PHYS 210: Introduction to Computational Physics Fall 2104 Homework 2
Version 4: October 5, 2014

Problems 1 and 2 due: Thursday, October 9, 11:59 PM
Problems 3, 4 & 5 (and optional Problem 6) due: Thursday, October 16, 11:59 PM

NOTE: This is the complete handout, and will be subject only to bug fixes. See http://laplace.phas.ubc.ca/210/Homework2.html for a complete list of such fixes.

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. Note that this homework is being distributed in two stages, with two separate due dates.
   (a) Problems 1 & 2, which I hope you will find relatively straightforward, are due Thursday, October 9th at 11:59 PM.
   (b) Problems 3, 4 & 5 and the *strictly optional* bonus Problem 6, are due Thursday, October 16th at 11:59 PM. The primary motivation for the split deadlines is still to ensure that you are left with sufficient time to complete the most challenging components of the assignment.
   (c) To repeat, Problem 6 is *strictly optional*. It involves extending the Chaos Game you will implement in Problem 5. It is worth some small but non-negligible amount of extra credit, and is also due on October 16.
2. As promised, this handout has ballooned in length, and is now 20 pages. Again, do not be taken aback by its length—most of the verbosity should be viewed as additional course note material that addresses topics which will be pertinent to the rest of the course work and, with luck, to some of your future computing programming endeavours. A rough measure of the number of lines of code you can expect to write is provided towards the end of this list.

   That said, you *are* expected to read all of the handout, and read it fairly carefully :-)
3. The following assignment requires
   (a) Using `xmaple` (the graphical version of `Maple`) to produce `Maple` worksheets (Problems 1 and 2).
   (b) Preparing source code for `Maple` procedures in plain-text files that can be input into `Maple` or `xmaple` via the `read` command (Problems 3, 4, & 5).
   (c) In addition, since Problem 5 requires the use of `Maple`'s plotting facilities you will need to use `xmaple` to complete it (although you can still use command-line `Maple`, with its funky character-based graphics, to get a start on it.)
4. **IMPORTANT!!** In order to complete your homework—especially for Problems 1, 2, & 5—you should be prepared to use the computer lab. If you try to use `xmaple` remotely on `hyper` (using, e.g., `Xming/putty` under Windows, or a Mac) you may find the performance unacceptably sluggish. However, for Problems 3 and 4 where you are to write procedures in text files, you should be able to work remotely using command line `Maple` running on `hyper`, and as just noted, it's certainly not *essential* to use `Maple`, to do all of your work for Problem 5.
5. **IMPORTANT!!** To complete the homework, all of the files specified below must exist and be in their proper locations within your `/phys210/$LOGNAME/hw2` directory. The TAs and I must be able to read all of the required worksheets and text files into `Maple` sessions of our own without encountering errors. Required file inventories are provided for each problem. Those inventories—including `README` files—constitute all of the work that you need to submit.
6. Whenever working with *any* worksheet in `xmaple`, be sure to save your work frequently, using, for example, `Ctrl-s`. This will minimize the amount of time and effort that you might lose should the interface crash (as it has been known to do occasionally)
7. **IMPORTANT!!** Although Problem 1 is straightforward, it will take some time to fully complete. You will be provided with some lab time to work on it (and the other problems), but you are advised not to leave its completion until just before the due date.

— Instructions continue on flip-side. —

8. **IMPORTANT!!** Much of this assignment involves programming:

   (a) If you are new to coding it is *strongly* recommended that you start working on the programming problems earlier rather than later.

   (b) **If you are new to coding this particularly pertains to Problem 5.**

   (c) If you are new to coding and are having difficulty with it, *ask for help!* I am not feeling overwhelmed with questions from the class yet, and, for example, if you are sitting in the lab figuratively (or perhaps literally) bashing your head against the screen, you can always send me a quick email, or text message (778-323-4887) to see whether I am available to pop down to assist. I don't mind doing this if I have the time.

9. **IMPORTANT!!** I have labelled the problems that I consider more challenging with ($\star$) (Problems 2.5 and 4.3) and ($\star\star$) (Problem 5), with the ($\star\star$) denoting more challenge than ($\star$). Again, these annotations aren't intended to frighten you, only to give you a heads-up that those questions may be more time consuming than the others.

10. None of the programming problems require you to write a lot of code, at least in principle. As a rough metric, here are the number of non-comment, non-empty lines in my implementations.

    - 3.1) `V_powers`: 8
    - 3.2) `V_powers2`: 12
    - 3.3) `M_identity_rotate`: 8
    - 4.1) `lreverse`: 8
    - 4.2) `ltail`: 10
    - 4.3) `lreplicate`: 15
    - 5) `chaos`: 23
    - Total: 84
    - Total - (procedure headers and the two other given statements for `chaos`): 75

    That is, the number of lines of code that you have to write can, in principle, fit on about 1.5 pages of this handout.

    Note that I have not included the lines for the test scripts in my count, which should amount to *roughly* the same length of code.

11. **Commenting**: Extensive commenting of your code is *not* required. For the simple procedures one or two lines stating what the procedure does will suffice. For `chaos` a few more lines may be in order, but you will not be penalized for a lack of commenting of *any* of the procedures.

12. **VERY IMPORTANT!!! Additional resources:** I have provided example procedure-definition and test-script files to be used in conjunction with Problems 3, 4 & 5. See the end of the respective problem definitions for details. I think that you will find many of these helpful and suggest that you look at them *before* you start coding. On the other hand, if you are determined to do things on on your own, and can do so in a reasonable amount of time, then by all means do so! The procedures that we covered in the labs may also be of use to you.

13. Please follow all instructions for each problem carefully, again ensuring that all requested files are in their correct locations—i.e. within subdirectories of `/phys210/$LOGNAME/hw2`—and with the correct names. Also note that any reference to the directory `hw2` below is implicitly a reference to `/phys210/$LOGNAME/hw2`.

