Numerical Phase 2

Team Members:

1. Ali Amr Abdullah ID:23010589
2. Belal Mohamed Hussien ID:23010341
3. Abdelarhman Ali Salaheldin ID:23010530
4. Mohamed Ahmed Salama ID:23010716
5. NourEldeen Mohamed Abas ID:23010920

BISECTION METHOD PSEUDOCODE

Algorithm: Bisection Method for Root Finding

INPUT PARAMETERS:

- **func**: The function f(x) for which to find the root
- **interval**: [xl, xu] - Initial bracket containing the root
- **epsilon**: Convergence tolerance (default: 0.00001)
- **max_iter**: Maximum number of iterations (default: 50)
- **precision**: Number of significant figures for calculations

OUTPUT:

- **root**: Approximate root of the equation
- **iterations**: Number of iterations performed
- **approx_error**: Approximate relative error percentage
- **steps**: Array of iteration details
- **status**: Convergence status message
- **significant_figures**: Number of reliable significant figures

---

PROCEDURE:

1. INITIALIZATION:

- Convert xl and xu to SigFloat with specified precision
- Set xr_old = 0
- Set ea (approximate error) = 100.0
- Create empty steps array

2. VALIDATE INITIAL BRACKET:

- Calculate f(xl) × f(xu)
- **IF** f(xl) × f(xu) > 0 **THEN**
    - **RETURN** "No root found in interval (no sign change)"
- **END IF**

3. ITERATE (i = 1 to max_iter):

a. CALCULATE NEW MIDPOINT:

- xr = (xl + xu) / 2

b. CALCULATE APPROXIMATE ERROR:

- **IF** xr ≠ 0 **AND** i > 1 **THEN**
    - ea = |((xr - xr_old) / xr)| × 100
- **ELSE**
    - ea = 100.0
- **END IF**

c. STORE ITERATION DATA:

- steps[i] = {iteration: i, root: xr, error: ea, xl: xl, xu: xu}

d. CHECK CONVERGENCE:

- **IF** ea < epsilon **THEN**
    - Calculate significant_figures from ea
    - **RETURN** {root: xr, status: "Converged", iterations: i, approx_error: ea, steps: steps, significant_figures: sig_figs}
- **END IF**

e. UPDATE BRACKET:

- test = f(xl) × f(xr)
- **IF** test < 0 **THEN**
    - xu = xr *(Root is in left half)*
- **ELSE IF** test > 0 **THEN**
    - xl = xr *(Root is in right half)*
- **ELSE IF** test = 0 **THEN**
    - **RETURN** {root: xr, status: "Exact Root Found", iterations: i, approx_error: 0.0, steps: steps, significant_figures: precision}
- **ELSE**
    - **RETURN** {status: "Error: Function calculation failed (NaN/Inf)"}
- **END IF**

f. UPDATE OLD VALUE:

- xr_old = xr

4. IF MAX ITERATIONS REACHED:

- Calculate significant_figures from final ea
- **RETURN** {root: xr, status: "Max Iterations Reached", iterations: max_iter, approx_error: ea, steps: steps, significant_figures: sig_figs}

KEY CONCEPTS:

Bisection Principle: The method repeatedly bisects the interval [xl, xu] where the function changes sign. Each iteration halves the search space by keeping the half where the sign change occurs.

Convergence Criterion: Convergence is measured by the relative approximate error (ea), defined as:

- ea = |((xr - xr_old) / xr)| × 100%

Significant Figures Calculation: When converged, the number of significant figures is estimated as:

- sig_figs = floor(2 - $\log_{10}$(2 × ea))

Guaranteed Convergence: The algorithm guarantees convergence if:

1. A root exists in the initial interval
2. The function is continuous in [xl, xu]
3. f(xl) and f(xu) have opposite signs

ALGORITHM COMPLEXITY:

- **Time Complexity**: O($\log_2$((xu - xl) / epsilon))
- **Space Complexity**: O(n) where n is the number of iterations stored

FALSE POSITION METHOD

INPUT PARAMETERS:

- **func**: The function f(x) for which to find the root
- **interval**: [xl, xu] - Initial bracket containing the root
- **epsilon**: Convergence tolerance (default: 0.00001)
- **max_iter**: Maximum number of iterations (default: 50)
- **precision**: Number of significant figures for calculations

OUTPUT:

- **root**: Approximate root of the equation
- **iterations**: Number of iterations performed
- **approx_error**: Approximate relative error percentage
- **steps**: Array of iteration details
- **status**: Convergence status message
- **significant_figures**: Number of reliable significant figures

---

PROCEDURE:

1. INITIALIZATION:

- Convert xl and xu to SigFloat with specified precision
- Set xr_old = 0
- Set ea (approximate error) = 100.0
- Create empty steps array

