In-class exercises and homework for week of Sept. 9

Due date: Friday, Sept. 23 by midnight.

Basin boundary dimensions

There are many definitions of dimension in mathematics. The most familiar is the (integer) Euclidean dimension: the Euclidean dimension of the real line is 1, the plane is 2, etc.

Fractals are characterized by a power law, as follows. Take an interval I in the real line (say the unit interval, $I = [0, 1]$). Pick $\epsilon > 0$, which can be as small as you like. The number of intervals of length epsilon, $N(\epsilon)$, into which I can be subdivided is proportional to $1/\epsilon$. We express this relation as $N(\epsilon) \propto (1/\epsilon)^1$.

Now take, say, the unit square, and consider squares of side length ϵ. Each side can be subdivided into about $1/\epsilon$ pieces, and there are two sides, so the number of ϵ-squares, $N(\epsilon)$, needed to cover the unit square is $N(\epsilon) \propto (1/\epsilon)^2$.

A similar argument for the unit cube gives $N(\epsilon) \propto (1/\epsilon)^3$, and the idea can be extended to general open connected regions. The *fractal dimension* of a set is the number d such that the number of ϵ-squares, cubes, etc., needed to cover it is proportional to $(1/\epsilon)^d$. For ordinary curves, rectangles, and cubes, the fractal dimension agrees with the Euclidean dimension.

For fractal sets, however, d is usually not an integer. We can estimate d for the basin of infinity for the Hénon map as follows.

Algorithm D:

Step 0. Apply Algorithm B over a given region R as described in the Aug. 26 assignment. Fix an $n \times n$ grid of some reasonably high resolution and imagine an associated $n \times n$ matrix $\mathbf{B}$ such that $B_{ij} = 1$ if the (i, j)th grid point tends to infinity (as determined by Algorithm B) and $B_{ij} = 0$ otherwise.

Step 1. Choose $\epsilon > 0$. Shift the grid by ϵ units to the right. Repeat Algorithm B to obtain the $n \times n$ matrix $\mathbf{B}^+$.

Step 2. For the same ϵ, shift the grid by ϵ units to the left and repeat Algorithm B to obtain the $n \times n$ matrix $\mathbf{B}^-$.

Step 3. We say that grid point (i, j) is ϵ-*uncertain* if $B^+_{ij} \neq B_{ij}$ or $B^-_{ij} \neq B_{ij}$. Let $N(\epsilon)$ be the number of such ϵ-uncertain points.

Step 4. Repeat Steps 1–3 for a sequence of ϵ's tending to 0. Output ϵ and $N(\epsilon)$.

Step 5. Read the output from Step 4 into a statistics package to estimate the constants C and p such that $N(\epsilon) = C\epsilon^p$. (The simplest way to do this is to take the logarithm of each side and perform a linear least-squares fit using your calculator, MATLAB, R, or similar package.) The fractal dimension d of the basin of infinity is $d = 1 + p$.

The assignment

Implement Algorithms B and D, described in the Aug. 26 lab, to compute the dimenson of the basin of attraction of the Hénon map in Fortran. Use a 4096×4096 grid on the square $[-3, 3] \times [-3, 3]$. Choose $\epsilon = 2^{-12}, 2^{-13}, 2^{-14}, \ldots, 2^{-21}$ and estimate d using linear least squares (details are given in at the end of this writeup).

The details of how you implement your code are up to you, with the following provisos.

1. I have included one file, `precision.f90`, containing declarations for single- and double-precision `kinds`.

2. The file `henondim.f90` should contain whatever routines you need to implement Algorithms B and D. Your final program does *not* have to implement any graphics—all you need to do is to print out a list of epsilons with the corresponding uncertain point counts. (Of course, you are welcome to output the basin itself into another file for plotting and testing; suggestions are given below.)

3. First write routines that you need to implement and test Algorithm B, which you already have working in MATLAB. Then write the code that you need to implement Algorithm D. All this code, including a driver routine for Algorithm D, go into `henondim.f90`.

4. The file main.f90 should contain your main program, which should read in whatever parameters you choose and call the driver routine in `henondim.f90` to output the epsilons and uncertain point counts.

5. Write as robust a program as you can on a single processor.

6. I have no particular requirements as to vectorization. If you can vectorize, then please do so—but in a compiled language on an x86 processor, good scalar code can be more efficient than vectorized code depending on how you handle memory (more details will be given in lecture next week).

7. For what it's worth, I was able to implement Algorithms B and D in about 67 lines of code (without comments). Final code length is not a grading criterion, but I mention this statistic to emphasize that you should be able to complete the assignment in a couple of pages of commented code.

8. A Google search will return many sites with (often contradictory) suggestions about comments. Do a little bit of research and choose a style that you like—then apply it consistently. The general rule is to be clear and concise. Assume that the reader is a literate programmer.

9. The files `precision.f90` and `henon_sample.f90` are provided to get you started. They also illustrate one method for documenting modules that (in my experience, at least) has proven useful for large projects. However, you are free to substitute your own documentation convention, as long as it is clear and concise.

10. The included Makefile will be discussed in lecture. To compile your program, type

```
make
```

at the command-line prompt. If you just want to compile just a piece of it to test for syntax errors, say (for example)

```
make henondim.o
```

11. You will also need to write a main program that reads in whatever variables that you need, allocates space for any required arrays, and writes out the uncertain point count as a function of epsilon. This main program goes into `main.f90`.

12. Suppose that you keep the list of epsilons and the uncertain point counts as the arrays `eps` and `uncertain`, respectively. We won't be fussy about formatting, so given `NEPS` elements in each, you can write them out with

```
do j=1,NEPS
     write(6,*) eps(j), uncertain(j)
enddo
```

13. Package your code as follows:

```
tar -c -f dimension.tar precision.f90 henondim.f90 main.f90 Makefile
```

and include any other data files that you require.

Suggestions for debugging

1. Start small; use a 100×100 grid for testing, then increase the size to the required 4096×4096.

2. Your final program does *not* need to do any graphics, and it does *not* need to write out any data to plot the basin boundary. However, you may wish to do so for testing purposes. A Fortran code skeleton might look like this:

```
program test_henon
use henondim
implicit none
integer, allocatable:: basin(:,:)    ! the basin boundary array
real(DOUBLE):: xmin, xmax, ymin, ymax
integer:: n
other declarations as necessary
read(5,*) n    ! grid size
allocate(basin(n,n))
...
your code for generating the basin here
open(unit=4,file='basin.out',form='unformatted',access='stream', &
   position='rewind')
```

```
write(4) basin
close(4)
...
stop
end program test_henon
```

3. To visualize the results, point MATLAB to the directory in which you wrote the output of your Fortran program. To read `basin.dat`, assuming that it is a 100×100 `integer` array, say

```
fid = fopen('basin.out')    no semicolon
A = fread(fid, [100,100], 'int32');    semicolon
```

   Don't put a semicolon after the `fopen` statement; if the result is nonzero, then your file has been opened successfully. (Otherwise, you will have to investigate the problem before continuing.) MATLAB reads `basin` in column-major order and Fortran writes `basin` in column-major order, so the indicated read and write statements are compatible.

4. The test program declares `basin` as an `allocatable` array and allocate space for it. Don't worry about handling allocation failures: if the allocation fails, then you can't perform the computation, so it's best to let the Fortran runtime report the problem and halt your program. The initial contents of `basin` are *garbage* following the allocation! You can initialize `basin`, say to all zeros, with a statement like

```
basin = 0
```