

# Numerical Methods Pseudocodes

## 1 Nonlinear Equations

### 1.1 Incremental Search

```
INCREMENTAL_SEARCH(f, x0, delta, nmax):
// Incremental search for sign changes of f(x).
// Checks successive intervals of length 'delta' and prints intervals
// where a root is likely.
x_prev <- x0
f_prev <- f(x_prev)
FOR i <- 0 TO nmax DO
    x_curr <- x_prev + delta
    f_curr <- f(x_curr)
    // If function changes sign, a root is in that interval
    IF f_prev * f_curr < 0 THEN
        PRINT "Root in interval [", x_prev, ", ", x_curr, "]"
    END IF
    x_prev <- x_curr
    f_prev <- f_curr
END FOR
```

### 1.2 Bisection Method

```
BISECTION(f, a, b, tol, nmax):
// Bisection method: reliable root-finding when f(a) and f(b) have
// opposite signs. tol is tolerance, nmax limits iterations.
fa <- f(a)
fb <- f(b)
// If no sign change in initial interval, method cannot proceed
IF fa * fb > 0 THEN
    PRINT "No root in interval"
    RETURN ERROR
END IF
FOR k <- 1 TO nmax DO
    xm <- (a + b) / 2
    fm <- f(xm)
    PRINT k, a, b, xm, fm
    // Stop if function value is small or interval is sufficiently small
    IF |fm| < tol OR (b - a)/2 < tol THEN
        RETURN xm
    END IF
    // Decide which half contains the root by sign change
    IF fa * fm < 0 THEN
        b <- xm
        fb <- fm
    ELSE
        a <- xm
```

```

        fa <- fm
    END IF
END FOR
RETURN xm

```

### 1.3 False Position

```

FALSE_POSITION(f, a, b, tol, nmax):
// Regula Falsi: uses linear interpolation to pick next point.
// Often converges faster than bisection, but can be slow if one
// endpoint stays fixed.
    fa <- f(a)
    fb <- f(b)
    IF fa * fb > 0 THEN
        PRINT "No root in interval"
        RETURN ERROR
    END IF
    FOR k <- 1 TO nmax DO
        // Compute intersection of secant line with x-axis
        x <- (a*fb - b*fa) / (fb - fa)
        fx <- f(x)
        PRINT k, a, b, x, fx
        IF |fx| < tol THEN
            RETURN x
        END IF
        // Replace endpoint with same sign as f(x)
        IF fa * fx < 0 THEN
            b <- x
            fb <- fx
        ELSE
            a <- x
            fa <- fx
        END IF
    END FOR
    RETURN x

```

### 1.4 Fixed Point

```

FIXED_POINT(g, x0, tol, nmax):
// Fixed-point iteration: find x such that x = g(x).
// Convergence depends on g's derivative near the fixed point.
    x <- x0
    FOR k <- 1 TO nmax DO
        x_new <- g(x)
        err <- |x_new - x|
        PRINT k, x, x_new, err
        // Stop when consecutive iterates are within tolerance
        IF err < tol THEN
            RETURN x_new
        END IF
        x <- x_new
    END FOR
    RETURN x

```

### 1.5 Newton-Raphson

```

NEWTON(f, f', x0, tol, nmax):
// Newton-Raphson method: uses derivative for fast convergence.
// Requires f'(x) != 0 near iterates to avoid division by zero.
    x <- x0
    FOR k <- 1 TO nmax DO
        fx <- f(x)
        fpx <- f'(x)
        IF fpx = 0 THEN
            PRINT "Zero derivative"
            RETURN ERROR
        END IF
        // Newton update
        x_new <- x - fx / fpx
        err <- |x_new - x|
        PRINT k, x, fx, fpx, x_new, err
        IF err < tol THEN
            RETURN x_new
        END IF
        x <- x_new
    END FOR
    RETURN x

