# Implementation of the Jacobian Matrix in Code

## Inputs and Preprocessing

• Sensor coordinates: sensors\_m with shape (N,3).  
• Parameters: [cx, cy, cz, ux, uy, uz, scale] (7 total).  
• Mesh: local coordinates and areas of the two end faces (xL, yL, zL) and dS of length Npts.  
• Orientation normalization: u\_raw → u\_hat = u\_raw/||u\_raw||. A local orthonormal basis (e1, e2, e3) is constructed.  
• Basis derivatives: functions d\_basis\_dalpha and d\_basis\_dbeta provide de1, de2, de3.  
• Chain rule: compute dα/du\_raw and dβ/du\_raw using intermediate dα/du\_hat, dβ/du\_hat and d u\_hat/d u\_raw.  
• Numerical safeguards: lower bounds for sinα and |r|.

## Kernel Function and Derivatives

• kernel\_and\_J(r): computes f = r/|r|^3 (3×1) and Jr = (|r|^2 I - 3 r r^T)/|r|^5 (3×3).  
• Weight factor: w = (μ0/(4π)) \* (Br/μ0) \* scale \* sign \* dS[k].  
• sign = +1 for the top face and -1 for the bottom face.

## Main Loop Structure

For each end face, for each sensor, for each mesh cell:  
1. Compute world coordinates of the surface element: P = c + e1\*xL + e2\*yL + e3\*zL.  
2. Compute r = s[i] - P.  
3. Compute f and Jr using kernel\_and\_J.  
4. Accumulate B, JB\_c, JB\_da, JB\_db using weights and derivatives.  
5. After looping over elements, assign the Jacobian block for this sensor:  
 - Columns 0–2: JB\_c (center derivatives)  
 - Columns 3–5: orientation contributions via chain rule  
 - Column 6: scale contributions

## Pseudocode

Algorithm 1: Analytic Jacobian for Cylindrical Magnet  
  
Inputs: sensors s[1..N], mesh cells k=1..Npts with (xL,yL,zL,dS), params θ=(c,u\_raw,scale), geom(Br)  
Output: B[3N], J[(3N)×7]  
  
u\_hat ← normalize(u\_raw)  
(e1,e2,e3), (de1\_da,de2\_da,de3\_da), (de1\_db,de2\_db,de3\_db) ← basis\_from(u\_hat)  
dα/du\_raw, dβ/du\_raw ← chain(u\_hat, u\_raw)  
  
for end in {+1, -1}:  
 sign ← end  
 for i in 1..N:  
 Bi ← (0,0,0); JB\_c ← 0\_{3×3}; JB\_a ← (0,0,0); JB\_b ← (0,0,0)  
 for k in 1..Npts:  
 P ← c + e1\*xL\_k + e2\*yL\_k + e3\*zL\_k  
 r ← s\_i - P  
 (f, Jr) ← kernel\_and\_J(r) # f∈R^3, Jr∈R^{3×3}  
 w ← (μ0/(4π))\*(Br/μ0)\*scale\*sign\*dS\_k  
 Bi ← Bi + w\*f  
 JB\_c ← JB\_c - w\*Jr # dB/dc  
 dP\_da ← de1\_da\*xL\_k + de2\_da\*yL\_k + de3\_da\*zL\_k  
 dP\_db ← de1\_db\*xL\_k + de2\_db\*yL\_k + de3\_db\*zL\_k  
 JB\_a ← JB\_a + w\*(Jr\*(-dP\_da)) # dB/dα  
 JB\_b ← JB\_b + w\*(Jr\*(-dP\_db)) # dB/dβ  
 B\_i ← B\_i + Bi  
 J\_i[:,0:3] ← JB\_c  
 J\_i[:,3:6] ← JB\_a ⊗ (dα/du\_raw) + JB\_b ⊗ (dβ/du\_raw)  
 J\_i[:,6] ← Σ\_k (μ0/(4π))\*(Br/μ0)\*sign\*dS\_k \* f(r\_k) # scale column  
return B, J  
  
Notes:  
- Guard |r|² with +1e-30; guard sinα with ≥ 1e-12.  
- Reuse f for the scale column; avoid recomputing kernel twice.  
- Optional: add tiny Tikhonov regularization on (ux,uy,uz) in the optimizer.

## Block Structure of the Jacobian

For one sensor: a 3×7 Jacobian block. For an array: stack all blocks vertically to form a (3N × 7) matrix.  
Columns correspond to: center (3), orientation vector (3), and scale (1).

## Code Correspondence

Python: kernel\_and\_J, sph\_basis\_from\_u, d\_basis\_dalpha, d\_basis\_dbeta, uhat\_and\_chain, ForwardModel.compute\_B\_and\_J.  
C++: kernel\_and\_J, sph\_basis\_from\_u, d\_basis\_dalpha, d\_basis\_dbeta, arrays d\_uhat\_duraw, dalpha\_duraw, dbeta\_duraw, accumulators JBc[9], JB\_a[3], JB\_b[3], and final write into out.J.

## Complexity and Robustness

Time complexity scales with the product of sensor count and mesh cells. Numerical safeguards are applied to avoid divisions by very small distances and near-singular angle terms.