PHYS 210: Introduction to Computational Physics Fall 2013 Homework 3

Version 2, October 30, 2013

Due date (revised): Thursday, November 14, 11:59 PM

PLEASE report all bug reports, comments, gripes etc. to Matt: `choptuik@physics.ubc.ca`

Please make careful note of the following information and instructions, which continue on page 2:

1. The following assignment requires you to implement several `Matlab` functions and scripts, all of which will be coded in text files (`.m` files). As usual, these files must be (1) named precisely as specified, and (2) located in the correct subdirectories. An inventory of requisite files is included at the end of each problem description. Should you create other `.m` files within the subdirectories for test purposes etc. it is fine to leave them there.

2. The three problems comprising the assignment vary *significantly* in terms of the time and effort you can expect to spend on their solution; please take this into consideration in your overall allocation of time for completion of the problem set.

   With luck, you will find Prob. 1 straightforward. Two of the functions that you are to code in this case are `Matlab` versions of `Maple` procedures that your wrote for the second assignment (albeit the coding of `proc_even_odd` requires some "manual" error checking that some of you may find slightly tricky). Although you may want to program these functions "from scratch", the problem can also be viewed as an exercise in translation of code from one language to another.

   Prob. 2 should also be straightforward and can be solved with just a few lines of `Matlab`.

   Prob. 3, on the other hand, is where I expect most/all of you to expend the bulk of your time and effort. In addition to the implementation of a core function, it requires that you write three distinct scripts which use that function to carry out computational "experiments", and to perform simple analyses of the results. In this sense it is quite different and more "realistic" than the type of programming you have done in this course thus far, especially in the assignments.

   This component of the homework has been designed in part to help you gain a level of proficiency in `Matlab` programming that will be necessary for the successful completion of your term projects. Thus, especially if you are new to programming, the earlier you get to work on it, the better off you will be in terms of completing your project expeditiously.

3. **Crucial!! Statement terminators:** Again, recall that you can enable/disable output from `Matlab` statements by omitting/including the semi-colon statement terminators. Enabling output can be very useful in code development and debugging. However, with the single exception of the display of the fit coefficients in Problem 3.2.4, *all* of the statements in the final versions of *all* functions and scripts that you write for this homework *must* be terminated with semi-colons so that execution of the code produces *no* output to the command window. Lest there be confusion about this point, the appearance of plotting windows, and the creation of JPEG files do *not* count as output in this respect; i.e. if the problem requires that a function/script makes a plot (which will be displayed in a plotting window), or that it saves a JPEG version of a plot, then it *must* do so.

   This is an extremely important point, and failure to adhere to this aspect of the instructions may lead to a significant penalty.

4. **Comments in code**: *No* comments are required for the functions in Problem 1. For the other two problems, the functions and scripts do not have to be extensively commented. Brief comments at the beginning of each `.m` file describing the purpose of the code defined therein, including, for the case of functions, the purpose of the input and output arguments, will suffice.

5. **Error checking**: The only function that must perform checks for validity of input arguments is `prod_even_odd` (Problem 1.3).

6. **Where to start `Matlab`**: As usual, you must create a separate subdirectory for each problem in which you will then create one or more `.m` files that contain the definitions of functions and scripts. When you are working on a particular question, be sure to start `Matlab` from within that problem's subdirectory so that `Matlab` can find the `.m` files for the problem.

7. **Plot titles/axis labels/legends**: Several of the questions involve the generation of plots with specified titles, axis labels and/or legends. The required annotations are given in typewriter font and are enclosed in quotes, e.g. "`This is a plot title`". Do *not* include the quotes in the annotations.

