ECE-210-B Homework #5

Advanced Manipulations

Cat Van West, Spring 2024

Contents

...broadcasting & indexing! There's a reason we had a second lesson dedicated to how vectors work in MATLAB– the techniques we've covered are powerful & efficient tools when wielded appropriately. This assignment should (hopefully) convince you of this.

Note that several of these questions have line count targets. (I won't take off anything if you go over by a little bit – they're just to ensure you're in the right ballpark.) These are generally set at the number of lines I used to solve it, unless I did something really tricky. While "a line of code" is a pretty fuzzy quantity, these should be doable in their stated line counts without sacrificing style. Vectorize, vectorize, vectorize... ;)

1 The Assignment

This one is a bit longer than the last two – broadcasting and vectorization are *extremely* important, and they merit a bit of meditation.

On The Air No reallocation, no loops, no help, no sanity... a typical day at the Copper Onion. Solve the following with vectorized operations. Broadcasting and `meshgrid` will be helpful here!

1. Create the matrix

$$R = \begin{pmatrix} -9+9i & -8+9i & \dots & 8+9i & 9+9i \\ -9+8i & -8+8i & \dots & 8+8i & 9+8i \\ \vdots & \vdots & & \vdots & \vdots \\ -9-8i & -8-8i & \dots & 8-8i & 9-8i \\ -9-9i & -8-9i & \dots & 8-9i & 9-9i \end{pmatrix}$$

   (effectively a matrix of coordinates) in the tidiest way you can. (*2 lines, 1 if* very *clever*)

2. Approximate

$$\int_0^1 \sin(t^n)\, dt$$

   for $n = 1, 2, \dots, 50$. Use `trapz` (the non-cumulative cousin of `cumtrapz`) and at least 10000 sample points. Try creating a vector of points and a vector of n values, combine them via broadcasting or `meshgrid`, and go from there. (*3 lines*)

3. Let's do something weird – like plot a sphere.

   (a) Create two matrices, `theta` and `phi`, as follows:

```
[theta, phi] = meshgrid(...
        linspace(0, 2*pi, 5), ...
        linspace(0, pi, 5));
```

   (This counts as a one-liner in my book.) These will serve as our spherical coordinates θ and ϕ.

   (b) Find the x, y, and z coordinates they correspond to using the spherical coordinate conversions

$$\begin{aligned} x &= \sin\phi\cos\theta, \\ y &= \sin\phi\sin\theta, \\ z &= \cos\phi. \end{aligned}$$

These should take the form of function calls and elementwise multiplies.

(c) Plot these points using `surf`. Call `pbaspect` with appropriate arguments to make the axes nice and square. Provide a proper title. All the usual plotting stuff.

Under The Sea Perform the following operations with overly clever indexing. All line counts *include* creation of data vectors/matrices.

1. Find the values in `sin(linspace(0, 5, 100).*linspace(-5, 0, 100).')` which are closest to $1/2$ (there are a few ties for closest), along with their 2D indices. (`min`, `abs`, and `find` may be useful!) (*3 lines*)
2. Approximate the volume contained above by the surface $z = e^{-(1-xy)^2}$ and below by the surface $z = \frac{1}{4}\sqrt{x^2 + y^2}$. You'll have to use logical indexing to selectively sum up volume where one curve is actually over the other. (Serious bonus points if you 3D-plot the enclosed volume!) (*3 lines, no plot*)

I Need a Vacation And now, the weather! Create the following matrices, using `figure` and `imshow` to visualize them. All these matrices are 256×256, and either contain floating-point values or logicals. Add a little comment describing each one (the meaning of "describe" is up to you, beyond basic technical information). *Note:* the notation a_{ij} means the element of A in row i, column j – i.e., `A(i_index, j_index)`.

1. A where a_{ij} is true iff $\sqrt{(i-99)^2 + (j-99)^2} < 29$.
2. B where b_{ij} is true iff $\sqrt{(i-62)^2 + (j-62)^2} < 58$.
3. C where c_{ij} is true iff $i - 4\sin(j/10) > 200$.
4. $S =$ `rand(256, 256, 3)` with its two "lower" layers (3rd index 1 and 2) zeroed out.
5. $M = A \cap B^C$ (^{C} denoting logical complement – in other words, `~`).
6. $Z = (C \cdot S) + M$ ($\cdot$ denoting elementwise multiply).

A Meshes & Broadcasting

The concepts of broadcasting & `meshgrid` are often tricky to understand, so some illustrated examples are likely in order. I'll start with `meshgrid`, because it comes naturally out of evaluating functions using matrices of coordinates – however, broadcasting (in my opinion) is a more versatile technique (and is often faster in practice), so keep reading to the end!

