Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

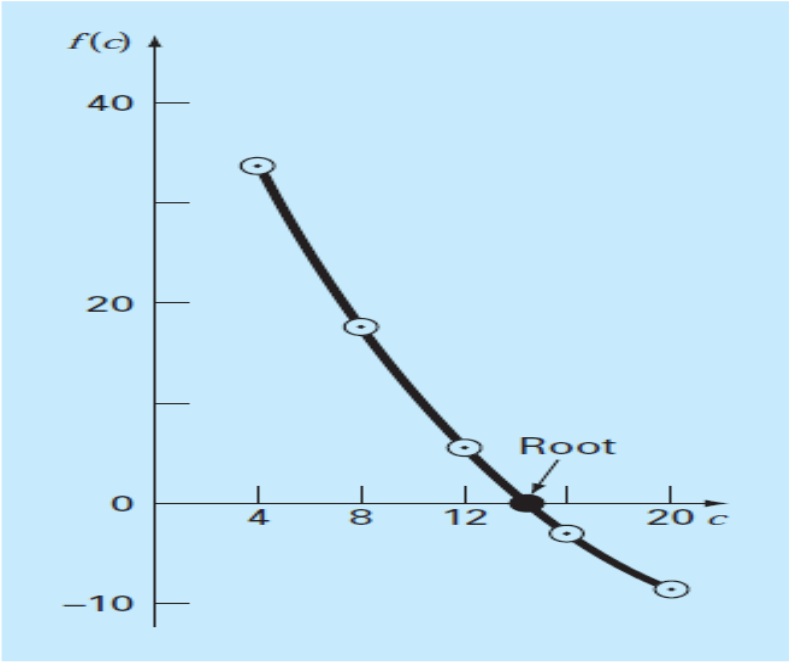
THE UNIVERSITY of EDINBURGH
School of Engineering

Roots of Equations: Bracketing Methods

1. There are many occasions when it is not possible or efficient to attempt the analytical solution of a known function.

2. In such cases, it is faster to determine the root of the equation f(x) (i.e. its value for x = 0), by using graphical methods (determination by inspection is almost instantaneous), or bracketing methods.

3. Graphical methods are fast, but can be imprecise without further refinement by testing estimated roots against function value.



Graphical determination of root

THE UNIVERSITY of EDINBURGH
School of Engineering

Roots of Equations: Bracketing Methods

3. Testing root estimates against function value is tedious and random without a sensible algorithm to save computational steps.
4. One of the oldest techniques is the bisection technique. A summary of this algorithm is provided as follows:

Step 1: Choose lower x_l and upper x_u guesses for the root such that the function changes sign over the interval. This can be checked by ensuring that $f(x_l)f(x_u) < 0$.

Step 2: An estimate of the root x_r is determined by

$$x_r = \frac{x_l + x_u}{2}$$

Step 3: Make the following evaluations to determine in which subinterval the root lies:

(a) If $f(x_l)f(x_r) < 0$, the root lies in the lower subinterval. Therefore, set $x_u = x_r$ and return to step 2.

(b) If $f(x_l)f(x_r) > 0$, the root lies in the upper subinterval. Therefore, set $x_l = x_r$ and return to step 2.

(c) If $f(x_l)f(x_r) = 0$, the root equals x_r; terminate the computation.

6. As presented this algorithm could proceed for a very large number of steps to determine an exact answer, but in practice one needs termination criteria to prevent it proceeding ad infinitum (i.e. forever or for a very long time).

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Roots of Equations: Bracketing Methods

Step 1: Choose lower x_l and upper x_u guesses for the root such that the function changes sign over the interval. This can be checked by ensuring that $f(x_l)f(x_u) < 0$.

Step 2: An estimate of the root x_r is determined by

$$x_r = \frac{x_l + x_u}{2}$$

Step 3: Make the following evaluations to determine in which subinterval the root lies:

(*a*) If $f(x_l)f(x_r) < 0$, the root lies in the lower subinterval. Therefore, set $x_u = x_r$ and return to step 2.

(*b*) If $f(x_l)f(x_r) > 0$, the root lies in the upper subinterval. Therefore, set $x_l = x_r$ and return to step 2.

(*c*) If $f(x_l)f(x_r) = 0$, the root equals x_r; terminate the computation.

1. The missing element in this code is the definition of an acceptable error which, once achieved, terminates the algorithm.

$$\varepsilon_a = \left| \frac{x_r^{\text{new}} - x_r^{\text{old}}}{x_r^{\text{new}}} \right| 100\%$$

2. To estimate the number of iterations based on the knowledge of the initial bracket size

$$n = \frac{\log(4/0.0625)}{\log 2} = 6$$

n = no of evaluations; 4 = initial interval;
0.0625 = final relative error

THE UNIVERSITY of EDINBURGH
School of Engineering

Roots of Equations: Bracketing Methods

3. The pseudo code used to implement the bisection method would be as follows:

```
FUNCTION Bisect(xl, xu, es, imax, xr, iter, ea)
  iter = 0
  DO
    xrold = xr
    xr = (xl + xu) / 2
    iter = iter + 1
    IF xr ≠ 0 THEN
       ea = ABS((xr − xrold) / xr) * 100
    END IF
    test = f(xl) * f(xr)
    IF test < 0 THEN
       xu = xr
    ELSE IF test > 0 THEN
       xl = xr
    ELSE
       ea = 0
    END IF
    IF ea < es OR iter ≥ imax EXIT
  END DO
  Bisect = xr
END Bisect
```

THE UNIVERSITY of EDINBURGH
School of Engineering

Roots of Equations: Bracketing Methods

4. To minimise function evaluations one adjust this to the code on the right: this reduces the number of **evaluations from 2n to n+1 evaluations making it far more efficient.**

```
FUNCTION Bisect(xl, xu, es, imax, xr, iter, ea)
  iter = 0
  DO
    xrold = xr
    xr = (xl + xu) / 2
    iter = iter + 1
    IF xr ≠ 0 THEN
       ea = ABS((xr − xrold) / xr) * 100
    END IF
    test = f(xl) * f(xr)
    IF test < 0 THEN
       xu = xr
    ELSE IF test > 0 THEN
       xl = xr
    ELSE
       ea = 0
    END IF
    IF ea < es OR iter ≥ imax EXIT
  END DO
  Bisect = xr
END Bisect
```

```
FUNCTION Bisect(xl, xu, es, imax, xr, iter, ea)
  iter = 0
  fl = f(xl)
  DO
    xrold = xr
    xr = (xl + xu) / 2
    fr = f(xr)
    iter = iter + 1
    IF xr ≠ 0 THEN
      ea = ABS((xr − xrold) / xr) * 100
    END IF
    test = fl * fr
    IF test < 0 THEN
      xu = xr
    ELSE IF test > 0 THEN
      xl = xr
      fl = fr
    ELSE
      ea = 0
    END IF
    IF ea < es OR iter ≥ imax EXIT
  END DO
  Bisect = xr
END Bisect
```

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Roots of Equations: Bracketing Methods

The Python code used to implement **the Bisection Method** would be as follows:

```
'''Approximate solution of f(x)=0 on interval [a,b] by bisection method.
   Parameters
   ----------
   f : function
  The function for which we are trying to approximate a solution f(x)=0.
   a,b : numbers
      The interval in which to search for a solution. The function returns
      None if f(a)*f(b) >= 0 since a solution is not guaranteed.
   N : (positive) integer
      The number of iterations to implement.
   Returns
   -------
   x_N : number
      The midpoint of the Nth interval computed by the bisection
method. The initial interval [a_0,b_0] is given by [a,b]. If f(m_n) == 0 for
some midpoint m_n = (a_n + b_n)/2, then the function returns this
solution.
 If all signs of values f(a_n), f(b_n) and f(m_n) are the same at any
 iteration, the bisection method fails and return None.
   Examples
   --------
   f = lambda x: x**2 - x - 1
   bisection(f,1,2,25)
   1.618033990263939
   f = lambda x: (2*x - 1)*(x - 3)
   >>> bisection(f,0,1,10) gives 0.5
   '''
def bisection(f,a,b,N):
if f(a)*f(b) >= 0:
      print("Bisection method fails.")
      return None
   a_n = a
   b_n = b
   for n in range(1,N+1):
      m_n = (a_n + b_n)/2
      f_m_n = f(m_n)
      if f(a_n)*f_m_n < 0:
         a_n = a_n
         b_n = m_n
      elif f(b_n)*f_m_n < 0:
         a_n = m_n
         b_n = b_n
      elif f_m_n == 0:
         print("Found exact solution.")
         return m_n
      else:
         print("Bisection method fails.")
         return None
   return (a_n + b_n)/2

f = lambda x: x**3 - x - 1
approx_phi = bisection(f,1,2,25)
print(approx_phi)
```

