Assignment: Computing π

Learning Outcomes

- Iteration structures
- Application of random numbers in solving engineering problems
- Understanding rounding errors generated by computations involving floating-point values
- Practice functional decomposition and algorithm design

Task: Computing π

The purpose of this assignment is to apply the topics discussed in class lectures to numerically compute the approximate value of π. Recall that π is a mathematical constant that represents the ratio of a circle's circumference to its diameter. π is irrational, meaning that it has infinite fractional digits which follow no pattern. You will implement three algorithms to approximate the value of π up to 12 fractional digits. Since the algorithms rely on computations involving floating-point values and random values, we begin with introductions to these topics.

Rounding errors with floating-point values

In class lectures, we discussed the ability of integral types [`char`, `short`, `int`, `long`] to represent precise values. For example, the `signed char` type can be used to precisely represent 256 "small" integral values in the range $[-128, 127]$. In contrast, floating-point types [`float`, `double`, `long double`] can only represent approximations of real values. Since there are infinite real values between any intervals of numbers, floating-point types cannot exactly represent these infinite real values. If you were asked to write the exact decimal notation for $\frac{1}{3}$, you couldn't. There are an infinite number of 3s after the decimal point in that real number. Well, computers don't have an infinite amount of bits either. So, what is to be done? Computers use the IEEE 754 format to approximate real values as floating-point values. These approximations are close, but they are not exact, and therefore introduce errors into calculations involving floating-point types. Usually, this isn't a big deal except for scientific and real-world applications. But, if you introduce many, many errors into the calculations, you will start to notice them even in your simpler applications.

Here's the idea and a hint at how to fix it. If you have a floating-point variable initialized as follows:

```c
double x = 1.0 / 3.0;
```

both of these statements are *supposed* to do the same thing:

```c
x + x + x + x + x + x;
6.0 * x;
```

First realize that floating-point variable `x` is not initialized with the exact value of $\frac{1}{3}$ but instead with an approximation to the exact value of $\frac{1}{3}$. The error between the exact and approximate values is called a *rounding error*. Now, each time you introduce an expression `x + x`, you introduce a little more rounding error. Why? Because you are taking an approximation and adding to it an approximation to get a new floating-point number that is further approximated. If you again add another approximate value to the previous approximate value, you will get an approximation with a larger rounding error. If you add enough `x`'s together, you will see that the result of all these additions will begin to drift away from the correct result much more than if you multiply the number once, as in `6.0 * x`.

The following code fragment has the same problem, although you may be fooled into thinking that you aren't adding `x` over and over again:

```
total = 0;
for (i = 0; i < 6; i++) {
  total += x;
}
```

Here's a code fragment that demonstrates the differences:

```
for (int count = 10; count <= 100000000; count *= 10) {
  double x = 1.0 / count;
  double multiplied = x * count;
  double added = 0;

  for (int i = 0; i < count; i++) {
    added += x;
  }

  printf("iterations: %i, x = %.8f\n", count, x);
  printf("    multiplied = %.14f\n", multiplied);
  printf("         added = %.14f\n", added);
  printf("\n");
}
```

The output clearly indicates that the rounding error is lower when performing multiplication such as `6.0 * x` rather than addition such as `x + x + x + x + x + x`:

```
iterations: 10, x = 0.10000000
    multiplied = 1.00000000000000
         added = 1.00000000000000

iterations: 100, x = 0.01000000
    multiplied = 1.00000000000000
         added = 1.00000000000000

iterations: 1000, x = 0.00100000
    multiplied = 1.00000000000000
         added = 1.00000000000000

iterations: 10000, x = 0.00010000
    multiplied = 1.00000000000000
         added = 0.99999999999991

iterations: 100000, x = 0.00001000
    multiplied = 1.00000000000000
         added = 0.99999999999808

iterations: 1000000, x = 0.00000100
    multiplied = 1.00000000000000
         added = 1.00000000000792
```

```

iterations: 10000000, x = 0.00000010
    multiplied = 1.00000000000000
         added = 0.99999999975017

iterations: 100000000, x = 0.00000001
    multiplied = 1.00000000000000
         added = 1.00000000228987

```

In closing, the above discussion should reiterate that you should always opt to implement code such as `6.0 * x` rather than `x + x + x + x + x+ x`.

