Lab 6: Problem Solving with Arrays: Statistical Measurements

Learning Outcomes

- Using arrays to store homogeneous data
- Looping over arrays
- Implementing functions with array parameters
- Implementing basic algorithms [maximum, minimum, mean, variance, standard deviation] on array-based data.

Task: Computing Summary Statistics

Analyzing data collected from engineering experiments is an important part of evaluating these experiments. This analysis ranges from simple computations on the data, such as calculating the average value, to more complicated analyses. Many of the computations or measurements using data are statistical measurements because they have statistical properties that change from one set of data to another. For example, $\sin(60^\circ)$ is an exact value that is the same value every time we compute it, but the number of miles to the liter that we get with our car is a statistical measurement, because it varies depending on parameters such as the temperature, the speed that the car is driven, the type of road, and whether you're driving on a freeway or in a city.

Rather than evaluating engineering data, you'll evaluate the performance of students in certain courses. The number of students enrolled in these courses fluctuates but can never be more than 1000. That is, the maximum enrollment per course is 1000. Conveniently, grades for these courses are stored in text files. Inconveniently, the number of enrolled students in a course is unknown and can only be determined by the number of grades in the corresponding grades file. Unconventionally, the grades are in range from 0 to 1000, respectively.

When evaluating a set of data, we often compute maximum, minimum, average, and median values. In this lab, you'll develop functions that can be used to compute these values using an array as input. These functions will be useful in many of the programs that you'll develop throughout your career.

Maximum

A function that returns the maximum value in an array of `int` values was presented in the lecture handout. Unlike that function, the maximum function you'll author must return the subscript of the element in the array with the maximum value. The function is declared as:

```c
int max_index(double const arr[], int size);
```

Minimum

A function that returns the minimum value in an array of `int` values was also presented in the lecture handout. Unlike that function, the minimum function you'll author must return the subscript of the element in the array with the minimum value. The function is declared as:

```c
int min_index(double const arr[], int size);
```

Average

The Greek symbol μ (mu) is used to represent the average or *mean* value. An equation, which uses summation notation, is as follows:

$$\mu = \frac{\sum_{k=0}^{n-1} x_k}{n} \tag{1}$$

where x represents an array of n values with

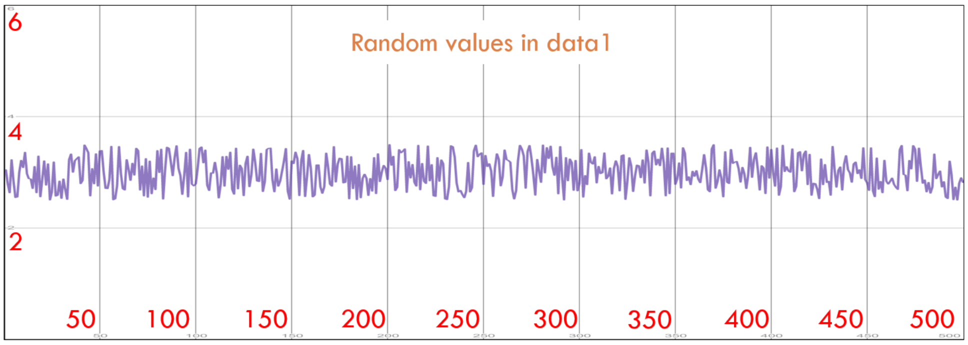
$$\sum_{k=0}^{n-1} x_k = x_0 + x_1 + x_2 + \cdots + x_{n-1}$$

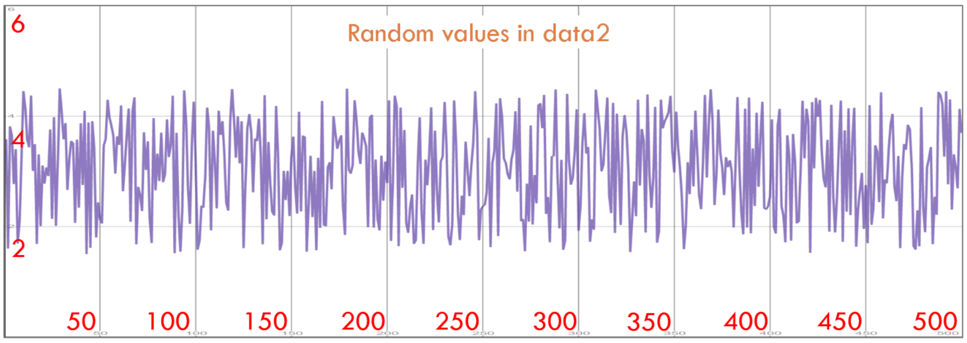
The average of a set of values is always a floating-point value, even if all the data values are integers. An algorithm to accumulate the values of array elements was described in the lecture [on arrays]. You'll have to adapt that algorithm to return the mean value of an array of `double` values. The function is declared as:

```c
double mean(double const arr[], int n);
```

Variance and Standard Deviation

One of the most important statistical measurements for a set of data is the variance. Before looking the mathematical definition for variance, let's develop an intuitive understanding. Consider 500 values in some arrays `data1` and `data2`, which are plotted below.