2. VALIDATE INITIAL BRACKET:

- Calculate f(xl) × f(xu)
- **IF** f(xl) × f(xu) > 0 **THEN**
  - **RETURN** "No root found in interval (no sign change)"
- **END IF**

3. ITERATE (i = 1 to max_iter):

a. CALCULATE FUNCTION VALUES:

- fxl = f(xl)
- fxu = f(xu)

b. CALCULATE NEW ROOT USING FALSE POSITION FORMULA:

- numerator = (xl × fxu) - (xu × fxl)
- denominator = fxu - fxl
- **IF** denominator = 0 **THEN**
  - **RETURN** {status: "Error: Denominator is zero (flat slope)"}
- **END IF**
- xr = numerator / denominator

c. CALCULATE APPROXIMATE ERROR:

- **IF** xr ≠ 0 **AND** i > 1 **THEN**
  - ea = |((xr - xr_old) / xr)| × 100
- **ELSE**
  - ea = 100.0
- **END IF**

d. STORE ITERATION DATA:

- steps[i] = {iteration: i, root: xr, error: ea, xl: xl, xu: xu, f(xr): f(xr)}

e. CHECK CONVERGENCE:

- **IF** ea < epsilon **THEN**
  - Calculate significant_figures from ea
  - **RETURN** {root: xr, status: "Converged", iterations: i, approx_error: ea, steps: steps, significant_figures: sig_figs}
- **END IF**

f. UPDATE BRACKET:

- test = f(xl) × f(xr)
- **IF** test < 0 **THEN**
  - xu = xr *(Root is in left half)*
- **ELSE IF** test > 0 **THEN**
  - xl = xr *(Root is in right half)*
- **ELSE IF** test = 0 **THEN**
  - **RETURN** {root: xr, status: "Exact Root Found", iterations: i, approx_error: 0.0, steps: steps, significant_figures: precision}
- **ELSE**
  - **RETURN** {status: "Error: Function calculation failed (NaN/Inf)"}
- **END IF**

g. UPDATE OLD VALUE:

- xr_old = xr

4. IF MAX ITERATIONS REACHED:

- Calculate significant_figures from final ea
- **RETURN** {root: xr, status: "Max Iterations Reached", iterations: max_iter, approx_error: ea, steps: steps, significant_figures: sig_figs}

KEY CONCEPTS:

False Position Formula: The method uses linear interpolation to find where the line between (xl, f(xl)) and (xu, f(xu)) crosses the x-axis:

- xr = xl - f(xl) × ((xu - xl) / (f(xu) - f(xl)))

Convergence Criterion: Convergence is measured by the relative approximate error (ea), defined as:

- ea = |((xr - xr_old) / xr)| × 100%

Significant Figures Calculation: When converged, the number of significant figures is estimated as:

- sig_figs = floor(2 - $\log_{10}$(2 × ea))

Advantages over Bisection:

1. Generally faster convergence than bisection
2. Uses function values to make better estimates
3. Always retains the bracket containing the root

COMPARISON WITH BISECTION METHOD:

Feature	Bisection	False Position
Convergence Rate	Linear (slower)	Super-linear (faster)
Formula	xr = (xl + xu) / 2	xr = (xl×f(xu) - xu×f(xl)) / (f(xu) - f(xl))
Function Evaluations	1 per iteration	2 per iteration
Bracket Update	Always halves interval	Uses interpolation
Reliability	Very reliable	Can stall if one-sided

ALGORITHM COMPLEXITY:

- **Time Complexity**: O(n) where n depends on function behavior
- **Space Complexity**: O(n) where n is the number of iterations stored
- **Typical Convergence**: Faster than bisection but slower than Newton-Raphson

NEWTON-RAPHSON METHOD

INPUT PARAMETERS:

- **func**: The function f(x) for which to find the root
- **initial_guess**: Starting value x_0
- **precision**: Number of significant figures for calculations (default: 5)
- **epsilon**: Convergence tolerance (default: 0.00001)
- **max_iter**: Maximum number of iterations (default: 50)

OUTPUT:

- **root**: Approximate root of the equation
- **iterations**: Number of iterations performed
- **approx_error**: Approximate relative error percentage
- **steps**: Array of iteration details

PROCEDURE:

1. INITIALIZATION:

- Set x_old = initial_guess (x_0)
- Convert x_old to SigFloat with specified precision
- Create empty steps array
- Set iteration counter itr = 0

2. ITERATE (itr = 1 to max_iter):

a. CALCULATE DERIVATIVE:

- derv = f'(x_old)
- Use numerical differentiation: derv = (f(x + h) - f(x)) / h
- Where h = 1e-6 (small perturbation)
- **IF** derv = 0 **THEN**
  - **RAISE ERROR**: "Derivative became zero – can't continue"
  - **STOP EXECUTION**
- **END IF**

b. APPLY NEWTON-RAPHSON FORMULA:

- x_next = x_old - (f(x_old) / f'(x_old))

c. CALCULATE APPROXIMATE ERROR:

- ea = |((x_next - x_old) / x_next)| × 100

d. STORE ITERATION DATA:

- steps[itr] = {iteration: itr, root: x_next, error: ea}

e. CHECK CONVERGENCE:

- **IF** ea < epsilon **THEN**
  - **BREAK** (Exit loop - converged)
- **END IF**

f. UPDATE FOR NEXT ITERATION:

- x_old = x_next

3. RETURN RESULTS:

- **RETURN** {root: x_next, iterations: itr, approx_error: ea, steps: steps}

NUMERICAL DERIVATIVE CALCULATION:

Function: derivative(f, x)

Purpose: Calculate f'(x) numerically using finite difference method

Method: Forward Difference Approximation

- f'(x) ≈ (f(x + h) - f(x)) / h
- Where h = 1e-6

KEY CONCEPTS:

Newton-Raphson Formula: The method uses the tangent line at the current point to estimate the next approximation:

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

Geometric Interpretation:

1. Start with initial guess x_0
2. Draw tangent line to f(x) at point (x_0, f(x_0))
3. Find where tangent crosses x-axis → this is x_1
4. Repeat process with x_1 to get x_2, and so on

Convergence Criterion:

- ea = |((x_next - x_old) / x_next)| × 100%
- Method converges when ea < epsilon

Convergence Conditions: The method converges quadratically if:

1. Initial guess x_0 is sufficiently close to the root
2. f'(x) ≠ 0 in the neighborhood of the root
3. f''(x) is continuous near the root

COMPARISON WITH OTHER METHODS:

Feature	Newton-Raphson	Bisection	False Position	Fixed Point
Convergence Rate	Quadratic (fastest)	Linear	Super-linear	Linear
Requires Derivative	Yes	No	No	No
Requires Bracket	No	Yes	Yes	No
Guaranteed Convergence	No	Yes	Yes	Conditional
Function Evaluations	2 per iteration	1 per iteration	2 per iteration	1 per iteration
Initial Guess Sensitivity	Very High	Low	Low	High
Typical Iterations	3-6	10-20	5-15	10-30

PRACTICAL IMPLEMENTATION NOTES:

Handling Division by Zero:

```
IF derv = 0 THEN
    RAISE ERROR: "Derivative is zero at x = ..."
    STOP EXECUTION
END IF
```

Initial Guess Selection:

1. Plot the function to visualize roots
2. Use bisection first to get rough estimate
3. Ensure x_0 is in region where f'(x) ≠ 0
4. For polynomial equations, use analytical methods for initial estimate

---

ALGORITHM COMPLEXITY:

- **Time Complexity**: O(n) where n is number of iterations (typically very small)
- **Space Complexity**: O(n) where n is the number of iterations stored
- **Typical Performance**: 3-6 iterations for most problems

EXAMPLE EXECUTION:

Problem: Find root of $f(x) = x^2 - 3$

Initial Setup:

- $f(x) = x^2 - 3$
- $f'(x) = 2x$
- initial_guess = 2
- epsilon = 0.00001

Iteration Process:

Iteration	x_i	$f(x_i)$	$f'(x_i)$	x_{i+1}	ea (%)
1	2.0	1.0	4.0	1.75	14.29
2	1.75	0.0625	3.5	1.7321	1.04
3	1.7321	~0	3.464	1.7321	< 0.001

Result: Root ≈ 1.7321 (√3) in 3 iterations

MODIFIED NEWTON-RAPHSON

INPUT PARAMETERS:

- **func**: The function f(x) for which to find the root
- **initial_guess**: Starting value x_0
- **multiplicity**: Multiplicity of the root (m) - default: 1
- **precision**: Number of significant figures for calculations (default: 5)
- **epsilon**: Convergence tolerance (default: 0.00001)
- **max_iter**: Maximum number of iterations (default: 50)

OUTPUT:

- **root**: Approximate root of the equation
- **iterations**: Number of iterations performed
- **approx_error**: Approximate relative error percentage
- **steps**: Array of iteration details

PROCEDURE:

1. INITIALIZATION:

- Set x_old = initial_guess (x_0)
- Convert x_old to SigFloat with specified precision
- Set m = multiplicity (root multiplicity)
- Create empty steps array
- Set iteration counter itr = 0

2. ITERATE (itr = 1 to max_iter):

a. CALCULATE DERIVATIVE:

- derv = f'(x_old)
- Use numerical differentiation: derv = (f(x + h) - f(x)) / h
- Where h = 1e-6 (small perturbation)
- Convert derv to SigFloat with specified precision
- **IF** derv = 0 **THEN**
  - **RAISE ERROR**: "Derivative became zero – can't continue"
  - **STOP EXECUTION**
- **END IF**

b. APPLY MODIFIED NEWTON-RAPHSON FORMULA:

- x_next = x_old - m × (f(x_old) / f'(x_old))