Random Numbers

Many engineering problems require the use of random numbers in the development of a solution. In some cases, the numbers are used to develop a simulation of a complicated problem. The simulation can be run over and over to analyze the results; each repetition represents a repetition of the experiment.

A sequence of random numbers is not defined by an equation; instead, the sequence has certain characteristics that define it such as the maximum, minimum, and average values of the sequence. Additional characteristics also indicate whether the possible values are equally likely to occur or whether some values are more likely to occur than others. Sequences of random numbers can be generated from experiments, such as tossing a coin or rolling a die. Sequences of random numbers can also be generated using the computer.

The C standard library contains a function `rand` that returns a random positive number of type `int` between `0` and `RAND_MAX`, where `RAND_MAX` is a system-dependent integer defined in `<stdlib.h>`. Most implementations of the library provide a value of $2^{31} - 1$ for `RAND_MAX`. Function `rand` has no arguments and is referenced by expression `rand()`. Thus, to generate and print a sequence of two random numbers, you could use this statement:

```c
printf("random numbers: %d %d\n", rand(), rand());
```

Each time that a program containing this statement is executed, the same two values are printed, because function `rand` generates integers in a specified sequence.

A subsequent pair of statements generate four different random numbers that are also different from the random numbers generated in the above statement:

```c
printf("random numbers: %d %d\n", rand(), rand());
printf("random numbers: %d %d\n", rand(), rand());
```

However, after a large number of such calls to function `rand`, the sequence of values will eventually begin to repeat and therefore, instead of a random sequence, the standard library's function `rand` will generate a *pseudo-random sequence*.

Each time that function `rand` is called in a program, it generates a new value; however, each time that the program is run, function `rand` generates the same sequence of values.

In order to cause a program to generate a new sequence of random values each time that it is executed, the random-number generator must be seeded with a value. Function `srand` declared in header file `<stdlib.h>` specifies the seed for the random-number generator; for each seed value, a new sequence of random numbers is generated by function `rand`. The argument of function `srand` is an integer of type `unsigned int` that is used in computations in the library that initializes the sequence. Note that the seed value is not the first value in the sequence. If function `srand` is not called before function `rand` is called, the library assumes a value of `1` for the seed value. Therefore, if the following statement is added to a program

```c
srand(1);
```

the program will generate the same sequence of values from function `rand` as if function `srand` was not called.

In the following program, the user is asked to enter a seed value, and then the program generates 10 random numbers. Each time the user executes the program and enters the same seed, the same set of 10 random integers is generated. Each time a different seed is entered, a different set of 10 random integers is generated.

```c
#include <stdio.h>  // printf, scanf
#include <stdlib.h> // srand, rand

int main(void) {
  // get seed value from the user ...
  printf("Enter a positive integer seed value: ");
  unsigned int seed;
  scanf("%u", &seed);
  srand(seed);

  // generate and print some random numbers ...
  printf("Random Numbers: ");
  for (int i = 0, N = 10; i < N; i++) {
    printf("%i%c", rand(), i!=N-1 ? ' ' : '\n');
  }
  return 0;
}

```

Experiment with the program: use the same seed to generate the same set of random numbers, and use different seeds to generate different sets of random numbers.

> ***Note that the values generated by function `rand` are system dependent; the same set of random numbers may not be generated by function `rand` from a different compiler.***

Generating random integers over a specified range is simple with function `rand`. Suppose you want to generate random integers between 0 and 9. The following statement first generates a random number that is between `0` and `RAND_MAX`; then it uses the modulus operator to compute the modulus of the random number and integer `10`:

```c
int x = rand() % 10;
```

The result of the modulus operation is the remainder after `rand()` is divided by `10`, so the value of `x` can assume integer values between `0` and `9`.
Suppose you want to generate a random integer between -30 and 30. The total number of possible integers is 61, and a single random number in this range can be computed with this statement:

```c
int y = rand()%61 - 30;
```

The expression `rand()%61` evaluates to a value between `0` and `60`. Subtracting `30` from this expression will yield a new value between `-30` and `30`.

