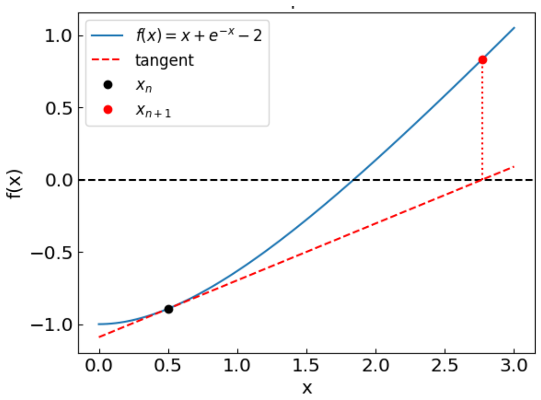
Computational Physics (PHYS6350)

Lecture 4: Non-linear equations and root-finding



January 24, 2023

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Non-linear equations

Suppose we have an equation *f(x)* $= 0$

We can evaluate *f(x)*, but we do not know how to solve it for *x*

Examples:

- Roots of high-order polynomials (e.g. Lagrange L_1 point)
- Transcendental equations
  - e.g. magnetization equation

$$M = \mu \tanh \frac{JM}{k_B T}$$

References: Chapter 6 of *Computational Physics* by Mark Newman

Chapter 9 of *Numerical Recipes Third Edition* by W.H. Press et al.

Root-finding techniques

Numerical root-finding method: iterative process to determine the root(s) of non-linear equation(s) to desired accuracy

Types:

- Two-point (bracketing)
  - Bisection method
  - False position method

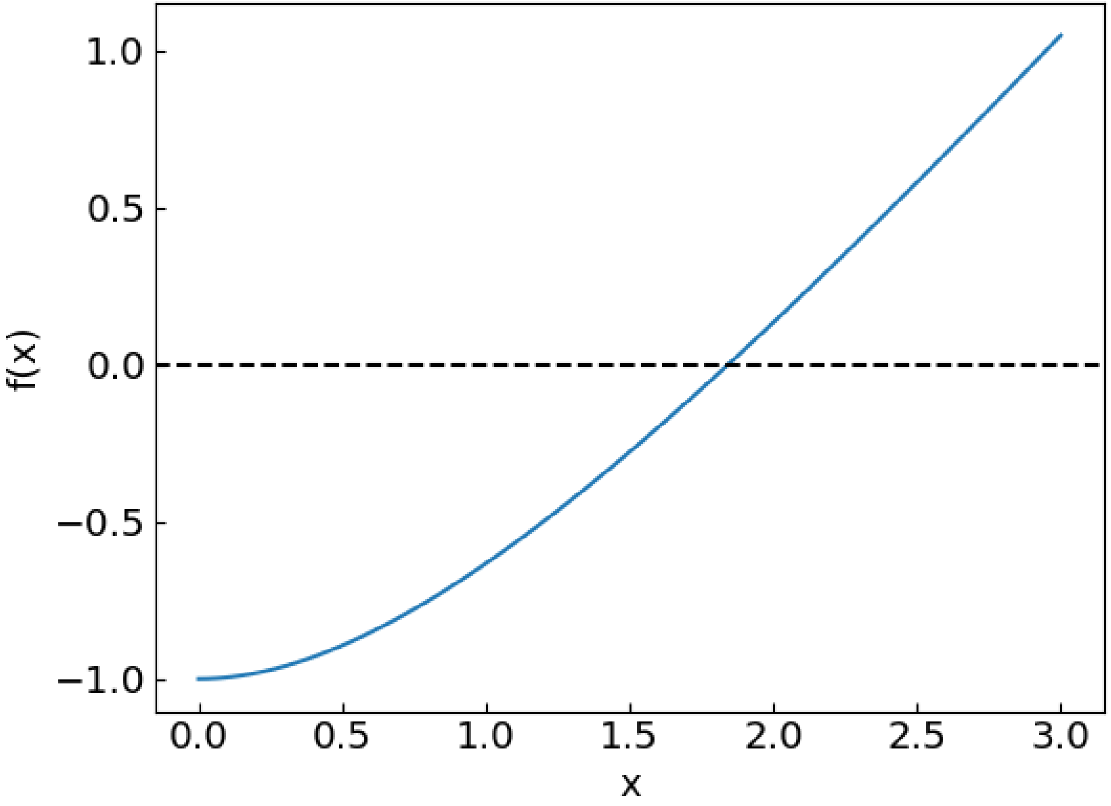
- Local
  - Secant method
  - Newton-Raphson method (using the derivative)
  - Relaxation method

- Multi-dimensional
  - Newton method
  - Broyden method

Non-linear equations

Consider an equation

$$x + e^{-x} - 2 = 0$$



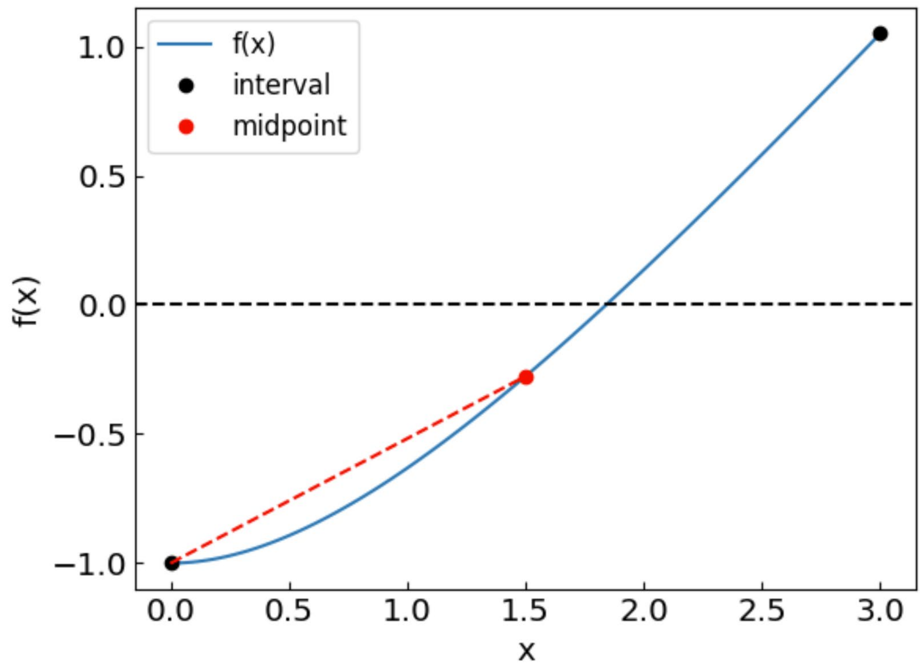
Bisection method

Bisection method:

1. Find an interval (a,b) which brackets the root x*
   - x* ∈ (a,b)
   - f(a) & f(b) have opposite signs

2. Take the midpoint $c = (a+b)/2$ and halve the interval bracketing the root

3. Repeat the process until the desired precision is achieved

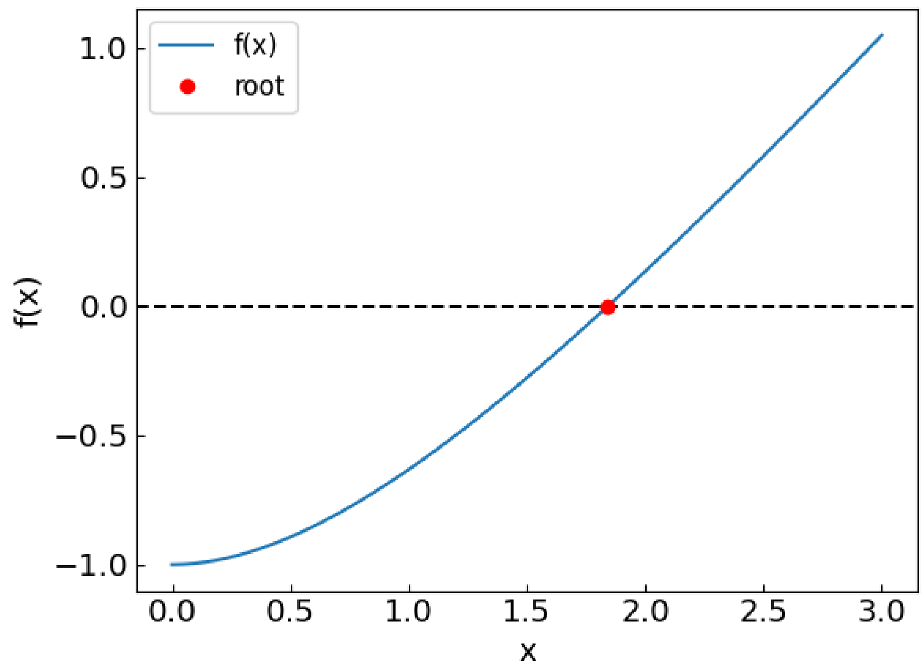


Method is guaranteed to converge to the root
The error is halved at each step – "linear" convergence

Bisection method

```python
def bisection_method(
    f,                      # The function whose root we are trying to find
    a,                      # The left boundary
    b,                      # The right boundary
    tolerance = 1.e-10,     # The desired accuracy of the solution
    ):
    fa = f(a)                             # The value of the function at the left boundary
    fb = f(b)                             # The value of the function at the right boundary
    if (fa * fb > 0.):
        return None                       # Bisection method is not applicable

    global last_bisection_iterations
    last_bisection_iterations = 0


    while ((b-a) > tolerance):
        last_bisection_iterations += 1
        c = (a + b) / 2.                  # Take the midpoint
        fc = f(c)                         # Calculate the function at midpoint


        if (fc * fa < 0.):
            b = c                         # The midpoint is the new right boundary
            fb = fc
        else:
            a = c                         # The midpoint is the new left boundary
            fa = fc

    return (a+b) / 2.
```