8. **Example functions/scripts:** The problem descriptions make reference to various `.m` files located in `~phys210/matlab/` that provide usage examples for `Matlab` functions which you will use in completing the problem set. Provided that you have copied the course default `setup.m` file to your directory `~/Documents/MATLAB`, you should be able to view the contents of any of the `.m` files and use the scripts/functions that they define from within any `Matlab` session. For example,

```
>> type errorreturn
```

   should display

```
function rval = errorreturn(a, b)
% errorreturn returns the sum of its two inargs, providing that the first
% is strictly positive, otherwise it prints an error message and
% exits using the return statement.

   % Assign a "default" value to the output argument to ensure that it
   % IS defined before the function returns.
   rval = NaN;

   if a <= 0
      % Print the error message and return

      % See lab notes, 'help fprintf' and/or online info for 'fprintf'
      % for additional details on fprintf usage

      fprintf('myreturn: First argument must be > 0');
      return;

   end

   rval = a + b;
end
```

   while executing

```
>> errorreturn(-2,3)
```

   should produce the output.

```
myreturn: First argument must be > 0
ans =

   NaN
```

   Should this not work for you, refer to the instructions given in Section 4.3 of the `Matlab` Programming Lab Notes, which are available via the Syllabus section of the course home page, or directly at

   `http://laplace.phas.ubc.ca/210/Labs/matlab-pgm-lab.txt`

9. **Working on a non-lab computer**: First, this assumes that your system configuration supports X11 (e.g. you have a Mac, or are running Windows with `Xming` installed), so that you can use `gedit` to create/edit `.m` files, and so that you will be able to do at least simple plotting with `Matlab`.

   If you haven't yet configured your personal machines so that X11 applications are supported, you are again encouraged to do so, not least since this will also allow you to work on your term projects outside the lab.

   You can `ssh` into `hyper` and run the *command-line* version of `Matlab` via

```
   % matlab -nodesktop -nosplash
```

   or, if you have completed the appropriate lab exercise, by simply typing `mat`. You can *try* to use GUI version of `Matlab` remotely, but will probably find the performance sluggish.

   However, and again assuming that X11 is working on your machine(s), the simple plots that you need to make should display without difficulty.

10. As always, should any aspect of the assignment be unclear to you, if you feel that you have been struggling with something for an inordinate amount of time, etc., please seek help!

Problem 1: Make the subdirectory ~`hw3/a1`. Within that subdirectory create `.m` files that contain definitions of `Matlab` functions with headers and functionalities as specified. As noted previously, the function definitions do not need to include comments.

1. *Header*: `function val = fcn5(x)`
   Functionality: `fcn5` returns a value defined as follows

$$\begin{array}{ccc} 0 & \text{for} & x \le -1 \\ 2-4\left(-x-\frac{1}{2}\right) & \text{for} & -1 < x \le -\frac{1}{2} \\ -4x & \text{for} & -\frac{1}{2} < x \le 0 \\ 4x & \text{for} & 0 < x \le \frac{1}{2} \\ 2-4\left(x-\frac{1}{2}\right) & \text{for} & \frac{1}{2} < x \le 1 \\ 0 & \text{for} & x > 1 \end{array}$$

2. *Header*: `function plot5(xmin, xmax, nx)`
   Functionality: `plot5` uses the `Matlab plot` function to generate a *single* graph that plots both `fcn5` and `-fcn5`—where `fcn5` is the procedure defined above—on the domain `xmin` $\le x \le$ `xmax`, and where the functions are evaluated at `nx` equally spaced values of x. `fcn5` is to be graphed with a red line, `-fcn5` with a green line. When the function is invoked, the plot should appear in the usual `Matlab` plotting window that has a title *Figure 1*. Give the plot a title "`Plot of fcn5 and -fcn5`", x and y axes labels "`x`" and "`y`", respectively, and a legend with entries "`fcn5(x)`" and "`-fcn5(x)`"
   Hints:
   (a) See ~`phys210/matlab/tplot.m` for a script that illustrates some basic plotting techniques/operations. In addition, ~`phys210/matlab/plotex.m`, which we examined in the lab, contains further examples and, of course, you can find many more details and examples via the `help` command and searches for online material.
   (b) In the labs we have seen how we can easily generate a vector of equally spaced values in `Matlab`. However, since `fcn5` is *not* a builtin in function, consider using a loop to make a vector (or vectors) of the "y values" that the function is to plot.