How should integers be generated between two specified integers m and n? First, the count of all integers between m and n, inclusive is determined: $num = n - m + 1$. Second, the value returned by function `rand` and num are given as operands to the modulus operator to obtain values in $[0, n - m]$. Finally, lower limit m is added to values in $[0, n - m]$ to obtain values in range $[m, n]$. All three steps are combined in one expression in the `return` statement of function `rand_int`:

```c
// return random integers in range [m, n]
int rand_int(int m, int n) {
  return rand()%(n-m+1) + m;
}
```

Function `rand_int` is illustrated in the following program that generates and prints 10 random integers between user-specified limits:

```c
#include <stdio.h>  // printf, scanf
#include <stdlib.h> // srand, rand
int rand_int(int m, int n); // declaration of rand_int

int main(void) {
  // get seed value and interval limits
  printf("Enter a positive integer seed value: ");
  unsigned int seed;
  scanf("%u", &seed);
  srand(seed);
  printf("Enter integer limits m and n (m < n): ");
  int m, n;
  scanf("%i %i", &m, &n);

  // generate and print some random numbers ...
  printf("Random Numbers: ");
  for (int i = 0, N = 10; i < N; i++) {
    printf("%i%c", rand_int(m, n), i!=N-1 ? ' ' : '\n');
  }
  return 0;
}

```

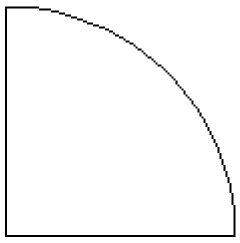
In many engineering problems, it is necessary to generate random floating-point values in a specified interval $[m, n]$. The computation to convert an integer between `0` and `RAND_MAX` to a floating-point value between m and n has three steps. First, the value returned by function `rand` is divided by `RAND_MAX` to generate a floating-point value in range $[0, 1]$. Next, values in this range are scaled by $n - m$ to values in range $[0, n - m]$. Finally, the lower limit m is added to values in $[0, n - m]$ to obtain values in range $[m, n]$. All three steps are combined in one expression in the `return` statement of function `rand_dbl`:

```
// return random double-precision floating-point values in range [m, n]
double rand_dbl(double m, double n) {
  return ((double)rand()/(double)RAND_MAX)*(n-m) + m;
}
```

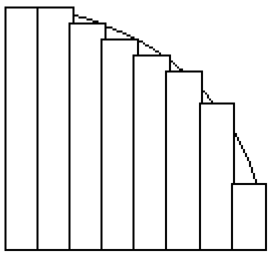
The earlier program can be easily modified to generate and print double-precision floating-point values.

Trapezoid method

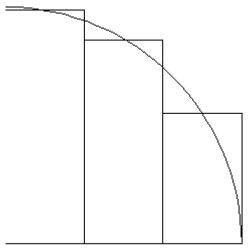
Consider a quarter circle with radius $r = 2$ units [the circle must have radius 2 and no other value for this algorithm] shown in the picture below.

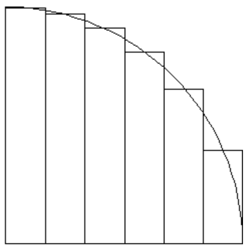


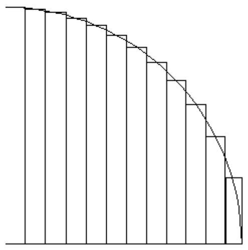
From circle area formula $area = \pi \times r^2$, the area of quarter circle with radius 2 units is $\frac{\pi \times 4}{4} = \pi$ square units. All that is required to computationally determine π's value is to compute the quarter circle's area with radius 2 units. The Trapezoid algorithm computes the quarter circle area by dividing it into a series of rectangles. These rectangles have similar width but different heights with a particular rectangle's height chosen such that the *circle's circumference passes through the midpoint of the rectangle's top*. This is shown in the picture below:

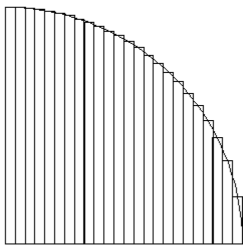


Increasing the number of rectangles provides a closer approximation of the circle's area and therefore a better approximate value for π. One consequence of increasing the number of rectangles is that each of the rectangles is thinner:


3 Rectangles


6 Rectangles


12 Rectangles


24 Rectangles

As the number of rectangles tends to infinity, each rectangle's width tends to zero, and the summation of the areas of all these rectangles will provide you with π's value.

