Section: Root-finding

Given a continuous function $f : \mathbb{R} \to \mathbb{R}$, we often need to find the root of f. Generally, the roots of a function cannot be expressed in a closed form, so root-finding algorithms can provide approximation to the root. There are different type of root-finding methods, here we introduce some of them:

Bracketing Methods (Bisection method, False positoin method...)

We make a initial guess of an interval such that f has different signs at the end points, then the function has a root in the interval by the intermediate value theorem. Then by iterating, we determine a smaller interval, so that the root can be found.

Iterative methods (Newton's method, Secant method...)

In iterative methods we use specific type of iteration by defining an auxiliary function, then iterate until we find the fixed point of the iteration function. This is an application of the fixed-point iteration.

Reference: Root-finding algorithm

[1] Bracketing Methods

[1.1] Bisection method (二分逼進法)

Condiser a continuous function $f(x) : [a, b] \to \mathbb{R}$ where $f(a)f(b) < 0$. We wish to find the root of $f(x) = 0$.

Since we already know that there must be a root in the interval, we split uniformly the interval into 2 and we know that at least one of the interval must contains a root. In this way, we get a interval that is half of length of the original one and we are sure that there must be a root exists in the interval. We then repeat the process to identify more accurately the location of the root.

Remark 1: The bisection method ensures that one can find one root. However, there might be multiple roots exist in the interval. Need to find it one by one.

Remark 2: The bisection method requires a bit of knowledge of the root, that is, we need to know in advance a and b such that $f(a)f(b) < 0$. Therefore, there is no way to find the root of $f(x) = (x - pi)^2$ where the function is always non-negative.

[1.1.1] The bisection method can be given as following:

Let $a_0 = a, b_0 = b$. For $i = 0, 1, 2, \cdots, n$

1. Calculate

$$c_i = \frac{a_i + b_i}{2},$$

   and calculate $f(c_i)$.
2. If either $|c_i - a_i|$ or $|f(c_i)|$ is smaller than a predefined small number $\epsilon \ll 1$, stop iterating.
3. - If $f(c_i)$ has the same sign as $f(a_i)$, let $a_{i+1} = c_i$ and $b_{i+1} = b_i$.
   - Otherwise, let $a_{i+1} = b_i$ and $b_{i+1} = c_i$.

Reference: Bisection method

Example: Square root of a number:

$$f_1(x) = x^2 - R$$

Bisection iteration

Initial guess: $a_0 = 0, b_0 = R$.

```julia
# Setup parameters
R = 3;

# eps1: epsilon for values
# eps2: epsilon for function values
eps1 = 1.0e-14; eps2 = eps1;

# Setup the initial interval [a, b]
a = 0; fa = a^2-R;
b = R; fb = b^2-R;

# n_iter_max: max. number of iterations
n_ite = 0; n_iter_max = 50;

# Initialize er1 (error between values) and er2 (error between function values)
er1 = 1; er2 = 1;

# Start the iteration
while er1 > eps1 && er2>eps2 && n_ite < n_iter_max
    # Define the midpoint c
    c = a + (b-a)/2;

    # Evaluate the function value at the midpoint c
    fc = c^2-R;

    # Determine where is the root
    if(fc*fa<0)
        b=c;
        fb = fc;
    else
        a=c;
        fa = fc;
    end

    # Evaluate errors and number of iterations
    er1 = b-a; er2 = abs(fc);
    n_ite = n_ite + 1;

    # Print the solution at n-th iteration and the exact error
    println("c= ", c, "\t\t", "error= ", abs(c-sqrt(R)))
end

# Print total number of iterations
println("Total number of iterations= ", n_ite)
```

```
c= 1.5      error= 0.2320508075688772
c= 2.25     error= 0.5179491924311228
c= 1.875        error= 0.1429491924311228
c= 1.6875       error= 0.04455080756887719
```

```
c= 1.78125      error= 0.04919919243112281
c= 1.734375     error= 0.002324192431122807
c= 1.7109375        error= 0.021113307568877193
c= 1.72265625       error= 0.009394557568877193
c= 1.728515625      error= 0.003535182568877193
c= 1.7314453125     error= 0.0006054950688771932
c= 1.73291015625        error= 0.0008593486811228068
c= 1.732177734375       error= 0.00012692680612280682
c= 1.7318115234375      error= 0.00023928413137719318
c= 1.73199462890625     error= 5.6178662627193177e-5
c= 1.732086181640625        error= 3.5374071747806823e-5
c= 1.7320404052734375       error= 1.0402295439693177e-5
c= 1.7320632934570312       error= 1.2485888154056823e-5
c= 1.7320518493652344       error= 1.0417963571818234e-6
c= 1.732046127319336        error= 4.680249541255677e-6
c= 1.7320489883422852       error= 1.8192265920369266e-6
c= 1.7320504188537598       error= 3.887151174275516e-7
c= 1.732051134109497        error= 3.265406198771359e-7
c= 1.7320507764816284       error= 3.1087248775207854e-8
c= 1.7320509552955627       error= 1.4772668555096402e-7
c= 1.7320508658885956       error= 5.8319718387878083e-8
c= 1.732050821185112        error= 1.3616234806335115e-8
c= 1.7320507988333702       error= 8.73550698443637e-9
c= 1.732050810009241        error= 2.4403639109493724e-9
c= 1.7320508044213057       error= 3.1475715367434987e-9
c= 1.7320508072152734       error= 3.536038128970631e-10
c= 1.7320508086122572       error= 1.0433800490261547e-9
c= 1.7320508079137653       error= 3.448881180645458e-10
c= 1.7320508075645193       error= 4.3578474162586645e-12
c= 1.7320508077391423       error= 1.7026513532414356e-10
c= 1.7320508076518308       error= 8.295364395394245e-11
c= 1.732050807608175        error= 3.929789826884189e-11
c= 1.7320508075863472       error= 1.7470025426291613e-11
c= 1.7320508075754333       error= 6.556089005016474e-12
c= 1.7320508075699763       error= 1.099120794378905e-12
c= 1.7320508075672478       error= 1.6293633109398797e-12
c= 1.732050807568612        error= 2.651212582804874e-13
c= 1.7320508075692942       error= 4.169997680492088e-13
c= 1.7320508075689531       error= 7.593925488436071e-14
c= 1.7320508075687826       error= 9.459100169806334e-14
c= 1.7320508075688679       error= 9.325873406851315e-15
c= 1.732050807568910        error= 3.3306690738754696e-14
c= 1.7320508075688892       error= 1.199040866595169e-14
c= 1.7320508075688785       error= 1.3322676295501878e-15
Total number of iterations= 48
```

[1.2] Method of false position

Condiser a continuous function $f(x) : [a, b] \to \mathbb{R}$ where $f(a)f(b) < 0$. We wish to find the root of $f(x) = 0$.

We know that the root is located between a and b. To have a guess of the root, intuitively one can use a linear approximation to it, that is, assuming that the function is linear and we draw a line to connect the two points $(a, f(a))$ and $(b, f(b))$, and look for the intersection between this line and the x-axis. This is exactly the idea of the method of false position.

[1.2.1] The method of false position (which is really similar to bisection method) can be given as following:

Let $a_0 = a, b_0 = b$. For $i = 0, 1, 2, \cdots, n$

1. Calculate

$$c_i = a_i - \frac{f(a_i)(b_i - a_i)}{f(b_i) - f(a_i)} = \frac{a_i f(b_i) - b_i f(a_i)}{f(b_i) - f(a_i)},$$

   and calculate $f(c_i)$.
2. If $|c_i - a_i|$ is small or $|f(c_i)|$ is small, stop iterating.
3.
   - If $f(c_i)$ has the same sign as $f(a_i)$, let $a_{i+1} = c_i$ and $b_{i+1} = b_i$.
   - Otherwise, let $a_{i+1} = b_i$ and $b_{i+1} = c_i$.

