David Cavaceci

Meteo 496

September 11, 2009

**Report: Fourth Milestone**

|  |  |  |
| --- | --- | --- |
| **Milestone** | **Task** | **Code** |
| *IV* | *Write a Fortran program to apply statistical analysis to the collected data* | ***Program****: ‘tdata’*  ***Module****: ‘my\_temps’*  ***Subroutines****: ‘file2array’, ‘array2screen’, ‘tavgnorms’, ‘high\_norm\_runs’, ‘low\_norm\_runs’, ‘avg\_norm\_runs’* |

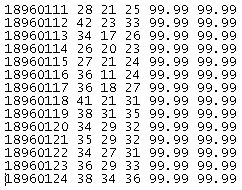
The fourth objective of the project concerned statistical analysis of the collected State College temperature data. Fortran is used instead of Perl for this objective, due to Fortran’s supreme computational speed and ability to handle very large data sets. Because the I/O formatting for Fortran is quite cryptic and different from that of Perl, a special module from the CPAN archive was utilized in the Perl scraping code in order to prepare the data for input to Fortran. The *Fortran::Format* module, explained in the previous report on milestones one through three, treated the data to organize it into a Fortran-friendly I/O format. The figure below displays the output of the Perl code in proper Fortran format, and the following table explains the variables provided in each column.

A B C D E F G H I J K L M N



|  |  |
| --- | --- |
| A | *Date* |
| B | *High temperature* |
| C | *Low temperature* |
| D | *Average temperature* |
| E | *Max-year normal max temperature* |
| F | *­Thirty-year normal max temperature* |
| G | *Ten-year normal max temperature* |
| H | *Max-year normal min temperature* |
| I | *Thirty-year normal min temperature* |
| J | *Ten-year normal min temperature* |
| K | *Snow depth* |
| L | *Amount of rain in inches* |
| M | *Amount of snow in inches* |
| N | *Heating degree days* |

The diagnostics chosen to use with the collected data includes the daily high, low, and average temperatures, and the thirty-year maximum, minimum, and average normal temperatures. Because of this, the program also created an output file containing only these data, which is displayed below.



The output above contains all needed temperature data for the State College climatology analysis in the correct format. Note that the output contains only five columns of data in addition to the dates column. This is because the average normal temperature data was not available for collection, and therefore must be manually computed later.

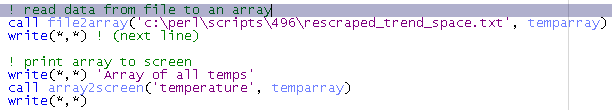
Moving on to the Fortran program, we start by looking at the flow of the program *tdata*. The program utilizes the *my\_*temps module, which allows the code to be more readable and understandable to outside parties. The module contains several subroutines, or functions, which are called by the main program during its execution to carry out the various tasks. Below we see the compressed view of the full code, indicating that that program will use one module.



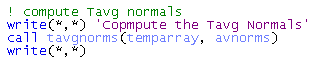
Since the script is over 280 lines long and the clip above displays only 6 lines, we know that the program and its underlying module are both fairly detailed. Extending the program *tdata*, we can see the definition of the module to be used and other variables not defined within the module itself.

C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

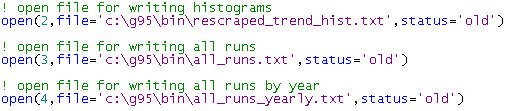
The line *use my\_temps* tells the program which module to include while executing. The following lines define the variables *avnorms* and *temparray*, both of which are of allocatable size. This method of defining an array allows the program to refrain from allocating memory for storage until a later time. This is beneficial when the desired size of the array is unknown until a preceding process is completed. In the case of my project, the size of the arrays defined above will not be known until the data file is scanned and the total number of days determined. After these declarations, the data can be read-in to the program, stored to an array, and printed to the screen for verification.