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

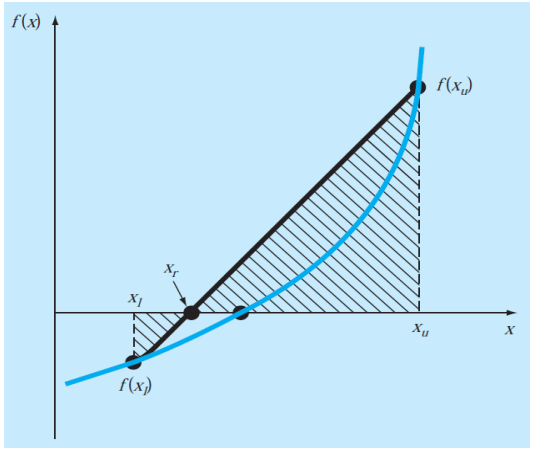
Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Bracketing Methods: False Position Method

1. The bisection method works relatively well but is inefficient by comparison with more advanced methods.
2. It is an example of a 'brute force' technique that relies on a sufficient number of iterations to achieve either the true answer, or one to within an acceptable error margin.
3. A more refined technique is the ***false position method***, which takes account of the magnitude of the function values on either bound of the interval x = l to x = u, i.e., the evaluations $f(x_l)$ and $f(x_u)$.
4. The false position algorithm is based on the geometry shown in the figure below.
5. Here, two similar triangles allow the following relation to be written:

$$\frac{f(x_l)}{x_r - x_l} = \frac{f(x_u)}{x_r - x_u}$$



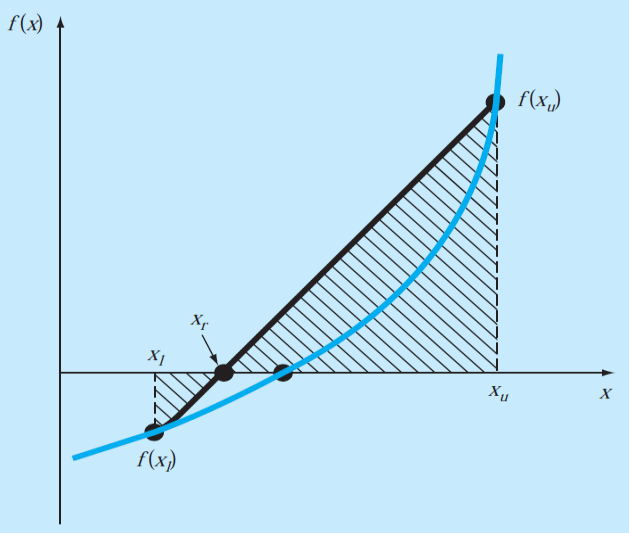
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Bracketing Methods: False Position Method



$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

6. This relation allows an intermediate root estimate, x_r, the common point of the two triangles to be calculated as follows:

7. The term false position refers to the fact that the first estimate is obviously a false estimate of the root.

8. The formula iterates until a satisfactory estimate is achieved.

Computational Methods and Modelling

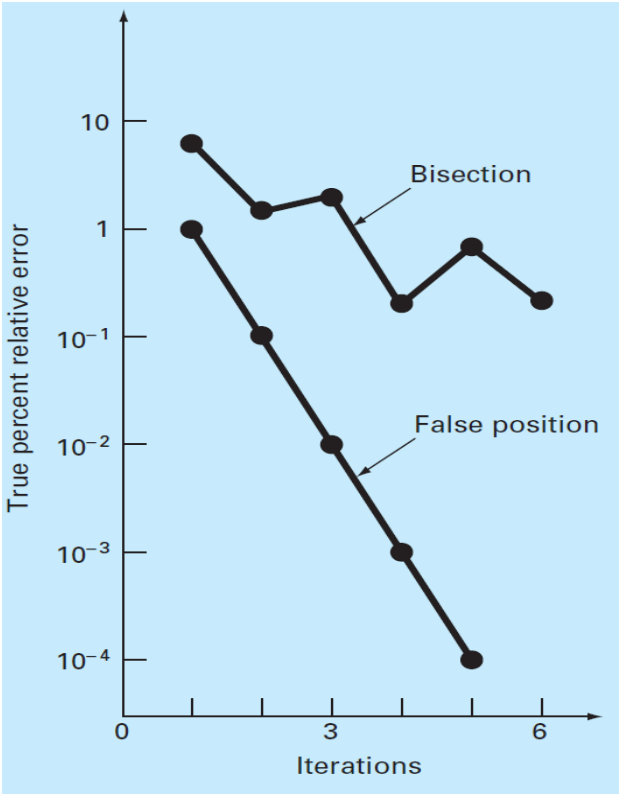
Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Bracketing Methods: False Position Method

1. A comparison of the rate of error elimination of the bisection and false position methods is given by Chapra et al. for Examples 5.4 and 5.5



2. Clearly, **in this case,** the false position method achieves rapid relatively accuracy (in this case the error is two orders of magnitude lower than that achieved with the bisection technique.

THE UNIVERSITY of EDINBURGH
School of Engineering

Bracketing Methods: False Position Method

3. However, FP does not always achieve such rapid convergence. It depends strongly on the type of function being solved, since the technique is dependent on the shape of the function curve.
4. On occasion, the bisection technique can achieve greater accuracy in fewer iterations.
5. Example $f(x) = x^{10}-1$. Evaluate the root of f(x) between x = 0 and x = 3 using bisection and false position techniques and compare convergence.

Bisection Technique: Results

Iteration	x_l	x_u	x_r	ε_a (%)	ε_t (%)
1	0	1.3	0.65	100.0	35
2	0.65	1.3	0.975	33.3	2.5
3	0.975	1.3	1.1375	14.3	13.8
4	0.975	1.1375	1.05625	7.7	5.6
5	0.975	1.05625	1.015625	4.0	1.6

False Position Technique: Results

Iteration	x_l	x_u	x_r	ε_a (%)	ε_t (%)
1	0	1.3	0.09430		90.6
2	0.09430	1.3	0.18176	48.1	81.8
3	0.18176	1.3	0.26287	30.9	73.7
4	0.26287	1.3	0.33811	22.3	66.2
5	0.33811	1.3	0.40788	17.1	59.2

THE UNIVERSITY of EDINBURGH
School of Engineering

Bracketing Methods: False Position Method

Bisection Technique: Results

Iteration	x_l	x_u	x_r	ε_a (%)	ε_t (%)
1	0	1.3	0.65	100.0	35
2	0.65	1.3	0.975	33.3	2.5
3	0.975	1.3	1.1375	14.3	13.8
4	0.975	1.1375	1.05625	7.7	5.6
5	0.975	1.05625	1.015625	4.0	1.6

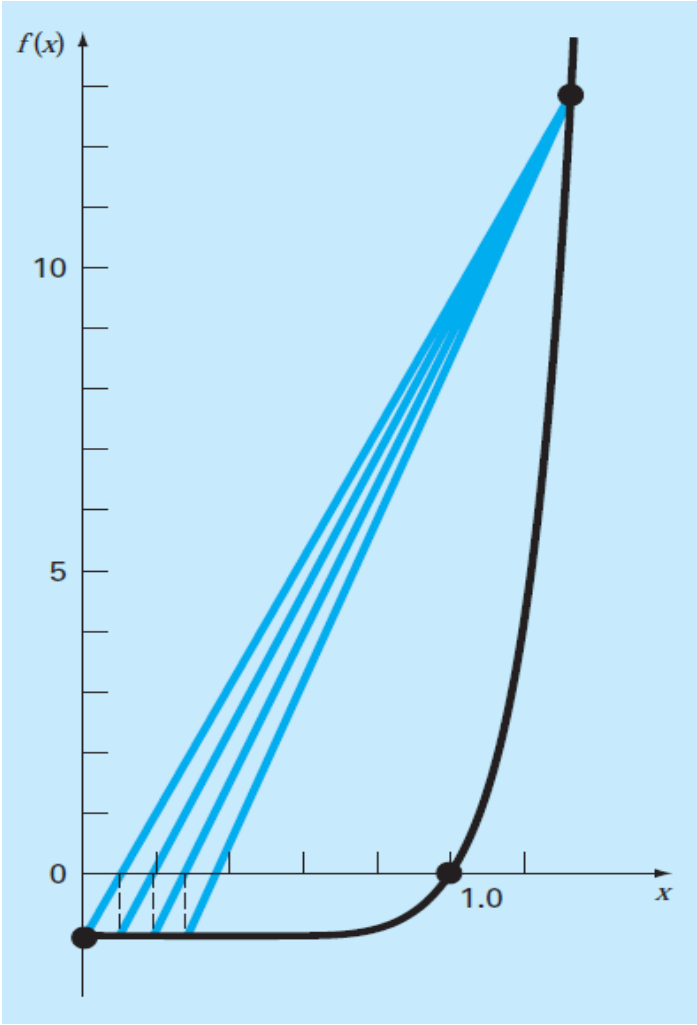
False Position Technique: Results

Iteration	x_l	x_u	x_r	ε_a (%)	ε_t (%)
1	0	1.3	0.09430		90.6
2	0.09430	1.3	0.18176	48.1	81.8
3	0.18176	1.3	0.26287	30.9	73.7
4	0.26287	1.3	0.33811	22.3	66.2
5	0.33811	1.3	0.40788	17.1	59.2

3. Comparing the two tables it is clear that, although the false position technique does achieve progressively lower errors with each iteration, it is clearly much slower to do this than the bisection method.
4. The reason for this is the shape of this particular function and the slow progress of each new estimated root along the x-axis. (next slide).

