In assignment 2, we are implementing various mathematical formulas in order to find the best approximation closest to pi and e.

For all functions we want to keep track of how many iterations it took until hitting epsilon (1e^-14). We can keep track of this by using a static variable.

Each function below should be returning the summation of each iteration. We want to stop iterating once the current iteration is greater than epsilon.

In the first function, calculation e, we want to implement the formula of 1 / k! We start from iteration 0 and add each iteration of 1 / k! while continuously incrementing after each iteration. We want to take the summation of the previous iteration and add that to the current iteration.

add is equal to 0
for infinite loop
    factorial = factorial times incrementing for loop
    divide 1 by factorial
    if epsilon is greater than the current iteration
        break out of loop
    add the current iteration
    increment by 1 to keep track of iterations

In the second function, calculating euler's solutions, we want to implement the following formula

$$p(n) = \sqrt{6 \sum_{k=1}^{n} \frac{1}{k^2}}$$

. We can do this by following the pseudocode provided:

fact is equal to 1

for infinite loop
    newfact = fact * for loop iteration;
    square = newfact / 2;
    add the iteration of square each time
    if epsilon is greater than square
        multiply by 6
        Take square root using newton.c
    increment static counter
    increment fact

In the third function, the Bailey Borwein Plouffe Formula (BBP), we want to implement the following formula $p(n) = \sum_{k=0}^{n} 16^{-k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)$. We can implement by this in a few steps:

1. Run a loop until the current iteration is greater than epsilon
2. Implement four different variables while assigning each division operation to a variable. For example: division1 = 4 / 8 * k + 1
3. We can divide by 16 each time to take the negative power
4. We can then implement the formula using the values from step 2 and 3
5. Add each iteration and increment our static variable

In the fourth function, the madhava series we are looking to implement the following formula $= \sqrt{12}\sum_{k=0}^{n}\frac{(-3)^{-k}}{2k+1}$ =. This is very similar to the BBP formula:

1. We can run a loop until the absolute value of the current iteration is epsilon. We need to take the absolute value since we are taking the power of a negative base.
2. Then we can divide by -3 each time similar to how we found the power in BBP
3. We then can take that current power and divide by the denominator
4. Same as step 5 from BBP
5. Finally, once broken out of the for loop we can take the total added iterations and multiply to the square root of 12 using the sqrt_newton function.

In the fifth function, viete formula, we are implementing the following formula.

$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \times \frac{\sqrt{2+\sqrt{2}}}{2} \times \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdots$

We are implementing a similar algorithm as the first function of finding e by taking the previous iteration and adding to the square root of 2. We can follow the provided pseudocode below:

for absolute value of current iteration is greater than epsilon
    ptr = square root of 2
    front = ptr;
    ptr /= 2
    Multiply each iteration
    Increment static counter
Divide 2 by the total added iterations

The sixth function, newton, should be taking in a value and converting that value to a square root. We can use the pseudo code provided to us below while also making sure to keep track of the iterations.

```
1 def sqrt(x):
2     z = 0.0
3     y = 1.0
4     while abs(y - z) > epsilon:
5         z = y
6         y = 0.5 * (z + x / z)
7     return y
```

All of the functions above, except e formula, should equate to the pi value after **returning the summation value.** The e formula should equate to the value of e (2.71828…..)

In mathlib-test, we want to use the command line options:

-a**:** Runs All Tests.

-e**:**Runs Approximation Test.

-b**:**Runs Bailey-Borwein-Plouffe π approximation test.

-m**:**Runs Madhava π approximation test.

-r**:** Runs Euler sequence π approximation test.

-v**:**Runs Viète π approximation test.

-n**:**Runs Newton-Raphson Square Root Approximation Tests.

- s**:** Enable printing of statistics to see computed terms and factors foreach nested function.

• -h**:** Display A Help Message Detailing Program Usage.

You can use getopt() to scan through the command line arguments while running a switch statement for each command line option which should call the formula that is being tested. Finally we want to take our approximations and compare it to the actual pi and e values in the <math.h> library. The output should print our value, the actual value of pi or e from math.h, and the difference between our value and the actual value while taking the square root.

Be sure to create your own makefile and provide README.md.

mathlib-test.c (main)

1. Use getopt() to receive command line arguments
2. Use switch statement and use the command line arguments for each case
3. In each case we want to print our approximation, the math.h number, and the difference between the two values while taking the absolute value
4. We also want to print out the number of terms but we need to check if -s is taken as an argument first so we can have an initial while loop at the beginning of the program checking if -s is in the command line. If -s is in the command line, we can set a boolean to true.
5. Then when running each case we can ask if that boolean is true and then print out the number of terms.
6. We can set a boolean to check if we never enter the loop in which case we want to print the help message.