3. *Header*: `function val = prod_even_odd(m, n)`
   Functionality: Given two integers, m and n, with $n > m$, `prod_even_odd` returns a value given by

$$\left(\sum_{i=m}^{n} i \mid i \text{ is even}\right)\left(\sum_{i=m}^{n} i \mid i \text{ is odd}\right)$$

   For example

$$\texttt{prod\_even\_odd(3, 12)} = (4+6+8+10+12)(3+5+7+9+11) = 1400$$

   Your implementation of `prod_even_odd` must make the following checks to ensure that the input arguments are valid:
   (a) Check that m and n are *integers* satisfying $n > m$; if not, the function is to print the error message

   ```
   prod_even_odd: Input arguments m and n must be integers satisfying n > m.
   ```

   and return `NaN` (not a number).
   (b) Check that m and n are scalars (single values). If not, the function is to print the error message

   ```
   prod_even_odd: Input arguments m and n must be scalars.
   ```

   and return `NaN`.

   Implement these error checks separately and in the order listed above.
   Hints:
   (a) ~`phys210/matlab/errorreturn.m` defines a function `errorreturn` that illustrates the use of `fprintf` and `return` to issue an error message and immediately exit from a function.

(b) You may find the `Matlab` `lookfor` command useful for locating one or more functions that will be helpful with the argument checking, but note that `isinteger` does not work the way that one might naively expect:

```
>> isinteger(100)

   ans = 0
```

Test your functions thoroughly, being sure to use invalid as well as valid input arguments when testing `prod_even_odd`. We will evaluate your implementations using our own input.

File inventory for this question:

1. `fcn5.m`
2. `plot5.m`
3. `prod_even_odd.m`

Problem 2: Consider a collection of n particles, with position vectors $\mathbf{r}_k$ defined by

$$\mathbf{r}_k = [x_k, y_k, z_k]\,, \quad k = 1, 2, \ldots, n \tag{1}$$

The distance between any two particles, labelled by l and m, with position vectors $\mathbf{r}_l$ and $\mathbf{r}_m$, respectively, is then given by

$$d_{lm} = \sqrt{(x_l - x_m)^2 + (y_l - y_m)^2 + (z_l - z_m)^2} \tag{2}$$

(Note that both l and m can take on any value in the range $1, 2 \ldots, n$).

Create the subdirectory `hw3/a2`. Within that subdirectory create the file `dpart.m` that implements a `Matlab` function `dpart` as follows:

Header: `function d = dpart(r)`
Functionality: The input argument, `r`, can be assumed to be an $n \times 3$ array with elements given by

`r(k,1)` $= x_k$
`r(k,2)` $= y_k$
`r(k,3)` $= z_k$

The function is to return an $n \times n$ array with elements

```
  d(l, m),  l, m  = 1, 2, ..., n
```

whose values are defined by (2). Note that the number of particles is *implicitly* given by the size of the input array. Your function does not have to perform any error checking of the input argument.

Test your implementation using input of your own design. We will grade your work using our own test input.

Hints:

1. Recall that the `size` command returns the dimensions of an array.
2. There is a demonstration function, `mytranspose`, defined in ~`phys210/matlab/mytranspose.m`, that you may find useful in solving this problem.

File inventory for this question:

1. `dpart.m`

Problem 3: *Two Dimensional Random Walks*

3.1) Description of the model

This problem is concerned with *random walks* of a particle on a two-dimensional square lattice. The lattice is the collection of points on the xy plane having coordinates $(x, y) = (i, j)$, where i and j are both *integers*. The possible positions of the particle are restricted to the set of lattice locations.

