PHYS 410/555: Computational Physics Fall 2004 Homework 3
DUE: Tuesday, November 2, 10:00 AM Report bugs to choptuik@physics.ubc.ca

The following assignment involves writing and testing several simple Fortran 77 programs. Follow the instructions carefully and please do all compilation and execution on your `lnx` *account, or ensure that all programs compile and execute on the* `lnx` *machines. All files required by the assignment must reside in the correct places on your* `lnx` *account for the homework to be considered complete.*

Important: *Your solutions to each and every one of the problems below MUST include a* `Makefile` *(or* `makefile`*) that defines, at a minimum, targets for the executable or executables required by the problem, and* `clean`*, where the* `clean` *target, as usual, removes the executables and object-code files. Furthermore, your makefiles MUST use the macros* `F77, F77FLAGS, F77CFLAGS, F77LFLAGS` *as discussed in class (i.e. you must define "portable" makefiles).*

Problem 1: In a directory `~/hw3/a1` on your `lnx` account, create Fortran source files `pgm1.f` and `pgm2.f` that are to be compiled into executables `pgm1` and `pgm2` with behaviour as described below.

Important: Use `real*8` arithmetic and "free-format" I/O (i.e. `read(*,*)` and `write(*,*)`) *here and throughout the homework.* In particular, in accordance with the "Unix philosophy", discussed in class, of keeping output from programs as "machine-readable" as possible, do *NOT*, put "descriptive" information such as "`f = `" in your output.

- `pgm1`: reads x, y pairs (2 `real*8` numbers per line) from standard input, computes and writes $f(x, y)$ to standard output, where $f(x, y)$ is defined as follows:

$$f(x,y) = \cos^2(x+6y)\left(\frac{2+e^{-\frac{x^2+y^2}{2}}}{2-e^{-\frac{x^2+y^2}{2}}}\right)$$

- `pgm2`: reads x, y, z triples (3 `real*8` numbers per line) from standard input, computes and writes $g(x, y, z)$ to standard output, where $g(x, y, z)$ is defined as follows:

$$g(x,y,z) = \begin{cases} \sqrt{x^4+y^4+z^2} & \text{if } x<0 \\ \frac{\pi}{4} & \text{if } x=0 \\ \cos(x+y+z)\sin(\min(x,y))\tan(\max(y,z)) & \text{if } x>0 \end{cases}$$

Minimal commenting of these programs is acceptable, but be sure to carefully test them. You may find it helpful to use `Maple` for testing purposes. (Recall that the `Maple` procedure `evalf()` evaluates an expression using floating-point arithmetic.)

Your programs should detect and ignore invalid input in the same fashion as the `mysum` example discussed in class. Should you wish to print messages if invalid input is encountered, be sure to direct the messages to standard *error* (unit 0). Your programs will be tested with input of my own design.

As there has been confusion concerning this in the past, be sure that both of your programs accept 0 or more lines of input—i.e. do not *assume that there will only be a* single *line of input.*

Finally, note that there is a `Fortran sqrt` function—see Appendix A of the *Professional Programmer's Guide to Fortran 77*, by Clive Page (available on-line via the Fortran Programming notes page) for a full list of the `Fortran 77` intrinsic (built-in) functions.

Problem 2: In a directory ~/hw3/a2 create source files `dvvfrom.f` and `dvvto.f` that define Fortran `subroutines` with the following headers:

```
      subroutine dvvfrom(fname,v1,v2,n,maxn)
         implicit        none
         character*(*)   fname
         integer         n,            maxn
         real*8          v1(maxn),     v2(maxn)


      subroutine dvvto(fname,v1,v2,n)
         implicit        none
         character*(*)   fname
         integer         n
         real*8          v1(n),        v2(n)
```

These routines are to behave completely analogously to the routines `dvfrom` and `dvto` covered in the class notes except that they are to read and write *pairs* of `real*8` vectors (one-dimensional arrays) from and to a file with name `fname`. Thus the format of the input and output is:

```
v1(1)    v2(1)
v1(2)    v2(2)
v1(3)    v2(3)
      .
      .
      .
```

If `fname .eq. '-'`, the routines should use standard input or standard output as appropriate. In the same directory, write test programs `tdvvfrom.f` and `tdvvto.f` and create executables `tdvvfrom` and `tdvvto` that test your two routines. The arguments to `tdvvfrom` and `tdvvto` should be identical to the programs `tdvfrom` and `tdvto`, respectively, which were covered in class. Specifically, the usage of the test programs should be:

```
% tdvvfrom <file name>
% tdvvto   <file name> <n>
```

`tdvvfrom` should report on standard error the length of the vectors that were read (i.e. how many pairs were read); `tdvvto` need not report anything under normal execution. Both testing programs should be adequately documented and must perform argument processing—this includes detection of insufficient or invalid arguments, in which case a usage message should be printed to standard error. Use the system routines `iargc` and `getarg` as well as `i4arg` from the `p410f` library discussed in class for argument parsing. You should also use the integer function `indlnb` from the `p410f` library to ensure that you don't create file names with trailing spaces. Your test programs, as well as your `dvvfrom` and `dvvto` routines themselves, will be evaluated with my own input and testing programs.