14. 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.

15. Finally, as always, let me know immediately if there is something that you do not understand, or if you encounter serious problems with any part of the assignment.

Problem 1: As an introduction to `Maple` we went through a worksheet that I created, and which was based on Chapter 2 of the *Maple Learning Guide*.

IMPORTANT!! I distributed two versions of the worksheet: the copy handed out on Sep. 30 had important amendments to do with the replacement of the `array` command with `Array`.

You *must* refer to the new version in what follows.

The updated worksheet is available online as a Postscript (not PDF) file via `Course Home Page -> Course Notes -> Maple -> Worksheet [PS ] showing calculations.`.

In your `hw2` directory create a subdirectory `a1`. In that subdirectory, make a facsimile of my worksheet, called `a1.mw`, by entering all of the Maple commands contained within it, and providing annotations for the various sections, commands, etc., as I have done.

However, and as described in the worksheet itself, do *not* reproduce any annotations that appear in *italic* (*slanted*) font.

This is so important that I will repeat it.

However, and as described in the worksheet itself, do *not* reproduce any annotations that appear in *italic* (*slanted*) font.

Recall that I also distributed a hardcopy of Chapter 2 of the *Learning Guide* in its entirety. You may wish to use this for supplemental information if you encounter, for example, difficulties getting commands to evaluate properly—it too is available online through the Course Notes web page.

We went through the procedure for inserting annotations (text comments) in a worksheet during a lab session. For completeness, that information is reproduced below.

Finally, please observe the cautions made in the preamble concerning:

1. The time it may take to complete this problem.
2. The wisdom of frequent use of `Ctrl-S`, or some other save mechanism when working with *any* `Maple` worksheet.

To create annotations (comments) of the sort I made, use the three icons located roughly under the "Drawing" label on the top tool bar. From right to left these are

- An icon that resembles an hourglass—hovering the mouse over it displays the text "Enclose the current selection in a document block, or create a new one"
- An icon representing the `Maple` prompt: "Insert Maple Input after the current execution group"
- An upper case T: "Insert plain text after the current execution group"

To insert a comment, position the cursor to the immediate right of the prompt on the line *before* the location where you want to do the insertion.

Click on the hourglass icon, and then on the T.

You can then type in plain text to create the annotation. Once the note (section heading, comment etc.) has been inserted, you can alter its appearance (font, font size, style etc.) by sweeping the text and using the icons and pulldowns immediately above the main input/output area.

Since text blocks are always inserted *after* the line on which the cursor is positioned, it is slightly troublesome to begin a worksheet with an annotation.

— Instructions continue on flip side. —

One way to do so is as follows. First, insert an execution group before the initial command of the worksheet by positioning the cursor beside the corresponding prompt and typing `Ctrl-k`. Using the above prescription, insert the text that is to appear at the beginning of the worksheet after the new execution group. Now delete the execution group before the annotation by repositioning the cursor beside the corresponding prompt, pressing and holding the `Ctrl` key and pressing the `Delete key` two times.

Your annotations do *not* have to match mine *precisely* (i.e. you don't have to use the same font [I used Lucida Sans], font size, style etc.), but, again, you should try to ensure that everything that I have typed into my worksheet—other than the italicized annotations—is included in yours.

File inventory:

1. `a1.mw`

Problem 2: Make the subdirectory `hw2/a2`, and within that subdirectory, and using `xmaple`, create a worksheet called `a2.mw` in which the following computations and plotting have been carried out:

$$\frac{\partial^3}{\partial x^2\,\partial y}\left(\left(\cos\left(\frac{\ln(3x+6)}{y}\right)\right)^2\right)\Bigg|_{x=1,y=4} \tag{2.1}$$

$$\int \frac{x^7+6x^3-4}{x^2-1}\,dx \tag{2.2}$$

$$\int_{y=1}^{y=3}\int_{x=1}^{x=2} \frac{x^3-y^2}{x^2+y^2}\,dx\,dy \tag{2.3}$$

Taylor series about $x=0$, up to and including the $O(x^{10})$ term, of $\sqrt{\cos(x)+\sin(x)+\tan(x)}$ (2.4)

A plot of the error in the above expansion (including the $O(x^{10})$ term), for $0 \le x \le 0.01$ (2.5) $(\star)$

Please note the following important points:

1. For (2.1) and (2.3), your answers *must* result in *floating point numbers* when you worksheet is executed, not general algebraic expressions.
2. For (2.4) and (2.5), "including the $O(x^{10})$ term" means that the explicit form of the term with x^{10} in it must appear in the expansion. As a concrete example, the following expansion for the exponential function includes the $O(x^3)$ term:
$$\exp(x) = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + O(x^4)$$
3. As the $(\star)$ indicates, you may find (2.5) a bit more challenging that the other parts of this problem. Let $F(x)$ denote the function which is being Taylor-expanded, and $T(x)$ its Taylor expansion. Then define the error, $e(x)$ (the quantity which is to be plotted), as
$$e(x) \equiv F(x) - T(x)\,.$$
Be sure to set `Digits` to a value sufficiently large to produce an accurate plot. Finally, note that you can't plot a Taylor series directly (due to the $O(x^p)$ term that generically appears, and which has no specific value). However, as discussed in class (in the handout "Some Useful Maple Commands", also available via the Course Notes web page), you can readily convert a series to a polynomial using the `convert` command.

File inventory:

1. `a2.mw`

Notation, and translation of mathematical expressions that use indexed objects into `Maple`.

The remainder of this assignment involves `Maple` *programming*. One of the key skills that a programmer needs to develop, particularly one working in a field of computational science, is the ability to translate problems expressed in traditional mathematical form to code.

