long-time behavior* and also *the initial value? That would leave only* one *remaining free parameter, which you could adjust fairly easily until you like what you see.*

c. *What values did you eventually choose for the parameters? Based on the development of the model in Physical Models, what can you deduce about HIV from your results?*

[*Remark:* You probably know that there are black-box software packages that do such "curve fitting" automatically. In this lab, you are to do it manually, just to see how the curves respond to changes in the parameters.]

4.2 BACTERIAL EXAMPLE

4.2.1 Two families of functions

Here are two more families of functions:

$$V(t) = 1 - \mathrm{e}^{-t/\tau}; \qquad W(t) = A\left(\mathrm{e}^{-t/\tau} - 1 + \frac{t}{\tau}\right). \tag{4.2}$$

The parameters τ and A are constants. Functions of this sort come up in *Physical Models* Chapter 10 in the context of the Novick-Weiner experiment. Proceed as in the HIV example above, but with these functions and some different data sets.

Assignment:

a. *Choose $A = 1$, $\tau = 1$, and plot $W(t)$ for $0 < t < 2$.*

b. *Make several lists `W1`, `W2`, `W3`, and so on, using different values of τ and A, and plot them simultaneously on the same graph.*

c. *Make the different lines different colors.*

d. *Explore some of the other graph options that are available; for example, wouldn't it look good to add a legend, to help the reader sort out all the curves?*

4.2.2 Fit

Go to the Data Sets section of `http://www.macmillanhighered.com/physicalmodels1e`.[3] Find the entry called `15novick`, and get the file `g149novickA.mat`, which sets up an array named `data` containing data about bacterial population in a culture versus time. Extract one vector from the first column of `data` and another vector from the second column. Now plot the data points. Don't join the points by line segments; make each point a symbol, for example, a small circle or plus sign. Label the axes of your plot.

Assignment:

a. *Superimpose the experimental data points on a multiple-curve plot of $V(t)$ (see Equation 4.2), similar to the one you made for $W(t)$ before. Select some reasonable values for the parameter τ in the model, and see if you can get a curve that fits the data well.*

b. *Now try the same thing using the data in `g149novickB.mat`. This time throw away all the data with time value greater than 10 hours, and attempt to fit the remaining data to the family of functions $W(t)$ in Equation 4.2.*

[*Hint:* At large values of t, both the data and the function $W(t)$ become straight lines. Find the slope and the y intercept of the straight line determined by Equation 4.2 in terms of the two unknown quantities A and τ. Next estimate the slope and y intercept of the straight line determined by the data. From this, figure out some pretty good guesses for the values of A and τ. Then tweak the values to get a nicer looking fit.]
[*Remark:* Again you are asked not to use an automatic curve fitting system. Using methods like the one suggested in the Hint can give you a far better feeling for the meaning of the math than just accepting whatever a black box spits out.]

[3] Resources mentioned here are also maintained at a mirror site: `http://www.physics.upenn.edu/biophys/PMLS/Student`.

CHAPTER 5

More MATLAB Constructions

At its highest level, numerical analysis is a mixture of science, art, and bar-room brawl.
—T. W. Koerner, *The Pleasures of Counting*

The preceding chapters have developed a basic set of techniques for importing, creating, and modeling data sets and visualizing the results. This chapter introduces additional techniques for exploring mathematical models and their results:

- Random numbers and probabilistic simulations
- Solutions of nonlinear equations of a single variable
- Solutions of linear systems of equations
- Numerical integration of functions
- Numerical integration of ordinary differential equations

In addition, this chapter introduces several new methods for visualizing data, including histograms, surface plots, contour plots, vector field plots, and streamlines.

We start with a discussion of writing your own functions, an invaluable tool in exploring models of physical and biological systems.

5.1 WRITING YOUR OWN FUNCTIONS

5.1.1 Don't duplicate

Section 2.4.4 introduced a principle:

Don't duplicate: define things once and then reuse the definitions.

In the context of parameters, this principle told us to enter a parameter's value just once at the start of our code. But code itself can contain duplications, if we need to do the same (or nearly the same) task many times. Just as with parameter values, you may later realize that something needs to be changed in your code. It would be tedious and error prone to change every instance. It's better to define a *function* once, then invoke it whenever needed. You may even need a snippet of code (perhaps a simulation module) in more than one of your scripts. If each script invokes the same externally defined function, then fixing that file once will fix it for all of your scripts.

Functions in MATLAB can carry out mathematical operations, like `cos(x)`; they can make plots; they can read and write files; and much more. Your own functions can do all of these things as well. Functions are ideal for writing code to accomplish a task once and reusing it often.

5.1.2 Defining functions

One way to specify a function is to place it in a separate file, called, for example, `myFunction.m`. That file should be located in the same folder as your main script.[1]

A function is a collection of MATLAB code, like a script. Unlike a script, however, a function file starts with a line like this:

```
function myval = myFunction(myInput, myOther, ...)
```

This line specifies the *name of the function* (in this case `myFunction`), some names for the *input arguments* (`myInput`, ...), and a variable name for the *value to return* (`myval`). Each of these can have any name you like, but the name of the function must match the filename; thus, in this case the code must be in a file named `myFunction.m`. Once you have finished setting up your function file (and saved it!), then any other MATLAB code in the same folder can include lines like

```
y = myFunction(u, w^3) + 3;
```

When this line is executed, `myFunction` acts just like **`sqrt`** or some other predefined function:

- MATLAB transfers control to `myFunction.m`, after first assigning the current value of the variable `u` in the main code to a variable called `myInput` in `myFunction`. Similarly, it evaluates `w^3` in the main code and assigns that value to `myOther` in `myFunction`.
- When `myFunction.m` is finished, MATLAB takes the current value of the output variable specified in its opening line (here `myval`) and substitutes it in place of `myFunction(u, w^3)` in the main script. It then finishes evaluating the expression containing that code and, in this case, assigns the answer to `y`.

A function can return anything to the code that invokes it:[2] a number, an array of numbers, a string, even nothing at all. The arguments can also be of any type (or there may not be any at all). For instance, you might like to perform a rotation. The first argument could be a column vector to be rotated; the second could be the angle of rotation. The output would then be the rotated vector. Here is a complete working example of a user-defined function that implements these steps:

```
function newvec = rotavec(oldvec, theta)
% MH Flambeau 8/2014 file  rotavec.m   MATLAB R2014a:  rotate a vector
% oldvec = 2D column vector
% theta = angle to rotate, in radians
% returns a 2D column vector
newvec = [cos(theta), -sin(theta); sin(theta), cos(theta)]*oldvec;
```

Note that the `*` in line 6 indicates matrix multiplication.

After creating the file `rotavec.m` with the above contents, try typing **`help`** `rotavec` at the Command Window prompt. MATLAB types back the first few lines of comments it finds in the file. Then try the command `rotavec([1;0], pi/2)` and so on.

> *After you edit a function in the Editor Window, you must manually save it before using it.* MATLAB *only automatically saves a top-level script when you click the (*RUN*) button.*

Section 5.5.1 gives a more concise way to set up short, user-defined functions.

---

[1] When you launch MATLAB, its default folder will be something like `<your username>/Documents/MATLAB/`. Files in this folder are always available to MATLAB, even if you change the default, so this is another place where you may wish to place useful functions. If you know about "paths," then you can figure out how to create other folders that are visible to MATLAB regardless of the current folder (**`doc path`**).

[2] A user-defined function can return two or more results via the syntax **`function`** `[val1, val2] = myFunction(...)`. The calling script receives these values via the syntax `[x, y] = myFunction(...)`.

5.1.3 Scope

Another feature of function files is subtle but important. The **function** command also tells MATLAB to create a "sandbox," in which *all variables in the main code's workspace are hidden* from the rest of your code. While the function is running, it knows nothing about the surrounding world that called it *except* the values of arguments that were explicitly passed to it. And those arguments are passed only as values; while the function is running, it does not even know, or need to know, what their names were in the calling code. Thus, in the example above, the code `myFunction.m` does not know that the code that called it has a variable named `u`. And if `myFunction.m` changes the value of its variable `myInput`, that change does not affect the value of `u` when control returns to the calling script.

Similarly, after the function finishes, the calling script doesn't need to know the name of the output variable in `myFunction.m`. All that it learns about the function is the *value* of its result. The two codes are therefore "insulated" from each other, communicating only through the arguments and outputs.[3]

Thus, a function interacts with other code only via a very narrow pipeline, the values of its input argument(s), and the returned value(s). It is easy to forget this feature, tell `myFunction` to use a variable from the surrounding code, and receive a perplexing error message that the variable is not defined. But this feature saves much more trouble than it causes. By limiting the scope of variables in this way, MATLAB saves you from the much more perplexing situation where you change a variable by inadvertently creating an unrelated variable with the same name inside a function! The *modularity* imposed by a functional structure is another key feature of good coding practice.