THE UNIVERSITY of EDINBURGH
School of Engineering

Bracketing Methods: False Position Method



5. This graphic explains why false position has been so much slower to converge than bisection.

6. The graph has a prolonged flat section which means that the algorithm crawls along the curve because it cannot 'see' the shape of the function in advance.

It is essential to graph a function prior to choosing a suitable estimation technique.

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Modified False Position Method

1. To address the deficiency of the FPM, a modified FPM may be used.
2. The unmodified FPM suffers from the problem that **one of the bounds can remain unchanged through many iterations** and so convergence is extremely slow to occur.
3. **Modified FPM overcomes this by halving the function value at the 'stuck' bound**, if the algorithm detects that the 'stuck' bound has not moved after a certain number of steps.

$$X_r = X_u - \frac{f(X_u)(X_l - X_u)}{f(X_l) - f(X_u)} \quad \Rightarrow \quad X_r = X_u - \frac{0.5\, f(X_u)(X_l - X_u)}{f(X_l) - 0.5 f(X_u)}$$

4. A Python code for this algorithm is given on the next slide.
5. The efficiency of this technique for the equation $f(x) = x^{10}-1$ to an error tolerance of 0.01% is shown in the table.

Bisection	Unmodified False Position	Modified False Position
14	39	12 steps

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Basic False Position Method: Python Code

```
MAX_ITER = 1000000
# The function is x^3 - x^2 + 2
def func( x ):
    return (x**3 - 4*(x*2) + 10)
# Prints root of func(x) in interval [a, b]
def regulaFalsi( a , b):
    if func(a) * func(b) >= 0:
        print("You have not assumed right a and b")
        return -1
    c = a
# Initialize result
    for i in range(MAX_ITER):
        # Find the point that touches x axis
        c = (a * func(b) - b * func(a))/ (func(b) -
func(a))
        # Check if the above found point is root
        if func(c) == 0:
            break
        # Decide the side to repeat the steps
        elif func(c) * func(a) < 0:
            b = c
        else:
            a = c
    print("The value of root is : " , '%.4f' %c)
```

```
# Test the function by
setting a and b and calling
the code by the filename
# Initial values assumed
a =-200
b = 300
regulaFalsi(a, b)
```

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Modified False Position Method: Python Code

```
MAX_ITER = 1000000
# The function is x^3 - x^2 + 2
def func( x ):
    return (x**3 - 4*(x*2) + 10)
# Prints root of func(x) in interval [a, b]
def regulaFalsi( a , b):
    if func(a) * func(b) >= 0:
        print("You have not assumed right a and b")
        return -1
    c = a
# Initialize result
    for i in range(MAX_ITER):
        # Find the point that touches x axis
        c1 = (a * 0.5*func(b) - b * func(a))/
(0.5*func(b) - func(a))
        c2 = (a * func(b) - b * 0.5* func(a))/
(func(b) - 0.5* func(a))
          If func(c1) < func(c2):
                    c = c1
          else:
                    c = c2
        # Check if the above found point is root
        if func(c) == 0:
            break
 # Decide the side to repeat
the steps
elif func(c) * func(a) < 0:
            b = c
        else:
            a = c
print("The value of root is : "
, '%.4f' %c)
```

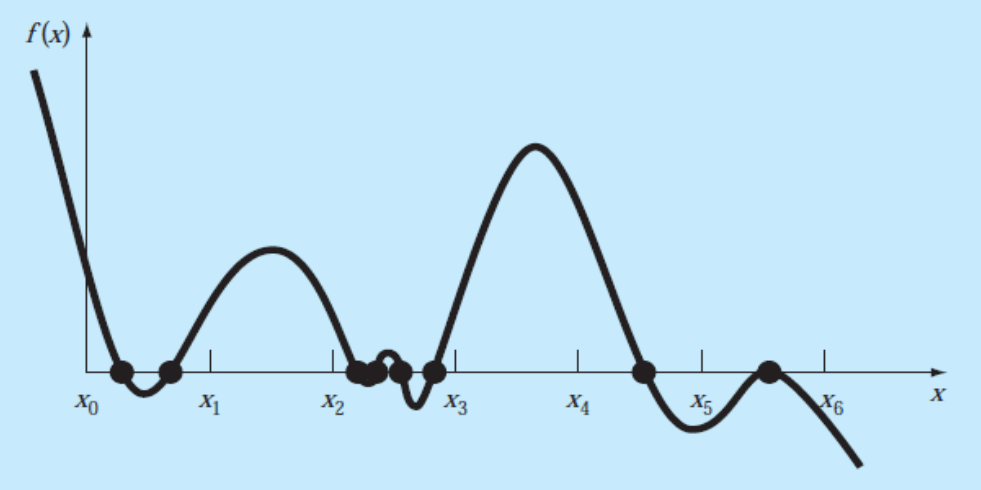
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

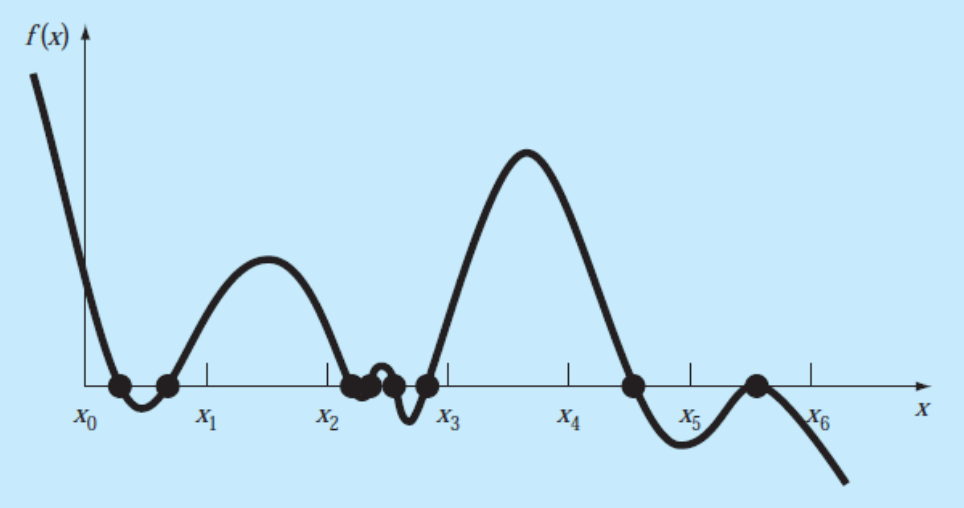
Incremental Search Techniques



1. Another category of root-finding techniques are the incremental search procedures.
2. These techniques consist in choosing one point at either end of an interval of interest around a root of the function (as seen on a graph).
3. Then the function is evaluated at arbitrary increments moving in one direction towards the real root until the value of the function is within the required error tolerance.
4. This approach will be effective provided that the size of the initial and successive increments are appropriately chosen.
5. It is also heavily dependent on the shape of the function and expensive to implement

THE UNIVERSITY of EDINBURGH
School of Engineering

Incremental Search Techniques



Cases where roots can be missed by large steps

6. If roots are too close together, a too-large step can completely miss the second root, or even completely miss whole groups of roots.
7. Therefore care is needed in selecting step size.
8. One can evaluate the derivative of the curve at each step to determine whether there has been a local or global maximum or minimum point in the function between points.
9. This can indicate the existence of a root, especially closely clustered ones (above).

