Exercise: Calling LAPACK

In this exercise, we'll use the same conventions and commands as in the batch computing exercise. You should refer back to the batch computing exercise description for details on various Unix commands.

You'll be editing a simple code that will need to use a certain LAPACK routine.

Here are the steps for this exercise:

1. Log in to the Linux cluster supercomputer (`sooner.oscer.ou.edu`).
2. Confirm that you're in your home directory:

   `pwd`
   `/home/yourusername`
3. Check that you have a `NCSIPARII2011_exercises` subdirectory inside your home directory:

   `ls`
   `NCSIPARII2011_exercises`
4. Copy the `LAPACK` directory into your `NCSIIPARII2011_exercises` directory:

   `cp -r ~hneeman/NCSIPARII2011_exercises/LAPACK/ ~/NCSIPARII2011_exercises/`
5. Go into your `NCSIPARII2011_exercises` subdirectory:

   `cd NCSIPARII2011_exercises`
6. Confirm that you're in your `NCSIPARII2011_exercises` subdirectory:

   `pwd`
   `/home/yourusername/NCSIPARII2011_exercises`
7. See what files or subdirectories (if any) are in the current working directory:

   `ls`
8. Go into your `LAPACK` subdirectory:

   `cd LAPACK`
9. Confirm that you're in your `LAPACK` subdirectory:

   `pwd`
   `/home/yourusername/NCSIPARII2011_exercises/LAPACK`
10. See what files or subdirectories (if any) are in the current working directory:

    `ls`
11. Choose which language you want to use (C or Fortran90), and `cd` into the appropriate directory:

    `cd C`

    OR:

    `cd Fortran90`
12. Confirm that you're in your `C` or `Fortran90` subdirectory:

    `pwd`
    `/home/yourusername/NCSIPARII2011_exercises/LAPACK/C`

13. See what files or subdirectories (if any) are in the current working directory:

    `ls`

14. Edit the batch script `lapack_test.bsub` to use your username and e-mail address.
15. If you haven't already examined `lapack_test.c` (or `lapack_test.f90`), do so now.
16. Edit the source file with your favorite text editor (for example, `nano`, `vi`, `emacs` etc):

    `nano  lapack_test.c`

    OR:

    `nano  lapack_test.f90`

17. Look for the line in the source file that says:

    ```
    /* Call to LAPACK routine goes here. */
    ```

    OR

    ```
    !... Call to LAPACK routine goes here.
    ```

    This is where you will write a call to the appropriate LAPACK routine.

18. To find the appropriate LAPACK routine:
    a. Open your favorite web browser (for example, Internet Explorer, Firefox, Safari).
    b. Go to:

       `http://www.netlib.org/lapack/`

    c. When that webpage loads, find:

       `single precision real`

       and click on that link.

    d. When that webpage loads, in the webpage find:

       `solve`

       In particular, you're looking for the solver for a general system of linear equations.

19. When you find that solver, click on the link for its Fortran source code – the file whose extension is `.f` (dot f). (Ignore the link for the source file plus its dependencies.)
20. Using the Fortran subroutine header

    ```
    SUBROUTINE subroutinename (argument1, argument2, ...)
    ```

    and the comments immediately below the header that describe the subroutine's arguments, determine how to call that subroutine from either `lapack_test.c` or `lapack_test.f90`.

21. Modify either `lapack_test.c` or `lapack_test.f90` to call that LAPACK routine.

NOTES

a. In this exercise, we're intentionally using statically allocated arrays. (If you don't know what that means, then don't worry about it.)

b. If you're calling the subroutine from `lapack_test.c`, bear in mind the following:

In Fortran (all versions), **ALL** arguments are *pass-by-reference*, meaning that the routine that you call is using the same memory location for each argument as the calling routine.

For example:

```
SUBROUTINE caller ()
  IMPLICIT NONE
  INTEGER :: actual_argument !! local variable inside caller
  actual_argument = 5
  PRINT *, "caller: before calling, actual_argument = ", actual_argument
  CALL callee(actual_argument)
  PRINT *, "caller: after  calling, actual_argument = ", actual_argument
END SUBROUTINE caller

SUBROUTINE callee (formal_argument)
  IMPLICIT NONE
  INTEGER :: formal_argument !! argument passed to callee from caller
  PRINT *, "callee: before, formal_argument = ", formal_argument
  formal_argument = formal_argument + 1
  PRINT *, "callee: after,  formal_argument = ", formal_argument
END SUBROUTINE callee
```

The output of this code fragment is:

```
caller: before calling, actual_argument =   5
callee: before, formal_argument =   5
callee: after,  formal_argument =   6
caller: after  calling, actual_argument =   6
```
← **NOTICE!!!**

In C, **ALL** arguments are *pass-by-copy*, meaning that the routine that you call is using a different memory location containing the same value for each argument as the calling routine – except for arrays, which are always passed by reference (don't ask).

```
void caller ()
{
  int actual_argument; /* local variable inside caller */
  actual_argument = 5;
  printf("caller: before calling, actual_argument = %d\n", actual_argument);
  callee(actual_argument);
  printf("caller: after  calling, actual_argument = %d\n", actual_argument);
}

void callee (int formal_argument)
{
  printf("callee: before, formal_argument = %d\n", formal_argument);
  formal_argument = formal_argument + 1;
  printf("callee: after,  formal_argument = %d\n", formal_argument);
}
```

The output of this code fragment is:

```
caller: before calling, actual_argument =   5
callee: before, formal_argument =   5
callee: after,  formal_argument =   6
caller: after  calling, actual_argument =   5
```
← **NOTICE!!!**

Why is this relevant?

