EAFIT University

SCHOOL OF APPLIED SCIENCES AND ENGINEERING

NUMERICAL ANALYSIS REPORT PROJECT

Teacher: Edwar Samir Posada Murillo

Names: Edy Julius López Rojas, Victor Daniel Arango Sohm, Samuel Madrid Ossa & Carlos David Sanchez Soto

2025

Contents

Chapter 1

Numeric Methods

1.1 Solution of Nonlinear Equations

Nonlinear equations arise frequently in applied mathematics and engineering. In many cases, exact analytical solutions are not available, so numerical methods are employed to approximate the roots of functions. These techniques iteratively approach the solution with increasing accuracy, providing practical tools for real-world problems.

1.1.1 Incremental Search

Consecutive intervals of length Δx are evaluated until an interval $[a,b]$ is found where $f(a) \cdot f(b) < 0$, indicating the existence of at least one root in that interval. in summary

$$f(a) \cdot f(b) < 0 \quad \Rightarrow \quad \exists\, r \in (a,b) : f(r) = 0$$

Pseudocode

The user provides the input parameters under the assumptions that:

- f is a continuous function.
- f has at least one root in the given interval.

Algorithm 1 Incremental Search Method

procedure INCREMENTALSEARCH($f, x_0, \Delta x, N$)
    $a \leftarrow x_0$
    $fa \leftarrow f(a)$
    for $k \leftarrow 1$ to N **do**
        $b \leftarrow a + \Delta x$
        $fb \leftarrow f(b)$
        if $fa \cdot fb < 0$ **then**
            return Interval $[a, b]$ ▷ Root detected
        end if
        $a \leftarrow b$
        $fa \leftarrow fb$
    end for
    return "No root found within N steps"
end procedure

Testing

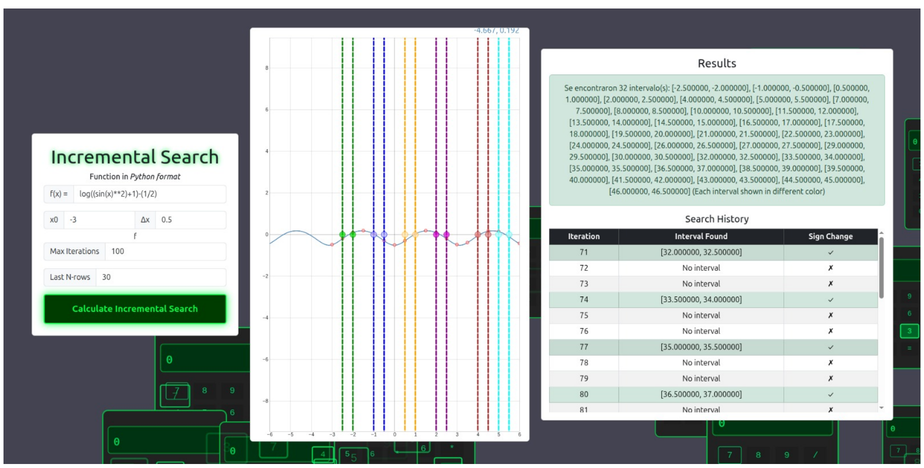


Figure 1.1: Testing on the page

1.1.2 Bisection

The bisection method is an iterative procedure to approximate real roots of a continuous function. Let $f : [a, b] \to \mathbb{R}$ be a continuous function on $[a, b]$, and suppose that

$$f(a) \cdot f(b) < 0,$$

then, by the *Intermediate Value Theorem*, there exists at least one root $r \in (a, b)$.

At each iteration, the midpoint is computed as

$$m = \frac{a+b}{2},$$

and the sign of $f(m)$ is evaluated. If $f(a) \cdot f(m) < 0$, then set $b \leftarrow m$; otherwise, set $a \leftarrow m$. Thus, the interval $[a,b]$ is halved at each step, ensuring it always contains a root.

The process continues until the absolute error $|x_k - x_{k-1}|$ is smaller than a given tolerance $\varepsilon > 0$, or until the maximum number of iterations $n_{\max}$ is reached.

If the initial interval is $[a,b]$, then after n iterations the absolute error satisfies

$$E_n \leq \frac{b-a}{2^n}.$$

This shows that the method converges linearly, with a convergence factor of $\frac{1}{2}$.

Pseudocode

Algorithm 2 Bisection Method

Require: Function $f(x)$, interval $[a,b]$, maximum iterations $n_{\max}$, tolerance ε
Ensure: Approximate root of $f(x) = 0$

$x_0 \leftarrow a$
for $i \leftarrow 1$ to $n_{\max}$ **do**
    $m \leftarrow \frac{a+b}{2}$ ▷ Midpoint
    $E \leftarrow |x_0 - m|$ ▷ Absolute error
    if $f(a) \cdot f(m) < 0$ **then**
        $b \leftarrow m$
    else
        $a \leftarrow m$
    end if
    Save i, a, b, m, E
    if $E < \varepsilon$ **then**
        return m, "Converged"
    end if
    $x_0 \leftarrow m$
end for
return m, "Max iterations reached"

Testing

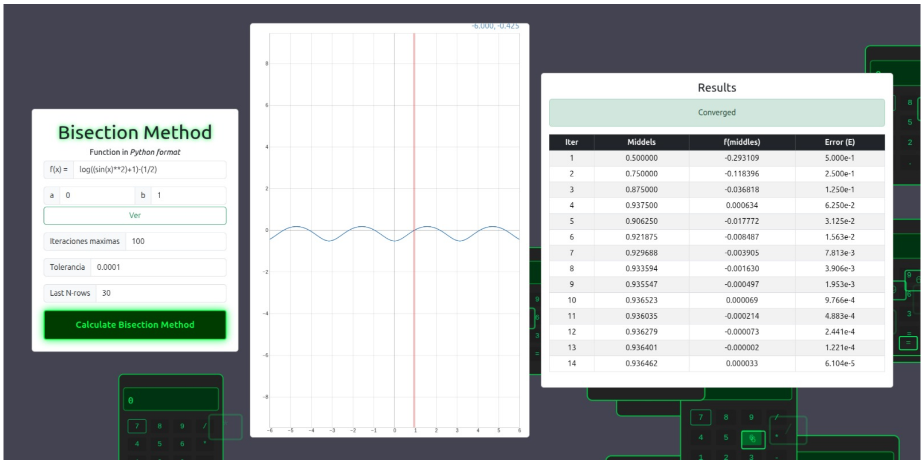


Figure 1.2: Testing on the page

1.1.3 False Position

Also known as the Regula Falsi method, it is a root-finding algorithm that starts with an interval $[a, b]$ such that $f(a) \cdot f(b) < 0$. Instead of taking the midpoint (as in the bisection method), it computes the point of intersection of the secant line between $(a, f(a))$ and $(b, f(b))$ with the x-axis:

$$x = \frac{af(b) - bf(a)}{f(b) - f(a)}.$$

The interval is then updated depending on the sign of $f(x)$, and the process is repeated until the root is approximated within a desired tolerance.

Algorithm 3 False Position Method

Require: Function $f(x)$, interval $[a,b]$, maximum iterations $n_{\max}$, tolerance ε
Ensure: Approximate root x^*, status message

if $f(a) \cdot f(b) > 0$ **then**
  return {"final_root": None, "message": "Function does not change sign on $[a,b]$"}
end if
$x_0 \leftarrow a$
for $i = 1$ to $n_{\max}$ **do**
  Compute $f(a)$ and $f(b)$
  $x_r \leftarrow b - f(b)\dfrac{b-a}{f(b)-f(a)}$
  Compute $f(x_r)$ and error $E = |x_r - x_0|$
  if $E < \varepsilon$ **or** $f(x_r) = 0$ **then**
    return {"final_root": x_r, "message": "Converged"}
  end if
  if $f(a) \cdot f(x_r) < 0$ **then**
    $b \leftarrow x_r$
  else
    $a \leftarrow x_r$
  end if
  $x_0 \leftarrow x_r$
end for
return {"final_root": x_r, "message": "Max iterations reached"}

Pseudocode

Testing

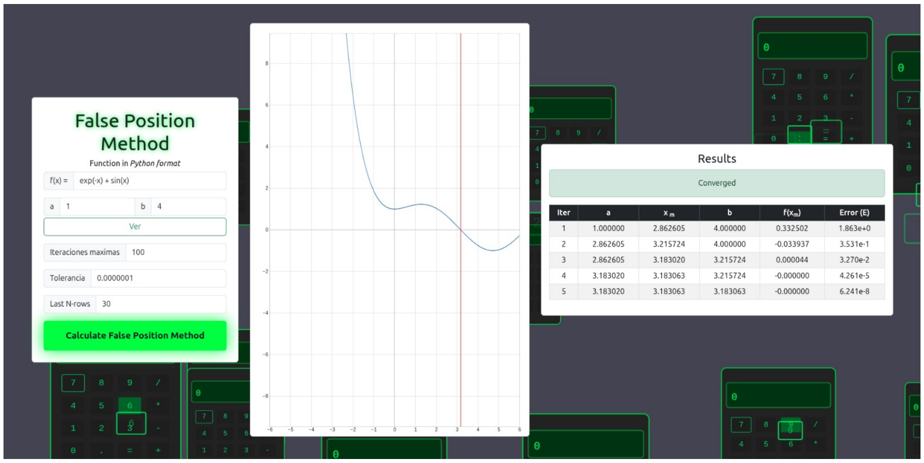


Figure 1.3: Enter Caption

1.1.4 Fixed Point

This root-finding technique rewrites the equation $f(x) = 0$ in the form $x = g(x)$. Starting from an initial guess x_0, the method generates a sequence defined by

$$x_{k+1} = g(x_k), \quad k = 0, 1, 2, \ldots$$

If $g(x)$ satisfies certain convergence conditions (e.g., $|g'(x)| < 1$ near the root), the sequence converges to the fixed point x^*, which is also a solution of $f(x) = 0$.

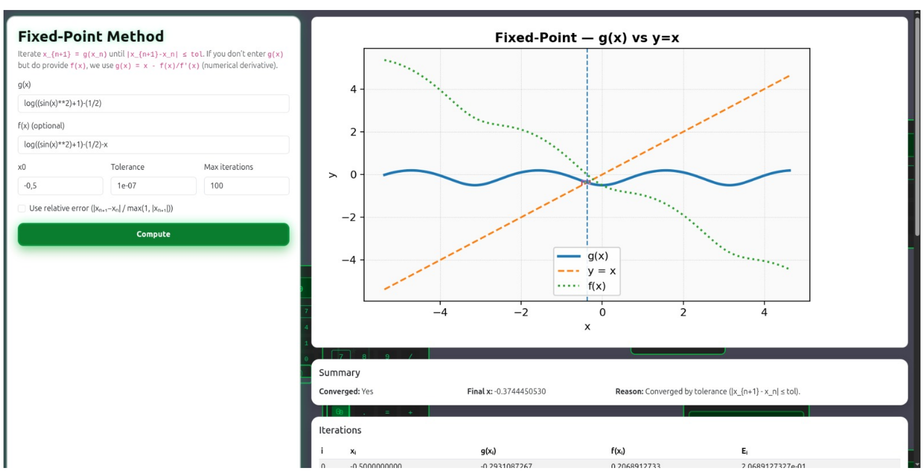
Pseudocode

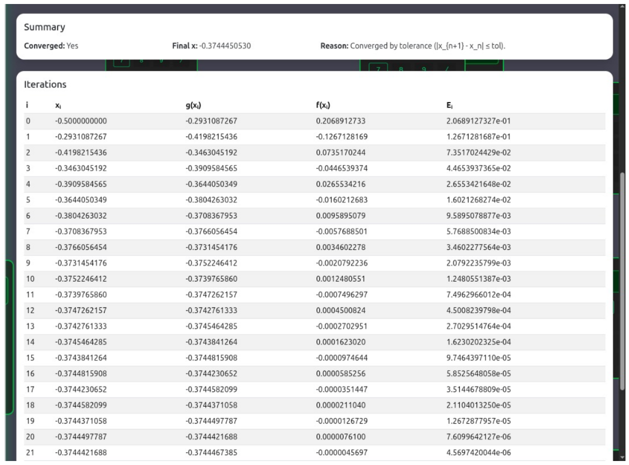
Algorithm 4 Fixed-Point Method

Require: Initial guess x_0, function $g(x)$, tolerance ε, maximum iterations $n_{\max}$
Ensure: Approximate root of $f(x) = 0$