Problem 3: *The Mandelbrot Set*
As some of you will know, the Mandelbrot set[1], is a region of the (2-dimensional) complex plane, that has a boundary with very interesting properties such as self-similarity (scale invariance) and fractional dimensionality (i.e. the boundary is a so-called *fractal*). Your task in this problem is to implement a straightforward, "brute force" algorithm, as the `Fortran` program `mandelbrot`, which can then be used to investigate the structure of the Mandelbrot set and, in particular its fractal boundary. You must also implement a front end `sh`-script, `Mandelbrot`, that runs `mandelbrot` for several parameter sets, and then uses `gnuplot` to generate (Postscript) plots of the results.

[1] Named after Benoit Mandelbrot, who studied the properties of such sets in detail, and was substantially responsible for bringing fractals into the general scientific and public consciousness.

The Mandelbrot set is defined by considering a specific mapping of the complex plane into itself, i.e. by considering a complex valued function, $f(z)$, of a single complex unknown, z. The particular function that will be considered here is

$$f(z) = z^2 - \frac{3}{4}\,. \tag{3.1}$$

Given this function (mapping), we can consider its repeated application to any point in the complex plane. That is, we choose *some* specific complex number, z_0, then generate a sequence of iterates

$$z_1, z_2, z_3 \cdots z_\infty\,,$$

such that

$$z_{n+1} = f\left(z_n\right) = z_n^2 - \frac{3}{4}\,.$$

Let us denote the n-fold iterated application of $f(z)$ as $f^{(n)}(z)$, so that we have

$$z_n = f^{(n)}(z_0)\,.$$

Then an arbitrary point, z, in the complex plane is defined to be in the Mandelbrot set if and only if

$$\lim_{n\to\infty}\left|f^{(n)}(z)\right| = \infty\,.$$

where $|z|$ is the *modulus* of z, i.e. $|z| = \sqrt{zz^\star}$. In words then, if in the limit of infinite iteration, (3.1) maps a point z to complex infinity, then the point is in the Mandelbrot set.

Given this introduction, you are to create in `~/hw3/a3` a Fortran 77 source file `mandelbrot.f` with corresponding executable `mandelbrot` that has the following usage

```
usage: mandelbrot <re0> <re1> <nre> <im0> <im1> <nim> <nmax>
```
(3.2)

Note that `mandelbrot` is thus to accept precisely 7 command line arguments whose types and interpretations will shortly be defined, as will the output from the program.

Preamble: It should be clear from the *definition* of the Mandelbrot set, appealing as it does to an infinitely iterated function, that you will have to introduce some approximations and/or simplifying assumptions into your implementation of `mandelbrot`. Specifically, then, you are to adopt the following operational definition for membership in the Mandelbrot set. Consider some suitably large modulus (distance from the origin of the complex plane), R, and some fixed but large (or largeish) number of iterations $n_{\rm max}$. Then a point z will be deemed to be in the Mandelbrot set if

$$\left|f^{(n)}(z)\right| \ge R \quad \text{for some } n \le n_{\rm max}\,.$$

Thus, in the simultaneous limit $n_{\rm max} \to \infty$, $R \to \infty$ we recover from the above the true definition of membership in the set.

Command line arguments: We now return to a discussion of the command line arguments to `mandelbrot`. The first six arguments define a uniform (discrete) lattice (aka mesh, lattice) of values $z_{j,k}$ in the complex plane as follows

$$z_{j,k} = \left[\mathrm{re}_0 + (j-1)\Delta\mathrm{re}\right] + i\left[\mathrm{im}_0 + (k-1)\Delta\mathrm{im}\right], \quad j = 1, 2, \ldots n_{\rm re}, \quad k = 1, 2, \ldots n_{\rm im} \tag{3.3}$$

where $i^2 = -1$, re_0, re_1, im_0, im_1 are real values satisfying $\mathrm{re}_0 < \mathrm{re}_1$, $\mathrm{im}_0 < \mathrm{im}_1$ that define a rectangular region in the complex plane

$$\mathrm{re}_0 \le \mathrm{Re}(z) \le \mathrm{re}_1, \quad \mathrm{im}_0 \le \mathrm{Im}(z) \le \mathrm{im}_1$$

