ECE-210-B Homework #4

Functions & Objects

Cat Van West, Spring 2024

Contents

Real programming languages have these, so matlab, in its efforts to gain respect, has them too. I entirely skipped objects last year, but I feel that was a disservice to the students in the end – Mathworks is pushing object-oriented apis for things like filter design now, so it's best to get used to it.

There are a few other things wrapped into this assignment, like structures, datatypes, control flow, etc. These topics are as close as matlab gets to growing up.

1 The Assignment

Mostly on functions, a little on classes if you like. Plus a bit of plotting. Could be worse. ;)

My Sinc is Broken We'll examine a function that you will see... perhaps too much in your tenure here. It's the Fourier transform of a rectangular pulse, the minimal-bandwidth pulse shape, it's

$$\operatorname{sinc} x = \begin{cases} 1, & x = 0 \\ \dfrac{\sin \pi x}{\pi x}, & x \neq 0 \end{cases}.$$

(This is sometimes defined as ${}^{\sin x}/_{x}$, but the above definition is the one MATLAB uses.) I For the following, you're free to use the provided `sinc` from MATLAB's signal processing toolbox or to define it yourself.

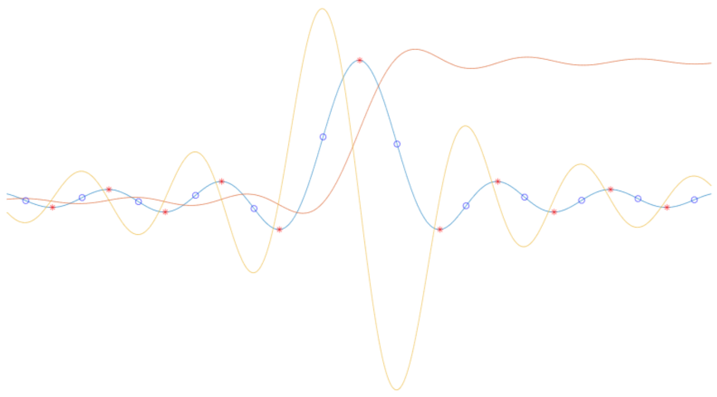
Note: if you write any of the functions below as *implicit* functions (which are wonderful things!), they will not be visible inside ordinary functions (defined with `function`). One way around this is to pass the implicit functions as arguments; another is…just don't use implicit functions.

Note: you're welcome to use MATLAB's control flow constructs for these, but they are not necessary – all of these functions are implementable without so much as an `if` statement.

1. Write functions `deriv(y, x)` and `antideriv(y, x)` which perform numerical differentiation and integration, respectively, on the function $y(x)$. These should each return vectors of the same length as their input (read: pad appropriately). You've done this before, now just make it… functions.
2. Write a function `switchsign(x)` which returns a logical array of the same length of `x` which is true wherever `x` switches sign. For example, `switchsign([-1 2 1 0 1 -3])` would return `[0 1 0 0 0 1]`.
3. Write functions `extrema(y, x)` and `inflections(y, x)` which use the first and second derivative tests to find local extrema and inflection points, respectively. These functions should each output two vectors, representing the x and y coordinates of these points.
4. Use these lovely functions to make a lovely plot of $\operatorname{sinc} x$ over a sufficiently large range showing its antiderivative, its derivative, its extrema, and its inflection points. Use a plotting command like

```
plot(x, y, x, antiderivative, x, derivative, ...
        x_extrema, y_extrema, 'r*', ...
        x_inflections, y_inflections, 'bo');
```

to do so. It should look something like this:



Note: if any features don't appear, try changing the number of points you're using in 1–3.

Objectification This one will require reading a few of the docs! You'll follow an example of object-oriented plotting (modifying properties of graphics objects using `.` notation, effectively).

1. Go to https://www.mathworks.com/help/matlab/visualize/creating-the-matlab-logo.html and go through their code to create the MATLAB logo.
2. Change the position of the lights. Change the colormap. Change the function. See how weird you can get it to look.
3. Use the object-oriented interface to `subplot` (the signature listed as `ax = subplot(...);` in Mathworks' documentation) to display the original MATLAB logo and your modifications side-by-side.
4. Use `get` one one of the objects you've created (the figure, the surface, the axes, etc.) to examine all of its properties. Pick three and try modifying them. What happens? Leave me a note!

(Extra Credit, +.5) Look Me in the Iris (Shamelessly stolen from Jon Lam, because it's a good exercise in MATLAB OOP. There's some info on user-defined classes in the lecture notes, and more in Mathworks' documentation – this tutorial might be a place to start.)

Take a minute to research the fisheriris dataset (it's a popular dataset, you might see it again in an ML class).

Load in the fisheriris dataset with `load fisheriris`. You should obtain (in your workspace) a 150 $\times$ 4 matrix called `meas`, as well as a 150 $\times$ 1 cell array called `species`. The i-th row in `species` corresponds to the i-th row in `meas`.

(a) Create a class `Flower` in its own file. (Look up the syntax for user-defined classes!) You should have the following properties:

Property	Value
`petalWidth`	double
`petalLength`	double
`sepalWidth`	double
`sepalLength`	double
`species`	char (array)

Note that you do not need to specify the data type of the properties when you are declaring a class; the following properties are just here to impress on you what type of properties you would be expecting.

(b) Create a constructor for `Flower`. It should take in four doubles and a character array corresponding, in order, to the five properties of your class.