for $i \leftarrow 0$ to $n_{\max} - 1$ **do**
    $x_{i+1} \leftarrow g(x_i)$
    if using relative error **then**
        $E \leftarrow \dfrac{|x_{i+1} - x_i|}{\max(1, |x_{i+1}|)}$
    else
        $E \leftarrow |x_{i+1} - x_i|$
    end if
    Save iteration data $(i, x_i, g(x_i), f(x_i), E)$
    if $E \leq \varepsilon$ **then**
        return x_{i+1}, "Converged"
    end if
end for
return $x_{n_{\max}}$, "Max iterations reached"

Testing





Summary

Converged: Yes **Final x:** -0.3744450530 **Reason:** Converged by tolerance (|x_{n+1} - x_n| ≤ tol).

Iterations

i	x_i	$g(x_i)$	$f(x_i)$	E_i
0	-0.5000000000	-0.2931087267	0.2068912733	2.0689127327e-01
1	-0.2931087267	-0.4198215436	-0.1267128169	1.2671281687e-01
2	-0.4198215436	-0.3463045192	0.0735170244	7.3517024429e-02
3	-0.3463045192	-0.3909584565	-0.0446539374	4.4653937365e-02
4	-0.3909584565	-0.3644050349	0.0265534216	2.6553421648e-02
5	-0.3644050349	-0.3804263032	-0.0160212683	1.6021268274e-02
6	-0.3804263032	-0.3708367953	0.0095895079	9.5895078877e-03
7	-0.3708367953	-0.3766056454	-0.0057688501	5.7688500834e-03
8	-0.3766056454	-0.3731454176	0.0034602278	3.4602277564e-03
9	-0.3731454176	-0.3752246412	-0.0020792236	2.0792235799e-03
10	-0.3752246412	-0.3739765860	0.0012480551	1.2480551387e-03
11	-0.3739765860	-0.3747262157	-0.0007496297	7.4962966012e-04
12	-0.3747262157	-0.3742761333	0.0004500824	4.5008239798e-04
13	-0.3742761333	-0.3745464285	-0.0002702951	2.7029514764e-04
14	-0.3745464285	-0.3743841264	0.0001623020	1.6230202325e-04
15	-0.3743841264	-0.3744815908	-0.0000974644	9.7464397110e-05
16	-0.3744815908	-0.3744230652	0.0000585256	5.8525648058e-05
17	-0.3744230652	-0.3744582099	-0.0000351447	3.5144678809e-05
18	-0.3744582099	-0.3744371058	0.0000211040	2.1104013250e-05
19	-0.3744371058	-0.3744497787	-0.0000126729	1.2672877957e-05
20	-0.3744497787	-0.3744421688	0.0000076100	7.6099642127e-06
21	-0.3744421688	-0.3744467385	-0.0000045697	4.5697420044e-06

Figure 1.4: Testing on a page

1.1.5 Newton Method

Also called the Newton–Raphson method, it is an iterative root-finding algorithm. Starting from an initial approximation x_0, the method uses the tangent line at the point $(x_k, f(x_k))$ to compute a better approximation:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, 2, \ldots$$

Under suitable conditions (when f is differentiable and $f'(x^*) \neq 0$), the sequence $\{x_k\}$ converges quadratically to the root x^*.

Pseudocode

Algorithm 5 Newton Method

Require: Function $f(x)$, initial guess x_0, tolerance ε, maximum iterations $N_{\max}$
Ensure: Approximate root of $f(x) = 0$

for $n \leftarrow 0$ to $N_{\max} - 1$ **do**
  if $f'(x_n) = 0$ **then**
    return "Error: Division by zero"
  end if
  $x_{n+1} \leftarrow x_n - \dfrac{f(x_n)}{f'(x_n)}$
  $E \leftarrow |x_{n+1} - x_n|$ ▷ Absolute error
  Save (n, x_n, E)
  if $E < \varepsilon$ **then**
    return x_{n+1}, "Tolerance satisfied"
  end if
  $x_n \leftarrow x_{n+1}$
end for
return $x_{N_{\max}}$, "Maximum iterations exceeded"

Testing

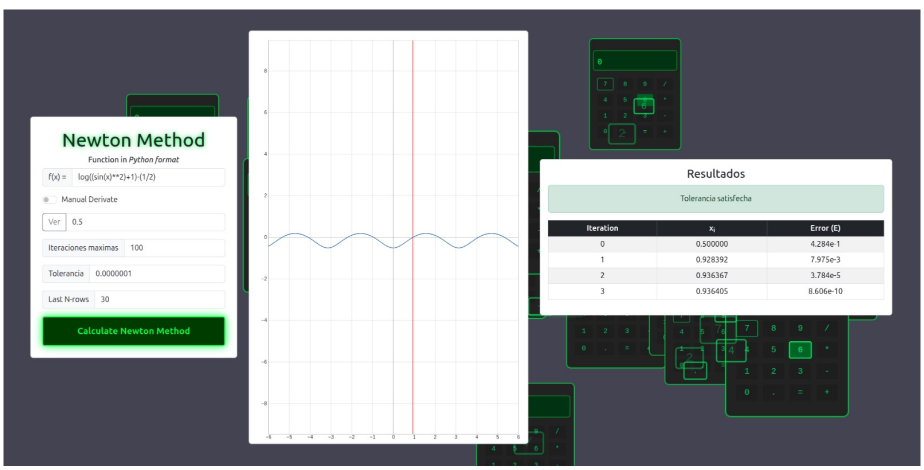


Figure 1.5: Testing on a page

1.1.6 Secant Method

This root-finding method is similar to Newton's method but does not require the derivative of $f(x)$. Instead, it approximates the derivative by using two initial guesses x_0 and

x_1. The iterative formula is:

$$x_{k+1} = x_k - f(x_k)\,\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}, \quad k = 1, 2, \ldots$$

The method generally converges faster than the bisection method, though its convergence is superlinear (slower than Newton's quadratic convergence).

