Introduction to Using OSCER's Linux Cluster Supercomputer

http://www.oscer.ou.edu/education.php

This exercise will help you learn to use Sooner, the Linux cluster supercomputer administered by the OU Supercomputing Center for Education & Research (OSCER), a division of the University of Oklahoma (OU) Information Technology (IT).

Actions and commands that you should perform or type are in the **`computer boldface`** font.

An account has been set up for you. Your user name is denoted here as `yourusername`, but may actually be of the form `mpi###`, where `###` is a specific user ID number. Or, if you're a permanent OSCER user, your user name may be tied to your name; for example, `hneeman` (Henry Neeman).

If you are registered in the spring 2009 instance of the "Supercomputing in Plain English" workshop series, but you don't yet have an account on Sooner, please send e-mail to Henry Neeman (`hneeman@ou.edu`).

If you have any difficulty with this exercise, then please send e-mail to:
`support@oscer.ou.edu`
which reaches all OSCER staff (including Henry).

The steps for this exercise are listed on the following pages.

There are actually two versions of this exercise: a version in C, and a version in Fortran90. Where possible, descriptions of both will be given, but in some cases, only the C description will be given, with the Fortran90 version assumed.

NOTE: We don't use Fortran77, because it's pure brain poison, but Fortran90 is a very nice language. Also, we don't use C++; instead, we assume that everyone who is comfortable in C++ will be served well enough by C.

I. LOG IN

1. From the PC of your choice (Windows, MacOS, Linux or whatever), bring up your web browser (Internet Explorer, Firefox, Opera, Safari, Chrome or whatever) and go to:
   `http://www.oscer.ou.edu/ssh_install.php`
   NOTICE the underscore in the URL, between `ssh` and `install`.

2. Following the instructions on that page, log in to:
   `sooner.oscer.ou.edu`

3. If you cannot log in to `sooner.oscer.ou.edu`, try logging in to one of the following:
   `sooner1.oscer.ou.edu`
   `sooner2.oscer.ou.edu`
   It turns out that `sooner.oscer.ou.edu` is an *alias* for these other computers; that is, when you log in to `sooner.oscer.ou.edu`, you'll actually get logged into one of these.

4. Once you log in, you'll get some text, and then a *Unix prompt* — probably but not necessarily a percent sign — with the cursor after it, like so:
   ```
   % █
   ```
   There may be some information before the prompt character, such as the name of the computer that you've logged in to (which may be different from `sooner.oscer.ou.edu`), your user name, and so on. For purposes of these materials, we'll generally use the percent sign `%` to indicate the Unix prompt.

5. Check the lines of text immediately above the Unix prompt. If there are lines of text that read something like:
   ```
   No directory /home/yourusername!
   Logging in with home = "/".
   ```
   then you should log out immediately by typing `exit` (followed by pressing **Enter**), and then log back in. If you repeatedly have this problem, then please send e-mail to:
   `support@oscer.ou.edu`
   which reaches all OSCER staff (including Henry).

6. Check to be sure that you're in your *home directory* (a *directory* in Unix is like a folder in Windows, and your *home directory* in Unix is like your desktop in Windows):
   ```
   %  pwd
   /home/yourusername
   ```
   This command is short for "Print working directory;" that is, "print the full name of the directory that I'm currently in." If your current working directory is just a slash (which means the *root directory*, which is like `C:\` in Windows), rather than something like
   `/home/yourusername`
   then you should log out immediately by typing `exit` (followed by pressing **Enter**), and then log back in. If you repeatedly have this problem, then please send e-mail to:
   `support@oscer.ou.edu`
   which reaches all OSCER staff (including Henry).

II. SET UP (FIRST TIME LOGGING IN ONLY)

1. From the PC of your choice (Windows, MacOS, Linux or whatever), bring up your web browser (Internet Explorer, Firefox, Opera, Safari, Chrome or whatever) and go to:
   http://www.oscer.ou.edu/password_change.php
   NOTICE the underscore in the URL, between `password` and `change`.

2. Follow the instructions on that page to change your OSCER password.
   Please note that this password change will affect only your accounts on OSCER machines, not anywhere else (including not affecting any other OU IT accounts if you're at OU).