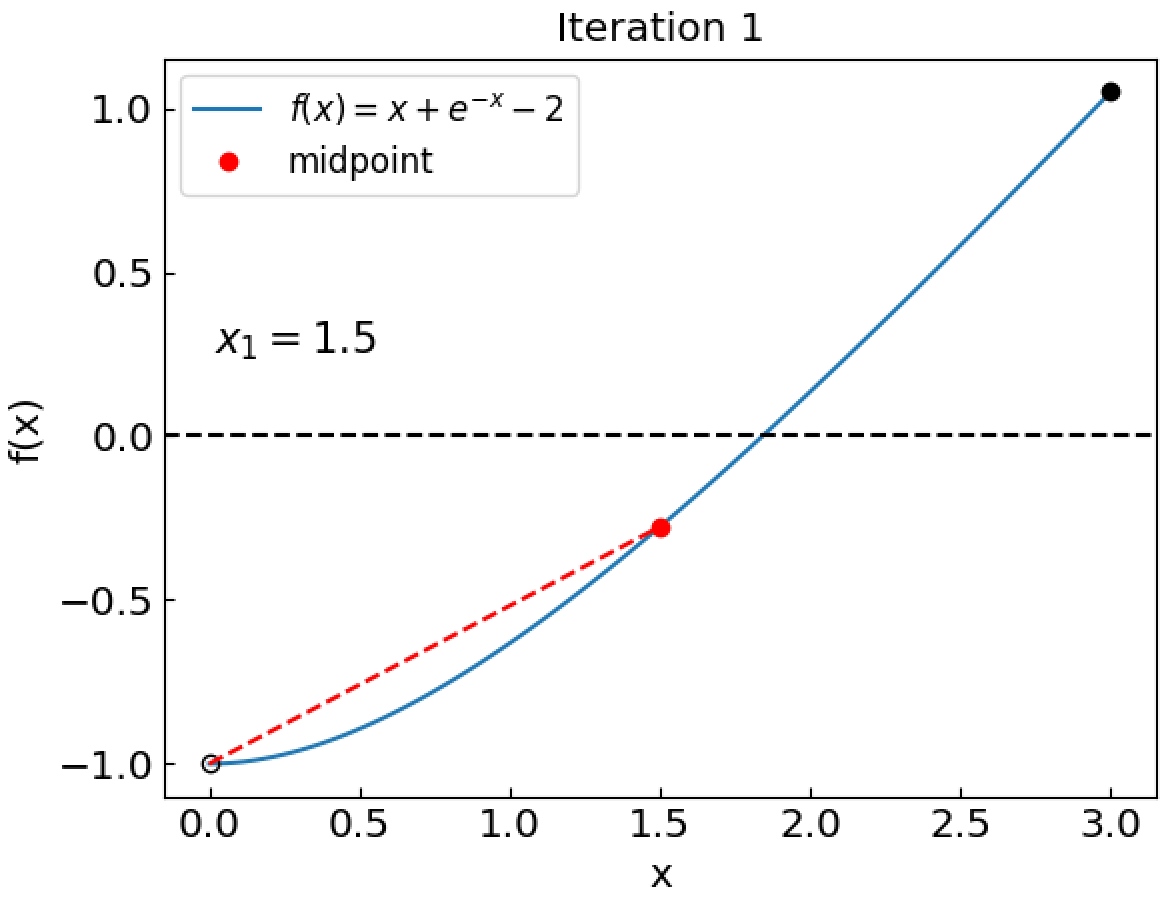
$$x + e^{-x} - 2 = 0$$




```
Solving the equation x + e^-x - 2 = 0 on an interval ( 0.0 , 3.0 ) using bisection method
The solution is x =  1.8414056604233338  obtained with  35  iterations
```

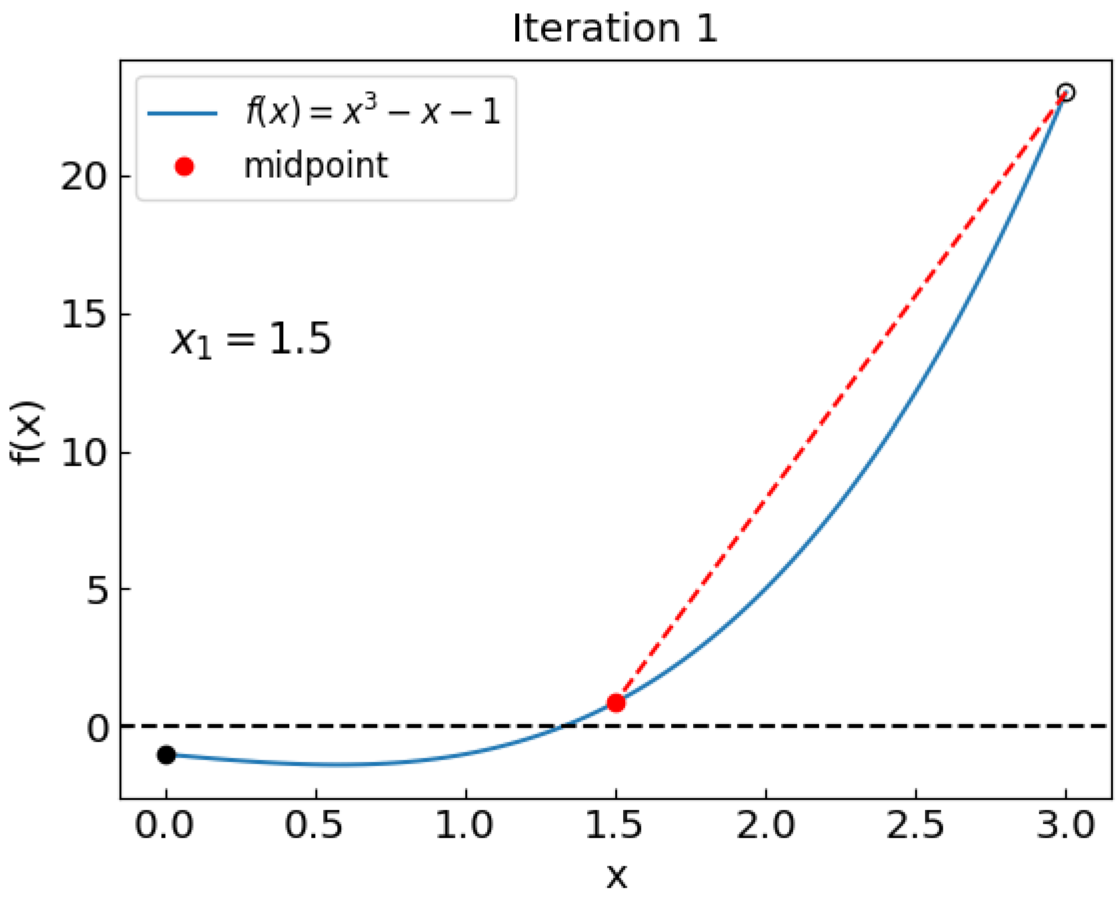
Bisection method: how the iterations look like

$$x + e^{-x} - 2 = 0$$



Bisection method: another

Let us consider another equation: $x^3 - x - 1 = 0$



35 iterations in both cases

False position method

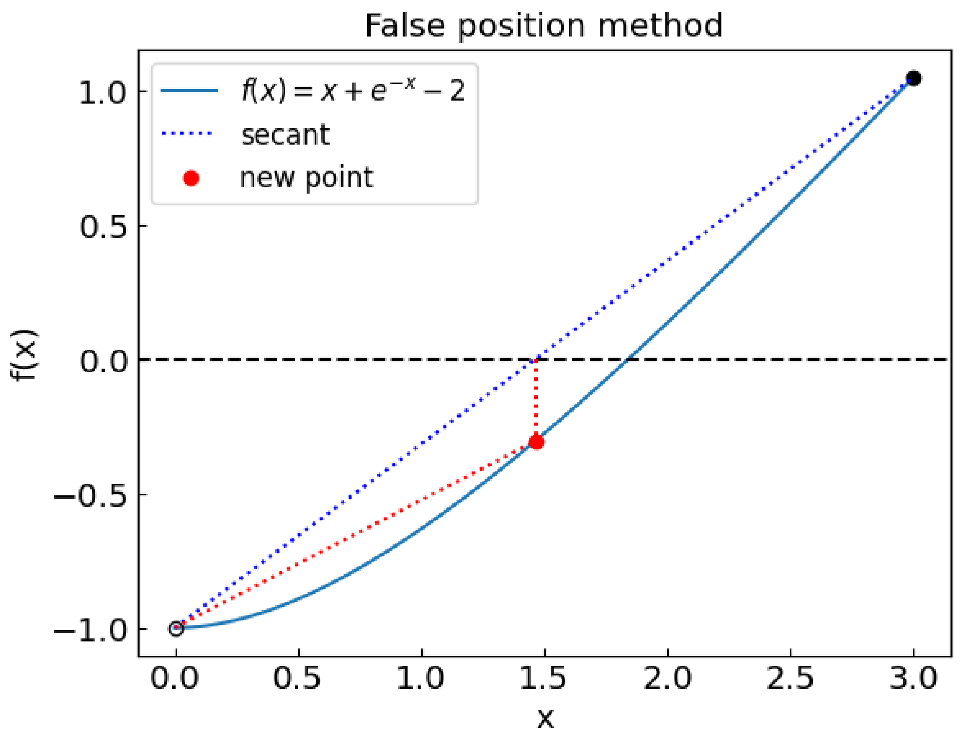
False position method:

1. Find an interval (a,b) which brackets the root x*, same bisection

2. Instead of midpoint take a point where the straight line between the endpoints crosses y = 0 axis

$$c = a - f(a)\frac{b-a}{f(b)-f(a)}$$

3. Repeat the process until the desired precision is achieved

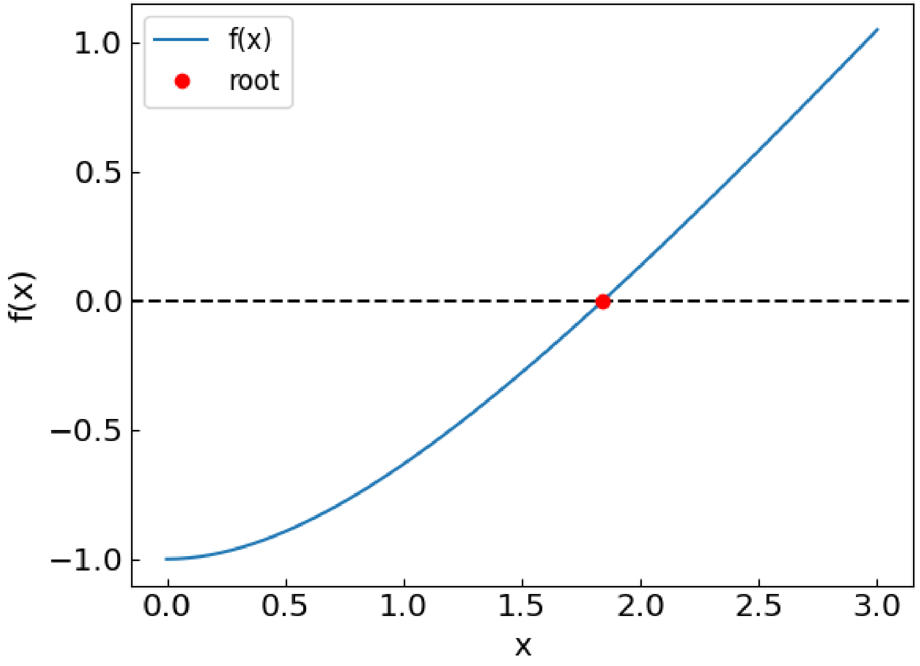


Method is guaranteed to converge to the root
"Linear" convergence; typically faster than bisection, but not always

False position method

```python
def falseposition_method(
    f,                      # The function whose root we are trying to find
    a,                      # The left boundary
    b,                      # The right boundary
    tolerance = 1.e-10,     # The desired accuracy of the solution
    max_iterations = 100    # Maximum number of iterations
    ):
    fa = f(a)                              # The value of the function at the left boundary
    fb = f(b)                              # The value of the function at the right boundary
    if (fa * fb > 0.):
        return None                        # False position method is not applicable

    xprev = xnew = (a+b) / 2.                      # Estimate of the solution from the previous step

    global last_falseposition_iterations
    last_falseposition_iterations = 0

    for i in range(max_iterations):
        last_falseposition_iterations += 1

        xprev = xnew
        xnew = a - fa * (b - a) / (fb - fa) # Take the point where straight line between a and b crosses y = 0
        fnew = f(xnew)                      # Calculate the function at midpoint

        if (fnew * fa < 0.):
            b = xnew                        # The intersection is the new right boundary
            fb = fnew
        else:
            a = xnew                        # The midpoint is the new left boundary
            fa = fnew

        if (abs(xnew-xprev) < tolerance):
            return xnew

    print("False position method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew - xprev))

    return xnew
```

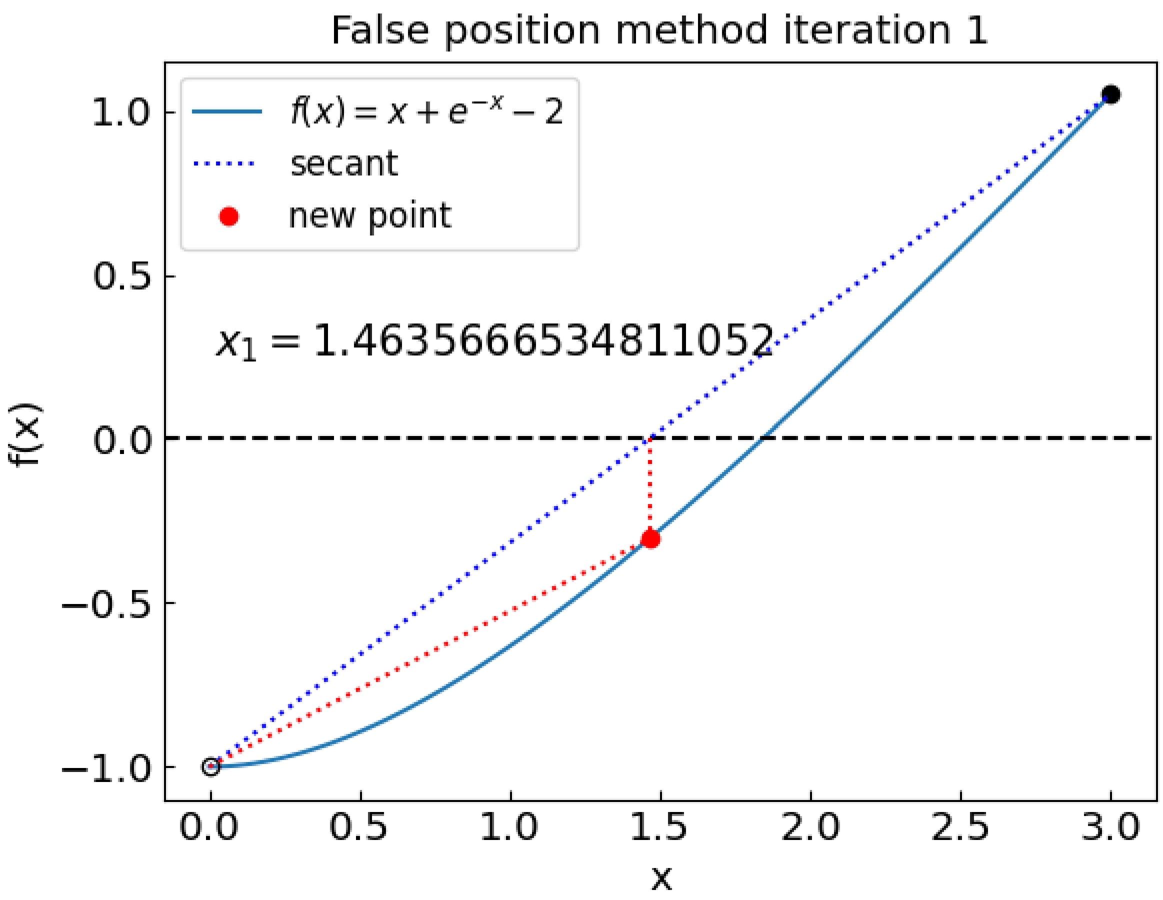
$$x + e^{-x} - 2 = 0$$




```
Solving the equation x + e^-x - 2 = 0 on an interval ( 0.0 , 3.0 ) using the false position method
The solution is x =  1.8414056604354012 obtained after  11  iterations
```

False position method

$$x + e^{-x} - 2 = 0$$



False position vs bisection (to 10 decimal digits)

$$x + e^{-x} - 2 = 0$$

Bisection method:

```
Iteration:     1, c =    1.500000000000000
Iteration:     2, c =    2.250000000000000
Iteration:     3, c =    1.875000000000000
Iteration:     4, c =    1.687500000000000
Iteration:     5, c =    1.781250000000000
Iteration:     6, c =    1.828125000000000
Iteration:     7, c =    1.851562500000000
Iteration:     8, c =    1.839843750000000
Iteration:     9, c =    1.845703125000000
Iteration:    10, c =    1.842773437500000
Iteration:    11, c =    1.841308593750000
Iteration:    12, c =    1.842041015625000
Iteration:    13, c =    1.841674804687500
Iteration:    14, c =    1.841491699218750
Iteration:    15, c =    1.841400146484375
Iteration:    16, c =    1.841445922851562
Iteration:    17, c =    1.841423034667969
Iteration:    18, c =    1.841411590576172
Iteration:    19, c =    1.841405868530273
Iteration:    20, c =    1.841403007507324
                   ...
Iteration:    35, c =    1.841405660466990
```

False position method:

```
Iteration:     1, x =    1.463566653481105
Iteration:     2, x =    1.809481253839539
Iteration:     3, x =    1.839095511827520
Iteration:     4, x =    1.841240588240115
Iteration:     5, x =    1.841393875903701
Iteration:     6, x =    1.841404819191791
Iteration:     7, x =    1.841405600384506
Iteration:     8, x =    1.841405656150106
Iteration:     9, x =    1.841405660130943
Iteration:    10, x =    1.841405660415115
Iteration:    11, x =    1.841405660435401
```

False position vs bisection: not always clear who wins

$$x^3 - x - 1 = 0$$

Bisection method:

```
Iteration:     1, c =    1.500000000000000
Iteration:     2, c =    0.750000000000000
Iteration:     3, c =    1.125000000000000
Iteration:     4, c =    1.312500000000000
Iteration:     5, c =    1.406250000000000
Iteration:     6, c =    1.359375000000000
Iteration:     7, c =    1.335937500000000
Iteration:     8, c =    1.324218750000000
Iteration:     9, c =    1.330078125000000
Iteration:    10, c =    1.327148437500000
Iteration:    11, c =    1.325683593750000
Iteration:    12, c =    1.324951171875000
Iteration:    13, c =    1.324584960937500
Iteration:    14, c =    1.324768066406250
Iteration:    15, c =    1.324676513671875
Iteration:    16, c =    1.324722290039062
Iteration:    17, c =    1.324699401855469
Iteration:    18, c =    1.324710845947266
Iteration:    19, c =    1.324716567993164
Iteration:    20, c =    1.324719429016113
                ...
Iteration:    35, c =    1.324717957206303
```