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Incremental Search Techniques

Python Code for Implementing Incremental Search

```
def naive_root(f, x_guess, tolerance, step_size):

    steps_taken = 0

    while abs(f(x_guess)) > tolerance:

        if f(x_guess) > 0:
            x_guess -= step_size
        elif f(x_guess) < 0:
            x_guess += step_size
        else:
            return x_guess

        steps_taken += 1

    return x_guess, steps_taken

f = lambda x: x**2 - 20
root, steps = naive_root(f, x_guess=4.5, tolerance=.01, step_size=.001)
print ("root is:", root)
print ("steps taken:", steps)
```

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

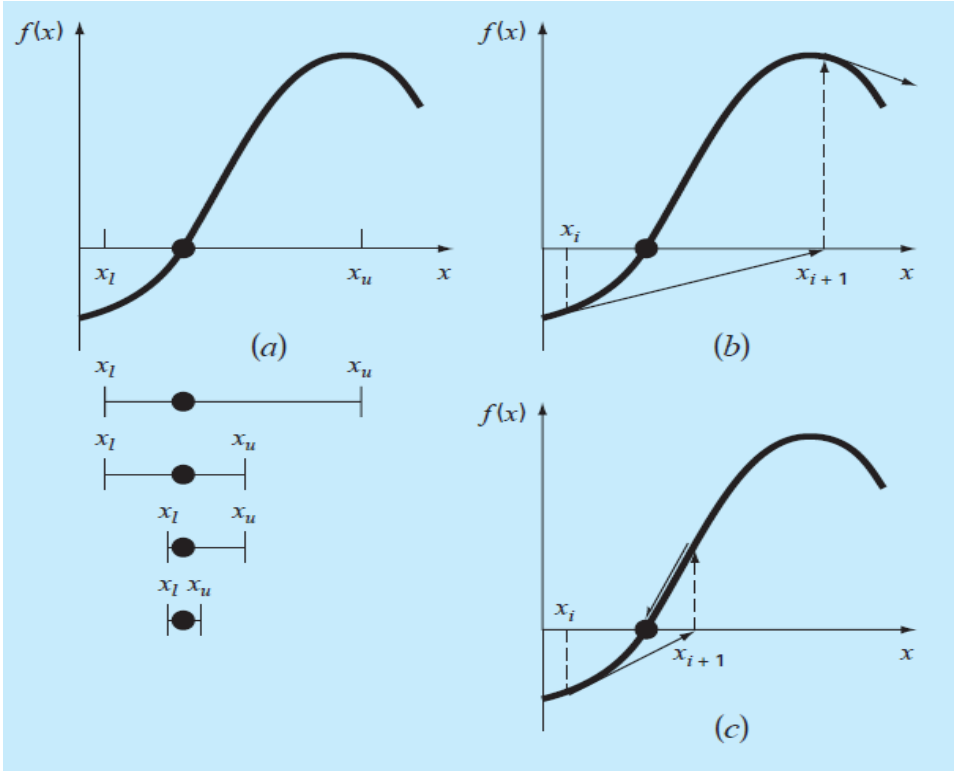
Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Open Root Finding Methods

1. **Open root finding techniques require only one initial starting point.**
2. Thus **there is no closed interval bounded by two points.**
3. They do not require preliminary selection of a region of interest, but can notionally solve for the root by approaching from any guess x-value.

Closed



Open

Computational Methods and Modelling

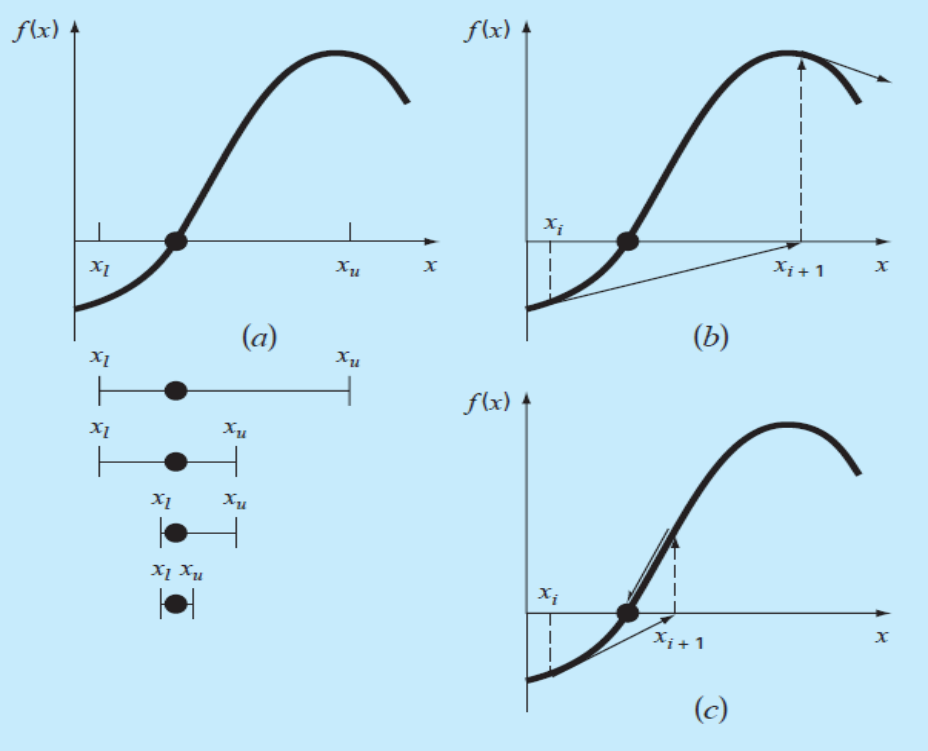
Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Open Root Finding Methods

Differences between Closed and Open Methods



1. Closed methods will always converge to a solution, provided that the root or roots actually lie in the domain of x chosen for investigation.

2. Open methods, by contrast, may not converge towards a root (case b in Figure), but if they do (case c) they tend to be far more efficient in reaching a solution in fewer steps than bisection.

THE UNIVERSITY of EDINBURGH
School of Engineering

Open Root Finding Methods

1. **Single fixed point iteration** is implemented by taking the known equation for the function and expressing it in terms of x.

2. Take the equation $x^2 - 2x + 3 = 0$

3. This can be re-expressed in terms of x on the left hand side as follows:

$$x = \frac{x^2 + 3}{2}$$

4. This allows one to predict a new value of x using a previous value. So if I posit an initial guess value of x, x_i, I can calculate a better estimate x_{i+1} as the left hand side of the equation above, i.e.

$$x_{i+1} = g(x) = \frac{x_i^{\,2} + 3}{2}$$

5. This algorithm is implemented through successive iterations until the difference between successive estimates is less than a pre-defined absolute error, i.e.,

$$\frac{x_{i+1} - x_i}{x_i} < \varepsilon$$

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Open Root Finding Methods

1. Take the example of the following function: $x_{i+1} = e^{-x_i}$
2. Implementing a **single fixed point iteration** on this function gives the following table through 10 iterations

i	x_i	ε_a (%)	ε_t (%)
0	0		100.0
1	1.000000	100.0	76.3
2	0.367879	171.8	35.1
3	0.692201	46.9	22.1
4	0.500473	38.3	11.8
5	0.606244	17.4	6.89
6	0.545396	11.2	3.83
7	0.579612	5.90	2.20
8	0.560115	3.48	1.24
9	0.571143	1.93	0.705
10	0.564879	1.11	0.399

3. The analytical solution to the function is x = 0.56714879. The relative error is 0.4%.
4. **This technique displays approximate linear convergence,** as the relative error on each successive iteration is approx. half that of the previous step.
5. **A variation on this technique is the two-curve graphical method** for complex functions.