If you attempted to draw a horizontal line through the middle of the values in each plot, this line would be at approximately 3.0. Thus, both arrays have approximately the same average or mean value of 3.0. However, the data in the two arrays clearly have some distinguishing characteristics. The values in `data2` vary more from the mean, or deviate more from the mean value. The *variance* of a set of values is defined as the average squared deviation from the mean; the *standard deviation* is defined as the square root of the variance. Thus, the variance and the standard deviation of the values in `data2` are greater than the variance and standard deviation for the values in `data1`. Intuitively, the larger the variance [or the standard deviation], the further the values fluctuate around the mean value.

Mathematically, the variance is represented by σ^2, where σ is the lowercase Greek symbol sigma. The variance for a set of data values [which we assume are stored in an array x] can be computed using the following equation:

$$\sigma^2 = \frac{\sum_{k=0}^{n-1}(x_k - \mu)^2}{n-1} \tag{2}$$

This equation is a bit intimidating at first, but if you look at it closely, it becomes much simpler. The term $x_k - \mu$ is the difference between x_k and the mean or the deviation of x_k from the mean. This value is squared so that we always have a positive value. We then add the square deviations for all data values. This sum is then divided by $n-1$, which approximates an average. The definition of variance has two forms: the denominator of a *sample variance* is $n-1$, and denominator of a *population variance* is n. Most engineering applications use the sample variance, as shown in the Equation (2). Thus, Equation (2) computes the average squared deviation of the data from the mean. The standard deviation is defined to be the square root of the variance:

$$\sigma = \sqrt{\sigma^2} \tag{3}$$

Both variance and standard deviation are commonly used in analyzing engineering data, so a function must be declared for computing the variance and an other function for computing the standard deviation. Note that the function for computing the standard deviation references the `variance` function and the `variance` function references the `mean` function.

> ***Note that there must be at least two values in the array, or the `variance` function will attempt to divide by zero. You must check for this and return zero if there is only one element in the array. Otherwise, the auto grader will crash and compute a zero grade for your submission!!!***

The declarations of these functions will look like this:

```c
double variance(double const[], int size);
double std_dev(double const[], int size);
```

Reading data into array from `stdin`

Suppose that source file main.c contains array definition `int grades[MAX_STUDENT_COUNT];` with the macro declared as `#define MAX_STUDENT_COUNT 1000` . Further suppose, that we wish to implement a function `read_data` in this source file to read an unknown number of `int` values from standard input up to a maximum of `MAX_STUDENT_COUNT` number of values into the array `grades`. The source file with a `main` function that exercises function `read_data` is shown below:

```c
#include <stdio.h>

// declaration of funtion read_data ...
int read_data(double array[], int max_num_vales);
#define MAX_STUDENT_COUNT 1000
```

```c

int main(void) {
  // fill grades with values
  double grades[MAX_NUM_GRADES];
  int count = read_data(grades, MAX_STUDENT_COUNT);

  // print grades to standard output
  for (int i = 0; i < count; ++i) {
    printf("i: %.2f\n", i, grades[i]);
  }
  return 0;
}
```

Presumably, users would store their grade data in a text file which would then be redirected to standard input stream `stdin`. That is, if q.out is the executable program's name and grades were stored in text file grades.txt, the user would run the program like this:

```
$ ./q.out < grades.txt
```

causing function `read_data` to read the grades from the standard input stream into an array.

Using the framework provided for writing functions with array parameters, function `read_data` can be declared as

```c
/*
Read values of type int from standard input and assign them -
in sequence - to indexed variables in array.
Fill the array with no more than max_count number of int values or
until end-of-file is reached, whichever comes first.
Notice that array is NOT declared as a read-only array since we want
to assign or write int values to the array.
*/
int read_data(double array[], int max_count);
```

Parameter `max_count` represents the maximum number of grades that must be read. If the file redirected to `stdin` contains 5000 grades but parameter `max_count` is 100, then only the first 100 grades in the file must be copied to array `array`. In some cases, the file redirected to `stdin` will contain a smaller number of entries than `max_count`. For example, the file redirected to `stdin` might have only 100 grades while `max_count` is 1000. In this case, the function will read all 100 grades from the file and store these values in array `array`. In all cases, the function will return the number of grades read from `stdin` and stored in the array.