Many of the problems that we encounter in numerical analysis and computational physics are naturally phrased in terms of objects such as vectors, matrices, lattices etc. which, in turn, are naturally represented in programming languages as arrays, or, should the language support them, lists. Most of the problems below require you to convert mathematical expressions and formulae into procedures that create, manipulate, and return `Maple` lists and arrays. It may help to think of the procedures as array- or list-fabrication machines—you provide some input, they return—i.e. give back to you—an array or a list.

When casting problem specifications in mathematical form, I will adopt the convention that elements of lists, vectors and matrices—the latter two surely being familiar to you from your physics and mathematics studies—can be denoted using the same type of subscript notation that we have seen `xmaple` use in its output display.

Thus, for example, l_i will denote the i-th element of a list l, V_j the j-th element (component) of a vector V, and A_{ij} the ij-th element of a matrix A, i.e. the element in the i-th row and j-th column. Although this may seem obvious to you, it is worth mentioning that there is nothing special about the characters i and j that I have chosen to use as subscripts: in all cases which specific character I use is *arbitrary*—it simply serves to label the various elements.

Additionally, for each list, vector and matrix, I will specify the range over which the subscripts vary using another standard mathematical notation.

For a list, l, this will be

$$l_i, \quad i = 1, 2 \ldots n_l \,, \tag{1}$$

where n_l is the length of the list (i.e. the number of elements in the list); for a vector, V,

$$V_j, \quad i = 1, 2 \ldots n_V \,, \tag{2}$$

where n_V is the number of elements/components of the vector; and for a matrix, A,

$$A_{ij}, \quad i = 1, 2, \ldots n_r, \quad j = 1, 2 \ldots n_c \,, \tag{3}$$

where n_r and n_c are the number of rows and columns, respectively, in the matrix, so that A has $n_r \times n_c$ elements.

Observe that in equation (3), the notation

$$i = 1, 2, \ldots n_r, \quad j = 1, 2 \ldots n_c \tag{4}$$

is to be interpreted as

> For *every* i running (ranging) over the values $1, 2, \ldots n_r$, j runs (ranges) over the values $1, 2 \ldots n_c$

which is precisely equivalent to

> For *every* j running (ranging) over the values $1, 2, \ldots n_c$, i runs (ranges) over the values $1, 2 \ldots n_r$

Again, although perhaps obvious, I'll point out that the subscripts l, V, r and c in the notations n_l, n_V, n_r and n_c serve a different purpose than those in l_i, V_j and A_{ij}. That is, n_l, n_V, n_r and n_c are all *single* integers—*not* range-valued quantities like i, j above—and the subscripts remind us which specific object we are referring to.

Finally, if isn't already clear to you, note the following:

1. To represent a length-n_l list, l, with typical element, l_i, in `Maple`, use a `list`. Recall that lists do *not* need to be explicitly defined; they can be created and modified "on the fly".
2. To represent a length-n_V vector, V, with typical element, V_j, in `Maple`, use a one-dimensional (1D) `Array`, created with

```
<vector_name> := Array(1 .. <vector_length>)
```

3. To represent an $n_r \times n_c$ matrix, A, with typical element, A_{ij}, in `Maple`, use a two-dimensional (2D) `Array`, created with

```
<matrix_name> := Array(1 .. <number_of_rows>, Array<1 .. <number_of_columns>)
```

Since I have used meta-syntax in the above, let me be more specific. If the problem description refers to a vector V with size n_V. then `V` and `n_V` (or variations on that theme) are natural names to choose for your `Maple` representation. Thus an appropriate `Maple` array-definition statement would be

```
V := Array(1 .. n_V);
```

Note that at the time that this command is executed, `n_V` must have been assigned an integer value greater than 0.

Similarly, if the problem description refers to a matrix A with n_r rows and n_c columns, then `A` and `n_r` and `n_c` (or `nr` and `nc` etc.) are natural names and the array definition would be

```
A := Array(1 .. n_r, 1 .. n_c);
```

Again, at the time that this command is executed, both `n_r` and `n_c` must have been assigned integer values greater than 0.

Finally, note that we could equally well adopt a notation where the elements of a vector or matrix are labelled using superscripts, rather than subscripts. So for a vector, W, for example we would have

$$W^j, \quad j = 1, 2, \ldots n_W$$

and, again, it would be natural to represent the elements of W as a 1D array in `Maple`.

Having gone to some length to (a) define and describe the notation that I will be using below, and (b) guide you in how expressions using that notation should be converted to `Maple`, I will keep the specifications of the problems that follow "mathematical" and fairly terse (at least by my normal standards), leaving to you the job of translating my wishes into your (`Maple`) commands—bad genie-reference-pun intended!

So, if you have difficulty figuring out exactly what I'm asking for in the following, please first re-read this section carefully. If that doesn't clear things up for you, then by all means ask for assistance. Again, however, being able to translate a specification given in traditional mathematical form to correct *code is a major part of what being a good scientific programmer is all about.*

Programming tips and suggestions.

1. First and foremost, *before you start to code*, do your best to
   - understand *precisely* what it is that you need to do, and
   - understand *precisely* how you are going to do it,

   Leave the hacking—making it up in front of the terminal—to the hackers. Don't be afraid to get out some paper and a pen or pencil and work things out that way first. You may find that it saves you a lot of time in the end.
2. Corollary: Think before you type.
3. Second and foremost, recognize that you *will* make mistakes in coding. Don't take it personally. Assume that you *will* screw up from time to time, and maybe even a lot of the time. Try to view finding and fixing your screw-ups as a challenge, or better yet, a game. You'll find that as you get more efficient in locating and repairing your mistakes, you'll also tend to make fewer of them.
4. Try to code incrementally, with continuous testing of your procedures for syntax errors as I illustrated when we implemented `ladd` together in the lab. If you "work from success", you greatly decrease the chances that you will have to spend a long time searching for one or more bugs in your code.