3. At the Unix prompt, type exactly the bold text below, excluding the percent sign, which indicates the Unix prompt (all commands **MUST** be followed by pressing **Enter**):

   ```
   %  cp  ~hneeman/.vimrc  ~
   ```

   This command means: "Copy the file named `.vimrc` that's in Henry's home directory into my home directory." You **WON'T** have to do this for future logins.
   NOTICE:
   - The Unix copy command is `cp`.
   - The first filename after `cp` is the *source* (the thing that you're making a copy of); the second is the *destination* (the name and/or location of the copy).
   - Henry's username on OSCER supercomputers is `hneeman`.
   - The filename `.vimrc` begins with a period (**very important**). The filename is pronounced "dot vim-are-see."
   - In Unix, filenames are *case sensitive,* meaning that it matters whether you use *upper case* (capital) or *lower case* (small) for each letter in a filename.
   - In Unix, filename pieces are separated by slashes, **NOT** by backslashes as in Windows.
   - The symbol `~` (known as a *tilde*, pronounced "TILL-duh") denotes **your** home directory (another way to denote your home directory is `~yourusername`).
   - The substring `~hneeman` means "the home directory of the user named `hneeman`."
   - If for some reason this doesn't work, try

     ```
     %  cp  /home/hneeman/.vimrc  /home/yourusername
     ```

4. Type the following command:

   ```
   % cp  ~hneeman/.nanorc  ~
   ```

   You **WON'T** have to do this for future logins.

5. Create a *subdirectory* named `SIPE2009_exercises`, like so:

   ```
   % mkdir  SIPE2009_exercises
   ```

   NOTICE: In the subdirectory name `SIPE2009_exercises`, the `SIPE2009` **MUST BE CAPITALIZED**; that is, the directory's name is "capital-S capital-I capital-P capital-E underscore exercises" with no spaces or other characters in between. This command means: "Create a directory named `SIPE2009_exercises` as a subdirectory inside the directory that I'm currently in" (it's like creating a new folder named `SIPE2009_exercises` on your desktop in Windows). You **WON'T** have to do this for future logins.

6. Confirm that you have successfully created your `SIPE2009_exercises` directory by *listing* the contents of the current working directory:

   ```
   % ls
   SIPE2009_exercises
   ```

   This command means: "List the names of the files and subdirectories in my current working directory." **NOTICE** that the command is "ell ess" — that is, small-L small-S — rather than "one ess" and that `ls` is short for "list."

7. Set the *permissions* on your `SIPE2009_exercises` directory so that only **you** can access it:

   ```
   % chmod  u=rwx,go=  SIPE2009_exercises
   ```

   This command means: "Change the *mode* (list of permissions) on my subdirectory named `SIPE2009_exercises` so that **I** (the **user**) can read files in it, write files in it, and go into (execute) it, but nobody else can." Your `SIPE2009_exercises` directory is now accessible **only to you.** The only other people who can access it are the *system administrators* (*sysadmins* for short) of this Linux cluster supercomputer; that is, OSCER operations staff (excluding Henry). You **WON'T** have to do this for future logins.

8. Log out of the Linux cluster supercomputer by typing `exit`. Once you have completed the setup steps in this section, you **WON'T** have to do them again when you log in later.

III. COPY HENRY'S Intro DIRECTORY INTO YOUR SIPE2009_exercises DIRECTORY

1. Log in again.
2. Confirm that you're in your home directory:
   ```
   %  pwd
   /home/yourusername
   ```
3. Check that you have a `SIPE2009_exercises` subdirectory inside your home directory:
   ```
   %  ls
   SIPE2009_exercises
   ```
4. Go into your `SIPE2009_exercises` subdirectory:
   ```
   % cd  SIPE2009_exercises
   ```
   This command means: "Change the working directory to `SIPE2009_exercises`, which is a subdirectory of the current working directory." (This is like double-clicking on a folder icon in Windows.)
5. Confirm that you're in your `SIPE2009_exercises` subdirectory:
   ```
   %  pwd
   /home/yourusername/SIPE2009_exercises
   ```
6. See what files or subdirectories (if any) are in the current working directory:
   ```
   %  ls
   ```
   You may get no output, just the Unix prompt.
7. **SIDEBAR:** To learn more about a particular Unix command, type:
   ```
   % man  commandname
   ```
   For example, try
   ```
   % man  chmod
   ```
   which will give you the online *manual page* for the `chmod` command. The output of `man` goes through another command, `more`, which shows one screenful at a time. To get the next **screenful**, press the spacebar; to get the next **line**, press **Enter**. To **quit** the `more` command, press **Q**.
8. Copy the subdirectory named `Intro` from Henry's `SIPE2009_exercises` directory into your `SIPE2009_exercises` directory:
   ```
   % cp  -r  ~hneeman/SIPE2009_exercises/Intro  ~/SIPE2009_exercises/
   ```
   This command means: "Copy the subdirectory named `Intro` of the directory named `SIPE2009_exercises` under the home directory of user `hneeman` into my directory `SIPE2009_exercises` under my home directory."
9. Confirm that the `Intro` subdirectory was copied into your `SIPE2009_exercises` directory:
   ```
   %  ls
   Intro
   ```
10. Go into your `Intro` subdirectory:
    ```
    % cd  Intro
    ```
11. Confirm that you're in your `Intro` subdirectory:
    ```
    %  pwd
    /home/yourusername/SIPE2009_exercises/Intro
    ```