```

## 1.6 Secant Method

```

SECANT(f, x0, x1, tol, nmax):
// Secant method: approximates derivative by slope between last
// two iterates. Requires two initial guesses. Convergence is
// superlinear but not quadratic.
    FOR k <- 1 TO nmax DO
        f0 <- f(x0)
        f1 <- f(x1)
        IF f1 - f0 = 0 THEN
            PRINT "Division by zero"
            RETURN ERROR
        END IF
        // Secant formula to compute next approximation
        x2 <- x1 - f1 * (x1 - x0) / (f1 - f0)
        err <- |x2 - x1|
        PRINT k, x0, x1, x2, err
        IF err < tol THEN
            RETURN x2
        END IF
        // Shift iterates
        x0 <- x1
        x1 <- x2
    END FOR
    RETURN x1

```

## 1.7 Multiple Roots

```

MULTIPLE_ROOTS(f, f', f'', x0, tol, nmax):
// Newton method adapted for multiple roots using second derivative.
// Modified formula restores quadratic convergence without knowing
// multiplicity.
    x <- x0
    FOR k <- 1 TO nmax DO

```

```

fx <- f(x)
fpx <- f'(x)
fppx <- f''(x)
denom <- fpx^2 - fx * fppx
IF denom = 0 THEN
    PRINT "Zero denominator"
    RETURN ERROR
END IF
// Modified Newton update for multiple roots
x_new <- x - (fx * fpx) / denom
err <- |x_new - x|
PRINT k, x, fx, fpx, fppx, x_new, err
IF err < tol THEN
    RETURN x_new
END IF
x <- x_new
END FOR
RETURN x

```

## 2 Linear Systems

### 2.1 Simple Gaussian Elimination

```

GAUSS_SIMPLE(A, b):
// Simple Gaussian elimination without pivoting.
// Solves Ax = b by reducing A to upper triangular form, then
// back substitution.
n <- number of rows
M <- [A | b] // Augmented matrix
// Forward elimination
FOR k <- 0 TO n-2 DO
    IF |M[k][k]| < epsilon THEN
        RETURN ERROR "Zero pivot"
    END IF
    FOR i <- k+1 TO n-1 DO
        factor <- M[i][k] / M[k][k]
        FOR j <- k TO n DO
            M[i][j] <- M[i][j] - factor * M[k][j]
        END FOR
    END FOR
END FOR
// Backward substitution
FOR i <- n-1 DOWNTO 0 DO
    sum <- M[i][n]
    FOR j <- i+1 TO n-1 DO
        sum <- sum - M[i][j] * x[j]
    END FOR
    x[i] <- sum / M[i][i]
END FOR
RETURN x

```

### 2.2 Gaussian with Partial Pivoting

```

GAUSS_PARTIAL(A, b):
// Gaussian elimination with partial pivoting for numerical stability.
// Swaps rows to place largest pivot element.

```

```

n <- number of rows
M <- [A | b]
FOR k <- 0 TO n-2 DO
    // Find row with largest pivot in column k
    max_row <- k
    FOR i <- k+1 TO n-1 DO
        IF |M[i] [k]| > |M[max_row] [k]| THEN
            max_row <- i
        END IF
    END FOR
    IF max_row != k THEN
        SWAP M[k] and M[max_row]
    END IF
    IF |M[k] [k]| < epsilon THEN
        RETURN ERROR "Zero pivot"
    END IF
    // Elimination
    FOR i <- k+1 TO n-1 DO
        factor <- M[i] [k] / M[k] [k]
        FOR j <- k TO n DO
            M[i] [j] <- M[i] [j] - factor * M[k] [j]
        END FOR
    END FOR
END FOR
// Backward substitution
FOR i <- n-1 DOWNTO 0 DO
    sum <- M[i] [n]
    FOR j <- i+1 TO n-1 DO
        sum <- sum - M[i] [j] * x[j]
    END FOR
    x[i] <- sum / M[i] [i]
END FOR
RETURN x