5. Take seriously the suggestion to always code a script to test your procedures at the same time or, better, *before* you start working on the procedures themselves. The test script need not be exhaustive or complete to begin with, but the purely mechanical tasks of (a) beginning to edit the test code, (b) entering the appropriate *read* command to input the procedure definitions and, (c) most importantly, defining some test data and at least a couple of sample invocations (calls) of the procedure(s), will get you going and ensure that you have at least a general idea of what it is you need to do.

6. *Always* enable tracing of your procedures until you are confident that they are correct. Follow the recommended practice for placement of the trace statements that was discussed in the lab. It is fine to leave them "commented-out" in your source files *and they* should *be commented-out in your submitted files if they haven't been expunged.*

7. *Always* end each `Maple` statement that you type with a semi-colon. Make it your goal to rid you worksheet of those pesky

   ```
   Warning, inserted missing semicolon at end of statement
   ```

   messages. `Maple` helps you out with this when you're typing in a worksheet, but *not* when you're coding a procedure. Forgetting a semi-colon can result in *many* different error messages. If `Maple` were able to tell you "you need a semi-colon exactly here" it wouldn't need statement terminators, would it?

   As we have discussed, you can also use colons to terminate statements, but since they will suppress tracing output, I advise you *not* to use them until your code is debugged.

8. Since `Maple` error messages frequently make reference to specific line numbers in the source file that is processing, be sure that you have line numbers enabled in your text editor. If you don't know how to do this, use your browser to search for "line numbers" in the Maple programming lab notes to find instructions for `kate` and `gedit`.

9. Remember that an error message complaining about a syntax error on such-and-such a line does *not* mean that the error is precisely where the caret (^) is positioned, or even that the error is in the line referenced by the error message. What it *does* mean is that the error is on that line *or before*, so start at the caret and carefully work your *backwards* through the code looking for the grammatical inexactitude(s).

10. If you find yourself staring at the computer screen for more than a couple of minutes trying to find a bug, give your brain a break, or at least change its mode of operation. Get up out of your chair for a few seconds or read your code quietly to yourself (it often works, believe me!) or whatever else might help you take a fresh look at the problem.

11. **Very important:** Pay very, very close attention to the spelling (misspelling?) of variable names. Although I consider `Maple` to be a good language to use in an introduction to programming, especially for students in mathematics and mathematically-oriented scientific disciplines, it *does* have some shortcomings, or "gotchas", that can easily trip up novices and experts alike.

    This one still gets me all the time: Remember, *all* `Maple` names, with the exception of those that the language reserves, *automatically* have a value, which is their name: this is the *atomic* property. That's why when you type `foo;` or `x;` in a new worksheet, `Maple` regurgitates `foo` or `x`. Many if not most programming languages do *not* operate like this.

    For example, as we will shortly see (or perhaps by the time you read this, as you have already seen :-)), if we type in a name in MATLAB which has not been assigned a value, it tells you as much:

    ```
    MATLAB PROMPT >> x
    Undefined function or variable 'x'.
    ```

    Because `Maple` does *not* require a name to have a value other than its name—and if it did, its ability to "do mathematics" using a natural syntax would be crippled—bugs such as the one in the following code can be difficult to detect, particularly if you don't abide by the dictum that you should code incrementally. See if you can spot the defect before you read past the definition:

    ```
    lsum_bug := proc(l::list)
       local i, nl, thesum;
       nl := nops(l);
    ```

```
   thesum := 0;
   for i from 1 to n1 do
      thesum := thesum + l[i];
   end do:
   thesum;
end proc;
```

Stumped? There's a good probability that I would miss it if I hadn't concocted the example.

If I read the procedure into `Maple`, I get no complaints. If I ask to see its definition using `op`, `Maple` assures me that it's syntactically correct:

```
> op(lsum_bug);

proc(l::list)
local i, nl, thesum;
    nl := nops(l);
    thesum := 0;
    for i to n1 do thesum := thesum + v[j] end do;
    thesum
end proc
```

So far so good. Now let's test it:

```
> lsum_bug([1, 2, 3, 4]);

Error, (in lsum_bug) final value in for loop must be
   numeric or character
```

So, if you didn't spot the bug, go back and stare at the code, taking into account the error message. Do you see the problem now?

If you don't, or if your eyes are starting to glaze over at this point, let me put you out of your misery: In the `for` loop I mistyped the variable name `nl` that I defined to be the length of the list—I used a `1` (one) rather than an `l`. Since the value of `n1` (n-one) is simply its name, not a number, `Maple` complains appropriately when it tries to start executing the `for` loop.

Again, other programming languages would tell me immediately and explicitly that `nl` "wasn't defined" (when the procedure was executing) or "wasn't declared" (when the procedure definition was being read), but `Maple` doesn't, so you need to look out for this sort of thing.

Maple features you are allowed to use

With the exception of Problem 6, and unless otherwise instructed, you should *not* make use of features of `Maple` that we have not covered in the lectures and labs. Please do not subvert the purpose of the programming component of this assignment by looking for particular commands, packages etc. that will accomplish a requested task more directly, succinctly, elegantly etc., etc., than "doing it yourself" with the relatively few statements and commands we have studied.

If you want to use a statement that doesn't appear in the notes, and think you should ask me about it, well, thanks in advance for asking, but the answer is almost certainly "no" :-)

Finally, as mentioned above, when you are ready to "hand in" your procedure definitions, please ensure that all `trace` *statements have been removed from your code (or commented-out).*

OK ... *finally*, on to the programming!