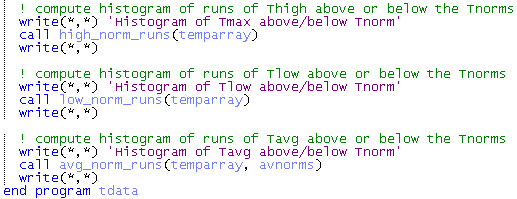
In the two blocks above, a comment line describes the process about to begin, and a descriptive line is printed to the terminal for progress and debugging purposes. The *call* command tells the program to access the defined subroutine (found within the module used by the program) with the given arguments. The *file2array* subroutine passes the location of the data file and the array *temparray*, which we declared at the beginning of the program. As we will see later, the arguments passed from the program to the subroutine depend on what the subroutine is expecting. In this case, *file2array* is expecting the location of the file to be read and an allocatable array in which the read data can be stored. Following this syntax, the subroutine *array2screen* is passed what it expects, the scalar value *temperature* and the array now storing the data. After these subroutines are complete, the data will have been read-in and printed to the terminal window. Since the thirty-year **average** normal temperatures were not collected, the next step in the program is computing these values manually. This step is illustrated below.



Above it can be seen that after displaying the stored data, the missing data needed (thirty-year average normal temperatures) is computed by calling the subroutine *tavgnorms*. This subroutine will be examined closer in the pages to come. After these normals are calculated, we have all the data needed to determine the frequency of various run lengths of high, low, and average daily temperatures exceeding or falling short of the thirty-year normals. Before these calculations are executed, however, three files are opened for writing the results to.

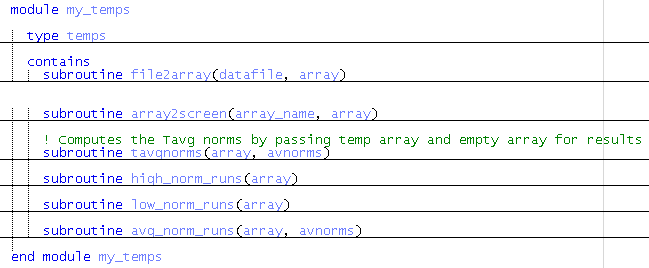


The thee files opened will be used for writing the histogram of runs above or below the corresponding normal for all years, the runs themselves for all years, and the runs categorized by each year. The difference in output came as a result of changing statistical analysis goals and debugging needs as the project progressed. Now the business at hand can be initiated, via the subroutines accessed in the illustration below.



The final three steps within program *tdata* are the calling of the subroutines *high\_norm\_runs*, *low\_norm\_runs*, and *avg\_norm\_runs*. As the names suggest, these functions will pass the stored data to a subroutine, and compute the histogram of the variable of interest above and below its respective normal (they are actually separate histograms, though each are computed simultaneously). The results of these computations are written to the screen and also to the output files defined in the last step. *Avg\_norm\_runs* differs from the other two subroutines in regard to its arguments, because the average normal temperature data was calculated manually, and thus must also be passed in for comparison.

After the histogram of average temperature runs has been computed, the program will end. The simplicity and logical flow of the program block allows a reader to easily understand the goals of the script, as well as the expectations of the coder. This aids the debugging process and allows the finer details to be saved for when they are needed. Select details, found within the module *my\_temps*, will now be reflected upon. Below, the semi-compressed view of the module is illustrated.

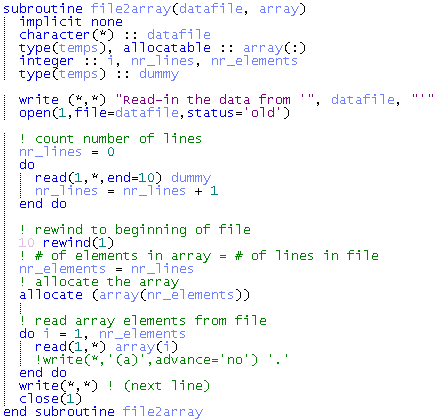


The first action by the program is to include the module *my\_temps* in the execution. The module is seen above to include the declaration of derived type variable *temps*, which will be explained later. The subroutines called by the program are also contained within the module, as is seen with the command *contains* after the derived type declaration. The order of these constituents is important to ensure that processes using data computed or stored to an array prior to the process are located where the program thinks they are. Now let’s take a look at the derived type variable, *temps.*

