PHYS 210: Introduction to Computational Physics Fall 2104 Homework 3

Version 2.1 Nov 4, 2014

Problems 1, 2 (and optional Problem 3) due: Monday, November 10, 11:59 PM
Problem 4 (and optional "freestyle" Problem 5) due: Wednesday, November 12, 11:59 PM

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

Please make careful note of the following information and instructions:

1. This homework is being distributed in two stages, and there are two distinct due dates:

   (a) **Problems 1 & 2 and the optional but highly recommended Problem 3, are due Monday, November 10 at 11:59 PM.**

   Please read the announcement in `http://laplace.phas.ubc.ca/210/Archive.html`, dated November 4 *carefully* for my rationale for making Problem 3 and, most importantly, how that alters the way that your overall grade will be determined: in brief, *you are now expected to invest more time and effort in Problem 4 than would otherwise be the case.*

   (b) **Problem 4 and the *strictly optional* Problem 5 are both due Wednesday, November 12 at 11:59 PM.** Problem 5 is an open invitation for you to extend Problem 4 in ways limited only by your imagination, ingenuity and free time, and is worth some small but non-negligible amount of extra credit. Given the length of the handout, that's all I'll say about it writing. Use your imagination and/or ask me if you need suggestions for what you might do.

2. You will note that you have somewhat less time to complete this homework than the previous ones. Additionally, I consider this assignment to be the most challenging of the three, and you should therefore not be surprised it it takes you significantly more time to complete than either of the previous ones.

   As usual, however, the length of the handout does not directly reflect the amount of work that you face; most of what follows is to be considered supplementary material that is important to you basic training in computational science. This is especially true for Problem 4 and Problem 3.

   However, despite all my dire warnings issued in this handout, in class, and in announcements made via connect and the course announcement page, some of you may find the homework pretty straightforward. Time will tell.

   The number of lines of Matlab code in my solutions (comments and whitespace excluded, and not including test scripts that you may write) will serve as an *approximate* measure of the relative time you can expect to spend on each of the four problems.

   The counts for the four questions are:

   (a) 16 (one script)

   (b) 35 total (four separate functions)

   (c) 44 total (three functions, two scripts)

   (d) About 350 lines (one functions, three scripts), *but*

      i. You are supplied with template functions and scripts for all parts of the questions, which total about 175 lines.

      ii. One part involves conversion of a function to a script. The script has about 100 lines as does the function (the function actually has fewer) and there are literally less that 10 statements between the two that are different.

      iii. The last script largely consists of four blocks of code that are identical to one another, with names of some variables and plot annotations changed.

   So there isn't anywhere near as much new code to write; i.e. code that can't essentially be cut and paste from some previously created source file.

   In any case, I *strongly* recommend that you start the first two problems as soon as possible so that you can get a sense for how much time you will need to allocate to complete the homework.

   Needless to say, unless you have a fairly complete mastery of the course material and well-honed programming skills, leaving the assignment until the last few days before it is due is *not* a good idea.

3. The fourth problem has been purposely structured to be somewhat of a mini-project. Working from the template script just mentioned, you will code a crude but complete simulation of a simple physical system and then carry out some basic "numerical experiments". Key aims are to (1) provide some additional context for how you should approach your term projects, (2) emphasize that correctly programming an algorithm is merely the first stage of a project in computational science and (3) expose you to the important and thorny issue of determining when computer-generated results can be trusted.

4. All four problems require that you implement one or more Matlab functions and/or scripts, all of which must be coded in `.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.

   Note that for administrative reasons I have already created the four subdirectories for the assignment:

   ```
   /phys210/$LOGNAME/hw3/a1
   /phys210/$LOGNAME/hw3/a2
   /phys210/$LOGNAME/hw3/a3
   /phys210/$LOGNAME/hw3/a4
   ```

   Should you code a Problem 5 solution, you will need to create

   ```
   /phys210/$LOGNAME/hw3/a5
   ```

   and add it to your Matlab path by modifying your `startup.m` file.

   Any reference to the directory `hw3` below is implicitly a reference to `/phys210/$LOGNAME/hw3`

5. **Example functions/scripts**

   The problem descriptions make reference to various `.m` files located in `~phys210/matlab` that provide usage examples for various Matlab functions which you will use in completing the problem set and which may prove generally useful in your coding tasks.

   I suggest that you point your browser to the via the Homework 3 code samples page (also available from the main course page):

   ```
   http://laplace.phas.ubc.ca/Students/phys210/hw3/
   ```

   and at least quickly browse through the samples *before* you start the corresponding problem: *doing so may end up saving you significant time and effort.*

   In addition to the specific demo scripts and and functions mentioned below, there are many other instructor-supplied pieces of code that may be of use here and for your term projects.

   Again, all of this code is contained in

   ```
   ~phys210/matlab
   ```

   on the lab machines, and it too can be easily browsed through the homework code samples page.

   Recall that the definition of any script or function that is in your Matlab path can be displayed in the command window using the `type` command. For example

   ```
   >> type tplot
   ```

6. **Crucial!! Statement terminators**

   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, except where explicitly stated otherwise, *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 image or movie 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, or that is saves an animation in AVI format, then it *must* do so.

This is a very important point, and failure to adhere to this aspect of the instructions may result in a significant penalty.

Note that checking whether your code conforms to this dictum provides a perfect opportunity to refresh/test your grepping skills.

Here's a pipeline that uses a sequence of `grep` commands to display all of the non-comment, non-white space lines in `foo.m` which do *not* end with a semicolon (or a semicolon with trailing whitespace), and thus which may require termination (of course it would be best for you to figure this out for yourself!):

```
% grep -v '^[ ]*%' foo.m | grep -v '^[ ]*$' | grep '[^;][ ]*$'
```

7. **Which Matlab features/functions/commands etc. can I use?**

   All of the scripts and functions that you must write for this assignment can be coded using Matlab features, functions and commands that we have discussed explicitly in class, or are otherwise contained in the Matlab lab notes (introductory *and* programming) or the instructor-supplied course functions and scripts.

   Please do your best to complete the homework without resorting to specialized functions or packages, or arcane techniques that you may find, e.g., by googling for solutions. If you are unsure as to whether use should use some particular feature, function, command etc., then ask me.

   Important! This particular directive does *not* apply to any of the bonus work. For that you are allowed and encouraged to use whatever features of Matlab you desire.

8. **Code commenting / documentation**

   None of the functions or scripts that you write to complete the assignment need to be *extensively* commented. For Problems 1, 2 and 3 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. For Problem 4 a little more commenting is warranted, but certainly not at the level of the base code that I supply. However, lest there be confusion, this does *not* mean that you have to *delete* comments from those pieces of program that you borrow wholesale from my code, unless you've been instructed to delete them.

   The degree to which your code is commented will not factor significantly into your grade.

9. **Error checking**

   With the exception of `windowy` *in Problem 2.4, none of the functions or scripts you code need to perform any error checking.*

   Additionally, in Problem 3, the function `newtonfda` must use the `warning` function in some circumstances to display a diagnostic message.

10. **Where/how to start Matlab**

    Provided that:

    (a) You have executed the script `addpath-hw3` on your PHAS account, which adds the following lines to your Matlab startup file, `~/Documents/MATLAB/startup.m`

    ```
    addpath('/phys210/phys210f/hw3/a1');
    addpath('/phys210/phys210f/hw3/a2');
    addpath('/phys210/phys210f/hw3/a3');
    addpath('/phys210/phys210f/hw3/a4');
    addpath('/phys210/phys210f/project');
    ```

    (b) You create the `.m` files for the assignment in the proper subdirectories of `hw3`

    then it should not matter how or where you start Matlab—the functions and scripts that you code should execute when you type the appropriate name: `plotdata` for the script `plotdata` defined in `hw3/a1/plotdata.m`, `inrectangle` for the function `inrectangle` defined in `hw3/a2/inrectangle.m`, etc. If you run into any problems in this regard, contact me immediately, and we'll get the issue sorted out.

Warning!!

Problems 1 and 4 involve scripts and/or functions which *generate* files. In those cases you will need to ensure that either:

(a) Matlab's working directory is the corresponding solution directory (`hw3/a1`, `hw3/a4`) when you execute the function/script.

(b) You move or copy the file from where it was generated to where it should be. Here you can use the Matlab `pwd` command to display Matlab's working directory. The file(s) should be there.

There are at least three ways that you can effect option 1 above.

(a) Change the working directory (folder) via the browser bar that is located near the top of the desktop interface (small folder with a / beside it)—this will probably be the most natural approach for the majority of you.

(b) Start Matlab from the bash command line in the appropriate solution directory.

(c) Use the Matlab (not bash) `cd` command at the Matlab command line to change Matlab's working directory.

11. **Using Matlab to work on the assignment remotely**

First, I assume that you have software installed on your computer that allows you establish a secure-shell terminal connection to `hyper` or `tau` and to run X11 applications. If you don't, see the Course Software page

```
http://laplace.physics.ubc.ca/210/Software.html
```

for installation instructions and contact me should you need help getting things working.

You should have no problem using command-line Matlab remotely, i.e. the version of Matlab that is started via

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

at the command line, or using the alias we defined

```
% mat
```

Typically, you should not notice much difference in the responsiveness between in-lab and remote work when the output from Matlab is strictly text. When plotting you will almost certainly notice a difference in the time that it takes for figure windows to appear, be refreshed etc., but I've done some experimentation recently with a medium-speed home network and even for the case of Problem 4 where the `plot` function is used to produce a basic real time animation, the performance was acceptable.

You can also experiment with using the desktop version remotely, but there really isn't any need to do so since it is only the plotting that *requires* X11, and that will be done identically no matter which version you use.

I note however, that I can't guarantee that you *will* be able to complete the assignment outside of the lab, and you will almost certainly find that the plotting, particularly for Problem 4, is more easily/quickly done in-lab.

Thus, as I have emphasized previously, you should be prepared to spend time in the lab outside of our regular sessions.

This comment also applies to your term project work.

12. Do not do any of your work, or save any files, in your home directory or anywhere else that is accessible by your fellow students.

13. **Overwhelmed? Confused? Frustrated?**

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!

This particularly applies to Problem 4.

Problem 1:

This is a Matlab version of Problem 5b from the first homework assignment.

The text file ~`phys210/hw3/data` contains 5 columns of numeric data. Copy it to the solution directory for this problem, `hw3/a1`. Now create a Matlab script file `plotdata.m` in the solution directory that contains all of the commands needed to perform the following plotting chore:

Make a *single* plot that graphs:

1. The third column versus the first using green open squares.
2. The fifth column versus the first using a blue line.

As in the first homework, I emphasize that "plot B versus A" means that the A-axis is horizontal and the B-axis is vertical.

Make the range of the horizontal axis match that of the data itself: that is, the leftmost values plotted should be located on the leftmost edge of the plot (i.e. on the y-axis), and the rightmost values should appear on the right edge. (*Hint:* Use the `xlim` function.)