Problem 3: Create the subdirectory `a3`. In that directory create two `Maple` source files as follows:

1. File name: `procs3`
   - Contents: Definitions of the three procedures described below.
2. File name: `tprocs3` (`Maple` script file)
   - Contents: Testing code of your design for all three of the procedures defined in `procs3`.

Important note concerning error testing:

Here and in Problem 4 it is up to you to devise and implement the testing script (`tprocs3` *here,* `tprocs4` *in Problem 4) that calls the procedures you define, using both valid and invalid input. I am not going to try to define* precisely *how should this be done (e.g. how many tests at a minimum you should have per procedure, etc.), not least since as computational scientists we must develop the ability to tell when—with a high degree of certainty—when our computer codes are working properly.*

However, you don't have to go overboard with your testing methodology and the significant bulk of the grade for these problems will be allocated for procedures that do the correct thing with valid input of our own design (i.e. myself and the TAs), since, after all, we can argue that in computational science, getting trustworthy results from our calculations is almost all that matters in the end.

Also note I have generated the sample usages and output using command-line `Maple` (`Maple`), so expect that you may see different-looking, but algebraically equivalent output when using `xmaple`.

Here then are the descriptions of the three procedures that you must code in the source file `procs3`. Observe that the `type` commands in the "Sample usage and output" sections serve to verify that the procedures are returning objects of the correct type.

Problem 3.1

1. Header: `V_powers := proc(x::numeric, n_v::integer)`
2. Function: Returns the vector V having values

   $$V_i = x^i, \quad i = 1, 2, \ldots n_V$$

   where x^i denotes x to the power i.
3. Required error-checking: None
4. Sample usage and output:

```
> V1 := V_powers(3.0, 4);
                      V1 := [3.0, 9.00, 27.000, 81.0000]

> type(V1, Array);
                                     true

> V2 := V_powers(1/2, 8);
            V2 := [1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/256]

> type(V2, Array);
                                     true
```

Problem 3.2

1. Header: `V_powers2 := proc(x::numeric, p_1::integer, p_2::integer)`
2. Function: Returns the vector V whose length, n_V, is defined implicitly by the values of the arguments, p_1 and p_2, and which has elements:

$$V_1 = x^{p_1},\ V_2 = x^{p_1+1},\ V_3 = x^{p_1+2},\ ...,\ V_{n_V} = x^{p_2}$$

   Note, dependent on the values of p_1 and p_2, the vector may have as few as one element.
3. Required error-checking: The arguments must satisfy $p_2 \geq p_1$. If they don't, the procedure must call `error` so as to reproduce the message seen below.
4. Sample usage and output:

```
> V1 := V_powers2(3.0, 2, 6);
             V1 := [9.00, 27.000, 81.0000, 243.00000, 729.000000]

> type(V1, Array);
                                      true

> V2 := V_powers2(1/2, -7, 2);
                 V2 := [128, 64, 32, 16, 8, 4, 2, 1, 1/2, 1/4]

> type(V2, Array);
                                      true

> V3 := V_powers2(evalf(Pi), 4, 4);
                               V3 := [97.40909108]

> type(V3, Array);
                                      true

> V4 := V_powers2(3.0, 8, 1);
Error, (in V_powers2) p_2 must be >= p_1
```

Note that the procedure should *not* force the elements of the returned vector to be floating point values.

Problem 3.3

1. Header: `M_identity_rotate := proc(n::integer)`
2. Function: Returns an $n \times n$ matrix M that has 1's along a diagonal running from the *top right corner* to *bottom left corner* of the matrix, and 0's elsewhere. I've chosen the name `M_identity_rotate` because the resulting matrix is what would obtain by visualizing the usual $n \times n$ identity matrix living in a plane and then rotating it by 90° about an axis perpendicular to the plane.
3. Required error-checking: None
4. Sample usage and output:

```
M1 := M_identity_rotate(2);
                                      [0    1]
                                M1 := [      ]
                                      [1    0]

> type(M1, Array);
                                      true
```

```
> M2 := M_identity_rotate(6);
                              [0    0    0    0    0    1]
                              [                           ]
                              [0    0    0    0    1    0]
                              [                           ]
                              [0    0    0    1    0    0]
                        M2 := [                           ]
                              [0    0    1    0    0    0]
                              [                           ]
                              [0    1    0    0    0    0]
                              [                           ]
                              [1    0    0    0    0    0]

> type(M2, Array);
                                        true

> M3 := M_identity_rotate(1);
                                     M3 := [1]

> type(M3, Array);
                                        true
```

Additional resources

- Source files defining sample procedures and a testing script that may be of use in solving this problem (also see the procedures covered in the lab notes).
  1. `/home/phys210/hw2/prob3/samples3`
  2. `/home/phys210/hw2/prob3/tsamples3`

File inventory

1. `procs3`
2. `tprocs3`

Problem 4:

First note that I accidentally skipped explicit coverage of `NULL` (which we can view as the "empty sequence") and the *empty list*, `[]`, although (a) I did mention them in the lab, and (b) we did use `NULL` when we coded `ladd`. I've added a section to the `Maple` Programming Lab notes that you should read through if you are unclear on those concepts.