How are you going to compute the area of each individual rectangle? Recall that given a rectangle's width w and height h, its area is $area = w \times h$. Assume there are n rectangles. Thus, the Trapezoid Rule divides interval $[0, 2]$ into n evenly spaced intervals of width $w = \frac{2}{n}$. As seen in the above pictures, each of these n rectangles will have the same width w but different heights - rectangles closer to the circle's center are taller than those near the edge. Based on the assumption that the circle's circumference passes through the midpoint of a rectangle's top edge, the height h of a rectangle whose midpoint is located x units from the circle center can be computed using Pythagorean theorem as $h = \sqrt{2^2 - x^2}$ [since the circle's radius is 2 units]. The approximate area can now be computed as the summation of each rectangle's area:

$$\pi = \left[w \times h\left(\frac{w}{2}\right)\right) + \left(w \times h\left(\frac{3w}{2}\right)\right) + \left(w \times h\left(\frac{5w}{2}\right)\right] + \cdots$$

where $h(x)$ represents the height of the rectangle whose horizontal midpoint is located x units from the circle center and is computed using Pythagorean theorem as $h(x) = \sqrt{2^2 - x^2}$. From the previous discussion on floating-point rounding errors, it is clear the previous equation is accumulating approximations in a manner similar to the expression `x + x + x + x + x + x`. Is it possible to replace this summing with a more accurate expression `6.0 * x`? The equation for the approximate area can be re-rewritten as a product by pulling the common factor w out:

$$\pi = w \times \left[h\left(\frac{w}{2}\right) + h\left(\frac{3w}{2}\right) + h\left(\frac{5w}{2}\right) + \cdots\right]$$

Since floating-point computations in the second equation are modeled as `6.0 * x`, the second equation will accumulate less floating-point errors and is therefore preferable to the first equation. Implement this second equation as the Trapezoid algorithm to calculate π's value in a function `trapezoid_pi` with prototype

```c
double trapezoid_pi(int num_of_rects);
```

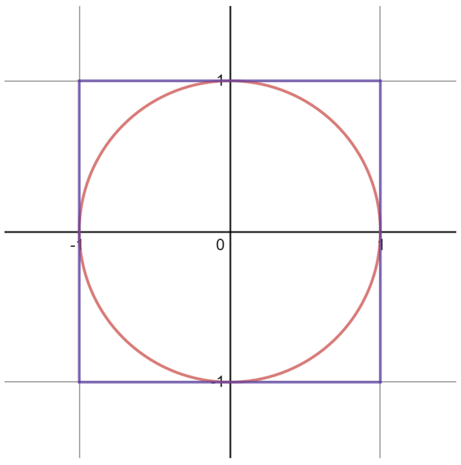
where parameter `num_of_rects` represents the count of rectangles. The function returns the area of the summed rectangles representing the value of π computed by the Trapezoid algorithm. Use C standard library function `sqrt` declared in `<math.h>` to compute $h = \sqrt{r^2 - x^2}$.

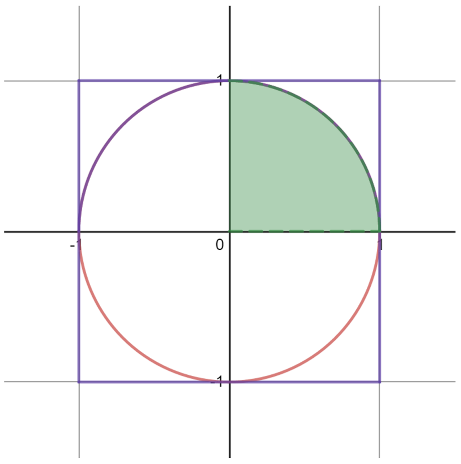
Monte Carlo simulation

Monte Carlo Simulation is a mathematical technique, which is used to estimate the possible outcomes of an uncertain event. The Monte Carlo Method was invented by [our old friend] John von Neumann and Stanislaw Ulam to understand the impact of risks on decision making during the development of the nuclear bomb. It was named after a well known casino in Monaco since the element of chance is core to the modeling approach. It is extensively used in engineering, science, economics, and finance to model complex situations where many random variables are involved.

A short description of the Monte Carlo method can be given as follows. The expected score of a player in any reasonable game of chance, however complicated, can in principle be estimated by averaging the results of a large number of plays of the game. You will use the Monte Carlo method to compute π by running a physical experiment that involves repeatedly playing the game of darts and keeping track of where the dart lands on the dartboard.