Using idioms developed throughout this semester, the definition of `read_data` is straightforward:

```c
/*
Here, the parameter array is non-const because the function's intention
is to copy double values from standard input to the array's index variables.
*/
int read_data(double array[], int max_array_size) {
  int i = 0;
  double num;
  while ( (scanf("%lf", &num) != EOF) && i < max_array_size ) {
    /*
    Rather than writing two separate statements:
    array[i] = num;
    ++i;
    we use the post-fix increment operator to assign the value read from
    standard input to the indexed variable array[i] and then let the
    postfix increment operator increment index i's value to i+1.
    */
    array[i++] = num;
  }

  /*
  i specifies the number of int values read from standard input -
  the only thing we care about is that a maximum of max_array_size
  number of elements have been read.
  */
  return i;
}
```

Implementation Details

Open a Window command prompt, change your directory to C:\sandbox [create the directory if it doesn't exist], create a sub-directory lab06 , and launch the Linux shell. Download driver source file qdriver.c and incomplete source and header files q.c and q.h, respectively.

Function declarations in q.h

Using Visual Code, open header file q.h and add file- and function-level documentation blocks and declarations of the necessary functions.

Do not include C standard library headers in q.h unless the function declarations in q.h rely on types declared in the C standard library. Why? Suppose you unnecessarily include header files in q.h and your clients in turn include q.h in their source files. When clients' source files are compiled, the preprocessor will include the unnecessary C standard header files into these source files by copying and pasting hundreds of lines from unused header files into source files. This will greatly increase compile times causing great annoyance to your clients. Instead, include any C standard header files required to define the functions directly in q.c.

In previous labs and assignments, it was not necessary to include q.h in q.c because you're defining functions in q.c that don't require calls to other functions being declared in q.h. However, you will now be required to include q.h in q.c since function `std_dev` references function `variance` which in turn references function `mean`.

Test your header file q.h by compiling (only) driver source file qdriver.c which includes q.h [download qdriver.c from the lab web page]:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c
qdriver.c -o qdriver.o
```

Stub functions in q.c

Use descriptions of summary statistics to devise complete algorithms that practices all the steps of the algorithm formulation process. Edit source file q.c by providing file-level and function-level [even though the function-level documentation is not graded] documentation blocks. Include necessary C standard library header file(s). Next, convert your previously constructed algorithms into definitions of functions `max_index`, `min_index`, `mean`, `variance`, `std_dev`, and `read_data` in file q.c.

Without completely defining these functions, you cannot successfully compile, link, and execute the program. This means that you cannot concentrate on implementing, testing, and verifying only one function at a time because you need all six to be implemented to just compile and link. It is also possible that you implement the first function incorrectly and transfer these errors to the other functions. Later, you're faced with the task of trying to debug multiple incorrect functions. Is there a better alternative?

The better alternative consists of implementing *stub functions*. A stub is a skeleton of a function that is called and immediately returns. It is syntactically correct - it takes the correct parameters and returns the proper values [although you may need to drop the -Werror flag to successfully compile in certain cases]. Here is an example of a stub function:

```
// return any value of type double to ensure that the definition is
// syntactically correct and will compile although with diagnostic
// warning messages because parameters arr and n are unused.
double mean(double const arr[], int n) {
  return 0.0;
}
```

Although a stub is a complete function, it does nothing other than to establish and verify the linkage between the caller and itself. But this is a very important part of coding, testing, and verifying a program. At this point, the program should be compiled, linked, and executed. Since you've only defined stub functions, parameters in these stub functions are unused causing the -Werror option to terminate compilation [with *unused parameter* error messages]. Therefore, with stub functions, you will need to temporarily drop the -Werror option to successfully compile [but with warnings].

Compile (only) the driver source file qdriver.c:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror=vla
-c qdriver.c -o qdriver.o
```

Compile (only) your source file q.c:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror=vla
-c q.c -o q.o
```

Link both object files and C standard library functions into executable file q.out:

```
$ gcc qdriver.o q.o -lm -o q.out
```

Chances are that you'll find some problems related to syntax such as missing semicolons or errors between function prototype declarations and the function definitions. Before you continue with the program, you should correct these problems.

Create a file, say test.txt containing a few grades. Test your executable like this:

```
$ ./q.out < test.txt
```

Since you only have defined a stub function `read_data`, function `main` should report the following message:

```
There are no grades to process!!!
Shutting program down ...

```

Implementation and testing

At this point, you've qdriver.c and you've defined stub functions in q.c. Begin by implementing function `read_data`. It is important to implement `read_data` first because without copying grades from a data file to an array, it is not possible to test the other functions which process array values.

Before implementing other functions, you must verify that function `read_data` behaves correctly. To test a function, you should know what input is given to the function and the expected output from the function. Begin by constructing a data file test.txt with a small number of easy to work with values such as:

```
9 6 19 2 5 11

```

Although grades can contain fractional values, it always more convenient to use integer values for hand calculations. Function `read_data` can be easily tested by writing temporary `printf` statements in qdriver.c that print the return value of function `read_data` and the contents of array `grades`. For example, the following code fragment can be temporarily inserted in file qdriver.c after the call to function `read_data`:

```c
printf("npts: %d\n", npts);
for (int i = 0; i < npts; ++i) {
  printf("i: %.2f\n", i, grades[i]);
}
```

You can verify the correctness of `read_data` by comparing the output written to standard output with the contents of test.txt. Verify further by adding and removing values in test.txt. Make sure to test edge cases such as a single value or a completely empty data file.

After completely verifying function `read_data`, you should perform hand calculations on the data set in file test.txt to compute mean, variance, and standard deviation. Maximum and minimum indices can be easily determined by visually inspecting the data. Next, you should work on a second function, say `max_index` and verify the correctness of your implementation by comparing the program's output with your hand calculations. Thoroughly test and verify the behavior of this second function before moving on to a third function. Make sure to implement and verify function `mean` before implementing function `variance`. This is so because function `variance` requires the data set's mean to compute its variance. For the same reason, you must implement and verify function `variance` before implementing function `std_dev`.

You can't yet assume your implementations are completely verified. You'll need to generate a new data set, perform hand calculations, and compare the results generated by the program with the hand derived results. Although the results returned by your function definitions agree with the previous hand calculations, testing isn't complete without verifying the calculations at boundary points. For this program, the test consists of checking the calculation with a data set consisting of a single value. Test with a data set containing the same values, such as all 0s; a data set consisting of all 100s. Another simple test is to use five 0s and five 100s.

Compiling, linking, and testing

Compile (only) the driver source file qdriver.c:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror=vla
-c qdriver.c -o qdriver.o
```

Compile (only) your source file q.c [which must include q.h]:

```
$ gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror=vla
-c q.c -o q.o
```

Link both object files and C standard library functions into executable file q.out:

```
$ gcc qdriver.o q.o -lm -o q.out
```

Create a file, say test.txt containing a few grades. Test your executable like this:

```
$ ./q.out < test.txt
```

Final tests

It is possible that you may have altered qdriver.c by adding diagnostic `printf` statements. Re-download qdriver.c and build an executable q.out. At this point, you should enable the -Werror option. Next, test the executable to process a larger data set in file grades25.txt [download data sets and output files from the lab web page]:

```
$ ./q.out < grades25.txt > your-grades25-out.txt
```

You can compare your output file and the correct output file grades25-out.txt using diff:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-grades25-out.txt
grades25-out.txt
```

If diff is not silent, then one or more of your function definitions is incorrect and will require further work.

Continue to test your functions with additional data sets provided to you.

File-level and Function-level documentation

Every source and header file you submit *must* contain file-level documentation blocks whose purpose is to provide human readers [yourself and other programmers] useful information about the purpose of this source file at some later point of time

Every function that you declare in a header file [and define in a corresponding source file] must contain a function-level documentation block.

> ***Don't copy and paste documentation blocks from previous assignments. Annoyed graders will definitely subtract grades to the full extent specified in the rubrics below when they detect such copy-and-paste scenarios.***

Submission and automatic evaluation

1. In the course web page, click on the submission page to submit the necessary files q.h and q.c.
2. Read the following rubrics to maximize your grade. Your submission will receive:
    1. F grade if your submission doesn't compile with the full suite of gcc options [shown above].
    2. F grade if your submission doesn't link to create an executable.
    3. $A+$ grade if the submission's output matches the correct output. Otherwise, a proportional grade is assigned based on how many incorrect results were generated by your submission.
    4. A deduction of one letter grade for each missing documentation block. Every submitted file must have one file-level documentation block. Every function that you declare in a header file must provide a function-level documentation block. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and the three documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the three documentation blocks are missing, your grade will be later reduced from C to F.