Create the subdirectory `a4`. In that directory create two `Maple` source files as follows:

1. File name: `procs4`
   - Contents: Definitions of the 3 procedures described below.
2. File name: `tprocs4` (`Maple` script file)
   - Contents: Testing code of your design for all 3 of the procedures defined in `procs4`

Problem 4.1

1. Header: `lreverse := proc(l::list)`
2. Function: Returns a list whose elements are those of `l`, but in reverse order.
3. Required error-checking: None
4. Sample usage and output:

```
> l1 := lreverse([a, b, x + y, z]);
                          l1 := [z, x + y, b, a]

> type(l1, list);
                                  true

> l2 := lreverse([[0,1], [1, 2, 3], x, y, {1, 2}]);
                  l2 := [{1, 2}, y, x, [1, 2, 3], [0, 1]]

> type(l2, list);
                                  true

> l3 := lreverse([]);
                                 l3 := []

> type(l3, list);
                                  true
```

Problem 4.2

1. Header: `ltail := proc(l::list)`
2. Function: Returns the list known as the "tail" of the input list, `l`, defined as follows. If the length L of the list is 0 or 1, then the tail of the list is the empty list, otherwise the tail of the list is the length-$(L-1)$ list consisting of all the elements in `l` *except the first* in the same order as they occur in `l`.
3. Required error-checking: None, but your procedure must handle empty list input correctly.
4. Sample usage and output:

```
> l1 := ltail( [[0,1], [1, 2, 3], x, y, {1, 2}] );
                       l1 := [[1, 2, 3], x, y, {1, 2}]

> type(l1, list);
                                  true

> l2 := ltail(ltail(ltail([1, 2, 3, 4])));
```

```
                                    l2 := [4]

> type(l2, list);
                                      true

> l3 := ltail( [1] );
                                    l3 := []

> type(l3, list);
                                      true

> l4 := ltail( [] );
                                    l4 := []

> type(l4, list);
                                      true
```

Problem 4.3 ($\star$)

1. Header: `lreplicate := proc(l::list, m::posint)`
2. Function: Given an input list `l`, returns a list of the same length whose i-th element is a `list` of length m which is the i-the element of `l` replicated m times.
3. Required error-checking: If `l` is the empty list, the procedure must call `error` so as to reproduce the message seen below.
4. Sample usage and output:

```
> l1 := lreplicate([1, 2, 3], 4);
              l1 := [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]]

> type(l1, list);
                                      true

> l2 := lreplicate([a, [b, c], 4, [e,f,g]], 2);
       l2 := [[a, a], [[b, c], [b, c]], [4, 4], [[e, f, g], [e, f, g]]]

> type(l2, list);
                                      true

> l3 := lreplicate([1], 1);
                                   l3 := [[1]]

> type(l3, list);
                                      true

> l4 := lreplicate([A], 2);
                                 l4 := [[A, A]]

> type(l4, list);
                                      true

> l5:= lreplicate([], 5);
Error, (in lreplicate) Input list l can not be empty.

> type(l5, list);
                                      false
```

Additional resources

- Source files defining sample procedures and a testing script that may be of use in solving this problem (also see the procedures covered in the lab notes).
    1. `/home/phys210/hw2/prob4/samples4`
    2. `/home/phys210/hw2/prob4/tsamples4`

File inventory

1. `procs4`
2. `tprocs4`

Problem 5 ($\star\star$): *The Chaos Game*

5.1 Mathematical specification

Consider the x-y plane and let

$$P^a \equiv (x^a, y^a) \quad a = 1, 2, 3 \tag{1}$$

be the vertices of an equilateral triangle that lies in that plane and which is inscribed in the unit circle defined by $x^2 + y^2 = 1$ (i.e. the circle with unit radius whose center is located at the origin $(0, 0)$). Specifically, the coordinates of the P^a are as follows:

$$P^1 = \left(P^1_x, P^1_y\right) \equiv \left(\cos\left(\frac{\pi}{2}\right), \sin\left(\frac{\pi}{2}\right)\right) \tag{2}$$

$$P^2 = \left(P^2_x, P^2_y\right) \equiv \left(\cos\left(\frac{7\pi}{6}\right), \sin\left(\frac{7\pi}{6}\right)\right) \tag{3}$$

$$P^3 = \left(P^3_x, P^3_y\right) \equiv \left(\cos\left(\frac{11\pi}{6}\right), \sin\left(\frac{11\pi}{6}\right)\right) \tag{4}$$

where all angles here and below are measured in radians. In case you are not familiar with the notation, observe that $\equiv$ means "is identically equal to" or "is equivalent to".

In general, one starts the chaos game by choosing *at random*, some arbitrary point, P^4 on the xy plane that does not coincide with one of the triangle's vertices, but to make life slightly simpler for you, we will choose our "random" point so that it has specific coordinates:

$$P^4 = \left(P^4_x, P^4_y\right) \equiv \left(\frac{5}{4}\cos\left(\frac{29}{17}\right), \frac{5}{4}\sin\left(\frac{29}{17}\right)\right)$$

The game proceeds by generating a further n_{steps} points

$$P^{j+4}, \quad j = 1, 2, \ldots, n_{\text{steps}} \quad = \quad P^5, P^6, P^7, \ldots, P^{n_{\text{steps}}+4},$$

where n_{steps} is the number of steps we play. Note that each of the additional points is also defined by an (x, y) pair of coordinates:

$$P^j \equiv \left(P^j_x,\, P^j_y\right)$$

Given the current point P^i, with $i \geq 4$, the next point P^{i+1} is determined using the following sequence:

1. One of the three vertices, P^a, $a = 1, 2, 3$, of the triangle is chosen *randomly*.
2. We consider the line segment $\overline{P^a P^i}$ between the current point, P^i, and the randomly chosen triangle vertex, P^a (a will always be one of 1, 2 or 3).
3. We bisect $\overline{P^a P^i}$ to define a new point, P^{i+1}, that thus lies exactly at the midpoint of $\overline{P^a P^i}$.
4. The new point P^{i+1} becomes the current point for the next step of the game.

If one plots all n_P of the points P^i that are generated—which we will take to include the triangle vertices and the "randomly" chosen initial point, P^4—an interesting pattern appears for large numbers of steps.

