Resuelve la cinemática inversa para alcanzar una posición deseada

`[q_sol, p_sol] = cinematica_inv(robot, p_des, tol, max_iter, alpha, numMuestras)` utiliza muestreo ortogonal para generar numSamples candidatos en el espacio articular (dentro de los límites robot.qMin y robot.qMax). Luego, para cada candidato se refina la solución mediante un descenso del gradiente (usando la pseudoinversa del Jacobiano) hasta que el error de posición sea menor que tol o se alcance el número máximo de iteraciones. Si algún candidato cumple la tolerancia, se retorna inmediatamente.

Entradas:

- r - Estructura del robot, que debe contener:
- `.NGDL` : Número de juntas (n).
- `.qMin` : Límites inferiores (nx1).
- `.qMax` : Límites superiores (nx1).
- `.q` : Configuración articular actual (nx1).
- `.A0` : Matriz base (4x4).
- `p_des` - Posición deseada del efector final (3x1).
- `tol` - Tolerancia para el error en la posición (ej. 1e-3).
- `max_iter` - Número máximo de iteraciones de refinamiento para cada candidato.
- `alpha` - Ganancia para el descenso del gradiente.
- `numSamples` - Número de muestras a generar mediante muestreo ortogonal.

Salidas:

- `q_sol` - Configuración articular final (nx1).
- `p_sol` - Posición del efector final correspondiente (3x1).

Cabe destacar que también contiene un poco de código que pueden usar para obtener la otientación, pero solo es útil en robots con configuraciones específicas o con más grados de libertad ya que suele

```
function [q_sol, p_sol] = cinematica_inv(r, p_des, tol, max_iter, alpha, numMuestras)
%     lambda = 0.1;   % Importancia del error de orientación
n = r.NGDL;
```

Para poder facilitar

Generar muestras usando un muestreo ortogonal. Si les interesa, pueden ver en wikipedia lo que dice, pero básicamente, genera valores aleatorios distribuidos de forma algo uniforme

```
X = olhsdesign(n, numMuestras);
```

Escalar las muestras a los límites de cada junta:

$$q_{\text{candidatas}_{n\times m}} = q_{\min_{n\times 1}} + \text{diag}(q_{\max} - q_{\min})_{n\times n} X_{n\times m}$$

Ahora, para facilitar las cosas, se ordenan las $q_{\text{candidatas}}$ desde la más cercana hasta la más lejana

```
for j = 1:numMuestras
    q_candidata = q_candidatas(:, j);
```

```
    % Actualizar el robot con la configuración candidata y extraer posición
    % actual
    e_p = p_des - p_actual;
    errors(j) = norm(e_p);
end
% Ordenar puntos por sus errores de menor a mayor
[errors, idx] = sort(errors);
q_candidatas = q_candidatas(:, idx);
```

Ahora se calcula el descenso del gradiente de cada $q_{\text{candidatas}}$

```
% Bucle externo: iterar sobre cada candidato
for j = 1:numMuestras
    i = 0;
    q_candidata = q_candidatas(:, j);
    % Actualizar el robot con la configuración candidata y calcular error
    % de posición
    e_p = p_des - p_actual;

    % Bucle interno: refinamiento local con descenso del gradiente
    while (errors(j) > tol) && (i < max_iter)
        % Calcular el Jacobiano geométrico (usamos solo la parte traslacional en este ejemplo)
        [Jv, ~] = completar;
```

Actualización usando la pseudoinversa del jacobiano:

$$e_p \approx \frac{\partial p}{\partial q} e_q \Longrightarrow e_q \approx \alpha \frac{\partial p}{\partial q}^{+} e_p$$

$$q_{\text{candidata},\, i} = q_{\text{candidata},\, i-1} + e_q$$

```
        % Saturar q_candidata para que se mantenga dentro de los límites
        q_candidata = saturar(q_candidata, r.qMin, r.qMax);

        % Actualizar el robot y recalcular el error de posición
        completar;
        completar;
        completar;
        errors(j) = norm(e_p);
        i = i + 1;
    end

    q_candidatas(:, j) = q_candidata;
```

Si algún candidato alcanza la tolerancia, se retorna inmediatamente

```
    if errors(j) <= tol
        q_sol = q_candidata;
        r = actualizar_robot(r, q_sol);
        p_sol = r.T(1:3,4,end);
```

```
        if coder.target('MATLAB')
            fprintf('Solución alcanzada en muestra %d, iteraciones %d, error = %e\n', j, i, err
        end
        return;
    end
end

% Si ningún candidato alcanza la tolerancia, se elige el que tenga el menor error.
[~, idx_best] = min(errors);
q_sol = q_candidatas(:, idx_best);
r = actualizar_robot_completo(r, q_sol);
p_sol = r.T(1:3,4,end);
if coder.target('MATLAB')
    fprintf('Ningún candidato alcanzó la tolerancia; se elige el candidato con error = %e\n', m
end
```