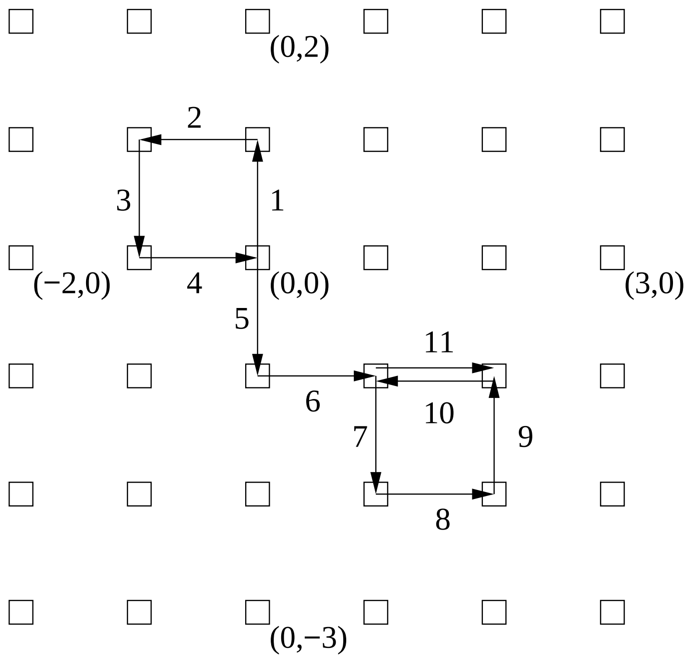
The walk unfolds in discrete time steps that will be labelled by another integer, k, where k ranges over the values $k = 0, 1, 2, \ldots n_{\mathrm{steps}}$, and where n_{steps} is the number of steps in the walk. We will also refer to n_{steps} as the length of the walk. We denote the particle coordinates at step k by (x_k, y_k): again, both x_k and y_k are integer-valued.

At the initial time, $k = 0$, the particle is situated at the origin; i.e. $(x_0, y_0) = (0, 0)$. At each step of the walk, the particle moves *randomly* to one of the four lattice sites closest to its current position. Specifically, for $k = 0, 1, 2, \ldots, n_{\mathrm{steps}} - 1$, the four possible movements of the particle are

$$\begin{aligned}
(x_k, y_k) &\rightarrow (x_{k+1}, y_{k+1}) = (x_k + 1, y_k) \\
(x_k, y_k) &\rightarrow (x_{k+1}, y_{k+1}) = (x_k - 1, y_k) \\
(x_k, y_k) &\rightarrow (x_{k+1}, y_{k+1}) = (x_k, y_k + 1) \\
(x_k, y_k) &\rightarrow (x_{k+1}, y_{k+1}) = (x_k, y_k - 1)
\end{aligned}$$

and there is equal probability that any of the four possibilities occurs.

Here is an illustration of a random walk with $n_{\mathrm{steps}} = 11$. The squares are the lattice sites, and the arrows, labelled with the step number $k = 1, 2, \ldots, 11$, show the trajectory of the particle. Notice that in the last step the particle "doubles back" on itself.



The sequence of $n_{\mathrm{steps}} + 1 = 12$ coordinate-pairs for this walk is

$$(0,0) \rightarrow (0,1) \rightarrow (-1,1) \rightarrow (-1,0) \rightarrow (0,0) \rightarrow (0,-1) \rightarrow (1,-1) \rightarrow (1,-2) \rightarrow (2,-2) \rightarrow (2,-1) \rightarrow (1,-1) \rightarrow (2,-1)$$

It is important to note that the total number of coordinate-pairs defining a walk is always $n_{\mathrm{steps}} + 1$, since the origin of the walk is included.

Given the above definition of a random walk, we can pose various questions concerning the nature of the discrete dynamics. For example, we can ask: *on average*, how far will a randomly-walking particle be from its starting location after a certain number of steps? It is crucial to pose this and other queries in terms of an average over many walks since, as you might expect, and as you will see explicitly as you work on this problem, when we we examine the details of the particle trajectories from walk to walk, we invariably see large variations (fluctuations).