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

    ```
    %  ls
    C  Fortran90
    ```

13. Go into either your `C` subdirectory or your `Fortran90` subdirectory:

    ```
    % cd  C
    ```

    OR

    ```
    % cd  Fortran90
    ```

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

    ```
    %  pwd
    /home/yourusername/SIPE2009_exercises/Intro/C
    ```

    OR the output of the `pwd` command might be:

    ```
    %  pwd
    /home/yourusername/SIPE2009_exercises/Intro/Fortran90
    ```

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

    ```
    %  ls
    makefile  my_number.bsub  my_number.c  my_number_input.txt
    ```

    OR the source file might be named `my_number.f90` instead of `my_number.c`.

IV. EDIT THE BATCH SCRIPT FILE TO CREATE YOUR OWN UNIQUE VERSION

1. Before you can run Henry's original version of the program, you need to modify your copy of the batch script file `my_number.bsub` to create a version that's uniquely yours. Using your preferred Unix text editor (whether `nano`, `pico`, `vim`, `vi`, `emacs` or whatever), edit your copy of `my_number.bsub`. For example, if you're using `nano`, then the edit command would be:
   ```
   % nano my_number.bsub
   ```
   This command means: "Edit the text in the file named `my_number.bsub` that's in my current working directory, using the text editor program named `nano`." If you need help using `nano`, please send e-mail to `support@oscer.ou.edu`.

2. In `nano`, notice the little help messages at the bottom of the screen:
   ```
   ^G Get Help ^O WriteOut ^R Read File ^Y Prev Pg  ^K Cut Text   ^C Cur Pos
   ^X Exit     ^J Justify  ^W Where is  ^V Next Pg  ^U UnCut Text ^T To Spell
   ```
   For example, consider `^W Where is`
   This means that you should press **Ctrl**-**W** (the caret ˆ indicates the **Ctrl** key) to search for a particular string of characters. Another example: `^C Cur Pos` is short for "Cursor Position" and causes `nano` to tell you what line number the cursor is located at. Another example: `^K Cut Text` means "delete the line that the cursor is currently on."

3. Using the text editor, make the following changes to `my_number.bsub`:
   (a) Everywhere throughout the file, change
   `yourusername`
   to your user name (which might be of the form `mpi###`, or perhaps is based on your name). **THIS IS EXTREMELY IMPORTANT!**
   (b) Everywhere throughout the file, change
   `youremailaddress@yourinstitution.edu`
   to your e-mail address. **THIS IS EXTREMELY IMPORTANT!**

4. Every few minutes while you're editing, you should save the work that you've done so far, in case your work is interrupted by a computer crashing. In `nano`, type **Ctrl**–**O** (the letter oh), at which point `nano` will ask you, near the bottom of the screen:
   ```
   File Name to write : my_number.bsub
   ```
   That is, `nano` wants to know what filename to save the edited text into, with a default filename of `my_number.bsub`). Press **Enter** to save to the default filename `my_number.bsub`.

5. The lines of text in the batch script file `my_number.bsub` should be less than 80 characters long, and ideally at most 72 characters long. (Your PuTTY window should be 80 characters wide.)

6. Some text editors, including `nano`, try to help keep text lines short, by breaking a long line into multiple short lines. For example, `nano` might break a line like

   ```
   #BSUB -o /home/yourusername/SIPE2009_exercises/Intro/C/my_number_%J_stdout.txt
   ```

   into two separate lines:

   ```
   #BSUB -o
   /home/yourusername/SIPE2009_exercises/Intro/C/my_number_%J_stdout.txt
   ```

   That is, `nano` automatically puts a carriage return when the line starts getting too long for `nano`'s taste. Unfortunately, the batch scheduler (LSF, for Load Share Facility) will consider this to be an error. Why? Because LSF cannot allow an individual batch directive — that is, a line starting with `#BSUB` — to use more than one line. So, you'll need to correct any such occurrences.