```

## 2.3 Gaussian with Total Pivoting

```

GAUSS_TOTAL(A, b):
// Gaussian elimination with total pivoting: swaps rows AND columns.
// Most numerically stable but requires tracking column permutations.
n <- number of rows
M <- [A | b]
marks <- [0, 1, 2, ..., n-1] // Track column permutations
FOR k <- 0 TO n-2 DO
    // Find largest element in remaining submatrix
    max_val <- 0
    max_row <- k
    max_col <- k
    FOR i <- k TO n-1 DO
        FOR j <- k TO n-1 DO
            IF |M[i] [j]| > max_val THEN
                max_val <- |M[i] [j]|
                max_row <- i
                max_col <- j
            END IF
        END FOR
    END FOR
    IF max_val < epsilon THEN

```

```

        RETURN ERROR "Singular matrix"
END IF
// Row swap
IF max_row != k THEN
    SWAP M[k] and M[max_row]
END IF
// Column swap
IF max_col != k THEN
    FOR i <- 0 TO n-1 DO
        SWAP M[i][k] and M[i][max_col]
    END FOR
    SWAP marks[k] and marks[max_col]
END IF
// Elimination
FOR i <- k+1 TO n-1 DO
    factor <- M[i][k] / M[k][k]
    FOR j <- k TO n DO
        M[i][j] <- M[i][j] - factor * M[k][j]
    END FOR
END FOR
// Backward substitution
x_temp <- vector of zeros size n
FOR i <- n-1 DOWNTTO 0 DO
    sum <- M[i][n]
    FOR j <- i+1 TO n-1 DO
        sum <- sum - M[i][j] * x_temp[j]
    END FOR
    x_temp[i] <- sum / M[i][i]
END FOR
// Reorder solution using marks
FOR i <- 0 TO n-1 DO
    x[marks[i]] <- x_temp[i]
END FOR
RETURN x

```

## 2.4 LU Factorization (Simple)

```

LU_SIMPLE(A, b):
// Simple LU factorization using Gaussian elimination without pivoting.
// Factorizes A = LU, then solves Ly = b and Ux = y.
n <- number of rows of A
L <- identity matrix nxn
U <- zero matrix nxn
M <- copy of A
// Forward elimination to produce U in M and multipliers in L
FOR i <- 0 TO n-2 DO
    FOR j <- i+1 TO n-1 DO
        IF M[j][i] != 0 THEN
            L[j][i] <- M[j][i] / M[i][i] // Save multiplier
            FOR k <- i TO n-1 DO
                M[j][k] <- M[j][k] - L[j][i] * M[i][k]
            END FOR
        END IF
    END FOR
    FOR k <- i TO n-1 DO
        U[i][k] <- M[i][k] // Copy row to U
    END FOR

```

```

        END FOR
    END FOR
    FOR k <- 0 TO n-1 DO
        U[n-1][k] <- M[n-1][k]
    END FOR
    // Solve Ly = b using forward substitution
    z <- FORWARD_SUB(L, b)
    // Solve Ux = z using backward substitution
    x <- BACKWARD_SUB(U, z)
    RETURN (x, L, U)

```

## 2.5 LU with Partial Pivoting

```

LU_PARTIAL(A, b):
// LU factorization with partial pivoting for numerical stability.
// Produces permutation matrix P such that PA = LU.
n <- number of rows of A
L <- identity matrix nxn
U <- zero matrix nxn
P <- identity matrix nxn
M <- copy of A
FOR i <- 0 TO n-2 DO
    // Find largest absolute value in column i below row i
    max_val <- |M[i][i]|
    max_row <- i
    FOR k <- i+1 TO n-1 DO
        IF |M[k][i]| > max_val THEN
            max_val <- |M[k][i]|
            max_row <- k
        END IF
    END FOR
    IF max_row != i THEN
        SWAP M[i] and M[max_row]
        SWAP P[i] and P[max_row]
        // Swap corresponding entries in L if multipliers computed
        IF i > 0 THEN
            FOR k <- 0 TO i-1 DO
                SWAP L[i][k] and L[max_row][k]
            END FOR
        END IF
    END IF
    // LU decomposition
    FOR j <- i+1 TO n-1 DO
        L[j][i] <- M[j][i] / M[i][i]
        FOR k <- i TO n-1 DO
            M[j][k] <- M[j][k] - L[j][i] * M[i][k]
        END FOR
    END FOR
    FOR k <- i TO n-1 DO
        U[i][k] <- M[i][k]
    END FOR
    END FOR
    FOR k <- 0 TO n-1 DO
        U[n-1][k] <- M[n-1][k]
    END FOR
    // Apply permutation to b, then solve
    Pb <- P * b