We thus will consider ensembles (collections) of a number of walks, denoted n_{walks}, each of length n_{steps}. Although the computer experiments you are to implement will be done with specific values of n_{walks} and n_{steps}, it is important to note that these quantities constitute key *control parameters* for the model. That is, were we to investigate the model in more detail than will be done here, we would want to do so in the context of a set of calculations in which both n_{walks} and n_{steps} were varied.

As just mentioned, one of the things that we can attempt to determine is the particle's average displacement from its starting position, and, for this particular flavour of random walk, it is actually more revealing to measure the square of that displacement. We will therefore compute d_k^2, defined by

$$d_k^2 \equiv {x_k}^2 + {y_k}^2 \quad \text{for} \quad k = 0, 1, 2, \ldots n_{\text{steps}} \tag{1}$$

and, for a collection of walks, the ensemble average of that quantity, denoted $\langle d^2 \rangle_k$:

$$\langle d^2 \rangle_k \equiv \frac{1}{n_{\text{walks}}} \sum_{M=1}^{n_{\text{walks}}} d_k^2\,(M) \quad \text{for} \quad k = 0, 1, 2, \ldots n_{\text{steps}} \tag{2}$$

In this last definition, the notation $d_k^2\,(M)$, where M labels a specific walk within the ensemble, is meant to emphasize that the specific values of d_k^2 will vary from walk to walk, i.e. will be a function of M.

Finally, before proceeding to the description of the `Matlab` code that you are to write, and the calculations you are to perform, we note that all of x_k, y_k, d_k^2 and $\langle d^2 \rangle_k$ are naturally represented in `Matlab` as vectors of length $n_{\text{steps}} + 1$. In your programming you should take advantage of this observation, and, as much as possible, `Matlab`'s vector/array-manipulation capabilities (i.e. try to keep the number of `for` loops in your code to a minimum).

Bear in mind, however, that vectors/arrays in `Matlab` always have a first index (address) 1, *not* 0. This means that the step label k we have been using cannot be *directly* mapped onto an array index, but this should not cause you undue difficulty. If you are confused by this point, ask for help!!

3.2) `Matlab` implementation and computational experiments

First, create the subdirectory `hw3/a3`. The problem involves the coding of one `Matlab` function that simulates an individual walk, and three `Matlab` scripts that use that function to carry out specific experiments. The text files that define these functions and scripts must all be located within `hw3/a3`.

Please read carefully through the entire *description that follows before starting to design and write code: in particular, pay close attention to the hints that are given. This will minimize frustration and wasted time. Should anything not be clear to you, ask me for clarification.*

3.2.1) `randwalk.m`

Create the file `randwalk.m` that implements a `Matlab` function with the following header and functionality:

1. *Header*: `function [x y dsq norig] = randwalk(nsteps)`
2. *Functionality*: The single input argument, `nsteps`, is the length of the walk. The four output arguments are defined as follows:
   (a) `x`: Row vector of length `nsteps + 1` containing the x coordinates of the walk.
   (b) `y`: Row vector of length `nsteps + 1` containing the y coordinates of the walk.
   (c) `dsq`: Row vector of length `nsteps + 1` containing the squared displacements, d_k^2, of the walk.
   (d) `norig`: The number of times that the particle returns to the origin $(0, 0)$ during the walk. Do *not* count the initial time as one of these visits.

Hint: You may find the `Matlab` function `randi`, discussed briefly in the October 17 lab, useful in your implementation.

