Computer lab exercises and homework for Oct. 15

Due date: Tuesday, Oct. 23 at midnight. Please read this writeup in its entirety before starting the next programming assignment. You need only be concerned with the In-Class Exercise for this afternoon.

The *QR* decomposition

One of the simplest statistical models is linear regression. Given a collection of data points $\{(x_i, y_i)\}_{i=1}^n$, assume that they satisfy the linear relation (straight-line fit)

$$y_i = b_1 + b_2 x_i + \varepsilon_i, \quad i = 1, 2, \ldots, n \tag{1}$$

where ε_i is a "noise" term—more specifically, a random variable drawn from a normal distribution with mean 0 and variance σ^2. In matrix form, Eq. (1) may be rewritten as

$$\underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}}_{\mathbf{y}_{n\times 1}} = \underbrace{\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix}}_{\mathbf{X}_{n\times 2}} \underbrace{\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}}_{\mathbf{b}_{2\times 1}} + \underbrace{\begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}}_{\mathbf{e}_{n\times 1}}. \tag{2}$$

Remark. To fit a quadratic model of the form $y_i = b_1 + b_2 x_i + b_3 x_i^2 + \varepsilon_i$, simply add a third column to $\mathbf{X}$ consisting of the x_i^2's.

Equation (2) can be solved in terms of the *normal equations*:

$$\mathbf{b} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}.$$

In practice, you do *not* want to compute $\left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}$ directly, because it can be subject to considerable roundoff error. A much better numerical algorithm involves computing the *QR factorization* of $\mathbf{X}$.

Given a linearly independent set of vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, you can generate an orthonormal set $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\}$ with the same span using Gram-Schmidt orthogonalization:

$$\begin{aligned} \mathbf{u}_1 &= \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|} \\ \mathbf{u}_2 &= \frac{\mathbf{x}_2 - (\mathbf{x}_2 \cdot \mathbf{u}_1)\mathbf{u}_1}{\|\mathbf{x}_2 - (\mathbf{x}_2 \cdot \mathbf{u}_1)\mathbf{u}_1\|} \end{aligned}$$

in a process that may be repeated inductively. The *QR* factorization of $\mathbf{X}$ is

$$\mathbf{X}_{n\times k} = \mathbf{Q}_{n\times k}\mathbf{R}_{k\times k}$$

where $\mathbf{Q}$ is orthonormal ($\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{I}_{k\times k}$) and $\mathbf{R}$ is upper triangular. Then

$$\mathbf{X}^{\mathrm{T}}\mathbf{X} = (\mathbf{QR})^{\mathrm{T}}(\mathbf{QR}) = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{QR} = \mathbf{R}^{\mathrm{T}}\mathbf{R}$$

and

$$\mathbf{X}^{\mathrm{T}} = (\mathbf{QR})^{\mathrm{T}} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}.$$

Therefore, the normal equations

$$\left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)\mathbf{b} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$$

reduce to

$$\mathbf{R}^{\mathrm{T}}\mathbf{R}\mathbf{b} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{y},$$

which implies that

$$\mathbf{R}\mathbf{b} = \mathbf{Q}^{\mathrm{T}}\mathbf{y}$$

since $\mathbf{R}$ is nonsingular. Let $\mathbf{z} = \mathbf{Q}^{\mathrm{T}}\mathbf{y}$; then we solve for $\mathbf{b}$ by back substitution in the triangular system $\mathbf{Rb} = \mathbf{z}$. The *QR* decomposition may be done with partial pivoting (i.e., by a rearrangement of the columns of $\mathbf{X}$) to reduce the effect of roundoff error.

LAPACK routines for *QR* decomposition

The LAPACK library contains a "driver" routine called dgels, which performs the *QR* decomposition and solves for the parameter vector $\mathbf{b}$. Its calling sequence is

```
call dgels('N', n, k, nrhs, X, ldx, y, ldy, work, lwork, info)
```

Here is a summary of the arguments. X, y, and work are double precision; the others are integer.

- X is an $n \times k$ array ($k = 2$ in Eq. (2)).
- work is a work array of length lwork, which depends on the size of the problem.
- y on entry is an $n \times$ nrhs matrix of observations. (In the problem above, nrhs $= 1$.) It is overwritten with the k regression parameters on return (in column 1 in this problem), followed by the sum of squares of the solution vector.
- work is a work vector of length lwork. The LAPACK documentation provides more detail on the value of lwork.
- info is an integer return value; it is 0 if no error is encountered and nonzero otherwise.
- ldx and ldy are the integer "leading dimensions" of X and y, respectively.