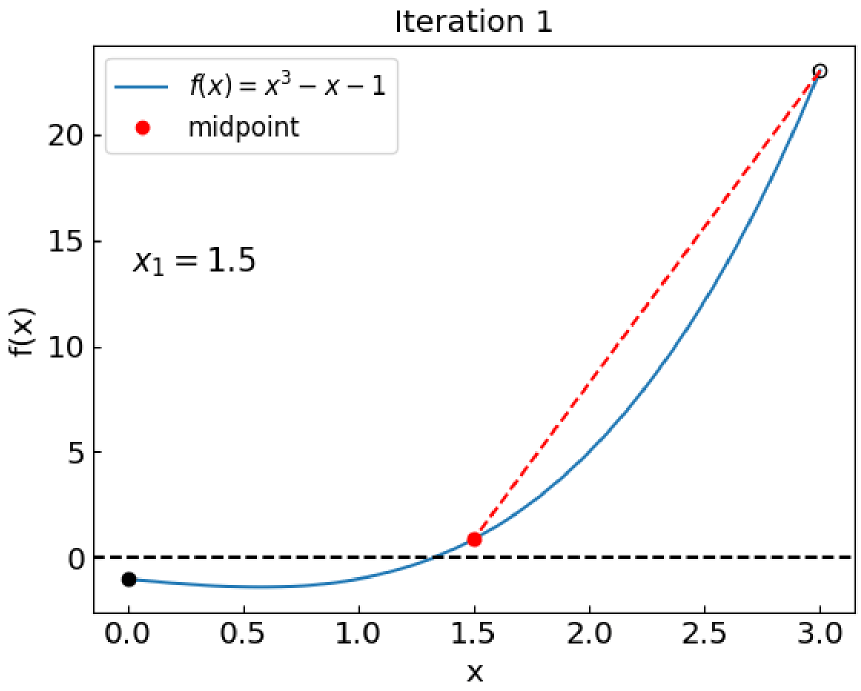
False position method:

```
Iteration:     1, x =    0.125000000000000
Iteration:     2, x =    0.258845437616387
Iteration:     3, x =    0.399230727605107
Iteration:     4, x =    0.541967526475374
Iteration:     5, x =    0.681365453934702
Iteration:     6, x =    0.811265467641601
Iteration:     7, x =    0.926423756077868
Iteration:     8, x =    1.023635980751716
Iteration:     9, x =    1.102112700940041
Iteration:    10, x =    1.163084623011103
Iteration:    11, x =    1.209004461867383
Iteration:    12, x =    1.242759715838447
Iteration:    13, x =    1.267123755869329
Iteration:    14, x =    1.284474915416815
Iteration:    15, x =    1.296712725379603
Iteration:    16, x =    1.305284823099690
Iteration:    17, x =    1.311260149895704
Iteration:    18, x =    1.315411216706803
Iteration:    19, x =    1.318288144277179
Iteration:    20, x =    1.320278742279728
                ...
Iteration:    66, x =    1.324717957079699
```

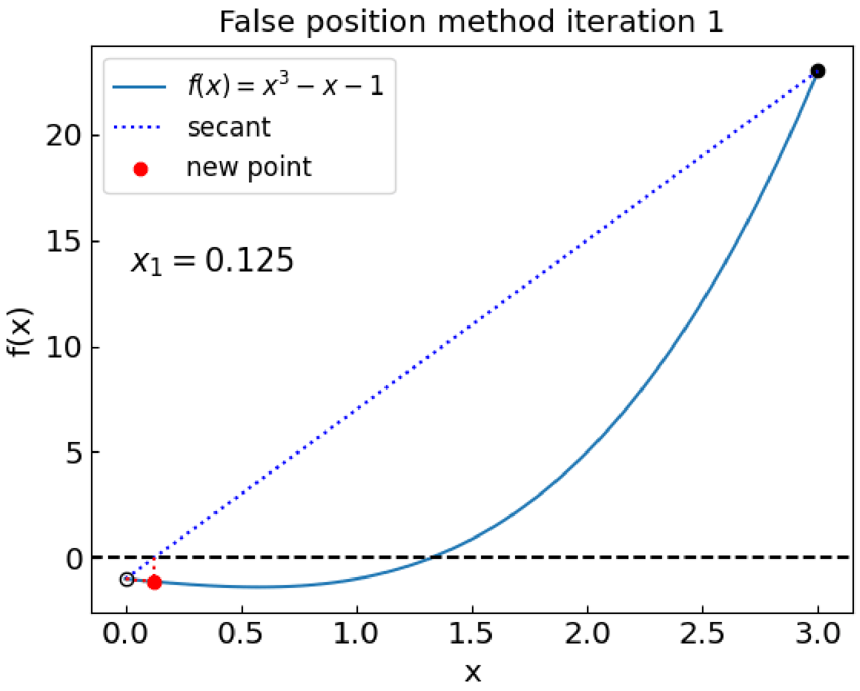
False position vs bisection: not always clear who wins

$$x^3 - x - 1 = 0$$

Bisection method:



False position method:



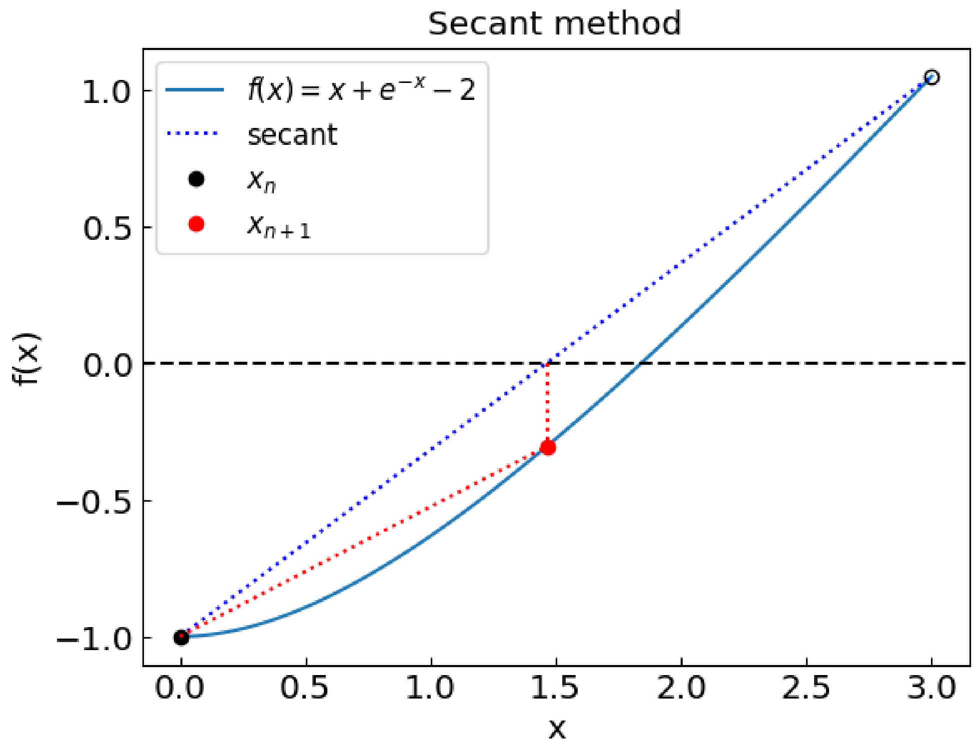
More advanced methods combine the two and add other refinements*

- Ridders' method
- Brent method

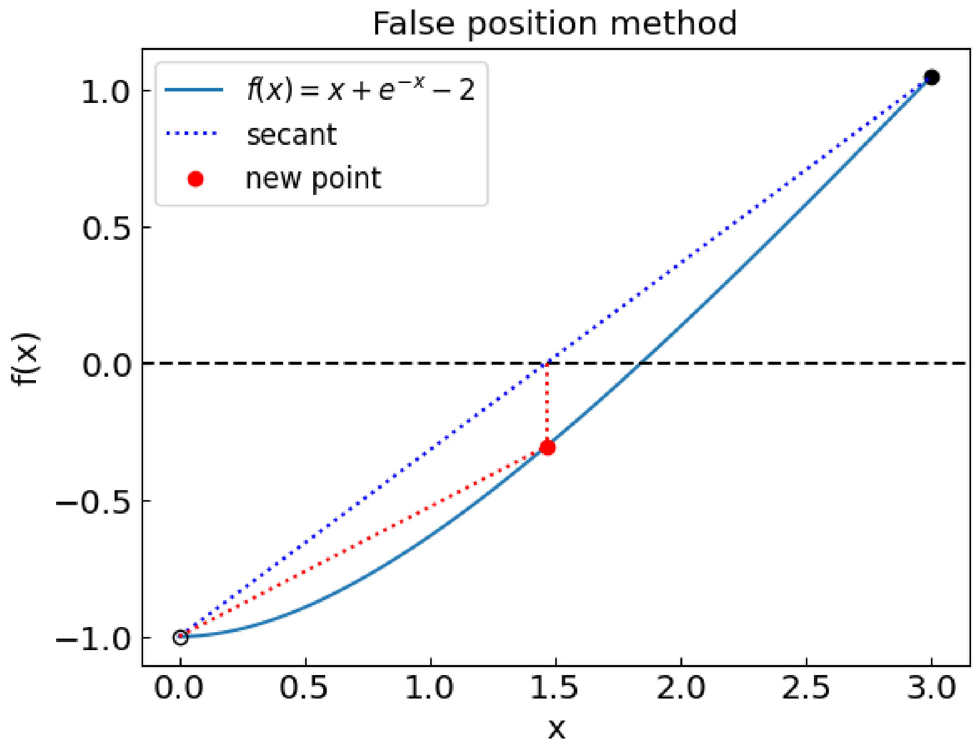
see chapters 9.2, 9.3 of *Numerical Recipes Third Edition* by W.H. Press et al.