and the integers $n_{\rm re}$ and $n_{\rm im}$ define how many grid points there are in the real and imaginary directions respectively. (Note that I have used the $[\cdots]$ brackets in (3.3) to highlight the real and imaginary parts of $z_{j,k}$.) $\Delta\mathrm{re}$ and $\Delta\mathrm{im}$ are to be defined so that

$$z_{n_{\rm re},n_{\rm im}} = \mathrm{re}_1 + i\,\mathrm{im}_1\,.$$

The correspondence between quantities defined above and the command line arguments are as follows

$$\begin{aligned}
\texttt{re0} &\rightarrow \mathrm{re}_0 \\
\texttt{re1} &\rightarrow \mathrm{re}_1 \\
\texttt{nre} &\rightarrow n_{\mathrm{re}} \\
\texttt{im0} &\rightarrow \mathrm{im}_0 \\
\texttt{im1} &\rightarrow \mathrm{im}_1 \\
\texttt{nim} &\rightarrow n_{\mathrm{im}} \\
\texttt{nmax} &\rightarrow n_{\mathrm{max}}
\end{aligned}$$

Use the functions `r8arg` and `i4arg` discussed in class and in the Fortran notes to parse the arguments from the command line. You can use code such as

```
c     Declare the type of r8arg
      real*8       r8arg

c     Define a real*8 value that you trust/hope the "user" will never use
      real*8       r8_never
      parameter  ( r8_never = -1.0d60 )

c     Declare a temporary for the argument to be parsed
      real*8        thingy
                .
                .
                .
c     Parse the 4th command line argument as a real
      thingy = r8arg(4,r8_never)

c     If r8_never is returned, argument either isn't there, or can't be parsed as a real
      if( thingy .eq. r8_never ) go to 900
                .
                .
                .
900   continue
         write(0,*) 'usage: ...'
      end
```

to deal with missing or non-parseable command line arguments. In addition to that error checking, your program must also ensure that the following constraints are satisfied.

1. $\mathrm{re}_0 < \mathrm{re}_1$
2. $\mathrm{im}_0 < \mathrm{im}_1$
3. $n_{\mathrm{re}} > 1$
4. $n_{\mathrm{im}} > 1$
5. $10 \le n_{\mathrm{max}} \le 1000$

Other Problem Parameters: Your implementation of `mandelbrot` should use a "hardcoded" value of $R = 1 \times 10^{10}$ (i.e. the modulus at which we deem that the mapping is taking the input complex number to complex-∞ is to be fixed at 10^{10}).

Program Output: Given an invocation of the form (3.2), `mandelbrot` must produce the following, *and only the following*, on standard output

```
         .
         .
         .
   rez_jk     imz_jk
         .
         .
         .
```

where

$$\begin{aligned} \texttt{rez\_jk} &\rightarrow \mathrm{Re}[z_{j,k}] \\ \texttt{imz\_jk} &\rightarrow \mathrm{Im}[z_{j,k}] \end{aligned}$$

and the indices $[j, k]$ run over all values such that $z_{j,k}$ has been deemed to be in the Mandelbrot set.

Thus, for example, sample output from the instructor's implementation of `mandelbrot` starts as follows

```
% mandelbrot -1.0 2.0 101 -1.5 1.5 101 100
  -1.00000000000000       -1.50000000000000
 -0.970000000000000       -1.50000000000000
 -0.940000000000000       -1.50000000000000
 -0.910000000000000       -1.50000000000000
                    .
                    .
                    .
```

Any additional output that your program generates (such as diagnostic or tracing output) should be output to *standard error* (logical unit 0).

Visualization of Output: Use `gnuplot` (or some other *equivalent* plotting package available on `Linux` systems[2]) to plot the output from `mandelbrot`. *Note that in doing the plotting it is essential to convince* `gnuplot` *to use a plotting symbol that is sufficiently small that the structure of the output is not obscured by the symbols* per se. (*Hint*: Try `help plot with`). If as you proceed with your solution you remain unsure of what this means, or you cannot get `gnuplot` to do as you wish, ask for further explanation/assistance.

Numerical Experiments: If possible, (in particular, if you have the time), once you are reasonably confident that you have implemented `mandelbrot` correctly, you should try to run some numerical experiments to see if you can "discover" the interesting features of the boundary of the Mandelbrot set mentioned above. *Ideally, you will do this before reading the rest of the problem specification, or, if you must read through the rest of the question first, at least try to ignore/forget any numbers that you encounter!* Report your findings, if any, briefly in `README`.