(c) Now, import the entries from `meas` and `species` into a 150 $\times$ 1 cell array of `Flower` instances. You can use a `for` loop to import the entries. Don't worry about vectorizing this – MATLAB syntax makes it messy.

Note that the name of the species are stored in a cell aray; make sure to extract the character array from the cell before storing them in the `Flower` instances.

(e) Create a method called `report` in the `Flower` class that will print out details about the object on the command window.

This function takes no arguments and returns nothing. It should print out a statement of the following form, but with the correct values of the object's properties:

```
The length and width of its sepal are 5.1 cm and
3.5 cm respectively, while the length and width
of its petall are 1.4 cm and 0.2 cm respectively.
It belongs to the setosa species.
```

Demonstrate that this function works on the 51st `Flower` object.

A Notes on Functions

While objects in MATLAB are, at best, hurriedly cobbled together and awkward, functions are exceedingly useful and should never be far from your fingertips. They allow abstracting away useful sections of code for later reuse, after all – who wants to write more than they have to?

A.1 Implicit Functions

Implicit functions are those declared with the syntax `@()`. They take any number of arguments (zero up) and evaluate the expression in their body with those arguments (along with whatever they captured from the surrounding environment). The syntax looks like this:

$$\underbrace{\texttt{pow}}_{\text{function name}} \ \texttt{=}\ \texttt{@}\underbrace{\texttt{(x, p)}}_{\text{arguments}}\ \underbrace{\texttt{x.\^{}p}}_{\text{body}}\texttt{;}$$

When called, the parameters are simply substituted into the body. An extremely simple example, including capture:

```
x = 3;
add_x = @(y) x + y; % x captured here
add_x(9); % prints 12

x = 4; % no effect, as x was captured earlier
add_x(9); % still prints 12
```

These functions behave much like anything else you can bind to a variable, and their value can be passed around, into and out of functions, created on the fly, etc. They're useful for short bits of calculation that you want to name – `volt2db = @(v) 20*log10(v);` might be a useful one, for instance.

A.2 "Ordinary" Functions

Ordinary functions are those defined using the keyword `function`. They're defined either at the end of the file in which they're used (local functions) or at the start of a `.m` file of the same name (public functions; see the lecture notes for an example). These functions may have several statements in their body and return several values. Most of the syntax is in the lecture notes, so I'll just call out the start of single-return

$$\texttt{function}\ \underbrace{\texttt{d}}_{\text{returned variable}}\ =\ \underbrace{\texttt{manhattan\_distance}}_{\text{function name}}\underbrace{\texttt{(point1, point2)}}_{\text{parameters}}$$

and multiple-return

$$\texttt{function}\ \underbrace{\texttt{[r, theta]}}_{\text{returned variables}}\ =\ \underbrace{\texttt{convert\_to\_polar}}_{\text{function name}}\underbrace{\texttt{(x, y)}}_{\text{parameters}}$$

functions.

The parameters are visible in the function body as variable, and the return values may be set by assigning to them within the function body:

```
function d = manhattan_distance(point_1, point_2)
        x_distance = abs(point_1(1) - point_2(1));
        y_distance = abs(point_1(2) - point_2(2));
        d = max([x_distance, y_distance]);
end
```

(Note that this function is, really, a one-liner, *and* is not vector-safe in its current implementation. The above is merely to illustrate syntax.)

A.3 The Matlab Search Path

Jacob has better documentation for this than I do – see his first lesson. Basically, the search path defines where MATLAB looks for files, and thus where public ordinary functions will be discovered. If you want to write a function in its own file, make sure that file is in this path, else you won't be able to use it! The current working directory (whatever folder is shown in the file explorer window) is in that path, so if you keep everything in the same directory, you'll be ok (which should be fine for this class).

B Vector Considerations

When writing functions, you sholud be cognizant of how people might use them. This seems obvious, but let's examine the above naïve implementation of `manhattan_distance` to see how MATLAB, specifically, can get us into trouble.

The function makes an assumption about its arguments: it wants them formatted as vectors whose first component is an x-coordinate, and whose second component is a y-coordinate. But MATLAB is a vector program, and most functions work in a vectorized way. Presumably we'd like to compute distances for a whole lot of points at once if we needed to, right? In that case, we'd have to amend our definition and make different assumptions about the data – perhaps that we'll get lists of xy-values:

```
% expects each row of p1, p2 to contain an x and a y
% coordinate of the point
function d = manhattan_distance(p1, p2)
        x_dists = abs(p1(:, 1) - p2(:, 1));
        y_dists = abs(p1(:, 2) - p2(:, 2));
        d = max([x_dists, y_dists], [], 2); % over second dim.
end
```

Great. But that only works in two dimensions, and what if we want three?

```
% expects each row of p1, p2 to contain a point in n
% dimensions. p1 & p2 should be the same shape.
function d = manhattan_distance(p1, p2)
        dists = abs(p1 - p2);
        d = max(dists, [], 2);
end
```

Interestingly, it got shorter once we realized that we're just subtracting corresponding elements of the inputs. This function is really a one-liner: we could write

```
% expects each row of p1, p2 to contain a point in n
% dimensions. p1 & p2 should be the same shape.
manhattan_distance = @(p1, p2) max(abs(p1 - p2), [], 2);
```

if we only needed it locally.

One more consideration that can bite you: not using a dotted operation when you really should. For instance,

```
normalize = @(vec) vec/sqrt(sum(vec));
```

breaks when you feed in a two-dimensional list of vectors. Changing `/` to `./` will fix it, but be aware of which dimension `sum` is operating over when you do so!