Description of the Physics-Informed Neural Network Model

Overview

The implemented model is a Physics-Informed Neural Network (PINN) designed to recover the thermodynamic and magnetic structure of the solar atmosphere from sparse observational constraints. It enforces physical laws of magnetohydrodynamics (MHD) alongside observational misfits to guide training.

Inputs and Outputs

The model receives as input the spatial coordinates (x, y, z) arranged in a 3D grid. The neural network predicts eight output channels: density (ρ), temperature (T), magnetic field magnitude (|B|), inclination (via two channels), azimuth (via two channels), and resistivity (η). These are reshaped into 3D volumes for further physical processing. From the magnetic field magnitude, inclination, and azimuth, the model reconstructs Cartesian components (Bx, By, Bz).

Physical Laws

The model enforces several physical principles:

- Force balance (MHD equilibrium):
  $\nabla p - J \times B + \rho g \approx 0$
  with $J = (\nabla \times B)/\mu_0$ and $p = \rho R_a T$.

- Divergence-free condition: $\nabla \cdot B \approx 0$.

- Energy balance along the magnetic field:
  $\nabla \cdot q + Q_{heat} + Q_{Joule} - Q_{rad} \approx 0$,
  where q is anisotropic Spitzer conduction along B, Q_heat is a phenomenological logistic heating term, $Q_{Joule} = \eta |J|^2$ is Joule heating, and Q_rad represents radiative losses (including synthetic EUV channel responses at 171, 193, and 335 Å).

Observational Constraints

The model includes observational misfits in its loss function:

- EUV intensities (171, 193, 335 Å) obtained by integrating radiative loss terms along z.
- Magnetic slice constraints at a given plane: observed B, inclination, and azimuth compared to model slices.
- Temperature image misfit on the same reference plane.

Residuals and Loss Function

Residuals are constructed from both physical equations and observational constraints:

- R0: Force balance residual.
- R1: Divergence of B.
- R3–R5: EUV intensity misfits.

• R6–R8: Slice constraints for Bx, inclination, azimuth.
• R9: Temperature misfit.
• R10: Energy balance residual.

Each residual is compared to zero using mean squared error (MSE). The total loss is a weighted sum of these contributions, with weights defined in opts.w.

Normalization and Numerical Treatment

To ensure balanced optimization, residuals undergo automatic normalization based on an exponential moving average of their RMS values, with warmup and clipping options. The model operates in SI units, converting coordinates from megameters to meters, and magnetic fields can be expressed in Tesla or Gauss. Spatial derivatives are calculated with mixed finite differences, respecting the non-uniform z grid. The framework is GPU-aware and uses dlarray to ensure compatibility with deep learning operations.

Remarks

• The framework is flexible and modular, allowing easy inclusion or exclusion of physical and observational terms.