Secant method

Secant method: similar to false position, but the interval *need not bracket the root*
Always uses the last two points



vs

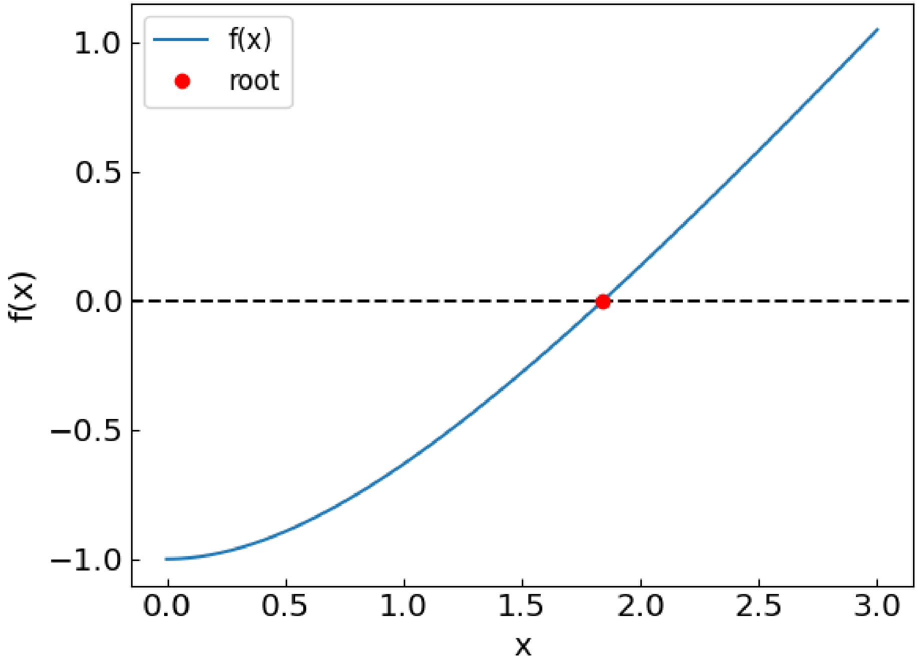


Typically "superlinear" convergence when works
Can still be slower than bisection or not converge at all (e.g. secant is parallel to $y = 0$ axis)

Secant method

```
def secant_method(
    f,                     # The function whose root we are trying to find
    a,                     # The left boundary
    b,                     # The right boundary
    tolerance = 1.e-10,    # The desired accuracy of the solution
    max_iterations = 100   # Maximum number of iterations
    ):
    fa = f(a)                             # The value of the function at the left boundary
    fb = f(b)                             # The value of the function at the right boundary

    xprev = xnew = a                             # Estimate of the solution from the previous step

    global last_secant_iterations
    last_secant_iterations = 0

    for i in range(max_iterations):
        last_secant_iterations += 1

        xprev = xnew
        xnew = a - fa * (b - a) / (fb - fa) # Take the point where straight line between a and b crosses y = 0
        fnew = f(xnew)                      # Calculate the function at midpoint

        b = a
        fb = fa
        a = xnew
        fa = fnew

        if (abs(xnew-xprev) < tolerance):
            return xnew

    print("Secant method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew - xprev))

    return xnew
```

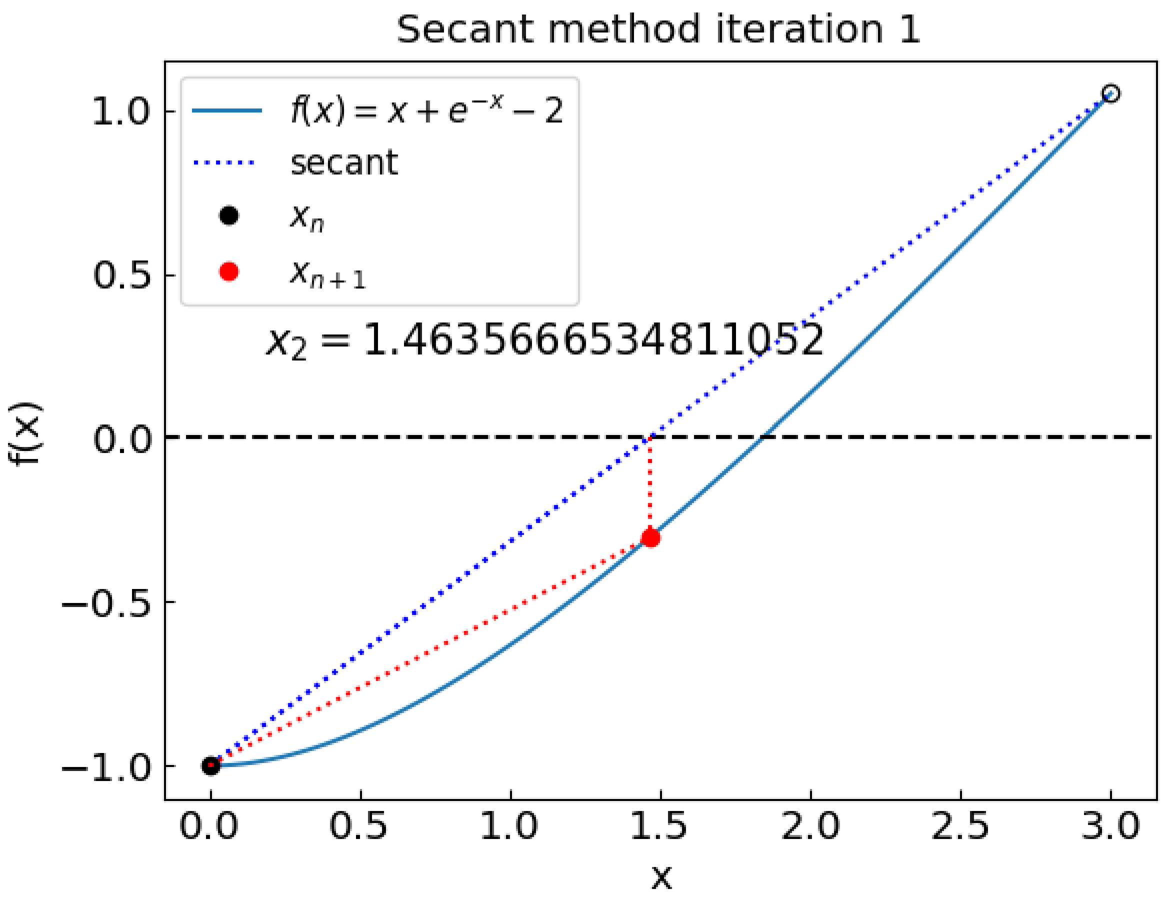
$$x + e^{-x} - 2 = 0$$




```
Solving the equation x + e^-x - 2 = 0 on an interval ( 0.0 , 3.0 ) using the secant method
The solution is x =  1.8414056604369606 obtained after  7  iterations
```

Secant method

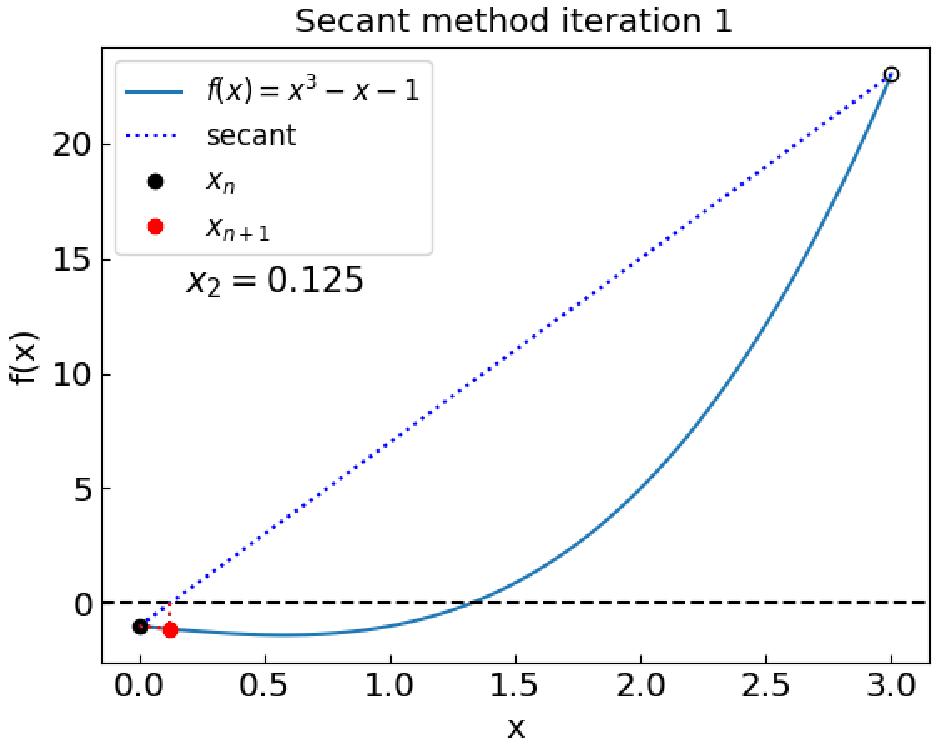
$$x + e^{-x} - 2 = 0$$



Secant method

$$x^3 - x - 1 = 0$$

```
Iteration:    1, x =    0.125000000000000
Iteration:    2, x =   -1.015873015873016
Iteration:    3, x =  -14.026092564115256
Iteration:    4, x =   -1.010979901305751
Iteration:    5, x =   -1.006133240911884
Iteration:    6, x =   -0.512666258317272
Iteration:    7, x =    0.273834681149844
Iteration:    8, x =   -1.287767830907429
Iteration:    9, x =    3.565966235528240
Iteration:   10, x =   -1.077368321415013
Iteration:   11, x =   -0.947522156044583
Iteration:   12, x =   -0.513174359589628
Iteration:   13, x =    0.447558454314033
Iteration:   14, x =   -1.325124217388110
Iteration:   15, x =    4.186373891812861
Iteration:   16, x =   -1.167930924631363
Iteration:   17, x =   -1.058303471905222
Iteration:   18, x =   -0.643978481189561
Iteration:   19, x =   -0.131674045244213
Iteration:   20, x =   -1.933586024088406
Iteration:   21, x =    0.157497929951306
Iteration:   22, x =    0.626623389695762
Iteration:   23, x =   -2.226715128003442
Iteration:   24, x =    1.093727500240917
Iteration:   25, x =    1.382563036703896
Iteration:   26, x =    1.310687668369503
Iteration:   27, x =    1.323983763313963
Iteration:   28, x =    1.324727653842468
Iteration:   29, x =    1.324717950607204
Iteration:   30, x =    1.324717957244686
Iteration:   31, x =    1.324717957244746
```