Consider a dartboard modeled as a 2×2 square $\mathcal{S}$ centered at origin $(0, 0)$ with an inscribed circle $\mathcal{C}$ of radius 1, as in the following figure to the left.





Suppose you randomly throw a dart at dartboard $\mathcal{S}$. If the dart strikes dartboard $\mathcal{S}$ at a random point P, then the probability that P is also in $\mathcal{C}$ is $\frac{area\ of\ \mathcal{C}}{area\ of\ \mathcal{S}} = \frac{\pi \times 1^2}{1 \times 1} = \pi$. In the Monte Carlo method, you'll run the experiment for N trails. In each trail, you'll randomly throw a dart and record position P and determine if P is also inside $\mathcal{C}$. At the end of the experiment, you'll have a count n of the number of times the dart is inside circle $\mathcal{S}$. The ratio $\frac{n}{N}$ will then provide an approximation of π. Point $P(x, y)$ is in [or on the] circle $\mathcal{C}$ if the square of the distance of P from the circle center $\left(x^2 + y^2\right)$ is *less than or equal* to the square of the circle's radius (1^2).

Because it is easier to generate positive random numbers, you will randomly throw a dart at the upper-right quadrant of the dartboard shaded in green in the above figure to the right. In this case, the probability that the random point P at which the dart strikes dartboard $\mathcal{S}$ is also in $\mathcal{C}$ is $\frac{\pi}{4}$. To compute the approximate value of π, you will then have to scale the result of the simulation by 4.

Implement this Monte Carlo simulation of throwing darts at the upper-right quadrant of the dartboard to approximate π in a function `montecarlo_pi` with prototype

```c
double montecarlo_pi(unsigned int seed, int num_of_trails);
```

where parameter `seed` represents the seed value to initialize the C standard library's random number generator while parameter `num_of_trails` represents the number of times the dart is randomly thrown at the dartboard. The function will return the approximate value of π. Re-read the section on random numbers to understand the meaning of parameter `seed` and to understand how to compute two random values in range $[0, 1]$ that can then represent the x and y coordinates of the random position P that the dart strikes in the upper-right quadrant of dartboard $\mathcal{S}$.

Leibniz series

Leibniz derived an infinite series of additions and subtraction to approximate π: $\frac{\pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots$. You will require an iteration statement to sum the elements of a finite number of elements in the series. As the number of terms in the series increases, the approximation will be closer to the value of $\dfrac{\pi}{4}$. After the summation of the terms, the value must be multiplied by 4 to arrive at the final approximation of π. As explained in the section on floating-point rounding errors on floating-point errors, you should multiply the summation by 4 rather than directly computing π by summing the series $\pi \approx 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$

Implement the Leibniz algorithm to calculate the value of π in a function with prototype

```
double leibniz_pi(int num_elements);
```

where function parameter `num_elements` represents the number of elements in the series. The function returns the approximate value of π.

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 ass04 , and launch the Linux shell.

Download driver source file qdriver.c and incomplete source and header files q.c and q.h, respectively.

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

> ***Do not include C standard library headers in q.h. Why? Suppose you unnecessarily include header files in q.h and your clients in turn include q.h to 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 three functions directly in q.c. Further, there is no need for q.c to include q.h because you're defining functions that don't require calls to other functions being declared in q.h.***

Test your header file q.h by compiling (only) driver source file qdriver.c [which includes q.h] :

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

Implementation tips for q.c

Use descriptions of Trapezoid, Monte Carlo, and Leibniz methods 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 `trapezoid_pi`, `montecarlo_pi`, and `leibniz_pi`.

You're expected to implement three functions in q.c and without completely defining these three 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 three 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 two functions. Later, you're faced with the task of trying to debug three incorrect functions. Is there a better alternative?

The better alternative consists of implementing *stub functions* for each of `main`'s sub-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]. 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 run. 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. After implementing, testing, and verifying the first function, you should replace the second function's stub with the actual code that implements the required functionality. This second implementation is then completely tested and verified before moving to the third function. Remember to turn on the -Werror flag if you have previously turned it off for compiling stub functions.