Note: The multiplicity factor m accelerates convergence for multiple roots

c. CALCULATE APPROXIMATE ERROR:

- **IF** x_next ≠ 0 **THEN**
  - ea = |((x_next - x_old) / x_next)| × 100
- **ELSE**
  - ea = 0
- **END IF**

d. STORE ITERATION DATA:

- steps[itr] = {iteration: itr, root: x_next, error: ea}

e. CHECK CONVERGENCE:

- **IF** ea < epsilon **THEN**
  - **BREAK** (Exit loop - converged)
- **END IF**

f. UPDATE FOR NEXT ITERATION:

- x_old = x_next

3. RETURN RESULTS:

- **RETURN** {root: x_next, iterations: itr, approx_error: ea, steps: steps}

NUMERICAL DERIVATIVE CALCULATION:

Function: derivative(f, x)

Purpose: Calculate f'(x) numerically using finite difference method

Method: Forward Difference Approximation

- f'(x) ≈ (f(x + h) - f(x)) / h
- Where h = 1e-6

KEY CONCEPTS:

Modified Newton-Raphson Formula: The method modifies the standard Newton-Raphson by including multiplicity factor m:

Standard Newton-Raphson:

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

Modified Newton-Raphson:

- $x_{i+1} = x_i - m \times (f(x_i) / f'(x_i))$

Where m is the multiplicity of the root.

Convergence Comparison:

Root Type	Standard N-R	Modified N-R (with m)
Simple root (m=1)	Quadratic	Quadratic
Double root (m=2)	Linear	Quadratic
Triple root (m=3)	Linear	Quadratic

COMPARISON WITH OTHER METHODS:

Feature	Modified N-R	Standard N-R	Bisection	False Position
Convergence for Simple Roots	Quadratic	Quadratic	Linear	Super-linear
Convergence for Multiple Roots	Quadratic	Linear	Linear	Linear
Requires Multiplicity	Yes	No	No	No
Requires Derivative	Yes	Yes	No	No
Typical Iterations (m=1)	3-6	3-6	10-20	5-15
Typical Iterations (m=2)	3-6	15-30	10-20	5-15

ALGORITHM COMPLEXITY:

- **Time Complexity**: O(n) where n is number of iterations (typically very small)
- **Space Complexity**: O(n) where n is the number of iterations stored
- **Performance**: Excellent for multiple roots when m is known

SECANT METHOD PSEUDOCODE

INPUT PARAMETERS:

- **func**: The function f(x) for which to find the root
- **initial_guess_1**: First starting value x_{-1}
- **initial_guess_2**: Second starting value x_0
- **epsilon**: Convergence tolerance (default: 0.00001)
- **max_iter**: Maximum number of iterations (default: 50)
- **precision**: Number of significant figures for calculations

OUTPUT:

- **root**: Approximate root of the equation
- **iterations**: Number of iterations performed
- **approx_error**: Approximate relative error percentage
- **steps**: Array of iteration details
- **status**: Convergence status message
- **significant_figures**: Number of reliable significant figures

---

PROCEDURE:

1. INITIALIZATION:

- Set x_prev = initial_guess_1 (x_{-1})
- Set x_curr = initial_guess_2 (x_0)
- Convert both to SigFloat with specified precision
- Set ea (approximate error) = 100.0
- Create empty steps array
- **OPTIMIZATION**: Calculate f_prev = f(x_prev) once (reuse in loop)

2. ITERATE (i = 1 to max_iter):

a. CALCULATE CURRENT FUNCTION VALUE:

- f_curr = f(x_curr)

b. CALCULATE DENOMINATOR:

- denominator = f_curr - f_prev
- **IF** denominator = 0 **THEN**
  - **RETURN** {status: "Error: Denominator is zero (Flat slope or same guesses)"}
- **END IF**

c. APPLY SECANT FORMULA:

- numerator = f_curr × (x_curr - x_prev)
- x_next = x_curr - (numerator / denominator)

Try-Catch Block:

- **TRY**:
  - Calculate x_next
  - Convert x_next to SigFloat with specified precision
- **CATCH** (OverflowError, ValueError):
  - **RETURN** {status: "Error: Diverged (Overflow/Value Error)"}
- **END TRY**

d. CALCULATE APPROXIMATE ERROR:

- **IF** x_next ≠ 0 **THEN**
  - ea = |((x_next - x_curr) / x_next)| × 100
- **ELSE**
  - ea = 100.0
- **END IF**

e. STORE ITERATION DATA:

- steps[i] = {iteration: i, root: x_next, error: ea, x_prev: x_prev, x_curr: x_curr, f(x_curr): f_curr}

f. CHECK CONVERGENCE:

- **IF** ea < epsilon **THEN**
  - Calculate significant_figures from ea
  - **RETURN** {root: x_next, status: "Converged", iterations: i, approx_error: ea, steps: steps, significant_figures: sig_figs}
- **END IF**

g. UPDATE FOR NEXT ITERATION:

- x_prev = x_curr
- f_prev = f_curr *(Efficiency: Recycle function value)*
- x_curr = x_next

3. IF MAX ITERATIONS REACHED:

- Calculate significant_figures from final ea
- **RETURN** {root: x_curr, status: "Max Iterations Reached", iterations: max_iter, approx_error: ea, steps: steps, significant_figures: sig_figs}

KEY CONCEPTS:

Secant Method Formula: The method approximates the derivative using two points:

Main Formula:

- $x_{i+1} = x_i - (f(x_i) \times (x_i - x_{i-1})) / (f(x_i) - f(x_{i-1}))$

Geometric Interpretation:

1. Start with two initial guesses x_{-1} and x_0
2. Draw a secant line through points $(x_{-1}, f(x_{-1}))$ and $(x_0, f(x_0))$
3. Find where secant line crosses x-axis → this is x_1
4. Repeat using x_0 and x_1 to get x_2, and so on

Derivative Approximation: The secant method approximates $f'(x_i)$ as:

- $f'(x_i) \approx (f(x_i) - f(x_{i-1})) / (x_i - x_{i-1})$

This makes it a **derivative-free** version of Newton-Raphson.

RELATIONSHIP TO NEWTON-RAPHSON:

Newton-Raphson:

- $x_{i+1} = x_i - f(x_i) / f'(x_i)$
- Requires derivative $f'(x)$
- Uses tangent line

Secant Method:

- $x_{i+1} = x_i - f(x_i) / ((f(x_i) - f(x_{i-1})) / (x_i - x_{i-1}))$
- No derivative needed
- Uses secant line

Key Insight: Secant replaces $f'(x_i)$ with finite difference approximation.

CONVERGENCE ANALYSIS:

Comparison of Convergence Orders:

Method	Convergence Order	Speed
Bisection	1 (Linear)	Slowest
False Position	~1.2 (Super-linear)	Slow
Secant	**1.618 (Super-linear)**	**Fast**
Newton-Raphson	2 (Quadratic)	Fastest

Convergence Conditions:

The method converges if:

1. Initial guesses x_{-1} and x_0 are sufficiently close to the root
2. $f(x_i) - f(x_{i-1}) \neq 0$ throughout iterations
3. Function is continuous and smooth near the root

ADVANTAGES AND DISADVANTAGES:

Advantages:

1. **No derivative required** - only needs function evaluations
2. Super-linear convergence (faster than bisection)
3. Often nearly as fast as Newton-Raphson
4. Simpler implementation than Newton-Raphson (no derivative calculation)
5. Works well when derivative is difficult or expensive to compute

Disadvantages:

1. Requires **two initial guesses** instead of one
2. Not guaranteed to converge
3. May fail if $f(x_i) - f(x_{i-1})$ becomes zero (flat slope)
4. Slower than Newton-Raphson (1.618 vs 2.0 convergence)
5. Sensitive to initial guesses
6. Can diverge with poor starting values

COMPARISON WITH OTHER METHODS:

Feature	Secant	Newton-Raphson	Bisection	False Position
Convergence Rate	1.618	2.0	1.0	~1.2
Requires Derivative	No	Yes	No	No
Initial Values	2 guesses	1 guess	2 (bracket)	2 (bracket)
Guaranteed Convergence	No	No	Yes	Yes
Function Evaluations/Iter	1*	2	1	2
Typical Iterations	4-8	3-6	10-20	5-15
Complexity	Low	Medium	Low	Low

EFFICIENCY CONSIDERATIONS:

Function Evaluation Optimization:

The implementation reuses calculated function values:

```
f_prev = f(x_prev)    // Calculate once before loop

FOR each iteration:
    f_curr = f(x_curr)
    // Use f_prev and f_curr
    ...
    f_prev = f_curr   // Recycle for next iteration
END FOR
```

Benefit: Only 1 function evaluation per iteration (after first iteration)

Efficiency Comparison:

Method	Evaluations per Iteration	Total for Convergence
Secant	1 (after first)	~5-10
Newton-Raphson	2 (f and f')	~6-12
Bisection	1	~10-20

When derivative is expensive: Secant may be faster overall than Newton-Raphson.