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Open Root Finding Methods

1. A Python Code for **Single Fixed Point Iteration** is given as follows:

```
def f(x):
    return x*x*x + x*x -1

# Re-writing f(x)=0 to x = g(x)
def g(x):
    return 1/math.sqrt(1+x)

# Implementing Fixed Point Iteration Method
def fixedPointIteration(x0, e, N):
    print('\n\n*** FIXED POINT ITERATION ***')
    step = 1
    flag = 1
    condition = True
    while condition:
        x1 = g(x0)
        print('Iteration-%d, x1 = %0.6f and f(x1) = %0.6f' % (step, x1,
f(x1)))
        x0 = x1
        step = step + 1

    if step > N:
            flag=0
            break
        condition = abs(f(x1)) > e

    if flag==1:
        print('\nRequired root is: %0.8f' % x1)
    else:
        print('\nNot Convergent.')

# Input Section
x0 = input('Enter Guess: ')
e = input('Tolerable Error: ')
N = input('Maximum Step: ')

# Converting x0 and e to float
x0 = float(x0)
e = float(e)

# Converting N to integer
N = int(N)

#Note: You can combine above three section like this
# x0 = float(input('Enter Guess: '))
# e = float(input('Tolerable Error: '))
# N = int(input('Maximum Step: '))
```

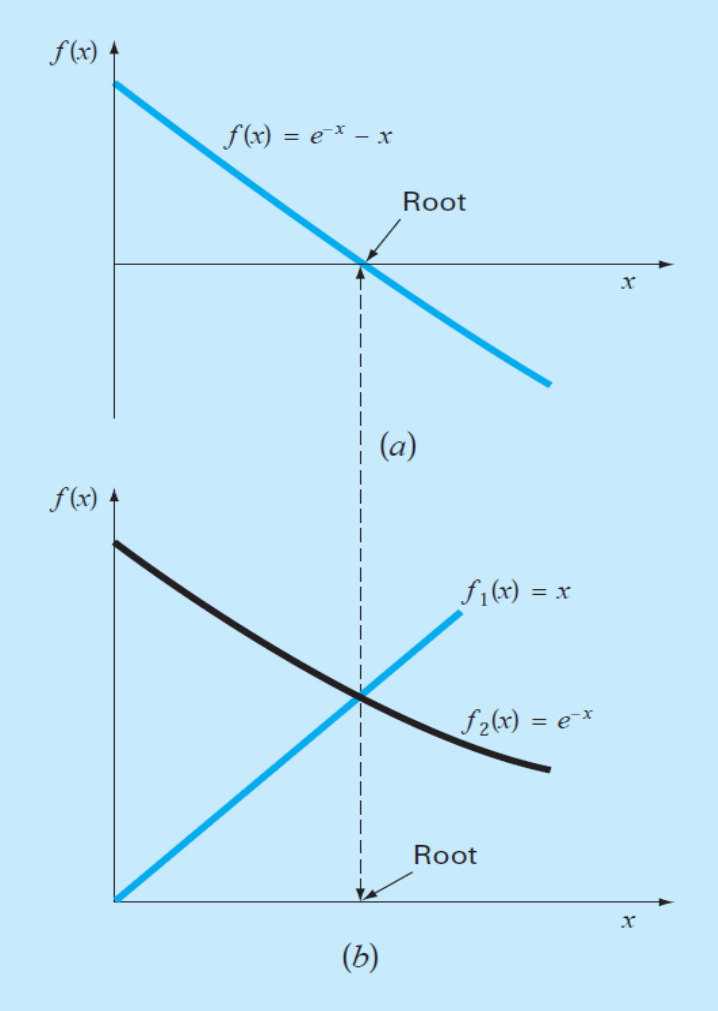
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Two point graphical method



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

1. This function is broken into the functions y = exp(-x) and y = x.
2. The root of the master function is simply the value of x for which these functions intersect.
3. However, the technique cannot be used uncritically in all situations…

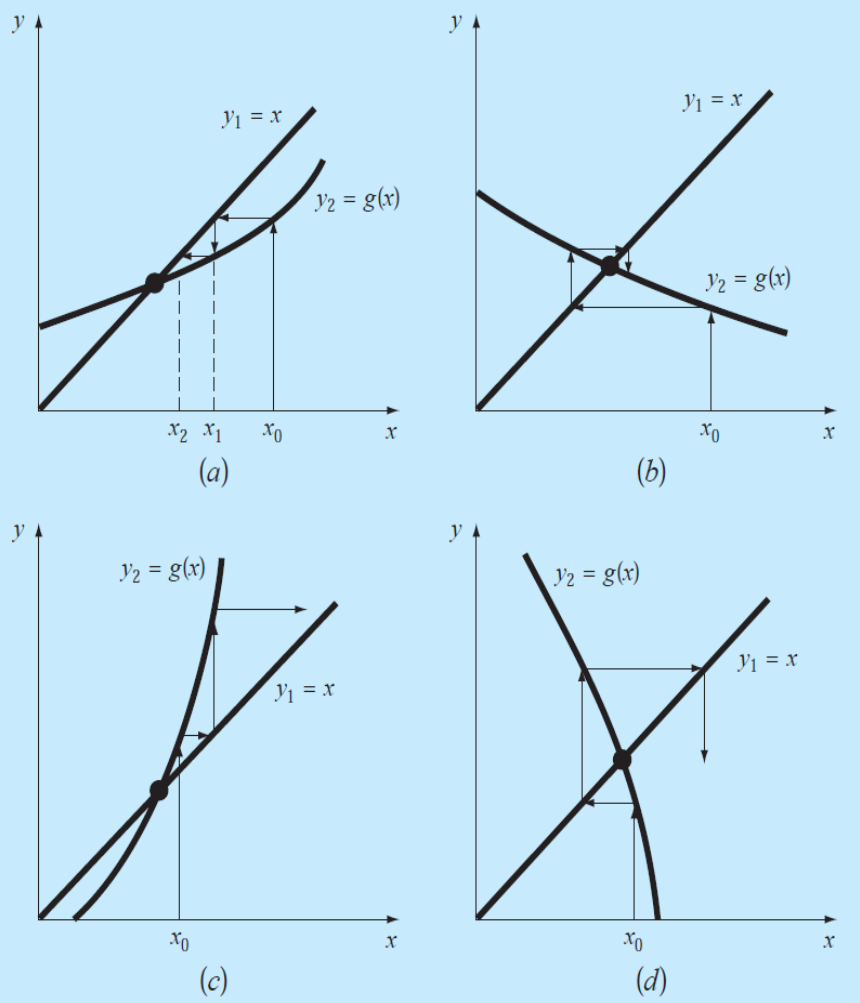
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Two point graphical method

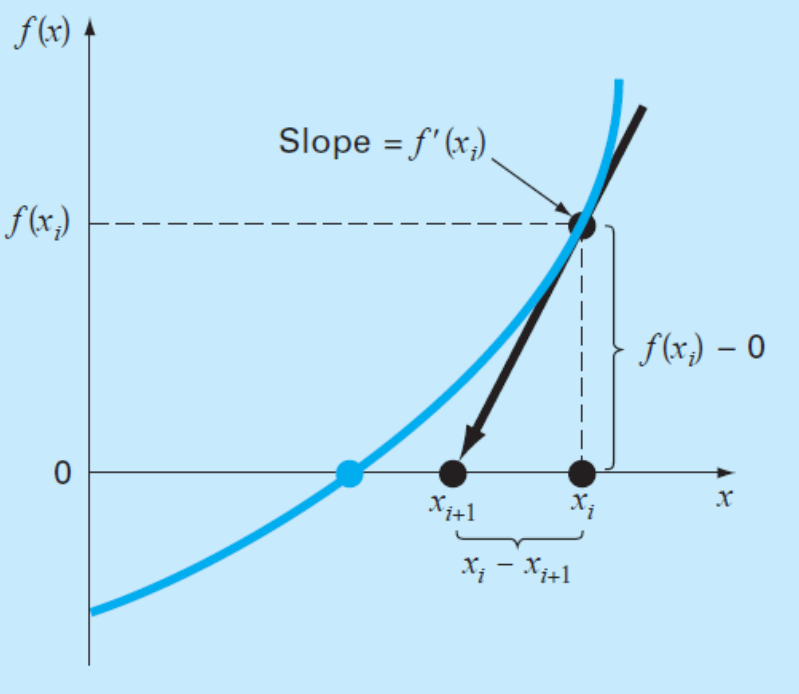


1. Four generic examples of function are given in the graph (left).
2. In each graph the progress of an iteration towards the intersection of each pair is shown.
3. a) and b) show converging solutions, while c) and d) show diverging non-solutions where the simple fixed iteration method fails.
4. Secondly, a) and c) represent monotone iteration patterns, while b) and d) represent spiral iteration patterns.
5. A solution is possible for the condition

$$|g'(x)| < 1$$

THE UNIVERSITY of EDINBURGH
School of Engineering

Newton Raphson Method



$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

1. The Newton Raphson Method is another open root-finding technique.
2. Here, an initial root value, x_i, is used to evaluate a function value $f(x_i)$ from which a tangent line is drawn to the x axis to produce a next estimate for the root, x_{i+1}.
3. This process continues until the function f(x) approximates 0 to within an error.