Pseudocode

Algorithm 6 Secant Method

Require: Function $f(x)$, initial guesses x_0, x_1, tolerance ε, maximum iterations $N_{\max}$
Ensure: Approximate root of $f(x) = 0$

```
for n ← 0 to N_max − 1 do
    if f(x_1) − f(x_0) = 0 then
        return "Error: Division by zero"
    end if
    x_2 ← x_1 − f(x_1) (x_1 − x_0) / (f(x_1) − f(x_0))
    E ← |x_2 − x_1|                          ▷ Absolute error
    Save (n, x_1, f(x_1), E)
    if E < ε then
        return x_2, "Tolerance satisfied"
    end if
    x_0 ← x_1, x_1 ← x_2
end for
return x_{N_max}, "Maximum iterations exceeded"
```

Testing

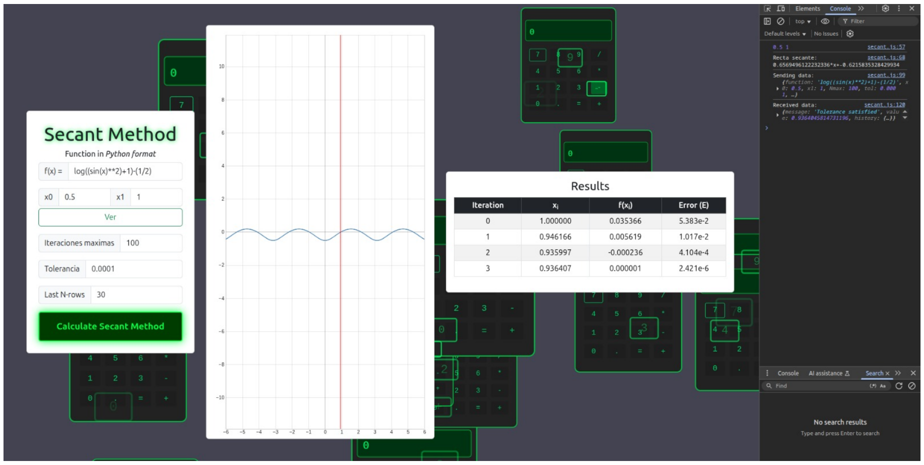


Figure 1.6: Testing on a page

1.1.7 Multiple Roots

This method is a modification of the classical Newton's method designed to find roots of a function $f(x)$ that have multiplicity greater than one. Standard Newton's method converges slowly for multiple roots, so this approach improves convergence. The iterative formula is:

$$x_{n+1} = x_n - \frac{f(x_n)f'(x_n)}{[f'(x_n)]^2 - f(x_n)f''(x_n)},$$

where $f'(x)$ and $f''(x)$ are the first and second derivatives of f. Starting from an initial guess x_0, the method iterates until the absolute difference $|x_{n+1} - x_n|$ is below a given tolerance or the maximum number of iterations is reached. This method achieves faster convergence for multiple roots compared to standard Newton's method.

Pseudocode

Algorithm 7 Multiple Roots

Require: Function $f(x)$, initial guess x_0, tolerance ε, maximum iterations $N_{\max}$
Ensure: Approximate root x^* or failure message

```
Define f'(x) and f''(x) (use given derivatives or compute symbolically)
for n = 0 to N_max − 1 do
    Compute denominator denom ← f'(x0)^2 − f(x0) · f''(x0)
    if denom = 0 then
        return "Denominator zero, method fails"
    end if
    Update x1 ← x0 − (f(x0)·f'(x0))/denom
    Compute absolute error E ← |x1 − x0|
    if E < ε then
        return x1 as approximate root
    end if
    x0 ← x1
end for
return x1 with message "Maximum iterations reached"
```

Testing

1.2 Solution of linear system equations

1.2.1 Gaussian Elimination

It is a direct method to solve a system of linear equations $A\mathbf{x} = \mathbf{b}$. The algorithm transforms the coefficient matrix A into an upper triangular form by applying elementary row operations (without pivoting). Once in triangular form, the solution is obtained through back-substitution.

Pseudocode

Algorithm 8 Gaussian Elimination (Simple)

Require: Matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$
Ensure: Solution vector x or failure message

Compute $\det(A)$
if $\det(A) \approx 0$ **then**
    return {"solution": None, "message": "Solutions can be unstable by higher divisions"}
end if
for $k = 0$ to $n-2$ **do**
    if $A[k,k] = 0$ **then**
        return {"solution": None, "message": "Zero pivot encountered"}
    end if
    for $i = k+1$ to $n-1$ **do**
        if $A[i,k] \neq 0$ **then**
            $m \leftarrow A[i,k]/A[k,k]$
            $A[i,k:n] \leftarrow A[i,k:n] - m \cdot A[k,k:n]$
            $b[i] \leftarrow b[i] - m \cdot b[k]$
        end if
    end for
end for
Initialize $x \leftarrow \mathbf{0} \in \mathbb{R}^n$
for $i = n-1$ downto 0 **do**
    if $A[i,i] = 0$ **then**
        return {"solution": None, "message": "Zero pivot in back substitution"}
    end if
    $x[i] \leftarrow \dfrac{b[i] - \sum_{j=i+1}^{n-1} A[i,j] \cdot x[j]}{A[i,i]}$
end for
return {"solution": x, "message": "Gaussian elimination completed successfully"}

Testing

Elimination Steps

Initial - Initial system. Determinant = 2286.0000

x1	x2	x3	x4	b
2.000000	-1.000000	0.000000	3.000000	1.000000
1.000000	0.500000	3.000000	8.000000	1.000000
0.000000	13.000000	-2.000000	11.000000	1.000000
14.000000	5.000000	-2.000000	3.000000	1.000000