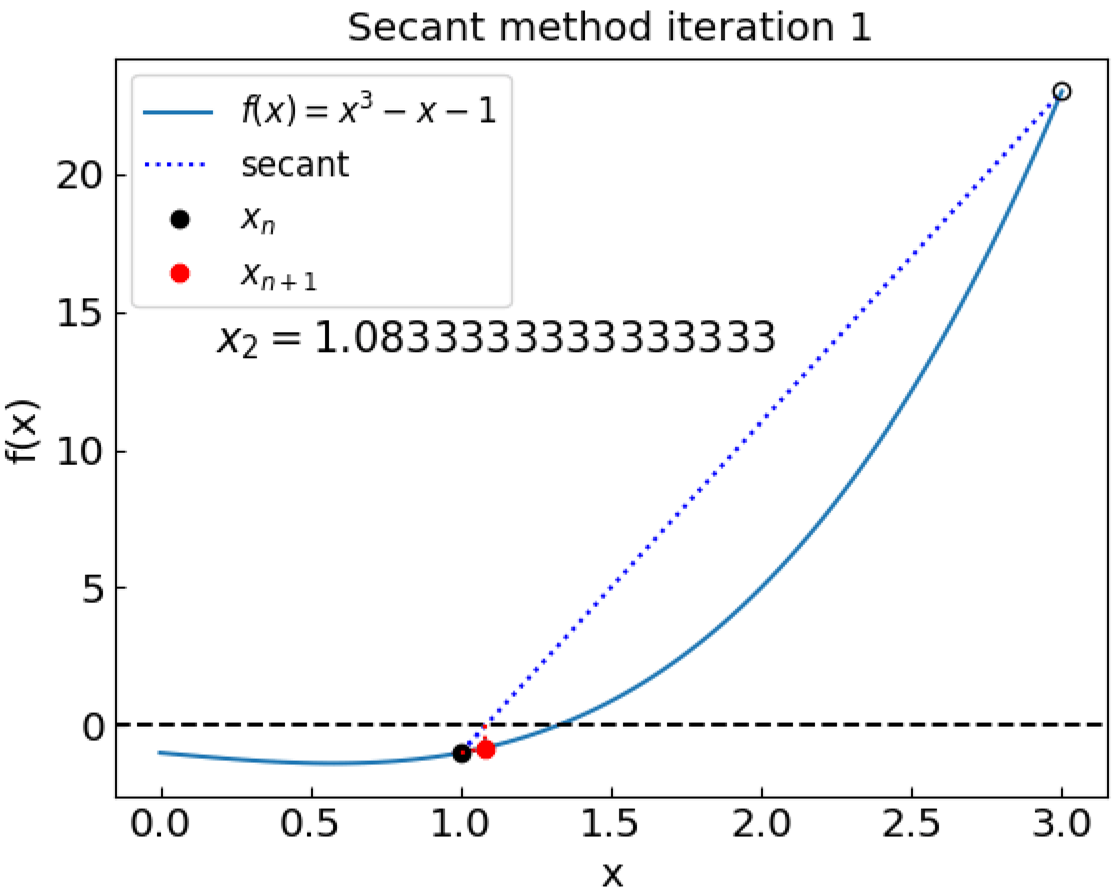
Because the method does not bracket the root, it is not guaranteed to converge

In this case, it managed to recover

Secant method

$$x^3 - x - 1 = 0$$

Choose the initial interval as (1,3) instead of (0,3)



Newton-Raphson method

Newton-Raphson method:

- Local method (uses only the current estimate to get the next one)
- Requires the evaluation of derivative

Idea: Assume that a given point x is close to the root x* [f(x*)=0]

Then

$$f(x^*) \approx f(x) + f'(x)(x^* - x)$$

and since f(x*) = 0 we have

$$x^* \approx x - \frac{f(x)}{f'(x)}$$

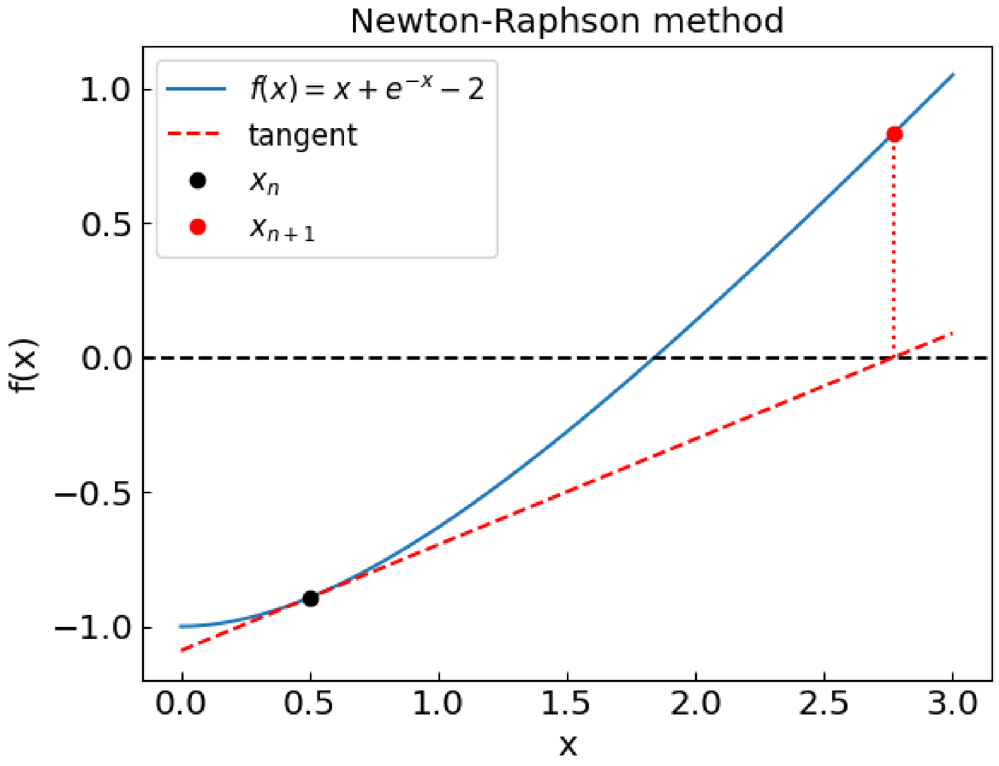
Iterative procedure:

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

starting from initial guess x_0

Newton-Raphson method

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



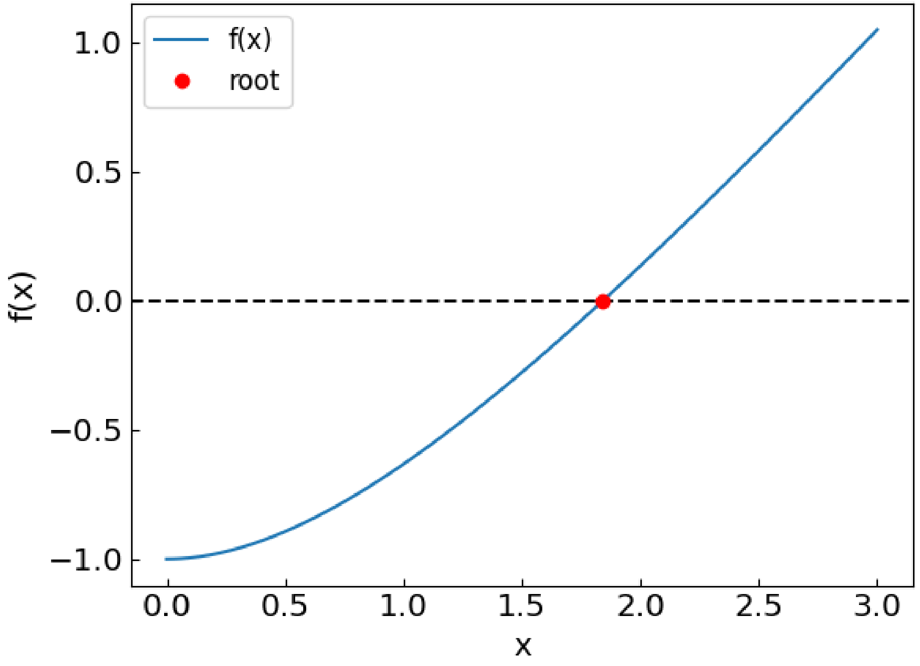
"Quadratic" convergence when works

When we are close to f' = 0 we have a problem

Newton-Raphson method

```python
def newton_method(
    f,                      # The function whose root we are trying to find
    df,                     # The derivative of the function
    x0,                     # The initial guess
    tolerance = 1.e-10,     # The desired accuracy of the solution
    max_iterations = 100    # Maximum number of iterations
    ):

    xprev = xnew = x0

    global last_newton_iterations
    last_newton_iterations = 0
    diff = 0.

    for i in range(max_iterations):
        last_newton_iterations += 1

        xprev = xnew
        fval  = f(xprev)                        # The current function value
        dfval = df(xprev)                       # The current function derivative value

        xnew = xprev - fval / dfval             # The next iteration

        if (abs(xnew-xprev) < tolerance):
            return xnew


    print("Newton-Raphson method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew-xprev))

    return xnew
```