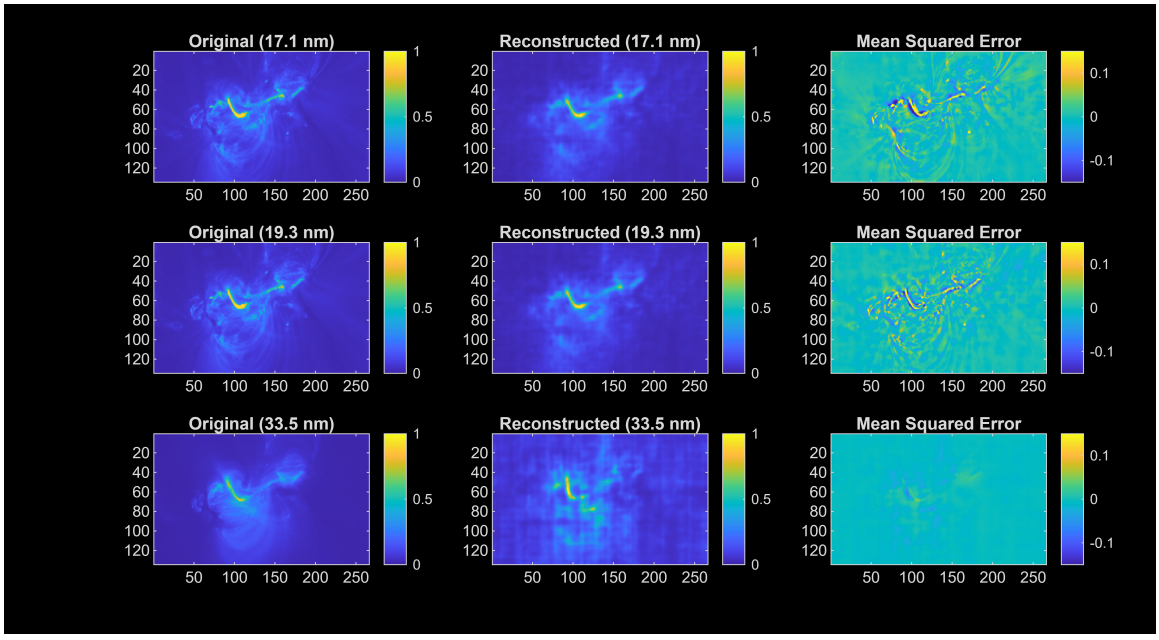

Figure 1: Comparison of the EVU observations and model.

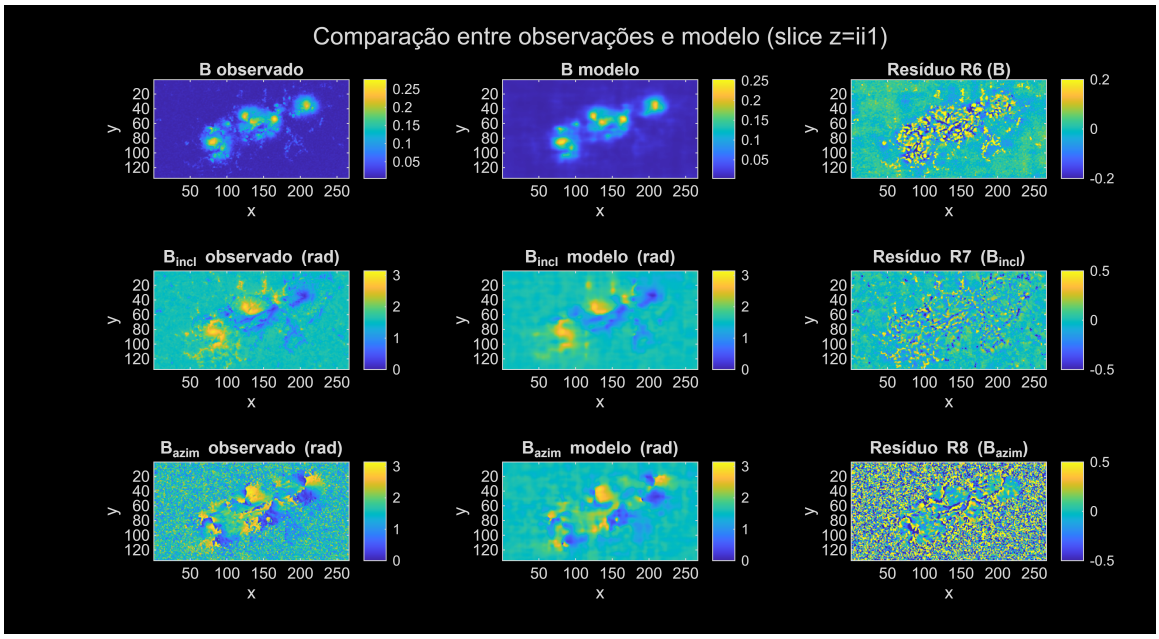

Figure 2: Comparison of the Magnetic field (HMI) observations and model.