Iteration 1 - Elimination at column 1 complete.

x1	x2	x3	x4	b
2.000000	-1.000000	0.000000	3.000000	1.000000
0.000000	1.000000	3.000000	6.500000	0.500000
0.000000	13.000000	-2.000000	11.000000	1.000000
0.000000	12.000000	-2.000000	-18.000000	-6.000000

Figure 1.7: Testing on the page

Iteration 2 - Elimination at column 2 complete.

x1	x2	x3	x4	b
2.000000	-1.000000	0.000000	3.000000	1.000000
0.000000	1.000000	3.000000	6.500000	0.500000
0.000000	0.000000	-41.000000	-73.500000	-5.500000
0.000000	0.000000	-38.000000	-96.000000	-12.000000

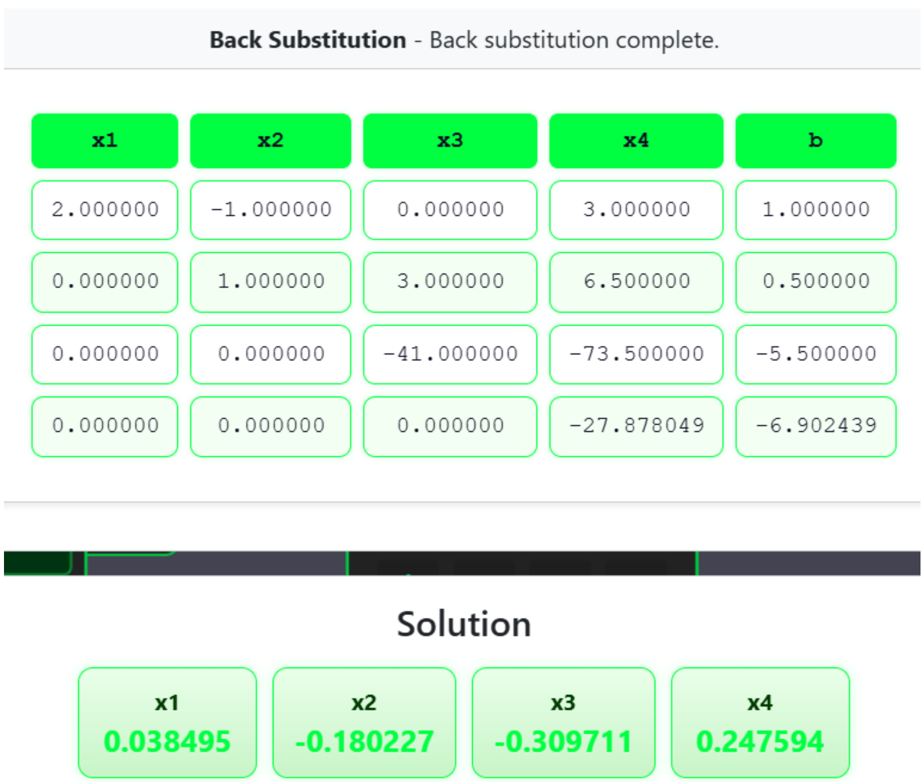
Iteration 3 - Elimination at column 3 complete.

x1	x2	x3	x4	b
2.000000	-1.000000	0.000000	3.000000	1.000000
0.000000	1.000000	3.000000	6.500000	0.500000
0.000000	0.000000	-41.000000	-73.500000	-5.500000
0.000000	0.000000	0.000000	-27.878049	-6.902439

Figure 1.8: Testing on the page

1.2.2 Gaussian Elimination with Partial Pivoting

This method improves numerical stability in solving a system $A\mathbf{x} = \mathbf{b}$. At each elimination step, the algorithm selects the row with the largest absolute pivot element in the current column and swaps it with the current row. Then, elementary row operations are applied to form an upper triangular system, which is solved by back-substitution.



Back Substitution - Back substitution complete.

x1	x2	x3	x4	b
2.000000	-1.000000	0.000000	3.000000	1.000000
0.000000	1.000000	3.000000	6.500000	0.500000
0.000000	0.000000	-41.000000	-73.500000	-5.500000
0.000000	0.000000	0.000000	-27.878049	-6.902439

Solution

x1	x2	x3	x4
0.038495	-0.180227	-0.309711	0.247594

Figure 1.9: Testing on the page

Pseudocode

Algorithm 9 Gaussian Elimination with Partial Pivoting

Require: Matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$
Ensure: Solution vector x or failure message

Compute $\det(A)$
if $\det(A) \approx 0$ **then**
  return {"solution": None, "message": "Solutions can be unstable by higher divisions"}
end if
for $k = 0$ to $n-2$ **do**
  $max\_row \leftarrow$ index of row with maximum $|A[i,k]|$ for $i = k..n-1$
  if $A[max\_row, k] = 0$ **then**
    return {"solution": None, "message": "All pivots in column $k+1$ are zero"}
  end if
  if $max\_row \neq k$ **then**
    Swap row k with row max_row in A and b
  end if
  for $i = k+1$ to $n-1$ **do**
    if $A[i,k] \neq 0$ **then**
      $m \leftarrow A[i,k]/A[k,k]$
      $A[i,k:n] \leftarrow A[i,k:n] - m \cdot A[k,k:n]$
      $b[i] \leftarrow b[i] - m \cdot b[k]$
    end if
  end for
end for
Initialize $x \leftarrow \mathbf{0} \in \mathbb{R}^n$
for $i = n-1$ downto 0 **do**
  if $A[i,i] = 0$ **then**
    return {"solution": None, "message": "Zero pivot in back substitution"}
  end if
  $x[i] \leftarrow \dfrac{b[i] - \sum_{j=i+1}^{n-1} A[i,j] \cdot x[j]}{A[i,i]}$
end for
return {"solution": x, "message": "Gaussian elimination with partial pivoting completed successfully"}

Testing

1.2.3 Gaussian Elimination with Total Pivoting

This variant of Gaussian elimination further increases numerical stability. At each step, the algorithm searches for the largest absolute element in the submatrix (rows and columns not yet eliminated), then swaps both rows and columns so that this element becomes the pivot. Afterward, elementary row operations are applied to form an upper triangular system, which is solved using back-substitution. Column swaps must also be tracked to correctly reorder the solution vector.