5.2 RANDOM NUMBERS AND SIMULATION

There are many interesting problems in which we do not have complete knowledge of a system, but we do have knowledge of the probabilities of the outcomes of simple events. For example, you know the probability of any given outcome in the roll of a single die is 1/6, but do you know how likely it is that the sum of a roll of 5 dice is less than 10? Rather than work out the combinatorics, you could instead roll dice many times and determine the probability empirically.

A random number generator makes it possible to "roll dice" millions of times per second, by using a computer. You can use a random number generator to determine the likely behavior of a system described by a stochastic model in which the probability distributions of the parameters are known, but it is impossible to determine the exact behavior. Such calculations are called Monte Carlo simulations.

5.2.1 Simulating coin flips

Suppose that you want to simulate flipping a coin one hundred times, record the number of heads, and then repeat that N times, arriving at N numbers, each lying in the range between 0 and 100, inclusive. First try typing

```
1 > 2
```

at the Command Window prompt, then repeat with `2 > 1`. You'll see that MATLAB assigns the values 1 or 0 to indicate the truth of a proposition. So you can simulate a coin flip by using the function **rand** to generate a Uniformly distributed variable on the range from 0 to 1, then convert it to a binary digit by comparing it to 1/2.

To make a long list of independent flips, first make a list called `rlist` with 100 Uniform random numbers. (Use **help rand** for a clue about how to do this easily.) Then you can say `rlist = (rlist > 0.5)`. MATLAB applies the condition to the list item by item, then substitutes the new values for the old ones. You can then count heads by using the **sum** command.

[3] T_2 This tutorial will not discuss the different rules governing "subfunctions."

5.2.2 Generating trajectories

Let's create a random walk of 500 steps. Thus, our trajectory will be a list of 500 x values and 500 y values. Following good practices (Sections 2.4.4 and 2.4.6), begin with

```
walklength = 500;
x = zeros(1,walklength); y = zeros(1,walklength);
```

Next, notice that the x and y coordinates of our random walker make independent steps. You know from the preceding section how to get a list of 500 random binary digits. Make two such lists called `stepx` and `stepy`. But we need lists consisting of random ± 1 entries, not ones and zeros.

Your Turn 5A

a. Do a simple operation on `stepx` to convert it to that form, and similarly for `stepy`.
b. Now you need to convert your lists of steps into actual successive positions of the random walker. Consult MATLAB's help for the command **`cumsum`** to see why it's what you need.
c. Complete your code by making a picture representing your random walk. Run it several times.

5.3 THE HIST COMMAND AND ITS RELATIVES

A histogram is a bar plot used to display a discrete, empirical probability distribution. To make a simple histogram, try[4]

```
hist([1, 2, 2, 2, 3, 10, 10])
```

Indeed, a figure appears, but a number of things have happened without your supervision. MATLAB has inspected the list of data and found its range, decided how many equally spaced bins to divide that range, counted how many entries in the data fall into each bin, and graphed the result as a set of bars. But you may want to set those divisions yourself, or at least know what MATLAB has chosen. If you only want a little control, you could add an optional argument telling MATLAB how many bins you want in that range. Compare the output of **`hist`**`(`**`rand`**`(1, 1000))` to **`hist`**`(`**`rand`**`(1, 1000), 50)`.

For even more control, you can tell MATLAB exactly where you want each bin to be centered; this is a bit tricky, so look carefully at **`help hist`**.

For control over presentation, you can instead write

```
[mycounts, mycenters] = hist([1, 2, 2, 2, 3, 10, 10]);
```

On the left of the equals sign, we are specifying that we wish for **`hist`** to return *two* results to us, and that they are to be placed in variables called `mycounts` and `mycenters`, respectively. MATLAB tells the **`hist`** function how many results were requested. If *no* returned values were requested, as at the start of this section, then **`hist`** draws a bar graph. If two results are requested, as above, then no graph is drawn; instead, **`hist`** returns an array with the numbers of counts in each bin, and a second array with the centers of each bin.

Your Turn 5B

Try it, and inspect the returned variable values to make sure you understand.

[4] **`hist`** is deprecated starting with MATLAB version R2014b; `histogram` is the recommended replacement.

We are now free to make the bar graph ourselves:

```
bar(mycenters, mycounts)
```

or to plot in some other style, or even to perform some other transformation prior to plotting. The **hist** function has simply done the binning (classifying and counting) for us.

Often it is more convenient to specify not the centers of the bins, but instead the range of values that each bin collects. The **histc** function accomplishes this; see its Help. Note that **histc** never draws a graph; you must do that yourself as described above.

5.4 SURFACE AND CONTOUR PLOTS

In earlier chapters, we have seen several methods for plotting data sets in which only a single parameter is varied. Such data sets lend themselves to two-dimensional plots. However, models with two or more independent parameters require higher dimensional plots.

5.4.1 Surfaces

A function of two variables, $h(x, y)$, can be visualized as a surface whose height over each point (x, y) has been specified, just as Earth's topography is specified by altitude as a function of latitude and longitude. To draw such a representation, we set up a grid of xy pairs, evaluate our function at each grid point, and invoke MATLAB's function **surf**.

Typically our grid consists of points regularly spaced in x and in y. We can specify it by making a list of desired x and another list of y, and then invoking **meshgrid**. Try

```
xrange = -1:1; yrange = 0:100:100;
[X, Y] = meshgrid(xrange, yrange)
```

Make sure you understand the result: **meshgrid** returned two outputs, `X` and `Y`, each an array with $3 \times 2 = 6$ entries giving the x and y coordinates of six grid points. You can now evaluate a function on those values, perhaps using vectorized arithmetic, to produce a third array called `Z`, and then invoke **surf**`(X, Y, Z)` to show the result.

Your Turn 5C

Show the function $z = x^2 + y^2$ over a suitable grid of values, where x and y range from -1 to 1.

The function **surf** uses color as a secondary indicator of the value of the function. If you'll be printing an assignment in grayscale, you may get better results if you tell MATLAB not to use color: After creating the graph, use the command **colormap**`('gray')` or **colormap**`('bone')`.

5.4.2 Contour plots

Generating a contour plot is similar to creating a surface: Set up a grid and evaluate your function as before, then use

```
contour(X, Y, Z);
```

The number of contour lines is 10 by default. To change this, add a fourth argument to the function call: **contour**`(X, Y, Z, 20)`. You can also label the contour lines by using the following commands:

```
[C, h] = contour(Z);
clabel(C, h);
```

5.5 NUMERICAL SOLUTION OF NONLINEAR EQUATIONS

It is often necessary to solve nonlinear equations when studying physical and biological system. For example, determining the fixed points in a single-gene toggle system requires finding the roots of a sixth-order polynomial, but there are no general analytic solutions for polynomials higher than the fourth degree. Biological oscillators involve even more complicated functions that are not even polynomials. Numerical methods for finding the roots of such expressions are therefore essential.

5.5.1 Function handles and anonymous functions

Numerical solvers, and other functions to be introduced later, involve a syntax that we haven't met yet. In order to tell the solver what to solve, we need to pass it an argument that is *itself* a function (the one whose roots we are seeking). We do this by creating a function **handle**, that is, a "pointer" to a function; the syntax is to preface the function's name by the `@` symbol. Thus `@sin` is a handle to the function **`sin`**, and so on.

The `@` symbol can also be used to *create* a function, if its definition is very simple. Try entering `myFn = @(x) x^2-1` at the command prompt. In the Workspace Window, you will now see `myFn` as a new symbol; typing `myFn(3)` confirms that it is a function.

More generally, when the `@` sign is followed by an opening parenthesis, not the name of a function, then it first creates a new function, and then returns a handle to it. The symbol(s) inside the first set of parentheses are variable(s) that will be assigned from the arguments supplied when function is used; the following expression involving those argument(s) defines what will be computed and returned. Thus, `@(x) x^2-1` creates a function ("square the argument and subtract 1") and returns a handle pointing to that function. If you wish to use that function directly, you can name it, as in the example above. If you only wish to pass it along to another function, however, then you needn't give it any name; just substitute the "anonymous function" handle `@(x) x^2-1` anywhere a function handle is required.

5.5.1.1 T2 Scope and anonymous functions

Unlike functions defined via **`function`**, anonymous functions defined by `@` can also "see" the values of variables in the surrounding MATLAB code. But those values are frozen at the time the `@` is executed; they are not the current values when the anonymous function itself is called.