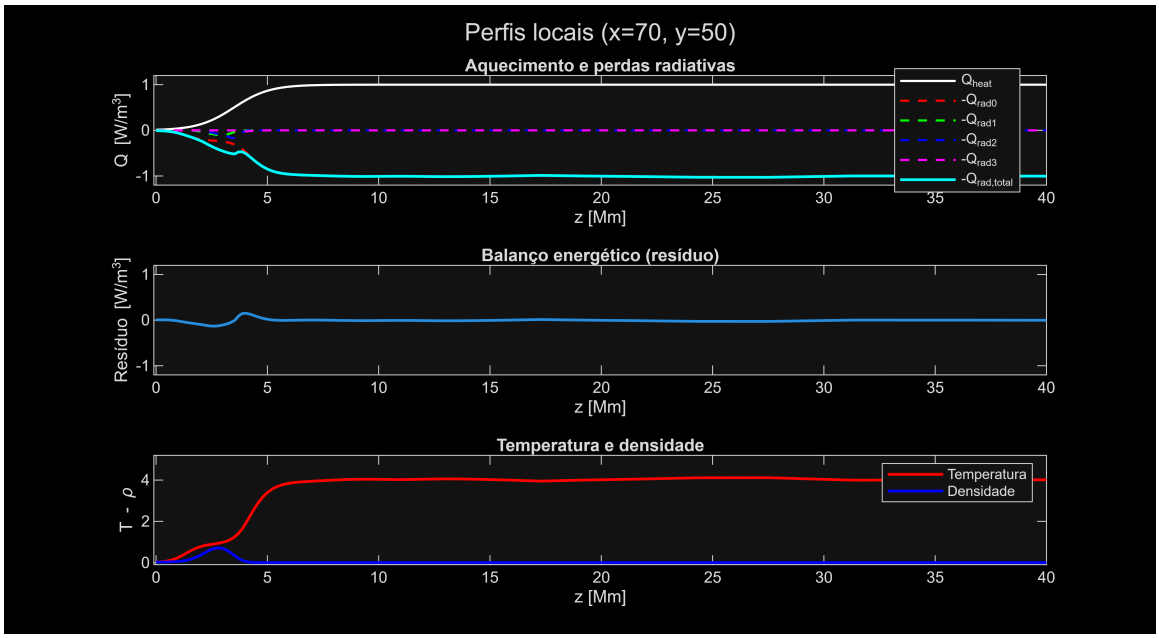

Figure 3: Profile of the energy balance terms, Temperature and density.