In the context of the discussion in the *Notation and translation ...* section above, we can consider P^i to be a *generalized* vector, each of whose elements is an (x, y) pair of coordinates, rather than a single number. [1] Then the length n_P of P is given by

$$n_P = n_{\text{steps}} + 4\,.$$

5.2 Implementation

Create the subdirectory `a5`. Copy the following files from `/home/phys210/hw2/prob5` to that directory.

```
runchaos_1
runchaos_2
runchaos_final
```

[1] In fact—and you may be thinking about this yourself—it seems natural to represent P as a *two*-dimensional array, with one index i labelling the coordinate direction, i.e. $i = 1$ means x and $i = 2$ means y, and the other labelling the point. However, for reasons that I won't go into here, I've chosen *not* to formulate the problem using 2D arrays

`/home/phys210/hw2/prob5` also contains the resource files for this problem:

```
samples5
tsamples5
```

and it is fine if you copy those to `a5` as well.

Continuing to work in `a5`, create a `Maple` source file, `chaos`, whose first two lines *must be*:

```
with(RandomTools):
roll := rand(1..3):
```

(You can terminate the statements with semi-colons rather than colons should you wish.)

Be very careful to duplicate these two lines exactly. If you don't, then the `roll` *procedure, which is to generate a random integer between 1 and 3 inclusive, will not be defined, and you* must *use* `roll` *to implement* `chaos`.

Following those two statements add `Maple` code that implements the procedure `chaos` whose header and precise behaviour are defined below.

The files `runchaos_1`, `runchaos_1` and `runchaos_final` are all `Maple` scripts that contain the command

```
read chaos;
```

Thus, when you read one of them into an `xmaple` or `Maple` session, e.g.

```
> read runchaos_1;
```

it will, in turn, read your definition of `chaos`. The script will then call your implementation of the procedure with certain arguments and plot the results that your code produces. Of course, you can see the details of what any of the scripts does simply by looking at its contents.

Independent of the programming language, a script (or program) such as `runchaos_1` `runchaos_2` or `runchaos_final` whose main purpose is to call some procedure (or another program) with one or more argument sets and display, analyze etc. the results, is often called a *driver*, since it "drives" or "controls" the routine which is actually doing the work. I will use that terminology here, and throughout the rest of the course. Additionally, in the context of `Maple` when I say "run driver `X`", I mean execute the command

```
> read X;
```

with the assumption that the source file `X` will contain a `read` command that inputs the appropriate procedure definition(s).

The driver scripts are designed so that by executing them, modifying your implementation, re-executing them, re-modifying your code ... there is a very good chance that you will know when your implementation is correct.

That said, you are free to develop your own script or scripts to test your code—this might, for example, involve copying and modifying one of the scripts that I provide. If you do this, you can leave the additional files that you create in the solution directory. However, the only source code file that is *required* to complete the question is `chaos`.

The procedure `chaos` must have the header

```
chaos := proc(nsteps::posint)
```

where `nsteps` is the number of steps to play as defined above.

The procedure `chaos` must return a two-element *list*

$$[X, Y] \ ,$$

where X and Y are 1D arrays (vectors) of length $n_{\text{steps}} + 4$, with elements

$$X^i \equiv P_x^i \, ,$$

$$Y^i \equiv P_y^i \, .$$

and where the P^i are the points generated during the game (again, including the three fixed vertices of the triangle and the "random" initial point).

The rules of the game stipulate that you have to randomly choose one of the triangle vertices at each step. Your implementation *must* use `roll` to do this.

All of the values returned by `chaos`; i.e. all of the elements of the vectors X and Y should be floating point numbers (floats).

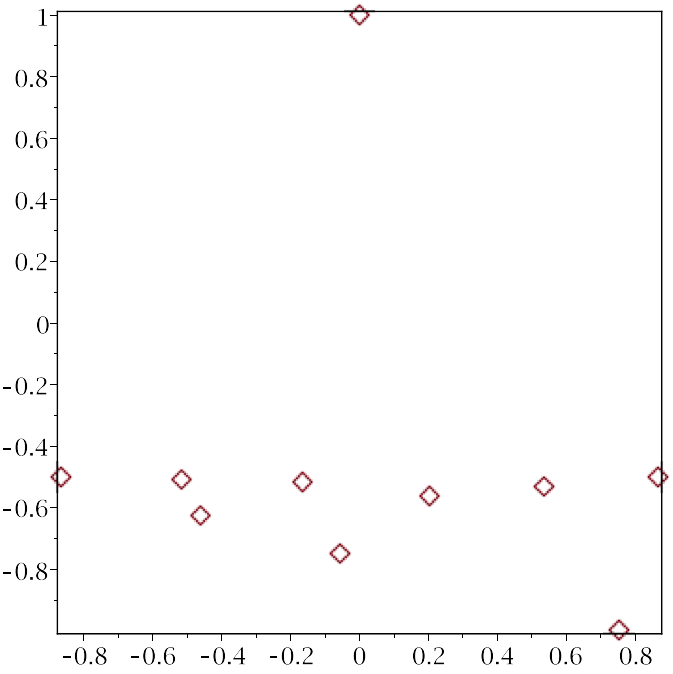
5.3 Sample output from a working implementation

Here is a screenshot of an `xmaple` session showing sample output from the invocation `chaos(6)`, including plotting of the points returned by the procedure. Note that `pts` is a 2-element list: each of the elements, `pts[1]` and `pts[2]` is a 10 element, one-dimensional array (vector). Thus `plot` knows how to plot the points when supplied with `pts[1]` and `pts[2]` as its first two arguments.

```
> read chaos;
> pts := chaos(6);
```

$$pts := \left[[-2.051033808\ 10^{-10}, -0.8660254034, 0.8660254040, 0.7532932956, \right.$$
$$-0.05636605390, -0.4611957286, 0.2024148377, 0.5342201210, -0.1659026412,$$
$$\left. -0.5159640225], [1., -0.5000000007, -0.4999999996, -0.9975215340, \right.$$
$$-0.7487607675, -0.6243803840, -0.5621901920, -0.5310950960,$$
$$\left. -0.5155475485, -0.5077737745] \right]$$

```
> plot(pts[1], pts[2], style=point, symbolsize=20, axes=boxed);
```



