In-class exercises and homework for Aug. 27

Due date: Wednesday, Sept. 5 at 11:59 p.m.

Preliminaries

The objective today is to use MATLAB to illustrate some concepts of vectorization and cache management. We will also implement a parallel algorithm that generates a fractal.

Note. If you cannot be MATLAB running on your laptop, then please work with another student, and the two of you may upload a joint paper.

To get started, perform the following steps.

1. After you have installed MATLAB on your computer, please start it by double-clicking the appropriate icon. You will get a workspace window with several subwindows, depending on the configuation. The Command Window is the most important one for this exercise.

2. Make note of the Current Folder, which appears below the icon ribbon at the top of the MATLAB window. It may say something like /Users/name/Documents/MATLA or similar, where name is your login name on your computer. You may change this folder to another one of your choosing by clicking on the name and typing the name of a new folder (be sure to create the new folder first).

3. From Blackboard, download the files v1.m, v2.m, and v3.m and move them to the Current Folder in your MATLAB session. This step is very important—if you don't move the file to the right folder, then you'll get error messages later.

Practice problem

Each of the MATLAB scripts v1.m, v2.m, and v3.m adds two 1000×1000 random matrices. The tic/toc statements start and end an internal timing process.

Run each script several times. (In the command window, type v1 and hit the return key, or double click on the file name to start the Editor and click on the green Run arrow. Proceeed similarly for the other scripts.)

Discuss the following questions with your classroom colleagues:

1. Which script executes the fastest?

2. What is the time factor between the fastest and the slowest scripts?

3. What are the main differences between the scripts?

4. Which script(s) are the most cache friendly?

You do not need to turn in your answers to these questions for grading.

Basin boundaries of planar diffeomorphisms

Consider a point (x, y) in the plane $\mathbb{R}^2$. An invertible differentiable function $f(x, y)$ whose output value is another point in $\mathbb{R}^2$ is called a *planar diffeomorphism*. One famous example of such a function is the Hénon map, named after the astronomer Michel Hénon who first introduced it [1]. One convenient form is given by

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} a - x_n^2 + b y_n \\ x_n \end{pmatrix}. \tag{1}$$

If we start with the point (x_0, y_0), called the *initial condition*, then we can generate the *orbit* $\{(x_1, y_1), (x_2, y_2), \ldots\}$ by *iterating* Eq. (1). (To iterate means to repeat. In this case, we simply apply the Hénon map repeatedly starting from the initial condition.)

The Hénon map has many interesting properties. For many values of a and b, the orbit of (x_0, y_0) can be chaotic and can generate what is sometimes called a *strange attractor*.

In this exercise, however, we are interested in a simpler question: Given (x_0, y_0), does the orbit go to infinity or does it stay bounded? The set of initial conditions that goes to infinity is called the *basin* of infinity. Your objective in this exercise is to compute and plot this basin.

The Hénon map exhibits the *lockout property*: once (x_n, y_n) exceeds a certain distance L from the origin, called the *lockout region*, subsequent points must go to infinity. To approximate the basin of infinity, we choose an integer K and assume that if (x_K, y_K) has not left the lockout region, then the orbit stays bounded.

Given these properties, the following algorithm determines, with good accuracy, whether a given initial condition (x_0, y_0) lies in the basin of infinity:

Algorithm B.

```
1. Fix $K$, $L$, $a$, and $b$.
2. Mark $(x_0, y_0)$ as not belonging to the basin of infinity.
3. For $j = 1, \ldots, K$ do:
    • $(x_j, y_j) \leftarrow$ Hénon$(x_{j-1}, y_{j-1})$
    • If $\|(x_j, y_j)\| > L$ then
        – Mark $(x_0, y_0)$ as belonging to the basin of infinity.
        – Exit the loop.
```

To plot the basin of infinity, we choose a box $B = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, generate a grid inside B, and apply Algorithm B to every (x_0, y_0) in the grid.

Algorithm B has some important properties:

- It is not necessary to store all the points on the orbit. You can start with a single point $(x, y) = (x_0, y_0)$ and overwrite it on each iteration.
- Algorithm B can be applied *independently* to each point (x_0, y_0) on the grid. Thus, the basin boundary computation is *embarrassingly parallel*.