ALGORITHM COMPLEXITY:

- **Time Complexity**: O(n × f) where n is iterations and f is function evaluation cost
- **Space Complexity**: O(n) where n is the number of iterations stored
- **Typical Performance**: 4-8 iterations for most problems

EXAMPLE EXECUTION:

Problem: Find root of $f(x) = x^3 - 2x - 5$

Initial Setup:

- $f(x) = x^3 - 2x - 5$
- initial_guess_1 = 2.0
- initial_guess_2 = 3.0
- epsilon = 0.00001

Iteration Process:

i	x_{i-1}	x_i	f(x_{i-1})	f(x_i)	x_{i+1}	ea (%)
0	2.0	3.0	-1.0	16.0	-	-
1	2.0	3.0	-1.0	16.0	2.0588	45.7
2	3.0	2.0588	16.0	-0.390	2.0946	1.71
3	2.0588	2.0946	-0.390	0.100	2.0946	0.00

Result: Root ≈ 2.0946 in 3 iterations

PRACTICAL IMPLEMENTATION NOTES:

Choosing Initial Guesses:

1. Select x_{-1} and x_0 on opposite sides of root (if possible)
2. Keep guesses reasonably close to expected root
3. Ensure $f(x_{-1}) \neq f(x_0)$ to avoid zero denominator
4. Use graphical analysis or bisection to get good starting points

When to Use Secant Method:

1. When derivative is difficult to calculate analytically
2. When derivative is expensive to compute
3. When you need faster convergence than bisection
4. When Newton-Raphson is impractical

Convergence Criteria:

```
IF ea < epsilon THEN
    sig_figs = floor(2 - log₁₀(2 × ea))
    CONVERGED
END IF
```

FIXED POINT ITERATION METHOD PSEUDOCODE

Algorithm: Fixed Point Iteration Method for Root Finding

INPUT PARAMETERS:

- **func**: The function g(x) - reformulated equation where x = g(x)
- **initial_guess**: Starting value x_0
- **epsilon**: Convergence tolerance (default: 0.00001)
- **max_iter**: Maximum number of iterations (default: 50)
- **precision**: Number of significant figures for calculations

OUTPUT:

- **root**: Approximate root of the equation
- **iterations**: Number of iterations performed
- **approx_error**: Approximate relative error percentage
- **steps**: Array of iteration details
- **status**: Convergence status message
- **correct_sf**: Number of correct significant figures
- **execution_time**: Time taken for computation

PROCEDURE:

1. INITIALIZATION:

- Set x_old = initial_guess (x_0)
- Apply significant figure rounding to x_old
- Create empty steps array
- Set approx_error = 0.0
- **START TIMER**: start_time = current_time()

2. SIGNIFICANT FIGURE HELPER FUNCTION:

Function: sig(val)

```
IF val = 0 THEN
    RETURN 0.0
END IF

IF precision is None THEN
    RETURN val (no rounding)
END IF

TRY:
    format_str = "{:." + (precision - 1) + "e}"
    RETURN float(format_str.format(val))
CATCH:
```

```
    RETURN val (if formatting fails)
END TRY
```

3. ITERATE (iteration = 1 to max_iter):

a. CALCULATE NEW APPROXIMATION:

- **TRY**:
    - x_new = sig(g(x_old))
- **CATCH** (OverflowError, ValueError):
    - **RETURN** {root: x_old, status: "Diverged", iterations: iteration, approx_error: approx_error, execution_time: 0.0, steps: steps}
- **END TRY**

b. CALCULATE APPROXIMATE ERROR:

- **IF** x_new ≠ 0 **AND** iteration > 1 **THEN**
    - approx_error = |((x_new - x_old) / x_new)| × 100
- **ELSE**
    - approx_error = 100.0
- **END IF**
- approx_error = sig(approx_error)

c. STORE ITERATION DATA:

- steps[iteration] = {iteration: iteration, x0: x_old, x1: x_new, approx_error: approx_error}

d. CHECK CONVERGENCE:

- **IF** approx_error < epsilon **THEN**
    - **STOP TIMER**: end_time = current_time()
    - Calculate correct_sf = calculate_correct_sf(approx_error)
    - **RETURN** {root: x_new, status: "Converged", iterations: iteration, approx_error: approx_error, steps: steps, correct_sf: correct_sf, execution_time: end_time - start_time}
- **END IF**

e. UPDATE FOR NEXT ITERATION:

- x_old = x_new

4. IF MAX ITERATIONS REACHED:

- **STOP TIMER**: end_time = current_time()
- Calculate correct_sf = calculate_correct_sf(approx_error)
- **RETURN** {root: x_new, status: "Max iterations reached", iterations: max_iter, approx_error: approx_error, steps: steps, correct_sf: correct_sf, execution_time: end_time - start_time}

CALCULATE CORRECT SIGNIFICANT FIGURES:

Function: calculate_correct_sf(error)

Purpose: Estimate number of correct significant figures based on error

Procedure:

```
IF error ≤ 0 THEN
    RETURN 15    // Maximum precision
END IF

TRY:
    // Formula: m = floor(-log₁₀(error))
    correct_sf = floor(-log₁₀(error))
    RETURN correct_sf
CATCH:
    RETURN 0    // If calculation fails
END TRY
```

Formula Explanation:

- If error = 0.1% → correct_sf = floor($-\log_{10}(0.1)$) = floor(1) = 1
- If error = 0.01% → correct_sf = floor($-\log_{10}(0.01)$) = floor(2) = 2
- If error = 0.001% → correct_sf = floor($-\log_{10}(0.001)$) = floor(3) = 3

KEY CONCEPTS:

Fixed Point Principle: To solve f(x) = 0, reformulate the equation as x = g(x), where g(x) is called the iteration function. The solution is a "fixed point" of g(x).