5.5.2 Roots of general real equations

MATLAB can find the roots (zeros) of a nonlinear function using **`fzero`**. This function takes two arguments, a function handle and a point in the domain of the function at which **`fzero`** begins searching for roots. The function can be defined in any of the ways discussed in Sections 5.1 and 5.5.1.

Example: Try the following and explain the results:

```
fzero(@(x) x^2-1, 0.5)
fzero(@(x) x^2-1, -0.5)
```

Solution: The equation $x^2 - 1 = 0$ has two roots, but which root **`fzero`** returns to us depends on where we tell the function to begin searching.

Example: Now try the following, and explain what you see:

```
fzero(@sin, 2)
fzero(@sin, 1)
```

Solution: The second expression does not return exactly 0. What you are seeing is an example of *numerical error*, caused by the finite number of binary digits that MATLAB uses to represent a number. The result is a number that is extremely close the correct solution, but not exact.

The phrase "numerical error" is standard, but unfortunate. *You* didn't make any error; neither did MATLAB. Your computer's hardware simply has limited precision.

Understanding how **fzero** operates also helps us understand the return value of **fzero**`(@(x) 1/(x-1), 2)`. Note the discontinuity in the function $1/(x-1)$ at the value $x = 1$. The function is positive when approaching the discontinuity from the right and negative when approaching it from the left. The function **fzero** iterates through the domain of the function looking for points where the value crosses zero, so in this situation it incorrectly finds $x = 1$ to be a root. By using the syntax

```
[root, fval, statusflag] = fzero(@(x) 1/(x-1), 2)
```

and checking the return value of `statusflag`, singular points (and a variety of other error conditions) can be detected. If your numerical result doesn't make sense, consult **help fzero**. It's also often helpful to make a graph of your function and *look* at it to understand what's going on.

You may be able to give **fzero** better instruction if you know that the root of interest lies between two values. Replace the starting point by a two-element list to supply the lower and upper bounds of the region you'd like **fzero** to search. (If there's no zero in that range, you'll get an error message.)

5.5.3 Complex roots of polynomials

Unfortunately, **fzero** is restricted to real numbers only. Consider the function $1/x = 1 + x^3$. In order to use **fzero**, we can manipulate this into $0 = x(1 + x^3) - 1$ and solve for the roots (assume that we have some good reason to guess that the real roots are near 1 and -1):

```
fzero(@(x) x*(1+x^3)-1, 1)
fzero(@(x) x*(1+x^3)-1, -1)
```

But a quartic equation has four roots; in the present case, **fzero** missed two roots because they are complex.[5]

MATLAB can find the roots of any polynomial by using the function **roots**, which takes a vector of coefficients (see **help roots** for details). For arbitrary nonlinear equations, however, Wolfram Alpha (or *Mathematica*) is probably the most effective tool. For example, the equation $1/x = 1 + x^{2.4}$ cannot be solved by using **roots**.[6] In fact, it has three solutions, but only one is real.

5.6 SOLVING SYSTEMS OF LINEAR EQUATIONS

Often we need to solve a set of simultaneous linear equations, with the general form

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

That is, we are given the a's and c's, and wish to know the x's. MATLAB has an *extremely* concise notation for the solution to this problem, represented by the backslash operator (see **help mldivide**). The code

```
A = [-1; 5];
C = [1, 3; 3, 4];
X = C\A
```

puts the solution to this problem into the column vector `X`, as you can confirm by computing `C*X`. There is a mnemonic to help you remember this notation: You can think of the solution of $A = \mathbf{C}X$ as "dividing both sides from the left by **C**." The backslash operator does this.

---

[5] One way to see this is to use a symbolic math package like Wolfram Alpha: `solve(x*(1+x^3) - 1,x)`. See Section 7.3.

[6] Actually, you *can* solve this particular equation by using **roots**. Rewrite it as $x^{3.4} = 1 - x$, raise both sides to the fifth power to generate a 17th-order polynomial, get all 17 roots from **roots**, then use a **for** loop to substitute them into the original expression and eliminate the extraneous solutions.

5.7 NUMERICAL INTEGRATION

There are several integration routines available, each with particular strengths. The all-purpose workhorse is called **integral**.

5.7.1 Integrating a predefined function

To set up a numerical integration, define a function that is to be your integrand, then give **integral** a handle for that function (Section 5.5.1). To illustrate this, let's evaluate $\int_0^{x_{\max}} \mathrm{d}x\ \cos(x)$ for various values of $x_{\max}$. Try this code:

```
ind = 0;
for xmax = 0:(3*pi/15):3*pi, % try various upper limits
    ind = ind + 1;
    result(ind) = integral(@cos, 0, xmax);
    end
plot(0:(3*pi/15):3*pi, result)
```

Remarks:

- The first line sets up an integer index `ind`, which keeps track of where each intermediate result is to be kept. Note that in MATLAB, the first element of an array has index 1, but we initialized `ind` to 0, so `ind` must be incremented before it is used.
- The next line sets up a list of values for `xmax`; we'd like to evaluate the definite integral with each of these. The actual integrations are done in the third line.
- The syntax `@cos` creates a handle for an existing function, in this case **cos**. The function must be able to accept a vector of input values and return a vector containing its values when evaluated item by item on that list. The built-in function **cos** can do this.
- Passing `@cos` as an argument to **integral** tells it what function you'd like it to integrate.

The **integral** function evaluates **cos** at a vector of input values and uses the output to compute the integral over the range given by its second and third arguments.

Your Turn 5D

Do the integral analytically, and check whether MATLAB got it right.

5.7.2 Integrating your own function

Next, suppose that you wish to integrate a function that is *not* predefined. If the function is long and complex, you'll want to define it in a separate file `myFunction.m`, as described in Section 5.1, then substitute the handle `@myFunction` for **@cos** in the above code. But if your function can be expressed in a single line, the following abbreviated syntax is more concise. Try entering this code in the Editor Window, then running it:

```
ind = 0;
for xmax = 0:(3*pi/15):3*pi; ind = ind + 1; % try various upper limits
    otherresult(ind) = integral(@(x) (1/10)*x.^2, 0, xmax); end
plot(0:(3*pi/15):(3*pi), otherresult)
```

Your Turn 5E

a. Why did we need to use the syntax `x.^2` in the above code?
b. Again check that MATLAB got the right answer.
c. Try a function whose integral you *don't* know: Evaluate

$$\int_0^{x_{\max}} \mathrm{d}x\, \mathrm{e}^{-x^2/2}$$

for values of $x_{\max}$ from 0 to 5, and plot the answer.

If a function can be integrated all the way to infinity, MATLAB may be able to give you that answer: Try replacing the upper limit by **`Inf`** (or the lower limit by **`-Inf`**) in the above question.

5.7.2.1 T_2 *Oscillatory integrands*

Sometimes we wish to evaluate an integral whose integrand oscillates rapidly. A variant of **`integral`**, named **`quadgk`**, is optimized for this purpose (see its documentation):

```
clear all; close all
i = 0;
for thetamax = 0:(3*pi/15):3*pi; i = i + 1; % try various upper limits
    result(i) = quadgk(@(theta) cos(theta), 0, thetamax, 'MaxIntervalCount', 5000);
end
plot(0:(3*pi/15):(3*pi), result)
```

Actually, for the calculation just given **`integral`** would have worked fine. But the code snippet above does illustrate an important option that you can give to **`quadgk`**: Setting `MaxIntervalCount` equal to 5000 tells MATLAB how hard to work on this calculation. Bigger values give you better results, but take longer to run.

5.8 NUMERICAL SOLUTION OF DIFFERENTIAL EQUATIONS

In both physical and life science, it is often possible to write down a set of differential equations that govern a system (or describe the behavior of a model), but impossible to solve this system in terms of known functions. A classical example is the three-body problem in classical mechanics: $\boldsymbol{F} = m\boldsymbol{a}$ and Newton's law of gravitation are all that is required to write down the differential equations, but no one in the last four centuries has been able to solve them!

Numerical stepping is a powerful tool in studying such systems. Starting from some initial configuration of a system, you can calculate the next configuration by using the differential equation(s). From *that* configuration, you can calculate the *next* configuration, again by using the differential equation(s), and so on. Computers are ideal for executing this repetitive operation, and many efficient libraries have been developed for the task.