Vectorization

We will revisit Algorithm B over the course of the semester. It is a simple but useful way to learn about the various forms of parallelism on modern computer architectures. Today we will start with the simplest version: vectorization.

Vectorization is a form of SIMD (Single Instruction Multiple Data) computing. If we have two n-vectors $\mathbf{x}$ and $\mathbf{y}$, a vectorized code computes their sum as a single statement:

$$\mathbf{z} = \mathbf{x} + \mathbf{y}$$

The compiler or interpreter generates code that loads several elements of $\mathbf{x}$ and $\mathbf{y}$ with a single instruction, adds them with a single instruction, and stores the sum into $\mathbf{z}$ with a single instruction. Most versions of the x86 (Intel-compatible) instruction set can operate on 4 single-precision (32-bit) elements per instruction. This capability, when it is applied to the innermost loops of a calculation, can significantly improve performance relative to the equivalent scalar (non-vectorized) code.

MATLAB

If you are not familiar with MATLAB, you can find a brief introduction here. You may find an overview of vectorization in MATLAB on the Mathworks website.

Vectorization is particularly important in MATLAB. Roughly speaking, MATLAB parses every statement one and a time and generates a sequence of subroutine calls to perform the requested operations. That overhead is significant. Therefore, the more work that you can do with each MATLAB statement, the faster your script will run, because the overhead is spread over many operations. For example, if x and y are n-vectors, then it is considerably more efficient to write

```
z = x + y;
```

to perform the addition and assignment operation than to do the same work with a loop like

```
for k=1:n
   z = x + y;
end
```

The reason is that the vector expression spreads the overhead of MATLAB processing over n operations, whereas each iteration of the loop incurs the MATLAB overhead for only one operation. MATLAB and Fortran provide excellent support for the vectorized programming model.

An equally important consideration is memory *stride*. As described in lecture, cache memory hierarchies operate most efficiently with programs that access data sequentially. If most of the computation involves manipulating arrays of data, then the running time of the program can be minimized by making sure that accesses are along the columns in MATLAB and Fortran.

The assignment

Write a vectorized implementation of Algorithm B in MATLAB. Your code will be a function of the form

```
function basin = basin_henon(n, xmin, xmax, ymin, ymax, param)
```

as follows:

- n is a positive integer and basin is an $n \times n$ array
- xmin, xmax, ymin, and ymax define the limits of a rectangular region in $\mathbb{R}^2$ over which the basin is to be computed.
- param is a data structure with the following components:
  - a and b, giving the parameters of the Hénon map
  - maxiter, the MAXimum umber of ITERations for any point (i.e., the value of K in Algorithm B)
  - lockout, the parameter L in Algorithm B.

Your program will generate n equally spaced points along the interval $[x_{\min}, x_{\max}]$ and n equally spaced points along $[y_{\min}, y_{\max}]$ to form an $n \times n$ grid of initial conditions. You will then apply Algorithm B (or an equivalent computation) to each of the points in a vectorized fashion. If the (j, k)th point in the grid goes to infinity, then set basin(j,k)=1. Otherwise, set basin(j,k)=0. Your program should be able to run in a few seconds (depending on your computer) for a 1000×1000 grid of initial conditions for maxiter $= 100$ and $L = 100$.

Please observe the following rules.

1. Place only one function in each MATLAB source file, whose name matches the function. For example, a function called f must be placed in a file called f.m.
2. Every function must be documented to explain its purpose and the meaning of all arguments. See "Add Help for Your Program Files" in the MATLAB help system for an example.
3. Bundle basin_henon.m and any other required MATLAB files in a tar file using the command of the form

   ```
   tar -c -f hw1.tar basin_henon.m henon.m
   ```

   You will upload a file named hw1.tar to the Blackboard site. Additional details will be provided in class.
4. You may use any computer you wish for this assignment, but you must submit a tar file (not a zip file).