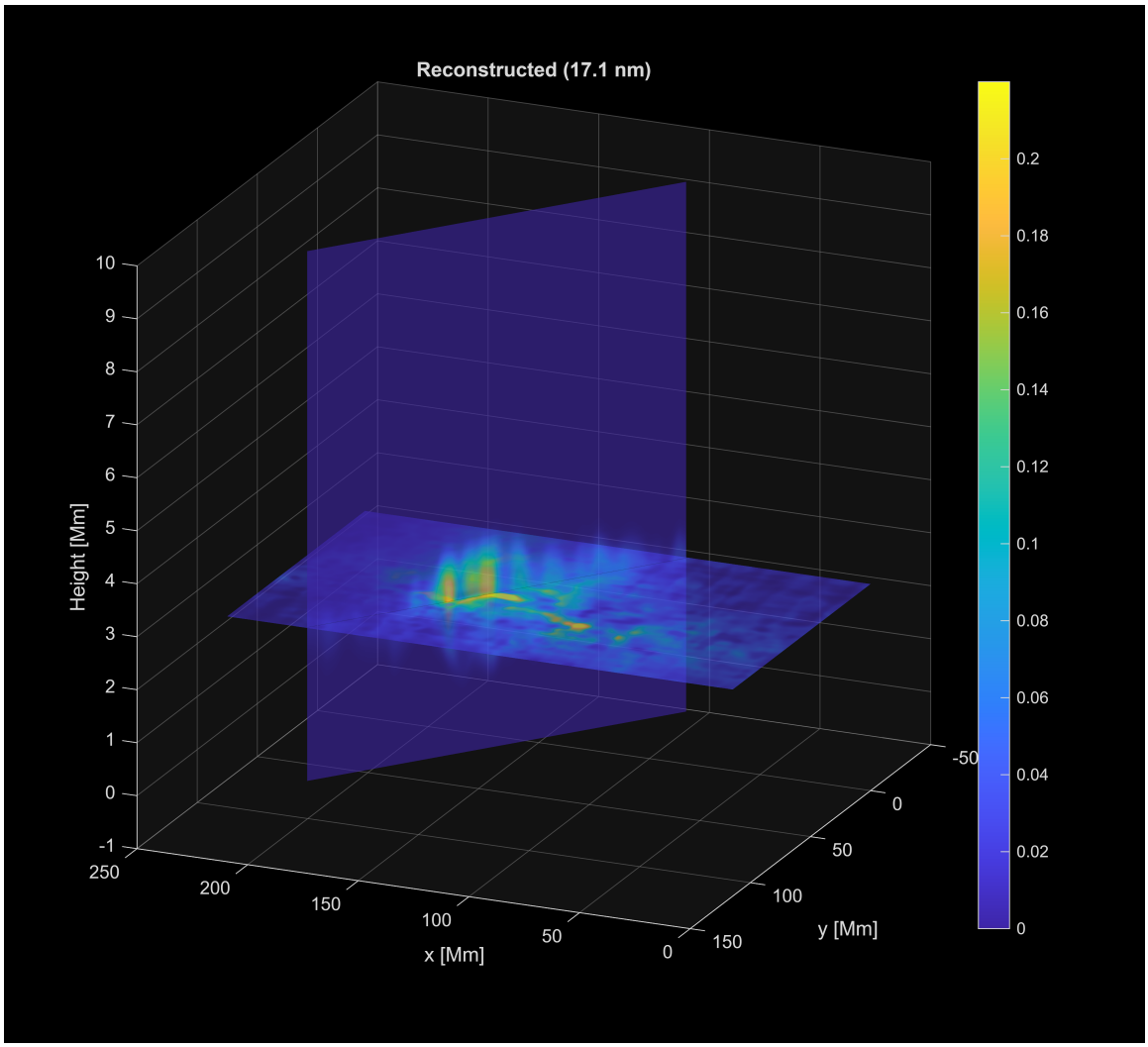

Figure 4: 3D emission in EVU (17.1 nm)

```
function [loss, a, gradients, R3, R4, R5, R6, R7, R8, R9, R10, I1, I2, I3, ...
          Qheat, Qrad0, Qrad1, Qrad2, Qrad3, pVol, divBVol, res_energy, ...
          BxVol, ByVol, BzVol, BVol, BinclVol,BazimVol, opts] = ...
    modelLossTBQ_v04(net, X, Bx1, By1, Bz1, x3g, y3g, z3g, Ic3g, Nx, Ny, Nz,
opts)
% modelLossTBQ_v03 — PINN loss with nonuniform-Z support (coords in Mm),
%                    SI physics (μ0, g, Spitzer κ∥), GPU awareness, and EMA
norm.

%Y    = forward(net, X);     % dlarray, (channels x batch)

%% --------------------------- defaults & options
%if nargin < 15 || isempty(opts), opts = struct; end
if ~isfield(opts,'w'),            opts.w = struct; end
if ~isfield(opts,'dx'),           opts.dx = 1; end      % Mm (uniform X step
if X3 absent)
if ~isfield(opts,'dy'),           opts.dy = 1; end      % Mm (uniform Y step
if Y3 absent)
if ~isfield(opts,'dz'),           opts.dz = 1; end      % (unused for Z; Z is
always nonuniform)
if ~isfield(opts,'forceModel'),   opts.forceModel = 'full'; end
if ~isfield(opts,'Bunit'),        opts.Bunit = 'T'; end % 'T' or 'G'
if ~isfield(opts,'device'),       opts.device = 'cpu'; end

% normalization defaults
if ~isfield(opts,'norm'), opts.norm = struct; end
nd = opts.norm;
if ~isfield(nd,'enable'),        nd.enable        = true;   end
if ~isfield(nd,'beta'),          nd.beta          = 0.99;   end
if ~isfield(nd,'warmup'),        nd.warmup        = 50;     end
if ~isfield(nd,'clip'),          nd.clip          = [1e-2, 1e2]; end
if ~isfield(nd,'freezeIters'),   nd.freezeIters   = inf;    end
if ~isfield(nd,'stats'),         nd.stats         = struct; end
opts.norm = nd;

% gravity options
if ~isfield(opts,'gravity'), opts.gravity = struct; end
gv = opts.gravity;
if ~isfield(gv,'mode'), gv.mode = 'constant'; end          % 'constant' |
'radial'
if ~isfield(gv,'g0'),   gv.g0   = 274*1e-6;         end          % m/s^2
(solar photosphere)
opts.gravity = gv;

% weights
defw = struct('force',1e-5,'divB',2,'I171',10,'I193',10,'I335',1, ...
              'Bx',10,'By',1,'Bz',1e-1,'Timg',1e-2,'energy',1);
w = opts.w; fn = fieldnames(defw);
for k=1:numel(fn), if ~isfield(w,fn{k}), w.(fn{k}) = defw.(fn{k}); end, end
opts.w = w;

% device
% if strcmpi(opts.device,'gpu')
%     X = dlarray(gpuArray(extractdata(X)));
% else
```

```
%     X = dlarray(extractdata(X));
% end

%% -------------------------- forward pass

Y    = forward(net, X);     % dlarray, (channels x batch)
rho  = Y(1,:);
T    = Y(2,:);
Bmag = Y(3,:);
Bincl = atan2(Y(4,:), Y(5,:)); % (-π,π]
Bazim = atan2(Y(6,:), Y(7,:)); % (-π,π]
eta = Y(8,:);

