```
function factorizaciónLUSimple(A, b)
	L, U = gauss_simple(A)
	z = resolver_matriz_traingular(U,b)
	x = resolver_matriz_triangular(L,z)
	return x
end

function factorizaciónLUConPivoteo(A)
	L, U, P = gauss_pivoteo_parcial(A)
	z = resolver_matriz_traingular(U, P*b)
	x = resolver_matriz_triangular(L, z)
	return x
end

function factorizacionDirecta(A)
	// Cholesky, Doolittle, Crout son similares
	L, U ← InicializaLU(n)
	for k = 1 to n do
		suma1 ← 0
		for p = 1 to k - 1 do
			suma1 ← suma1 + l_kp * u_pk
		end for
		l_kk * u_kk ← a_kk - su1
		for i = k + 1 to n do
			suma2 ← 0
			for p = 1 to k - 1 do
				suma2 ← suma2 + l_ip * u_pk
			end for
			l_ik = (a_ik - suma2)/u_kk
		end for
		for j = k + 1 to n do
			suma3 ← 0
			for p = 1 to k - 1 do
				suma3 ← suma3 + l_kp * u_pj
			end for
			u_kj = (a_kj - suma3)/l_kk
		end for
	end for
	return L, U
end function

function metodoJacobi(M, b, x0, tolerance, max_iterations)
	D, L, U = obtener_diagonales(M)
	T = inv(D)*(L + U)
	C = inv(D)*b
	i = 0
	err = inf
	while err > tolerance
	   if i >= max_iterations
	      print Se alcanzó el máximo número de iteraciones
	   end if
	   xn = T*x0 + C
	   err = max(abs(xn-x0))
	   x0 = xn
	   i = i + 1
	end while
	return xn
end function
```

```
function matrixGauss_s(M, b, x0, tolerance, max_iterations, w = 1)
        // para w = 1 es gauss seidel, para cualquier otro 0 < w < 2 es SOR
        D, L, U = obtener_diagonales(M)
        T = inv(D-w*L)*((1-w)*D + w*U)
        C = inv(D - w*L) * b
        err = inf
        i = 0
        while err > tolerance
           if i >= max_iterations
              print Se alcanzó el máximo número de iteraciones
           end if
           xn = T*x0 + C
           err = max(abs(xn-x0))
           x0 = xn
           i += 1
        end while
        return xn
end function

function matriz_de_vandermonde(x, f)
        // x son los puntos y f el valor en y
        // de cada punto
        n = len(x) // la cantidad de puntos
        mat // Matriz de nxn
        b // vector nx1
        for i = 1 to n
           # exponent
           exponent = n-1
           b_i = f(x[i])
           for j in range(n):
              mat_ij = x[i]^ex
              ex = ex - 1
           end for
        end for
        return matrix
end function

function diferencias_divididas(x, f)
        n = len(x) // Número de puntos
        mat // Matriz de nxn
        for i = 0 to n
           mat_i0 = f(x[i])
        end for
        for j = 1 to n
           for i = j to n
              mat_ij = (mat_(i-1)(j-1) - mat_(i)(j-1))/(x[i-j] - x[i])
           end for
        end for
        return mat
end function
```

```
def lineal_spline(x, f):
    n = len(x) // El numero de puntos
    mat // matriz de tamaño (2*n - 2, (n-1)*2)
    // La restricción de interpolación inicial
    mat_00 = x[0]
    mat_01 = 1

    // Añadir el resto de restricciones de interpolación
    j = 0
    for i = 1 to n-1
        mat_ij = x[i]
        mat_i(j+1) = 1
        j += 2

    // Añadir las restricciones de continuidad
    j = 0
    for i=1 to n-1:
        index = i + n - 1
        mat_(index)j = x[i]
        mat_(index)(j+1) = 1
        mat_(index)(j+2)  = -x[i]
        mat_(index)(j+3)  = -1
        j += 2

    // Construcción del vector y
    y = concatenar(f(x), 0*(n - 2))

