Computer lab exercises and homework for Sept. 16

Due date: Friday, Sept. 30 at midnight.

Exercise 1: Vectorized forward and backward substitution

The fundamental linear algebra algorithms for solving lower- and upper-triangular systems of equations are called forward and back substitution. For example, to solve the lower-triangular system

$$\begin{aligned} x_1 & = 1 \\ 2x_1 + 3x_2 & = 10 \\ -x_1 + 2x_2 - x_3 & = 5 \end{aligned}$$

we first solve for x_1 using the first equation. Then we substitute x_1 into the second equation and solve for x_2, and so on. To solve an upper-triangular system like

$$\begin{aligned} -x_1 + 2x_2 - x_3 & = 5 \\ 2x_2 + 3x_3 & = 10 \\ x_3 & = 1 \end{aligned}$$

we first solve for x_3 in the last equation and work backwards in a similar way.

The problem with the naïve algorithm is that it doesn't access memory efficiently in MATLAB or Fortran, since it marches across the rows of the matrix. Implement the algorithms so that they access memory columnwise. **Use double precision arithmetic.** One pleasant consequence is that the revised algorithms are vectorizable—and so will parallelize these computations to the extent practical. Package your subroutines into a file called `substitution.f90` as follows:

```
module substitution
use precision
implicit none
contains
subroutine forward_subs(a, n, b, info)
integer,intent(in):: n
real(DOUBLE),intent(in):: a(n,n)
real(DOUBLE),intent(inout):: b(n)
integer,intent(out):: info
…
return
end subroutine forward_subs
```

```
subroutine backward_subs(a, n, b, info)
similarly
end subroutine backward_subs
end module substitution
```

Comment the module and the subroutines in MATLAB style. Although A is declared $n \times n$, your code should access only the lower or upper triangle as appropriate. On entry to each subroutine, `b` is the right-hand side to be solved for; on return, `b` is *overwritten* with the solution x. Unless some diagonal element of A is *exactly* equal to 0, set `info` to 0 on return; otherwise, set `info` to a nonzero value of your choice (and document it). We won't worry about any other floating-point problems here.

Exercise 2: Linear least squares approximation

Write a Fortran program to fit a least-squares line of the form $y = a + b_x$ to a collection of data points $\{(x_i, y_i)\}$. Assume that the input consists of *exactly* two values per line, x_i, y_i. Use the `sgels` routine from LAPACK to determine the parameters a and b.

Your program should operate in a manner similar to other Linux utilities. In particular, it should read from the file named on the command line or the standard input otherwise, and write a and b to the standard output. Include an option for logarithmic fits, as follows:

- The `-Ly` option should fit the model $\log y_i = a + bx_i$, i.e., $y = a_1 e^{bx}$, where $a_1 = e^a$.
- The `-Lx` option should calculate the logarithm of x and the output should be the result of fitting the model $y_i = a + b \log x_i$.
- Finally, `-Lx -Ly` should fit the model $\log y_i = a + b \log x_i$ (as should `-Ly -Lx`).

Here are the requirements for the final code.

1. Use single-precision arithmetic and assume that there are no more than 100 (x, y) pairs. You do not need to issue an error message if there are more than 100 such pairs.
2. If you read in the data from a single main program, then include the intrinsic module `iso_fortran_env`, like this:

```
    program least_squares
    use, intrinsic:: iso_fortran_env
    ...
    do j=1,MAXDATA
       read(INPUT_UNIT,*,end=90) x(j),y(j)
    enddo
 90 continue
```

3. You can output your parameters with

```
write(OUTPUT_UNIT,*) a,b
```

4. Write a subroutine to grab the command-line arguments, if any. The intrinsic (built-in) function `command_argument_count()` returns an integer value. If it is zero, then there are no arguments, so your program must have been invoked as

   ```
   ./least_squares < file.dat
   ```

   or as

   ```
   other_program | ./least_squares
   ```

   In this case, read from the standard input.

5. If the argument count is positive, then there are command line arguments. The only options are `-Lx` and `-Ly`. Your logic goes as follows:

   ```
   character(80):: arg    80 characters should suffice
   ...
   call get_command_argument(k, arg)    kth argument
   if(arg(1:2) == '-L') then
           Inspect arg(3:3) and proceed accordingly
   else    Assume arg is a file name
       close(INPUT_UNIT)
       open(INPUT_UNIT,file=arg,status='old')
   endif
   ```

   The idea is to wrap all this into a `do` loop.

6. Set appropriate flags for taking logarithms. The intrinsic natural logarithm function is `log(x)`. When taking logarithms, you should check for nonpositive data and write an appropriate error message to `ERROR_UNIT`.

7. If the data file cannot be found, then `open` will terminate the program with an error message, so you do not need any extra error handling.

8. The output is simply the two coefficients, without annotation. For example, output

   ```
   1.543  -0.226
   ```

   instead of

   ```
   The first coefficient is 1.543 and the second is -0.226
   ```

   The first form is preferable when you have many data files to process; then it's easier to write a shell script that outputs the results in a form like

```
0.123 4.56 file1.dat
0.789 3.14 file2.dat
```

and so on.

Makefile instructions

Include a makefile whose first target is

```
all: substitution least_squares
```

In this way, typing `make` or `make all` compiles both programs. Use last week's makefile as a template. Your makefile should work correctly on the lab computers in ECA 221.

Linking with LAPACK and BLAS

The general compilation line looks like

```
gfortran precision.o substitution.o -llapack -lblas -o substitution
```

However, there is a configuration glitch on the ECA computers. Here's what you need to do to get around it—and you need to do this *only once*. From your home directory, run the following commands:

```
mkdir lib       create your own library directory
cd lib      point the shell to that directory
ln -s /usr/lib/liblapack.so.3 liblapack.so
ln -s /usr/lib/libblas.so.3 libblas.so
cd       back to your home directory
```

In your makefile, you'll need to include a line of the form

```
LIBS = -L$(HOME)/lib -llapack -lblas
```

Then you may proceed with appropriate edits to last week's makefile.

Submission instructions

You will be graded on the following files:

1. `precision.f90`
2. `substitution.f90`
3. `least_squares.f90`

4. `Makefile`

In each file, please include your name as part of the file's comments. You can create a tar archive using the following command-line statement:

```
tar --c --f hw0916_yourname.tar precision.f90 substitution.f90 least_squares.f90
Makefile
```

Substitute your last name for `yourname`. Upload the tar file to the Blackboard site. You will have up to *two* attempts—if you encounter difficulty or find a bug in the first attempt, then you can fix the problem and try again. If you do submit more than once, then I will grade *only* the second attempt. Assignments are due by Friday, Oct. 1 at midnight.