Reference: [False position method]

Example: Square root of a number:

$$f_1(x) = x^2 - R$$

False position iteration

Initial guess: $a_0 = 0, b_0 = R$.

```julia
# Setup parameters
R = 3;

# eps1: epsilon for values
# eps2: epsilon for function values
eps1 = 1.0e-14; eps2 = eps1;

# Setup the initial interval [a, b]
a = 0; fa = a^2-R;
b = R; fb = b^2-R;

# n_iter_max: max. number of iterations
n_ite = 0; n_iter_max = 50;

# Initialize er1 (error between values) and er2 (error between function values)
er1 = 1; er2 = 1;

# Start the iteration
while er1 > eps1 && er2>eps2 && n_ite < n_iter_max
    # Evaluate c and f(c)
    c = a - fa*(b-a)/(fb-fa);
    fc = c^2-R;

    # Determine where is the root
    if(fc*fa<0)
        b=c;
        fb = fc;
    else
        a=c;
        fa = fc;
    end

    # Evaluate errors and number of iterations
    er1 = b-a; er2 = abs(fc);
    n_ite = n_ite + 1;

    # Print the solution at n-th iteration and the exact error
    println("c= ", c, "\t\t", "error= ", abs(c-sqrt(R)))
end

# Print total number of iterations
println("Total number of iterations=", n_ite)
```

```
c= 1.0      error= 0.7320508075688772
c= 1.5      error= 0.2320508075688772
c= 1.6666666666666667       error= 0.06538414090221045
c= 1.7142857142857142       error= 0.017765093283163003
c= 1.7272727272727273       error= 0.0047780802961499
c= 1.7307692307692308       error= 0.0012815767996463556
c= 1.7317073170731707       error= 0.0003434904957064777
c= 1.731958762886598        error= 9.204468227919094e-5
c= 1.7320261437908497       error= 2.4663778027456118e-5
c= 1.7320441988950277       error= 6.608673849495261e-6
c= 1.7320490367775832       error= 1.7707912940423398e-6
c= 1.7320503330866026       error= 4.74482274581689e-7
c= 1.7320506804317222       error= 1.2713715502599143e-7
c= 1.7320507735025783       error= 3.4066298892909685e-8
c= 1.73205079844084     error= 9.128037214978235e-9
c= 1.732050805123027        error= 2.44585018904786e-9
c= 1.7320508069135137       error= 6.553635412132053e-10
c= 1.7320508073932732       error= 1.75603975804961e-10
c= 1.7320508075218244       error= 4.705280609584861e-11
c= 1.7320508075562695       error= 1.2607692667643278e-11
c= 1.732050807565499        error= 3.3781866193294263e-12
c= 1.7320508075679721       error= 9.050538096744276e-13
c= 1.7320508075686347       error= 2.424727085781342e-13
c= 1.7320508075688124       error= 6.483702463810914e-14
c= 1.7320508075688599       error= 1.7319479184152442e-14
c= 1.7320508075688725       error= 4.6629367034256575e-15
c= 1.732050807568876        error= 1.1102230246251565e-15
Total number of iterations=27
```

[2] Iterative methods

[2.1] Newton's method (切線法)

Suppose we are given the point $(x_n, f(x_n))$ and the slope at this point $f'(x_n)$, we can then draw the tangent line and look for the intersection between this tangent line and the x-axis. This should be a good guess of the root and this is exactly the idea of Newton's method.

[2.1.1] The Newton's method is given as the following:

Given an initial guess x_0, for $n = 0, 1, 2, \cdots$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

and iterate until $|x_{n+1} - x_n|$ is small enough or we exceeded the maxium number of iterations.