A.1 The Grid

Suppose you want to find the area under the surface $z = x + y$ over the unit square $(0 \leqslant x \leqslant 1,\ 0 \leqslant y \leqslant 1)$. In order to do this in MATLAB, we could do a discrete Riemann sum: evaluate the function at points on a grid, transform each of those points into a rectangular prism, and sum the areas of those prisms. So… how do we create the grid of points? One way might be as follows:

```
x = [ 0 .2 .4 .6 .8  1; ...
      0 .2 .4 .6 .8  1; ...
      0 .2 .4 .6 .8  1; ...
      0 .2 .4 .6 .8  1; ...
      0 .2 .4 .6 .8  1; ...
      0 .2 .4 .6 .8  1];
y = [ 0  0  0  0  0  0; ...
     .2 .2 .2 .2 .2 .2; ...
     .4 .4 .4 .4 .4 .4; ...
     .6 .6 .6 .6 .6 .6; ...
     .8 .8 .8 .8 .8 .8; ...
      1  1  1  1  1  1]
z = x + y;
```

Note the pattern in x and y here: each element of x contains its own x-coordinate, and each element of y contains its own y-coordinate (flipped vertically). We're treating these matrices as gridded data: each element of x and y contains information about its corresponding point in the plane (namely, one of its coordinates). Evaluating the expression for z elementwise thus inserts the correct x & y coordinates at the correct positions in the matrix: the upper left corner has $x = 0$ and $y = 0$, so $z = 0$; the lower right corner has $x = 1$ and $y = 1$, so $z = 2$ – in general, each of the coordinates winds up in the right place, and z becomes

```
z =

     0    .2    .4    .6    .8   1.0
    .2    .4    .6    .8   1.0   1.2
    .4    .6    .8   1.0   1.2   1.4
    .6    .8   1.0   1.2   1.4   1.6
    .8   1.0   1.2   1.4   1.6   1.8
   1.0   1.2   1.4   1.6   1.8   2.0
```

which is a grid of evaluations of $z = x + y$. This can be used for any function (the notes show this technique used to evaluate $z = \exp(-x^2 - y^2)$, which translates to MATLAB code as `z = exp(-x.^2 - y.^2)`), and, as such, is a handy technique. So much so that it has a name: those *x & y* matrices we created form a *mesh*, two matrices that form a gridded set of coordinates, and such meshes are typically generated not by hand (which is a pain) but by `meshgrid`. The following code generates identical matrices to those handcrafted above:

```
[x, y] = meshgrid(0:.2:1); % same as before
z = x + y;
```

`meshgrid` takes in a vector (or two vectors, for non-square matrices) of coordinates and creates matrices which function as x, y coordinate inputs to vectorized functions. Try it out and see what it does!

A.2 How To Broadcast

`meshgrid` is usually only used where meshes are really necessary – places where being able to traverse the space of evaluation in two or three dimensions is necessary. Surface plots, like the sphere in this assignment, are one of those places: each z-coordinate needs an easy association with an x and y coordinate so that the plotting engine knows where the hell to put all the points.

If, however, we just want the results of the evaluation (or want another technique to construct matrices really quickly), *broadcasting* is an alternative. You met broadcasting in the very first assignment when you performed $C + A$ – an operation notably *not* valid in mathematics, because C and A were different shapes! They were, however, *broadcastable* shapes – C was a 4×4 matrix and A was a 4×1 column vector, so MATLAB simply repeated A four times and added it to C:

$$\begin{aligned}
C + A &\to \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix} + \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \\
&\to \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix} + \left[\begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \right] \\
&\to \begin{pmatrix} c_{11} + w & c_{12} + w & c_{13} + w & c_{14} + w \\ c_{21} + x & c_{22} + x & c_{23} + x & c_{24} + x \\ c_{31} + y & c_{32} + y & c_{33} + y & c_{34} + y \\ c_{41} + z & c_{42} + z & c_{43} + z & c_{44} + z \end{pmatrix}.
\end{aligned}$$

Any two matrices with dimensions that can be made to match by this sort of repetition are broadcastable, and any elementwise operation may cause broadcasting to occur. For example, we could compute an elementary multiplication table very quickly using `.*`:

```
first_factor = 0:9; % row vector, 1×10
second_factor = (0:9).'; % column vector (transposed), 10×1
mult_table = first_factor .* second_factor; % becomes 10×10
```

To perform this calculation, MATLAB recognizes that if `first_factor` is repeated 10 times vertically and `second_factor` is repeated 10 times horizontally, both will be the same size, so it (effectively) does that, producing a 10×10 result.

These examples are, of course, simple, to facilitate understanding. There's more that can be done… but that's up to you!