5.8.1 Reformulating the problem

An ordinary differential equation system (ODE) is one that only involves functions of a single parameter, often considered as "time" and denoted t. A textbook example is the driven harmonic oscillator:

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = -y + g(t),$$

where $g(t)$ is some given "driving" function. MATLAB's function **ode45** can "only" solve ODEs of the form

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \boldsymbol{F}(t, \boldsymbol{y}). \tag{5.1}$$

Fortunately, *any* ODE can be put into this form. For the harmonic oscillator example, first define two variables that will be the components of a vector $\boldsymbol{y}$:

$$y_1 = y \qquad\qquad y_2 = \frac{\mathrm{d}y_1}{\mathrm{d}t}.$$

Next, write the derivatives of y_1 and y_2 in terms of y_1, y_2, and t:

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = \frac{\mathrm{d}y}{\mathrm{d}t} = y_2 \qquad\qquad \frac{\mathrm{d}y_2}{\mathrm{d}t} = \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -y + g = -y_1 + g.$$

This allows us to cast our problem, a second-order differential equation, in the form required by **ode45** (Equation 5.1). $\boldsymbol{y}$ is a vector with two entries, and $\boldsymbol{F}(t, \boldsymbol{y})$ returns a vector with two entries:

$$\boldsymbol{F}(t, \boldsymbol{y}) = \begin{bmatrix} y_2 \\ -y_1 + g(t) \end{bmatrix} \tag{5.2}$$

Any ordinary differential equation, or coupled set of such equations, can be recast in a similar manner.

5.8.2 Explicit solutions

The function $\boldsymbol{F}$ in Equation 5.1 specifies what ODE we would like MATLAB to solve. The function $\boldsymbol{F}$ must specify a vector field on state space, possibly with explicit dependence on time (Equation 5.2). That is, $\boldsymbol{F}$ should accept a time value and a vector $\boldsymbol{y}$, and return a vector. Use **help ode45** to see how we specify that information. Our goal is to create a function that evaluates $\boldsymbol{F}(t, \boldsymbol{y})$.

Consider a simpler example, exponential growth, for which $\mathrm{d}y/\mathrm{d}t = 0.05y$. In this case, we create a function, called `expGrowth`, in a file called `expGrowth.m`. Note that, although in this simple case $F(y)$ has no dependence on t, we must still include the variable `t` as its first argument, because **ode45** requires it:

```
function out = expGrowth( t, y )
% file expGrowth.m:  MATLAB R2014a: evaluate right side of differential equation
k = 0.05;
out = k*y;
```

Now we can solve the equation by using **ode45**:

```
% solveODE.m: MATLAB R2014a:  illustrate solution of an ODE
clear all; close all
y0 = [1, 2]; % solve equation for two different initial values
t0 = 0;      % starting time
tfinal = 100;% ending time
kvalue = 0.05;
[t, y] = ode45(@expGrowth, [t0, tfinal], y0);
plot(t, y(:, 1), 'og'); hold('on');
plot(t, y(:, 2), 'or');
plot(t, y0(1)*exp(kvalue*t), 'g');     %check against explicit solution for y0 = 1
plot(t, y0(2)*exp(kvalue*t), 'r');     %check against explicit solution for y0 = 2
hold('off');
```

Comments:

- As with numerical integration, the solver expects a function handle, coded with the `@` construction (Section 5.5.1).
- The second argument is a vector with two entries, specifying the start and end times for the desired solution.
- The solver returns a list of times in the range that we specified and another list containing the solution evaluated at those times.
- The code above gave **ode45** multiple initial values (`y0` is a list) in a single function call. The solver in turn produced multiple solutions (so actually `y` is an array), which we plotted individually.

For a simple function like exponential growth, we could have used an anonymous function, instead of writing an external `.m` file. In this case, **ode45**`(@(t, ytmp)(kvalue*ytmp), ...` works equally well, avoids the secondary `.m` file, and moreover implements the "Don't duplicate" principle, because now the value `0.05` is only entered once. Finally, using an anonymous function allows us to put the whole thing in a loop over several values of `kvalue`, generating solutions to a family of similar equations.

5.8.2.1 T2 Advanced syntax

For more complex functions, we can use a method called "parameter binding" to accomplish the same thing. For this, we redefine our function (here `expGrowth`) to include dependence on the parameter `k`:

```
function out = expGrowth2(t, y, k)
% file expGrowth2.m  MATLAB R2014a:
out = k*y;
```

We cannot give this function directly to the solver, because it doesn't have the expected two arguments. Instead we use an indirect method:

```
k_range = [0.03,  0.04,  0.05];
colors = ['r',  'g',  'm'];
y0 = 1;
t0 = 0;
tfinal = 100;
%% expGr_h is a handle to an anonymous function of k, which itself returns a handle:
expGr_h = @(k)(@(t,y) expGrowth2(t,y,k));
for ndx = 1:size(k_range,2)
    kvalue = k_range(ndx);
    color = colors(ndx);
% since expGr_h is already a handle, we do not use @
    [t,y] = ode45(expGr_h(kvalue), [t0 tfinal], y0);
    plot(t, y, [color, 'o']); hold('on');
% check against explicit solution for y0 = 1:
    plot(t, y0*exp(kvalue*t), color);
end
hold('off');
```

5.9 VECTOR FIELDS AND STREAMLINES

5.9.1 Vector fields

A vector field is a function whose value at any point in space is a vector. Common examples from physics include the electric field, the magnetic field, and the velocity field of a fluid. MATLAB provides two useful functions for visualizing vector fields and their streamlines.

You can plot a 2D vector field by using the **quiver** function. It requires four arguments, all of which are arrays of the same size. The first two arguments define a grid of xy values, as in Section 5.4.1. But instead of specifying an ordinary function at these points, we specify a 2D *vector* $\boldsymbol{V}(x, y)$ at each one. And instead of constructing a smooth surface interpolating between the points, MATLAB will draw an arrow at each one, with direction and length given by the corresponding 2D vector.

Your Turn 5F

For example, try the script

```
close all;
[X, Y] = meshgrid(-1:.2:1, -1:.2:1);
Vx = Y; Vy = -X;
quiver(X, Y, Vx, Vy)
```

and explain the "vortex" pattern you get.

5.9.2 Streamlines

Systems of first order, ordinary differential equations can be visualized in terms of a vector field. Another way to find solutions is to follow the arrows, generating "streamlines." The word comes from an analogy to water flow: The velocity of the water defines a vector field; the trajectory of any one water molecule is a streamline. Electric and magnetic "field lines" are also streamlines.

MATLAB's **streamline** function will follow a vector field and make a graph of the resulting trajectory. The syntax is

```
streamline(x, y, Vx, Vy, startx, starty)
```

The first four arguments are the same as **quiver**: They specify the vector field to follow. The fifth argument can be a single entry with the x coordinate of the starting point, and similarly for the last one. Or `startx` and `starty` can be lists, if you wish to draw more than one streamline.

To illustrate the process, begin with the same vector field you drew in Section 5.9.1:

```
clear all; close all
gridMin = -2; gridMax = 2; gridStep = 0.2;
[Xfine, Yfine] = meshgrid(gridMin:(gridStep/10):gridMax);
Vx = Yfine; Vy = -Xfine;
streamline(Xfine, Yfine, Vx, Vy, [1, 0, .5], [1, 1.3, 1.7])
axis('equal')
```

Notice that this time, we chose a finer grid of points than before. Such a fine grid would have made a very cluttered picture of the vector field (too many arrows). But for the present purposes, a fine grid means more accurate results (MATLAB doesn't need to interpolate as much as it would otherwise).

Your Turn 5G

a. Perhaps the picture drawn in the preceding example was a bit too tame and predictable. Replace line 4 with

```
Vx = Yfine - 0.1*Xfine; Vy = -Xfine - 0.1*Yfine;
```

and explain the streamlines that you find.

b. For even greater excitement, replace line 4 with

```
Vx = Xfine; Vy = -Yfine;
```

and explain what happens.

CHAPTER 6

Second Computer Lab

In this lab, you'll use MATLAB to generate some two-dimensional random walks, plot a few such trajectories, and then take a look at the distribution of a large number of random walkers. Our goals are to

- Generate some random walk trajectories, each of which begins at the origin and proceeds in random diagonal steps according to

$$x_{n+1} = x_n \pm 1, \; y_{n+1} = y_n \pm 1. \tag{6.1}$$

- Plot the full trajectories of three such walks in three separate figures.
- Plot all the end points of the trajectories in a single figure to see how they are distributed.
- Compute the average position of the walkers and their average distance from the origin, and compare each to an expected result.

First review Section 5.2.

6.1 GENERATING AND PLOTTING TRAJECTORIES

Our first task is to create a random walk of 500 steps, each of the form in Equation 6.1. So our trajectory will be a list of 500 x values and 500 y values. It's good programming practice to let MATLAB know in advance how long the lists will be:

```
walklength = 500;
x = zeros(1, walklength); y = zeros(1, walklength);
```

Assignment:

a. *Use the ideas in Section 5.2 to make a random walk trajectory, then plot it.* To remove the default distortion in this picture, use the command **axis**('equal') after making the plot. Also, MATLAB may give each plot a different magnification. Review the Help for the command **xlim**. Then give each of your plots the same x and y limits, so that they may be compared properly.

b. *Now make two more such trajectories, and look at all three side by side.* (To create a new plot window, give the **figure** command before **plot**.)