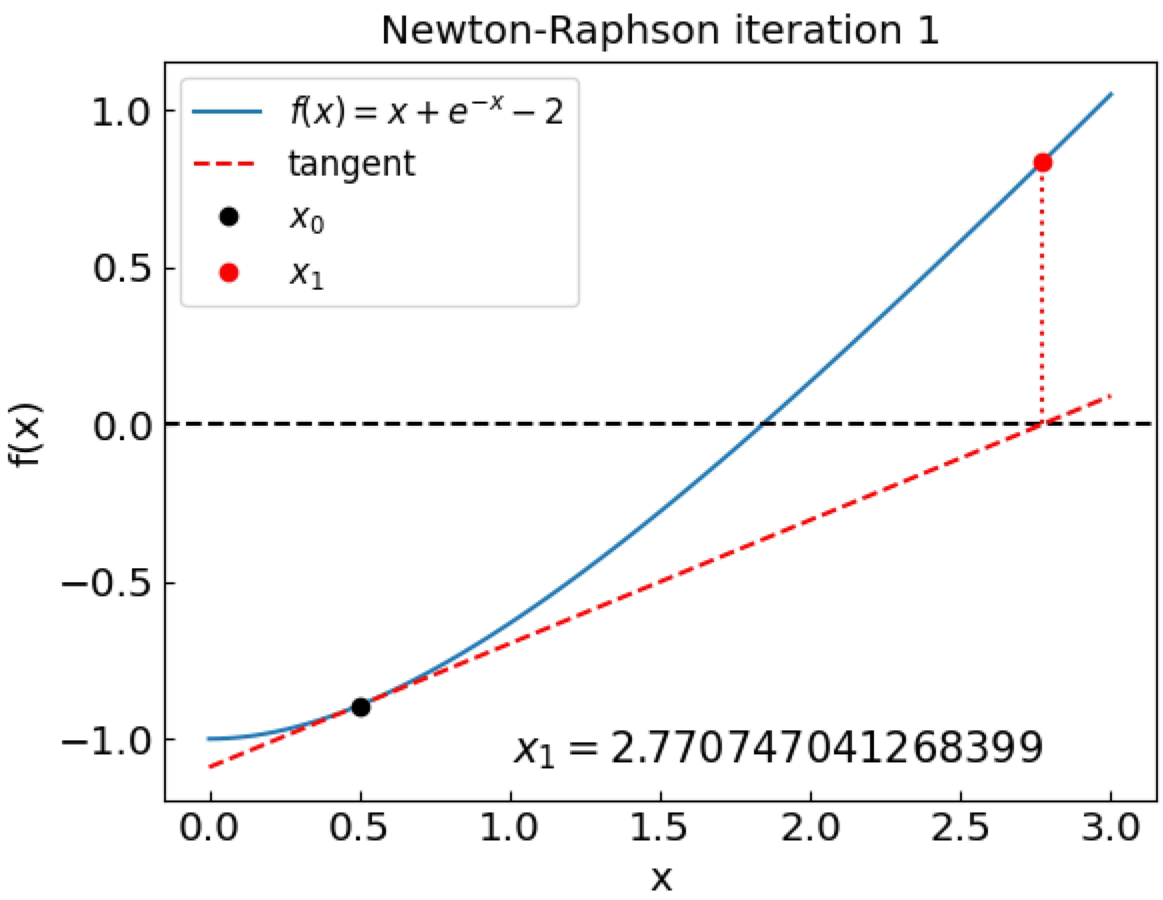
$$x + e^{-x} - 2 = 0$$




```
Solving the equation x + e^-x - 2 = 0 with an initial guess of x0 =  0.5
The solution is x =  1.8414056604369606 obtained after  6  iterations
```

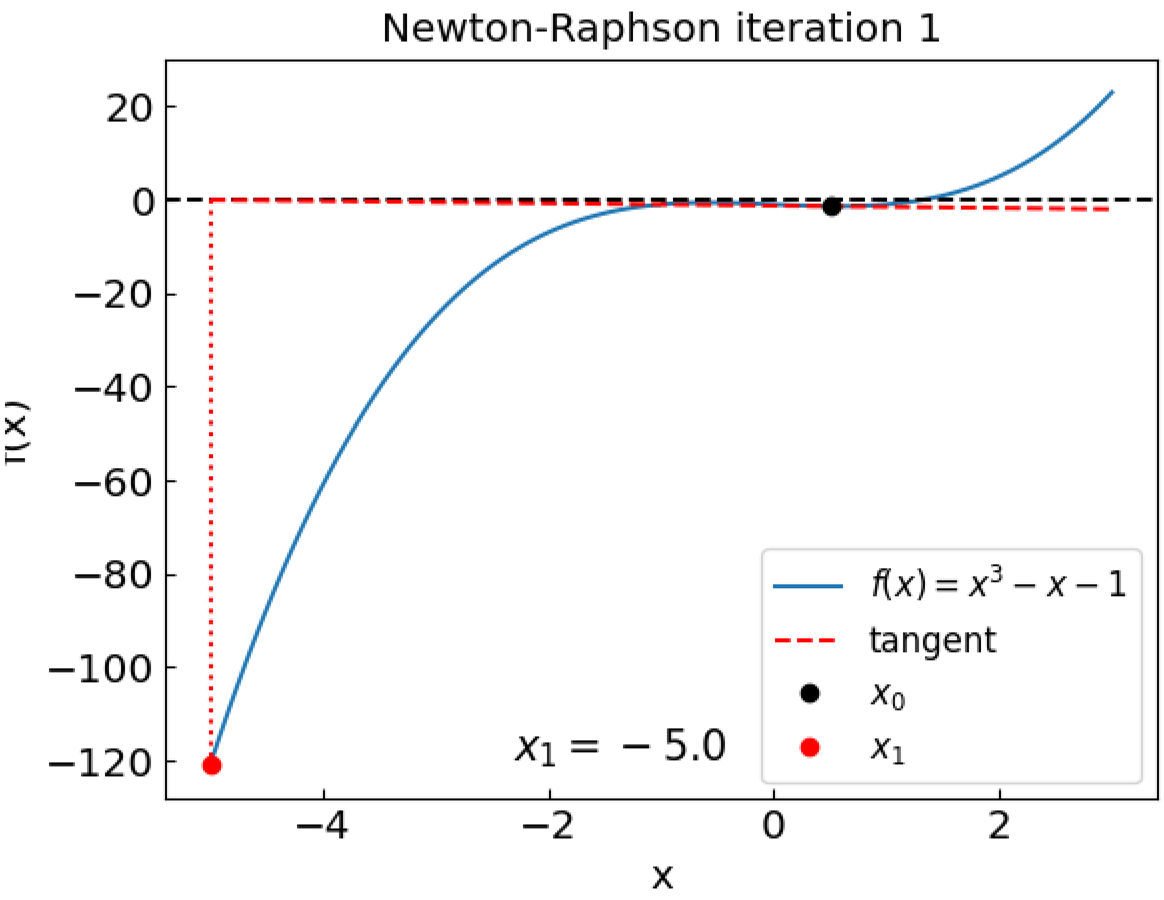
Newton-Raphson method

$$x + e^{-x} - 2 = 0$$



Newton-Raphson method: issues

$$x^3 - x - 1 = 0$$



Similar issue as with secant method; reason: f' $= 0$ at x $= 0.577...$

Newton-Raphson method: issues

Try finding the root of $f(x) = x^3 - 2x + 2$ with an initial guess of $x_0 = 0$

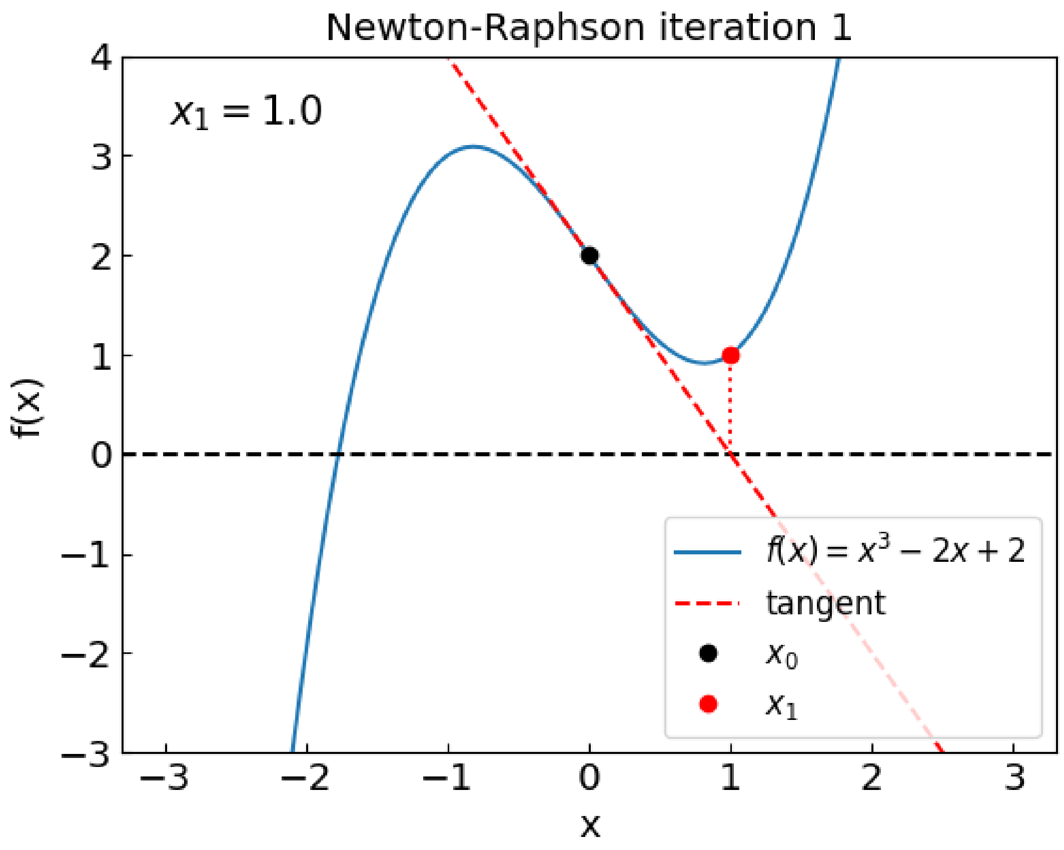
Iteration 1: $f(x_0) = 2, \quad f'(x_0) = -2$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 1$$

Iteration 2: $f(x_1) = 1 \quad f'(x_1) = 1$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 0$$

We are back to x_0!