THE UNIVERSITY of EDINBURGH
School of Engineering

Newton Raphson Technique

1. If we take the same function we solved earlier using the open fixed point method, Newton Raphson results in the following tabulation of results and solution.

i	x_i	ε_t (%)
0	0	100
1	0.500000000	11.8
2	0.566311003	0.147
3	0.567143165	0.0000220
4	0.567143290	$< 10^{-8}$

4. Note the much faster convergence and the far lower error than that which was achieved using single point fixed method.
5. **Exercise: Apply this algorithm to our equation that we solved earlier using the single point fixed method, i.e.,**

$$x^2 + 5x + 8 = 0$$

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Newton Raphson Technique

1. A Python Implementation of Newton Raphson is as follows:

```
def newton(f,Df,x0,epsilon,max_iter):
    '''Solution of f(x)=0 by Newton's method.

    Parameters
    ----------
    f : function for which we are searching for a solution
f(x)=0.
    Df : Derivative of f(x).
    x0 : Initial guess for a solution f(x)=0.
    epsilon : number
        Stopping criteria is abs(f(x)) < epsilon.
    max_iter : integer
        Maximum number of iterations
  Examples
    --------
    >>> f = lambda x: x**2 - x - 1
    >>> Df = lambda x: 2*x - 1
    >>> newton(f,Df,1,1e-8,10)
    Found solution after 5 iterations.
    1.618033988749989
    xn = x0
    for n in range(0,max_iter):
        fxn = f(xn)
        if abs(fxn) < epsilon:
            print('Found solution after',n,'iterations.')
            return xn
        Dfxn = Df(xn)
        if Dfxn == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn/Dfxn
    print('Exceeded maximum iterations. No solution found.')
    return None

f = lambda x: x**4 - x - 1
df= lambda x: 4*x**3 - 1
x0=1
epsilon=0.001
max_iter=100
solution = newton(f,df,x0,epsilon,max_iter)
print(solution)
```

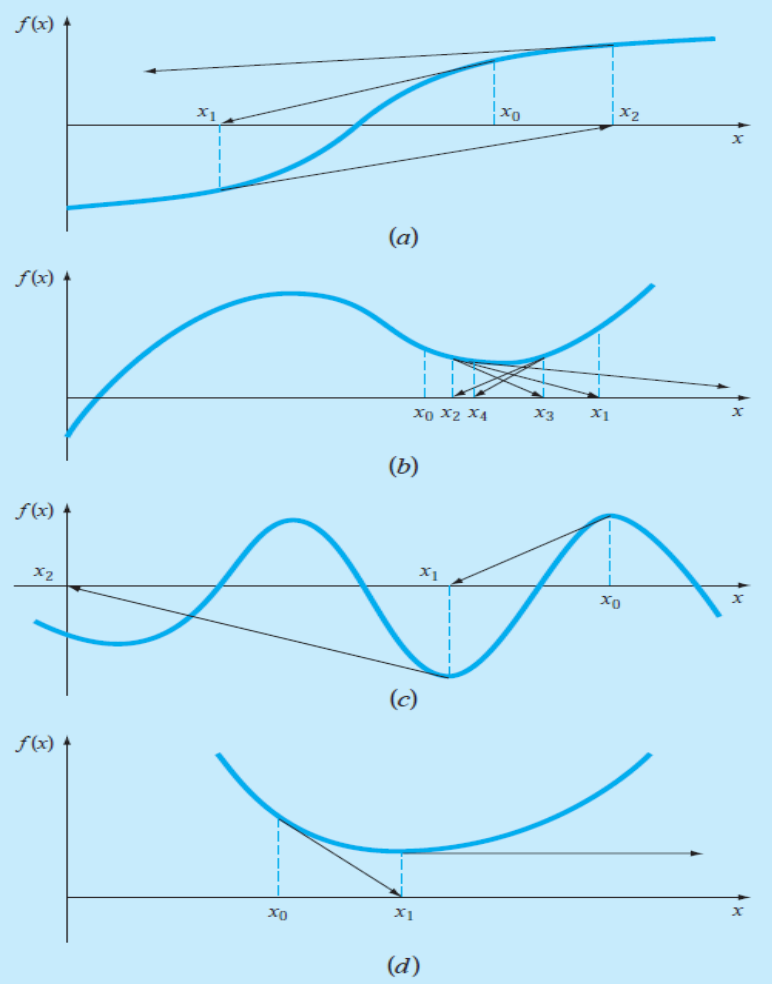
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Newton Raphson Technique



Functions not solvable

1. Asymptotic functions
2. Functions with multiple minima: no solution will be found.
3. Cyclic/periodic functions
4. Functions with no real root in the domain.

Measures to mitigate

- Graph the function to see the function shape and establish whether a root exists in the domain (range of x) of interest.
- Check each solution to establish its closeness to zero.
- Include an upper limit in the number of iterations to prevent infinite cycling.

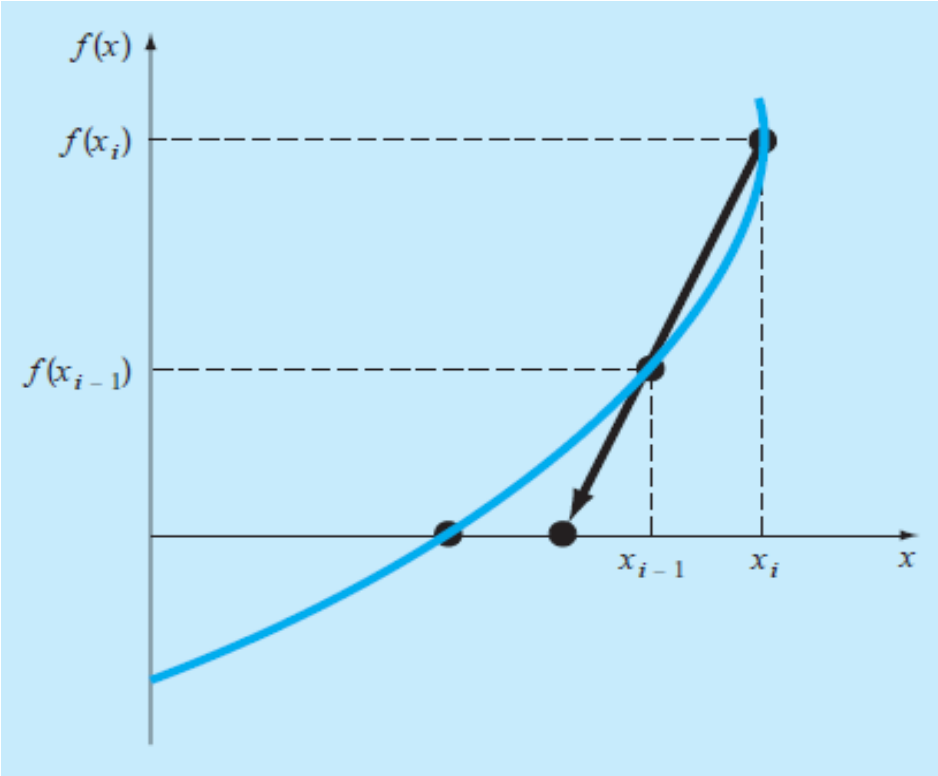
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Newton Raphson / Secant Techniques



1. A complication for the Newton-Raphson method is that **it requires the calculation of an exact derivative.**
2. This is fine for known functions which are analytically differentiable, but more problematic for those which aren't.
3. For the latter functions, the Secant Method may be used: this calculates an approximation to the point derivative, but is otherwise the same as Newton Raphson.

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Secant Technique

A Python Code for **the Secant Technique** is given as below (with suggested test).

```
def secant(f,a,b,N):

'Examples -------- >>> f = lambda x: x**2 -
x - 1 >>> secant(f,1,2,5) 1.6180257510729614
if f(a)*f(b) >= 0:
          print("Secant method fails.")
return None
          a_n = a b_n = b
for n in range(1,N+1):
          m_n = a_n - f(a_n)*(b_n -
          a_n)/(f(b_n) - f(a_n)) f_m_n =
          f(m_n)
if f(a_n)*f_m_n < 0:
          a_n = a_n b_n = m_n
elif f(b_n)*f_m_n < 0:
          a_n = m_n b_n = b_n
elif f_m_n == 0:
          print("Found exact solution.")
return m_n
else: print("Secant method fails.")
          return None
return a_n - f(a_n)*(b_n - a_n)/(f(b_n) -
f(a_n))
```

Implement code by defining function, f, and calling secant. The result is allocated to variable 'solution'.

```
f = lambda x: x**4 - x - 1
solution =
secant(f,1,2,25)
print(solution)
```

