**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 $\Delta x$ 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

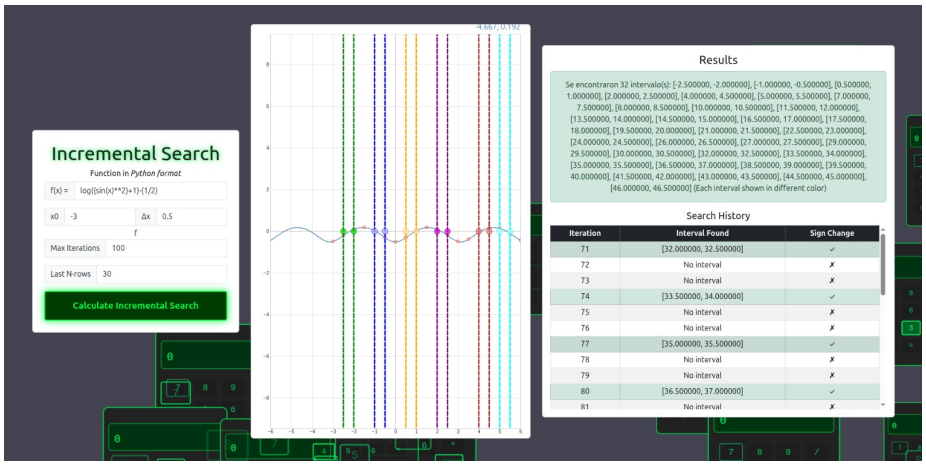



Figure 1.1: Testing on the page

## 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 $\varepsilon$
**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**

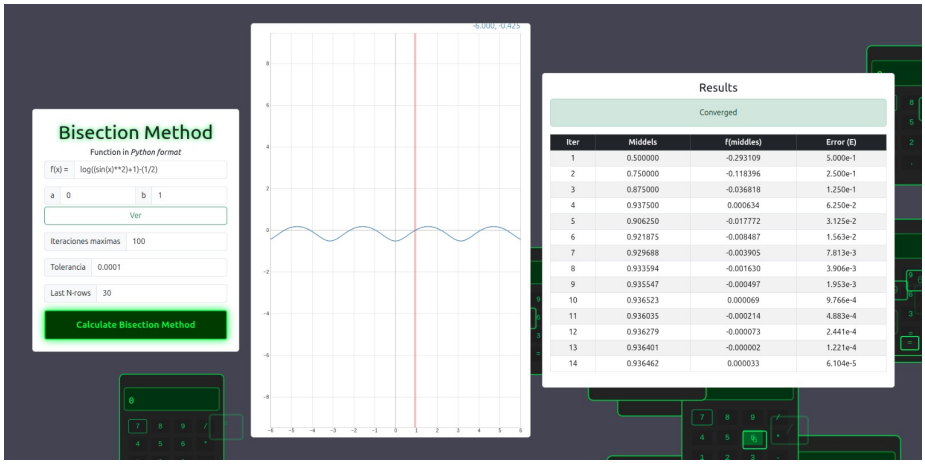



Figure 1.2: Testing on the page

### 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 $\varepsilon$
**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**

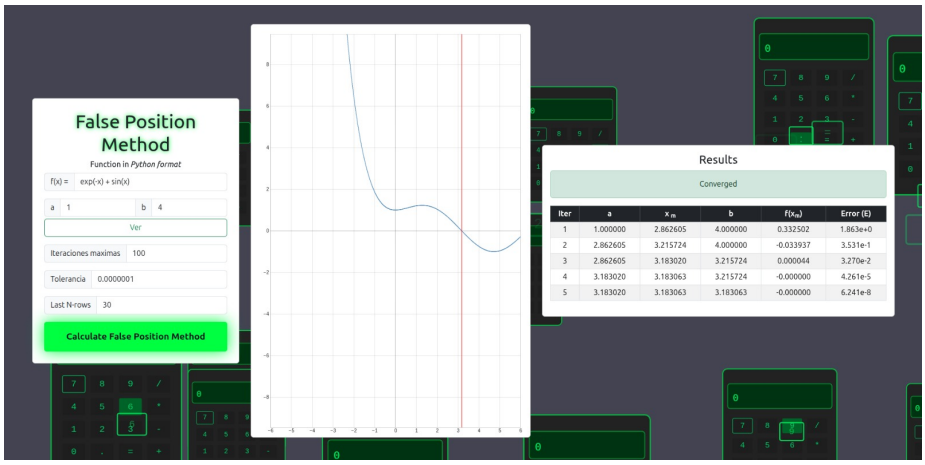



Figure 1.3: Enter Caption

### 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 $f(x) = 0$.

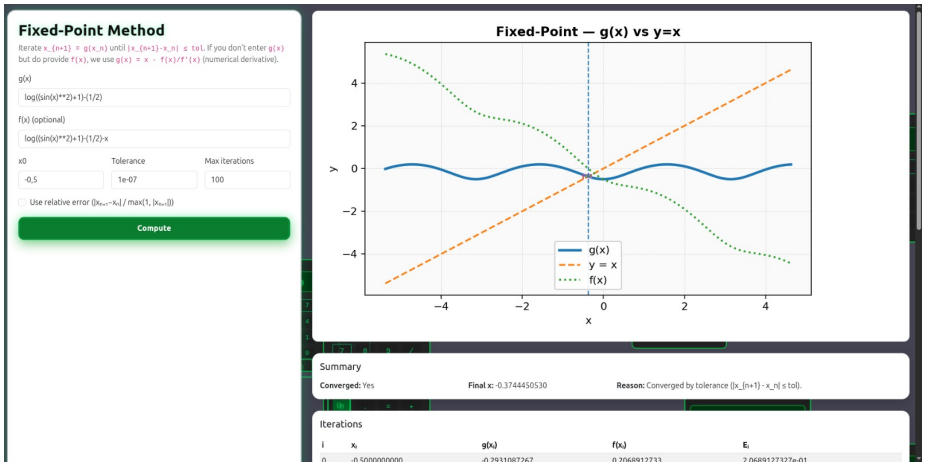
## Pseudocode

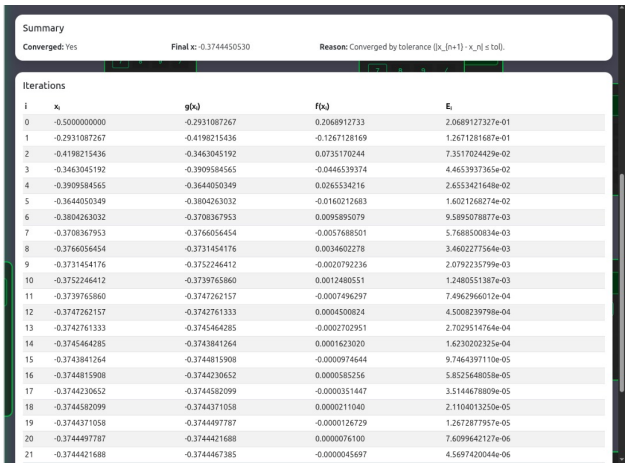
**Algorithm 4** Fixed-Point Method

**Require:** Initial guess $x_0$, function $g(x)$, tolerance $\varepsilon$, 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 ($|x_{n+1} - x_n| \leq$ 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 $\varepsilon$, 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**

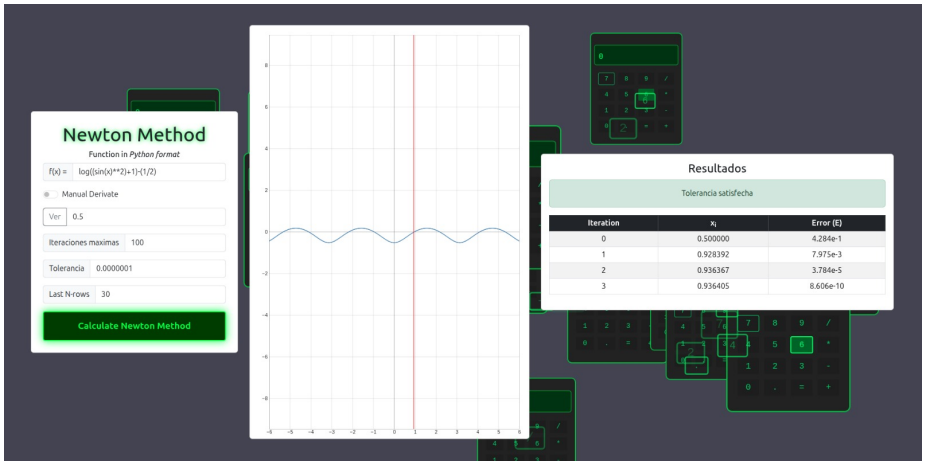



Figure 1.5: Testing on a page

### 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 $\varepsilon$, maximum iterations $N_{\max}$
**Ensure:** Approximate root of $f(x) = 0$

**for** $n \leftarrow 0$ to $N_{\max} - 1$ **do**
  **if** $f(x_1) - f(x_0) = 0$ **then**
    **return** "Error: Division by zero"
  **end if**
  $x_2 \leftarrow x_1 - f(x_1)\dfrac{x_1 - x_0}{f(x_1) - f(x_0)}$
  $E \leftarrow |x_2 - x_1|$ ▷ Absolute error
  Save $(n, x_1, f(x_1), E)$
  **if** $E < \varepsilon$ **then**
    **return** $x_2$, "Tolerance satisfied"
  **end if**
  $x_0 \leftarrow x_1, x_1 \leftarrow x_2$
**end for**
**return** $x_{N_{\max}}$, "Maximum iterations exceeded"

**Testing**

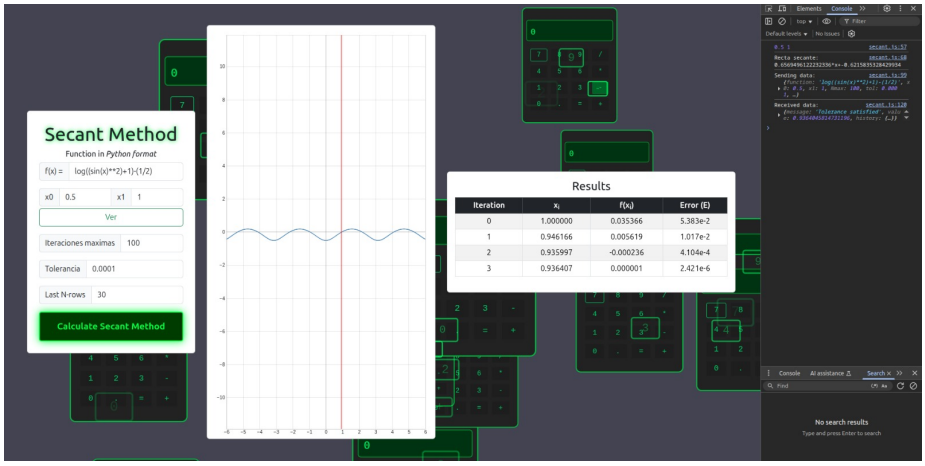



Figure 1.6: Testing on a page

### 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 $\varepsilon$, 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 $denom \leftarrow f'(x_0)^2 - f(x_0) \cdot f''(x_0)$
    **if** $denom = 0$ **then**
        **return** "Denominator zero, method fails"
    **end if**
    Update $x_1 \leftarrow x_0 - \frac{f(x_0) \cdot f'(x_0)}{denom}$
    Compute absolute error $E \leftarrow |x_1 - x_0|$
    **if** $E < \varepsilon$ **then**
        **return** $x_1$ as approximate root
    **end if**
    $x_0 \leftarrow x_1$
**end for**
**return** $x_1$ with message "Maximum iterations reached"