Pseudocode

Algorithm 10 Gaussian Elimination with Total Pivoting

Require: Matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$
Ensure: Solution vector x or failure message

Initialize $col\_order \leftarrow [0, 1, \ldots, n-1]$
Compute $\det(A)$
if $\det(A) \approx 0$ **then**
    return {"solution": None, "message": "Solutions can be unstable by higher divisions"}
end if
for $k = 0$ to $n-2$ **do**
    Find (max_row, max_col) = indices of largest $|A[i, j]|$ in submatrix $A[k : n-1, k : n-1]$
    if $A[max\_row, max\_col] = 0$ **then**
        return {"solution": None, "message": "All pivots in submatrix are zero"}
    end if
    if $max\_row \neq k$ **then**
        Swap row k with row max_row in A and b
    end if
    if $max\_col \neq k$ **then**
        Swap column k with column max_col in A
        Swap $col\_order[k]$ with $col\_order[max\_col]$
    end if
    for $i = k+1$ to $n-1$ **do**
        if $A[i,k] \neq 0$ **then**
            $m \leftarrow A[i,k]/A[k,k]$
            $A[i,k:n] \leftarrow A[i,k:n] - m \cdot A[k,k:n]$
            $b[i] \leftarrow b[i] - m \cdot b[k]$
        end if
    end for
end for
Initialize $x \leftarrow \mathbf{0} \in \mathbb{R}^n$
for $i = n-1$ downto 0 **do**
    if $A[i,i] = 0$ **then**
        return {"solution": None, "message": "Zero pivot in back substitution"}
    end if
    $x[i] \leftarrow \dfrac{b[i] - \sum_{j=i+1}^{n-1} A[i, j] \cdot x[j]}{A[i,i]}$
end for
Reorder x according to col_order to get x_final
return {"solution": x_final, "message": "Gaussian elimination with total pivoting completed successfully"}

Testing

1.2.4 LU with simple pivot

Pseudocode

Algorithm 11 LU Factorization without Pivoting

Require: Matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$
Ensure: Solution vector x satisfying $Ax = b$

```
Initialize L ← I_n                                  ▷ Identity matrix of size n
Initialize U ← A
                                    ▷ Step 1: LU Factorization (no pivoting)
for k = 0 to n − 2 do
    pivot ← U[k,k]
    if pivot = 0 then
        Stop: Zero pivot encountered (method fails)
    end if
    for i = k + 1 to n − 1 do
        L[i,k] ← U[i,k]/pivot
        for j = k to n − 1 do
            U[i, j] ← U[i, j] − L[i,k] · U[k, j]
        end for
        U[i,k] ← 0                                  ▷ Clean lower part explicitly
    end for
end for
                                    ▷ Step 2: Forward Substitution (Ly = b)
for i = 0 to n − 1 do
    sum ← 0
    for j = 0 to i − 1 do
        sum ← sum + L[i, j] · y[j]
    end for
    y[i] ← b[i] − sum
end for
                                    ▷ Step 3: Backward Substitution (Ux = y)
for i = n − 1 down to 0 do
    sum ← 0
    for j = i + 1 to n − 1 do
        sum ← sum + U[i, j] · x[j]
    end for
    x[i] ← (y[i] − sum)/U[i,i]
end for
return x
```

Testing

1.2.5 LU with partial pivot

Pseudocode

Algorithm 12 LU with Partial Pivoting

Require: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$
Ensure: x and factors P, L, U with $PA = LU$

$P \leftarrow I_n$, $L \leftarrow 0$, $U \leftarrow A$
for $k = 0$ **to** $n-2$ **do**
  $p \leftarrow \arg\max_{i\in\{k,\ldots,n-1\}} |U_{i,k}|$ ▷ partial pivot
  if $p \neq k$ **then**
    swap rows $k \leftrightarrow p$ in U and P
    swap rows $k \leftrightarrow p$ in L **for columns** $0{:}k-1$
  end if
  if $U_{k,k} = 0$ **then**
    return failure (singular)
  end if
  for $i = k+1$ **to** $n-1$ **do**
    $L_{i,k} \leftarrow U_{i,k}/U_{k,k}$
    $U_{i,k:n} \leftarrow U_{i,k:n} - L_{i,k}U_{k,k:n}$
  end for
end for
for $i = 0$ **to** $n-1$ **do**
  $L_{i,i} \leftarrow 1$
end for
Solve $Ly = Pb$; then $Ux = y$
return x, P, L, U

Testing

1.2.6 Crout

Pseudocode

Algorithm 13 Crout LU Factorization Method (Numerical Core)

Require: Square matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$
Ensure: Solution vector x to $Ax = b$

Initialize $L \leftarrow 0_{n\times n}$ ▷ Zero matrix
Initialize $U \leftarrow I_{n\times n}$ ▷ Identity matrix
for $j = 0$ **to** $n-1$ **do**
  for $i = j$ **to** $n-1$ **do**
    $L_{i,j} \leftarrow A_{i,j} - \sum_{k=0}^{j-1} L_{i,k} \cdot U_{k,j}$
  end for
  for $i = j+1$ **to** $n-1$ **do**
    $U_{j,i} \leftarrow \dfrac{A_{j,i} - \sum_{k=0}^{j-1} L_{j,k} \cdot U_{k,i}}{L_{j,j}}$
  end for
end for
Forward substitution: solve $Ly = b$
for $i = 0$ **to** $n-1$ **do**
  $y_i \leftarrow b_i - \sum_{k=0}^{i-1} L_{i,k} \cdot y_k$
end for
Backward substitution: solve $Ux = y$
for $i = n-1$ **down to** 0 **do**
  $x_i \leftarrow y_i - \sum_{k=i+1}^{n-1} U_{i,k} \cdot x_k$
end for
return x

