PSEUDOCÓDIGO ALGORITMO BISECCIÓN

```
biseccion(f, a, b, tol)  → puede devolver varias cosas { c
                                                          err
                                                          nIter
     cinargin?
    if f(a)*f(b) > 0
    |    error;  → intervalo no apto
    end

    nIter = 0;
    err = abs(b-a);
    if(err < tol)
    |    error;  → error inicial menor que la tolerancia requerida
    end

    if f(a) < eps
    |    c = a;
    |    err = a;
    |    return;
    end

    c = (a+b)/2;
    nIter++;  → primera iteración
    if abs(f(c)) < eps
    |    err = abs(f(c))
    |    return
    end

    while err > tol
    |    c = (a+b)/2;
    |    nIter++;
    |    if abs(f(c)) < eps   → ¿solución encontrada?
    |    |    return
    |    end
    |    if f(c)*f(a) < 0     } buscamos el intervalo
    |    |    b = c;          } bueno para la siguiente
    |    else                 } iteración
    |    |    a = c;
    |    end
    |    err = abs(b-a);
    end
end
```

PSEUDOCODIGO ALGORITMO PUNTO FIJO

Existen dos variantes para la implementación del algoritmo del punto fijo: en una nos dan la g y en otra la f:

1. Nos dan la $f \to f(x)=0$

```
puntoFijo(f, x0, λ, tol, maxI)   → dev [c, err, iter]
    g = @(x) x - λ*f(x);   → conseguimos g(x) = x
    ¿nargin?
    iter=0;
    err = 100;
    while err>tol & iter<maxI
        iter++;
        x1 = g(x0);
        err = abs(x1-x0);
        x0 = x1;
    end
    c = x1;
end
```

2. Nos dan la $g \to g(x)=x$

```
puntoFijo(g, x0, tol, maxI)   → dev [c, err, iter]
    ¿nargin?
    iter=0;
    err = 100;
    while err>tol & iter<maxI
        iter++;
        x1 = g(x0);
        err = abs(x1-x0);
        x0 = x1;
    end
    c = x1;
end
```

PSEUDOCÓDIGO MÉTODO DE NEWTON

```
metodoNewton (f, df, x0, tol1, tol2, maxI)   → dev [root, err, iter]
```

df → también con diff(f) o a mano pq f==polino

```
    ¿nargin?
    iter=1;  → 1 iter fuera del bucle
    F = f(x0);
    D = df(x0);

    if D==0
    |   error
    end

    err = abs(F/D);

    while abs(F) > tol1 & err > tol2 & iter < maxI
    |   iter++;
    |   F = f(x0);
    |   D = df(x0);
    |   if D==0
    |   |   error        } derivada en x0 ==0
    |   end
    |   x0 = x0 - (F/D);
    |   err = abs(F/D);
    end
    root = x0;  → podríamos devolver x0 directamente
end
```

PSEUDOCÓDIGO MÉTODO DE LA SECANTE

```
metodoSecante(f, x1, x0, tol1, tol2, maxI) -> dev [root, err, iter]

    ¿nargin?
    iter = 1;   -> 1 fuera del bucle
    N = (x1 - x0) * f(x1);
    D = f(x1) - f(x0);
    if D == 0
    |     error
    end
    err = abs(f(x1));
    while err > tol1 & abs(N/D) > tol2 & iter < maxI
                        -> 2 tol -> ¿2 error?
    |     iter++;
    |     N = (x1-x0) * f(x1);
    |     D = f(x1) - f(x0);
    |     if D == 0  }
    |     |    error } derivada igual a cero
    |     end        }
    |     x2 = x1 - (N/D);
    |     x0 = x1;
    |     x1 = x2;
    |     err = abs(f(x1));   (*) podríamos
    end                           devolver los
                                  dos errores
    root = x1;   -> podríamos devolver
                    x1 directamente
end
```

PSEUDOCÓDIGO FACTORIZACIÓN LU

```
factLU(A) —> dev [L, U, err]
    r,c = size(A)
    if r~=c
        error;
    end

    n=r;
    U=A;
    L=eye(n);

    for K=1:n-1
        if U(K,K) == 0
            error;   % e.g. ('U(K,K) igual a cero al paso %d', K)
        end

        for i=K+1:n
            L(i,K) = U(i,K) / U(K,K);

            %MÉTODO CON FOR%
            for j=K:n
                U(i,j) = U(i,j) - (L(i,K) * U(K,j));
            end

            % VECTORIZACIÓN %
            U(i,K:n) = U(i, K:n) - L(i,K)* U(K,K:n);

        end
    end
```

for j=K:n para fact. LU
for j=K+1:n si solo queremos resolver un sist.

```
    % OBSERVACIÓN %
    % Podríamos hacer el for de j = K+1:n
    % U no sería triangular alta, pero podríamos usar la función
    % U = triu(U); en este sitio del pseudocódigo, así po-
    % dríamos versatilizar esta función.

    err = max(max(abs(A- L*U)));

end
```