Reference: Newton's method

Example 1: Square root of a number:

$$f_1(x) = x^2 - R$$

Newton's iteration

$$x_{n+1} = x_n - \frac{f_1(x_n)}{f_1'(x_n)} = x_n - \frac{x_n^2 - R}{2x_n} = \frac{x_n}{2} + \frac{R}{2x_n}$$

```julia
# Setup parameters
R = 3;

# eps: epsilon for function values
eps = 1.0e-14;

# Initial guess x0
x0 = 1;

# Initialize er: error between values
er = 1;

# n_iter_max: max. number of iterations
n_ite = 0; n_iter_max = 50;

# Start the iteration
while er > eps && n_ite<n_iter_max
    # Evaluate the next point
    x1 = x0/2 + R/(2*x0);

    # Evaluate error, number of iterations
    er = abs(x1-x0);
    n_ite = n_ite + 1;

    # Initialize the next iteration
    x0 = x1;

    # Print the solution at n-th iteration and the exact error
    println("x1= ", x1, "\t\t", "error= ", abs(x1-sqrt(R)))
end

# Print total number of iterations
println("Total number of iterations=", n_ite)
```

```
x1= 2.0		error= 0.2679491924311228
x1= 1.75		error= 0.017949192431122807
x1= 1.7321428571428572		error= 9.204957398001312e-5
x1= 1.7320508100147274		error= 2.44585018904786e-9
x1= 1.7320508075688772		error= 0.0
x1= 1.7320508075688772		error= 0.0
Total number of iterations=6
```

Example 2: Square of a number but with difference function

$$f_2(x) = (x^2 - R)^2$$

Newton's iteration

$$x_{n+1} = x_n - \frac{(x_n^2 - R)^2}{4x_n(x_n^2 - R)} = \frac{3x_n}{4} + \frac{R}{4x_n}$$

```julia
# Setup parameters
R = 3;

# eps: epsilon for function values
eps = 1.0e-14;

# Initial guess x0
x0 = 1;

# Initialize er: error between values
er = 1;

# n_iter_max: max. number of iterations
n_ite = 0; n_iter_max = 50;

# Start the iteration
while er > eps && n_ite<n_iter_max
    # Evaluate the next point
    x1 = 3*x0/4 + R/(4*x0);

    # Evaluate error, number of iterations
    er = abs(x1-x0);
    n_ite = n_ite + 1;

    # Initialize the next iteration
    x0 = x1;

    # Print the solution at n-th iteration and the exact error
    println("x1= ", x1, "\t\t", "error= ", abs(x1-sqrt(R)))
end

# Print total number of iterations
println("Total number of iterations=", n_ite)
```

```
x1= 1.5     error= 0.2320508075688772
x1= 1.625       error= 0.1070508075688772
x1= 1.6802884615384617      error= 0.05176234603041552
x1= 1.7065682774843183      error= 0.02548253008455892
x1= 1.7194046690076297      error= 0.012646138561247522
x1= 1.7257509912233235      error= 0.006299816345553655
x1= 1.7289066487316345      error= 0.0031441588372427276
x1= 1.73048015762807l       error= 0.0015706499408061347
x1= 1.731265838993956       error= 0.0007849685749212743
x1= 1.7316584122590377      error= 0.00039239530983947724
x1= 1.7318546321432375      error= 0.0001961754256396553
x1= 1.7319527254114888      error= 9.808215738837944e-5
x1= 1.7320017678788049      error= 4.9039690072305575e-5
x1= 1.7320262880709671      error= 2.451949791004715e-5
x1= 1.7320385479067     error= 1.2259662177216413e-5
x1= 1.7320446777594827      error= 6.129809394517238e-6
x1= 1.7320477426696035      error= 3.0648992737081215e-6
x1= 1.7320492751205963      error= 1.5324482809386808e-6
x1= 1.7320500413450757      error= 7.66223801518251e-7
x1= 1.7320504244570611      error= 3.831118160491087e-7
x1= 1.7320506160129905      error= 1.9155588670827228e-7
x1= 1.7320507117909392      error= 9.577793802506562e-8
x1= 1.7320507596799095      error= 4.788896768026518e-8
x1= 1.732050783624394       error= 2.394448328502108e-8
x1= 1.7320507955966358      error= 1.1972241420465934e-8
x1= 1.7320508015827565      error= 5.986120710232967e-9
x1= 1.7320508045758167      error= 2.993060466138786e-9
x1= 1.732050806072347       error= 1.496530233069393e-9
x1= 1.7320508068206122      error= 7.482650055123941e-10
x1= 1.7320508071947447      error= 3.7413250275619703e-10
x1= 1.7320508073818108      error= 1.8706636240040098e-10
x1= 1.7320508074753442      error= 9.353295915559556e-11
x1= 1.7320508075221106      error= 4.6766590600100244e-11
x1= 1.732050807545494       error= 2.3383295300050122e-11
x1= 1.7320508075571857      error= 1.1691536627722598e-11
x1= 1.7320508075630314      error= 5.845768313861299e-12
x1= 1.7320508075659542      error= 2.922995179233112e-12
x1= 1.7320508075674157      error= 1.461497589616556e-12
x1= 1.7320508075681464      error= 7.30748794808278e-13
x1= 1.7320508075685117      error= 3.6548541970660153e-13
x1= 1.7320508075686945      error= 1.8274270985330077e-13
x1= 1.7320508075687857      error= 9.14823772291129e-14
x1= 1.7320508075688315      error= 4.574118861455645e-14
x1= 1.7320508075688543      error= 2.2870594307278225e-14
x1= 1.7320508075688656      error= 1.1546319456101628e-14
x1= 1.7320508075688714      error= 5.773159728050814e-15
Total number of iterations=46
```