**Testing**

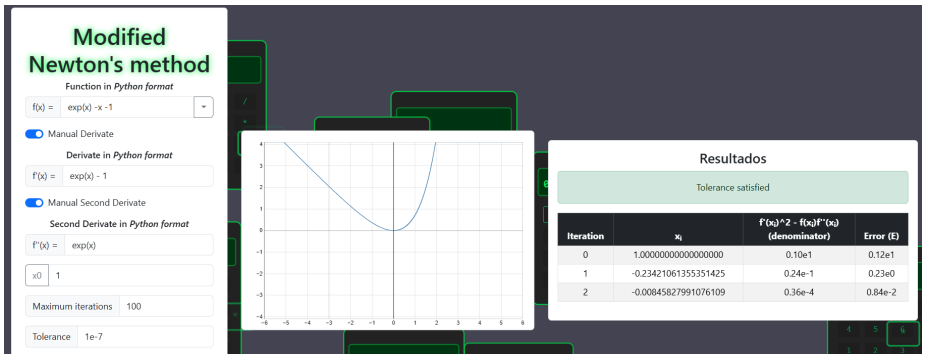



Figure 1.7: Testing on the page

## 1.2 Solution of linear system equations

### 1.2.1 Gaussian Elimination

This method is also known as (Modified Newton's method) 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 1.8: 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 1.9: 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 1.10: 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

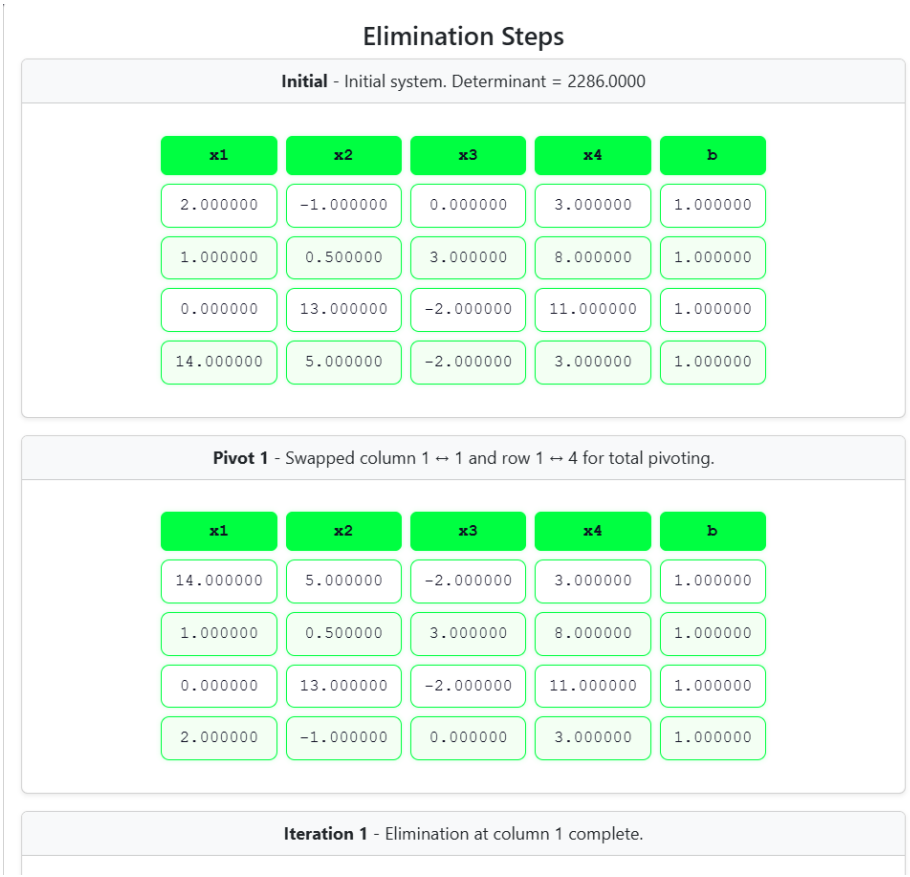



Figure 1.11: Testing on the page

**Iteration 1** - Elimination at column 1 complete.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |
| 0.000000 | 0.142857 | 3.142857 | 7.785714 | 0.928571 |
| 0.000000 | 13.000000 | -2.000000 | 11.000000 | 1.000000 |
| 0.000000 | -1.714286 | 0.285714 | 2.571429 | 0.857143 |

**Pivot 2** - Swapped column 2 ↔ 2 and row 2 ↔ 3 for total pivoting.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |
| 0.000000 | 13.000000 | -2.000000 | 11.000000 | 1.000000 |
| 0.000000 | 0.142857 | 3.142857 | 7.785714 | 0.928571 |
| 0.000000 | -1.714286 | 0.285714 | 2.571429 | 0.857143 |

Figure 1.12: Testing on the page

**Iteration 2** - Elimination at column 2 complete.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |
| 0.000000 | 13.000000 | -2.000000 | 11.000000 | 1.000000 |
| 0.000000 | 0.000000 | 3.164835 | 7.664835 | 0.917582 |
| 0.000000 | 0.000000 | 0.021978 | 4.021978 | 0.989011 |

**Pivot 3** - Swapped column 3 ↔ 4 and row 3 ↔ 3 for total pivoting.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | 3.000000 | -2.000000 | 1.000000 |
| 0.000000 | 13.000000 | 11.000000 | -2.000000 | 1.000000 |
| 0.000000 | 0.000000 | 7.664835 | 3.164835 | 0.917582 |
| 0.000000 | 0.000000 | 4.021978 | 0.021978 | 0.989011 |

Figure 1.13: Testing on the page

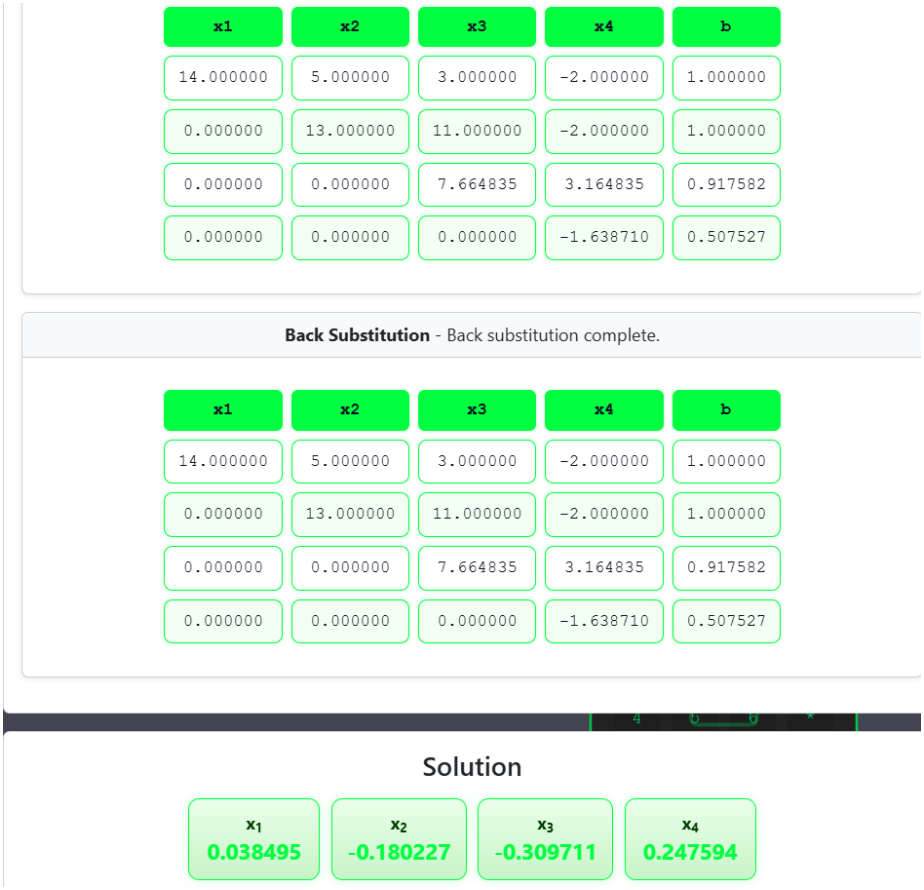



Figure 1.14: Testing on the page

### 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

**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 |

**Pivot 1** - Rows 1 and 4 swapped for partial pivoting.

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

**Iteration 1** - Elimination at column 1 complete.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|

Figure 1.15: Testing on the page

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |
| 0.000000 | 0.142857 | 3.142857 | 7.785714 | 0.928571 |
| 0.000000 | 13.000000 | -2.000000 | 11.000000 | 1.000000 |
| 0.000000 | -1.714286 | 0.285714 | 2.571429 | 0.857143 |

**Pivot 2** - Rows 2 and 3 swapped for partial pivoting.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |
| 0.000000 | 13.000000 | -2.000000 | 11.000000 | 1.000000 |
| 0.000000 | 0.142857 | 3.142857 | 7.785714 | 0.928571 |
| 0.000000 | -1.714286 | 0.285714 | 2.571429 | 0.857143 |

**Iteration 2** - Elimination at column 2 complete.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |

Figure 1.16: Testing on the page

**Iteration 2** - Elimination at column 2 complete.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |
| 0.000000 | 13.000000 | -2.000000 | 11.000000 | 1.000000 |
| 0.000000 | 0.000000 | 3.164835 | 7.664835 | 0.917582 |
| 0.000000 | 0.000000 | 0.021978 | 4.021978 | 0.989011 |

**Iteration 3** - Elimination at column 3 complete.

| x1 | x2 | x3 | x4 | b |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 3.000000 | 1.000000 |
| 0.000000 | 13.000000 | -2.000000 | 11.000000 | 1.000000 |
| 0.000000 | 0.000000 | 3.164835 | 7.664835 | 0.917582 |
| 0.000000 | 0.000000 | 0.000000 | 3.968750 | 0.982639 |