PSEUDOCÓDIGO FACTORIZACIÓN PLU

```
factLUPivotaje(A) ──▷ dev [P, L, U, err]
    r,c = size(A);
    if r~=c
    |    error
    end

    n = r;
    L = eye(n);
    U = A;
    P = eye(n);

    for K = 1:n
        [~, pos] = max(abs(U(K:n, K)));
        if pos ~= 1
            aux = U(K,K:n);                              Cambiando
            U(K,K:n) = U(pos+K-1, 1:K-1);                filas en U
            U(pos+K-1, 1:K-1) = aux;

            aux = L(K,1:K-1);                            Cambiando
            L(K,1:K-1) = L(pos+K-1, 1:K-1);              filas en L
            L(pos+K-1, 1:K-1) = aux;

            aux = P(K,:);                                Cambiando
            P(K,:) = P(pos+K-1, :);                      filas en P
            P(pos+K-1, :) = aux;
        end
        for i = K+1:n
            L(i,K) = U(i,K)/U(K,K);
        1.{ for j = K:n                                  1. for type
          |     U(i,j) = U(i,j) - (L(i,K) * U(K,j));
          end
        2.{ U(i,K:n) = U(i,K:n) - L(i,K)*U(K,K:n);      2. vecto-
                                                            rization
        end
    end
    err = max(max(abs(P*A - L*U)));
end
```

PSEUDOCÓDIGO RESOLVER SISTEMA LU

```
solveSistLU(A, b) ---> dev [X, err]
    [L, U, ~] = factLU(A);
    n = length(A);
    (*) ---> b fila o columna CHECKING
    y = b;
    for i = 2:n
        1. { for j = 1:i-1
           |     y(i) = y(i) - L(i,j) * y(j);      for mode
           { end
        2. { y(i) = y(i) - L(i, 1:i-1) * y(1:i-1);      vectorization
    end

    X = y./diag(U);
    for i = n:-1:1
        1. { for j = i+1:n
           |     X(i) = X(i) - ((U(i,j) * X(j)) / U(i,i));      for mode
           { end
        2. { X(i) = X(i) - ((U(i, i+1:n) * X(i+1:n)) / U(i,i));      vectorizatio
    end
                                  ¿b es fila o columna?
    err = max(abs(A * X - b));
end
```

PSEUDOCÓDIGO RESOLVER SISTEMA PLU

```
solveSistLUPivot(A,b) ---> dev[X,err]
    [P,L,U,~] = factLUPivotaje(A);
    n= length(A);
    (*) ---> b fila o columna CHECKING
    y= P*b;
    resto (2 fors) igual a lo anterior
end
```

PSEUDOCÓDIGO MÉTODO DE JACOBI

metodo Jacobi (A, b, tol, iterMax) → dev [x, err, nIter]

```
    m,n = size(A);
    l = length(b);
    if m~=n    error;  → matriz A no cuadrada
    if m~=l    error;  → m!=l  nº filas de A != filas b
```

(*) CHECKINGS de b (¿vector? ¿columna o fila?

(*) CHECKING A es matriz dominada por la diagonal →

```
for i = 1:n
    aux = 0
    d = abs(A(i,i));
    for j = 1:n
        aux += abs(A(i
    end
    aux -= d;
    if aux >= d
        warning;
        break;
    end
end
```

```
    err = 1000;
    nIter = 0;
    x = ones(l,1);
```

$D = diag(diag(A)); \equiv \begin{pmatrix} d1 & 0 & 0 \\ 0 & d2 & 0 \\ 0 & 0 & d3 \end{pmatrix}$

$D1 = diag(1./diag(A)); \equiv \begin{pmatrix} 1/d1 & 0 & 0 \\ 0 & 1/d2 & 0 \\ 0 & 0 & 1/d3 \end{pmatrix}$

```
    while (err > tol & nIter < iterMax)
        nIter++;
        x1 = D1*(b-(A-D)*x);          ] ITERACIÓN JACOBI
        err = max(abs(x1-x));         → norma infinito
        x = x1;
    end
end
```

norma infinito

También valdría la norma 2:

err = norm(A*x - b);

PSEUDOCÓDIGO MÉTODO DE GAUSS-SEIDEL

```
metodoGaussSeidel (A, b, tol, maxIter) → dev[x, err, nIter]

    m,n = size(A);
    l = length(b)
    if m ~= n    error;  ] → mismos errores Jacobi
    if m ~= l    error;  ]
     * CHECKING del vector b

     * CHECKING MATRIZ DOMINADA A

    err = 1000;
    nIter = 0;
    x = ones(l, 1);
    x1 = x;

    while err > tol && nIter > maxIter
    |     nIter++;
    |     for i = 1:n
    |     |     sumat1 = 0;
    |     |     sumat2 = 0;
    |     |     for j = 1:i-1
    |     |     |     sumat1 = sumat1 + A(i,j) * x1(j);
    |     |     end
    |     |
    |     |     for j = i+1:n
    |     |     |     sumat2 = sumat2 + A(i,j) * x(j);
    |     |     end                       → b tiene que ser vector colum
    |     |                                                       ] ITERACIÓ
    |     |     x1(i) = (b(i) - sumat1 - sumat2) / A(i,i);       ] GAUSS-SEID
    |     end
    |     err = max(abs(x1-x));  → norma infinito
    |                                Norma 2:
    |     x = x1;                    err = norm(A*x - b);
    end

end
```