The "leading dimension" convention in Fortran lets you allocate one $n \times k$ matrix that can be passed to subroutines to solve linear systems of size $\leq n$. For example, if the actual array X is 5×5, then we would set ldx $= 5$, but we can solve a 3×2 least-squares problem by setting n $= 3$ and k $= 2$. In this case, dgels uses the upper 3×2 block of X:

$$\mathbf{X} = \begin{pmatrix} \mathsf{x} & \mathsf{x} & \mathsf{o} & \mathsf{o} & \mathsf{o} \\ \mathsf{x} & \mathsf{x} & \mathsf{o} & \mathsf{o} & \mathsf{o} \\ \mathsf{x} & \mathsf{x} & \mathsf{o} & \mathsf{o} & \mathsf{o} \\ \mathsf{o} & \mathsf{o} & \mathsf{o} & \mathsf{o} & \mathsf{o} \\ \mathsf{o} & \mathsf{o} & \mathsf{o} & \mathsf{o} & \mathsf{o} \end{pmatrix}$$

The elements marked as o are not accessed, and their values are unchanged.

In-class exercise

Write a Fortran subroutine, linfit, that acts as a wrapper around dgels for a linear regression of an n-vector of data $\mathbf{y}$ that is a function of a single variable $\mathbf{x}$. Let the interface be as follows, assuming the use of the precision module from the previous homework:

```
subroutine linfit(model, n, x, y, param, ierr)
character(*), intent(in):: model
integer, intent(in):: n
real(DP), intent(in):: x(n), y(n)
real(DP), intent(out):: param(2)
integer, intent(out):: ierr
```

The file lsq.f90 contains a code skeleton. The meaning of the arguments is as follows:

- model is a character string:

  'linear' or 'Linear' $\quad y = b_1 + b_2 x$
  'power' or 'Power' $\quad y = b_1 x^{b_2}$

  You need only to check the first letter of model to distinguish between the two possibilities:

```
select case(model(1:1))
case('L', 'l')
   straight-line model
case('P', 'p')
   power-law model
case default
   unknown model; set ierr appropriately
end select
```

A power law can be fit as a linear regression by taking the logarithm of both sides:

$$y = b_1 x^{b_2} \iff \log y = \log b_1 + b_2 \log x.$$

- x and y are the n input data points.
- On successful return, param(1:2) holds the model parameters $[b_1, b_2]$.
- ierr is an output error flag with the same meaning as in sgels. If the return value is nonzero, then an error occurred and the contents of param are undefined.

Working storage of variable size can be allocated on the stack simply by declaring it within linfit:

```
real(DP):: xmatrix(n,2)
...
xmatrix(:,1) = 1.0
xmatrix(:,2) = x
```

The expression log(x) returns the natural logarithm of x, which may be a scalar, vector, or matrix; the result is returned elementwise.

Remarks on genericity. The file lsq.f90 contains an *interface block* that does two things. First, it supplies an *explicit interface* for the LAPACK routines sgels and dgels, i.e., it provides type, kind, and rank (TKR) information for each subroutine argument in the call list. Type refers to a built-in type (e.g., integer, real) or a user-defined type. Kind refers to single or double precision in the case of real arguments and to the width of integer arguments. Rank refers to the number of scripts on each variable (0 for scalars, 1 for singly-subscripted vectors, 2 for matrices, etc.) The compiler is required to diagnose and report TKR mismatches between the actual arguments and the dummy arguments when an explicit interface is in scope. Second, the interface block defines a generic name, xgels. A statement of the form

```
call xgels(tr, n, k, nrhs, x, ldx, y, ldy, work, lwork, ierr)
```

causes sgels to be called when the floating-point arguments are single precision and dgels for double precision. The supplied arguments *must match exactly one* of the subroutines supplied in the interface block. If, based on TKR information, the compiler cannot find a match or cannot distinguish between the subroutines, then it throws a fatal error.

Precision-independent code. One common convention is to declare floating-point variables in terms of a Working Precision, which is specified once when the code is compiled. For example, we could write

```
use precision
integer, parameter:: WP=DP     working precision is double
...
real(WP):: x(ldx,k), y(ldy,nrhs), work(lwork)
...
call xgels('N', n, k, nrhs, x, ldx, y, ldy, work, lwork, ierr)
```

and the compiler calls dgels. To change the precision to single, change the assignment in the third line to WP=SP and recompile; the compiler calls sgels instead.

Floating-point constants can be expressed in a similar way:

```
real(WP):: x0 = 0.1_WP
```

This code initializes x0 to either a single- or double-precision representation of $1/10$. The code

```
real(WP):: x0 = 0.1
```

initializes x0 to a *single-precision* copy of $1/10$, because by default, an expression like 0.1 is single precision.