Use the `legend` function to label the curves `raw data` and `fit`, respectively, and position the legend in the `north` location.

Label the horizontal and vertical axes `x` and `y`, respectively.

Give the plot a title

```
<Your name> HW3: Problem 1
```

replacing `<Your name>` with your name.

Use the `print` function to save the plot in JPEG format as the file `hw31.jpg`

To repeat, `plotdata.m` *must contain all of the commands that are needed to generate the plot and save it as a JPEG as described above.*

Once you are satisfied that your script is running correctly, incorporate the JPEG image that it generates into your course web page. Ensure that the image is located at the bottom of the web page and note that *you will need to copy the image file to* `/phys210/$LOGNAME/public_html` *so that my web server will be able to export it to the outside world.* If you had trouble with this process in the first homework be sure to ask for help as necessary.

Sample code that may be of use:

Recall that:

1. Within a Matlab command window you can display the definitions of any of the following scripts with the `type` command. E.g.

```
>> type tsaveload
```

2. All of the code is browseable via

```
http://laplace.phas.ubc.ca/Students/phys210/hw3/
```

1. `tsaveload.m`
2. `tplot.m`
3. `plotex.m`

Problem description continues on next page

File inventory:

1. `plotdata.m`
2. `hw31.jpg`

Problem 2

In directory `hw3/a2` create Matlab files that define functions as follows:

Problem 2.1

1. Filename: `inrectangle.m`
2. Header: `function truthval = inrectangle(pt, rect)`
3. Description:
   Consider a rectangle, R, lying in the x-y plane, whose lower-left and upper-right corners have coordinates

   $$(x_{\min}, y_{\min}) \qquad \text{and} \qquad (x_{\max}, y_{\max})$$

   respectively.
   Represent R as the length-4 row vector

   $$\texttt{rect} \equiv \texttt{[xmin xmax ymin ymax]}$$

   Let P be an arbitrary point in the plane with coordinates (P_x, P_y) and represent P as the length-2 row vector

   $$\texttt{pt} \equiv \texttt{[ptx, pty]}$$

   Then `inrectangle[pt, rect]` returns 1 (Matlab's "true") if P lies within R (including the boundary of R) and 0 (Matlab's "false") otherwise.
4. Error checking: None
5. Sample usage and output:

```
>> inrectangle([0 0], [-1 1 -1 1])
   ans =

        1

>> inrectangle([-1.1 0], [-1 1 -1 1])
   ans =

        0

>> inrectangle([0 1.1], [-1 1 -1 1])
   ans =

        0

>> inrectangle([0, 1 - eps], [-1 1 -1 1])
   ans =

        1

>> inrectangle([0, 1 + eps], [-1 1 -1 1])
   ans =

        0
```

Problem 2.2

1. Filename: `aijsum.m`
2. Header: `function a = aijsum(m, n)`
3. Description: `aijsum` returns the $m \times n$ matrix with elements a_{ij} defined by

$$a_{ij} = i + j, \quad i = 1, 2, \ldots m, \quad j = 1, 2, \ldots n$$

4. Error checking: None
5. Sample usage and output:

```
>> aijsum(1, 1)
ans =

     2

>> aijsum(1, 2)
ans =

     2     3

>> aijsum(2, 1)
ans =

     2
     3

>> aijsum(2, 3)
ans =

     2     3     4
     3     4     5
```

6. Note: *A small bonus will be awarded for implementations that do not use one or more loops, i.e. that use only matrix operations.*

Problem 2.3

1. Filename: `zebra.m`
2. Header: `function z = zebra(n)`
3. Description: `zebra` returns the $n \times n$ matrix whose main diagonal is all 1's and whose upper and lower diagonals, enumerated away from the main diagonal, are alternating all-0's and all-1's
4. Error checking: None
5. Sample usage and output:

```
>> zebra(1)
ans =

     1

>> zebra(2)
ans =

     1     0
     0     1
```

```
>> zebra(3)
ans =

     1     0     1
     0     1     0
     1     0     1

>> zebra(6)
ans =

     1     0     1     0     1     0
     0     1     0     1     0     1
     1     0     1     0     1     0
     0     1     0     1     0     1
     1     0     1     0     1     0
     0     1     0     1     0     1
```

6. Notes:
   (a) There are many ways to implement this function, but you may first want to study the output from `aijsum(n,n)` for various values of `n` to see whether you can detect a pattern or property that may be of use here.
   (b) *A small bonus will be awarded for implementations that do not use one or more loops*

Problem 2.4

1. Filename: `windowy.m`
2. Header: `function [xout, yout] = windowy(xin, yin, ymin, ymax)`
3. Description: Let x^{in} and y^{in} be length-n^{in} vector whose components are to be viewed as x–y pairs:

$$\left(x_i^{\text{in}},\ y_i^{\text{in}}\right), \quad i = 1, 2, \ldots\ n^{\text{in}} \tag{1}$$

   Then x^{out} and y^{out} are length n^{out} vectors with components also to be viewed as pairs:

$$\left(x_j^{\text{out}},\ y_j^{\text{out}}\right), \quad j = 1, 2, \ldots\ n^{\text{out}} \tag{2}$$

   The pairs defined by (2) are the subset of those defined by (1) such that

$$y_{\min} \le y_j^{\text{out}} \le y_{\max}$$

   for all j. The output components must appear in the same order as they do in the input vectors. Note that n^{out} may be 0.
4. Error checking: `windowy.m` must check that `xin` and `yin` are both *vectors* (use the `isvector` function) and have the same length. If the arguments do not satisfy these conditions it must invoke the `error` function *precisely* as follows:

```
error('xin and yin must be vectors and the same length.');
```

   Note that if and when `error` is called, Matlab will immediately stop execution, not only of the function, but also, for example, any script that you are running to test the function. Thus if you want to test the function with more than one set of invalid data non-interactively, you will either need to code different scripts or, more conveniently, test each invalid call by commenting-out the others as discussed in the lab notes.

Problem description continues on next page

6. Sample usage and output:

(a)

```
>> x = 1:9
>> y = x.^2
>> [xw, yw] = windowy(x, y, 3, 30)

x =

     1     2     3     4     5     6     7     8     9


y =

     1     4     9    16    25    36    49    64    81


xw =

     2     3     4     5


yw =

     4     9    16    25
```

(b)

```
>> format long;
>> x = linspace(-3, 3, 1001);
>> y = cos(x.^2);

>> [xw, yw] = windowy(x, y, 0.99999999, 1.2)
xw =

  -0.006000000000000                   0   0.006000000000000


yw =

   0.999999999352000   1.000000000000000   0.999999999352000
```

(c)

If $n^{\text{out}} = 0$, then both output arguments from `windowy` must be the empty matrix `[]`. For example.

```
>> [xw, yw] = windowy(x, y, 1.1, 1.2)
xw =

     []


yw =

     []
```

(d)

Here are some examples where the input arguments are not valid.

```
>> First input is a matrix, not a vector.
>> [xw, yw] = windowy([[1 0]; [0 1]], y(1:end-1), 1.1, 1.2)
Error using windowy (line <n>)
xin and yin must be vectors and the same length.

>> Inputs don't have the same length.
>> [xw, yw] = windowy(x, y(1:end-1), 1.1, 1.2)
Error using windowy (line <n>)
xin and yin must be vectors and the same length.
```

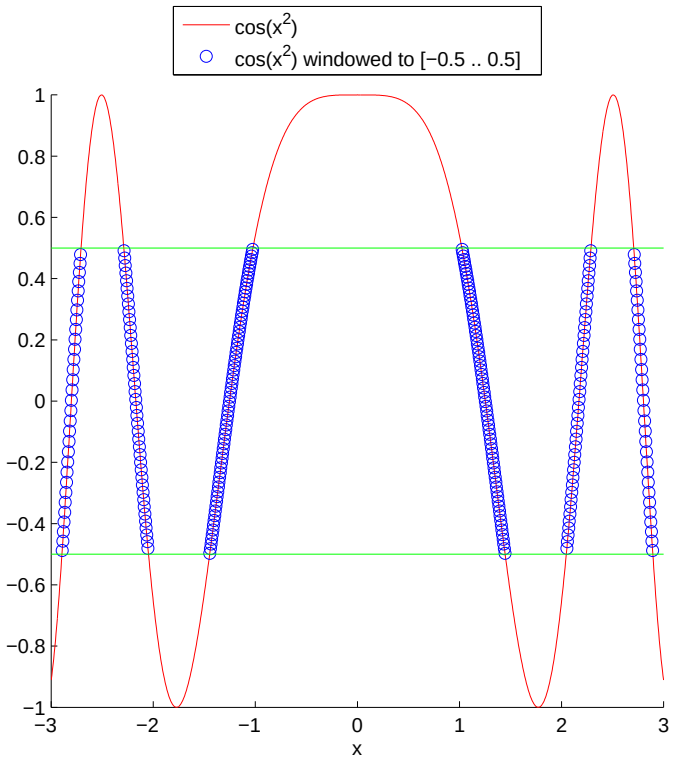
Note that the `<n>` in the error messages above will be the specific line number of the `error` statement in your implementation.

(e)

Finally, here's an invocation that includes a plot of the original and "windowed" data.

```
>> x = linspace(-3, 3, 1001);
>> y = cos(x.^2);
>> [xw, yw] = windowy(x, y, -0.5, 0.5);
% Make plot.
```

.
.
.



Problem description continues on next page

Notes:

(a) It is OK to implement this function using explicit definition of the output array elements. I.e. you are not expected to initialize the output arrays to their final sizes, using `zeros(...)` for example, before you define the actual values.

(b) *A small bonus will be awarded for implementations that do not use one or more loops.*

Sample code that may be of use:

1. `mytranspose.m`
2. `tmytranspose.m`

File inventory:

1. `inrectangle.m`
2. `aijsum.m`
3. `zebra.m`
4. `windowy.m`

Problem 3: *Newton's method using a finite difference approximation*

The task of solving a nonlinear equation in a single unknown, also known as *root finding*, arises frequently in numerical analysis and computational science. One technique that is very often used for this purpose is Newton's method (also known as the Newton-Raphson method). In this problem you will write a Matlab function that uses Newton's method to find a solution of some given nonlinear equation, which equation will itself be coded as a Matlab function.

3.1 Background

We first require that the nonlinear equation to be solved is written in a so-called *canonical* form (i.e. a standard form):

$$f(x) = 0, \tag{1}$$

where x is the unknown, i.e. so that nothing appears on the right hand side of the equation. In general, nonlinear equations may have more than one solution—at a minimum you've been familiar with this fact since you learned about the solution of quadratic equations, but our concern here will be to find *one* solution at most at a time. We will denote a solution of (1) by $x^\star$ so that

$$f(x^\star) \equiv f(x)|_{x=x^\star} = 0\,.$$

Newton's method requires that we start with some estimate, or guess, for $x^\star$. We will denote this value $x^{(0)}$. Newton's method is *iterative*: starting from the initial estimate, a sequence of values is generated

$$x^{(0)} \to x^{(1)} \to x^{(2)} \to \cdots \to x^{(n)} \to x^{(n+1)} \to \cdots \tag{2}$$

and, if the method *converges*, we have

$$\lim_{n\to\infty} x^{(n)} = x^\star\,. \tag{3}$$

Whether or not the method *does* converge depends, in general, both on the equation being solved and the quality of the initial guess, but we will not delve into those details here.