ii = find(isnan(extractdata(Y)));
%fprintf('length(ii) = %d\n', length(ii))



% Volumes (Nx x Ny x Nz)
xVol   = reshape(X(1,:), Nx, Ny, Nz);     % coords in Mm (as fed to the
network)
yVol   = reshape(X(2,:), Nx, Ny, Nz);     % coords in Mm (as fed to the
network)
zVol   = reshape(X(3,:), Nx, Ny, Nz);     % coords in Mm (as fed to the
network)

TVol   = reshape(T(1,:),   Nx, Ny, Nz);
rhoVol = reshape(rho(1,:), Nx, Ny, Nz);
BVol   = reshape(Bmag(1,:),Nx, Ny, Nz);
BinclVol = reshape(Bincl(1,:), Nx, Ny, Nz);
BazimVol = reshape(Bazim(1,:), Nx, Ny, Nz);
etaVol   = reshape(eta(1,:),   Nx, Ny, Nz);

% Convert to Cartesian B
[BxVol, ByVol, BzVol] = hmi_vector_to_cartesian(BVol, BinclVol, BazimVol);

%ii = find(isnan(extractdata(BxVol)));
%fprintf('length(ii) = %d\n', length(ii))

likeRef = TVol;

%% -------------------------- SI constants & unit handling
mu0  = 4*pi*1e-7;     % H/m
epsv = 1e-25;         % numeric epsilon
Mm2m = 1e6;           % m per Mm

% B units: allow Gauss -> Tesla
if strcmpi(opts.Bunit,'G')
    sG = 1e-4; % T/G
    BxVol = BxVol * sG; ByVol = ByVol * sG; BzVol = BzVol * sG;
end

% Ideal gas per unit mass: p = ρ * R_a * T
kB = 1.380649e-23;  m_p = 1.6726219e-27;  mu_mean = 0.6;
Ra = kB / (mu_mean*m_p); % ~1.38e4 J/(kg·K)
```

```
pVol = TVol .* rhoVol .* Ra;

%ii = find(isnan(extractdata(pVol)));
%fprintf('length(ii) = %d\n', length(ii))

%% -------------------------- enforce nonuniform Z (always)
% X3 = []; Y3 = [];
% if isfield(opts,'grid') && isfield(opts.grid,'Z3') && ~isempty(opts.grid.Z3)
%     Z3 = castLike(opts.grid.Z3, likeRef);      % Possibly Ny x Nx x Nz or Nx
x Ny x Nz (Mm)
% else
%     Z3 = castLike(zVol, likeRef);              % fallback from input (Mm)
% end
% % Optional nonuniform X/Y
% if isfield(opts,'grid') && isfield(opts.grid,'X3') &&
~isempty(opts.grid.X3), X3 = castLike(opts.grid.X3, likeRef); end
% if isfield(opts,'grid') && isfield(opts.grid,'Y3') &&
~isempty(opts.grid.Y3), Y3 = castLike(opts.grid.Y3, likeRef); end

% Accept Ny x Nx x Nz: permute to Nx x Ny x Nz and permute fields
% if isequal(size(Z3), [Ny,Nx,Nz])
%     Z3    = permute(Z3,   [2 1 3]);
%     if ~isempty(X3), X3  = permute(X3, [2 1 3]); end
%     if ~isempty(Y3), Y3  = permute(Y3, [2 1 3]); end
%     BxVol = permute(BxVol,[2 1 3]);  ByVol = permute(ByVol,[2 1 3]);  BzVol
= permute(BzVol,[2 1 3]);
%     TVol  = permute(TVol, [2 1 3]);  rhoVol= permute(rhoVol,[2 1 3]); zVol
= permute(zVol, [2 1 3]);
%     BVol  = permute(BVol, [2 1 3]);  BinclVol = permute(BinclVol,[2 1 3]);
BazimVol = permute(BazimVol,[2 1 3]);
%     [Nx,Ny,~] = size(Z3);
% end

dx_m = opts.dx * Mm2m;   % used if X3 is empty
dy_m = opts.dy * Mm2m;   % used if Y3 is empty
