In-class exercises and homework, Sept. 24

Due date: Monday, Oct. 1 at midnight.

Preliminaries

I would like for you to try to complete this assignment using Agave. All the exercises are very simple, and you are welcome to develop your programs on your laptop, but I would like for you to become familiar with Agave's development environment for future assignments.

Required software

- You will need to download the Cisco SSL VPN software from myapps.asu.edu by logging in with your ASUrite credentials. There are versions for Windows, Mac, and Linux. You *must* run the VPN *before* attempting to log into Agave when you are off campus.
- If you run Windows, download PuTTY from myapps.asu.edu. It includes an SSH client, which you will need to log into Agave. (Skip this step if you run Linux or MacOS.)
- If you wish to develop code on your laptop, then you will need a Fortran compiler. Here is one source for Gfortran.
- Optionally, as a student, you may be able to download Intel compilers for free or for a nominal charge from developer.intel.com by creating an account and following the appropriate links. *Caution:* The compiler packages are very large! Download them only from a fast wireless connection on campus.

Logging into and out of Agave

1. From the PuTTY software, select SSH and enter agave.asu.edu as the computer to log into and your ASUrite ID as your user name. From Linux or Mac, start a terminal and type ssh yourname@agave.asu.edu (replace yourname with your ASUrite ID).
2. If you have not previously logged into Agave, you will get a message to that effect, and you will be asked if you wish to continue. Type yes.
3. Once you are logged into Agave, type interactive to start an interactive session on a compute node. *Please do not develop or debug code on the login nodes.*
4. Once on an interactive node, type

```
module load intel/2018x
```

This command makes the Intel compilers available to you, and you can start on the exercise.

5. When you are finished, type exit to log out of the compute node. Type exit again on the login node to log out of Agave entirely. *Please ensure that you are logged out completely before ending your laptop session.*

Introduction

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.

Practice problem

There are three files that implement this naïve algorithm: precision.f90, naive.f90, and test_naive.f90. There is also a data file called matrix.dat. Download these files to your laptop (and upload them to Agave as necessary).

1. File transfers are accomplished using the graphical user interface on PuTTY.

2. On Linux or Mac, move these files from your Download folder to another convenient directory, say apm525. Once you are in the apm525 directory, you can type

   ```
   scp precision.f90 naive.f90 test_naive.f90 matrix.dat yourname@agave.asu.edu:
   ```

   Notice the colon at the end of the terminal command.

3. In either case, you should see these files in your home directory on Agave. View the contents of your home directory by typing ls.

4. Compile the first two files:

   ```
   ifort -c precision.f90 naive.f90
   ```

Substitute gfortran if you're using that compiler on your laptop.

Remarks on text editors. To use Agave efficiently, you need to learn a Unix text editor. The simplest one is pico—you move the cursor around to where you want to edit or insert text, type control-S to save, then control-X to exit. More powerful editors are emacs and vi (or vim). The issue of which is better is something of a religious question; emacs is more customizable than vi and clones, but it has a longer learning curve. Try both of them and decide which one you prefer. Tutorials for all of them are readily found with a web search.

Now try running a simple test problem, as follows.

1. Type ifort -c test_naive.f90. You should see an error message. Find and fix the error, then recompile.

2. Link all the object files together by typing

   ```
   ifort precision.o naive.o test_naive.o -o naive
   ```

   We will discuss a Unix utility to automate this process on Wednesday.

3. Run the program by typing

   ```
   ./naive < matrix.dat
   ```

4. The test program and associated data file solve the system

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 2 & 3 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 5 \\ 14 \end{pmatrix}.$$

5. The test program uses a Fortran i/o method called namelist, which allows input data to be stored in a self-documenting format. The file matrix.dat lists the elements of the matrix by rows, but we could have specified the elements by columns instead.

Homework assignment

The problem with the naive algorithm is that it doesn't access memory efficiently, since it marches across the rows of the matrix one at a time. Implement the algorithms so that they access memory columnwise. **Use double precision arithmetic.** One pleasant consequence is that the do-loops in the revised algorithms are vectorizable—and so will parallelize the computation to the extent possible on one processor.

Package your subroutines into a file called substitution.f90 as follows:

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

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

Comment the module and the subroutines in MATLAB style. Although the array 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 ierr to 0 on return; otherwise, set ierr to a nonzero value of your choice (and document it). We won't worry about any other floating-point problems here. Be certain that your programs can work for a general $n \times n$ system. You will need to write a test program, but you do not need to upload it for grading.

Submission instructions

Please submit two files:

1. precision.f90
2. substitution.f90

(I do not need your test program.) In each file, please ensure that your name appears within the comments. On Agave, you can create a tar archive using the following command-line statement:

```
tar -c -f hw2.tar precision.f90 substitution.f90
```

Upload the tar file to the Blackboard site. You will have up to *three* 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 last attempt. Assignments are due by Monday, Oct. 1 by midnight.