Newton's method also requires that the function $f(x)$ be differentiable. Denoting the derivative of $f(x)$ by $f'(x)$, the algorithm that generates the $n+1$-th iterate from the n-th is simply

$$x^{(n+1)} = x^{(n)} - \frac{f\left(x^{(n)}\right)}{f'\left(x^{(n)}\right)} \tag{4}$$

The problem of finding a square root of a real number a provides a convenient simple example of the technique. The equation we wish to solve is

$$f(x) = x^2 - a = 0$$

Eq. 3 then gives us

$$\begin{aligned} x^{(n+1)} &= x^{(n)} - \frac{f\left(x^{(n)}\right)}{f'\left(x^{(n)}\right)} \\ &= x^{(n)} - \frac{\left[x^{(n)}\right]^2 - a}{2x^{(n)}} \\ &= \frac{1}{2}\left(x^{(n)} + \frac{a}{x^{(n)}}\right) \end{aligned}$$

Let's work out the first few iterations by hand for $a = 2$—so we are computing $\sqrt{(2)}$ (we'll assume that we'll get the positive root)—starting with the initial estimate $x^{(0)} = 1$:

$$x^{(1)} = x^{(0)} - \frac{f\left(x^{(0)}\right)}{f'\left(x^{(0)}\right)} = 1 - \frac{(1^2-2)}{(2)(1)} = 1.5$$

$$x^{(2)} = x^{(1)} - \frac{f\left(x^{(1)}\right)}{f'\left(x^{(1)}\right)} = 1.5 - \frac{(1.5^2-2)}{(2)(1.5)} = 1.416666666666667 = 1.41\bar{6}$$

$$x^{(3)} = x^{(2)} - \frac{f\left(x^{(2)}\right)}{f'\left(x^{(2)}\right)} = 1.41\bar{6} - \frac{((1.41\bar{6})^2-2)}{(2)(1.41\bar{6})} = 1.4142\underline{1}5686274510$$

Here's a table that summarizes the results for a total of five iterations:

n	$x^{(n)}$
0	$\underline{1}.00000000000000$
1	$\underline{1}.500000000000000$
2	$1.4\underline{1}6666666666667$
3	$1.4142\underline{1}5686274510$
4	$1.41421356237\underline{4}690$
5	1.414213562373095

In each entry of the table the last correct digit of $x^{(n)}$ is underlined. The table illustrates one of the most appealing features of Newton's method: *when the method converges, it does so rapidly.* In general we can expect the number of correct digits to roughly *double* with each iteration, a property which is known in root finding as *quadratic* convergence.

Because Newton's method is iterative, we need some mechanism to decide when we should stop the iteration. In practice, we will usually want to do this when we think that the approximation to a root is "good enough" and, again, what is "good enough" will be problem-dependent in general. One common tactic is to stop when the absolute value of the change in the estimate is less than or equal to some specified positive tolerance ϵ. Note that we can write (4) as the two-step process:

$$\delta x = \frac{f\left(x^{(n)}\right)}{f'\left(x^{(n)}\right)}\,, \tag{4a}$$

$$x^{(n+1)} = x^{(n)} - \delta x\,. \tag{4b}$$

The stopping criterion is then

$$|\delta x| \leq \epsilon\,. \tag{5}$$

Note that when we code this test in conjunction with (4a) and (4b), we should have the iteration stop *after* we have made the update (4b) for the last step, i.e. when $|\delta x|$ satisfies the convergence criterion. This will yield a more accurate estimate of the root than if we stopped without making the last adjustment.

Also, to guard against the case when the method does *not* converge, we should always limit the algorithm to some maximum number of steps, $n_{\max}$. Once more, a suitable choice for this parameter is problem- and initial-estimate-dependent, but we will take it to be a value of 20 or so.

3.2 The problem per se

3.2.1 *Implementation of the Newton method*

In the directory `hw3/a3` create a Matlab file `newtonfda.m` which a function `newtonfda` that employs Newton's method to solve a non-linear equation.

However, rather than using the exact expression for $f'(x)$—as was done in the square-root example above—the function is to use the second-order centred finite difference approximation:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{6}$$

in formula (4). This approach obviates the need to code a Matlab function that computes the derivative of f in addition to one that evaluates f itself.

The header for the function is

```
function x = newtonfda(f, x0, h, epsilon, nmax)
```

The 5 input arguments are defined as follows:

1. `f` is a so-called *function handle* which:
   (a) Will effectively evaluate to the name of whatever actual function name is supplied by the calling script or function: see the demonstration function `fcncaller` and associated driver script `tfcncaller` for an illustration of the mechanism.
   (b) Defines the non-linear equation to be solved.
2. `x0` is the initial estimate, $x^{(0)}$, of the root.
3. `h` is the positive value to be used in computing the approximation of $f'(x)$ using the FDA (6).
4. `epsilon` is the convergence criterion, $\epsilon > 0$, in (5).
5. `nmax` is the maximum number of iterations, $n_{\max}$ that are to be performed.

The single output argument, `x` is a *vector* which is defined as follows:

1. If the method converges in n iterations, where $n \leq n_{\max}$, `x` is a vector of length $n+1$ whose elements are the values
   $$x^{(0)},\ x^{(1)},\ x^{(2)},\ ...,\ x^{(n)}$$
   In this case `x(end)` will contain the final (and best) approximation of the root.
2. If the method does *not* converge in $n_{\max}$ or fewer iterations, `x` is a vector of length $n_{\max}+2$ whose first $n_{\max}+1$ elements are the values
   $$x^{(0)},\ x^{(1)},\ x^{(2)},\ ...,\ x^{(n_{\max})}$$
   and whose last value is `NaN` (not a number). Recall that we can assign the value `NaN` as if it were a numerical value, e.g.

   ```
   >> a = NaN

   a =

      NaN
   ```

   In this case `x(end)` will contain `NaN`.

Error checking:

No error checking is required.

However, in the case that the iteration does *not* converge, the function must use the Matlab `warning` function as follows:

```
warning('No convergence within specified maximum number of iterations.');
```

Note that, unlike `error`, execution of `warning` does *not* cause the executing program to halt, or the function that uses it to return.

3.2.2 *Application of the Newton solver*

First, here's some sample output of my implementation of `newtonfda` as applied to the square-root example from Sec. 3.1 above.

I've created a file `froot2.m` with contents:

```
function f = froot2(x)
% froot2 Function for solving the equation x^2 = 2.
   f = x.^2 - 2;
end
```

which thus implements the nonlinear equation that is to be solved,

$$x^2 = 2,$$

but recast in the canonical form (1)

$$x^2 - 2 = 0\,.$$

Note that in the function definition I have used component-wise vector syntax. Although this is not necessary from the point of view of how `newtonfda` calls the function, it is convenient to do so in general. In particular, the algorithm will find only one root at a time, and as has been mentioned, may fail to converge should the initial guess $x^{(0)}$ not be good enough. In order to determine:

1. how many solutions roots the equation has,
2. good initial guesses for any and all of those solutions,

it can be very useful to first make a plot of the function. Coding the function using vector operations then makes this easy to do in Matlab.

With the function `froot2` in hand, I use `newtonfda` to compute an approximate root, first of $+\sqrt{2}$:

```
>> format long

>> x = newtonfda(@froot2, 1.0, 1.0e-6, 1.0e-12, 20)

x =

  Columns 1 through 4

   1.000000000000000   1.499999999985622   1.416666666659811   1.414215686274276

  Columns 5 through 7

   1.414213562374690   1.414213562373095   1.414213562373095

>> x(end)

ans =

   1.414213562373095
```

Now compute $-\sqrt{2}$:

```
>> x = newtonfda(@froot2, -3.0, 1.0e-6, 1.0e-12, 20)

x =

  Columns 1 through 4

  -3.000000000000000  -1.833333333496408  -1.462121212105660  -1.414998429890655

  Columns 5 through 7

  -1.414213780047123  -1.414213562373112  -1.414213562373095

>> x(end)

ans =

  -1.414213562373095
```

Note carefully the syntax `@froot2` that is used to pass the "handle" (called a "pointer" in other languages) for the function `froot2` to `newtonfda`.

I.e. when you want to pass a function to another function, *simply prefix the name of the function that is to be passed with the "at sign"*, `@`.

Here's an artificially concocted invocation where I limit the calculation to 3 iterations, resulting in "non-convergence" and the triggering of the warning message:

```
>> x = newtonfda(@froot2, 1.0, 1.0e-6, 1.0e-12, 3)

Warning: No convergence within specified maximum number of iterations.
> In newtonfda at 15
  In tnewtonfda at 9

x =

  Columns 1 through 4

   1.000000000000000   1.499999999985622   1.416666666659811   1.414215686274276

  Column 5

                 NaN
>> x(end)

ans =

   NaN
```

OK, now it's your turn. There are *two* separate applications that you need to implement. Each requires the coding of *two* separate `.m` files, for a total of *four* files.

3.2.2.1 *Application 1*

Consider the non-linear equation

$$\cos^3(x) = x^3 \tag{7}$$

where I emphasize that $\cos^3(x) \equiv (\cos(x))^3$.

In the solution directory, create a file `f1.m` that contains the definition of a function with the header

```
function f = f1(x)
```

and that can be used to determine a root of (7). Be sure to code the function using elementwise array operations so that if it is supplied with a vector argument it will return a vector of values.

Now, create a file `tf1.m` that contains a script file that does the following:

1. Makes a single plot that graphs the function and its absolute value using 1001 uniformly spaced values of x, ranging from $x = -\pi$ to $x = \pi$ inclusive. You do not have to label the plot in any way, nor do you have to save the plot as an image-format file. However, when the TAs run your script in a Matlab session, the plot must appear in a normal Matlab plotting window.
2. Calls your root finder using the statement

   ```
   x = newtonfda(@f1, x0, 1.0e-6, 1.0e-12, 20)
   ```

   where, prior to the invocation, `x0` is assigned a value that you have gleaned from inspection of the plot. Note that—as you will be able to verify for yourself—there is no need to try to determine the initial estimate `x0` to a *high* degree of accuracy.
3. Displays the computed root via the statement

   ```
   x(end)
   ```

Important: The last two statements of your script should read

```
>> x = newtonfda(@f1, ...)
>> x(end)
```

where the `...` is something that you will have to fill in. These should be the *only* statements in the script that are *not* terminated with semi-colons. Thus, the *only* output from the script (other that the plot) should be from those two lines of code.