6.2 PLOTTING THE DISPLACEMENT DISTRIBUTION

Your three plots all look different! And yet, there is some family resemblance between them. Let us begin to understand in what sense they resemble each other, by considering the question "How far does the random walker get after 500 steps?" More precisely, we want to know the distance from the starting point $(0, 0)$ to the ending point (x_{500}, y_{500}), for each of many random walks.

Instead of three walks, we now want many, say, 50. You could manually examine all 50 plots, but it would be hard to see the common features. Instead we'll ask MATLAB to generate them all, but only show us a summary.

One straightforward way to make 50 walks is to take the code you already wrote and "wrap" it in a **for** loop. Just before the loop's **end**, you could then say something like

```
xfinal(j) = x(500); yfinal(j) = y(500);
D(j) = sqrt(x(500)^2 + y(500)^2);
```

After the **end**, you'll have three lists: `xfinal`, `yfinal`, and `D`. (There's a clever, alternate construction that will run faster than this method. Feel free to discover it if you wish.)

Now we can summarize the results, in three ways. First, you have a lot of end points (`xfinal`, `yfinal` pairs), so you can make a "scatterplot" using **plot** or **scatter**. Then, you can look at the final displacements `D`, or at their squares.

Assignment:

a. *Now that you have a code that works, increase from 50 to 600 random walks.*[1] *Make a scatterplot of the end points.*

b. *Use* **hist** *to make a histogram of all the* `D` *values.*

c. *Also make a histogram of the values of the quantity* D^2.

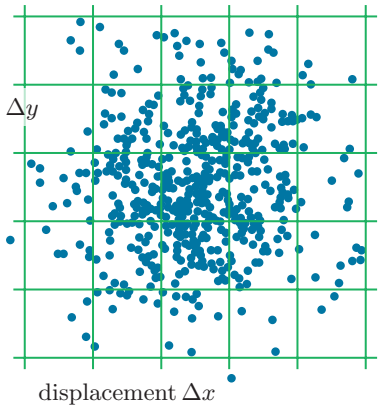
d. *Your answer to (c) may inspire a guess as to the mathematical form of the histogram. How could you change your graph to make it easier to test that guess? Try it.*

e. *Also use* **mean** *to find the average value of* D^2 *(the "mean-square displacement" of the 500-step walk).*

f. *Repeat to find the mean-square displacement of a 1000-step walk.*

Physical Models Chapter 5 discusses a theoretical expectation that agrees with what you found; it turns out that random walks are partially predictable after all. Out of all the randomness comes systematic behavior, partly visible in your answers to (b–d).

Experimental data also agree with these predictions. The random walk, although stripped of much of the complexity of real Brownian motion, nevertheless captures nontrivial aspects of reality that are not self-evident from its original formulation. See if your output qualitatively resembles the experimental data shown in *Physical Models* Chapter 3 (below) for the diffusion of a micrometer-size particle.



[1] See "Good practice," Section 2.4.4.

6.3 RARE EVENTS

6.3.1 The Poisson distribution

Imagine an extremely unfair coin that lands heads with probability ξ equal to 0.08 (not 0.5). Each trial consists of flipping the coin 100 times. You might expect that we'd then get "about 8" heads in each trial, although we could in principle get as few as 0, or as many as 100.

Physical Models Chapter 4 defines a discrete probability distribution called "Poisson." In the situation just described, the relevant formula is

$$\mathcal{P}(\ell) = \mathrm{e}^{-8} 8^{\ell} / (\ell!), \tag{6.2}$$

where ℓ is an integer ≥ 0.

Assignment:

a. *Before you get on with the flipping, plot this function for some interesting range of ℓ values.*

[*Hints:*
You need not take values of ℓ all the way out to infinity! You'll see that $\mathcal{P}(\ell)$ gets negligibly small after a while.
In MATLAB, the elements of a vector are always numbered 1, 2, 3,...; but our ℓ is an integer starting from *zero*. So the first entry of your variable, which you may call `Pvalues(1)`, should be the number $\mathcal{P}(0)$, and so on; thus, the value of $\mathcal{P}(\ell)$ is contained in the list element `Pvalues(ell + 1)`. To make your graph, you'll plot a list like `[0:20]` versus `Pvalues`.]

b. *Perform N "coin flip trials," each consisting of N sets of 100 flips, which land heads only 8% of the time.* [Good practice: Eventually you may want to take N to be a huge number. But while developing your code, make it not so huge, say, $N = 500$, so that your code will run fast.]

c. *Get* MATLAB *to count the number of heads for each set. Then use* **`hist`**, *which will create a histogram populated according to the frequencies of outcomes of those N trials. If you don't like what you see, consult* **`help hist`**. *(For example,* **`hist`** *may make a decision about how to bin the data that isn't what you want.)*

d. *Make a graph of the Poisson distribution (Equation 6.2 above) times N. What's the most probable outcome? Superimpose that plot on the histogram in (c). Repeat for $N = 20\,000$, and comment.* Click the (RUN) button over and over to see that your distribution is a bit different every time, and yet each plot has a general similarity to the others.

6.3.2 Waiting times

If we flip our imagined coin once every second, then our string of heads and tails becomes a time series, similar to something *Physical Models* calls a "Poisson process" (or "shot noise"). Flipping heads is a rather rare event, because $\xi = 0.08$. We expect long strings of tails, punctuated by occasional heads. Now a question arises: After we get a heads, how many many flips go by before we get the next heads? More precisely, what's the *distribution* of the "waiting times" from one heads to the next?

Here's one way to get MATLAB to answer that question. We can make a long list of ones and zeros, then search it for each occurrence of a 1 by using MATLAB's **`find`** function. This function's argument is a list of numbers; it returns the positions within that list of the nonzero entries. Experiment with **`find`**`([1, 0, 0, -1])` to understand this. You can subtract successive entries in that list to find the numbers of zeros that intervene between ones, then make a graph showing the frequencies of those outcomes. (Each waiting time is the length of a run of zeros, plus one.)

You should try to guess what this distribution will look like, before you get the answer. Nick reasoned, "Because heads is a rare outcome, once we get a tails we're likely to get a lot of them in a row, so short strings of zeros will be less probable than medium-long strings. But eventually we're bound to get a heads, so *very* long strings of 0's are also less common than medium-long strings." Think about it—is that the right reasoning? Now get your answer. If your output doesn't look like what you expected, think some more.

Assignment:

Construct a random string of length 1000, and plot the frequencies of runs of zeros with lengths 0, 1, 2, ..., as outlined above. Also make a semilog plot of these frequencies. Is this a familiar-looking function? Repeat with a random string of length 20 000.

CHAPTER 7

Still More Techniques

It's nice to know that the computer understands the problem. But I would like to understand it, too.
—Eugene Wigner

This short section introduces some tools for image manipulation and animation in MATLAB, and discusses symbolic (analytic) calculation in other systems.

7.1 IMAGES ARE ARRAYS OF NUMBERS

7.1.1 Basics

MATLAB can import and display image files. This makes sense because MATLAB works with arrays of numbers, and in the case of a black and white digital image, each pixel is simply a number corresponding to the intensity (mean photon rate) sensed at a point in the xy plane of the detector. A digital camera takes the light intensity in each pixel and reports it as an integer between 0 and $2^m - 1$, where m is the "bit depth." A common choice is $m = 8$ (256 distinct light levels). A color image consists of *three* such arrays, reporting the intensities of red, blue, and green light.

Because an image is just a two-dimensional array of numbers, we can take advantage of all of MATLAB's functions to do calculations on the data that the image represents. But to do that, we first need to give MATLAB access to some image data. Go to the Data Sets section of `http://www.macmillanhighered.com/physicalmodels1e`.[1] Find the entry called `catphoto` and get the file `bwCat.tif`. Use

```
A = imread('bwCat.tif');
```

to load the image (don't forget the semicolon!). Check that this is indeed represented as an array of numbers by looking at some particular elements of `A` (also check it in the Workspace Window).