Testing your routine. One way to test your routine is to write a main program that generates a set of data points that are related by a straight line and corrupt them with noise:

```
program test_linfit
use linfit
implicit none
real(DP):: x(20), y(20), noise(20), param(2)
integer:: j
call random_number(noise)    uniformly distributed in [0, 1]
do j=1,20
   x(j) = j
   y(j) = 1.0 + 2*x(j) + 0.05*(noise(j) - 0.5)
enddo
...
call linfit(...)
if(ierr == 0) write(6,*) param
```

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 $\varepsilon > 0$, which can be as small as you like. The number of intervals of length epsilon, $N(\varepsilon)$, into which I can be subdivided is proportional to $1/\varepsilon$. We express this relation as $N(\varepsilon) \propto (1/\varepsilon)^1$.

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

A similar argument for the unit cube gives $N(\varepsilon) \propto (1/\varepsilon)^3$, 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 $(1/\varepsilon)^d$. 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 previously in the Aug. 27 MATLAB vectorization assignment. Fix an $n \times n$ grid of some reasonably high resolution (see more details below) 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 $\varepsilon > 0$. 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 $N(\varepsilon)$ be the number of such ε-uncertain points.

Step 4. Repeat Steps 1–3 for a sequence of ε's tending to 0.

Step 5. Pass the ε's and $N(\varepsilon)$'s to your linfit subroutine to estimate the constants C and p such that $N(\varepsilon) = C\varepsilon^p$. The fractal dimension d of the basin of infinity is $d = 1 + p$.

The homework assignment

Due date: Tuesday, Oct. 23 at midnight. Implement Algorithms B and D to compute the dimenson of the basin of attraction of the Hénon map in Fortran, C, or C++ using OpenMP to thread the computation. Use a 4096×4096 grid on the square $[-3,3] \times [-3,3]$. Choose $\varepsilon = 2^{-12}, 2^{-13}, 2^{-14}, \ldots, 2^{-21}$ and estimate d using linear least squares (details are given at the end of this writeup). The details of how you implement your code are up to you, with the following provisos.

1. The file henondim.f90 should contain whatever routines you need to implement Algorithms B and D. A code skeleton is provided to get you started. Your final program does *not* have to implement any graphics, and it does *not* need to write out the basin boundary (but you are welcome to do so debugging; see suggestions below). All you need to do is to print out the estimated basin dimension.

2. To access the LAPACK routines when using the Intel compilers, include the -mkl flag (for Math Kernal Library) on the link line.

3. First write Fortran 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. Include a file called main.f90 that contains your main program, which should read in whatever parameters you choose and call the driver routine in henondim.f90 to compute the epsilons and uncertain point counts. Then your main program can call linfit to compute a power-law fit to the basin data.

5. Write as robust a program as you can on a single processor *before* you implement threading.

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.

7. For what it's worth, I was able to implement Algorithms B and D in about 67 lines of code (without comments and without linfit). 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. You are free to attempt threading over any part of the algorithm. In general, the best scaling occurs when you thread at a high level. You may thread by dividing up the entire computation by epsilons, or by dividing the domain into suitable pieces, or by some other method.

9. Do not spend a lot of effort to get the fastest possible code. Instead, grading will be based in part upon how well your program scales—is your code about twice as fast on two threads as on one? three times as fast on three threads? For example, your code might take 120 seconds on one processor but 65 seconds when threaded with OMP_NUM_THREADS=2 and 45 seconds on three threads. A different code might run in 90 seconds on a single cpu, 60 seconds on two threads, and 50 seconds on three; although faster on one processor, the latter code does not scale as well as the former.

10. Scaling is limited by cache coherency and memory contention. Try to get good scaling on up to 4 cpus on Agave. (It may be difficult to get good timing data if many other codes are running on the same node; run top to get a list.)

11. A Google search will return many sites with (often contradictory) suggestions about commenting program source code. 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.

12. If you program in C or C++, then you will still need to call dgels. This is an exercise in inter-language calling (and you'll have better numerics than if you simply code

the naïve formulas for straight-line fitting). The general rule is to pass arguments to Fortran by *address*. You may also need to append an underscore to the Fortran name: dgels_ or similar, depending on the C and Fortran compilers that you use.

13. The file henondim.f90 is provided to get you started. It illustrates 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.

14. Write a makefile for your project. Include the appropriate OpenMP options for whichever compiler family (Intel or GNU) that you use.

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))
basin=0     the contents are garbage otherwise!
...
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.

Instructions for uploading

Upload a tar file, henon.tar, containing the following files:

- Makefile
- main.f90 and henondim.f90. Replace the f90 suffix with c if programming in C and with cc if programming in C++.

Be sure that your name appears *in each source file*. If you choose to work with another student, then so indicate in *each source file*. You both get the same grade.

Grading rubric

Plagiarism warning: Substantial copying of source code without attribution will result in a grade of zero on this assignment. Please see the syllabus for the rules. Oherwise, this assignment is worth 20 points, as follows.

Compiles without errors or warnings and produces correct results	8 points
Uses OpenMP and scales on up to 4 threads	5 points
Calls dgels for least squares	3 points
Includes a working Makefile	2 points
Includes adequate documentation in the source code	2 points
Total	20 points