Recall that in the instructions at the beginning of this handout I showed you a `grep`-based pipeline that can be of use in determining whether your code is missing some semi-colons.

3.2.2.2 *Application 2*

The following nonlinear equation arises in the study of the bound states of a quantum particle in a finite potential well:

$$\cos\left(\sqrt{E}\right) - \frac{(E-10)\sin\left(\sqrt{E}\right)}{\sqrt{E\,(20-E)}} \tag{8}$$

Here, E is the energy of the particle. The equation has been "non-dimensionalized", meaning that dimensionful parameters including the mass of the particle and Planck's constant $\hbar$ have been set to values such as 1 or 1/2—which is possible through an appropriate choice of units. This results in an algebraically simpler equations. Additionally, the constants 10 and 20 that appear reflect a particular choice I have made for the depth of the potential well, expressed in the new system of units. Note that when we solve an equation in non-dimensionalized form we will frequently want to "back convert" the solution value to the appropriate units (standard *MKS* for example), but this is a straightforward task.

Your job here parallels that of the previous application.

First create a file `f2.m` that defines a function `f2` with the header

```
function f = f2(E)
```

that can be supplied to `newtonfda` to compute a root of (8). Again, implement the function using elementwise array operations.

Then create a script file `tf2.m` that

1. Graphs the function and its absolute value using 1001 uniformly spaced values in x ranging from 0.1 to 18.0 inclusive.
2. Calls the root finder as many times as necessary, with distinct initial values `x0` deduced from the plot, to compute approximations of all of the roots in the interval $0.1 \leq x \leq 18.0$. It is up to you to determine how many roots there are.

Each invocation to `newtonfda` should use the same values of `h`, `epsilon`, and `nmax` that were used in the previous application, and should be followed by a statement which displays the estimate of the root. That is, the final lines in your script should be a certain number of pairs of statements of the form

```
>> x = newtonfda(@f2, ...)
>> x(end)
```

where, again, these statements should *not* be terminated with semi-colons and should be the *only* statements in the script that do not end with semi-colons.

Sample code that may be of use:

1. `fcncaller.m`
2. `tfcncaller.m`
3. `myfcn.m`
4. `taylorsin.m`
5. `ttaylorsin.m`

File inventory:

1. `newtonfda.m`
2. `f1.m`
3. `tf1.m`
4. `f2.m`
5. `tf2.m`

Problem 4: *The bouncing ball*

4.1 Introduction

This problem deals with a simulation of a very simple dynamical system: a bouncing ball. The aim here is *not* to model the dynamics accurately, but to give you experience with:

1. Creating a complete program that performs a simulation in which various parameters that control the dynamics can be adjusted.
2. Using visualization techniques to expedite code development and to identify and display qualitative behaviour.
3. Running "numerical experiments" and performing a basic analysis of the results.

This problem is qualitatively different from any of the previous homework questions, and as mentioned in the instructions, should be viewed as a warm-up for your projects.

Important!

I realize that you all have many other demands on your time, and that you also have your term projects to get working on. I thus do *not* expect you to spend an inordinate amount of time on this question and the grading of your work will *not* be stringent. In other words, do the best that you can and please don't worry too much about the impact on your final grade should you not do especially well here.

> Just don't ""blow off" the problem, please—that *will* cost you.

If, as the saying goes, a picture is worth a thousand words, then in the case of scientific computing, particularly for simulations of a dynamical system, an animation is worth a few orders of magnitude more. You should thus begin this work by viewing (or re-viewing) the animation at

`http://laplace.phas.ubc.ca/210/Animations/bounce_credits.avi`

Most of you have already seen the animation at least once already and know that it gives us an immediate sense for what is being simulated.

The rest of this problem description is organized as follows: Secs. 4.2–4.5 describe the base scenario that you are to extend. Sec. 4.6 then details the actual work that you need to do. I have purposely structured Sec. 4.2–4.5 to reflect the stages involved in the completion of a typical term project, despite the fact that the approach may seem excessively heavy-handed for this apparently simple problem. As part of this approach I am going to completely ignore the fact that the problem can be solved using traditional pen-and-paper techniques (i.e. "analytically'). I also note that the discussion parallels those in the class notes for the non-linear pendulum and N-body dynamics, so you may also find it fruitful to review that material as you read through this.

Important: Even if you are fully capable of completing the problem without reference to the code that I am supplying, I ask that you follow the outlined procedure, which begins with the modification of a base script, not least since it will aid the TAs with the very non-trivial task of grading your work.

4.2 Physical setup and mathematical formulation

In colloquial terms we want to model the motion of a ball that falls towards a surface—the floor—from which it can bounce.

More precisely, we will approximately solve the equations of motion for a point particle[1] moving under the influence of:

1. A spatially uniform (constant) gravitational force, and only that gravitational force, when it is not interacting with the floor.[2]

[1] I will thus frequently use the term "particle" rather than "ball" in the following, but the two terms are to be understood to be synonymous.

[2] These are the conditions for the so-called *free fall* of an object and I will use that terminology below.

2. An additional, impulsive (instantaneous) force, with an adjustable coefficient of restitution, when it interacts with the floor.

November 9: Begin clarification:

We work in an x-y spatial coordinate system with an origin $(0, 0)$.

The *spatial solution domain* for the problem is the unit square, defined by:

$$0 \leq x \leq 1 \qquad 0 \leq y \leq 1\,. \tag{1}$$

The *only* dynamics we are concerned with is that when the ball is within the unit square, including the boundaries. The instant that the ball leaves the computational domain (assuming that it does exit), then for the purpose of the analysis we will make of the results, the *simulation is over*.

For the purposes of plotting and making animations, it is not the best idea to make the limits of the plotting region coincide with those of the solution domain. If we did then the floor (and the walls that you will add in implementing `mbounce` below) would coincide with the axes, and we would lose some visual clarity in the animations. Thus, in my simulation of the basic bouncing ball and in your extension of it, the *plotting domain* includes a border (margins if you will) around the unit square.

November 11: Important bug fix

The border is defined by the variable `dlim`, which in the original version of `bounce.m` that I provided, was incorrectly set in the "Graphics section" of the main `for` loop. It should be, and is now, properly set in the code *before* the loop. Excluding comments and whitespace/empty lines, the relevant sequence of statements in the code now reads:

```
fwidth = 0;
dlim = 0.15;
avienable = 0;
```

Please make note of this, and I profusely apologize for the bug.

Again, however, it is important to understand that we are *not* interested in what happens to the ball if and when it gets outside the unit square. For your implementation of `mbounce` I will ask you, somewhat arbitrarily, to stop simulating when the ball exits *the plotting region*—I could have equally well required you to stop the calculation when it leaves the *computational domain, that is the unit square*; it would make *absolutely no difference* in the subsequent analysis.

One more point: none of the boundaries of the *plotting domain*—the actual axes seen in my animation, and which are defined by the Matlab code

```
axis square;
box on;
xlim([-dlim, 1 + dlim]);
ylim([-dlim, 1 + dlim]);
```

are to be treated as reflective surfaces. So if you see your ball rolling on or sliding down one of them, for example, it's a clear sign that your implementation is buggy.

November 9: End clarification:

The floor is abstracted as a line segment with left and right endpoints:

$$\left(x^{\mathrm{f}}_{\mathrm{min}}, y^{\mathrm{f}}\right) \qquad \text{and} \qquad \left(x^{\mathrm{f}}_{\mathrm{max}}, y^{\mathrm{f}}\right)\,. \tag{2}$$

The specific choice I will make is—not surprisingly if you've been following my discussion on nondimensionalization and related issues—a unit-length length floor with a left endpoint at the origin, i.e. so that

$$x^{\mathrm{f}}_{\mathrm{min}} = 0 \tag{3}$$
$$x^{\mathrm{f}}_{\mathrm{max}} = 1 \tag{4}$$
$$y^{\mathrm{f}} = 0 \tag{5}$$

As can been from the animation, and assuming that the ball's x-velocity component is positive, if and when the ball's trajectory carries it beyond $x = x^{\rm f}_{\rm max}$, and it falls below the level of the floor, $y = y^{\rm f}$, then it simply disappears. We won't be concerned with what happens to the unfortunate ball after that.

In our model, the *independent variable* is the time, t. The fundamental *dependent variable*, and the quantity that we want to determine in the simulation, is the particle's time-dependent, two-dimensional position vector, $\mathbf{r}(t)$. However, $\mathbf{r}(t)$ can be written in component form as

$$\mathbf{r}(t) \equiv (x(t), y(t)) \,, \tag{6}$$

where $x(t)$ and $y(t)$ are the particle's time-dependent coordinates which are *scalar* functions of t. We can thus equivalently view $x(t)$ and $y(t)$ as the *pair* of dependent variables for the system. It is customary in dynamics to refer to the dependent variables as *dynamical variables*, and when we have the problem written in a form where the dynamical variables are scalar functions, then the number of those scalar functions is called the number of *degrees of freedom* in the system. Thus, our problem has *two degrees of freedom*.

The interval of time over which we will simulate the system is given by

$$0 \le t \le t_{\rm max} \,, \tag{7}$$

where $t_{\rm max}$ is a parameter of the problem.

> The goal of the simulation is to determine approximate values for $\mathbf{r}(t)$, or equivalently $x(t)$ and $y(t)$, for all t in that interval, for *specified* initial conditions (i.e. the value of $\mathbf{r}(0)$ or equivalently $x(0)$ and $y(0)$).

The equations of motion for the system follow from Newton's second law, the universal law of gravitation applied to the case of a uniform gravitational field, and the idealized description of the particle's interaction with the floor.

We define the particle's velocity and acceleration vectors, $\mathbf{v}(t)$ and $\mathbf{a}(t)$, respectively, by

$$\mathbf{v} \equiv \frac{d\mathbf{r}(t)}{dt} \,, \tag{8}$$

$$\mathbf{a} \equiv \frac{d\mathbf{v}(t)}{dt} \,, \tag{9}$$

as usual. As we did with the position vector $\mathbf{r}$, we write $\mathbf{v}$ and $\mathbf{a}$ in terms of their x and y components, which are scalar functions of t:

$$\mathbf{v}(t) \equiv (v_x(t), v_y(t)) \,, \tag{10}$$

$$\mathbf{a}(t) \equiv (a_x(t), a_y(t)) \,. \tag{11}$$

Let us first consider those periods of time when the particle is freely falling. We have

$$m\mathbf{a} = \sum_i \mathbf{F}_i \,, \tag{12}$$

where the $\mathbf{F}_i$ are the external forces acting on the particle. In our case, there is only one force in the sum, and we have

$$m\mathbf{a} = \mathbf{F}_G \,. \tag{13}$$

Here, $\mathbf{F}_G$ is the gravitational force which has components given by

$$\mathbf{F}_G = m\mathbf{g} \,, \tag{14}$$

where $\mathbf{g}$ is the constant acceleration vector of any object freely falling in a uniform gravitational field. Substituting (14) in (13) we have

$$m\mathbf{a} = m\mathbf{g} \,. \tag{15}$$

Using the fact that particle mass m appears on both sides of the equation, we find

$$\mathbf{a} = \mathbf{g} \,, \tag{16}$$

or

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{g} \,, \tag{17}$$

where the acceleration vector $\mathbf{g}$ can be written in component form as

$$\mathbf{g} = (0, -g) \ . \tag{18}$$

Here, g is the scalar *magnitude* of the acceleration due to gravity. Note that (17) is a *second-order* ordinary differential equation (ODE) for $\mathbf{r}(t)$, where in this context second-order means that the highest order *derivative* (and in fact the only derivative) that appears in the equation is a *second* derivative. Second-order in this sense is not to be confused with second-order as it applies to finite difference approximations, where order refers to the rate at which the error in the FDA vanishes as the mesh spacing tends to 0.