Example 3: Square of a number but with difference function

$$f_3(x) = x^4 - R^2$$

Newton's iteration

$$x_{n+1} = x_n - \frac{(x_n^4 - R^2)}{4x_n^3} = \frac{3x_n}{4} + \frac{R^2}{4x_n^3}$$

Julia

```julia
# Setup parameters
R = 3;

# eps: epsilon for function values
eps = 1.0e-14;

# Initial guess x0
x0 = 1;

# Initialize er: error between values
er = 1;

# n_iter_max: max. number of iterations
n_ite = 0; n_iter_max = 50;

# Start the iteration
while er > eps && n_ite<n_iter_max
    # Evaluate the next point
    x1 = 3*x0/4 + R^2/(4*x0^3);

    # Evaluate error, number of iterations
    er = abs(x1-x0);
    n_ite = n_ite + 1;

    # Initialize the next iteration
    x0 = x1;

    # Print the solution at n-th iteration and the exact error
    println("x1= ", x1, "\t\t", "error= ", abs(x1-sqrt(R)))
end

# Print total number of iterations
println("Total number of iterations=", n_ite)
```

```
x1= 3.0     error= 1.2679491924311228
x1= 2.3333333333333335      error= 0.6012825257644563
x1= 1.9271137026239067      error= 0.1950628950550295
x1= 1.75971932364339        error= 0.027668516074512706
x1= 1.732696552935594       error= 0.0006457453667167989
x1= 1.7320511684660165      error= 3.608971392843333e-7
x1= 1.73205080756899        error= 1.127986593019159e-13
x1= 1.7320508075688772      error= 0.0
x1= 1.7320508075688774      error= 2.220446049250313e-16
Total number of iterations=9
```

Example 4:

$$f_4(x) = \frac{1}{\sqrt{x^2 - R}}$$

Newton's iteration

$$x_{n+1} = x_n - \frac{\frac{1}{\sqrt{x_n^2 - R}}}{\frac{-x_n}{(x_n^2-R)^{2/3}}} = 2x_n - \frac{R}{x_0}$$

```julia
# Setup parameters
R = 3;

# eps: epsilon for function values
eps = 1.0e-14;

# Initial guess x0
x0 = 1;

# Initialize er: error between values
er = 1;

# n_iter_max: max. number of iterations
n_ite = 0; n_iter_max = 50;

# Start the iteration
while er > eps && n_ite<n_iter_max
    # Evaluate the next point
    x1 = 2*x0 - R/x0;

    # Evaluate error, number of iterations
    er = abs(x1-x0);
    n_ite = n_ite + 1;

    # Initialize the next iteration
    x0 = x1;

    # Print the solution at n-th iteration and the exact error
    println("x1= ", x1, "\t\t", "error= ", abs(x1-sqrt(R)))
end

# Print total number of iterations
println("Total number of iterations=", n_ite)
```

```
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
x1= 1.0     error= 0.7320508075688772
x1= -1.0        error= 2.732050807568877
Total number of iterations=50
```

This loop goes on forever.

[2.2] Secant method (割線法)

Secant method can be thought as an extension of Newton's method, and also share some idea of the method of false position. In Newton's method, we require the knowledge of the function value as well as its derivative, so that one can draw the tangent line at a given point and find the root of that line. However, there might be circumstance where the derivative is not known or not easy to obtain so that we don't have the tangent line exactly. The idea of secant method is to approximate this tangent line using the function values at two points, then we find the root of this approximated tangent line as the next guess.

Suppose we are given the points $(x_{n-1}, f(x_{n-1}))$ and $(x_n, f(x_n))$, we can then draw a line passing through this two points and look for the intersection between this approximated tangent line and the x-axis. This should be a good guess of the root and this is exactly the idea of secant method.

[2.2.1] Secant method is given as the following:

Given initial guesses x_0 and x_1, for $n = 1, 2, \cdots$

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

and iterate until $|x_{n+1} - x_n|$ is small enough or we exceeded the maxium number of iterations.

Reference: [Secant method]

Example 5: Square root of a number:

$$f_1(x) = x^2 - R$$

Secant iteration:

Initial guess: $x_0 = 0, x_1 = R$.

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} = x_n - (x_n^2 - R)\frac{(x_n - x_{n-1})}{(x_n^2 - R) - (x_{n-1}^2 - R)} = \frac{x_n x_{n-1} + R}{x_n + x_{n-1}}$$

```julia
# Setup parameters
R = 3;

# eps: epsilon for function values
eps = 1.0e-14;

# Initial guess x0 and x1
x0 = 1;
x1 = 2;

# Initialize er: error between values
er = 1;

# n_iter_max: max. number of iterations
n_ite = 0; n_iter_max = 50;

# Start the iteration
while er > eps && n_ite<n_iter_max
    # Evaluate the next point
    x2 = (x0*x1+R)/(x0+x1);

    # Evaluate error, number of iterations
    er = abs(x2-x1);
    n_ite = n_ite + 1;

    # Initialize the next iteration
    x0 = x1;
    x1 = x2;

    # Print the solution at n-th iteration and the exact error
    println("x2= ", x2, "\t\t", "error= ", abs(x2-sqrt(R)))
end

# Print total number of iterations
println("Total number of iterations=", n_ite)
```

```
x2= 1.6666666666666667      error= 0.06538414090221045
x2= 1.7272727272727273      error= 0.0047780802961499
x2= 1.7321428571428572      error= 9.204957398001312e-5
x2= 1.7320506804317224      error= 1.2713715480394683e-7
x2= 1.7320508075654992      error= 3.3779645747245013e-12
x2= 1.7320508075688772      error= 0.0
x2= 1.7320508075688772      error= 0.0
Total number of iterations=7
```

Summary

So, as a brief summary. We have shown several examples of finding the root, including bisection, false position, Newton's and secant method.

Regarding bracketing methods, we see that the bisection and false position method both converges to the true solution. Also, false position method seems to be a little bit faster in convergence. So the questions are the following: * Can we show/prove that the bisection and false position method indeed converge to the solution? * Is it true that false position method converges faster than Bisection method?

The second set of questions concerns about the Newton's method: * In what circumstances does the Newton's method converges? * If the Newton's iteration converges, what is the rate of convergence?