Figure 1.17: Testing on the page

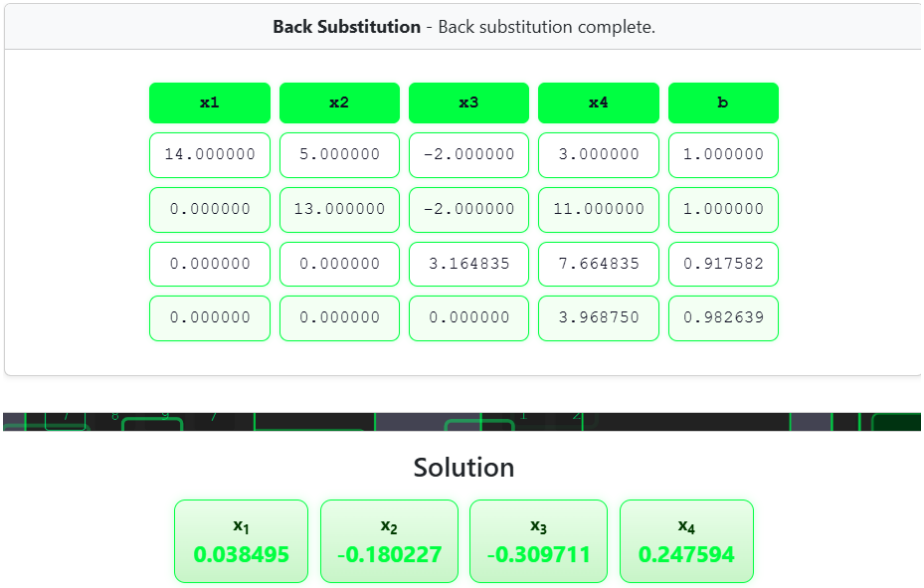



Figure 1.18: Testing on the page

### 1.2.4 LU with simple pivot

The LU decomposition with simple pivoting (without row exchanges) factors a square matrix $A$ into the product $A = LU$, where $L$ is a lower triangular matrix with ones on its diagonal and $U$ is an upper triangular matrix. The method proceeds by using the entries of $A$ to eliminate terms below the main diagonal, storing the multipliers in $L$ and the resulting row operations in $U$. Simple pivoting assumes that all pivot elements $a_{ii}$ are non-zero and sufficiently large, since no row swapping is performed to correct small or zero pivots.

**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 $L \leftarrow I_n$ ▷ Identity matrix of size $n$
Initialize $U \leftarrow A$

▷ **Step 1: LU Factorization (no pivoting)**

**for** $k = 0$ to $n-2$ **do**
    $pivot \leftarrow U[k,k]$
    **if** $pivot = 0$ **then**
        **Stop:** Zero pivot encountered (method fails)
    **end if**
    **for** $i = k+1$ to $n-1$ **do**
        $L[i,k] \leftarrow U[i,k]/pivot$
        **for** $j = k$ to $n-1$ **do**
            $U[i,j] \leftarrow U[i,j] - L[i,k] \cdot U[k,j]$
        **end for**
        $U[i,k] \leftarrow 0$ ▷ Clean lower part explicitly
    **end for**
**end for**

▷ **Step 2: Forward Substitution** ($Ly = b$)

**for** $i = 0$ to $n-1$ **do**
    $sum \leftarrow 0$
    **for** $j = 0$ to $i-1$ **do**
        $sum \leftarrow sum + L[i,j] \cdot y[j]$
    **end for**
    $y[i] \leftarrow b[i] - sum$
**end for**

▷ **Step 3: Backward Substitution** ($Ux = y$)

**for** $i = n-1$ down to $0$ **do**
    $sum \leftarrow 0$
    **for** $j = i+1$ to $n-1$ **do**
        $sum \leftarrow sum + U[i,j] \cdot x[j]$
    **end for**
    $x[i] \leftarrow (y[i] - sum)/U[i,i]$
**end for**
**return** $x$

**Testing**

**Solution**

| | | | |
|---|---|---|---|
| 0.525109 | 0.255459 | -0.410480 | -0.281659 |

**L Matrix**

| | | | |
|---|---|---|---|
| 1.000000 | 0.000000e+0 | 0.000000e+0 | 0.000000e+0 |
| 0.250000 | 1.000000 | 0.000000e+0 | 0.000000e+0 |
| 0.000000e+0 | -0.082540 | 1.000000 | 0.000000e+0 |
| 3.500000 | 0.539683 | 0.964467 | 1.000000 |

**U Matrix**

| | | | |
|---|---|---|---|
| 4.000000 | -1.000000 | 0.000000e+0 | 3.000000 |
| 0.000000e+0 | 15.750000 | 3.000000 | 7.250000 |
| 0.000000e+0 | 0.000000e+0 | -3.752381 | 1.698413 |
| 0.000000e+0 | 0.000000e+0 | 0.000000e+0 | 13.949239 |

Figure 1.19: Testing on the page

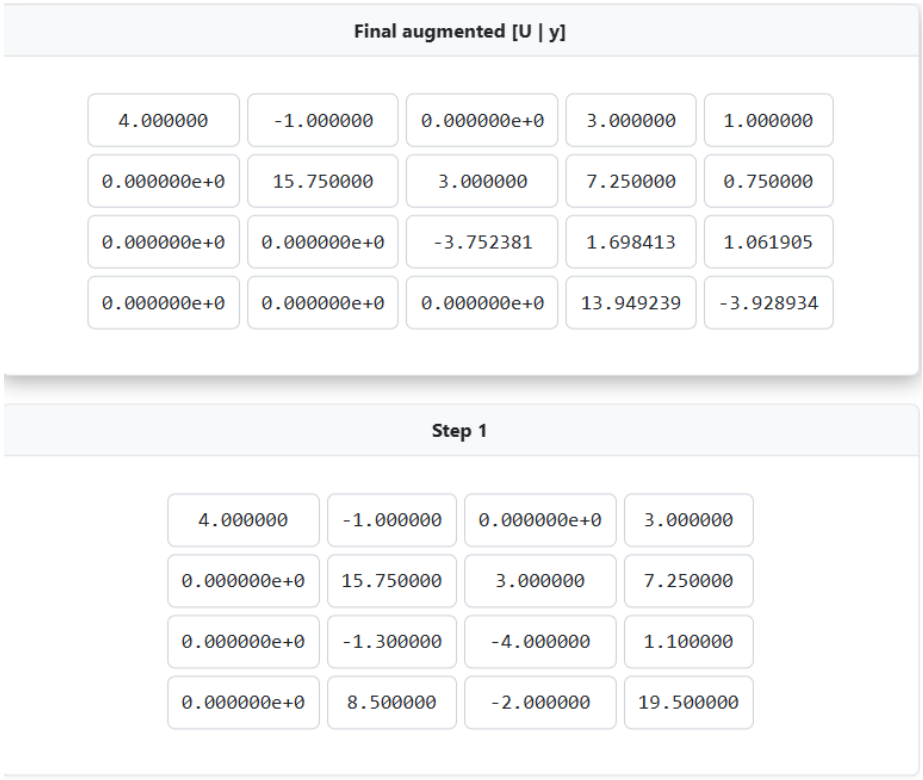



Figure 1.20: Testing on the page


Figure 1.21: Testing on the page

### 1.2.5 LU with partial pivot

The LU decomposition with partial pivoting factors a matrix $A$ into $PA = LU$, where $P$ is a permutation matrix that records row exchanges, $L$ is a lower triangular matrix with ones on the diagonal, and $U$ is an upper triangular matrix. At each step, the algorithm selects the largest available pivot in the current column (by absolute value) and swaps rows to place it on the diagonal. This improves numerical stability and avoids division by small or zero pivots while performing the standard elimination to form $L$ and $U$.

**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**
  $p \leftarrow \arg\max_{i \in \{k,\dots,n-1\}} |U_{i,k}|$ ▷ 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**

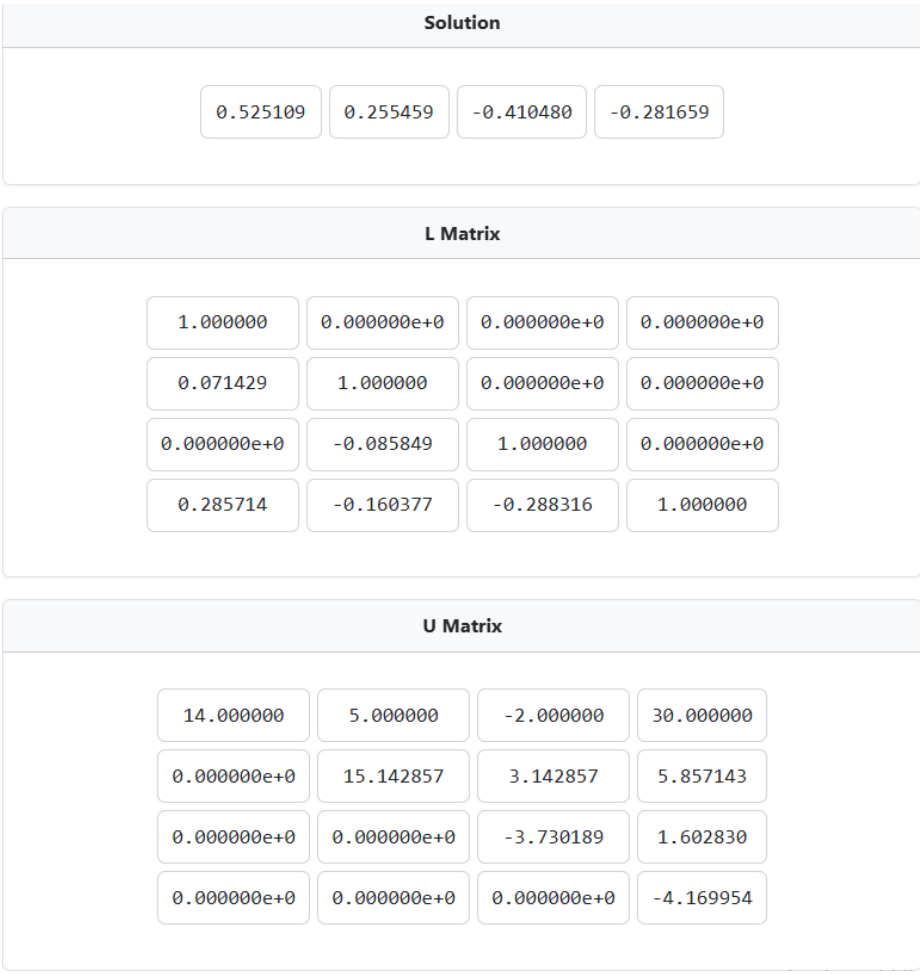



Figure 1.22: Testing on the page

**Permutation Matrix P**

| | | | |
|---|---|---|---|
| 0.000000e+0 | 0.000000e+0 | 0.000000e+0 | 1.000000 |
| 0.000000e+0 | 1.000000 | 0.000000e+0 | 0.000000e+0 |
| 0.000000e+0 | 0.000000e+0 | 1.000000 | 0.000000e+0 |
| 1.000000 | 0.000000e+0 | 0.000000e+0 | 0.000000e+0 |

