Barrett Schonefeld

HW4

2)

I tested each of the functions using 15 temperatures (0-14) with the tic toc function. The results are displayed below.

For-Loop:

>> evalTempFor

0 degrees Celsius equals 32 degrees Fahrenheit

1 degrees Celsius equals 33.8 degrees Fahrenheit

2 degrees Celsius equals 35.6 degrees Fahrenheit

3 degrees Celsius equals 37.4 degrees Fahrenheit

4 degrees Celsius equals 39.2 degrees Fahrenheit

5 degrees Celsius equals 41 degrees Fahrenheit

6 degrees Celsius equals 42.8 degrees Fahrenheit

7 degrees Celsius equals 44.6 degrees Fahrenheit

8 degrees Celsius equals 46.4 degrees Fahrenheit

9 degrees Celsius equals 48.2 degrees Fahrenheit

10 degrees Celsius equals 50 degrees Fahrenheit

11 degrees Celsius equals 51.8 degrees Fahrenheit

12 degrees Celsius equals 53.6 degrees Fahrenheit

13 degrees Celsius equals 55.4 degrees Fahrenheit

14 degrees Celsius equals 57.2 degrees Fahrenheit

Elapsed time is 0.062795 seconds.

While-Loop:

>> evalTempWhile

0 degrees Celsius equals 32 degrees Fahrenheit

1 degrees Celsius equals 33.8 degrees Fahrenheit

2 degrees Celsius equals 35.6 degrees Fahrenheit

3 degrees Celsius equals 37.4 degrees Fahrenheit

4 degrees Celsius equals 39.2 degrees Fahrenheit

5 degrees Celsius equals 41 degrees Fahrenheit

6 degrees Celsius equals 42.8 degrees Fahrenheit

7 degrees Celsius equals 44.6 degrees Fahrenheit

8 degrees Celsius equals 46.4 degrees Fahrenheit

9 degrees Celsius equals 48.2 degrees Fahrenheit

10 degrees Celsius equals 50 degrees Fahrenheit

11 degrees Celsius equals 51.8 degrees Fahrenheit

12 degrees Celsius equals 53.6 degrees Fahrenheit

13 degrees Celsius equals 55.4 degrees Fahrenheit

14 degrees Celsius equals 57.2 degrees Fahrenheit

Elapsed time is 0.012949 seconds.

Vectorized:

Enter F2C for Fahrenheit to Celsius or C2F for Celsius to Fahrenheit: c2f

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 degrees Celsius equals 32 33.8 35.6 37.4 39.2 41 42.8 44.6 46.4 48.2 50 51.8 53.6 55.4 57.2 degrees Fahrenheit

Elapsed time is 0.002110 seconds.

The vectorized code computes the temperatures the fastest.

4)

This program takes an initial value, 2.0, and takes the square root of the value n times. Then, after the nth square root, the program squares that root n times. The value for that iteration, n, is printed with the initial input value, 2.0, and the output after the square roots and squares. This is done for all values of n between 1 and 60 to compare the results for each value of n.

With 1 sqrt, then 1 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000000004

With 2 sqrt, then 2 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999999996

With 3 sqrt, then 3 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999999996

With 4 sqrt, then 4 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999999964

With 5 sqrt, then 5 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999999964

With 6 sqrt, then 6 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999999964

With 7 sqrt, then 7 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999999714

With 8 sqrt, then 8 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000000235

With 9 sqrt, then 9 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000000235

With 10 sqrt, then 10 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000000235

With 11 sqrt, then 11 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000000235

With 12 sqrt, then 12 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999991336

With 13 sqrt, then 13 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999973292

With 14 sqrt, then 14 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999973292

With 15 sqrt, then 15 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999999973292

With 16 sqrt, then 16 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000117746

With 17 sqrt, then 17 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000408580

With 18 sqrt, then 18 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000000408580

With 19 sqrt, then 19 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000001573586

With 20 sqrt, then 20 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000001573586

With 21 sqrt, then 21 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000001573586

With 22 sqrt, then 22 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000010885857

With 23 sqrt, then 23 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000029511749

With 24 sqrt, then 24 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000066771721

With 25 sqrt, then 25 times ^2 operations, the number 2.0000000000000000 becomes: 2.0000000066771721

With 26 sqrt, then 26 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999917775542

With 27 sqrt, then 27 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999917775542

With 28 sqrt, then 28 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999917775542

With 29 sqrt, then 29 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999917775542

With 30 sqrt, then 30 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999917775542

With 31 sqrt, then 31 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999999917775542

With 32 sqrt, then 32 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999990380770896

With 33 sqrt, then 33 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999971307544144

With 34 sqrt, then 34 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999971307544144

With 35 sqrt, then 35 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999971307544144

With 36 sqrt, then 36 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999971307544144

With 37 sqrt, then 37 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999971307544144

With 38 sqrt, then 38 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999360966436217

With 39 sqrt, then 39 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999360966436217

With 40 sqrt, then 40 times ^2 operations, the number 2.0000000000000000 becomes: 1.9999360966436217

With 41 sqrt, then 41 times ^2 operations, the number 2.0000000000000000 becomes: 1.9994478907329654

With 42 sqrt, then 42 times ^2 operations, the number 2.0000000000000000 becomes: 1.9984718365144798

With 43 sqrt, then 43 times ^2 operations, the number 2.0000000000000000 becomes: 1.9965211562778555

With 44 sqrt, then 44 times ^2 operations, the number 2.0000000000000000 becomes: 1.9965211562778555

With 45 sqrt, then 45 times ^2 operations, the number 2.0000000000000000 becomes: 1.9887374575497223