```

```

z <- FORWARD_SUB(L, Pb)
x <- BACKWARD_SUB(U, z)
RETURN (x, L, U, P)

```

## 2.6 Crout's Method

```

CROUT(A, b):
// Crout's LU factorization: L has full diagonal, U has unit diagonal.
// Useful for in-place computation.
n <- size of A
L <- identity matrix nxn
U <- identity matrix nxn
FOR i <- 0 TO n-2 DO
    // Compute L column entries for column i
    FOR j <- i TO n-1 DO
        sum <- 0
        FOR k <- 0 TO i-1 DO
            sum <- sum + L[j][k] * U[k][i]
        END FOR
        L[j][i] <- A[j][i] - sum
    END FOR
    // Compute U row entries for row i
    FOR j <- i+1 TO n-1 DO
        sum <- 0
        FOR k <- 0 TO i-1 DO
            sum <- sum + L[i][k] * U[k][j]
        END FOR
        U[i][j] <- (A[i][j] - sum) / L[i][i]
    END FOR
END FOR
// Last diagonal element of L
sum <- 0
FOR k <- 0 TO n-2 DO
    sum <- sum + L[n-1][k] * U[k][n-1]
END FOR
L[n-1][n-1] <- A[n-1][n-1] - sum
z <- FORWARD_SUB(L, b)
x <- BACKWARD_SUB(U, z)
RETURN (x, L, U)

```

## 2.7 Doolittle's Method

```

DOOLITTLE(A, b):
// Doolittle's LU factorization: U has full diagonal, L has unit diagonal.
// Common textbook approach for LU decomposition.
n <- size of A
L <- identity matrix nxn
U <- identity matrix nxn
FOR i <- 0 TO n-2 DO
    // Compute U row i
    FOR j <- i TO n-1 DO
        sum <- 0
        FOR k <- 0 TO i-1 DO
            sum <- sum + L[i][k] * U[k][j]
        END FOR
        U[i][j] <- A[i][j] - sum
    END FOR

```

```

    END FOR
    // Compute L column i
    FOR j <- i+1 TO n-1 DO
        sum <- 0
        FOR k <- 0 TO i-1 DO
            sum <- sum + L[j][k] * U[k][i]
        END FOR
        L[j][i] <- (A[j][i] - sum) / U[i][i]
    END FOR
END FOR
// Last diagonal element of U
sum <- 0
FOR k <- 0 TO n-2 DO
    sum <- sum + L[n-1][k] * U[k][n-1]
END FOR
U[n-1][n-1] <- A[n-1][n-1] - sum
z <- FORWARD_SUB(L, b)
x <- BACKWARD_SUB(U, z)
RETURN (x, L, U)

```

## 2.8 Cholesky Factorization

```

CHOLESKY(A, b):
// Cholesky factorization for symmetric positive-definite matrices.
// A = L*L^T. More efficient and stable for this matrix class.
n <- size of A
L <- zero matrix nxn
U <- zero matrix nxn
FOR i <- 0 TO n-2 DO
    // Compute diagonal element of L
    sum <- 0
    FOR k <- 0 TO i-1 DO
        sum <- sum + L[i][k] * U[k][i]
    END FOR
    L[i][i] <- sqrt(A[i][i] - sum)
    U[i][i] <- L[i][i] // U is transpose of L
    // Compute off-diagonal elements
    FOR j <- i+1 TO n-1 DO
        sum_L <- 0
        FOR k <- 0 TO i-1 DO
            sum_L <- sum_L + L[j][k] * U[k][i]
        END FOR
        L[j][i] <- (A[j][i] - sum_L) / U[i][i]
        sum_U <- 0
        FOR k <- 0 TO i-1 DO
            sum_U <- sum_U + L[i][k] * U[k][j]
        END FOR
        U[i][j] <- (A[i][j] - sum_U) / L[i][i]
    END FOR
END FOR
// Last diagonal element
sum <- 0
FOR k <- 0 TO n-2 DO
    sum <- sum + L[n-1][k] * U[k][n-1]
END FOR
L[n-1][n-1] <- sqrt(A[n-1][n-1] - sum)
U[n-1][n-1] <- L[n-1][n-1]