Reformulation Examples:

1. **Original**: $x^2 - 2x - 3 = 0$
   - **Reformulated**: $x = (x^2 - 3) / 2$ → $g(x) = (x^2 - 3) / 2$
2. **Original**: $x^3 + x - 1 = 0$
   - **Reformulated**: $x = 1 - x^3$ → $g(x) = 1 - x^3$
   - **Alternative**: $x = \sqrt[3]{(1 - x)}$ → $g(x) = \sqrt[3]{(1 - x)}$
3. **Original**: $e^x - x - 2 = 0$
   - **Reformulated**: $x = \ln(x + 2)$ → $g(x) = \ln(x + 2)$

Iterative Formula:

- $x_{i+1} = g(x_i)$

Convergence Criterion:

- ea = |((x_new - x_old) / x_new)| × 100%
- Method converges when ea < epsilon

Convergence Condition: The method converges if:

- $|g'(x)| < 1$ in the neighborhood of the root

Where:

- $g'(x)$ is the derivative of the iteration function
- If $|g'(x)| \geq 1$, the method will diverge

SIGNIFICANT FIGURE HANDLING:

Purpose:

Control numerical precision throughout calculations to maintain specified significant figures.

Implementation:

Uses scientific notation formatting to round values:

- precision = 5 → format as 1.234e+00 (4 decimal places in mantissa)
- Ensures consistent precision across all calculations

Example:

```
IF precision = 5:
    123.456789 → 1.2346e+02 → 123.46
    0.00123456 → 1.2346e-03 → 0.0012346
END IF
```

ADVANTAGES AND DISADVANTAGES:

Advantages:

1. **Simple implementation** - only one function evaluation per iteration
2. **No derivative required** - only needs $g(x)$
3. **Works well** when convergence conditions are met
4. **Flexible** - multiple reformulation options
5. **Performance tracking** - includes execution time

Disadvantages:

1. **Convergence depends on g(x) choice** - may require trial and error
2. **May diverge** if $|g'(x)| \geq 1$
3. **Generally slower** than Newton-Raphson
4. **Requires careful reformulation** of the original equation
5. **No guaranteed convergence**

COMPARISON WITH OTHER METHODS:

Feature	Fixed Point	Newton-Raphson	Bisection	Secant
Convergence Rate	Linear	Quadratic	Linear	1.618
Requires Derivative	No	Yes	No	No
Requires Bracket	No	No	Yes	No
Guaranteed Convergence	Conditional	Conditional	Yes	Conditional
Function Evaluations	1 per iteration	2 per iteration	1 per iteration	1 per iteration
Initial Values	1 guess	1 guess	2 (bracket)	2 guesses
Reformulation Needed	Yes	No	No	No

ALGORITHM COMPLEXITY:

- **Time Complexity**: O(n) where n is the number of iterations
- **Space Complexity**: O(n) where n is the number of iterations stored
- **Convergence**: Depends on $|g'(x)|$ near the root
- **Performance**: Includes timing for benchmarking

EXAMPLE EXECUTION:

Example 1: Solving $x^3 + x - 1 = 0$

Reformulation: $x = 1 - x^3 \rightarrow g(x) = 1 - x^3$

Check Convergence:

- $g'(x) = -3x^2$
- Near root $x \approx 0.68$: $|g'(0.68)| = |-3(0.68)^2| \approx 1.39 > 1$ ✕ **Will Diverge!**

Better Reformulation: $x = \sqrt[3]{1 - x} \rightarrow g(x) = \sqrt[3]{1 - x}$

Check Convergence:

- $g'(x) = -1/(3\sqrt[3]{(1-x)^2})$
- Near root $x \approx 0.68$: $|g'(0.68)| \approx 0.44 < 1$ ✓ **Will Converge!**

Iteration Process:

Iteration	x_old	x_new = ∛(1 - x_old)	ea (%)
1	0.5	0.7937	37.0
2	0.7937	0.6427	23.5
3	0.6427	0.7098	9.5
4	0.7098	0.6736	5.4
5	0.6736	0.6933	2.8
10	...	0.6823	< 0.001

Result: Root ≈ 0.6823 in ~10 iterations

Example 2: Solving cos(x) = x

Reformulation: x = cos(x) → g(x) = cos(x)

Check Convergence:

- g'(x) = -sin(x)
- Near root x ≈ 0.739: |g'(0.739)| = |-sin(0.739)| ≈ 0.674 < 1 ✓ **Will Converge!**

Iteration Process:

Iteration	x_old	x_new = cos(x_old)	ea (%)
1	0.5	0.8776	43.0
2	0.8776	0.6390	37.3
3	0.6390	0.8027	20.4
4	0.8027	0.6948	15.5
5	0.6948	0.7682	9.6
12	...	0.7391	< 0.001

Result: Root ≈ 0.7391 in ~12 iterations

DATA STRUCTURES USED IN NUMERICAL METHODS

Overview

This document describes all data structures used across the numerical methods implementations (Bisection, False Position, Fixed Point, Newton-Raphson, Modified Newton, and Secant methods).