The main issue is, again, we have points with f' $= 0$ in the neighborhood

Relaxation method

Relaxation method:

- Cast the equation $f(x) = 0$ in a form

$$x = \varphi(x)$$

- For example $\varphi(x) = f(x) + x$ but this choice is not unique

- The root is approximated by iterative procedure

$$x_{n+1} = \varphi(x_n)$$

Convergence criterion:

$$|\varphi'(x_n)| < 1, \qquad \text{for all } x_n$$

Relaxation method

```python
def relaxation_method(
    phi,                    # The function from the equation x = phi(x)
    x0,                     # The initial guess
    tolerance = 1.e-10,     # The desired accuracy of the solution
    max_iterations = 100    # Maximum number of iterations
    ):

    xprev = xnew = x0

    global last_relaxation_iterations
    last_relaxation_iterations = 0

    for i in range(max_iterations):
        last_relaxation_iterations += 1

        xprev = xnew
        xnew = phi(xprev) # The next iteration

        if (abs(xnew-xprev) < tolerance):
            return xnew


    print("The relaxation method failed to converge to a required precision in " + str(max_iterations) + " iterations")
    print("The error estimate is ", abs(xnew - xprev))

    return xnew
```

Relaxation method

$x + e^{-x} - 2 = 0$ as $x = 2 - e^{-x}$ i.e. $\phi(x) = 2 - e^{-x}$

Starting with x_0=0.5 we have

```
Solving the equation x = 2 - e^-x with relaxation method an initial guess of x0 =  0.5
Iteration:     0, x =    0.500000000000000, phi(x) = 1.393469340287367
Iteration:     1, x =    1.393469340287367, phi(x) = 1.751787325113973
Iteration:     2, x =    1.751787325113973, phi(x) = 1.826536369684999
Iteration:     3, x =    1.826536369684999, phi(x) = 1.839029855597129
Iteration:     4, x =    1.839029855597129, phi(x) = 1.841028423293983
Iteration:     5, x =    1.841028423293983, phi(x) = 1.841345821475382
Iteration:     6, x =    1.841345821475382, phi(x) = 1.841396170032424
Iteration:     7, x =    1.841396170032424, phi(x) = 1.841404155305379
Iteration:     8, x =    1.841404155305379, phi(x) = 1.841405421731432
Iteration:     9, x =    1.841405421731432, phi(x) = 1.841405622579610
Iteration:    10, x =    1.841405622579610, phi(x) = 1.841405654432999
Iteration:    11, x =    1.841405654432999, phi(x) = 1.841405659484766
Iteration:    12, x =    1.841405659484766, phi(x) = 1.841405660285948
Iteration:    13, x =    1.841405660285948, phi(x) = 1.841405660413011
Iteration:    14, x =    1.841405660413011, phi(x) = 1.841405660433162
Iteration:    15, x =    1.841405660433162, phi(x) = 1.841405660436358
The solution is x =  1.8414056604331623 obtained after  15  iterations
```

Not as fast as Newton-Raphson but does not require evaluation of derivative

Relaxation method

$x^3 - x - 1 = 0$ as $x = x^3 - 1$ i.e. $\varphi(x) = x^3 - 1$

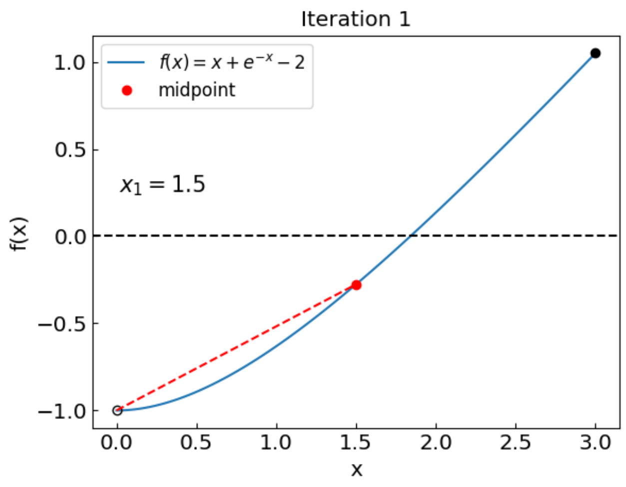
Starting with x_0=0.5 we have

```
Solving the equation x = x^3 - 1 with relaxation method an initial guess of x0 =  0.0
Iteration:     0, x =    0.000000000000000, phi(x) = -1.000000000000000
Iteration:     1, x =   -1.000000000000000, phi(x) = -2.000000000000000
Iteration:     2, x =   -2.000000000000000, phi(x) = -9.000000000000000
Iteration:     3, x =   -9.000000000000000, phi(x) = -730.000000000000000
Iteration:     4, x = -730.000000000000000, phi(x) = -389017001.000000000000000
Iteration:     5, x = -389017001.000000000000000, phi(x) = -58871587162270591457689600.000000000000000
Iteration:     6, x = -58871587162270591457689600.000000000000000, phi(x) = -20404090132275264698947825968051310952675782605
6202557355691431285390611316736.000000000000000
Iteration:     7, x = -204040901322752646989478259680513109526757826056202557355691431285390611316736.000000000000000, phi
(x) = -849477147223738769124261153859947219933304503407088864329587058315002861225858314510130211954336728493261609772281413
1127104275290993706669943943557518825041720139256751756296514363510463501782805696167407096791414943273033163341824.00000000
0000000
```

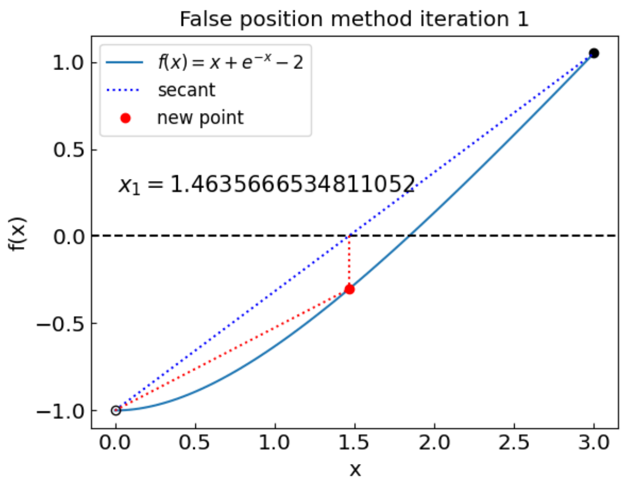
Divergent!

Reason: $|\varphi'(x_n)| < 1$ violated [come up with better form of $\varphi(x)$?]

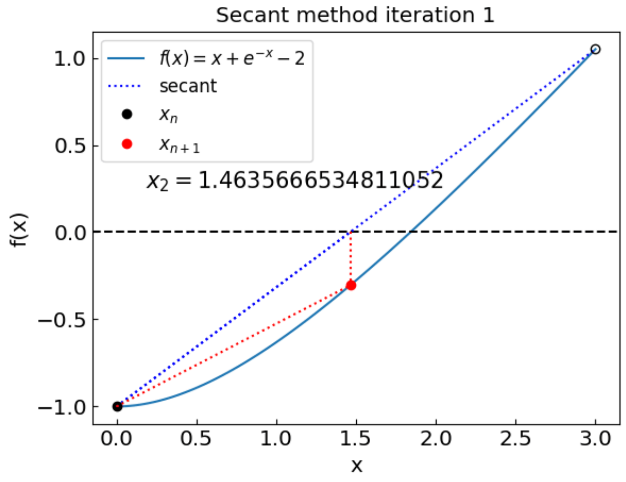
Summary



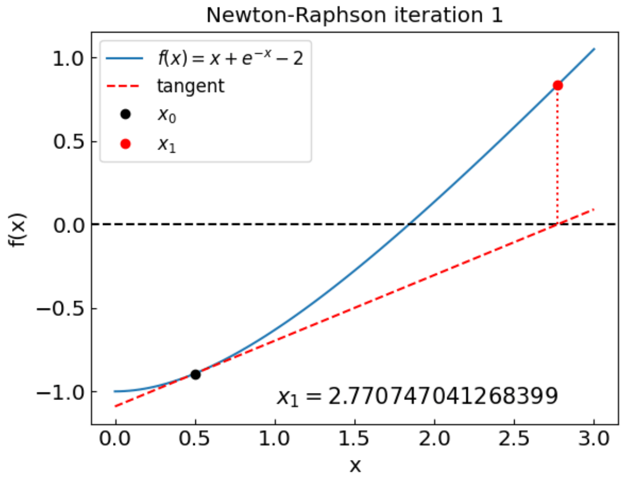
Iteration 1
$f(x) = x + e^{-x} - 2$
midpoint
$x_1 = 1.5$
f(x)
x



False position method iteration 1
$f(x) = x + e^{-x} - 2$
secant
new point
$x_1 = 1.4635666534811052$
f(x)
x



Secant method iteration 1
$f(x) = x + e^{-x} - 2$
secant
x_n
x_{n+1}
$x_2 = 1.4635666534811052$
f(x)
x



Newton-Raphson iteration 1
$f(x) = x + e^{-x} - 2$
tangent
x_0
x_1
$x_1 = 2.770747041268399$
f(x)
x

Summary

Bisection method:
- Guaranteed to converge with fixed rate
- Need to bracket the root

False position method:
- Guaranteed to converge
- Can be faster than bisection but not always
- Need to bracket the root

Secant method:
- Typically faster than bisection and false position
- May not always converge

Newton-Raphson method:
- Very fast when converges
- Can be sensitive to initial guess
- May not converge if $f'(x)=0$
- Requires evaluation of derivative at each step

Relaxation method:
- Simple to implement
- Does not require derivative
- Often does not converge