**Final augmented [U | y]**

| | | | | |
|---|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 30.000000 | 1.000000 |
| 0.000000e+0 | 15.142857 | 3.142857 | 5.857143 | 0.928571 |
| 0.000000e+0 | 0.000000e+0 | -3.730189 | 1.602830 | 1.079717 |
| 0.000000e+0 | 0.000000e+0 | 0.000000e+0 | -4.169954 | 1.174507 |

Figure 1.23: Testing on the page

**Step 1**

| | | | |
|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 30.000000 |
| 0.000000e+0 | 15.142857 | 3.142857 | 5.857143 |
| 0.000000e+0 | -1.300000 | -4.000000 | 1.100000 |
| 0.000000e+0 | -2.428571 | 0.571429 | -5.571429 |

**Step 2**

| | | | |
|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 30.000000 |
| 0.000000e+0 | 15.142857 | 3.142857 | 5.857143 |
| 0.000000e+0 | 0.000000e+0 | -3.730189 | 1.602830 |
| 0.000000e+0 | 0.000000e+0 | 1.075472 | -4.632075 |

Figure 1.24: Testing on the page

**Step 3**

| | | | |
|---|---|---|---|
| 14.000000 | 5.000000 | -2.000000 | 30.000000 |
| 0.000000e+0 | 15.142857 | 3.142857 | 5.857143 |
| 0.000000e+0 | 0.000000e+0 | -3.730189 | 1.602830 |
| 0.000000e+0 | 0.000000e+0 | 0.000000e+0 | -4.169954 |



Figure 1.25: Testing on the page

### 1.2.6 Crout

The Crout method is an LU decomposition technique in which a matrix $A$ is factored as $A = LU$, where $L$ is a lower triangular matrix with general (non-unit) diagonal entries and $U$ is an upper triangular matrix whose diagonal elements are all equal to 1. The algorithm computes the columns of $L$ and the rows of $U$ sequentially, ensuring that each entry of $A$ is matched by the corresponding product of $L$ and $U$. Crout's method is efficient and numerically reliable when properly combined with pivoting.

**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

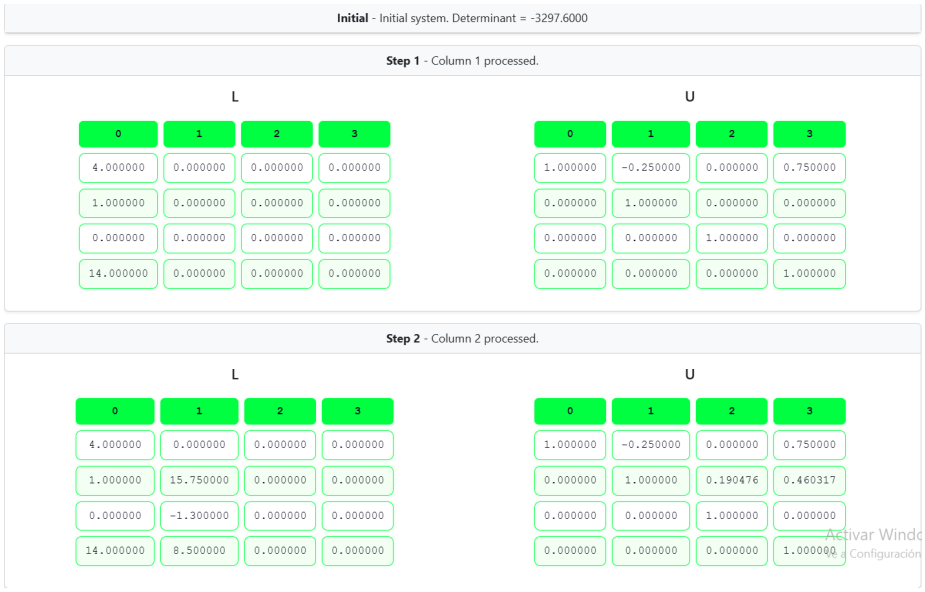



Figure 1.26: Testing on the page


Figure 1.27: Testing on the page

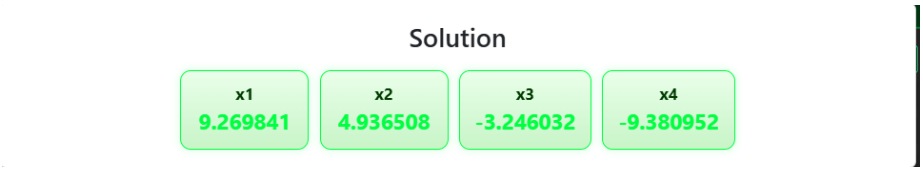



Figure 1.28: Testing on the page

### 1.2.7 Doolittle

The Doolittle method is an LU decomposition technique in which a matrix $A$ is factored as $A = LU$, where $L$ is a lower triangular matrix with ones on the diagonal and $U$ is an upper triangular matrix with general (non-unit) diagonal entries. The algorithm computes the rows of $U$ and the columns of $L$ step by step, ensuring that each element of $A$ is reproduced by the product $LU$. Doolittle's method is straightforward to implement and is often combined with pivoting for improved numerical stability.

**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

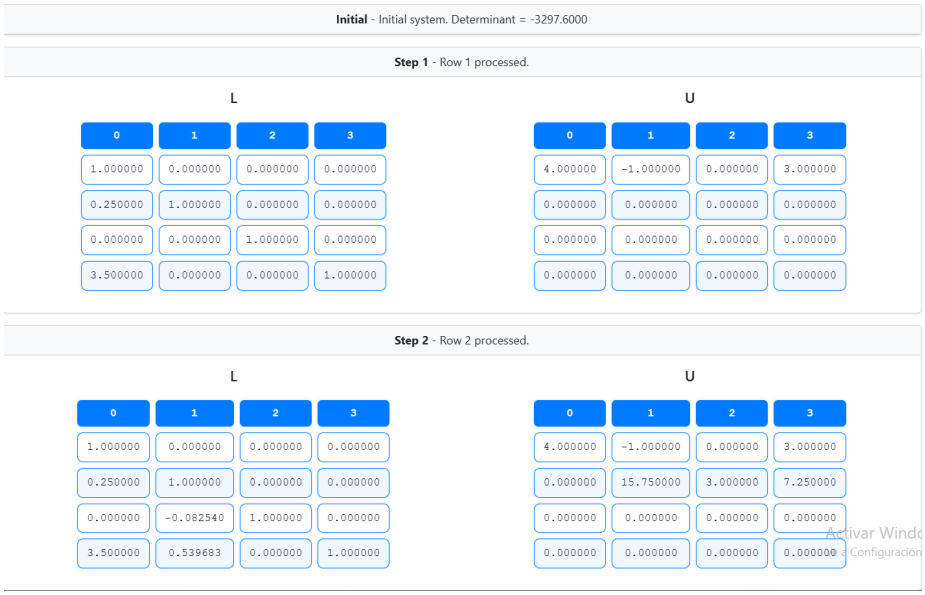



Figure 1.29: Testing on the page


Figure 1.30: Testing on the page

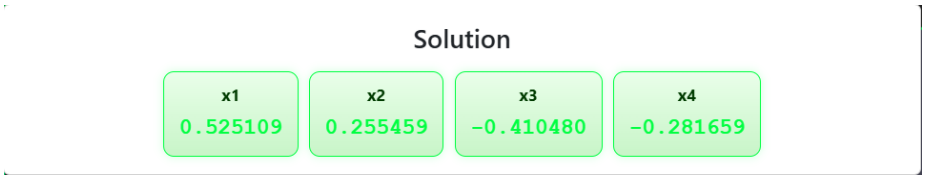



Figure 1.31: Testing on the page

### 1.2.8 Cholesky

The Cholesky method factors a symmetric, positive definite matrix $A$ into $A = LL^T$, where $L$ is a lower triangular matrix with positive diagonal entries. The algorithm computes each element of $L$ by matching the entries of $A$ with the product $LL^T$, using square roots for the diagonal terms and simple division for the off-diagonal terms. Cholesky decomposition is computationally efficient and numerically stable, making it a preferred method for solving systems with symmetric positive definite matrices.

**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**
  $t \leftarrow A_{k,k} - \sum_{s=0}^{k-1} L_{k,s}^2$
  **if** $t \le 0$ **then**
    **return** failure (matrix not SPD)
  **end if**
  $L_{k,k} \leftarrow \sqrt{t}$
  **for** $i = k+1$ **to** $n-1$ **do**
    $L_{i,k} \leftarrow \dfrac{A_{i,k} - \sum_{s=0}^{k-1} L_{i,s}L_{k,s}}{L_{k,k}}$
  **end for**
**end for**
Solve $Ly = b$ (forward); then $L^\top x = y$ (back)
**return** $L, L^\top, y, x$