    sol = solve(mat,y) // resolver el sistema

    return mat,y,sol


function cuadratic_spline(x, f)
    n = len(x) // El número de puntos
    mat = np.zeros((4*n - 4, (n-1)*4)) // matriz de tamaño 4*(n-1) , 4*(n-1)
    // La restricción de interpolación inicial
    m_00 = x[0]^2
    m_01 = x[0]
    m_02 = 1
    // Añadir las restricciones de interpolación
    j = 0
    for i = 1 to n-1
        mat_ij = x[i]^2
        mat_(i,j+1) = x[i]
        mat_(i,j+2) = 1
        j += 3

   // Añadir las restricciones de continuidad
    j = 0
    for i = 1 to n-1
        index = i + n - 1
        mat_indexj = x[i]^2
        mat_(index,j+1) = x[i]
        mat_(index,j+2) = 1

        mat_(index,j+3) = -x[i]^2
        mat_(index,j+4) = -x[i]
        mat_(index,j+5) = -1
```

```
        j += 1

    // Añadir las restricciones de la primera derivada
    j = 0
    for i = 1 to n-1
        index = i + n - 1
        mat_indexj = 2*x[i]
        mat_(index,j+1) = 1
        mat_(index,j+2) = 0

        mat_(index,j+3) = -2*x[i]
        mat_(index,j+4) = -1
        mat_(index,j+5) = 0
        j += 3

    // Añadir las condiciones de frontera
    mat_(3*n-4, 0)  = 2

    //Construcción del vector y
    // Concatenar los valores de f(x) para cada punto y agregar (2*n 3) ceros
    y = concatenar(f(x), 0*(2*n - 3))

    sol = solve(mat,y) // resolver el sistema

    return mat,y,sol
end function

function cubic_spline(x, f)
    n = len(x) // El número de puntos
    mat // matriz de tamaño 4*(n-1) , 4*(n-1)
    // La restricción de interpolación inicial
    mat_00 = x[0]^3
    mat_01 = x[0]^2
    mat_02 = x[0]
    mat_03 = 1
    // Añadir las restricciones de interpolación
    j = 0
    for i = 1 to n-1
        mat_ij = x[i]^3
        mat_(i,j+1) = x[i]^2
        mat_(i,j+2) = x[i]
        mat_(i,j+3) = 1
        j += 4

    // Añadir las restricciones de continuidad
    j = 0
    for i = 1 to n-2
        index = i + n - 1
        mat_ij = x[i]^3
        mat_(index,j+1) = x[i]^2
        mat_(index,j+2) = x[i]
        mat_(index,j+3) = 1

        mat_(index,j+4) = -x[i]^3
        mat_(index,j+5) = -x[i]^2
        mat_(index,j+6) = -x[i]
        mat_(index,j+7) = 1
        j += 4
```

```
    // Añadir las restricciones de la primera derivada
    j = 0
    for i = 1 to n-2
        index = i + n - 1
        mat_indexj = 3*x[i]^2
        mat_(index,j+1) = 2*x[i]
        mat_(index,j+2) = 1
        mat_(index,j+3) = 0

        mat_(index,j+4) = -(3*x[i]^2)
        mat_(index,j+5) = -(2*x[i])
        mat_(index,j+6) = -1
        mat_(index,j+7) = 0
        j += 4

    // Añadir las restricciones de la segunda derivada
    j = 0
    for i = 1 to n-2
        index = i + n - 1
        mat_indexj = 6*x[i]
        mat_(index,j+1) = 2
        mat_(index,j+2) = 0
        mat_(index,j+3) = 0

        mat_(index,j+4) = -(6*x[i])
        mat_(index,j+5) = -(2)
        mat_(index,j+6) = 0
        mat_(index,j+7) = 0
        j += 4

   // Añadir las condiciones de frontera
    mat_(4*n-6, 0)  = 6
    mat_(4*n-6, 1)  = 2
    mat_(4*n-5, -4)  = 6
    mat_(4*n-5, -3)  = 2

    // Construcción del vector y
    // Concatenar los valores de f(x) para cada punto y agregar (3*n -4) ceros
    y = concatenar(f(x), 0*(3*n - 4))

    sol = solve(mat,y) // resolver el sistema

    return mat,y,sol
end function
```