With 46 sqrt, then 46 times ^2 operations, the number 2.0000000000000000 becomes: 1.9887374575497223

With 47 sqrt, then 47 times ^2 operations, the number 2.0000000000000000 becomes: 1.9887374575497223

With 48 sqrt, then 48 times ^2 operations, the number 2.0000000000000000 becomes: 1.9887374575497223

With 49 sqrt, then 49 times ^2 operations, the number 2.0000000000000000 becomes: 1.8682459487159784

With 50 sqrt, then 50 times ^2 operations, the number 2.0000000000000000 becomes: 1.6487212645509468

With 51 sqrt, then 51 times ^2 operations, the number 2.0000000000000000 becomes: 1.6487212645509468

With 52 sqrt, then 52 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 53 sqrt, then 53 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 54 sqrt, then 54 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 55 sqrt, then 55 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 56 sqrt, then 56 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 57 sqrt, then 57 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 58 sqrt, then 58 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 59 sqrt, then 59 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

With 60 sqrt, then 60 times ^2 operations, the number 2.0000000000000000 becomes: 1.0000000000000000

After running the code, we get 60 printed statements that illustrate the rounding errors in MATLAB. With more iterations, the values start to stray farther from the initial value, 2.0. On the 52nd iteration, the output value becomes 1.0, and this maintains all the way through the 60th iteration. This means the 52nd square root of 2.0 is rounded in MATLAB to 1.0. Therefore, any square root after 52 is just considered 1.0, and any square of the n >= 52 square root remains as 1.0. The results of these calculations should actually be 2.0; however, MATLAB’s formulas for the square roots and squares are not perfect, and they are subject to rounding errors that may significantly alter the results of a calculation over time.

5)

This code takes an initial value of eps of 1.0. It adds then performs a while loop when 1.0 is not equal to 1.0 + eps. The while loop displays the current value of eps then divides eps by two then loops again until the statement ‘1.0 is not equal to 1.0 + eps’ is false. When the while loop is exited, the final eps value is displayed.

1

0.5

0.25

0.125

0.0625

0.03125

0.015625

0.0078125

0.0039062

0.0019531

0.00097656

0.00048828

0.00024414

0.00012207

6.1035e-05

3.0518e-05

1.5259e-05

7.6294e-06

3.8147e-06

1.9073e-06

9.5367e-07

4.7684e-07

2.3842e-07

1.1921e-07

5.9605e-08

2.9802e-08

1.4901e-08

7.4506e-09

3.7253e-09

1.8626e-09

9.3132e-10

4.6566e-10

2.3283e-10

1.1642e-10

5.8208e-11

2.9104e-11

1.4552e-11

7.276e-12

3.638e-12

1.819e-12

9.0949e-13

4.5475e-13

2.2737e-13

1.1369e-13

5.6843e-14

2.8422e-14

1.4211e-14

7.1054e-15

3.5527e-15

1.7764e-15

8.8818e-16

4.4409e-16

2.2204e-16

final eps:1.1102e-16

The value of eps never actually equals zero, so the while loop should never be exited. However, when eps becomes very small, MATLAB rounds eps to zero. Therefore, MATLAB evaluates the 1.0 ~ = 1.0 + eps as false once eps rounds to zero and exits the while loop. The final value printed shows the value of eps that rounds to zero.

7)

Recursive Outputs:

Please enter a non-negative integer or type stop: 10

fib(10) = 55

timeit(getFib(10)) = 4.5759e-05

Please enter a non-negative integer or type stop: 15

fib(15) = 610

timeit(getFib(15)) = 0.00017526

Please enter a non-negative integer or type stop: 20

fib(20) = 6765

timeit(getFib(20)) = 0.0011268

Please enter a non-negative integer or type stop: 25

fib(25) = 75025

timeit(getFib(25)) = 0.0084185

Please enter a non-negative integer or type stop: 30

fib(30) = 832040

timeit(getFib(30)) = 0.095892

Please enter a non-negative integer or type stop: 35

fib(35) = 9227465

timeit(getFib(35)) = 1.0616

For-Loop Outputs:

Please enter a non-negative integer or type stop: 10

fib(10) = 55

timeit(getFib(10)) = 9.5005e-06

Please enter a non-negative integer or type stop: 15

fib(15) = 610

timeit(getFib(15)) = 5.8865e-06

Please enter a non-negative integer or type stop: 20

fib(20) = 6765

timeit(getFib(20)) = 3.528e-06

Please enter a non-negative integer or type stop: 25

fib(25) = 75025

timeit(getFib(25)) = 5.1762e-06

Please enter a non-negative integer or type stop: 30

fib(30) = 832040

timeit(getFib(30)) = 4.8096e-06

Please enter a non-negative integer or type stop: 35

fib(35) = 9227465

timeit(getFib(35)) = 3.5283e-06

The for-loop is much faster than the recursive method. The times for the for-loop are all of the same magnitude (between 1e-7 and 3e-6 seconds); moreover, finding the 35th number in the sequence was almost three times faster than finding the 10th number in the sequence. The for-loop remains very efficient even when finding numbers later in the sequence.

The recursive form, on the other hand, slows drastically from the 10th number in the sequence to the 35th number, changing magnitudes by and order of 1e5 seconds. The recursive function is much slower because it has to call the function for (n-1) and (n-2) every time it to get the next the next value in the sequence. Since it has to go back to get the last two values every time, the code runs much slower.

The for-loop does not have to go back to find the previous two values because it saves the values of (n-1) and (n-2) after each iteration. The computer can do the addition extremely quickly, so finding a number later in the sequence does not take much longer than the numbers early in the sequence.