**Testing**

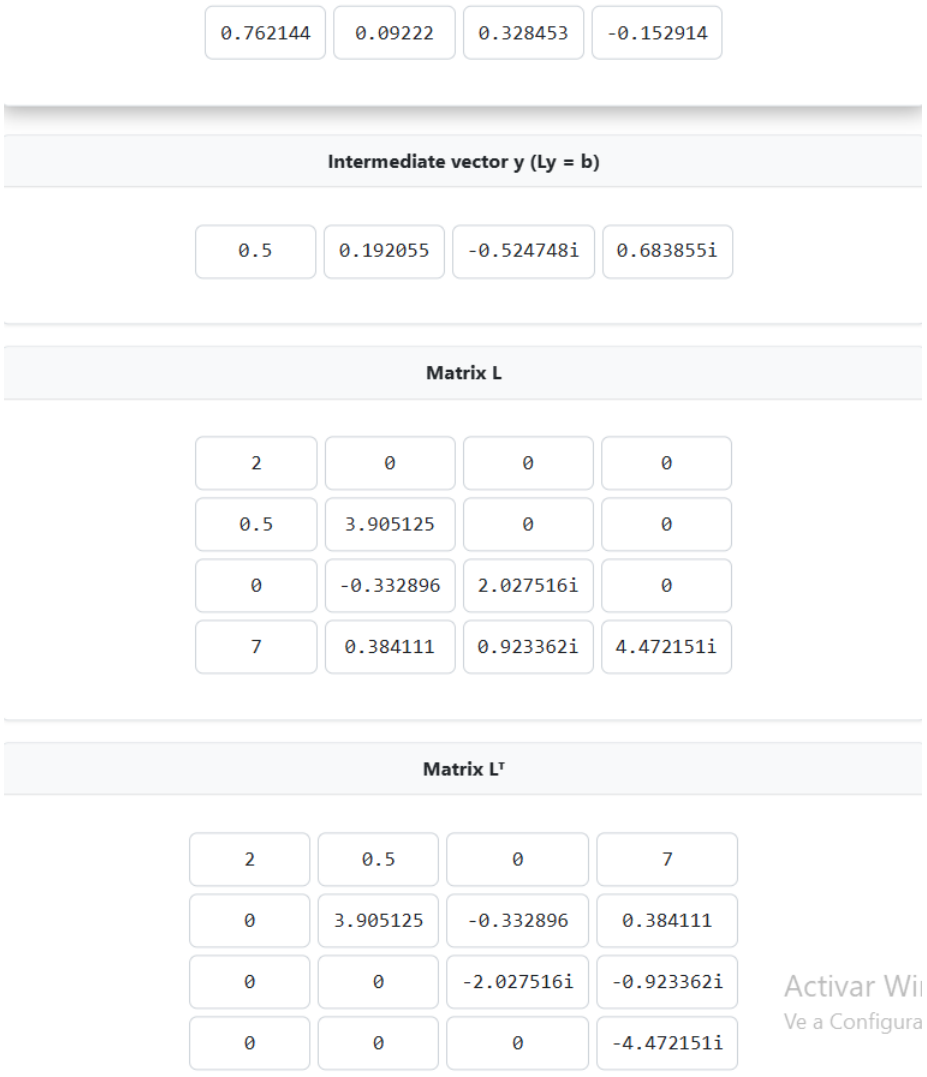



Figure 1.32: Testing on the page

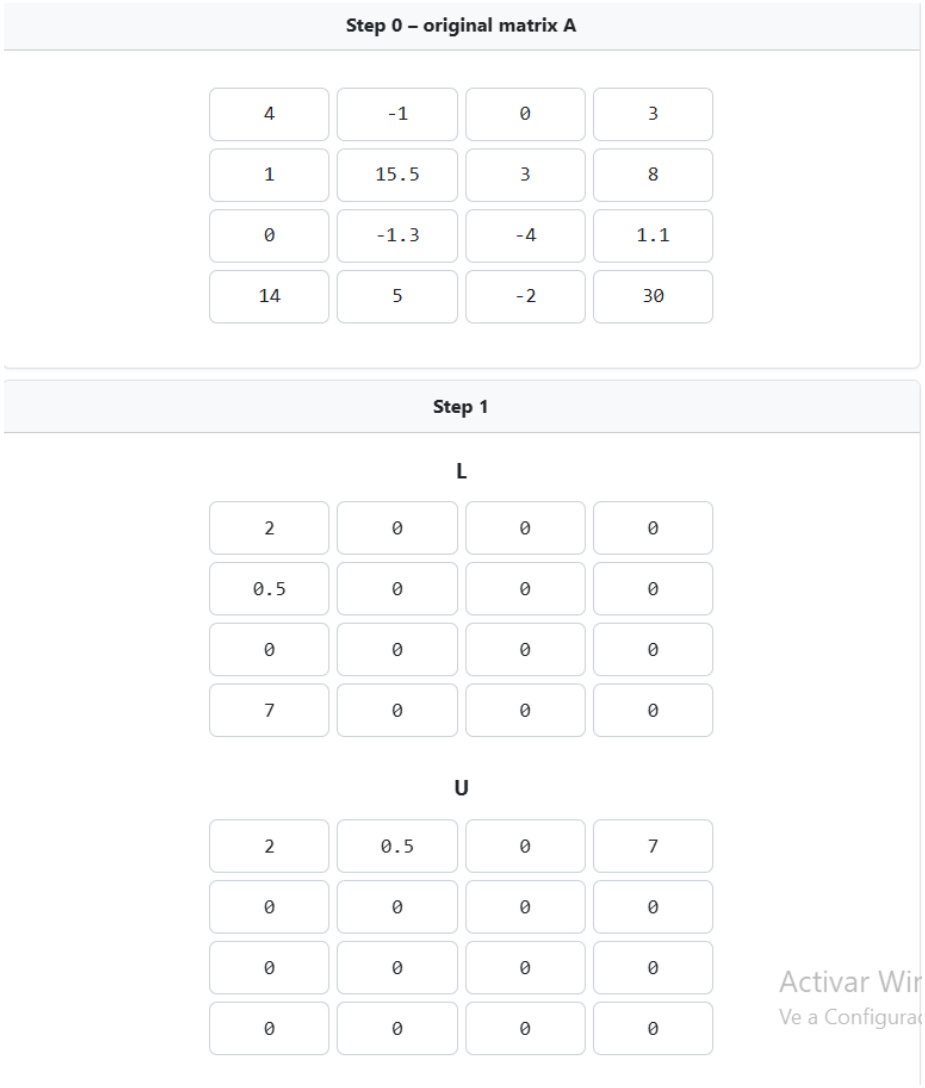



Figure 1.33: Testing on the page


Figure 1.34: Testing on the page

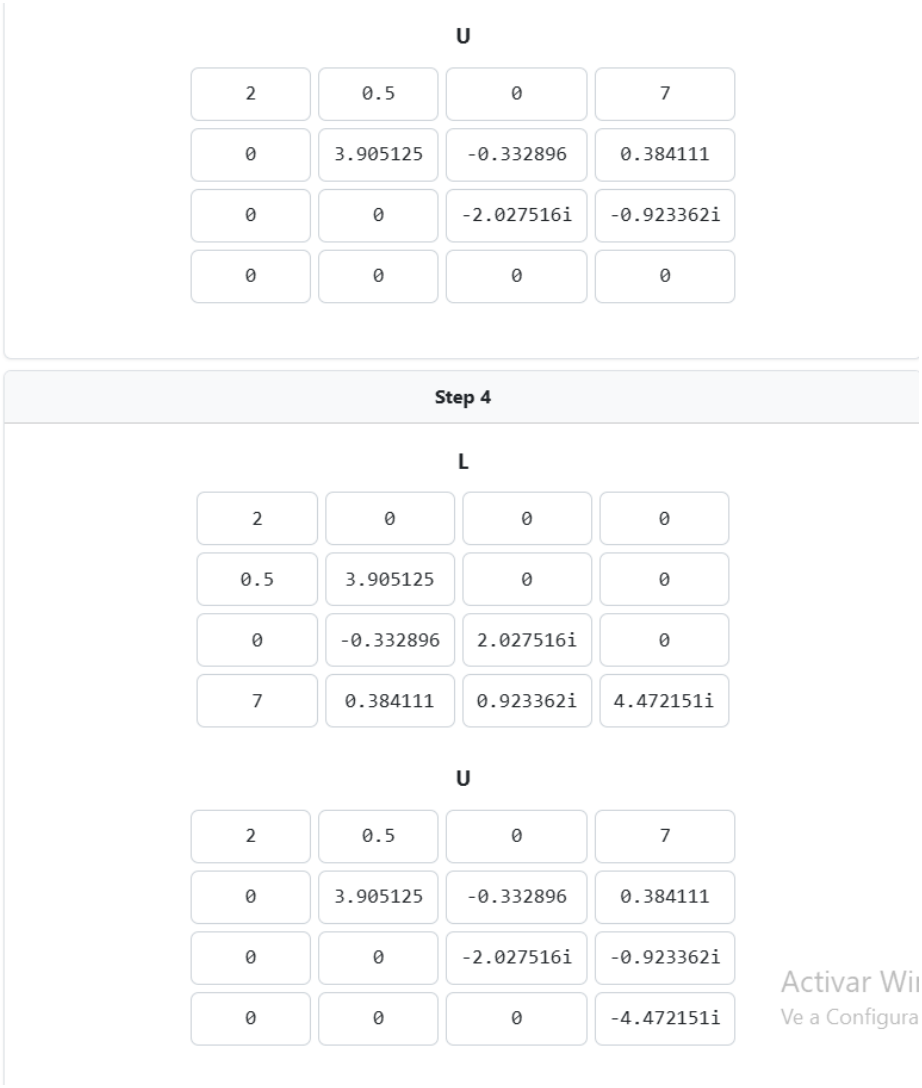



Figure 1.35: Testing on the page

### 1.2.9 Jacobi

The Jacobi method is an iterative technique for solving a linear system $A\mathbf{x} = \mathbf{b}$ by decomposing the coefficient matrix as $A = D + L + U$, where $D$ is the diagonal part of $A$, $L$ its strict lower triangular part, and $U$ its strict upper triangular part. Rearranging the system gives the iteration formula

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

The transition (iteration) matrix is therefore

$$T_J = -D^{-1}(L+U),$$

which governs the convergence of the method. Jacobi converges when the spectral radius $\rho(T_J) < 1$, commonly satisfied for diagonally dominant or symmetric positive definite matrices.

**Pseudocode**

**Algorithm 16** Jacobi Method

**Require:** Matrix $A \in \mathbb{R}^{n\times n}$, vector $b \in \mathbb{R}^n$, initial guess $x^{(0)}$, tolerance tol, maximum iterations $n_{\max}$, chosen norm.

**Ensure:** Approximate solution $x$ of the system $Ax = b$.

Let $D = \operatorname{diag}(A)$ be the diagonal matrix of $A$
**if** any $D_{ii} = 0$ **then**
    **error:** Jacobi cannot be applied.
**end if**
Compute $R = A - D$
Compute $D^{-1}$ as the diagonal matrix with entries $\frac{1}{A_{ii}}$
Set $x^{(0)}$ as the initial vector
**for** $k = 0$ **to** $n_{\max} - 1$ **do**
    Compute the new approximation:

$$x^{(k+1)} = D^{-1}\left(b - Rx^{(k)}\right)$$

    Compute the error:

$$e^{(k+1)} = \|x^{(k+1)} - x^{(k)}\|$$

    **if** $e^{(k+1)} <$ tol **then**
        **stop**
    **end if**
**end for**
**return** $x^{(k+1)}$ as the approximate solution