```

```

z <- FORWARD_SUB(L, b)
x <- BACKWARD_SUB(U, z)
RETURN (x, L, U)

```

## 2.9 Jacobi Method

```

JACOBI(A, b, x0, tol, nmax):
// Jacobi iterative method: solves Ax = b by splitting A = D - (L + U).
// Convergence requires spectral radius of iteration matrix < 1.
    D <- diagonal matrix of A
    L <- -lower_triangular(A) + D
    U <- -upper_triangular(A) + D
    T <- inverse(D) * (L + U) // Iteration matrix
    C <- inverse(D) * b        // Constant term
    x_old <- x0
    err <- 1000
    iter <- 0
    WHILE err > tol AND iter < nmax DO
        x_new <- T * x_old + C
        err <- ||x_old - x_new|| // Norm of difference
        x_old <- x_new
        iter <- iter + 1
    END WHILE
    RETURN (x_new, iter, err)

```

## 2.10 Gauss-Seidel Method

```

GAUSS_SEIDEL(A, b, x0, tol, nmax):
// Gauss-Seidel method: uses new component values immediately.
// Often converges faster than Jacobi.
    D <- diagonal matrix of A
    L <- -lower_triangular(A) + D
    U <- -upper_triangular(A) + D
    T <- inverse(D - L) * U // Iteration matrix
    C <- inverse(D - L) * b
    x_old <- x0
    err <- 1000
    iter <- 0
    WHILE err > tol AND iter < nmax DO
        x_new <- T * x_old + C
        err <- ||x_old - x_new||
        x_old <- x_new
        iter <- iter + 1
    END WHILE
    RETURN (x_new, iter, err)

```

## 2.11 SOR Method

```

SOR(A, b, x0, omega, tol, nmax):
// Successive Over-Relaxation: introduces relaxation parameter omega.
// Optimal omega depends on matrix A; typically 1 < omega < 2.
    D <- diagonal matrix of A
    L <- -lower_triangular(A) + D
    U <- -upper_triangular(A) + D
    T <- inverse(D - omega*L) * ((1 - omega)*D + omega*U)

```

```

C <- omega * inverse(D - omega*L) * b
x_old <- x0
err <- 1000
iter <- 0
WHILE err > tol AND iter < nmax DO
    x_new <- T * x_old + C
    err <- ||x_old - x_new||
    x_old <- x_new
    iter <- iter + 1
END WHILE
RETURN (x_new, iter, err)

```

## 2.12 Forward Substitution

```

FORWARD_SUB(L, b):
// Forward substitution for lower triangular system Lx = b.
// Solves from first equation downward.
n <- number of rows of L
x <- vector of zeros size n
FOR i <- 0 TO n-1 DO
    IF |L[i][i]| < epsilon THEN
        RETURN ERROR "Zero diagonal element"
    END IF
    sum <- b[i]
    FOR j <- 0 TO i-1 DO
        sum <- sum - L[i][j] * x[j]
    END FOR
    x[i] <- sum / L[i][i]
END FOR
RETURN x