7. After you've finished editing, go back up to the top of the batch script file, and **CAREFULLY READ THE ENTIRE BATCH SCRIPT FILE FROM START TO FINISH.** This will give you a much clearer understanding of what batch computing is and how it works.

8. Understanding batch computing:
   As an analogy, imagine that you're at a football game and you want a drink. You get up and walk to the concession stand. If there are a lot of people at the concession stand, then you're going to have to wait a while before a server serves you, but if you're the only person in line, or more generally if there are at least as many servers behind the counter as customers lined up to buy, then you'll be served quickly.
   Batch computing is analogous, except that instead of food and drink, you and the other users want your jobs to be run, and instead of food servers, the servers are computers that can run jobs.
   In the case of Sooner, OSCER's big Linux cluster supercomputer, the number of users is roughly 500, and the number of servers (computers) is also roughly 500 — but most users want to use many computers at the same time.
   The only way to make this work is for a computer program known as a *scheduler* — in this case, LSF — to decide whose jobs run on which computers, and when.
   Compare getting food at a football game to getting food at home, where you just walk up to your fridge or cupboard or whatever, and take out what you want. But if you've got hundreds of people getting food, that method won't work: it doesn't *scale* to hundreds of people sharing one source of food, because you can't fit all of them in front of the one fridge; instead, everyone has to wait their turn at the counter, and work with a server to get served.
   Likewise with computing: your normal way of interacting with your laptop won't work when hundreds of people are sharing one source of computing.

9. After you've finished reading the batch script file, exit the text editor. For example, in `nano`, type `Ctrl`–`X`. If you have made any changes since the last time you typed `Ctrl`–`O`, then `nano` will ask you, near the bottom of the screen,

   ```
   Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ?
   ```

   To save your most recent changes to the file (which is probably what you want to do), press the `Y` key; to avoid saving your most recent changes, press the `N` key. After that, `nano` will behave the same as if you had typed `Ctrl`–`O`.

V. LOOK AT, MAKE (COMPILE) AND RUN THE ORIGINAL VERSION OF THE PROGRAM

1. For your own understanding, look at the contents of the source file:

   ```
   % cat  my_number.c
   ```

   OR:

   ```
   % cat  my_number.f90
   ```

   This command means: "Output the contents of the text file named `my_number.c` (or `my_number.f90`) to the terminal screen." **NOTICE** that the command to output the contents of a text file to the terminal screen **without** using the `more` command is `cat`, which is short for "concatenate," a word that means "output one text file after another in sequence." The output of the `cat` command goes to the terminal screen, and in this case, we are only concatenating a single text file, so we're simply outputting the text file's contents to the terminal screen.

   If you're using `PuTTY` as your SSH client, and the contents of the file exceeds the height of the `PuTTY` window, then you can scroll up or down using the scrollbar on the right side of the window; most other SSH clients have similar capability.

2. For your own understanding, look at the contents of the makefile:

   ```
   % cat  makefile
   ```

3. *Make* (compile) the *executable* program for Henry's original version of `my_number.c` (or `my_number.f90`):

   ```
   % make  my_number
   gcc -O -c my_number.c
   gcc -O -o my_number my_number.o
   ```

   (It could be the case that the compiler is `gfortran` and the source file is `my_number.f90`.)

   NOTICE:

   - In the `make` command, the *command line argument* `my_number` is the name of the *executable* (the file that can actually be run) that you are making.
   - The `make` command runs the C compiler `gcc` (or the Fortran90 compiler `gfortran`) to compile the source file named `my_number.c` (or `my_number.f90`). In the compile command, the *command line option*

     `-o  my_number`

     indicates that `my_number` is to be the name of the executable; if that option had been left out, then by default the name of the executable would be `a.out` ("the output of the assembler"), **WHICH WOULD BE BAD**, because then the executable's filename wouldn't explain the executable's purpose.

4. Submit the batch script file `my_number.bsub` to the batch scheduler:

   ```
   % bsub   <   my_number.bsub
   ```

   NOTICE the less than symbol `<` which is **EXTREMELY IMPORTANT.**

   You should get back output something like this:

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

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

5. Check the status of your batch job:

   ```
   % bjobs
   ```

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

   ```
   No unfinished job found
   ```

   OR:

   ```
   JOBID  USER         STAT  QUEUE FROM_HOST EXEC_HOST JOB_NAME  SUBMIT_TIME
   ###### yourusername PEND  debug sooner1             my_number Jan 23 21:56
   ```

   where `######` is replaced by a batch job ID number, and `yourusername` is replaced by your user name.

