**PHYS 410/555: Computational Physics Fall 2000 Homework 3**
**DUE: Thursday, October 12, 10:30 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* `sgi1.physics.ubc.ca`*. All files required by the assignment must reside in the correct places on your* `sgi1` *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`*) which 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 `sgi1` account, create Fortran source files `pgm1.f` and `pgm2.f` which 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:

$$f(x,y) = \sin(3xy)\left(\frac{1.0 + e^{-\frac{x^2+y^2}{2}}}{1.0 - e^{-\frac{x^2+y^2}{2}}}\right)$$

- `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:

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

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). I will evaluate your programs using my own input.

**Problem 2:** In a directory `~/hw3/a2` create source files `dvvfrom.f` and `dvvto.f` which 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` which 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 which 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. I will test your test programs, as well as your `dvvfrom` and `dvvto` routines themselves, with my own input and testing programs.

**Problem 3:** *The Chaos Game*
Let

$$V_i \equiv (\tilde{x}_i, \tilde{y}_i) \qquad i = 1, 2, 3$$

be the vertices of an equilateral triangle with side length 1 lying in the $xy$ plane. The orientation and position of the triangle in the plane is not important but, for concreteness, you may wish to "center" the triangle at the origin $(x, y) = (0, 0)$. The game is initialized by choosing, *at random*, some point in the $xy$ plane, such that the point lies outside of the triangle. Call this randomly chosen location the *active point*. One then *randomly* chooses one of the three triangle vertices, $V_i$, draws an imaginary line between the vertex and the active point, bisects the line and makes the point of bisection the active point. This last sequence of randomly choosing a vertex, bisecting the line between it and the active point and updating the active point with the point of bisection is then repeated some specified number of times. If one plots all the active points which are generated, an interesting pattern appears after a sufficiently large number of iterations.

In a directory `~/hw3/a3` on your `sgi1` account write a well-documented Fortran program called `chaos2d` which implements this algorithm. Note that, as discussed in class, on `sgi1` (and other SGI machines), the built in `real*8` function `rand` can be used to generate a random number between 0 and 1—it is up to you to figure out how to use `rand` to generate the particular randomness required by the algorithm. Be sure that you declare the function properly in the main program, and note that since it has no arguments, it should be invoked with an empty argument list:

```
      real*8      rand

      write(*,*) 'This is a pseudo-random number ', rand()
```

Your program must take a single integer argument which specifies the number of iterations to perform, and must echo a usage message to standard error if the argument is missing, or invalid, e.g.

```
 usage: chaos2d <nsteps>
```

Again, I suggest that you use the `i4arg` function discussed in class and available in the `p410f` library to parse the argument. Your program must write to standard output the $x$ and $y$ coordinates of all of the active points which are generated (including the original randomly chosen one). Specifically, use a `write` statement like the following

```
write(*,*) xactive, yactive
```

in your iteration loop, so that the coordinate pairs appear one per line on standard output. You may optionally output (also to standard output) the $(x, y)$ coordinates of the three triangle vertices $V_i$. If you choose to do so, output them *once only*, presumably at the beginning of the game. Finally, although the initial active point is to be chosen randomly, ensure that its distance from the center of the triangle does not exceed 2.5 (recall that the sides of the triangle are of length 1.0). Run your program for 10, 100, 1000 and 10000 iterations and redirect the output to files `out10, out100, out1000,` and `out10000` respectively. Use `gnuplot` to plot the points defined by each of the output files and save the plots as postscript files `out10.ps, out100.ps, out1000.ps,` and `out10000.ps`. When you make your plots, you should use an aspect ratio of 1 (same range of $x$ as $y$) and should not be afraid to clip a *few* points in order to focus on the region of interest (i.e. the base triangle and its interior). Also, be sure to use an appropriate "style" in `gnuplot` so that the interesting structure of the plot is visible (i.e. plot with `dots`).

**OPTIONAL for ALL students**: Generalize the game in various ways such as

1. Adding an option which 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 which controls the placement of new points: new points should continue to be placed along a line 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 a new source file and executable—`chaos2d` must execute precisely as specified above. Leave comments describing your generalization(s) in a `README` file in the solution directory.

**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 should again use the built-in `real*8` function `rand` on `sgi1` to complete this problem.