**Testing**

**Solution x**

| | | | |
|---|---|---|---|
| 0.525109 | 0.255458 | -0.410480 | -0.281659 |

**Iterations**

| k | x1 | x2 | x3 | x4 | error |
|---|---|---|---|---|---|
| 1.000000 | 0.250000 | 0.064516 | -0.250000 | 0.033333 | 0.360934 |
| 2.000000 | 0.241129 | 0.079570 | -0.261801 | -0.110753 | 0.145621 |
| 3.000000 | 0.352957 | 0.156793 | -0.306317 | -0.109909 | 0.143008 |
| 4.000000 | 0.371630 | 0.157759 | -0.331183 | -0.177933 | 0.074801 |
| 5.000000 | 0.422890 | 0.196476 | -0.350203 | -0.188466 | 0.067818 |
| 6.000000 | 0.440469 | 0.202287 | -0.365683 | -0.220108 | 0.039795 |
| 7.000000 | 0.465653 | 0.220480 | -0.376273 | -0.230312 | 0.034373 |
| 8.000000 | 0.477854 | 0.226172 | -0.384992 | -0.245803 | 0.022299 |
| 9.000000 | 0.490895 | 0.235067 | -0.391102 | -0.253027 | 0.018404 |
| 10.000000 | 0.498537 | 0.239137 | -0.395979 | -0.261002 | 0.012742 |
| 11.000000 | 0.505536 | 0.243705 | -0.399495 | -0.265572 | 0.010154 |
| 12.000000 | 0.510105 | 0.246292 | -0.402236 | -0.269834 | 0.007297 |
| 13.000000 | 0.513948 | 0.248727 | -0.404249 | -0.272580 | 0.005683 |
| 14.000000 | 0.516617 | 0.250286 | -0.405796 | -0.274914 | 0.004170 |



Figure 1.36: Testing on the page

| | | | | | |
|---|---|---|---|---|---|
| 34.000000 | 0.525080 | 0.255441 | -0.410464 | -0.281636 | 1.466980e-5 |
| 35.000000 | 0.525087 | 0.255445 | -0.410468 | -0.281642 | 1.105470e-5 |
| 36.000000 | 0.525092 | 0.255448 | -0.410471 | -0.281646 | 8.329473e-6 |
| 37.000000 | 0.525097 | 0.255451 | -0.410473 | -0.281649 | 6.276644e-6 |
| 38.000000 | 0.525100 | 0.255453 | -0.410475 | -0.281652 | 4.729419e-6 |
| 39.000000 | 0.525102 | 0.255454 | -0.410476 | -0.281654 | 3.563776e-6 |
| 40.000000 | 0.525104 | 0.255455 | -0.410477 | -0.281655 | 2.685321e-6 |
| 41.000000 | 0.525105 | 0.255456 | -0.410478 | -0.281656 | 2.023460e-6 |
| 42.000000 | 0.525106 | 0.255457 | -0.410479 | -0.281657 | 1.524697e-6 |
| 43.000000 | 0.525107 | 0.255457 | -0.410479 | -0.281658 | 1.148893e-6 |
| 44.000000 | 0.525107 | 0.255457 | -0.410479 | -0.281658 | 8.657059e-7 |
| 45.000000 | 0.525108 | 0.255458 | -0.410480 | -0.281658 | 6.523267e-7 |
| 46.000000 | 0.525108 | 0.255458 | -0.410480 | -0.281659 | 4.915377e-7 |
| 47.000000 | 0.525108 | 0.255458 | -0.410480 | -0.281659 | 3.703829e-7 |
| 48.000000 | 0.525109 | 0.255458 | -0.410480 | -0.281659 | 2.790893e-7 |
| 49.000000 | 0.525109 | 0.255458 | -0.410480 | -0.281659 | 2.102988e-7 |
| 50.000000 | 0.525109 | 0.255458 | -0.410480 | -0.281659 | 1.584635e-7 |
| 51.000000 | 0.525109 | 0.255458 | -0.410480 | -0.281659 | 1.194050e-7 |
| 52.000000 | 0.525109 | 0.255458 | -0.410480 | -0.281659 | 8.997367e-8 |



Figure 1.37: Testing on the page

### 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 17** 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

## Testing

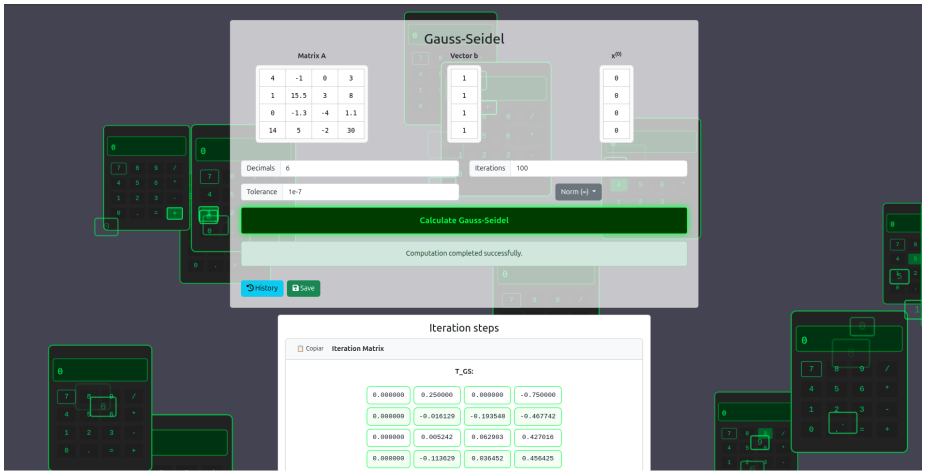



Figure 1.38: Testing Gauss Seidel on page 1

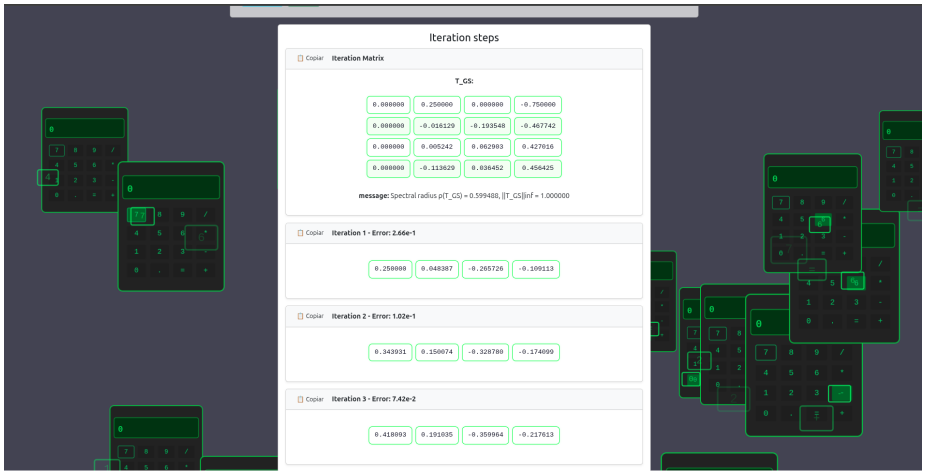



Figure 1.39: Testing Gauss Seidel on page 2

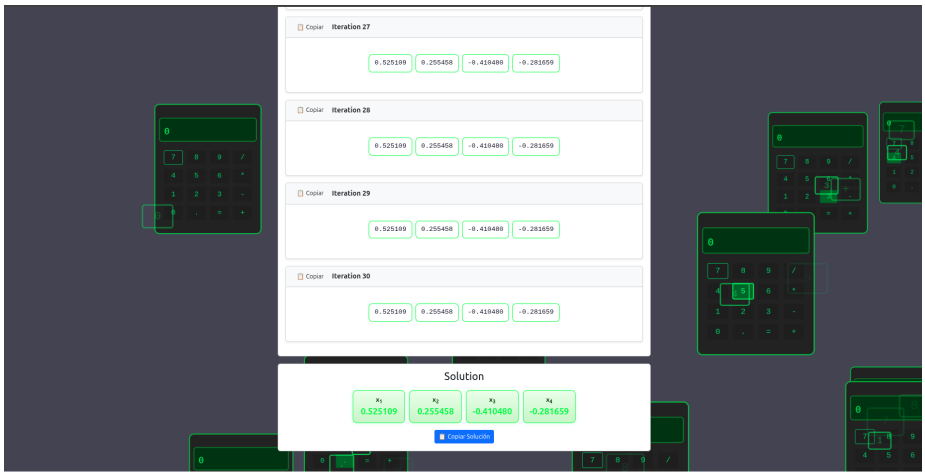



Figure 1.40: Testing Gauss Seidel on page 3

### 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** $\omega$, 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 18** Successive Over-Relaxation (SOR) Method

**Require:** Matrix $A \in \mathbb{R}^{n \times n}$, vector $b \in \mathbb{R}^n$, relaxation parameter $\omega$, tolerance $\varepsilon > 0$, maximum iterations $n_{max}$

**Ensure:** Approximate solution $\mathbf{x}$ satisfying the tolerance

Initialize $\mathbf{x}^{(0)} \leftarrow \mathbf{0}$ if no initial guess is provided
Initialize history arrays for $\mathbf{x}$, absolute error, and iteration count
**for** $k \leftarrow 1$ to $n_{max}$ **do**
    $\mathbf{x}_{old} \leftarrow \mathbf{x}$
    **for** $i \leftarrow 0$ to $n-1$ **do**
        $sum1 \leftarrow 0$
        **for** $j \leftarrow 0$ to $i-1$ **do**
            $sum1 \leftarrow sum1 + A[i][j] \cdot x[j]$
        **end for**
        $sum2 \leftarrow 0$
        **for** $j \leftarrow i+1$ to $n-1$ **do**
            $sum2 \leftarrow sum2 + A[i][j] \cdot x_{old}[j]$
        **end for**
        $x[i] \leftarrow (1-\omega) \cdot x_{old}[i] + (\omega / A[i][i]) \cdot (b[i] - sum1 - sum2)$
    **end for**
    $error \leftarrow \|\mathbf{x} - \mathbf{x}_{old}\|_\infty$
    Store $\mathbf{x}$, $error$, and $k$ in history
    **if** $error < \varepsilon$ **then**
        **return** $\mathbf{x}$ with message *"Tolerance satisfied"*
    **end if**
**end for**
**return** $\mathbf{x}$ with message *"Maximum number of iterations reached"*