We will treat the particle–floor interaction as follows. We will assume that:

1. There is no force acting in the x direction, so that v_x is unaffected by the interaction.
2. The force in the y direction has the following net effect on the y-component, $p_y = mv_y$, of the particle's momentum:

   $$p_y \to -\kappa p_y \, . \tag{19}$$

   Here κ (the Greek letter "kappa") is a parameter known as the *coefficient of restitution* and quantifies how much y-momentum the particle loses (or gains) due to the interaction. Of course, if p_y changes due to the bounce while p_x stays the same, the energy of the particle will change as well.

Since the mass of the particle is constant we can summarize the net effect of an interaction which happens at some instant of time, t_I by

$$v_x(t_I) \ \to \ v_x(t_I) \, , \tag{20}$$
$$v_y(t_I) \ \to \ -\kappa v_y(t_I) \, . \tag{21}$$

Let us now assemble results to produce the form of the equations that we will solve numerically.

First, and in contrast to the procedure that we have followed for the pendulum and N-body cases, we will solve the free-fall equation of motion (17) in so-called *first-order form* wherein we evolve the position $\mathbf{r}$(t) and velocity $\mathbf{v}$(t) as separate quantities. Here, the terminology *first-order* refers to the order of the derivatives appearing in the ODEs, and is not to be confused with first-order as we use it when discussing the accuracy of FDAs.

We convert the *single* ODE (17) which is second order in time for the single dependent variable, $\mathbf{r}(t)$, to a system of two ODE's, each first order in time, for the *pair* of dependent variables, $\mathbf{r}(t)$ and $\mathbf{v}(t)$, as follows:

The ODE for $\mathbf{r}(t)$ follows from the definition of $\mathbf{v}(t)$:

$$\frac{d\mathbf{r}(t)}{dt} = \mathbf{v}(t) \, , \tag{22}$$

while that for $\mathbf{v}(t)$ comes from the original second-order form of the equation of motion:

$$\frac{d\mathbf{v}(t)}{dt} = \mathbf{g} \, , \tag{23}$$

where we recall that the constant vector $\mathbf{g}$ has components given by

$$\mathbf{g} = (0, -g) \ . \tag{24}$$

In contrast to our treatment of pendulum and N-body problems, I am *not* going to set $g = 1$ in what follows since for the purposes of making reasonable-looking simulations, i.e. where the ball's falling motion appears neither absurdly slow nor absurdly fast, it is convenient to be able to tune g.

Eqns. (22)–(23) are the final form of our equations for the particle's free fall motion, and are what we will subsequently discretize and solve numerically.

In addition, we have the equations defining the particle-floor interaction.

$$v_x(t_I) \ \to \ v_x(t_I) \, , \tag{25}$$
$$v_y(t_I) \ \to \ -\kappa v_y(t_I) \, . \tag{26}$$

at any instant $t = t_I$ when the particle meets the floor.

To complete the mathematical specification of the equations of motion, we observe that we need to supply initial conditions for the particle, that is its initial position $\mathbf{r}(0)$ and velocity $\mathbf{v}(0)$. Since the position and velocity each have two components, this amounts to a total of 4 values defined by

$$\mathbf{r}(0) \equiv (x_0, y_0)\,, \tag{27}$$
$$\mathbf{v}(0) \equiv (v_{x0}, v_{y0})\,. \tag{28}$$

The four initial values x_0, y_0, v_{x0} and v_{y0}, which we *must* specify as part of the setup of any specific simulation, are also parameters of the model.

Finally, I'm going to introduce one more quantity which we will view as an additional *output* of the simulation, namely the number of times, n_{bounce}, the ball bounces during the simulation interval, $0 \le t \le t_{\max}$. We will call this a *derived* quantity, since once we know the solution $\mathbf{r}(t)$, $\mathbf{v}(t)$ itself, we can determine n_{bounce}: there is no additional equation that independently "governs" how n_{bounce} evolves.

Let us summarize by enumerating the basic quantities that enter into the simulation, which fall into three categories:

1. *Independent variable:*
   (a) t
2. *Dependent variables:*
   (a) $\mathbf{r}(t)$
   (b) $\mathbf{v}(t)$
3. *Derived quantity:*
   (a) n_{bounce}
4. *Parameters*
   (a) $t_{\max}$
   (b) $x^{\text{f}}_{\min}, x^{\text{f}}_{\max}, y^{\text{f}}$
   (c) g
   (d) κ
   (e) x_0, y_0, v_{x0}, v_{y0}

Note that all of these quantities pertain to the *continuum* model and have thus nothing to do *per se* with any numerical approach we adopt to approximately solve the equations of motion. You can also see that even for a very simple simulation the number of things that we need to keep track of, and make sure are defined—the parameters in particular—can quickly add up!

4.3 Computational approach to the simulation

For simplicity, we adopt a very crude finite difference method to discretize the equations. It is only *first order* in the time step and is commonly called the *forward Euler method* or simply the *Euler method*. It also may already be familiar to you from other contexts.

We replace the continuum solution domain

$$0 \le t \le t_{\max}\,, \tag{29}$$

with a set of n_t discrete times

$$t^n \equiv \Delta t, \quad n = 1, 2, \ldots, n_t\,, \tag{30}$$

where $n_t \Delta t = t_{\max}$. At those discrete times we will compute approximate values, $\mathbf{r}^n$ and $\mathbf{v}^n$, of the continuum variables $\mathbf{r}$ and $\mathbf{v}$:

$$\mathbf{r}^n \equiv \mathbf{r}(t^n), \quad n = 1, 2, \ldots, n_t \tag{31}$$
$$\mathbf{v}^n \equiv \mathbf{v}(t^n), \quad n = 1, 2, \ldots, n_t \tag{32}$$

subject to the initial conditions

$$\mathbf{r}^1 \equiv \mathbf{r}(0) = (x_0, y_0) \tag{33}$$

$$\mathbf{v}^1 \equiv \mathbf{v}(0) = (v_{x0}, v_{y0}) \tag{34}$$

Once more, I emphasize that in the finite differencing notation I have adopted, the superscript n used above, as well as the superscript 1 on $\mathbf{r}^1$ and $\mathbf{v}^1$ are *labels* that will map directly to indexes of Matlab arrays.

We now use the first order forward FDA to replace the time derivatives in (22) and (23) with algebraic expressions. Specifically, we have

$$\frac{df(t)}{dt} \to \frac{f(t+\Delta t) - f(t)}{\Delta t}\,, \tag{35}$$

or using the finite difference notation

$$\left.\frac{df}{dt}\right|_{t=t^n} \equiv \left[\frac{df}{dt}\right]^n \to \frac{f^{n+1} - f^n}{\Delta t}\,. \tag{36}$$

We note that this approximation is perfectly valid for vector quantities such as $\mathbf{r}$ and $\mathbf{v}$ so using it in (22) and (23), we have

$$\frac{\mathbf{r}^{n+1} - \mathbf{r}^n}{\Delta t} = \mathbf{v}^n\,, \tag{37}$$

$$\frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\Delta t} = \mathbf{g}\,. \tag{38}$$

We can immediately rewrite these in a form suitable for use in advancing the simulation from one time step[3] t^n, to the next, t^{n+1}:

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \Delta t\,\mathbf{v}^n\,, \qquad n = 1, 2, 3, \ldots,\, n_t - 1 \tag{39}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \Delta t\,\mathbf{a}\,, \qquad n = 1, 2, 3, \ldots,\, n_t - 1 \tag{40}$$

These are the simple difference equations that we will solve to approximate the motion of the particle when it is freely following. Note that n runs over the values $1, 2, \ldots n_t - 1$ in the above. Since $\mathbf{r}^1$ and $\mathbf{v}^1$ are set by the initial conditions, this ensures that when are done we will have defined all n_t discrete values of the particle's position and velocity.

To deal with the particle-floor interaction, we use a somewhat *ad hoc* approach, whose specific order of accuracy in Δt we will not attempt to analyze, but which *is* consistent with the continuum model in the limit that $\Delta t \to 0$.

At any time step, n, we treat the position at step $n+1$ that we compute from (39):

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \Delta t \mathbf{v}^n\,, \tag{41}$$

as a *provisional* quantity. We check to see whether:

1. The new position of the particle would be below the level of the floor, i.e. whether

$$v_y^{n+1} < y^{\mathrm{f}}\,, \tag{42}$$

2. The particle is still over the floor—which we recall has a finite length—i.e. whether

$$x^{\mathrm{f}}_{\min} \le x^n \le x^{\mathrm{f}}_{\max}\,. \tag{43}$$

   (it is arbitrary, and for the purposes of this simulation, irrelevant whether we use x^n or the provisional advanced-time value x^{n+1} in this check). Note that if we didn't perform this check, our gaffe would be immediately apparent in the animation, since the ball would keep bouncing at $y = 0$ even when there was no floor underneath it.

[3]Note that I have a tendency to use "time step" in two distinct ways that may thus cause confusion, both here and when I speak about solving time-dependent problems using FDAs. First I use it to refer to any of the discrete times t^n at which approximate values are computed: this is the sense in which it is being used in the sentence to which this footnote is attached. Second, I use it to denote the discrete time interval, Δt (which I also refer to as the grid spacing, or mesh spacing), but then it is always referred to as *the* time step. You should usually be able to determine which of the two I mean from the context and/or the use of an indefinite vs definite article.

If both of these conditions are satisfied, then rather than using (39) and (40) to update the y components of the position and velocity, we do the following:

1. Flip the sign of v_y and scale its magnitude by the coefficient of restitution, κ:
$$v_y^{n+1} = -\kappa v_y^n \tag{44}$$
2. Leave the y-position unchanged:
$$y^{n+1} = y^n \tag{45}$$

However, we still use the x-components of (39) and (40) to update x and v_x.

Note that in converting the continuum description to an algorithmic prescription for the simulation (i.e. for the solution of the equations of motion), we have introduced precisely *one* additional parameter, namely the grid spacing, Δt.