6. You may need to check the status of your batch job repeatedly, using the `bjobs` command, until it runs to completion. You'll know that it has finished running when it no longer appears in the list of your batch jobs (in which case it may say "No unfinished job found").

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

   ```
   % ls -ltr
   ```

   NOTICE that the command is "ell ess space hyphen ell tee are" — that is, small-L small-S blank hyphen small-L small-T small-R — rather than "one ess" or "one tee are" and that `ls` is short for "list" and `-ltr` is short for "long detailed listing, sorted by time of most recent modification, in reverse order so that the most recently modified file is at the bottom."
   Using this command, you should see files named
   `my_number_######_stdout.txt`
   and
   `my_number_######_stderr.txt`,
   (where `######` is replaced by the batch job ID). These files should contain the output of `my_number`.
   Ideally, the size of `my_number_######_stderr.txt` should be zero.

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

   ```
   % cat  my_number_######_stdout.txt
   ```

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

9. 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
   my_number.c
   my_number.bsub
   my_number_######_stdout.txt
   my_number_######_stderr.txt
   ```

10. Congratulations! You've just run your first batch job.

VI. EDIT THE C SOURCE FILE TO CREATE YOUR OWN UNIQUE VERSION

1. Now that you've run Henry's original version of the program, it's time to modify your copy of the source file `my_number.c` (or `my_number.f90`) to create a version that's uniquely yours. Using your preferred Unix text editor (whether `nano`, `pico`, `vim`, `vi`, `emacs` or whatever), edit your copy of `my_number.c` (or `my_number.f90`). For example, if you're using `nano`, then the edit command would be:

   ```
   % nano  my_number.c
   ```

   OR

   ```
   % nano  my_number.f90
   ```

2. Using the text editor, make the following changes to `my_number.c` (or `my_number.f90`):
   (a) In the *declaration section*, change the constant values assigned to `minimum_number`, `maximum_number`, `close_distance` and `computers_number`. You may select any **integer** values that you want, as long as they are different from 1, 5, 10 and 1 respectively, and `minimum_number < computers_number < maximum_number`, and they are sufficiently spread out that you can actually do the runs properly.
   (b) In the *execution section* (also known as the *body* of the program), change the following sequences of character text to your own words:
      i. `Hey!`
      ii. `That's amazing!`
      iii. `Close, but no cigar.`
      iv. `Bzzzt!  Not even close.`

3. Every few minutes while you're editing, you should save the work that you've done so far, in case your work is interrupted by a computer crashing. In `nano`, type **`Ctrl`**-**`O`** (the letter oh), at which point `nano` will ask you, near the bottom of the screen:

   ```
   File Name to write : my_number.c
   ```

   OR:

   ```
   File Name to write : my_number.f90
   ```

   That is, `nano` wants to know what filename to save the edited text into, with a default filename of `my_number.c` (or `my_number.f90`). Press **`Enter`** to save to the default filename `my_number.c` (or `my_number.f90`).

4. A *character string literal constant*, also known as a *character string literal* or a *string literal* for short, is a sequence of characters between a pair of double quotes. For example, in the C `printf` statement

   ```
   printf("This is a printf statement.\n");
   ```

   the following is a string literal:

   ```
   "This is a printf statement.\n"
   ```

   Likewise, in the Fortran90 `PRINT` statement

   ```
   PRINT *, "This is a PRINT statement."
   ```

   the following is a string literal:

   ```
   "This is a PRINT statement."
   ```

   We say that the pair of double quotes *delimits* the sequence of characters in the string literal. Note that, in C, the `\n` at the end of the string literal tells the program to print a carriage return (also known as a *newline*) at the end of the line of output text. (In Fortran90, the carriage return is implied by the end of the `PRINT` statement.)

5. The lines of text in the source file `my_number.c` (or `my_number.f90`) should be less than 80 characters long, and ideally no more than 72 characters long. (Your PuTTY window should be 80 characters wide.)

6. Some text editors, including `nano`, try to help keep text lines short, by breaking a long line into multiple short lines. For example, `nano` might break a line like

   ```
     printf("This is a long line and nano will probably break part of it off.\n");
   ```

   into two separate lines:

   ```
     printf("This is a long line and nano will probably
   break part of it off.\n");
   ```