C:\DOCUME~1\Owner\LOCALS~1\Temp\msohtmlclip1\01\clip_image001.png

In Fortran, variables may contain data only of the same type. That is, a scalar variable defined as type *character* may only contain characters; an array of type *real* may only contain real numbers, et cetera. The derived type declaration allows the coder to define a new type of variable, which can contain any mixture of data types inherent to the Fortran language. In this case, the custom variable *temps* contains four integer variables, and two real variables, corresponding to the date, the high, low, and average daily temperatures, and the high and low thirty-year normal temperatures for that day.

In Fortran, subroutines are defined with *subroutine name(arguments)*, and are closed with the statement *end subroutine name*. Any data found outside the subroutine which will be manipulated within it must be passed-in via the *arguments*. The arguments defined here determine the requirements of what must be passed-in for the subroutine execution to be successful. Note that the names of the variables passed to the subroutine need not match the names used as arguments within the subroutine definition, but must be identical in number and type. Additional variables within the subroutine (i.e. step counters, dummy arrays, etc.) are defined within the subroutine before they are called upon for use. Below, the subroutine *file2array* is examined.



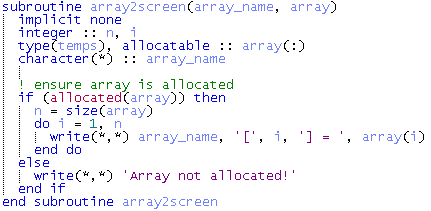
From the image above, we see that the subroutine requires the passing of two variables, *datafile* and *array* before execution can begin. The types of these variables must be defined within the subroutine, such that improper data passed-in will be recognized and an error message given. This further explains why the **names** of the arguments passed from the external program are not required to match the argument names used by the subroutine, while the type and number of arguments must be identical. In this case, the subroutine argument *datafile* is defined to be of type *character* and of length to be determined by the variable value passed-in. Note that the value of the character argument passed to the subroutine in the program is *'c:\perl\scripts\496\rescraped\_trend\_space.txt',* which satisfies the requirements. Additionally, the argument *array* is defined to be of type *temps* and of allocatable length. This makes sense because the argument passed-in (*temparray*) is an array storing variables of type *temps*, though the length of the array is not yet known by the subroutine.

The *open(handle, file, status)* statement opens the data file to be read-in by the subroutine. The *handle “1”* is a required syntax by Fortran and associates a handle to the opened file for later access. The *file* is defined as variable *filename* here, which is a character argument passed in from the program, and the, *status* value is defined as *old* so that Fortran will know to write over any preexisting text in the file.

The “count number of lines” block is used to determine how many lines of data are within the text file. This is important for allocating arrays and determining other variables, in this case the number of days in the period of record. Within this block we encounter the first instance of flow control within the script. The code *do … end do* tell Fortran to execute the lines of code within the do block while some condition is true. Since no conditions are explicitly defined, the statement *read* will be controlling the flow. The *read* statement tells Fortran to read the first line in the file with handle 1, following the standard Fortran format, and to jump to line-handle 10 once the reading is complete. The variable *dummy* assigned at the end of this read statement tells Fortran to store the read data to this array. Looking back at the data file being read and the declaration of derived data type *temps*, it can be seen that the definition of type *temps* was determined by the context of the data file. Thus, since *dummy* was defined to be a variable of type *temps*, reading the data and storing it to the array is simple. The integer variable *nr\_lines* is defined to value 0 before entering the do loop, and is incremented with each pass. In this way, *nr\_lines* will contain the value of the total number of lines in the file upon exiting the loop (because Fortran reads files one line at a time).