Again, you may think that this has been an overly lengthy and elaborate description to yield a prescription for the simulation that you might well have been able to concoct intuitively. Thus, less you miss the forest for the trees, let me emphasize that my approach is intended to hammer home the importance of *having* a precise mathematical prescription for *any* simulation. Moreover, once we have worked out everything precisely using a simple FDA and physical scenario, it is then easier to extend and improve our simulation using, for example, a more accurate FDA for the equations of motion, or, as you will do, by adding additional surfaces from which the particle can reflect.

4.4 Implementation

4.4.1 Base script: `bounce.m`

The script which implements the simulation described above, including the generation of the AVI movie that I showed in class, and which is available via the *The road ahead* link on the course home page, is called `bounce.m`.

Your actual work in this problem will start with a modification of `bounce.m`.

Additionally, there is a version of `bounce` called `bounce_commented` which is *extensively* commented: `bounce` has about 75 lines of Matlab code[4] and about 225 lines of comments, `bounce_commented` has the same number of lines of code and more than 200 additional lines of comments.

Because I have described the mathematical formulation of the formula in such detail, and because `bounce` and `bounce_commented` are heavily documented, I am not going to discuss all of the implementation details here: some of that will be left to you to work out by reading through the code (especially `bounce_commented` if you're having problems), referring to the mathematical description above as necessary.

I do however want to point out one main difference between how `bounce` is coded, relative to `pendulum` and what I will suggest for your term projects.

> The distinction between `bounce` and the implementation of the pendula simulations, `pendulum` and `lpendulum`, that we worked with in the lab is that `bounce` does *not* store the entire solution. That is, it does *not* create an array that can contain the values of the particle position and velocity for *all* n_t of the discrete times, t^n, and then fill those arrays as the simulation proceeds.
>
> Instead, it only keeps track of the particle's current position and velocity, each of which is a length-two row vector. Every time that those two quantities are updated, `bounce` makes a new plot that shows the new position of the ball, and then as the script runs, we see an animation of the dynamics. That's all I wanted `bounce` to do, and since there's no need to store all the positions and velocities for further analysis, I don't.
>
> I'm hoping that this will also make your life a little easier when you modify and extend the code, since you will dealing with 1D arrays (vectors) rather than 2D arrays.
>
> *However*, for most of your term projects, you *will* want to store your dependent variables in arrays that hold all of the values, i.e. where one dimension has indexes that run from 1 to n_t, corresponding to all of the discrete time steps $t^n, n = 1, 2 \ldots n_t$. Among other things this will ensure that you can use the visualization utilities that I will provide for the N-body and cellular automata projects *directly*, without any additional coding on your part.

[4] Again, "a line of Matlab code" means something with is not a comment line nor an empty or whitespace-only line.

Mathematical Description	*Implementation*
t	`t`
Δt	`deltat`
$\mathbf{r}(t)$	`r` $(1 \times 2$ array$)$
$\mathbf{v}(t)$	`v` $(1 \times 2$ array$)$
$t_{\max}$	`tmax`
$x^{\mathrm{f}}_{\min}$	`xfmin`
$x^{\mathrm{f}}_{\max}$	`xfmax`
g	`g`
x_0, y_0, v_{x0}, v_{y0}	`x0, y0, vx0, vy0`
n_{bounce}	`nbounce`

Table 1: Definition of variable names used in the Matlab script `bounce` in terms of quantities appearing in the mathematical description of the model.

I'll wrap up this section with two final items:

First, Table 1 shows how the independent and dependent variables, as well as the main parameters for the problem are implemented in `bounce`, i.e. what variable names I'm using in the code for the various quantities that appear in the description of the model. They are all scalar variables, with the exception of `r` and `v` which are length-2 row vectors.

Second, here's the core code from `bounce` that does the bulk of the simulating:

```
for t = 0 : deltat : tmax
   % Plotting code omitted ...
   v = v + deltat * a;
   dr = deltat * v;
   if ( r(2) + dr(2) < fxy(2,1) )  &  ( fxy(1,1) <= r(1) & r(1) <= fxy(1,2) )
      v(2) = - kappa * v(2);
      dr(2) = 0;
      nbounce = nbounce + 1;
   end
   r = r + dr;
end
```

You can see that

1. It's not very long.
2. Parts of it are immediately recognizable as translations of the finite difference equations (39) and (40) into Matlab.

4.4.2 Conversion of the base script to a function: `bounce_fcn.m`

`bounce` is a Matlab script. When it runs, it displays the animation of the simulation, then saves the animation as an AVI file `bounce.avi`. However, the plotting (animation) and video recording can both be disabled by setting certain parameters as described in detail in `bounce_commented`.

All of the parameters that are used in the simulation are given values through assignment statements in the script. For example:

```
kappa = 0.98;
```

If we want to run a different simulation with different values of one or more parameters, we simply have to change the right hand sides of the appropriate assignment statements and rerun the script.

Recall that variation of problem parameters—both those that control the physics, such as κ, and those that control the numerics[5], such as Δt, is a key aspect of *any* problem in computational science.

When we are developing code, interactive experimentation by manually changing parameter values, and rerunning a script to see what happens is a great approach. However, when we get to the point that we want to *systematically* study what happens as parameters are varied—so called *parameter space surveys*—the manual approach is unwieldy, error-prone, too slow if a simulation completes significantly faster than you can type the changes to the assignment statements, mind-numbingly boring if a very large number of runs have to be made etc., etc.

What we want to do in this case is to convert the script to a *function*, so that the parameter or parameters that we want to change are supplied as *input arguments* to the function, and the results from the simulation that we wish to analyze are returned as *output arguments* from the function. Here I emphasize that when I say "convert" I mean "make a copy of something, give it a new name, and change the copy, *leaving the original unchanged."* Once we can pass values for the parameters that we wish to adjust ("play with") to the function, it is much easier to write additional code, a simple `for` loop for example, that can iterate over a bunch of parameter values, call the function with each, store the results in some fashion (likely in some type of an array), and then analyze the results when the survey is complete.

With this in mind let us consider the function `bounce_fcn`[6] which is a "functionized" version of `bounce`.

It has a header

```
function nbounce = bounce_fcn(tmax, deltat, kappa, vx0, ...
      plotenable, pausesecs, avienable, avifilename)
```

so it has 8 input arguments (I won't relist them!) and 1 output argument, `nbounce`, which is the number of times the ball bounced during the simulation.

The conversion of the script `bounce` to the function `bounce_fcn` is very straightforward. The function definition starts with the header as given above, and is terminated with an `end` statement that "matches" the header.

The body of the function is quite literally *identical* to the contents of the script, *except that the assignment statements that, in the script, give specific values to those variables that are now input arguments, i.e.*

```
tmax, deltat, kappa, vx0, plotenable, pausesecs, avienable, avifilename
```

are deleted. Since those 8 quantities, which were previously *variables* in the script, are now *arguments* to the function, they do not have to have their values set with assignment statements in the function body and, in fact, *should not have their values set there!* That would be akin to us passing the input $\pi/7$ to the $\sin(x)$ function, i.e. $x = \pi/7$, and having it "internally" change x, i.e. $\pi/7$, to $\pi/2$ and return 1.

And that's all there is to the conversion. In particular, I don't have to do *anything* in terms of the *output argument*, `nbounce` other than to ensure that its name appears in the correct place in the function header. Since all that we need to do to have a Matlab function return values is to ensure that the *output arguments have been assigned values*, and since the code in the script (and now in the function) does that, things work as they should.

When you look at the code for `bounce_fcn` you will see that I have "commented out" the assignment statements that needed to be deleted, rather than removing all trace of them from the code. I have also inserted the text `XXX` in each of those comments. This is strictly so that it is a little clearer from the code *per se* how the conversion works, and whether or not you want to adopt such a style when you do your own functionization—both to work this problem, as well as with your term projects—is entirely up to you.

Now that we have a function form of `bounce`, we will of course want a driver script for it, and `tbounce_fcn` is precisely such a beast.

It consists of exactly one line of Matlab code:

[5] "Numerics" is a colloquialism for numerical analysis and, more generally, computer calculations of the sort we are considering.

[6] From this point on, I will *not* usually remind you that the function `bounce_X` is defined in the source file `bounce_X.m`—that fact will always be implied.

```
nb = bounce_fcn(3.0, 0.01, 0.98, 0.5, 1, 0.01, 1, 'tbounce_fcn.avi');
```

and when this script runs, it does *exactly* what `bounce` does (with the parameter settings as defined in the original source file ~`phys210/matlab/bounce.m`), except that the AVI movie that is generated is called `tbounce_fcn.avi` rather than `bounce.avi`.

Note that I can simply change the function call in the script to

```
nb = bounce_fcn(3.0, 0.01, 0.98, 0.5, 1, 0.01, 0, 'tbounce_fcn.avi');
```

to disable the AVI generation (the 7th input argument, `avienable`, is now 0), or use

```
nb = bounce_fcn(3.0, 0.01, 0.98, 0.5, 0, 0.01, 1, 'tbounce_fcn.avi');
```

to disable plotting altogether (the 5th input argument, `plotenable`, in now 0, and as described in the code, disabling plotting automatically disables AVI generation, so that `plotenable=0` "overrides" `avienable=1`.

This last possibility anticipates our use of `bounce_fcn` for a parameter-space survey, where we vary one of the "physical" parameters to study the effect on the physical behaviour exhibited by our simulation. When we do such a survey, we generally want it to execute as quickly as possible and, as you will see, disabling plotting output makes the calculations complete *much* more rapidly.

4.5 Numerical experiments with `bounce_fcn`: `bounce_survey`

The script `bounce_survey` uses `bounce_fcn` to investigate the functional dependence of n_{bounce} on v_{x0}, that is, how many times the ball bounces as a function of how fast the ball is moving in the x-direction at the initial time. Since v_x remains constant in our simple model—there are never any forces acting in the x-direction—we don't need to do a simulation to figure out what $n_{\text{bounce}}(v_{x0})$ is going to look like. You should be able to convince yourself that—at least in the limit that $v_{x0} \to 0$—we should have

$$n_{\text{bounce}} \propto \frac{1}{v_{x0}} \,. \tag{46}$$

I promise that the bouncing behaviour will be more interesting when you do your surveys.