Here are some more tips and common programming errors related to function declarations and definitions:

1. Make sure your definitions consistently use type `double` in all computations. In general, a $\text{IEEE } 754\ 64-$bit `double` can represent any $32-$bit integer. Therefore, using a `double` type to hold integral values should not cause roundoff errors. However, a $\text{IEEE } 754\ 32-$bit `float` can only represent a $23-$bit integer.
2. Several possible errors are related to passing parameters.
   1. It is a compile error if the types in the prototype declaration and function definition are incompatible. For example, the types in the following code fragment are incompatible:

      ```c
      // function prototype declaration
      double divide(int dividend, int divisor);

      // function definition
      double divide(float dividend, float divisor) {
        // perform some sort of division here ...
      }
      ```

   2. It is a compile-time error to have a different number of arguments in the function call than there are parameters in the prototype.

3. It is a logic error if you code the parameters in the wrong order. Their meaning will be inconsistent in the called function. For example, in the following code fragment, the types are the same but the meaning of the variables is reversed:

   ```
   // function prototype declaration
   double divide(float dividend, float divisor);

   // function definition
   double divide(float divisor, float dividend) {
     // perform some sort of division here ...
   }
   ```

3. It is a compile-time error to define local variables with the same identifiers as formal parameters, as shown below:

   ```
   // function prototype declaration
   double divide(float dividend, float divisor);

   // function definition
   double divide(float divisor, float dividend) {
     // local definitions
     float dividend;

     // perform some sort of division here ...
   }
   ```

4. Using a `void` return with a function that expects a return value or using a return value with a function that expects a `void` return is a compile-time error.

5. Each parameter's type must be individually specified; you cannot use multiple definitions like you can in variables. For example, the following is a compile-time error because `y` doesn't have a type:

   ```
   double bad(float x, y);
   ```

6. Forgetting the semicolon at the end of a function prototype is a compile-time error.

   ```
   double badder(float x, float y)
   ```

7. Similarly, using a semicolon at the end of the header in a function definition is a compile-time error.

   ```
   double baddest(float x, float y);
   {
     // do bad stuff here ...
   }
   ```

8. It is most likely a logic error to call a function from within itself or one of its called functions. This is known as recursion, and its correct use is covered later.

9. It is a compile-time error to attempt to define a function within the body of another function.

```c
double hilarious(float x, float y) {
  float z = x + y;
  double naughty(float z) {
    // do naughty stuff here ...
  }
  // do hilarious stuff here ...
}
```

10. It is a run-time error to code a function call without the parentheses, even when the function has no parameters.

```c
// function definition of a function print_hello that takes an
// unknown number of parameters and returns nothing
void print_hello() {
  printf("Hello World\n");
}

// somewhere inside another function, say main() ...
int main() {
  // function name evaluates to address of first statement in function
  print_hello;
  // use the function call operator () to invoke a function
  print_hello();
}
```

11. It is a compile-time error if the type of data in the `return` statement doesn't match the function return type.

Compiling, linking, and testing

Compile (only) your source file q.c using the full suite of gcc options:

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

Link both these object files plus C standard library functions [such as `printf` and `sqrt`] into an executable file:

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

When program q.out is executed, it must be supplied an option that specifies the number of iterations [and trials and series elements]. Option s implies small number of iterations, while option m implies medium number of iterations, while option l implies large number of iterations. If you run the program with option m, as in:

```
$ ./q.out m
```

the output should look like this:

```
Approximations for pi
Iterations      Trapezoid         Monte Carlo          Leibniz
-------------------------------------------------------------------
```

```
1             3.464101615138     0.000000000000     4.000000000000
100           3.141936857900     3.080000000000     3.131592903559
200           3.141714389345     3.160000000000     3.136592684839
300           3.141658925611     3.173333333333     3.138259329516
400           3.141635700957     3.160000000000     3.139092657496
500           3.141623456820     3.152000000000     3.139592655590
600           3.141616086929     3.133333333333     3.139925988081
700           3.141611249652     3.142857142857     3.140164082890
800           3.141607874419     3.120000000000     3.140342654078
900           3.141605409563     3.151111111111     3.140481542822
1000          3.141603544913     3.156000000000     3.140592653840

```

Download text files qoutput-s.txt, qoutput-m.txt, and qoutput-l.txt that contain correct outputs from options s, m, and l, respectively. Redirect your program's output to a file. Finally, remember that you can compare your output file, say your-output-m.txt and correct output file qoutput-m.txt using diff:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-output-m.txt qoutput-m.txt
```

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

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.