```

## 2.13 Backward Substitution

```

BACKWARD_SUB(U, b):
// Backward substitution for upper triangular system Ux = b.
// Solves from last equation upward.
n <- number of rows of U
x <- vector of zeros size n
FOR i <- n-1 DOWNTON 0 DO
    IF |U[i][i]| < epsilon THEN
        RETURN ERROR "Zero diagonal element"
    END IF
    sum <- b[i]
    FOR j <- i+1 TO n-1 DO
        sum <- sum - U[i][j] * x[j]
    END FOR
    x[i] <- sum / U[i][i]
END FOR
RETURN x

```

## 3 Interpolation

### 3.1 Vandermonde

```

VANDERMONDE(X, Y):
// Compute polynomial interpolation by solving Vandermonde system.
// Warning: Vandermonde matrices are ill-conditioned for large n.
    n <- length(X)
    V <- zero matrix nxn
    // Fill columns with powers of X (highest power first)
    FOR i <- 0 TO n-1 DO
        FOR j <- 0 TO n-1 DO
            V[i][j] <- X[i]^(n - j - 1)
        END FOR
    END FOR
    // Solve V * Coef = Y for polynomial coefficients
    Coef <- SOLVE(V, Y)
    RETURN Coef

```

### 3.2 Divided Differences

```

DIVIDED_DIFFERENCES(X, Y):
// Newton's divided differences to compute polynomial in Newton form.
// Produces coefficients for incremental polynomial evaluation.
    n <- length(X)
    D <- zero matrix nxn
    D[:,0] <- Y // First column is Y values
    // Compute divided differences
    FOR j <- 1 TO n-1 DO
        FOR i <- j TO n-1 DO
            D[i][j] <- (D[i][j-1] - D[i-1][j-1]) / (X[i] - X[i-j])
        END FOR
    END FOR
    // Newton coefficients are the diagonal of D
    Coef <- diagonal(D)
    RETURN Coef

```

### 3.3 Lagrange

```

LAGRANGE(X, Y):
// Lagrange interpolation: constructs basis polynomials L_i(x).
// Combines them with Y values to get final polynomial coefficients.
    n <- length(X)
    L <- zero matrix nxn
    FOR i <- 0 TO n-1 DO
        // Create list of X values excluding X[i]
        X_excl <- X without X[i]
        // Start polynomial (x - X[i])
        poly <- [1, -X_excl[0]]
        // Multiply polynomials to build basis numerator
        FOR j <- 1 TO n-2 DO
            poly <- CONVOLVE(poly, [1, -X_excl[j]])
        END FOR
        // Normalize so that L_i(X[i]) = 1
        denom <- EVALUATE(poly, X[i])
        FOR k <- 0 TO n-1 DO
            L[i][k] <- poly[k] / denom
        END FOR
    END FOR
    // Combine basis polynomials with Y values

```

```

Coef <- Y * L
RETURN (L, Coef)

```

### 3.4 Linear Spline

```

LINEAR_SPLINE(X, Y):
// Linear spline: piecewise linear segments between data points.
// Simplest form of spline interpolation.
n <- length(X)
Coef <- empty list
FOR i <- 0 TO n-2 DO
    IF X[i+1] = X[i] THEN
        RETURN ERROR "Duplicate x-values"
    END IF
    // Compute slope and y-intercept for segment i
    slope <- (Y[i+1] - Y[i]) / (X[i+1] - X[i])
    intercept <- Y[i] - slope * X[i]
    Coef.append([slope, intercept])
END FOR
RETURN Coef

```

### 3.5 Quadratic Spline

```

QUADRATIC_SPLINE(X, Y):
// Quadratic spline with continuity and smoothness conditions.
// Forms global linear system for coefficients.
n <- length(X)
m <- 3 * (n - 1)
A <- zero matrix mxm
b <- zero vector m
// Interpolation conditions
FOR i <- 0 TO n-2 DO
    row <- i + 1
    col <- 3 * i
    A[row][col] <- X[i+1]^2
    A[row][col+1] <- X[i+1]
    A[row][col+2] <- 1
    b[row] <- Y[i+1]
END FOR
A[0][0] <- X[0]^2
A[0][1] <- X[0]
A[0][2] <- 1
b[0] <- Y[0]
// Continuity of value at interior nodes
FOR i <- 1 TO n-2 DO
    row <- n - 1 + i + 1
    col <- 3 * i - 3
    A[row][col] <- X[i]^2
    A[row][col+1] <- X[i]
    A[row][col+2] <- 1
    A[row][col+3] <- -X[i]^2
    A[row][col+4] <- -X[i]
    A[row][col+5] <- -1
    b[row] <- 0
END FOR
// Smoothness: first derivative continuity