---

1. DICTIONARY

Purpose

Primary data structure for passing parameters and returning results.

Usage

Input Parameters (kwargs)

```
{
    'interval': (xl, xu),           # For bracketing methods
    'initial_guess': float,         # For open methods
    'initial_guess_1': float,       # For Secant method
    'initial_guess_2': float,       # For Secant method
    'epsilon': float,               # Convergence tolerance
    'max_iter': int,                # Maximum iterations
    'precision': int,               # Significant figures
    'multiplicity': int             # For Modified Newton
}
```

Methods Using This

All methods (Bisection, False Position, Fixed Point, Newton-Raphson, Modified Newton, Secant)

---

3. SIGFLOAT CUSTOM OBJECT

Purpose

Custom class to maintain numerical precision throughout calculations.

Attributes

```
class SigFloat:
    value: float          # The actual numerical value
    precision: int        # Number of significant figures
```

Usage

```
xl = SigFloat(0.0, precision=6)
xu = SigFloat(3.0, precision=6)
xr = (xl + xu) / SigFloat(2, precision=6)
```

Advantages

- Controls significant figures consistently
- Prevents precision loss during operations
- Wraps standard float operations
- Maintains accuracy across calculations

Methods Using This

- Bisection
- False Position
- Newton-Raphson
- Modified Newton
- Secant

Methods NOT Using This

- Fixed Point (uses direct float with sig() helper function)

---

4. TUPLE

Purpose

Store interval boundaries for bracketing methods.

Structure

```
(xl, xu)  # Example: (0, 3) represents interval [0, 3]
```

Usage

```
interval = kwargs.get('interval', (0, 0))
xl, xu = interval
```

Advantages

- Immutable - prevents accidental modification
- Compact representation of two values
- Natural mathematical notation for intervals
- Lightweight memory usage

Methods Using This

- Bisection
- False Position

---

7. NUMPY ARRAY

Purpose

Generate iteration ranges efficiently.

Structure

```
numpy.ndarray  # 1-dimensional array of integers
```

Usage

```
import numpy as np
for itr in np.arange(1, max_iter + 1):
    # iteration code
```

Advantages

- Efficient iteration over ranges
- NumPy integration for mathematical operations
- Memory efficient for large ranges

Methods Using This

- Newton-Raphson
- Modified Newton

SUMMARY TABLE

Data Structure	Purpose	Used In Methods	Complexity
Dictionary	Parameters & results	All	O(1) access
List[Dict]	Iteration history	All	O(1) append, O(n) space
SigFloat	Precision control	All except Fixed Point	O(1) operations
Tuple	Interval boundaries	Bisection, False Position	O(1) access
Float	Numerical values	All	O(1) operations
Integer	Counters & config	All	O(1) operations
NumPy Array	Iteration ranges	Newton-Raphson, Modified Newton	O(n) creation
String	Status messages	All	O(1) access

PROGRAM SETUP AND EXECUTION GUIDE

1. Required Libraries and Installation

Python Version

- **Python 3.7 or higher** is required

Required Libraries

1. NumPy

Purpose: Numerical computations, array operations, mathematical functions

Installation:

```
pip install numpy
```

Usage in Program:

- Mathematical operations (floor, log10, etc.)
- Iteration ranges (`np.arange`)
- Numerical calculations

2. Standard Python Libraries

The following libraries are built-in with Python (no installation needed):

- **time**: For performance tracking and execution time measurement
- **math**: For mathematical functions (exp, log, etc.)

---

2. Installation Steps

Step 1: Install Python

1. Download Python from python.org
2. During installation, check "Add Python to PATH"
3. Verify installation:
4. `python --version`

Step 2: Install Required Packages

Method 1: Using pip (Recommended)

```
pip install numpy
```

Method 2: Using pip3 (for Linux/Mac)

```
pip3 install numpy
```

Method 3: Install all at once

```
pip install numpy
```

Step 3: Verify Installation

```
python -c "import numpy; print(numpy.__version__)"
```

Expected output: Version number (e.g., `1.24.3`)

Summary

Quick Start Commands

```
# 1. Install Python 3.7+
# 2. Install NumPy
pip install numpy

# 3. Verify installation
python -c "import numpy; print(numpy.__version__)"
```

Required Libraries Summary

Library	Installation	Purpose
numpy	`pip install numpy`	Numerical computations
time	Built-in	Execution time tracking
math	Built-in	Mathematical functions