Testing

1.2.7 Doolittle

Pseudocode

Algorithm 14 Doolittle Factorization Method

Require: Square matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$
Ensure: Solution vector x to $Ax = b$

Initialize $L \leftarrow I_{n\times n}$ ▷ Identity matrix
Initialize $U \leftarrow 0_{n\times n}$ ▷ Zero matrix
for $j = 0$ **to** $n-1$ **do**
  for $k = j$ **to** $n-1$ **do**
    $U_{j,k} \leftarrow A_{j,k} - \sum_{m=0}^{j-1} L_{j,m} \cdot U_{m,k}$
  end for
  for $i = j+1$ **to** $n-1$ **do**
    $L_{i,j} \leftarrow \dfrac{A_{i,j} - \sum_{m=0}^{j-1} L_{i,m} \cdot U_{m,j}}{U_{j,j}}$
  end for
end for
Forward substitution: solve $Ly = b$
for $i = 0$ **to** $n-1$ **do**
  $y_i \leftarrow b_i - \sum_{j=0}^{i-1} L_{i,j} \cdot y_j$
end for
Backward substitution: solve $Ux = y$
for $i = n-1$ **down to** 0 **do**
  $x_i \leftarrow \dfrac{y_i - \sum_{j=i+1}^{n-1} U_{i,j} \cdot x_j}{U_{i,i}}$
end for
return x

Testing

1.2.8 Cholesky

Pseudocode

Algorithm 15 Cholesky Decomposition and Solve

Require: SPD $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$
Ensure: L, $L^\top$, y, x

```
for k = 0 to n − 1 do
    t ← A_{k,k} − Σ_{s=0}^{k−1} L_{k,s}^2
    if t ≤ 0 then
        return failure (matrix not SPD)
    end if
    L_{k,k} ← √t
    for i = k + 1 to n − 1 do
        L_{i,k} ← (A_{i,k} − Σ_{s=0}^{k−1} L_{i,s} L_{k,s}) / L_{k,k}
    end for
end for
Solve Ly = b (forward); then L^⊤ x = y (back)
return L, L^⊤, y, x
```

Testing

1.2.9 Jacobi

Pseudocode

Testing

1.2.10 Gauss-Seidel

The method is based on decomposing the matrix A into its diagonal component D, lower triangular component L, and upper triangular component U:

$$A = D - L - U$$

Substituting this decomposition into the system, we obtain:

$$(D - L)\mathbf{x} = U\mathbf{x} + \mathbf{b}$$

and therefore:

$$\mathbf{x} = (D - L)^{-1}U\mathbf{x} + (D - L)^{-1}\mathbf{b}$$

The matrix $(D-L)^{-1}U$ is known as the **iteration matrix** T_{GS}, and $(D-L)^{-1}\mathbf{b}$ as the vector $\mathbf{c}$. The convergence of the method depends on the **spectral radius** $\rho(T_{GS})$, which must satisfy:

$$\rho(T_{GS}) < 1$$

for the iterative process to converge.

At each iteration, the new approximation $x_i^{(k+1)}$ is computed using the most recent available values:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

Pseudocode

The following pseudocode outlines the steps of the Gauss-Seidel iterative method, including matrix validation, construction of the iteration matrix, and convergence control.

Algorithm 16 Gauss-Seidel

Require: Matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$, initial approximation $x^{(0)} \in \mathbb{R}^n$, tolerance $\varepsilon > 0$, maximum iterations n_{max}
Ensure: Approximate solution $\mathbf{x}$ within given tolerance

Verify that A is square and that size(b) and size$(x^{(0)})$ match A
Decompose $A = D - L - U$
Compute $(D-L)^{-1}$, iteration matrix $T_{GS} = (D-L)^{-1}U$ and vector $\mathbf{c} = (D-L)^{-1}\mathbf{b}$
Compute spectral radius $\rho(T_{GS})$ and $\|T_{GS}\|_2$
Initialize $\mathbf{x}^{(0)}$
for $k \leftarrow 1$ to n_{max} **do**
    for $i \leftarrow 1$ to n **do**
        $s_1 \leftarrow \sum_{j=1}^{i-1} a_{ij} x_j^{(k)}$
        $s_2 \leftarrow \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)}$
        $x_i^{(k)} \leftarrow \frac{b_i - s_1 - s_2}{a_{ii}}$
    end for
    Compute error $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2$
    if error $< \varepsilon$ **then**
        return $\mathbf{x}^{(k)}$
    end if
end for
return Solution Object

1.2.11 Successive Over-Relaxation (SOR)

The Successive Over-Relaxation (SOR) method is an extension of the Gauss-Seidel iterative method designed to accelerate convergence for solving the linear system:

$$A\mathbf{x} = \mathbf{b}$$

where $A \in \mathbb{R}^{n\times n}$ is a square, non-singular matrix, $\mathbf{x}$ is the vector of unknowns, and $\mathbf{b}$ is the constant vector.

The method introduces a **relaxation parameter** ω, with $0 < \omega < 2$, that adjusts the correction applied at each iteration to improve convergence speed. When $\omega = 1$, the method reduces to the standard Gauss-Seidel method.

The iterative update for each component $x_i^{(k+1)}$ is defined as:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

Pseudocode

The following algorithm presents the steps of the SOR method for solving $A\mathbf{x} = \mathbf{b}$, using programming-oriented operations instead of symbolic summations.