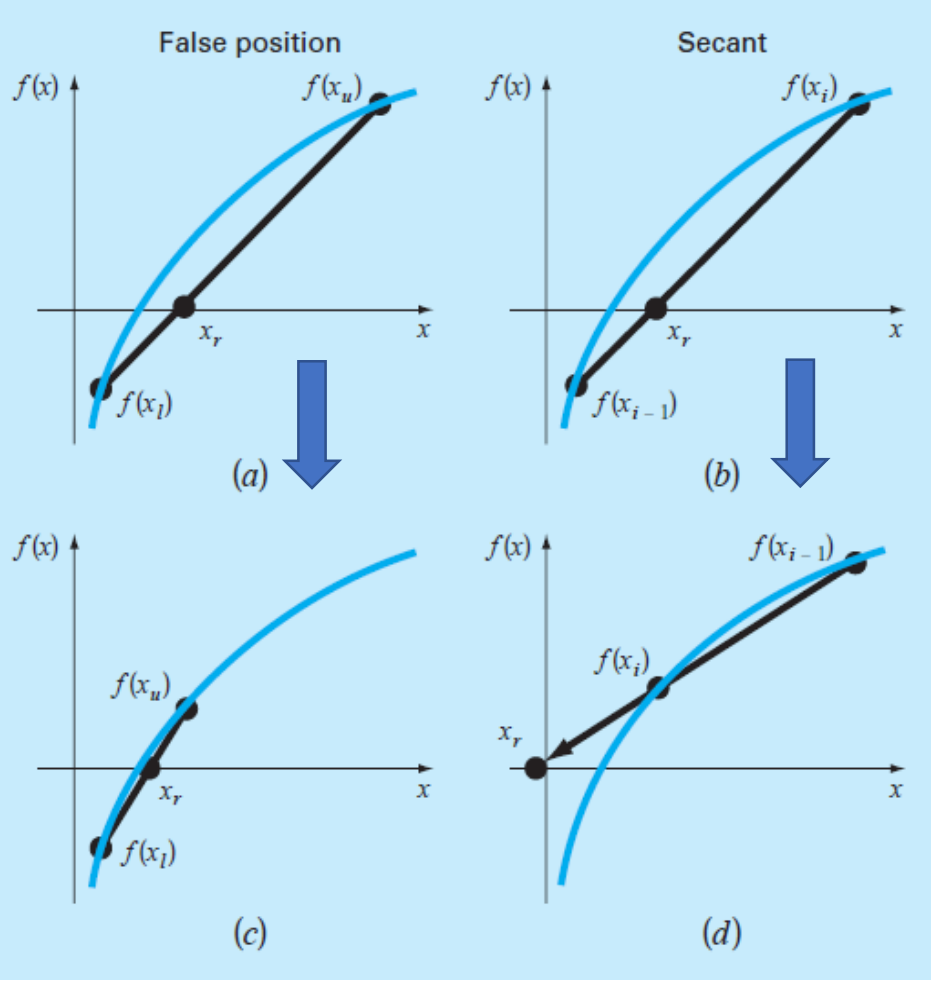
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Secant Technique v. False Position



1. Secant and False Position are similar.
2. However, False Position always sets the new right bound of the new line to be at the new estimate for the root.
3. Secant uses the new estimate to cut the curve directly above it.
4. In the example to the left, this means that for this function, False Position is more effective.
5. **FP is guaranteed to find the root, Secant not.**

THE UNIVERSITY of EDINBURGH
School of Engineering

Modified Secant Technique

1. A modification of the Secant Technique involves the alteration of how derivatives are evaluated, specifically at which points.
2. In the Secant Technique, two separate points on the function that are significantly apart from each other are used to compute the derivative for the given step
3. However, in the modified Secant Technique, the new derivative is calculated by retaining one old point and taking a small increment on this value (or perturbation) as the second reference point.

$$f'(x_i) \cong \frac{f(x_i + \delta x_i) - f(x_i)}{\delta x_i}$$

$$x_{i+1} = x_i - \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i)}$$

4. **Exercise: Modify the Secant Code on Slide 31 to include the equation above.**
5. **Exercise: Perform the Modified Secant technique on the function f(x) = exp(-x)-x, and compare the outcome with that of the Secant Method.**

Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Modified Secant Technique

A Python Code for **the Modified Secant Technique** is given as below (with suggested test).

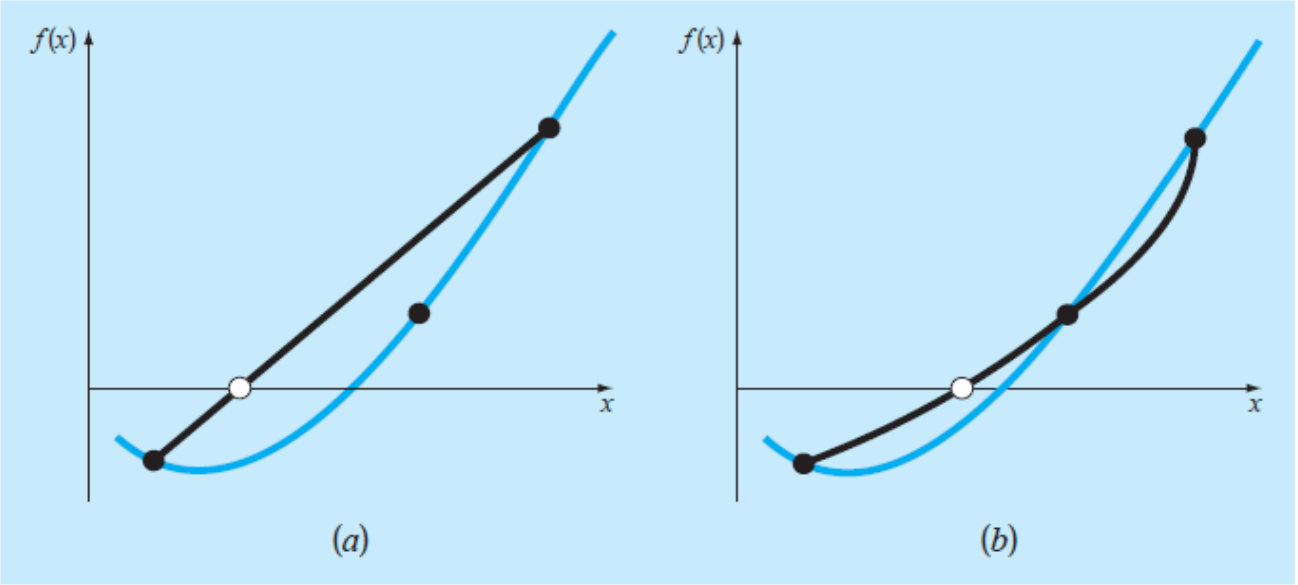
```
def secant_method(func, x0, alpha=1.0, tol=1E-9,
maxit=200):
    """
    Uses the secant method to find f(x)=0.

    INPUTS

        * f     : function f(x)
        * x0    : initial guess for x
        * alpha : relaxation coefficient:
modifies Secant step size
        * tol   : convergence tolerance
        * maxit : maximum number of iteration, default=200
    """

x, xprev = x0, 1.001*x0
    f, fprev = x**4 - x - 1, xprev**4 - xprev - 1

    rel_step = 2.0 *tol
    k = 0
    print('{0:12} {1:12} {2:12} {3:12} {4:12}'\
    .format('Iteration', 'x', 'f(x)', 'Rel step', 'alpha *
Delta x'))
```

```
while (abs(f) > tol) and (rel_step) > tol
and (k<maxit):
        rel_step = abs(x-xprev)/abs(x)

        # Full secant step
        dx = -f/(f - fprev)*(x - xprev)

        # Update `xprev` and `x`
simultaneously
        xprev, x = x, x + alpha*dx

        # Update `fprev` and `f`:
        fprev, f = f, func(x)

        k += 1

        print('{0:10d} {1:12.5f} {2:12.5f}
{3:12.5f} {4:12.5f}'\
        .format(k, xprev, fprev, rel_step,
alpha*dx))

    return x

    func = lambda x: x**4 - x - 1
    solution =
secant_method(func,0,alpha=1.0,tol=1E-
9,maxit=200)
    print(solution)
```

THE UNIVERSITY of EDINBURGH
School of Engineering

Inverse Quadratic Interpolation

1. Using a second order curve to model a curve that is third order or higher is more effective than using a linear secant as the second order curve (a parabola) follows the target function more closely.
2. Inverse Quadratic Interpolation uses a parabola x = f(y), i.e., a parabola lying on its side, to model a function instead of the secant line we used earlier.