Front-end Script: Write a `sh`-script called `Mandelbrot` that serves as a "front-end" to `mandelbrot` and that has usage

```
usage: Mandelbrot <res>

       <res> = number of discrete points in Re/Im directions
```

`Mandelbrot` is to execute the following invocations of `mandelbrot` (assuming that the first command line argument to the script has been parsed into the shell variable `res`):

```
mandelbrot -1.0   2.0   $res -1.5   1.5   $res 100
mandelbrot -0.43  0.31  $res  0.24  0.98  $res 100
mandelbrot -0.140 0.045 $res  0.544 0.729 $res 100
```

[2]Check with the instructor if you are unsure of the "equivalence" of a given plotting package to `gnuplot`

The script should create Postscript files containing `gnuplot` plots of the output from the above invocations. The plot files should be named

```
lev0-${res}.ps
lev1-${res}.ps
lev2-${res}.ps
```

respectively, where again, it is assumed that the first command line argument to `Mandelbrot` has been parsed into the shell variable `res`.

Ensure that the *aspect ratio* of your plots is unity, so that the plotting area is *square*. (*Hint:* Try `help set size`.)

`Mandelbrot` can do minimal error-checking of its command-line arguments. It should detect the wrong number of arguments, and exit with a usage message in that case. However, if there *is* a single argument supplied, *it can assume that the argument can be parsed as an integer.*

Required Output: Once you are sure that `mandelbrot` and `Mandelbrot` are both working properly (including usage messages and handling of wrong numbers/types of arguments as described above), execute

```
% Mandelbrot 501
```

in the solution directory, ensure that the plots

```
lev0-501.ps
lev1-501.ps
lev2-501.ps
```

exist and look as they should (by this time, if the problem is going well for you, you will know what they *should* look like). Relax and/or celebrate. You're essentially done the homework.

Summary of Required Files: Upon completion of this problem your solution directory should contain (at a minimum), files as follows:

```
% ls -FC
Makefile     lev0-501.ps  lev2-501.ps  mandelbrot.f
Mandelbrot*  lev1-501.ps  mandelbrot*  mandelbrot.o
```

Your solution may also involve a `REAMDE` file, and you are free to leave other files that you have created in-place.

Other Hints:

- See `democomplex.f` available via the *Miscellaneous* section of the course Software page for tips on complex arithmetic in `Fortran`.
- Consult the instructor if anything in the problem specification is unclear, or if you are having undue problems with any aspect of this question.

WARNING!!: Improper implementations of `mandelbrot` and/or overly large values of `nre`, `nim` (a 501×501 mesh is arguably large enough) and/or improper plotting techniques may lead to the generation of *very large data and/or plot files.* Beware of exhuasting disk space on the `lnx` machines (learn how to use `du` to generate summaries of disk usage if necessary), and avoid *printing* very large files (i.e. in excess of 2-3 megabytes) if at all possible, since you'll tend to tie up the printer.

Problem 4: (for 555 credit, OPTIONAL for 410 students)
In a directory `~/hw3/a4` create a `Fortran 77` source file `rw2d.f` and corresponding executable `rw2d`, which simulates and analyzes random walks of a particle on a two dimensional integer lattice. Specifically, if at step n the current position of the particle performing the walk is $(x, y) = (i, j)$, where i and j are integers, then, with equal probability (i.e. $p = 1/4$), the position at step $n + 1$ is either $(i + 1, j)$, $(i - 1, j)$, $(i, j + 1)$ or $(i, j - 1)$. `rw2d` should accept two integer arguments as illustrated in this usage message:

```
usage: rw2d <nsteps> <nwalks>
```

where `<nsteps>` is the number of steps to take per walk, and `<nwalks>` is the number of separate walks to be generated. Once again, note that you can use the `i4arg` function from the `p410f` library to parse the command-line arguments. For each walk, start the particle at $(0, 0)$ and use `real*8` values for the particle coordinates, even though the coordinates are restricted to integer values. When your program is finished simulating the random walks, it must write on standard output the average of the distance-squared of the particle from its starting point (i.e. compute $\langle r^2 \rangle$ using a simple average to compute the expectation value, and average over all walks) after each step, as a function of step number. Specifically, the output should consist of two numbers per line, generated using code such as the following:

```
do i = 1 , nsteps
   write(*,*)  i,   rsqavg(i)
end do
```

Document and test your program thoroughly, then make runs of 10, 100, and 1000, 5000-step walks, and save the output in files `out10`, `out100` and `out1000` respectively. Use `gnuplot`, or another instructor-approved plotting package (contact me if you're not sure whether a given package is instructor-approved!) to produce a *single* plot of average-distance-squared versus step number for all three data-sets. Save a postscript version of the plot in `distance.ps`. What can you say about the behaviour of $\langle r^2 \rangle$ versus step number as the number of walks gets large? Answer in a file called `README`. Note that you can use the `drand48` function defined in the `p410f` library, and discussed in class, in order to complete this problem.