5. You may download MATLAB on your own computer from myapps.asu.edu. Please see http://help.asu.edu/sims/selfhelp/SelfhelpKbView.seam?parature_id=8373-8193-6992&source=Selfhelp&cid=26107 for installation and license file instructions. *Caution:* MATLAB is a big program (several gigabytes). Do *not* try to download MATLAB using your local coffee shop's wireless network—the download is likely to fail and you will not make friends with the proprietor. Use the ASU network instead.

Hints and suggestions

1. Start by writing a function of the form

   ```
   function y = henon(x, param)
   ```

   which applies the Henon map to one (or many) points x. If x is a 2-vector, then y is also a 2-vector and is computed using Eq. (1) directly. However, you should also allow x to be either an $n \times 2$ (or $2 \times n$) array at your option, each row (respectively column) of which is one initial condition. In this case, y is an array with the same dimensions as x whose corresponding rows (columns) are given by Eq. (1). When x is an array, you should code Eq. (1) using vectorized operations. Your code should use unit strides whenver practical (points will be deducted otherwise). Here is a code skeleton to get you started:

   ```
   function y = henon(x, param)
       [n, m] = size(x);
       if(n*m == 2) % X is a 2-vector
           y = zeros(n, m); % row or column vector according to X
           y(1) = param.a - x(1)^2 + param.b*x(2);
           y(2) = x(1);
       else % assuming X is N-by-2
           y = zeros(n, 2);
           your vectorized code here
       end
       return
   end % henon
   ```

2. In MATLAB, $*$ refers to matrix multiplication, which causes an error if the dimensions of the operands are not compatible. If you want to multiply two n vectors elementwise, say x.*y instead. Exponentiation works the same way: x.^2 squares each element of x. (Addition of vectors is defined elementwise so you can simply write x+y). You can multiply a vector or array by a scalar: 2*x is perfectly legal whether x is a scalar, vector, or array.

3. Variables require no declarations in MATLAB. However, it is more efficient to preallocate space for vectors and matrices before you access them. For example,

```
A = zeros(n);
B = ones(n);
```

allocates $n \times n$ arrays; each element of A is set to 0 and each element of B is set to 1. You can also say

```
A = zeros(m, n);
```

to allocate an $m \times n$ array. Use this method to preallocate space for basin (and for any other temporary arrays that you may need). See the help menu for more details.

4. Suppose that x and y are both n-vectors. The expression

```
y(x < 0) = 0;
```

sets y_j to 0 if x_j is negative (and leaves y_j unchanged otherwise), $j = 1, \ldots, n$. This is a more efficient construct than a for loop containing an if statement.

5. Alternatively, you can obtain a list of indices in x of negative elements and zero the corresponding elements of y with

```
ix = find(x < 0);
y(ix) = 0;
```

The expression numel(ix) returns the number of negative elements of x. You can then use ix wherever it is convenient.

6. If you want to save only the negative elements of x and their counterparts in y, you can say

```
y = y(x < 0);
x = x(x < 0);
```

This overwrites each of x and y with shorter vectors (possibly of zero length). Be sure to do the assignment to y *before* overwriting x! You can also write x = x(ix); y = y(ix); if you use find.

7. It suffices to determine whether a point is outside the lockout region by testing whether the absolute value of either component exceeds L.

8. Once you determine that a point is outside the lockout region, you want to set the appropriate element of basin and then discard the point. Otherwise, your program will generate many floating-point overflows, which will significantly degrade its performance. Use one of the above methods to filter diverging points.