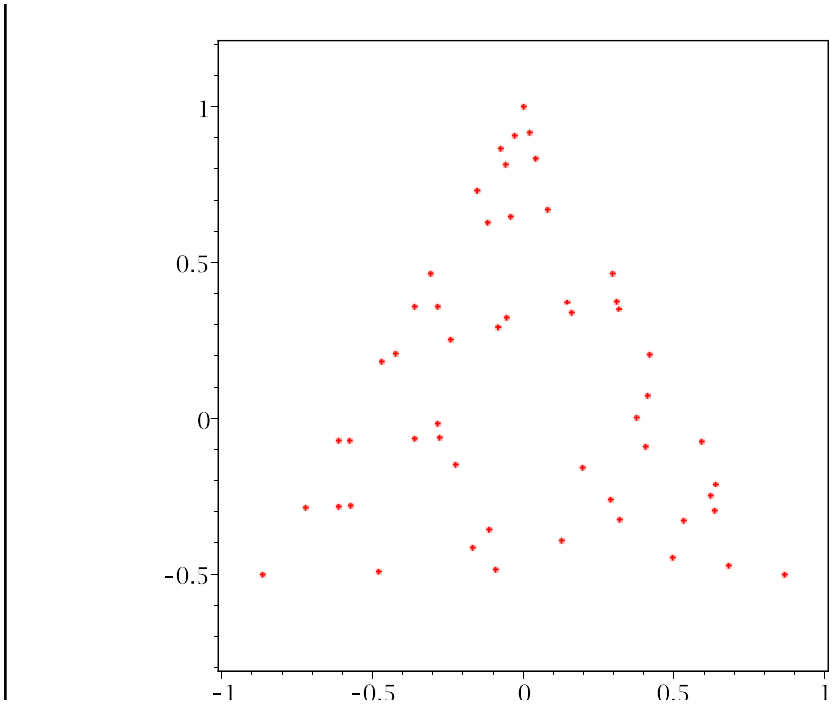
5.3 Evaluation of your implementation using the driver scripts

Run the drivers ("run"≡ "read the script file into your `Maple` session") in the order

```
runchaos_1
runchaos_2
runchaos_final
```

trying to ensure that the output from `runchaos_1` looks reasonable before moving on to `runchaos_2` and that the output from `runchaos_2` looks reasonable before moving on to `runchaos_final`.

The plot that is produced from `runchaos_1` should look *something* like this:



It is quite likely that yours won't look *exactly* like the above due to the random (stochastic) nature of the algorithm.

Once everything seems to be in order, consider the questions that appears just before `runchaos_final` finishes execution, and provide answers for them in a `README` file.

Congratulations, you're done!

Should you wish, you can also try the bonus question, Problem 6, which involves generalizing the game and investigating the resulting change in behaviour.

5.4 Additional resources

- Source files defining sample procedures and a testing script that may be of use in solving this problem (also see the procedures covered in the lab notes).
  1. `/home/phys210/hw2/prob5/samples5`
  2. `/home/phys210/hw2/prob5/tsamples5`

5.5 File inventory

1. `chaos`
2. `README`

Problem 6 ($\star\star\star$): *Strictly optional for bonus credit, and your own edification.*

Generalizing the Chaos Game

Generalize the game in one or more ways including, but not necessarily limited to:

1. Adding an option that controls the number of fixed vertices used (i.e. so that you can play with 4, 5, 6, ... fixed vertices, rather than just 3).
2. Adding an option that controls the placement of new points: new points should continue to be placed along a line segment connecting the previous point and the randomly chosen vertex, but need not be at the mid-point of the line segment.
3. Implementing the game in three dimensions.

If you *do* chose to generalize the game, be sure to create one or more *new* `Maple` files that defines one or more `Maple` procedures with names different than `chaos` as well as one or more drivers analogous to `runchaos_final` that demonstrate your extensions.

All of the new files should, naturally enough, reside in the subdirectory `a6` of `hw2` which, as usual, you will need to create.

Your original implementation of `chaos` must remain in `a5/chaos` so that when the graders evaluate it with the supplied drivers, everything works properly as specified above.

You are encouraged to use your customized procedures to make and save one of more "pretty pictures" in `JPEG` format, and add them along with suitable captions to your course web page.

Outputing plots in an image format is easy in `Maple`. You can "capture" the output of any `plot` command as a plot object, and then use `exportplot` to output that object in a variety of formats. Note that `exportplot` is part of the `plottools` package, which you will need to explicitly load using the

```
> with(plottools);
```

Better still, consider putting the above command in your ∼`/.mapleinit` file.

For example:

```
> with(plottools);
 annulus, arc, arrow, circle, cone, cuboid, curve, cutin, ...
 disk, dodecahedron, ellipse, ellipticArc, exportplot, ...
                          .
 tetrahedron, torus, transform, translate]


> myplot := plot([sin(x), cos(x), tan(x)], x = -2*Pi .. 2*Pi):

> # Display in xmaple session itself (same behaviour as I get
> # if I don't assign the return value of plot(...)

> print(myplot);

> # Output as JPEG-format image file "myplot.jpg" ... be
> # sure to use the "" to project the .
> exportplot("myplot.jpg", myplot);
```

Leave comments describing your generalization(s) in a `README`, including instructions for running your driver(s), and summarize what you discovered in terms of their effect on the results of the game, especially relative to the original. If you *do* post images of some of your results on your web page, make note of that fact in the `README` as well.