Here's the majority of the script, with comments and whitespace stripped:

```
deltat = 0.01;
kappa = 0.98;
plotenable = 0;
pausesecs = 0;
avienable = 0;
avifname = '';

vx0min = 0.05;
vx0inc = 0.01;
vx0max = 1.00;

tmax = (1 / vx0min) + 3;

vx0 = vx0min : vx0inc : vx0max;

nb = zeros(1, length(vx0));
for ii = 1 : length(vx0)
   ii
   nb(ii) = bounce_fcn(tmax, deltat, kappa, vx0(ii), plotenable, ...
      pausesecs, avienable, avifname);
end
```

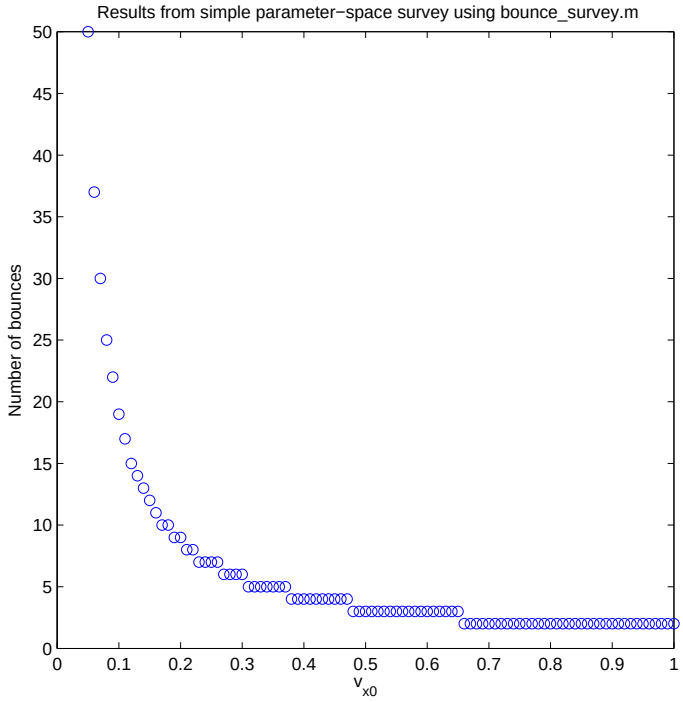


Figure 1: Results from a parameter survey implemented as the script `bounce_survey` which in turn calls the function `bounce_fcn` to compute the number of bounces, $n_{\rm bounce}$, as a function of v_{x0}.

```
clf;
plot(vx0, nb, 'bo');
```

The script runs `bounce_fcn` with 96 separate values of v_{x0}:

$$v_{x0} = 0.05,\, 0.06,\, 0.07,\, \ldots\, 0.98,\, 0.99,\, 1.00\,, \tag{47}$$

accumulates the corresponding bounce counts in the length-96 1D array (vector), `nb`, generates the plot shown in Fig. 1, and saves a JPEG version of the graph.

As anticipated, we see that the number of bounces goes like $1/v_{x0}$ for small v_{x0}.

Now, `bounce_survey` isn't a very long script, but it could still be a lot shorter since I've only introduced and given values to all of the variables from `deltat` and `kappa` through `tmax` and `vx0`, which are *not* varying, for readability.

Thus, the following few lines of code perform precisely the same set of calculations.

```
vx0 = 0.05 : 0.01 : 1.00;
nb = zeros(1, length(vx0));
for ii = 1 : length(vx0)
   nb(ii) = bounce_fcn(23, 0.01, 0.98, vx0(ii), 0, 0, 0, '');
end

clf;
plot(vx0, nb, 'bo');
```

Again, I'll leave it to you decide whether you want to adopt a terse or verbose coding style for your parameter-survey script: the overriding consideration is that the script does the correct job.

4.6 Problem specification: Summary of what you need to do

OK, at long, long last it's your turn, and here's the executive summary of your mission:

1. Modify `bounce` to produce a new script `mbounce` ("m" for "multi") that performs the same simulation—with the same animation and AVI movie generation facilities—as `bounce`, but with two reflective surfaces added. I showed a video from my implementation of `mbounce` in the Nov. 4 lecture.
2. Run the script interactively and modify one of the problem parameters until you find a simulation satisfying a certain condition that I will specify. Make a movie of that simulation and incorporate it in your course web page.
3. Convert `mbounce` to a function `mbounce_fcn` with a specified header and code a simple driver `tmbounce_fcn` that reproduces the action of the script for the particular parameter setting that you determined in the previous step.
4. Code a script `mbounce_survey` that uses `mbounce_fcn` to perform four separate but related surveys of the type done by `bounce_survey` and plot the results.
5. Provide an interpretation of the results of the survey in a `README` file (speculations will be acceptable).

And you'll be done ... Honestly ... I promise ... Unless you want to extend the simulation for bonus credit ...

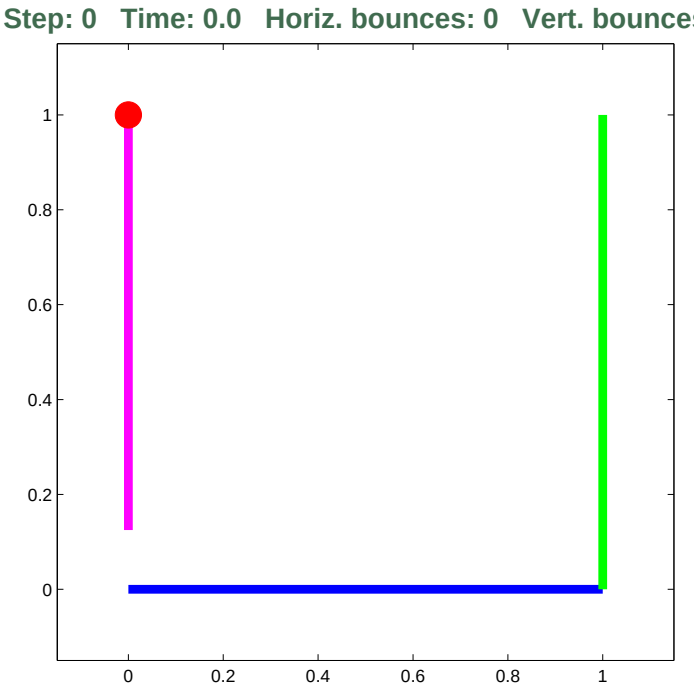


Figure 2: Setup for `mbounce` simulation. Note the addition of two walls: the one on the left does *not* intersect the floor. Also shown is the format for the plot title—which includes fields for: (1) the time step number (defined by $t/\Delta t$), (2) the time, t, (3) the horizontal bounce count, $n^{\mathrm{H}}_{\mathrm{bounce}}$, and (4) the vertical bounce count, $n^{\mathrm{V}}_{\mathrm{bounce}}$.

IMPORTANT NOTES!!

1. Remember that all of the code I am providing you is located in ~`phys210/matlab`. I will *not* repeatedly tell you where files are to be copied in the following.
2. Please, please, please do *not* write `mbounce` from scratch, even if you know you can do so. It will needlessly complicate the already non-trivial grading task that the TAs face.
3. Every script and function that you write gets defined in a separate source file (text file) with a `.m` extension.

Here are the details, kept purposely brief now:

Problem 4.6.1: `mbounce.m`

If you haven't yet done so, copy `bounce.m` to your solution directory and then make a copy of that copy called `mbounce.m`.

Modify `mbounce`[7] so that it incorporates two additional surfaces from which the ball can bounce (reflect). The geometry of the additional surfaces is shown in Fig. 2

The additional elements are both walls, i.e. vertical surfaces as opposed to the floor which is a horizontal surface. The coordinates of the endpoints of the left wall are

$$(0, 0.125) \qquad \text{and} \qquad (0, 1)\,, \tag{48}$$

and the coordinates of the endpoints of the right wall are

$$(1, 0) \qquad \text{and} \qquad (1, 1)\,. \tag{49}$$

[7] Again, when I say "modify `mbounce`", for example, it is to be understood that I mean "modify the definition of `mbounce` by editing the source file `mbounce.m`."

I have colored the left and right walls magenta and green respectively (color available online)—you are free to use whatever colors you wish, both for the floor *and* the walls, as long as each is a different color and that all three colors are clearly distinct.

Implement the interactions between the particle and the walls in the same way that we treated the interaction between the particle and the floor in `bounce`, *except assume that the coefficient of restitution of both walls is 1 (unity).* Thus, at each time step compute a provisional position using (39) and check *each* surface to see whether the particle's trajectory has intersected it. If it has, and the surface is the floor, then `mbounce` should do exactly what `bounce` does. If, on the other hand, the surface is a wall, then reverse the x-component of the velocity (leaving the y-component unchanged), and don't change the particle's x-position. Again, there is no factor of κ in the change of the x-velocity component for a wall interaction: simply code $v_x \to -v_x$.

Important

Be sure to check for intersections with *all* of the surfaces before doing the final update of the position, which in `bounce` is the statement:

```
r = r + dr;
```

If you don't, you may find that the simulation "fails" under certain circumstances.

Additional changes relative to `bounce` that you must make

1. The script must keep *separate* counts of the number of horizontal bounces, $n^{\mathrm{H}}_{\mathrm{bounce}}$ (bounces off the walls) and vertical bounces, $n^{\mathrm{V}}_{\mathrm{bounce}}$ (bounces off the floor). Note that there is only *one* horizontal bounce count which gets incremented if the ball hits either wall.
2. The script must display (1) the time step number (defined by $t/\Delta t$), (2) the time, t, (3) the horizontal bounce count, $n^{\mathrm{H}}_{\mathrm{bounce}}$, and (4) the vertical bounce count, $n^{\mathrm{V}}_{\mathrm{bounce}}$ in the plot title as shown in Fig. 2, and just to be clear, the displayed values must change as the simulation proceeds, as is the case for `bounce.` You'll need to use the `sprintf` function to create the title string. If you can't figure out how to do this from the supplied code, refer to the Matlab programming lab notes where I've added a description of it (search for `sprintf` in the notes), use the Matlab built-in documentation facility, or ask for help as necessary.
3. **Very, very, important:** The script **must exit** from the main time-stepping `for` loop if and when the particle moves beyond the boundary of the plotting region which is defined by

$$-0.15 \le x \le 1.15 \qquad -0.15 \le y \le 1.15 \tag{50}$$

   That is, within the time-stepping loop you will code an `if` statement of the form

```
if <particle has left the plotting region>
   break;
end
```

   where it is up to you to replace `<particle has left the plotting region>` appropriately. *Hint:* Earlier in this problem set you wrote (or should have written!) a function that could be useful here. If you do this correctly, then the last frame of your animations should always show the ball "frozen" on some point of the boundary of the plotting region, as the movie I showed on Nov. 4 did. If you need more information on how the `break` statement works, refer to the Matlab Programming Labs notes, where there is a brief discussion of it that we didn't explicitly cover in the labs, and ask for help as necessary.
4. *Clarified:* As with `bounce`, if `avienable` is non-zero, an AVI file must be generated but its name must be `mbounce.avi`, not `bounce.avi`.

Once you are satisfied that your script is running properly, set the following values

```
plotenable = 1;
pausesecs = 0;
avienable = 0;
tmax = 1000;
deltat = 0.01;
kappa = <your value here>
g = -20;
x0 = 0.0;
y0 = 1.0;
vx0 = 0.4;
vy0 = 0.0;
```

VERY IMPORTANT: Note that `vx0` has a different value here than in `bounce`. Be sure to use 0.4, *not* 0.5.