The array `A` arrives in a numeric format that MATLAB finds inconvenient. So before proceeding, convert it to a more usual form by saying

```
A=single(A);
```

To view a black and white image using MATLAB, we first need to specify the "color map." Do this by typing

```
co = (0:255)/256;  colormap([co; co; co]');
```

(Notice the punctuation!) The values in the list `co` tell MATLAB how brightly to illuminate each pixel for each allowed value in the image. We convert it to a 256×3 array and call the function **`colormap`**, which sets a property of the currently open window (its color map).[2] Here we have made a linear choice; other choices would distort the image display in interesting ways, without changing the values in `A`.

Having set the color map, use

[1] Resources mentioned here are also maintained at a mirror site: `http://www.physics.upenn.edu/biophys/PMLS/Student`.
[2] The list `co` appears three times in **`colormap`** because, for a grayscale image, we always want equal amounts of red, green, and blue. If you don't set the color map, you'll get MATLAB's alarming default choice.

```
image(A);
```

to display the image. You can then export the image in a photographic format by using `Save>Save As...` from the figure window's menu. Or you can manipulate the image first by applying mathematical operations to `A`.

Your Turn 7A

Try this:
`A = (A < mean(A(:)))*256;`
Display the image again, and explain what you see.

7.1.2 Manipulate and embellish

After your calculations are done, your image array may not be a list of numbers that fits perfectly into the range from 1 to `size(colormap,1)`. The alternative display function `imagesc(A)` attempts to scale the entries in `A` to utilize your color map's full range.

The `image` and `imagesc` commands may distort your image's length/width ratio, just like other plotting commands. Follow them by

```
axis('image')
```

(similar to `axis('equal')`) to correct this. If you don't want axes and their labels, you can then also issue `axis('off')`.

By default, `image` displays your image as pixels, with larger column numbers to the right of smaller ones, and with larger row numbers lower down than smaller ones. This convention is natural if you think of your image as an array. But it conflicts with another expectation, that y values on a graph should increase as we move in the upward direction! If you wanted the latter convention, your image will appear flipped relative to what you expected. It can be unflipped by issuing the command `set(gca, 'YDir', 'normal')` after `image`.

7.2 ANIMATION

A picture may be worth a thousand words, but a motion picture can be even better than a bunch of still ones. There are several ways to create animated graphics in MATLAB. Here we show one simple method that outputs a GIF animation. The advantage of this method is that anyone can view the resulting graphic without having any special software. For example, any Web browser can open a local GIF file and view it; you can embed such a file into a Web page; and so on. The GIF format permits only a limited set of colors (no more than 256 distinct colors), but that's plenty for any animated graph.

The following example code

- Sets up axes with appropriate properties (line 9).
- Creates a graph for each desired video frame (line 14).
- Converts the graph to a frame, and then further converts to the indexed-color scheme used by the GIF format (lines 17–22).
- Writes the completed collection of frames to a file.

When we view the resulting file, the frames are presented in quick succession, creating a motion picture. In this example, the animation shows a traveling sine wave (see line 12).

```
%% William Parkin 4/2013 gifanimate.m   MATLAB R2014a:  make a gif animation
clear all; close all
```

```
%% initialize
% create our ranges
xo = 5;
x = -xo:.1:xo;
ct = (-xo):(.1):(xo);
%% create video frames
figure
set(gca, 'nextplot', 'replacechildren', 'Color', 'white','FontSize',14)
title(['time ranges from ', num2str(-xo), ' to ', num2str(xo)])
xlabel('x'); ylabel('sin(x-ct)')
for j = 1:size(ct,2),
    result = sin(x - ct(j));
    plot(x, result, 'r')
    ylim([-1.1, 1.1])          % fix this to avoid frame-to-frame jitter
% get frame to fill gif
    f = getframe(gcf);
    if j==1
        [im(:, : ,1 ,j), map] = rgb2ind(f.cdata, 256, 'nodither');
    else                       % every frame uses the same color map
        im(:, :, 1, j) = rgb2ind(f.cdata,map, 'nodither');
    end
end
%% save the gif animation
imwrite(im, map, 'DancingPeaks.gif', 'DelayTime', 0, 'LoopCount', inf)
```

7.3 ANALYTIC CALCULATIONS

MATLAB can also do analytic (symbolic) math. But for our purposes, it can be easier to turn to a free resource like `http://wolframalpha.com`. Go there, and try entering the examples below. Naturally this system doesn't use precisely the same syntax as MATLAB, but for simple calculations it's not too hard to adapt.

7.3.1 Integrals

The definite integral

$$\int_{-\infty}^{\infty} \mathrm{d}x \, \frac{1/\pi}{x^2+1} \frac{1/\pi}{(a-x)^2+1}$$

represents the convolution of two Cauchy distributions, discussed in *Physical Models* Chapter 5. After doing the integral, you get a number that depends on the value of a, that is, a function of a real variable a. It looks daunting, but try entering it into Wolfram Alpha (or *Mathematica*[3]) as

```
Integrate[1/(Pi^2*((a-x)^2+1)*(x^2+1)),{x,-Infinity,Infinity}]
```

Notice the syntax used here: The **Integrate** function accepts two arguments, enclosed in square brackets. The first is a symbolic representation of the integrand. The second is itself a list (created by the curly brackets) with three entries: The first is the variable to be integrated (we don't want the computer to integrate over a). The next two specify the range of integration. Note that in this system, predefined functions (**Integrate**) and constants (**Pi** and **Infinity**) are capitalized.

[3] Wolfram Alpha and *Mathematica* are registered trademarks of Wolfram Research, Inc.

Remarkably, the result of the integration is that this convolution is itself a Cauchy distribution, though broader than the two that we started with.

Here is another example relevant to *Physical Models* Chapter 6. The integral $\int_a^b \mathrm{d}x\, x^n(1-x)^{M-n}$ arises when we study credible intervals for the parameter of a Bernoulli trial, given that M trials yielded n "successes." We can make progress evaluating it by entering

```
Integrate[x^n * (1-x)^(M-n),{x}]
```

This time, the second argument of **Integrate** is a list with only one entry, the integration variable; because we don't specify any specific integration range, Wolfram Alpha responds with the indefinite integral, a function of x, M, and n. It may not seem helpful to be told that this function is the "incomplete beta function," but now we can see if MATLAB knows about that function (even if we don't!). It does, so we can now write code that evaluates the integral by evaluating that function. The answers that we obtain this way will generally be more accurate, and faster to run, than if we had directly attempted a brute-force numerical integration.

If all you need is the normalization integral, that's the case where we integrate from 0 to 1. Wolfram Alpha can evaluate this via

```
beta(1, 1 + n, 1 + k - n) - beta(0, 1 + n, 1 + k - n)
```

We learn that, for this case, the integral equals $\Gamma(n+1)\Gamma(k-n+1)/\Gamma(k+2)$, or equivalently $n!(k-n)!/(k+1)!$.

7.3.2 Sums

You may have forgotten the result of the infinite discrete sum $\sum_{j=0}^{M} j^3$, but typing

```
Sum[i^3, {i,0,M}]
```

into Wolfram Alpha tells you it's $M^2(M+1)^2/4$, a result you may need for computing the third moment of a Uniform probability distribution.

7.3.3 Ordinary differential equations

The differential equation $\mathrm{d}v/\mathrm{d}t = -Av + B\mathrm{e}^{-ct}$ arises in our study of virus dynamics, where A, B, and c are constants. We can ask Wolfram Alpha to solve it:

```
solve dv/dt = -A*v + B*exp(-c*t), v(0) = vzero
```

To get a definite solution, we need to specify an initial value; above we required that $v(t)$ should equal some constant `vzero` at time zero.

Another first-order ordinary differential equation, arising in the chemostat problem (*Physical Models* Chapter 9), can be solved similarly:

```
solve dx/dt = t-x, x(0) = 0
```

CHAPTER 8

Third Computer Lab: Import, Display, and Manipulate Image Data

This lab will show you how to import images into MATLAB and display them. We will also talk about an important operation in image analysis: convolution. If you've ever played with photographic software such as GIMP[1] (or a commercial alternative), you've probably already used convolutions to smooth or sharpen images, but perhaps you didn't know it. In fact, your eyes and brain are constantly doing convolutions whenever you look at anything!

Moreover, many kinds of lab data, although not photographic images, are nevertheless arrays of numbers with spatial meaning, and so can conveniently be represented to our brains as images. (Examples include atomic force microscopy data, computed tomography data, and so on.) Basic image manipulations, like the one in this lab, are useful in order to make those images more meaningful to humans.

Our goals are to

- Explore the effects of various kinds of local averaging on an image.
- See how to use such averaging to damp down noise.
- Use specialized filters to emphasize specific visual features in an image.