9. The statement

```
x = linspace(xmin, xmax, n);
```

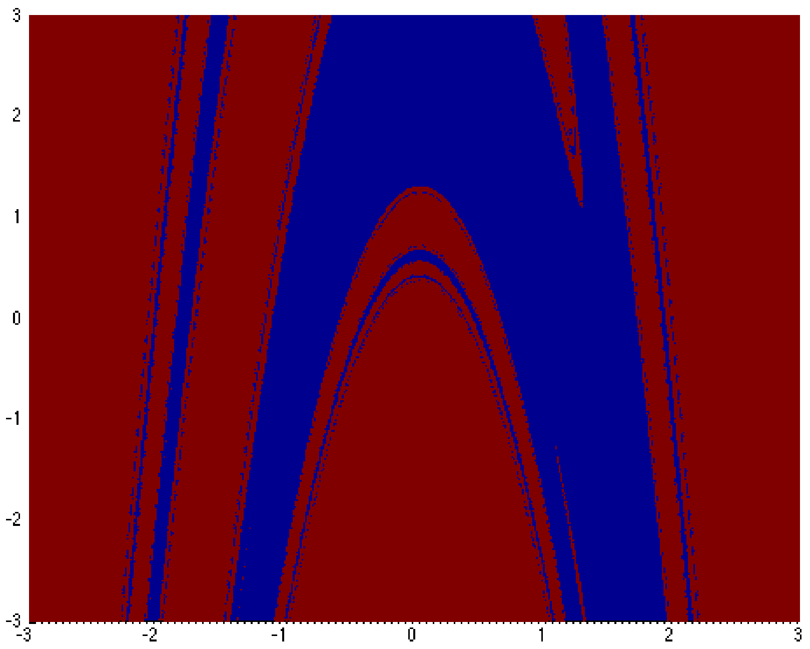

Figure 1: Basin of infinity for the Hénon map. Red points tend to infinity; blue points remain bounded.

returns an n-vector x such that x(1) equals xmin, x(n) equals xmax, and the remaining elements are equally spaced in between. It is useful for generating your grid.

10. You can plot the basin with the following commands:

```
x = linspace(xmin, xmax, n);
y = linspace(ymin, ymax, n);
surf(x, y, basin);
view(0, 90);
```

Figure 1 shows a plot of the basin of infinity for a 1000×1000 grid in the region $[-3, 3] \times [-3, 3]$ for the parameters $a = 2.12$, $b = -0.3$, $L = 100$ and maxiter = 100. (There is nothing special about this choice of a and b. Feel free to experiment with other values. If you fix b, you will notice that the basin undergoes a series of bifurcations as a increases from about $a = 1$ or so. The suggested value of maxiter is conservative; you can get a similar picture with values of 50 or possibly even less. The reason is that once x_n gets large enough, it tends to infinity at a rate proportional to x^{2^n}—i.e., very rapidly.)

11. **Due date:** Friday, Sept. 9 at 11:59 p.m.

Program design considerations

Although no synopsis can be comprehensive, the guidelines presented here will help you design a readable and efficient MATLAB program. Here are some additional considerations that apply to any programming language:

1. Every routine should perform one easily explained task (no more, no less). This keeps your code comprehensible and prevents any one section of code from becoming overly complex.

2. Functions and subroutines should not exceed one page (about 60 lines) in length. Functions that are significantly longer than one page almost always can be split into simpler pieces.

3. Every dummy argument needs a comment describing its type and contents.

4. Argument lists should not be excessively long (i.e., more than 6–8 quantities). If you find yourself passing the same two or three arguments repeatedly, then consider grouping them into a derived type (data structure or class).

5. Avoid global variables without good reason.

6. Avoid "kitchen sink" classes, i.e., classes containing nearly every program variable. Such classes function effectively like global variables.

7. Loop nests should not be more than two levels deep in any function. Array sections and good design can avoid excessive nesting.

8. Observe the following rule for variable naming conventions: the greater the scope of the variable, the more descriptive the name should be. For example, a name like k suffices for the loop counter for a simple local loop, particularly if k is also used as an array subscript:

   ```
   for k=1:n
       y(:,k) = x(:,k);
   end
   ```

   Excessively long names can make your code hard to read:

   ```
   for arrayCopyLoopCounter=1:n
       y(:,arrayCopyLoopCounter) = x(:,arrayCopyLoopCounter);
   end
   ```

   In the context of Algorithm B, a name like `maxiter` is concise and mnemonic. It is a matter of taste whether to prefer a variable name like `MaximumIterationCount`. Similar comments apply to the names `a` and `b` to refer to the parameters in Eq. (1).

9. There are two conventions for naming long variables: `long_variable_name` and `LongVariableName` (or `longVariableName`). Choose one or the other, but don't mix the two.

References

[1] M. Hénon, *Comm. Math. Phys.* **50**, 69 (1976).