Now, run `mbounce` interactively, with plotting enabled, but AVI generation disabled and vary κ (`kappa`) until you find a value for which the ball bounces off the floor 40 to 45 times, that is where

$$40 \leq n_{\text{bounce}}^{\text{V}} \leq 45\,. \tag{51}$$

Once you find a value of κ that meets this criterion, rerun the simulation with

```
avienable = 1;
```

and

```
avifilename = 'mbounce.avi';
```

and incorporate the AVI movie that your script generates into your web page, inserting it below the plot from Problem 1. *Clarified:* Note that the above statements do *not* necessarily have to be on consecutive lines in your script as previous versions of the homework might have suggested they should be—they aren't in `bounce`. As is the case for image files, you will need to *copy* `mbounce.avi` to `/phys210/$LOGNAME/public_html` to ensure that my web server can export it.

Clarified: By "incorporated" I mean to add a link that points to the AVI file, in the same way that I have done in the main course web page, i.e. you click on the *The road ahead!* link and, depending on how your browser is configured it either displays the movie in-place, or starts an external video player. If you want to do something *slightly* more sophisticated, see the *Homework 3 help page* for one technique.

Important

To check that you *have* incorporated the AVI movie correctly, ensure that you can access it from a browser session that is *not* running on one of the lab machines (for example, from your smartphone or laptop or home PC) by navigating to your course page that you made in the first homework and clicking on the link to the video.

Important

The most straightforward way to proceed with this problem is to add additional code that is "specialized" for each of the walls, i.e. that uses separate arrays to define the end-points, separate calls of `plot` to draw them etc. Unless you are skilled at Matlab programming, this is the approach I advocate you take, at least at first.

However, another approach, which then makes a simulation with a more or less arbitrary number of horizontal or vertical surfaces very easy, is to store the information about the surfaces collectively, i.e. using arrays that have an additional dimension that enumerates the surfaces (i.e. whose indices would be 1, 2 and 3 in this case), and then code an internal loop within the main time-step loop that iterates over the surfaces and looks for intersections. Similarly, the plotting of all of the surfaces can then be done in a loop. You are certainly free to try such an approach if you wish.

Important

If you run a script or function repeatedly with AVI recording enabled, you may notice a significant degradation in performance as you make more and more videos. In this case I suggest that you first try using the `clear` command (but note that this will wipe all of the variable assignments that you have made in the session), or restart your Matlab session.

In general it is best to keep AVI recording disabled until you are sure that you have something that you really want to record.

Problem 4.6.2: `mbounce_fcn` and `tmbounce_fcn`

Convert `mbounce` to a function `mbounce_fcn` with a header

```
function nbounce = mbounce_fcn(tmax, deltat, kappa, plotenable, pausesecs)
```

where the input arguments `tmax, deltat, plotenable, pausesecs` are as previously, `kappa` is the coefficient of restitution of the floor, κ, and the output argument, `nbounce` is a length-2 row vector with elements:

$$\texttt{nbounce(1)} = n^{\mathrm{H}}_{\mathrm{bounce}} \tag{52}$$

$$\texttt{nbounce(2)} = n^{\mathrm{V}}_{\mathrm{bounce}} \tag{53}$$

To do this, and as described above for the `bounce` to `bounce_fcn` conversion, first *copy* `mbounce.m` to `mbounce_fcn.m` and then modify `mbounce_fcn.m`, leaving `mbounce.m` as is.

Test your conversion by coding a driver script, `tmbounce_fcn.m`, that is completely analogous to `tbounce_fcn.m` and which can be as short as one line of Matlab.

Specifically, when you submit your homework, execution of this driver, `tmbounce_fcn`, should cause your `mbounce_fcn` function to perform precisely the same simulation that execution of the original script `mbounce` does (i.e. make sure that all of the parameter settings match up).

Problem 4.6.3: Parameter space surveys:

Copy the script file `mbounce_survey_template.m` to your solution directory, and execute it:

```
>> mbounce_survey_template
```

If you have implemented `mbounce_fcn` properly you should see a plot similar to that shown in Fig. 3.

Now, copy `mbounce_survey_template.m` to a new script file `mbounce_survey.m`. Reading and following the instructions in the code for `mbounce_survey_template` carefully, modify the code for `mbounce_survey.m` to implement the parameter surveys and plotting tasks that are described within it.

VERY IMPORTANT

That is, the instructions for this last part of the job, which in some sense is the most important, since it's where the interesting results are generated, are not here in the handout, but in the source file `mbounce_survey_template.m`.

Be sure to heed the warnings about making sure that your implementation of `mbounce_fcn` is working properly before you attempt to do the final surveys. If you don't, if may take far too long for the calculations to complete. If things are working properly, `mbounce_survey` should take only 4 or 5 minutes or so total to run on one of the lab machines. Seek assistance if your computations seem to be taking a long time, and you can't determine what is wrong.

When you run your final version of `mbounce_survey`, it should generate five JPEG files: `hw34_s1.jpg`. `hw34_s2.jpg`, `hw34_s3.jpg`, `hw34_s4.jpg` and `hw34_s5.jpg`

You do *not* have to incorporate these into your web page, but can do so if you wish.

You will then be done with the calculations: but study the structure of the plots carefully.

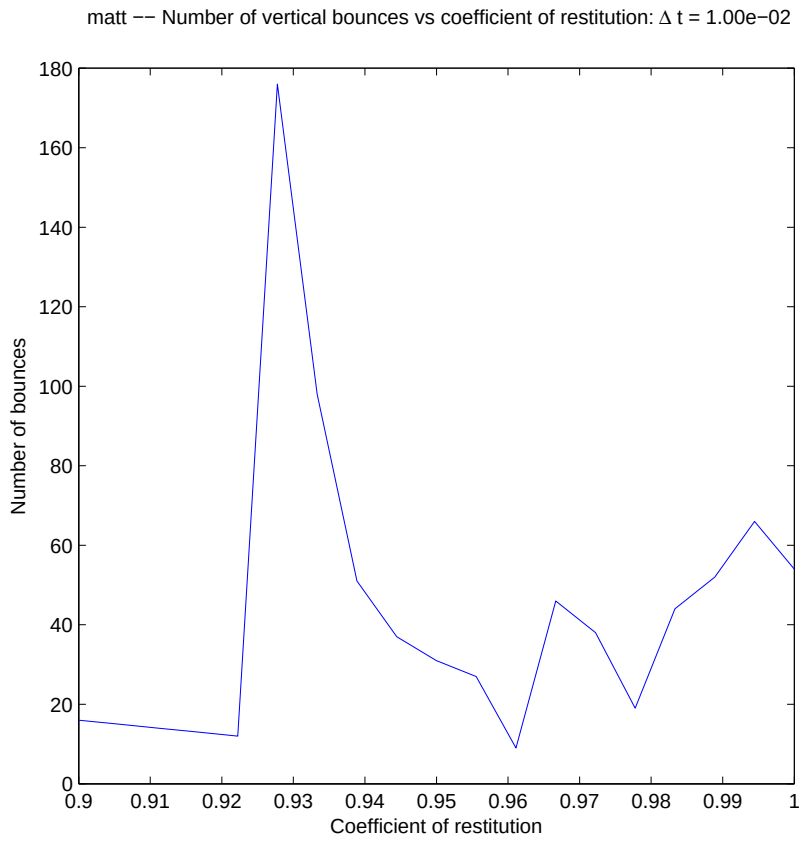


Figure 3: Plot generated by running `mbounce_survey_template` with the instructor's implementation of `mbounce_fcn`. The plot you see when you run the script and it invokes *your* implementation of `mbounce_fcn` should be similar. Note that you need to code `mbounce_fcn` *before* you run the script, and that you need to code it precisely to the specifications.

Problem 4.6.3: Interpretation of results

Provide answers to the following questions (speculate as need be). Your comments do *not* have to be extensive, but you should attempt to put some thought into them.

- What do you notice about the first four plots, especially relative to Fig. 1.
- Was the "structure" of the plots—their overall appearance, smoothness, or lack thereof, etc.—a surprise to you? If yes, what you were expecting? If no, explain briefly why you were not surprised.
- Provide a brief interpretation of the scatter plot (the fifth and last plot). If we generated and plotted similar data with `bounce_survey`–i.e. reran the set of simulations in that script with $\Delta t = 0.001$ and then made a scatter plot with the $\Delta t = 0.01$ data, what would you expect to see? Note that it wouldn't be too difficult to modify `bounce_survey` so it did the extra runs and made the scatter plot, so you could test your conjecture.
- On the basis of the limited evidence that you have gathered from the numerical experiments, do you think the nature of the results as illustrated by the plots is due to

  1. Inherent physically-based chaotic behaviour in the system.
  2. Limitations of the numerical technique used.
  3. Both.
  4. Something else: specify.
  5. I have no clue (perfectly fine to admit this!)

- If you can, suggest how you might modify and/or improve the simulation with an aim to determine whether what you have observed ("measured") is a reflection of the physics being modeled, or the numerical method being used, or something else.

IMPORTANT CONCLUDING REMARKS

Obviously, there's quite a bit to do here, and it might not be possible for you to finish it all in the allotted time, especially without jeopardizing your performance in your other courses.

Please just do your best and rest assured that the marking won't be harsh. This does *not* mean, however, that you should "blow off" this problem. In terms of what you should get out of the course, and how it will help you with your term projects, it is by far the most important exercise among the homework problems I have assigned this term.

Template and sample code

1. `bounce.m`
2. `bounce_fcn.m`
3. `tbounce_fcn.m`
4. `bounce_survey.m`
5. `mbounce_survey_template.m`

File inventory:

First, note the following:

- All scripts and functions must have plotting and video recording options set as indicated.
- All scripts and functions must have values for parameters, arguments to functions etc. set so that when the TAs execute them, they produce the same plots and, *if the TA chooses to turn AVI recording on in your code*, the same AVI movie that you are submitting. For example, if you found that a value $\kappa = 0.852$ produced 41 bounces, and that's the value that you used to make `mbounce.avi`, don't submit your work with κ set to some other value in `mbounce.m`.
- **IMPORTANT:** Video recording should be *disabled* in all scripts and functions that you submit. None of your code should make an AVI movie unless the TAs change it so that recording is enabled.

1. `mbounce.m` (configured with `plotenable=1` and `avienable=0`)
2. `mbounce_fcn.m` (configured with `avienable=0`)
3. `tmbounce_fcn.m` (configured so that `mbounce_fcn` is called with the value 1 for its argument `plotenable`)
4. `mbounce_survey.m`
5. `mbounce.avi` (and incorporate into Web page)
6. `hw34_s1.jpg`
7. `hw34_s2.jpg`
8. `hw34_s3.jpg`
9. `hw34_s4.jpg`
10. `hw34_s5.jpg`
11. `README`

Congratulations!! You're done!!