## Testing

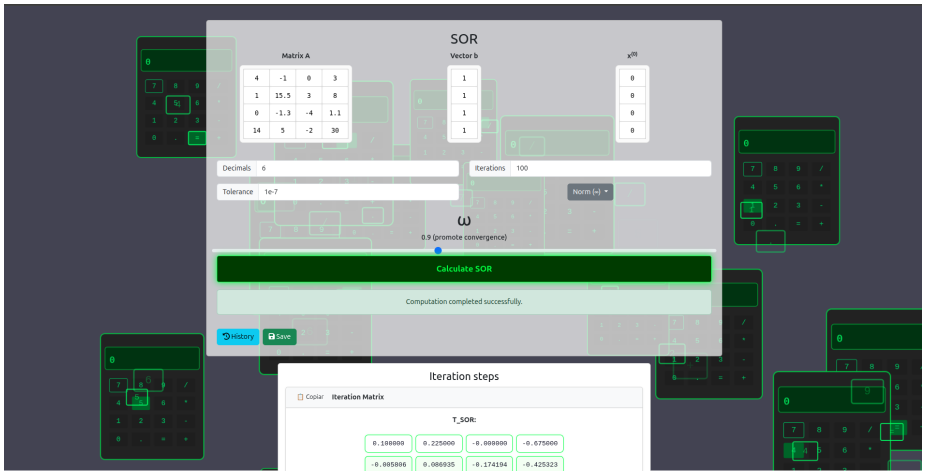



Figure 1.41: Testing SOR on page 1

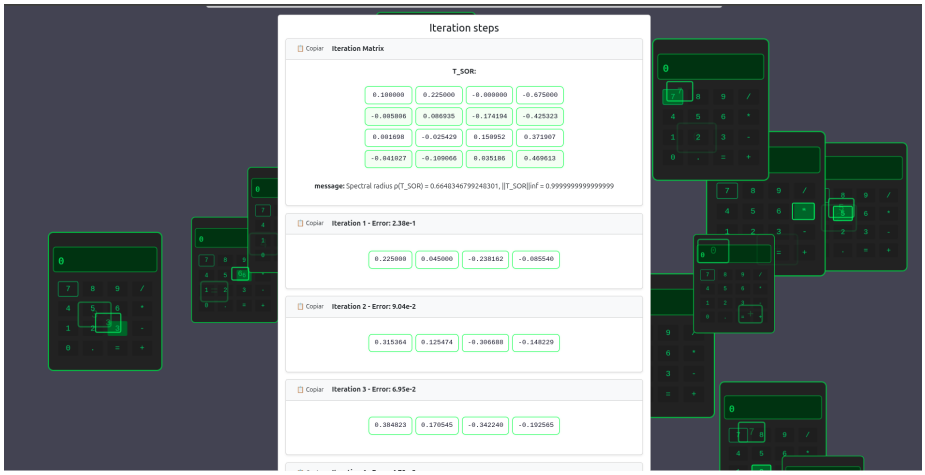



Figure 1.42: Testing SOR on page 2

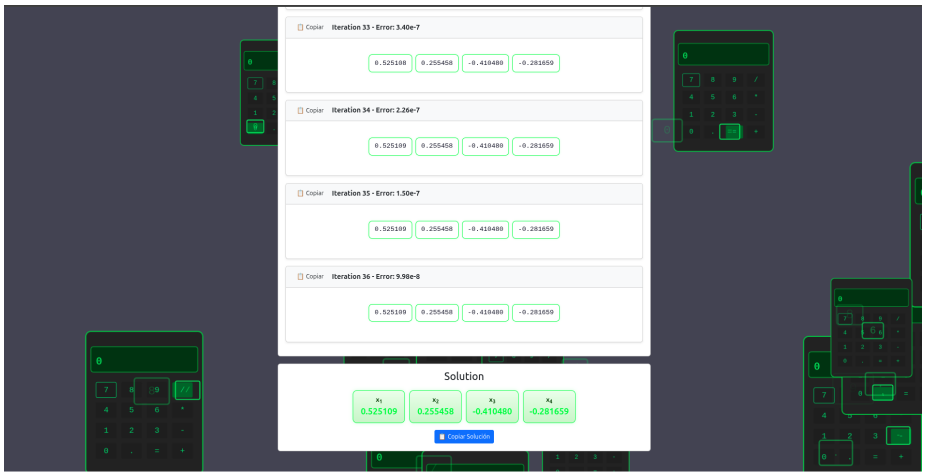



Figure 1.43: Testing SOR on page 3

# 1.3 Interpolation

## 1.3.1 Vandermonde

### Pseudocode

**Algorithm 19** 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**

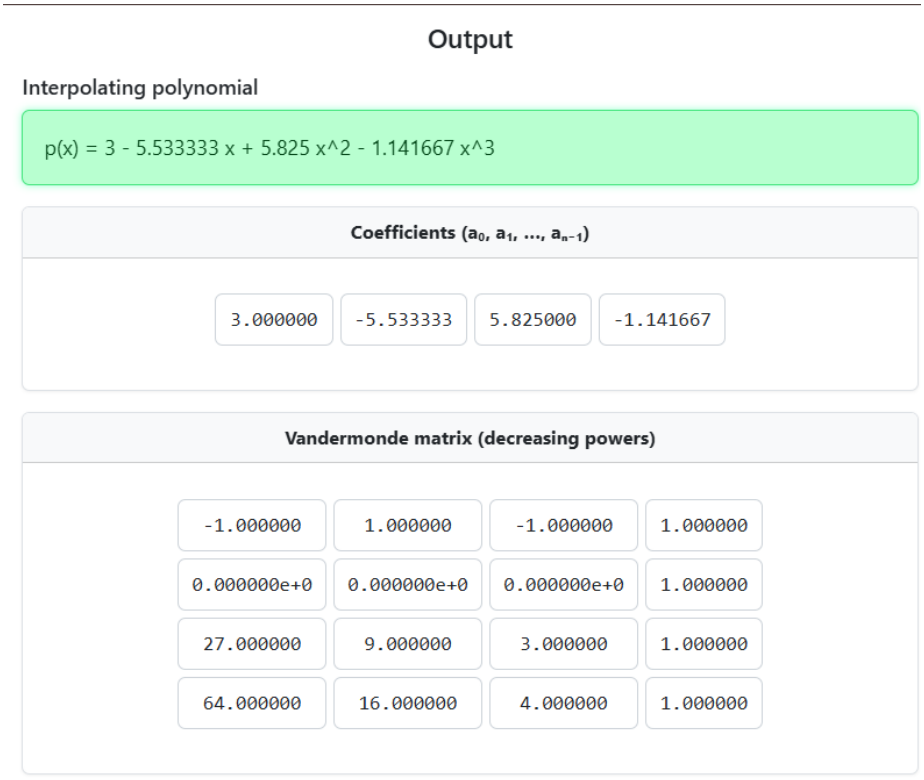



Figure 1.44: Testing on the page

### 1.3.2 Newton Interpolation

The Newton method constructs the unique interpolating polynomial $p(x)$ of degree $n$ that passes through the $n+1$ points $(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))$. The polynomial is expressed using **divided differences** and follows a nested product form.

The interpolating polynomial is given by:

$$p(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \cdots + f[x_0, \ldots, x_n](x - x_0) \cdots (x - x_{n-1})$$

where $f[x_0, \ldots, x_k]$ denotes the divided difference of order $k$.

#### Pseudocode

The procedure calculates the interpolating polynomial using divided differences and constructs both symbolic and Newton-form representations.

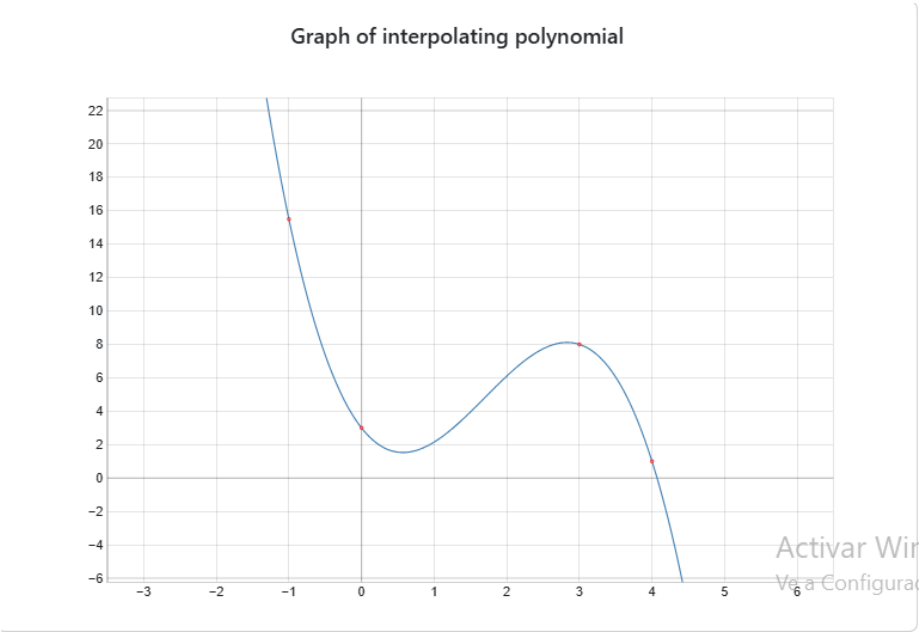



Figure 1.45: Testing on the page (graphic)

**Algorithm 20** Newton Interpolation Method

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

$n \leftarrow \text{length}(X)$
Initialize difference table $diff \in \mathbb{R}^{n \times n}$
$diff[:,0] \leftarrow Y$ ▷ First column contains function values
**for** $j \leftarrow 1$ to $n-1$ **do** ▷ Compute divided differences
    **for** $i \leftarrow 0$ to $n-j-1$ **do**
        $denom \leftarrow X[i+j] - X[i]$
        **if** $|denom| < \varepsilon$ **then** ▷ Check for small denominator
            **return** Error: "Denominator too small"
        **end if**
        $diff[i,j] \leftarrow (diff[i+1, j-1] - diff[i, j-1])/denom$
    **end for**
**end for**
$b \leftarrow [diff[0,0], diff[0,1], \ldots, diff[0,n-1]]$ ▷ Coefficients for Newton form
$P(x) \leftarrow b[0]$ ▷ Initialize polynomial
$prod \leftarrow 1$
**for** $i \leftarrow 1$ to $n-1$ **do** ▷ Build Newton polynomial
    $prod \leftarrow prod \cdot (x - X[i-1])$
    $P(x) \leftarrow P(x) + b[i] \cdot prod$
**end for**
**return** $P(x)$ ▷ Newton interpolating polynomial in symbolic and expanded form

**Testing**

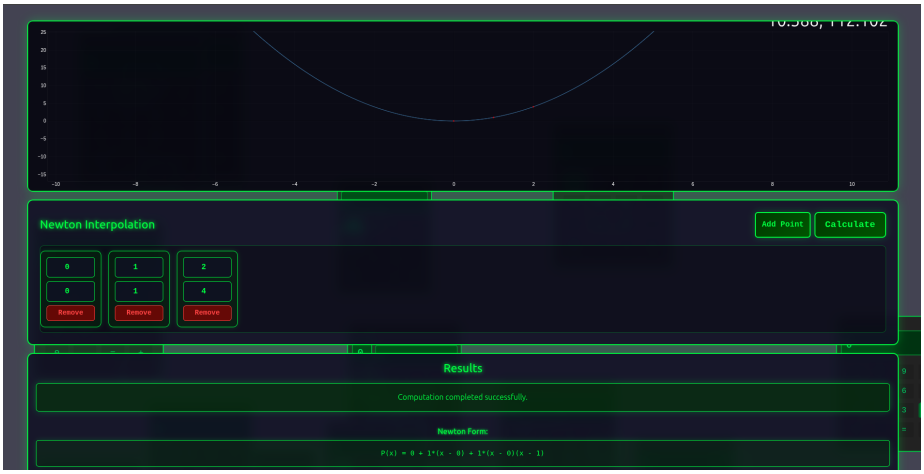



Figure 1.46: Testing on a page

### 1.3.3 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 21** Lagrange Interpolation Method

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

$n \leftarrow \text{length}(X) - 1$
$p_L(x) \leftarrow 0$ ▷ Initialize Lagrange Polynomial
**for** $k \leftarrow 0$ to $n$ **do**
    $L_k \leftarrow 1$ ▷ 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** $p_L(x)$ ▷ Lagrange Polynomial $p_L(x)$ with 15 presicion digits