3.2.2) `srandwalk1.m`

Create a `Matlab` script file, `srandwalk1.m` that

1. Uses `randwalk` to generate three distinct random walks with $n_{\text{steps}} = 1000$.
2. Uses the `plot` command to make a *single* figure (plot/graph) showing the three walks and using red, green and blue lines, respectively (specifically, plot y_k versus x_k for the walks). Give the plot a title "`Three random walks`", and label the x and y axes "`x`" and "`y`", respectively.
3. Uses the `print` command to save the figure as a `JPEG` image in the file `walks.jpg`

3.2.3) `srandwalk2.m`

Create a `Matlab` script file, `srandwalk2.m` that

1. Uses `randwalk.m` to compute $\langle d^2 \rangle_k$ with $n_{\text{walks}} = 1000$ and $n_{\text{steps}} = 10000$.
2. Uses the `Matlab` function `polyfit` to perform a linear fit to $\langle d^2 \rangle_k$; i.e. uses `polyfit` to determine coefficients a and b such that

   $$ak + b$$

   minimizes the least-square residuals of the fit.

   The script must output the values of `a` and `b`, i.e. the fit coefficients *must* be assigned to `Matlab` variables `a` and `b`, and those values must then be displayed by using the statements

   ```
   a
   b
   ```

   This is the single exception to the "end every statement with a semicolon" dictum enunciated in the instructions.
3. Makes a single figure in which $\langle d^2 \rangle_k$ (red line) and $ak + b$ (green line) are plotted versus k. Provide a legend for the figure with entries "`<displacement squared>`" and "`linear fit`", and label the x-axis "`step`" (no title or y-axis label).
4. Saves the figure as a `JPEG` image in the file `dsqmean.jpg`

Record the values of a and b that are computed in a `README` file. On the basis of the value of a, as well as the figure created by the script, what can you conjecture about the behaviour of $\langle d^2 \rangle_k$ as a function of k? In other words, how does the expectation value of the square of the displacement appear to vary with time? Answer in `README`.

Hint: You can use `Matlab`'s help facility and/or online material to learn about `polyfit`. There is also a sample script ~`phys210/matlab/tpolyfit.m` that provides an illustrative example of its usage.

3.2.4) `srandwalk3.m`

Create a `Matlab` script file, `srandwalk3.m` that

1. Uses `randwalk.m` to generate $n_{\text{walks}} = 10000$ walks, each of length $n_{\text{steps}} = 1000$.
2. Determines the maximum number of times among those walks that the particle returns to the origin. I will call this value `maxnorig`.
3. Uses the `Matlab` `hist` command to make a histogram with `maxnorig` + 1 bins that shows the number of walks with `norig` origin returns as a function of `norig`. Give the histogram a title "`Histogram of number of origin returns for nwalks=10000, nsteps=1000`", and label the x and y axes "`Number of origin returns`" and "`Count`", respectively.
4. Saves the histogram as a `JPEG` image in the file `norig.jpg`

Hints:

1. Use `Matlab` help or the web to learn about `hist`
2. There is a sample script ~`phys210/matlab/thist.m` that demonstrates usage of `hist`.
3. **Important:** I have purposefully *not* told you precisely how you are to collect the data to be histogrammed. This you should try to figure out on your own.

3.2.5) *Final hints and remarks*

When implementing and testing any of the four functions/scripts, but especially for `srandwalk1.m` and `srandwalk2.m`, use values of n_{steps} and n_{walks} sufficiently small that you can quickly develop and debug your code, and experiment with it. Only when you are satisfied that your coding is correct should you run with the required values of n_{walks} and n_{steps}.

Additionally, for the values of n_{walks} and n_{steps} specified above you can expect `srandwalk2` and `srandwalk3` to each take on the order of a minute to complete. If either n_{walks} or n_{steps} is reduced by a factor of 10, say, the execution time should also go down by a factor of about 10. You can use this information to gauge whether your algorithm appears to be consuming excessive amounts of computer time, which, in turn, *may* signal one or more flaws in your coding.

3.2.6) *File inventory for this problem:*

1. `randwalk.m`
2. `srandwalk1.m`
3. `srandwalk2.m`
4. `srandwalk3.m`
5. `walks.jpg`
6. `dsqmean.jpg`
7. `norig.jpg`
8. `README` (contains answers for **3.2.3**)