a) Secant method b) Inverse Quadratic Interpolation

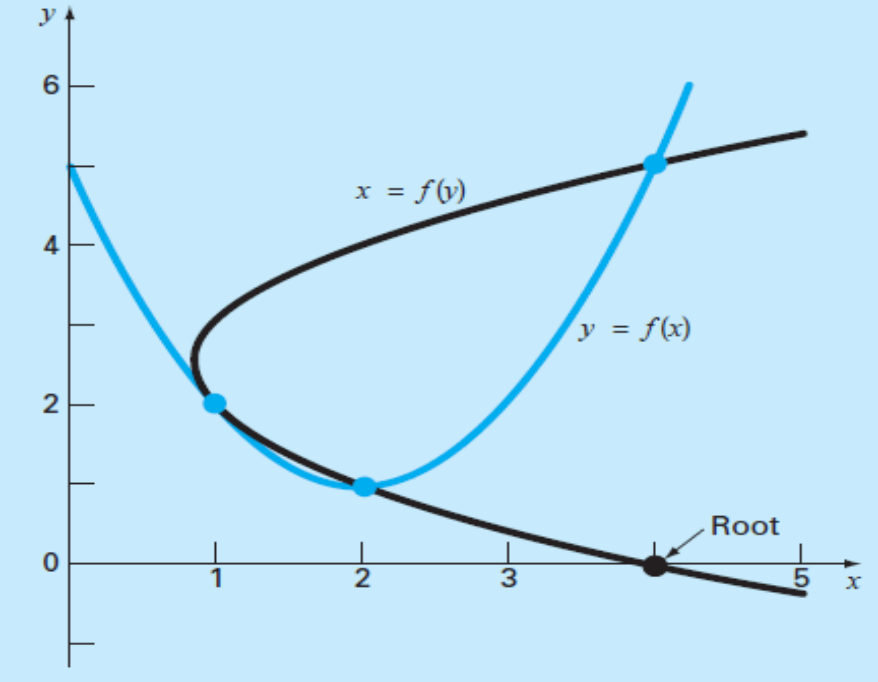
Computational Methods and Modelling

Lecture 2: Dr. Edward McCarthy

Topic 1: Finding Roots of Equations Numerically

THE UNIVERSITY of EDINBURGH
School of Engineering

Interpolation Functions without Real Roots



$$x_{i+1} = \frac{y_{i-1}y_i}{(y_{i-2}-y_{i-1})(y_{i-2}-y_i)}x_{i-2} + \frac{y_{i-2}y_i}{(y_{i-1}-y_{i-2})(y_{i-1}-y_i)}x_{i-1} + \frac{y_{i-2}y_{i-1}}{(y_i-y_{i-2})(y_i-y_{i-1})}x_i$$

1. In the left example an attempted interpolation function y = f(x) for three points gives a parabola that has no real roots.
2. The method crashes immediately.
3. IQI creates an inverted parabola x = f(y) of y = f(x) that cuts the curve of interest (blue) in two points (figure below).
4. Then the equation for the next estimate of the root (the only estimate for the root) is given below left.
5. This particular implementation means that the three points in x y are modelled by a new function which is the inverse of the original one, and a root exists for this new expression.

THE UNIVERSITY of EDINBURGH
School of Engineering

Inverse Quadratic Interpolation: Code

```
def inverse_quadratic_interpolation(f, x0, x1, x2, max_iter=20,
tolerance=1e-5):
    steps_taken = 0
    while steps_taken < max_iter and abs(x1-x0) > tolerance: # last
guess and new guess are v close
        fx0 = f(x0)
        fx1 = f(x1)
        fx2 = f(x2)
        L0 = (x0 * fx1 * fx2) / ((fx0 - fx1) * (fx0 - fx2))
        L1 = (x1 * fx0 * fx2) / ((fx1 - fx0) * (fx1 - fx2))
        L2 = (x2 * fx1 * fx0) / ((fx2 - fx0) * (fx2 - fx1))
        new = L0 + L1 + L2
        x0, x1, x2 = new, x0, x1
        steps_taken += 1
    return x0, steps_taken

f = lambda x: x**2 - 20

root, steps = inverse_quadratic_interpolation(f, 4.3, 4.4, 4.5)
print ("root is:", root)
print ("steps taken:", steps)
```

THE UNIVERSITY of EDINBURGH
School of Engineering

Multiple (Repeated) Root Technique

1. A multiple root exists for the condition that a point on a function **touches** the x-axis.

$$f(x) = (x-3)(x-1)(x-1)$$

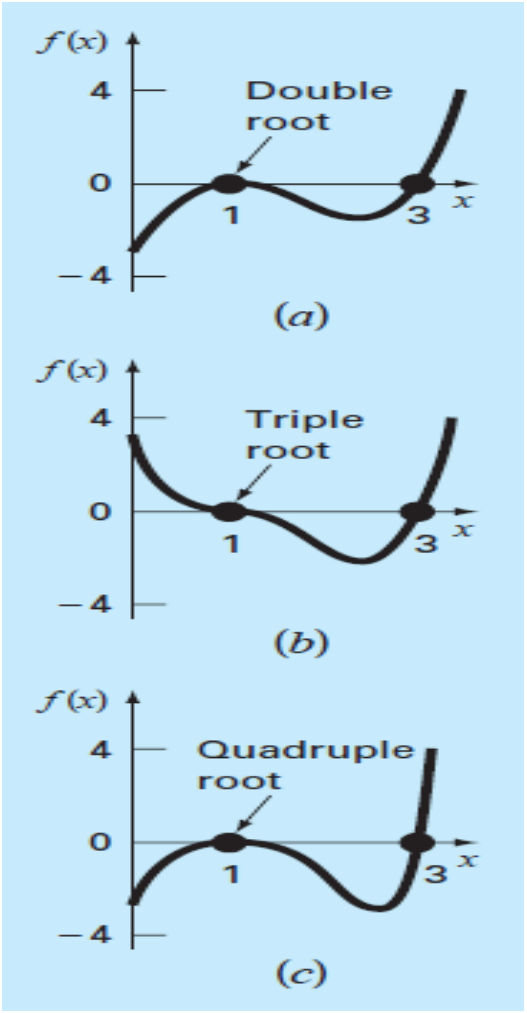
2. In the function above, there are two terms for which the same solution, x = 1 will deliver a function evaluation of zero.

3. This occurs at the leftmost tangential point of the curve with the x-axis in Figure a) opposite, with the remaining root at an independent point at x = 3.

4. A triple root exists for the similar equation.

$$f(x) = (x-3)(x-1)(x-1)(x-1)$$

5. In this case, the function crosses the point once.



THE UNIVERSITY of EDINBURGH
School of Engineering

Multiple (Repeated) Root Technique

General Rules for Multiple Roots

1. Odd numbers of multiple roots cross the axis, whereas even numbers of multiple roots do not.
2. Multiple roots cannot be solved using bracketing methods because they do not change sign.
3. Only open search methods may be used to solve.

Implications for Solution Strategy to be Used

4. For Newton Raphson and Secant, it is difficult to execute calculations around the repeated root zone where f'(x) is zero.
5. Also, NR is only linearly, rather than quadratically convergent in the vicinity of a repeated root.
6. One solution to these issues is to use **the Ralston-Rabinowitz method.**

THE UNIVERSITY of EDINBURGH
School of Engineering

Ralston-Rabinowitz for Multiple Roots

1. Firstly, we define a function which is the ratio of the function at x to its derivative at x.

$$u(x) = \frac{f(x)}{f'(x)}$$

2. This function u(x) and its derivative u'(x) are now substituted for f(x) and f'(x) in the Newton-Raphson expression as follows:

$$x_{i+1} = x_i - \frac{u(x_i)}{u'(x_i)}$$

3. To find u'(x), we need to compute it in terms of f'(x) and f(x) as follows:

$$u'(x) = \frac{f'(x)f'(x) - f(x)f''(x)}{[f'(x)]^2}$$

4. Substituting this into Newton-Raphson we get:

$$x_{i+1} = x_i - \frac{f(x_i)f'(x_i)}{[f'(x_i)]^2 - f(x_i)f''(x_i)}$$

THE UNIVERSITY of EDINBURGH
School of Engineering

Ralston-Rabinowitz for Multiple Roots

9. Let's apply this approach to the function we discussed earlier

$$f(x) = (x-3)(x-1)(x-1)$$

10. The first and second derivatives of this are:

$$f'(x) = 3x^2 - 10x + 7, \qquad f''(x) = 6x - 10,$$

11. Implementing Ralston-Rabinowitz we get:

$$x_{i+1} = x_i - \frac{(x_i^3 - 5x_i^2 + 7x_i - 3)(3x_i^2 - 10x_i + 7)}{(3x_i^2 - 10x_i + 7)^2 - (x_i^3 - 5x_i^2 + 7x_i - 3)(6x_i - 10)}$$

12. The tabulated solutions using this algorithm are as follows:

i	x_i	ε_t (%)
0	0	100
1	1.105263	11
2	1.003082	0.31
3	1.000002	0.00024

13. **Exercise: Write a code to implement this technique in Python.**