**Testing**

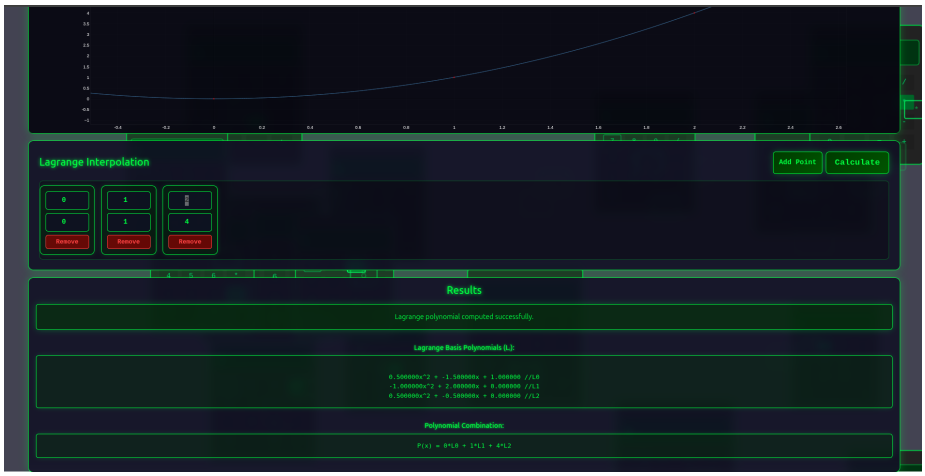



Figure 1.47: Testing on a page

## 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 Linear tracers

#### Pseudocode

**Algorithm 22** Piecewise Linear Spline Construction

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

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

#### Testing

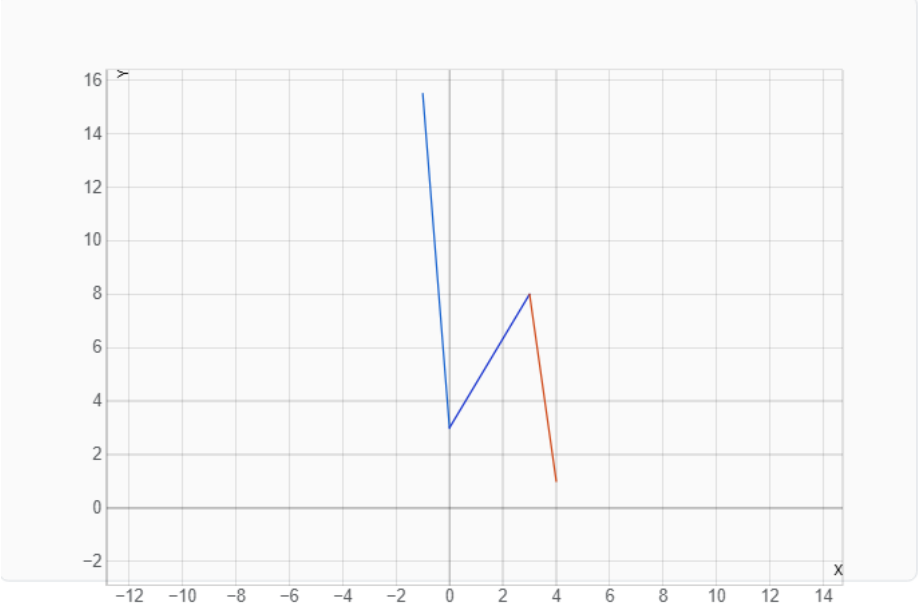



Figure 1.48: Testing on the page (graphic)


Figure 1.49: Testing on the page

### 1.4.2 Quadratic tracers

#### Pseudocode

**Algorithm 23** 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

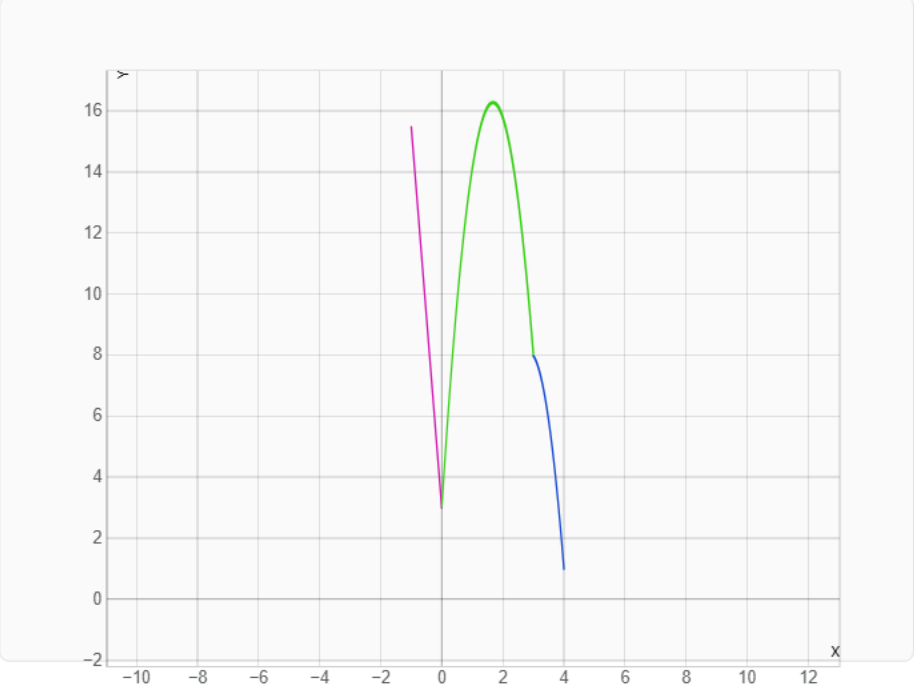



Figure 1.50: Testing on the page

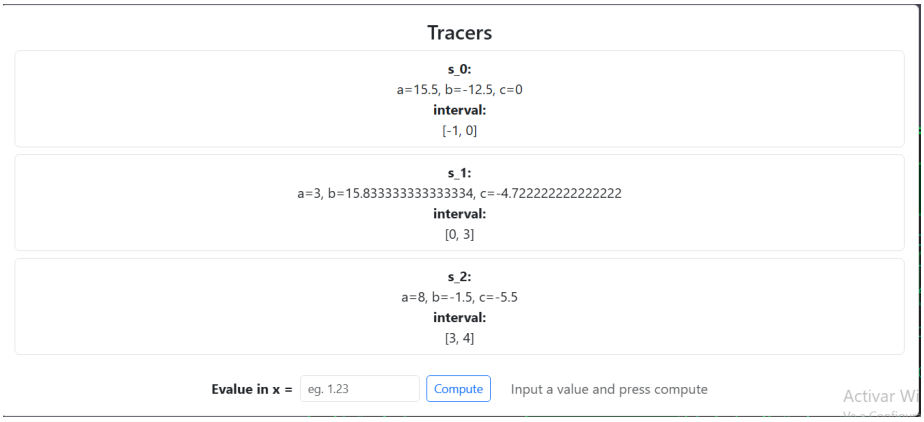



Figure 1.51: Testing on the page

### 1.4.3 Cubic tracers

**Pseudocode**

**Algorithm 24** Cubic Spline Construction

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

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

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

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

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

## Testing

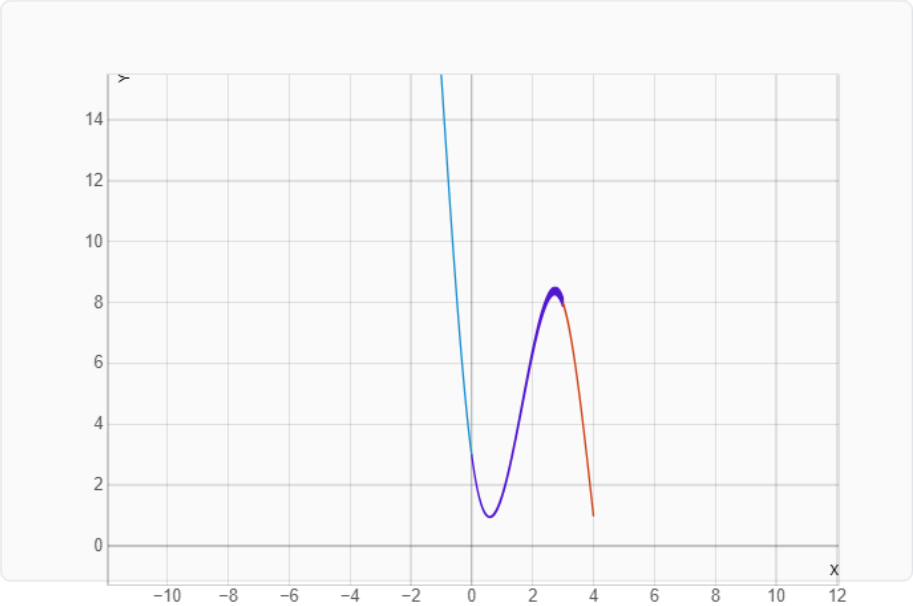



Figure 1.52: Testing on the page

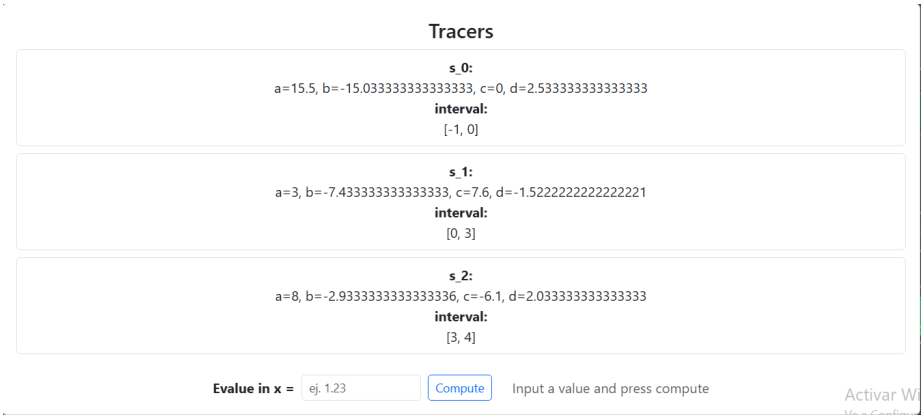



Figure 1.53: Testing on the page