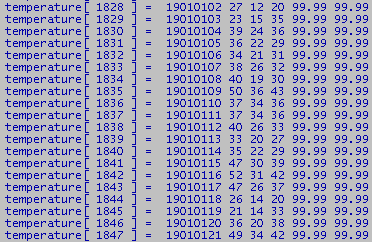
The line beginning with the number 10 is the “line-handle” to which the *end* argument within the read statement is referring. Once the read is complete, Fortran jumps to this line to begin flowing through the program again. In this case, the file is rewound, which is a leftover command from the early days of Fortran when cassettes of data had to be physically rewound in order to access them again. Today, this step merely allows us to access the file again in the future, by focusing Fortran’s attention within the file back at the beginning.

After rewinding, the data array is allocated to the size of *nr\_lines* which just finished determining the number of lines in the file. In our case, this equals the number of elements, so the ­*nr\_lines = nr\_elements* step is unnecessary and simply a practice of proper coding methodology. Once this is complete, the data can be read line-by-line and stored into the data array in the same manner as the first time around. This time however, initial and final conditions are set on the do loop, utilizing the step variable *i*. This is necessary so that the data array *array*, which has been allocated to the correct size, can receive one line of data for each index in the array. Upon completion, the *file2array* subroutine is complete, and the data has been read-in and stored in the data array *array*. When passed back to the program, this data array is transferred to *temparray* because that was the variable name passed-in to the subroutine.

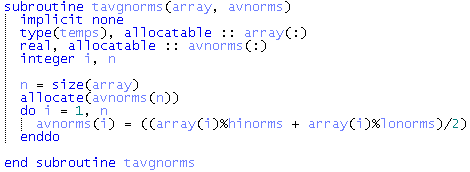


The block above displays the subroutine *array2screen*, which receives arguments *array\_name* and *array*. The *array* argument is once again passed the *temparray* array from the program, but this time it is allocated and filled with the read-in data. This is allowed because *temparray* was defined at the head of the program and not within a subroutine, as was allocated in the previous subroutine. This array is assigned to the array *array* in this subroutine, which is allocatable because the subroutine is not yet aware of the size of array *temparray*. The *array\_name* argument is defined as a character variable, the size of which being determined by the passed-in value, *temperature*.

Since the array passed to argument *array* was previously allocated, the subroutine sees its argument as allocated. Thus, the *if* flow control is included to ensure that the data has been passed in successfully. If it hasn’t, an error message is displayed (as seen under the *else* block). If *array* has been found to be allocated, the integer variable *n* is assigned the value of the size of the array, which we know is the total number of days in the period. As a result, the following *do* loop will execute for all values of integer variable *I* (defined again here because it was previously defined **locally**) between 1 and the size of the array. In other words, “Do the following for each day in the period of record.” The *write* statement to be executed for each pass prints the array name (*temperature*, passed in from the program), the index number (1 to the size of the array), and the value of the array for that index (the temperature data for that day) to the screen for each day. What we see at the terminal during this process is the addition of lines of data containing the name of the array, the index number, and the data values for each date (See below).



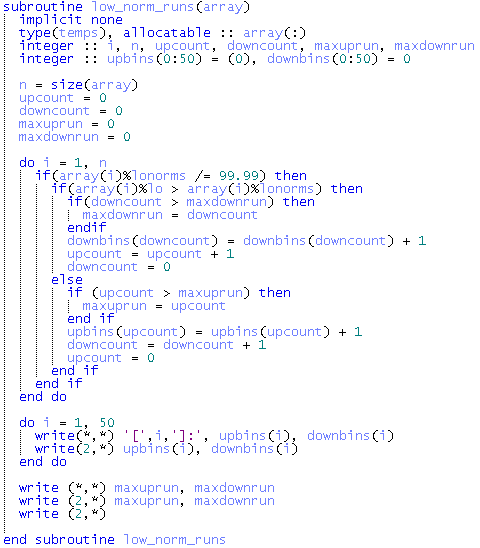
At this point, the data has been read-in and stored to an array, and the contents of the array have been printed to the screen for verification. Next, the average temperature normals must be computed to allow for daily average temperature runs to be compared with what is expected. This is done via the *tavgnorms* subroutine, which is illustrated in the figure below.