```

```

FOR i <- 1 TO n-2 DO
    row <- 2*n - 3 + i + 1
    col <- 3 * i - 3
    A[row][col] <- 2 * X[i]
    A[row][col+1] <- 1
    A[row][col+2] <- 0
    A[row][col+3] <- -2 * X[i]
    A[row][col+4] <- -1
    A[row][col+5] <- 0
    b[row] <- 0
END FOR
// Boundary condition
A[m-1][0] <- 2
b[m-1] <- 0
// Solve for coefficients
S <- SOLVE(A, b)
Coef <- empty list
FOR i <- 0 TO n-2 DO
    Coef.append(S[3*i : 3*i+2])
END FOR
RETURN Coef

```

### 3.6 Cubic Spline

```

CUBIC_SPLINE(X, Y):
// Cubic spline with natural boundary conditions.
// Enforces value, first and second derivative continuity.
n <- length(X)
m <- 4 * (n - 1)
A <- zero matrix mxm
b <- zero vector m
// Interpolation conditions
FOR i <- 0 TO n-2 DO
    row <- i + 1
    col <- 4 * i
    A[row][col] <- X[i+1]^3
    A[row][col+1] <- X[i+1]^2
    A[row][col+2] <- X[i+1]
    A[row][col+3] <- 1
    b[row] <- Y[i+1]
END FOR
A[0][0] <- X[0]^3
A[0][1] <- X[0]^2
A[0][2] <- X[0]
A[0][3] <- 1
b[0] <- Y[0]
// Value continuity at interior nodes
FOR i <- 1 TO n-2 DO
    row <- n - 1 + i + 1
    col <- 4 * i - 4
    A[row][col] <- X[i]^3
    A[row][col+1] <- X[i]^2
    A[row][col+2] <- X[i]
    A[row][col+3] <- 1
    A[row][col+4] <- -X[i]^3
    A[row][col+5] <- -X[i]^2
    A[row][col+6] <- -X[i]

```

```

A[row][col+7] <- -1
b[row] <- 0
END FOR
// First derivative continuity
FOR i <- 1 TO n-2 DO
    row <- 2*n - 3 + i + 1
    col <- 4 * i - 4
    A[row][col] <- 3 * X[i]^2
    A[row][col+1] <- 2 * X[i]
    A[row][col+2] <- 1
    A[row][col+3] <- 0
    A[row][col+4] <- -3 * X[i]^2
    A[row][col+5] <- -2 * X[i]
    A[row][col+6] <- -1
    A[row][col+7] <- 0
    b[row] <- 0
END FOR
// Second derivative continuity
FOR i <- 1 TO n-2 DO
    row <- 3*n - 5 + i + 1
    col <- 4 * i - 4
    A[row][col] <- 6 * X[i]
    A[row][col+1] <- 2
    A[row][col+2] <- 0
    A[row][col+3] <- 0
    A[row][col+4] <- -6 * X[i]
    A[row][col+5] <- -2
    A[row][col+6] <- 0
    A[row][col+7] <- 0
    b[row] <- 0
END FOR
// Natural boundary conditions
A[m-2][0] <- 6 * X[0]
A[m-2][1] <- 2
b[m-2] <- 0
A[m-1][m-4] <- 6 * X[n-1]
A[m-1][m-3] <- 2
b[m-1] <- 0
// Solve for coefficients
S <- SOLVE(A, b)
Coef <- empty list
FOR i <- 0 TO n-2 DO
    Coef.append(S[4*i : 4*i+3])
END FOR
RETURN Coef

```