Question 1:

a.) Compute f(x) = sin (x) using a Taylor series expansion. Compute sin(x) for 4 different values, though be careful not to use too large a value. Generate two versions of your code, first defining x and sin(x) to use floats (SP), and second, defining them as doubles (DP). Discuss any differences you find in your results for f(x). You should provide an in-depth discussion on the results you get and the reasons for any differences.

The Taylor series approximation was implemented in two separate functions, one for single precision and the other for double precision.

Approximating sin(1):

|  |  |  |
| --- | --- | --- |
| Terms: | Float SP Time (ms) | Double DP Time (ms) |
| 40 | 0.030 | 0.007 |
| 80 | 0.037 | 0.019 |
| 400 | 0.576 | 0.648 |
| 800 | 1.722 | 1.769 |

Approximating sin(2):

|  |  |  |
| --- | --- | --- |
| Terms: | Float SP Time (ms) | Double DP Time (ms) |
| 40 | 0.035 | 0.025 |
| 80 | 0.048 | 0.023 |
| 400 | 0.551 | 0.668 |
| 500 | 0.719 | 0.824 |

Approximating sin(4):

|  |  |  |
| --- | --- | --- |
| Terms: | Float SP Time (ms) | Double DP Time (ms) |
| 40 | 0.034 | 0.008 |
| 80 | 0.049 | 0.023 |
| 120 | 0.12 | 0.094 |
| 800 | 0.141 | 0.193 |

Approximating sin(8):

|  |  |  |
| --- | --- | --- |
| Terms: | Float SP Time (ms) | Double DP Time (ms) |
| 40 | 0.033 | 0.008 |
| 80 | 0.066 | 0.024 |
| 100 | 0.067 | 0.558 |
| 150 | 0.113 | 0.148 |

As the approximating value x of sin(x) increases, the number of terms to be approximated before reaching NaN decreases. However, it is evident that, across all cases, the single precision approximation is initially less efficient, but as the number of approximations increases, the double precision efficiency is less than single precision.

It would be expected that due to the compromise of lower precision, single precision approximation would be more efficient for all terms. For smaller term approximations, overhead is likely present in handling float operations. This is because CPUs such Intel Xeon which I ran the code on, are optimized for double operations, so for small approximations the overhead of using floats outweighs the slower speed of doubles. As the terms increase, the cost of using doubles begins to greatly increase and eventually floats become more efficient. More floats can fit into the cache since they are 32 bits, which reduces memory latency compared to the 64-bit doubles. Since the factorials in the approximation increase the size of computation, as the value of x increases, the maximum number of approximations before reaching a NaN final value decreases. However, since doubles are more accurate, this maximum is greater for doubles, enabling for greater precision.

b.) Provide both IEEE 754 single and double precision representations for the following numbers: 2.1, 6300, and -1.044.

2.1:

Single – 1 bit for sign bit, 8 bits for exponent, 23 bits for fraction:

Sign Bit: 0

(-1)^s \* 1.f \* 2^e, and s = 0, so

1.f \* 2^e

To convert 0.1 to binary, multiply by 2 and take integer part of the result (0 or 1). Repeat this process 0.1 \* 2 = 0.2, 0.2 \* 2 = 0.4, 0.4 \* 2 = 0.8, 0.8 \* 2 = 1.6, 0.6 \* 2 = 1.2 0.2 \* 2 = 0.4

Biased Exponent: e + 127 = 1 + 127 = 128, 128 = 1000000

Fraction: 00001100110011001100110

**0 10000000 00001100110011001100110**

Double – 1 bit for sign bit, 11 bits for exponent, 52 bits for fraction:

Sign bit and exponent are the same, fraction is a more accurate continuation:

**0 10000000000 0000110011001100110011001100110011001100110011001101**

6300:

Single – 1 bit for sign bit, 8 bits for exponent, 23 bits for fraction:

Sign Bit: 0

To convert 6300 to binary, e + 127, and 1.100010011100 \* 2 ^ 12 exponent is 12, 12 + 127 leads to 10001011. 6300 divided by 2 is 3150 with remainder 0. 3150 divided by 2 is 1575, remainder 0. 1575 /2=787 rem 1. 787/2=393 rem1. 393/2=196 rem1. 196/2=98 rem0, etc. till 0. Remainders: 1100010011100.

**0 10001011 10001001110000000000000**

Double – 1 bit for sign bit, 11 bits for exponent, 52 bits for fraction:

E = 12, so e + 1023 = 1035 or 100000001011

**0 10000001011 1000100111000000000000000000000000000000000000000000**

-1.044:

Single – 1 bit for sign bit, 8 bits for exponent, 23 bits for fraction:

Sign bit: 1

To convert –1.044, e + 127, and 1.0000101101000011100 \* 2 ^ 0, so binary is 01111111. To convert to binary, first, handle the integer part. 1 in binary is 1. The fractional part is 0.044. Converting 0.044 to binary: multiply by 2, take the integer part each time. 0.044 \*2=0.088, 0. 0.088\*2=0.176, 0. 0.176\*2=0.352, 0. 0.352\*2=0.704, 0. 0.704\*2=1.408, 1. 0.408\*2=0.816, 0. 0.816\*2=1.632, etc.

**1 01111111 00001011010000111001011**

Double – 1 bit for sign bit, 11 bits for exponent, 52 bits for fraction:

E = 0 + 1023 = 1023, so e is 0111111111

**1 01111111111 0000101101000011100101011000000100000110001001001110**