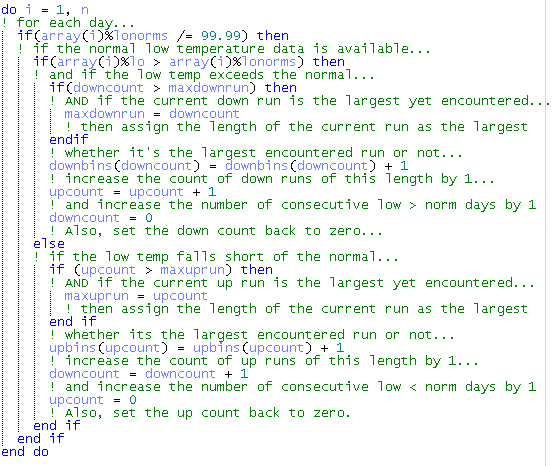
As in the previous subroutines, the argument *array* is passed to the subroutine, allowing it to access the temperature data. Also, the integer variables *I* and *n* are declared to be used as step variables. Unique to this subroutine is the *avnorms* variable, which was declared at the head of the main program so that it could be retrieved back from the subroutine after storage. That is, we know comparison between the daily average temperatures and the average temperature normals will be done later, so we must allow for future access to the array holding the manually computed average normal temperatures.

Following the patterns seen previously, the step variable *I* controls the flow of the do loop from 1 to *n*, which is assigned the value of the size of the data array (the number of days). Also, the array to be used for storing the computed normal average temperatures, *avnorms*, is allocated to the size of n. The human-language version of the do block is then, “For **this** date, sum the high and low normal temperatures, divide the result by 2, store the average normal temperature in **this** index of array *avnorms*, and then move on to the **next** date. This results in array *avnorms* containing the average normal temperatures for each date in the record.

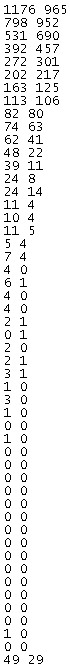
The final process to be completed in the script is to determine the lengths of all high, low, and average daily temperature runs above or below their respective normals, and create a histogram of the occurrence of these lengths. This is accomplished with separate subroutines for the high, low, and average temperatures, but the process is the same for each. It should be noted that these could be simplified into one subroutine with the use of more advanced array-accessing techniques, though time restraints prohibited such simplification within this project. As a representative for each of the histogram computations, the low temperature runs subroutine is illustrated below.



The only argument passed-in to this subroutine is the data array *temparray*, and this array is assigned to array *array* within the subroutine (just as in previous subroutines). Integer arrays *upbins* and *downbins* are defined within the declaration block to be of size 51 (0 to 50). These will be the holders of the histograms of low temperature runs above and below the normal. Initially, these values are all 0, but each time a run length of **x** is realized, the corresponding array is incremented by one in index **x** (In other words, is the low temperature exceeds the normal low for 5 days, *upbins(****x)*** is increased by one). It is clear, then, that at the end of the looping test, the bins arrays will contain values reflecting the occurrence of each run length. Below, an explanation of the do loop is provided within the code block.



The completion of this loop results in two arrays, *upbins* and *downbins*, containing a histogram of low temperature runs above and below the normal low, respectively. These histograms are then written both to the screen and to an output file via another do block, just after the calculation block. Also, the largest runs above and below the normal is written to both media. The low temperature histograms written to the output text file are displayed below, with the first row representing runs of length 1, and incrementing by 1 all the way down through 50. The maximum run lengths are written after the histograms, even though these can be deduced easily from the histograms.



The full program (*ms\_iv.f95*) is available at *www.personal.psu.edu/dac5039*