8.1 CONVOLUTION

Physical Models Chapter 4 defines convolution as an operation we sometimes perform on probability distributions, but it has others uses as well. MATLAB defines the two-dimensional, discrete convolution C of an image array I with a filter array F as a new array of numbers:

$$C(n_1, n_2) = \sum_{k_1,k_2} F(k_1, k_2) I(n_1 - k_1 + 1, n_2 - k_2 + 1), \tag{8.1}$$

where the ranges of the sums are over all values of k that refer to legal entries of both F and I. That is, all the indices must be larger than zero and not larger than the size of their respective arrays.[2]

Your Turn 8A

a. Consider the trivial transformation for which F is a 1×1 array with a single entry equal to 1. Make sure you understand why in this case C is exactly the same as I.
b. Suppose that the size of F is $m_1 \times m_2$, and that of I is $n_1 \times n_2$. Make sure you understand why the size of C is then $(m_1+n_1-1)\times(m_2+n_2-1)$.

Thus, when we convolve an image with a filter, we get another image. The expression in Equation 8.1 is a set of instructions for constructing this new image: To create each pixel in C, we take the pixels from a subset of the original image, multiply them by their respective weights assigned by the filter, and add up the

[1] GIMP is freeware: `http://www.gimp.org`.
[2] You may wonder about the +1's in this formula. MATLAB introduces this shift because its array indices always start from one, not zero.

result. It's a simple recipe that has powerful applications and can have very different results depending on the type of filter you design.

8.1.1 Averaging

The simplest filter we can choose assigns the same weight to each pixel in a fixed range. Make a 3×3 array F in which each element equals 1/9. (Why does it make sense to choose the value 1/9?) We'll call this array the "small square filter."

Go to the Data Sets section of `http://www.macmillanhighered.com/physicalmodels1e`.[3] Find the entry called `catphoto`, get the file `bwCat.mat`, and load it into MATLAB.

Assignment:

a. *Use the* ***conv2*** *command to convolve your new filter with the image you downloaded, and display the result. How does the image change?*

[*Hint:* You can retain the previous picture for comparison by first saying

```
figure(2);  colormap([co; co; co]');
```

Note that you need to declare the color map separately for each figure window that you create.]

b. *Repeat this procedure with a* 15×15 *array with an appropriate, constant value in each entry (the "large square filter"). How does the image change, and how does it compare with the result of the smaller filter?*

c. *Use the definition of convolution to show that the above procedure results in an image in which each pixel is the average of some of the neighboring pixels in the original.*

8.1.2 Smoothing with a Gaussian

Go to the Data Sets section of `http://www.macmillanhighered.com/physicalmodels1e`. Find the entry called `catphoto`, get the file `gaussFilt.mat`, and load it into MATLAB. You should now have a variable called `gauss`, which we'll call the "Gaussian filter." Review Section 5.4.1 on the function **surf**.

Assignment:

a. *Use* ***surf*** *to graph the array* `gauss`*. Find the result of the convolution of* `gauss` *with the original image and display it as an image.*

b. *Imagine the kind of surface plot you would have gotten from the filter you used in question 8.1.1b. Then explain in words how convolution with* `gauss` *differs from that filter, and why one might prefer* `gauss`*.*

8.2 DENOISING AN IMAGE

8.2.1 Random noise

Measuring instruments, including your eyes, inevitably introduce some randomness, or "noise." You can simulate this by making a noisy version of the original image. To do this, multiply each pixel in the original image by a random number. If you take the default result of **rand**, which is a random variable with mean 0.5, the image will get darker.

[3] Resources mentioned here are also maintained at a mirror site: `http://www.physics.upenn.edu/biophys/PMLS/Student`.

8.2.2 Suppressing noise by using convolution

Assignment:

a. *To correct for the darkening, multiply the noisy image by something a bit bigger, like* `1.2*rand()`. *[Hint: Review the useful* MATLAB *syntax* `.*` *for this (see Section 2.3.2).] Display the image again.*

b. *Apply each of the three filters from Sections 8.1.1–8.1.2 (small square, big square, Gaussian) to the noisy image from part (a). If they improve the pictures, explain why. Which one works the best? Why?*

8.3 EMPHASIZING FEATURES

You have probably news people say "Geeks at NASA's Jet Propulsion Laboratory have enhanced these images . . . ." Let's go.

In between "features" (real things of interest to us) and "noise" (random things), experimental images may contain things that are real, but not of interest to us. We may wish to deemphasize such things. Or we may wish to quantify some visual feature instead of leaving it subjective. A. Zemel and coauthors encountered such a situation when making fluorescence images of mesenchymal stem cells.[4] When subjected to mechanical stress (stretching), the cells polarize: The internal network of "stress fibers" begins to align in the direction of the stretch. Zemel and coauthors sought to quantify the extent to which the cell was polarized, at every point in the cell.

Go to the Data Sets section of `http://www.macmillanhighered.com/physicalmodels1e`.[5] Find the entry `stressFibers`, get the file `stressFibers.mat`, and load and view it as before. This file has a feature that we have not yet met: Look at the maximum and minimum numerical values of the numbers in the array `A`. They are not 0 and 256, respectively, so you'll need to modify something in your code.

The image shows the long, slender stress fibers. Let us apply a filter that emphasizes long, slender objects oriented vertically.

Assignment:

a. *Use the following code, and then explain why it creates the picture that results:*

```
[x, y] = meshgrid(-25:25);
gFilter = exp(-.5*(x.^2/2 + y.^2/45));
figure; surf(x, y, gFilter)
```

b. *The array* `gFilter` *that you created is nearly right, but it changes the overall brightness of an image. Use the following "black box" code to modify it slightly:*

```
laplaceFilter = [0, -1, 0; -1, 4, -1; 0, -1, 0];
K = conv2(laplaceFilter, gFilter);
combinedFilter = K(2:(end-1), 2:(end-1));
```

c. *Now use* **`conv2`** *to apply the filter* `combinedFilter` *to* `A`*, display your results, and comment.*

d. *In order to emphasize horizontal objects, repeat the above steps with a different choice for* `gFilter`. *Optional: Make a third choice, like the first two but emphasizing objects at 45° to vertical.*

[4] A fluorescent label for non-muscle myosin IIa was used to tag the stress fibers. See Zemel et al., 2010.

[5] Resources mentioned here are also maintained at a mirror site: `http://www.physics.upenn.edu/biophys/PMLS/Student`.

APPENDIX A

Answers to "Your Turn" questions

Your Turn 2A:
MATLAB starts filling in the (1,1) cell of the array. Each comma tabs over to the next column in the same row. The semicolon tabs down to the next row, first column. The result is an array with three columns and two rows (a "2×3" array).

Your Turn 2B:
linspace ends exactly at 10, and uses whatever spacing it needs to accomplish that.
The colon construction uses spacing exactly 1.5, and ends wherever it needs to in order not to exceed 10.

Your Turn 2C:
The first line sets up two arrays, each with one row and 20 columns, and then stacks them vertically, to create a 2×20 array. Finally, the apostrophe operator transposes this array to be 20×2.
The second line slices out the second column, and in it also removes the last entry.

Your Turn 2D:
We need to use concatenation to insert a blank space between words.

Your Turn 2E:
a.
```
y = exp(-(1:10).^2)
```
b.
```
jlist=0:10;
exp(-mu)*(mu.^jlist)./factorial(jlist)
```

Your Turn 3A:
```
num_points = 50;
x_list = linspace(0,4,num_points);
y_list = x_list.^2;
plot(x_list, y_list,'r','LineWidth',2)
set(gca,'FontSize',15)
title('My Little Plot')
xlabel('x')
ylabel('e^x')
```

Note that we set the font size for the current axes (**gca**); this applies to the axis numbers, and to the subsequent plot title and axis labels. You can instead do this separately for individual commands, if you don't want them all the same size.

Your Turn 3B:
```
legend('sin(2\pi x)','sin(4\pi x)','sin(6\pi x)')
```

Your Turn 5A: a. `stepx = stepx*2 - 1`
b. `x = cumsum(stepx)`

c.

```
walklength = 500;
stepx = rand(walklength, 1); stepy = rand(walklength, 1);
stepx = 2*stepx - 1; stepy = 2*stepy - 1;
plot(cumsum(stepx), cumsum(stepy))
axis('equal')
```

Your Turn 5B:

hist set up ten equally spaced bins. The first bin's lowest point is at 1, the smallest entry in the list. The last bin's highest point is at 10, the largest entry in the list. Thus, each bin's width is 9/10. The first bin's center is at its lower limit plus one half the bin width, or 1.45, so it catches all entries between 1 and 1.9. There is only one list entry in that range, so the first entry of `mycounts` gets the value 1, and so on.