Algorithm 17 Successive Over-Relaxation (SOR) Method

Require: Matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$, relaxation parameter ω, tolerance $\varepsilon > 0$, maximum iterations n_{max}
Ensure: Approximate solution $\mathbf{x}$ satisfying the tolerance

```
Initialize x^(0) ← 0 if no initial guess is provided
Initialize history arrays for x, absolute error, and iteration count
for k ← 1 to n_max do
    x_old ← x
    for i ← 0 to n − 1 do
        sum1 ← 0
        for j ← 0 to i − 1 do
            sum1 ← sum1 + A[i][j] · x[j]
        end for
        sum2 ← 0
        for j ← i + 1 to n − 1 do
            sum2 ← sum2 + A[i][j] · x_old[j]
        end for
        x[i] ← (1 − ω) · x_old[i] + (ω/A[i][i]) · (b[i] − sum1 − sum2)
    end for
    error ← ||x − x_old||_∞
    Store x, error, and k in history
    if error < ε then
        return x with message "Tolerance satisfied"
    end if
end for
return x with message "Maximum number of iterations reached"
```

1.3 Interpolation

1.3.1 Vandermonde

Pseudocode

Algorithm 18 Vandermonde Interpolation

Require: distinct nodes $x_0, \ldots, x_{n-1}$; values $y_0, \ldots, y_{n-1}$
Ensure: coefficients $a_0, \ldots, a_{n-1}$ and V
1: Build V with $V_{i,j} \leftarrow x_i^j$ (increasing powers)
2: Solve $Va = y$ (e.g., LU with partial pivoting)
3: **return** a and V

Testing

1.3.2 Newton differences

Pseudocode

Testing

1.4 Polynomial Interpolation

Polynomial interpolation is a technique that consists of finding a polynomial of degree n that passes through $n+1$ given data points. This polynomial can then be used to estimate values between the data points (interpolation) or outside them (extrapolation).

1.4.1 Lagrange Method

The Lagrange method constructs the unique polynomial $p(x)$ of degree n that interpolates the $n+1$ points $(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))$. The polynomial is a linear combination of the function values $f(x_k)$ multiplied by the **Lagrange basis polynomials** $L_k(x)$.

The interpolating polynomial is given by:

$$p(x) = \sum_{k=0}^{n} L_k(x) f(x_k)$$

where the basis polynomial $L_k(x)$ is defined as:

$$L_k(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}$$

Pseudocode

The procedure calculates the interpolated value for a specific x based on a set of data points (x_k, y_k).

Algorithm 19 Lagrange Interpolation Method

Require: Vectors $X, Y \in \mathbb{R}^n$ $\triangleright$ X: array of x-coordinates $(x_0, \dots, x_n)$ $\triangleright$ Y: array of y-coordinates $(y_0, \dots, y_n)$

$n \leftarrow \text{length}(X) - 1$
$p_L(x) \leftarrow 0$ $\triangleright$ Initialize Lagrange Polynomial
for $k \leftarrow 0$ to n **do**
    $L_k \leftarrow 1$ $\triangleright$ Initialize Lagrange basis polynomial $L_k(x)$
    for $j \leftarrow 0$ to n **do**
        if $j \neq k$ **then**
            $L_k \leftarrow L_k \cdot \frac{(x - X[j])}{(X[k] - X[j])}$
        end if
    end for
    $p_L(x) \leftarrow p_L(x) + Y[k] \cdot L_k$
end for
return $p_L(x)$ $\triangleright$ Lagrange Polynomial $p_L(x)$ with 15 presicion digits

1.4.2 Linear tracers

Pseudocode

Algorithm 20 Piecewise Linear Spline Construction

Require: strictly increasing $x_0 < \cdots < x_{n-1}$; values $y_0, \dots, y_{n-1}$
Ensure: segments $\{(m_i, b_i, [x_i, x_{i+1}])\}_{i=0}^{n-2}$ and equations

for $i = 0$ **to** $n - 2$ **do**
    $m_i \leftarrow \dfrac{y_{i+1} - y_i}{x_{i+1} - x_i}, \quad b_i \leftarrow y_i - m_i x_i$
    store $S_i(x) = m_i x + b_i$ on $[x_i, x_{i+1}]$
end for
return all (m_i, b_i) and equations

Testing

1.4.3 Quadratic tracers

Pseudocode

Algorithm 21 Natural Quadratic Spline Construction

Require: strictly increasing $x_0 < \cdots < x_n$; values $y_0, \ldots, y_n$
Ensure: quadratic segments $\{(a_i, b_i, c_i, [x_i, x_{i+1}])\}_{i=0}^{n-1}$

compute $h_i \leftarrow x_{i+1} - x_i$ for $i = 0, \ldots, n-1$
set $c_0 \leftarrow 0$
for $i = 1$ **to** $n-1$ **do**
    $c_i \leftarrow \left(\dfrac{y_{i+1} - y_i}{h_i} - \dfrac{y_i - y_{i-1}}{h_{i-1}}\right)\dfrac{h_{i-1}}{h_{i-1} + h_i}$
end for
for $i = 0$ **to** $n-2$ **do**
    $a_i \leftarrow y_i$
    $b_i \leftarrow \dfrac{y_{i+1} - y_i}{h_i} - c_i h_i$
    store $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2$ on $[x_i, x_{i+1}]$
end for
return all (a_i, b_i, c_i) and spline equations

Testing

1.4.4 Cubic tracers

Pseudocode

Algorithm 22 Cubic Spline Construction

Require: strictly increasing $x_0 < \cdots < x_n$; values $y_0, \ldots, y_n$
Ensure: cubic segments $\{(a_i, b_i, c_i, d_i, [x_i, x_{i+1}])\}_{i=0}^{n-1}$

compute $h_i \leftarrow x_{i+1} - x_i$ for $i = 0, \ldots, n-1$
form and solve the tridiagonal system $Ac = b$ with

$$A_{0,0} = A_{n,n} = 1, \quad A_{i,i-1} = h_{i-1},\, A_{i,i} = 2(h_{i-1} + h_i),\, A_{i,i+1} = h_i,$$

$$b_i = 3\left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right), \quad i = 1, \ldots, n-1$$

for $i = 0$ **to** $n-1$ **do**
  $a_i \leftarrow y_i$
  $b_i \leftarrow \dfrac{y_{i+1} - y_i}{h_i} - \dfrac{h_i}{3}(2c_i + c_{i+1})$
  $d_i \leftarrow \dfrac{c_{i+1} - c_i}{3h_i}$
  store $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$ on $[x_i, x_{i+1}]$
end for
return all (a_i, b_i, c_i, d_i) and spline equations

Testing