As you can see from the LAPACK routine source file, LAPACK is written in Fortran. So if you're working in Fortran, you're fine, and you don't have to worry about this.

But if you're working in C, then when you call the LAPACK Fortran routine, for your scalar (non-array) arguments, you have to accomplish pass-by-reference via fakery, by passing a pointer to (that is, the address of) each scalar argument.

For example:

```
void caller2 ()
{
  int actual_argument; /* local variable inside caller */
  actual_argument = 5;
  printf("caller2: before calling, actual_argument = %d\n", actual_argument);
  callee2(&actual_argument);
  printf("caller2: after  calling, actual_argument = %d\n", actual_argument);
}

void callee2 (int* formal_argument)
{
  printf("callee2: before, *formal_argument = %d\n", *formal_argument);
  *formal_argument = *formal_argument + 1;
  printf("callee2: after,  *formal_argument = %d\n", *formal_argument);
}
```

The output of this code fragment is:

```
caller2: before calling, actual_argument =   5
callee2: before, formal_argument =   5
callee2: after,  formal_argument =   6
caller2: after  calling, actual_argument =   6
```
← **NOTICE!!!**

On the other hand:

- **<u>YOU DON'T HAVE TO DO THIS FOR ARRAY ARGUMENTS;</u>**
- **<u>YOU DON'T HAVE TO WORRY ABOUT THIS IN FORTRAN</u>**.

c. The LAPACK implementation you'll be linking to is part of Intel's Math Kernel Library (MKL), rather than a hand-compiled version of an implementation from Netlib or a similar open source provider. We have no specific bias in favor of or against MKL, but we don't have time to build the relevant libraries from source code.

22. Save the source file and close your favorite text editor.

23. Compile using the `make` command:

    `make`

24. If the program doesn't compile, edit the source file and try again. Continue to do this until the compilation succeeds.

25. Submit the batch script file `lapack_test.bsub` to the batch scheduler:

    `bsub < lapack_test.bsub`

    <u>NOTICE</u> the less than symbol < which is **<u>EXTREMELY IMPORTANT</u>**.

    You should get back output something like this:

    ```
    Job <######> is submitted to queue <parii_q>.
    ```

    where `######` is replaced by the batch job ID for the batch job that you've just submitted.

26. Check the status of your batch job:

    bjobs

    You'll get one of the following outputs, either:

    ```
    No unfinished job found
    ```

    (if you get this right after the `bjobs` command, try it several more times, because sometimes there's a pause just before the batch job starts showing up, as shown below):

    ```
    JOBID   USER         STAT  QUEUE    FROM_HOST  EXEC_HOST  JOB_NAME     SUBMIT_TIME
    4081250 yourusername PEND  parii_q  sooner1               lapack_test  Oct 17 14:58
    ```

    where `######` is replaced by a batch job ID number, and `yourusername` is replaced by your user name, and where `PEND` is short for "pending," meaning that your job is waiting to start, OR:

    ```
    JOBID   USER         STAT  QUEUE    FROM_HOST  EXEC_HOST  JOB_NAME     SUBMIT_TIME
    4081250 yourusername RUN   parii_q  sooner1    c127       lapack_test  Oct 17 14:58
    ```

27. You may need to check the status of your batch job repeatedly, using the `bjobs` command, until it runs to completion. **This may take several minutes (occasionally much longer).**

    You'll know that the batch job has finished when it no longer appears in the list of your batch jobs:

    ```
    No unfinished job found
    ```

28. Once your job has finished running, find the ***standard output*** and ***standard error*** files from your job:

    ls -ltr

    Using this command, you should see files named

    ```
    lapack_test_######_stdout.txt
    ```

    and

    ```
    lapack_test_######_stderr.txt
    ```

    (where `######` is replaced by the batch job ID).

    These files should contain the output of `lapack_test`. Ideally, the `stderr` file should have length zero.

29. Look at the contents of the standard output file:

    cat lapack_test_######_stdout.txt

    (where `######` is replaced by the batch job ID). You may also want to look at the `stderr` file:

    cat lapack_test_######_stdout.txt

30. If this run had **ANY** problems, then send e-mail to:

    support@oscer.ou.edu

    which reaches all OSCER staff (including Henry), and attach the following files:

    ```
    makefile
    lapack_test.c
    lapack_test.bsub
    lapack_test_######_stdout.txt
    lapack_test_######_stderr.txt
    ```