Your Turn 5C:

```
[X, Y] = meshgrid(-1:.1:1, -1:.1:1);
surf(X, Y, X.^2+Y.^2)
```

Your Turn 5E:

a. **integral** expects a function that can operate item-by-item on a list of values.

c.

```
ind = 0;
for xmax = 0:.1:5; ind = ind + 1; %try various upper limits
result(ind) = integral(@(x)exp(-x.^2/2), 0, xmax); end
plot(0:.1:5, result)
```

Your Turn 5F:

At the point x, y, we have placed the vector with components $(y, -x)$. This arrow is always perpendicular to the vector from the origin to its base point, just like the velocity vector field of a rigid, spinning disk.

Your Turn 5G:

a. This example gives trajectories that all spiral into the origin, because we added a small component to $\boldsymbol{V}$ that points radially inward.

b. This example gives a fixed point at the origin of "saddle" type. One of the initial conditions runs smack into the fixed point at the origin, but the others veer away and run off to infinity.

Your Turn 7A:

See Section 2.1.3 for the syntax `A(:)`. It's a negative image with no shades of gray, just black and white. That's because everything darker than the mean lightness became fully white, and everything lighter than the mean became fully black.

APPENDIX B

More Errors and Error Messages

Now would be a good time to start making errors. Whenever you learn a new feature, you should try to make as many errors as possible, as soon as possible. When you make deliberate errors, you get to see what the error messages look like. Later, when you make accidental errors, you will know what the messages mean.

—A. Downey (2008)

Everyone makes mistakes. When MATLAB detects something's wrong, it attempts to tell you. But of course it doesn't really know what you are trying to accomplish. So if you enter something syntactically and mathematically correct, but not what you need to solve your problem, MATLAB can't point that out to you. Even when it does detect something wrong, it may not be able to tell you what you need to fix.

This appendix makes no attempt to catalog MATLAB's error messages. Instead, we show a few examples and decode them. After you start to understand how MATLAB interprets your code, you'll get insights into what other error messages mean, too. We also show a couple of examples of "silent" errors, to help you be alert for similar things. Several of these examples were drawn from Downey (2008).

B.1 ERRORS RELEVANT TO CHAPTER 1

• Suppose that you ignored the warning in Section 1.2.1, and cut and pasted the code `100^2` from this document into the Editor Window. (Try it.) When you click `Run`, MATLAB replies

```
Error: File: untitled.m Line: 1 Column: 4
Unexpected MATLAB operator.
```

Worse, if you paste the same thing into the Command Window, you may not even get an error message—just the wrong answer. Your only clue that something is amiss is that in each case, the ^ character appears in MATLAB in red. This indicates that it's an unknown character that just *looks* like the Shift-6 key that you wanted.

• Suppose that, as often happens, you use a single equals sign when you wanted to test for a relation, for example, typing **if** `q = 3, 'foo'; ` **else**, `'bar'; ` **end**. MATLAB replies

```
 if q = 3, 'foo'; else, 'bar'; end
      |
Error: The expression to the left of the equals sign is not a valid target for an assignment.
```

Here the vertical bar pinpoints where MATLAB's scanner had arrived when it detected something wrong. A single equals sign must have a single expression to its left, and "**if** `q`" is not a single expression: It is not a valid "target" for the assignment that the equals sign is requesting.

• The standard mathematical notation $0 < x < 10$ means "x is greater than 0 and less than 10." If you enter

```
x = 5; 0 < x < 10
```

the answer will be 1 (TRUE), and you may be satisfied. But try setting x equal to -1; you again get 1. Neither case generates any warning message.

What has happened is that MATLAB treats a relation symbol as a binary operator, and proceeds from left to right. When $x = -1$, it first evaluates `0 < x`, which equals 0 (FALSE). Then it evaluates `0 < 10`, which is TRUE! The right way to code this condition is to use an explicit **and** operator:

```
x=5; (0 < x) & (x < 10)
```

- MATLAB regards **hold** `on` as an abbreviation for **hold**`('on')`. That may save you a bit of typing, but rules must be followed consistently. So if you follow the standard mathematical notation $\sin\theta$ and type

```
sin theta
```

then MATLAB regards this as an abbreviation for **sin**`('theta')`, and responds with

```
Undefined function 'sin' for input arguments of type 'char'.
```

The message is telling you that you cannot evaluate the **sin** function with a string (character, or `char`) argument.

B.2 ERRORS RELEVANT TO CHAPTER 3

- A common mistake with **integral**, **fzero**, **ode45**, and their kin is to supply the name of a function (such as **sin**) where a handle is required (**@sin**). Try making this error intentionally with the code listed in Section 5.7.1 to see the rather confusing message that you get:

```
Error using cos
Not enough input arguments.

Error in untitled (line 4)
result(ind) = integral(cos, 0, xmax);
```

The same mistake (omitting `@`) can generate various other, equally confusing messages.

It's best to read such messages starting from the bottom. In this case, the last line is the line of your code that initiated the error; it is line 4. Moving upward in the message, we see that the problem has something to do with **cos**. That's your clue to look at how you used it, and, if you're lucky, you'll figure out what's wrong. If you don't, the next step might be **help integral** to recall the syntax of this function.

- Another common problem is to send **integral** a function that can only process single numbers, for example:

```
integral(@(x) x^2, 0, 1)
```

integral tries to evaluate your function on a list, and another confusing error message results.

- When creating your own functions, it's easy to forget the crucial last step of assigning a value to the function's output variable before terminating. Try this: Create a file called `badFn.m`, containing

```
function outValue = badFn(x)
y = x^2
```

Remember to save the file before invoking the function. This code computes the square of its argument, and even prints it out. So if you type `badFn(2)`, you'll see what you expect. But `y = badFn(2)` gives the error

```
Error in badFn (line 2)
y = x^2
Output argument "outValue" (and maybe others) not assigned during call to
"/.../badFn.m>badFn".
```

The explanation is that, although `badFn` did assign the answer to a variable called `y`, that variable is internal to the function and disappears when the function terminates. In fact, `badFn` returns *no* value. The way to get an answer out of your function, and return it to the calling code, is to assign its value to the output variable that you named in the **`function`** command (here `outValue`).

Acknowledgments

Many students and colleagues taught us tricks that ended up in this tutorial. PN acknowledges particularly helpful suggestions from my teaching assistants André Brown, Ed Banigan, David Chow, Matt Hickman, Jan Homann, and Asja Radja. Additional code was written by William Parkin. André Brown contributed the stress filaments image data.

Major sections of this tutorial are based on fragments originally written by Jesse Kinder, Jason Prentice, and Tom Dodson. The front page graphic was realized in MATLAB by Jason Prentice.

Steve Hagen and Teresa Miller gave many invaluable suggestions for clarifications, as did all my students over several years.

This tutorial was set in LaTeX using the `listings` package written by Prof. Jobst Hoffman, who also kindly supplied personal help. Jodi Isman supervised production.

This tutorial is partially based on work supported by the United States National Science Foundation under Grants EF–0928048 and DMR–0832802. The Aspen Center for Physics, which is supported by NSF grant PHYS–1066293, also helped to bring it to completion. Any opinions, findings, conclusions, jokes, or recommendations expressed in this book are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The University of Pennsylvania Research Foundation provided additional support for this project.

References

There are innumerable other introductions to MATLAB, including "Getting Started with MATLAB" in the main built-in Help system (**doc** MATLAB). So instead of citing many such guides, we will mention just one here. For the many advanced features that you'll need, again you'll find good, systematic treatments in the built-in Help system. However, you may not need a systematic treatment; starting from the foundation in this tutorial, you may be able to get the advanced material you need for a particular problem just from Web searches.

Many extensions to MATLAB have been written as "toolboxes," open-source add-ons. One relevant example is `http://www.sbtoolbox.org`, described in Schmidt & Jirstrand, 2006.

Downey, A B. 2008. *Physical modeling in* MATLAB. Needham MA: Green Tea Press. `http://www.greenteapress.com/matlab/`.

Kinder, J M, & Nelson, P C. 2015. *Student's guide to physical modeling with Python.* `http://lulu.com`.

Nelson, P. 2015. *Physical models of living systems.* New York: W. H. Freeman and Co.

Rougier, N P, Droettboom, M, & Bourne, P E. 2014. Ten simple rules for better figures. *PLoS Comput Biol*, **10**(9), e1003833.

Schmidt, H, & Jirstrand, M. 2006. Systems Biology Toolbox for MATLAB: a computational platform for research in systems biology. *Bioinformatics*, **22**(4), 514–515.

Zemel, A, Rehfeldt, F, Brown, A E X, Discher, D E, & Safran, S A. 2010. Optimal matrix rigidity for stress-fibre polarization in stem cells. *Nature Physics*, **6**(6), 468–473.

Index