   That is, `nano` automatically puts a carriage return when the line starts getting too long for `nano`'s taste. Unfortunately, the C compiler (or the Fortran90 compiler) will consider this to be an error. Why? Because C (or Fortran90) cannot allow an individual string literal to use more than one line (in Fortran90, there's a goofy way to do it, but it's bad practice). So, the correct way to write the above example is:

   ```
     printf("This is a long line and nano will probably");
     printf(" break part of it off.\n");
   ```

   OR:

   ```
     PRINT *, "This is a long line and nano will probably"
     PRINT *, " break part of it off."
   ```

7. Like the lines of source text, the lines of output text should be less than 80 characters long, and ideally no more than 72 characters long. You can break a long line of output text into shorter pieces by making it into two `printf` (or `PRINT`) statements. For example:

   ```
   printf("Why you big old stinker!  That's not between %d and %d!\n",
       minimum_number, maximum_number);
   ```

   This single `printf` statement can be converted into two `printf` statements, like so:

   ```
   printf("Why you big old stinker!  That's not between\n");
   printf("  %d and %d!\n", minimum_number, maximum_number);
   ```

   Or, in Fortran90:

   ```
   PRINT *, "Why you big old stinker!  That's not between"
   PRINT *, minimum_number, " and ", maximum_number, "!"
   ```

8. After you've finished editing, exit the text editor. For example, in `nano`, type **Ctrl**-**X**. If you have made any changes since the last time you typed **Ctrl**-**O**, then `nano` will ask you, near the bottom of the screen,

   ```
   Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ?
   ```

   To save your most recent changes to the file (which is probably what you want to do), press the **Y** key; to avoid saving your most recent changes, press the **N** key. After that, `nano` will behave the same as if you had typed **Ctrl**-**O**.

9. Edit the input file `my_number_input.txt` to replace the original input values with input values relevant to your new unique version of the program, specifically:

   (a) an integer value less than your value for `minimum_number`

   (b) an integer value greater than your value for `maximum_number`

   (c) an integer value between your value for `minimum_number` and your value for `maximum_number` (inclusive), but far from your value for `computers_number`

   (d) an integer value close to your value for `computers_number` (that is, within your value for `close_distance` of your value for `computers_number`)

   (e) your value for `computers_number`

VII. MAKE (COMPILE), RUN AND DEBUG YOUR OWN UNIQUE VERSION

1. Make (compile) your own unique version of the executable program:

   ```
   % make  my_number
   gcc -O -c my_number.c
   gcc -O -o my_number my_number.o
   ```

   (It could be the case that the compiler is `gfortran` and the source file is `my_number.f90`.)

2. If the program doesn't compile, then you'll need to edit it and figure out where things went wrong. In the worst case, if you're totally stumped, then copy the original from Henry's directory again, and start editing from the beginning.

3. Submit the batch script file `my_number.bsub` to the batch scheduler:

   ```
   % bsub   <   my_number.bsub
   ```

   NOTICE the less than symbol `<` which is **EXTREMELY IMPORTANT.**
   You should get back output something like this:

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

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

4. Check the status of your batch job:

   ```
   % bjobs
   ```

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

   ```
   No unfinished job found
   ```

   OR:

   ```
   JOBID  USER         STAT  QUEUE FROM_HOST EXEC_HOST JOB_NAME  SUBMIT_TIME
   ###### yourusername PEND  debug sooner1             my_number Jan 23 21:56
   ```

   where `######` is replaced by a batch job ID number, and `yourusername` is replaced by your user name.

5. You may need to check the status of your batch job repeatedly, using the `bjobs` command, until it runs to completion. You'll know that it has finished running when it no longer appears in the list of your batch jobs (in which case it may say "No unfinished job found").

6. Find the *standard output* and *standard error* files from your job:

   ```
   % ls -ltr
   ```

   NOTICE that the command is "ell ess space hyphen ell tee are" — that is, small-L small-S blank hyphen small-L small-T small-R — rather than "one ess" or "one tee are" and that `ls` is short for "list" and `-ltr` is short for "long detailed listing, sorted by time of most recent modification, in reverse order so that the most recently modified is at the bottom."
   Using this command, you should see files named
   `my_number_######_stdout.txt`
   and
   `my_number_######_stderr.txt`,
   (where `######` is replaced by the batch job ID). These files should contain the output of `my_number`.
   Ideally, the size of `my_number_######_stderr.txt` should be zero.

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

   ```
   % cat  my_number_######_stdout.txt
   ```

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

8. Congratulations! You've just completed this exercise.