1.

(i) Sign bit: The left-most bit of the binary number represents the sign of the number. 0 is positive, 1 is negative.

(ii) 1's complement: -N is the negation of every bit of N.

(iii) 2's complement: -N is the negation of every bit of N, then plus 1.

In this question, I will use 16-bit binary number (2 registers) to represent -161.

(i) Sign bit: 1000 0000 1010 0001

(ii) 1's complement: 1111 1111 0101 1110

(iii) 2's complement: 1111 1111 0101 1111

2.

(a)

The decimal number should be -3.4500e-12

```
>> q2

result =

  -3.4500e-12

>>
```

(b)

The maximum should be 3.4028e+38

```
>> q2

result =

   3.4028e+38
```

(c)

The minimum should be 1.1755e-38

```
>> q2

result =

   1.1755e-38
```

(d)

It should also be 1.1755e-38

(e)

The significant figure should be 6

3.

This is the MATLAB result when x = 0.5, 5, 50, epsilon = 0.000001

```
>> q3_1
The computation takes 4 turns
The original function result is 0.479425538604203
The Taylor series evaluation result is 0.479425538616416
>> q3_1
The computation takes 11 turns
The original function result is -0.958924274663138
The Taylor series evaluation result is -0.958924293212820
>> q3_1
The computation takes 73 turns
The original function result is -0.262374853703929
The Taylor series evaluation result is 52246.808511876195553
>>
```

When evaluating Taylor series, the accuracy decreases and computing time increases as x increases. When x is very large (x=50), this expansion result is far away from the accurate value.

For the reason, when x is too large, the term x^n/n! will take many turns to converge. This cause the long computation time. Also, during the process, the value of x^n/n! will become very large (even to 20 significant figures). In this case, the accuracy of the number is insufficient, which will lead to errors in small values. But this small value is very large compared to the accurate result. In my computation of x=50, when n is around 20, the value of x^n/n! is about 1e20. This leads to an error of 5000, which is quite small compared to 1e20, but very large compared to final accurate value -0.262.

To solve this problem, it is better than we can modulo x by 2pi to get 0 <= x <= 2*pi. In this case, the value of x is not too large, which will save time and guarantee accuracy.

```
>> q3_2
The computation takes 2 turns
The original function result is 1.000499875062461
The Taylor series evaluation result is 1.000499875000000
>> q3_2
The computation takes 3 turns
The original function result is 1.004987562112089
The Taylor series evaluation result is 1.004987562500000
>> q3_2
The computation takes 5 turns
The original function result is 1.048808848170152
The Taylor series evaluation result is 1.048808867187500
>> q3_2
The computation takes 61 turns
The original function result is 1.378404875209022
The Taylor series evaluation result is 1.378405325707288
>>
```

The same problem also happens. When x is large, the accuracy of computation decreases, and it requires more turns to converge. However, the value of x is limited between (-1,1). Therefore, the

error is not so large as in the Taylor series of sin function.

4.

This test is important because floating points are stored in binary numbers in computer. When converting a decimal number to a binary number, some decimal numbers don't have a precise binary representation. In this case, two different binary number may refer to the same decimal number, if we ignore the very small difference. So, it is reasonable if we could tolerate a certain degree of difference when doing comparison. That is why we need epsilon test.

However, it is not good enough if we just set a fixed epsilon in comparison. For example, the errors of a very large binary number is quite large compared to the errors of a very small binary number. It will be better if we could use a proportional tolerance instead of a fixed tolerance, for instance, if *abs(a – b) < ε * abs(a)* then return *a* and *b* are equal.

Here, for example, a=3, b=4, c=5, this is the result:

```
>> q4
2 complex roots, which are -0.66667+1.10554j and -0.66667+-1.10554j
```

And for a = 2, b=4, c=2:

```
>> q4
Only 1 root, which is -1.00000
>>
```

5.

(i) first order approximation: $f(x_{i+1}) = x_i^2 + \sin(2x_i) + (2x_i + 2\cos(2x_i))(x_{i+1} - x_i)$

(ii) second order approximation: $f(x_{i+1}) = x_i^2 + \sin(2x_i) + (2x_i + 2\cos(2x_i))(x_{i+1} - x_i) + (2 - 4\sin(2x_i))(x_{i+1} - x_i)^2$

(iii) *f(1) = 1.9093*

First order approximation: *f(1.2)=2.1428*

Second order approximation: *f(1.2)=2.0774*