dz_m = opts.dz * Mm2m;

%% -------------------------- gravity field (m/s^2)
switch lower(opts.gravity.mode)
    case 'radial'
        % g(z) = g0 * (Rsun / (Rsun+z))^2  (z in m above the photosphere)
        Rsun = 6.957e8;
        z_m  = Z3 * Mm2m;
        gVol = opts.gravity.g0 .* (Rsun ./ max(Rsun + z_m, 1)) .^ 2;
    otherwise
        gVol = opts.gravity.g0; % scalar broadcast
end

%% -------------------------- pressure gradient & body force
if w.force > 0

     [dpx,dpy,dpz] = grad_mixed_SI(pVol, dx_m, dy_m, dz_m);

     dp = [dpx(:)'; dpy(:)'; dpz(:)'];
```

```
    gx = zeros(size(pVol), 'like', likeRef);
    gy = gx;
    gz = rhoVol .* gVol; % N/m^3
    g1 = [gx(:)'; gy(:)'; gz(:)'];
else
    dp = dlarray(zeros(3, numel(pVol), 'like', likeRef));
    g1 = dp;
end

ii = find(isnan(extractdata(pVol)));
%fprintf('length(ii) = %d\n', length(ii))
%% -------------------------- curl, div, JxB
needBops = (w.force>0) || (w.divB>0) || (w.Bx>0) || (w.By>0) || (w.Bz>0);
if needBops
  %  size(X3)
  %   size(Y3)
  %    size(Z3)


    [dBx_dx,dBx_dy,dBx_dz] = grad_mixed_SI(BxVol, dx_m, dy_m, dz_m);
    [dBy_dx,dBy_dy,dBy_dz] = grad_mixed_SI(ByVol, dx_m, dy_m, dz_m);
    [dBz_dx,dBz_dy,dBz_dz] = grad_mixed_SI(BzVol, dx_m, dy_m, dz_m);

     % ii = find(isnan((dx_m)));
%fprintf('dx length(ii) = %d\n', length(ii))

    %ii = find(isnan(extractdata(dBx_dx)));
%fprintf('dBx_dx length(ii) = %d\n', length(ii))

    %ii = find(isnan(extractdata(dBx_dy)));
%fprintf('dBx_dy length(ii) = %d\n', length(ii))


    curlBx = dBz_dy - dBy_dz;
    curlBy = dBx_dz - dBz_dx;
    curlBz = dBy_dx - dBx_dy;

    divBVol = dBx_dx + dBy_dy + dBz_dz;

    JxVol = curlBx / mu0;
    JyVol = curlBy / mu0;
    JzVol = curlBz / mu0;

    J2Vol = JxVol.^2 + JyVol.^2 + JzVol.^2;

   % max(JxVol(:))

    JxB_x = JyVol .* BzVol - JzVol .* ByVol;
    JxB_y = JzVol .* BxVol - JxVol .* BzVol;
    JxB_z = JxVol .* ByVol - JyVol .* BxVol;
    JxB   = [JxB_x(:)'; JxB_y(:)'; JxB_z(:)'];
else
    divBVol = zeros(size(BxVol), 'like', likeRef);
    JxB = dlarray(zeros(3, numel(pVol), 'like', likeRef));
end
```

```
%ii = find(isnan(extractdata(JxB)));
%fprintf('JxB length(ii) = %d\n', length(ii))

%ii = find(isnan(extractdata(divBVol)));
%fprintf('divB length(ii) = %d\n', length(ii))

% R0 (force balance)
if w.force > 0
    if strcmpi(opts.forceModel,'JxB_only')
        R0 = JxB;
    else
        R0 = dp - JxB + g1;
    end
else
    R0 = dlarray(zeros(size(JxB), 'like', likeRef));
end

% R1 (divB)
if w.divB > 0
    R1 = divBVol(:)'; % CB
else
    R1 = dlarray(zeros(1, numel(divBVol), 'like', likeRef));
end



%% --------------------------- energy / emissivity
Qheat = zeros(size(TVol), 'like', likeRef);
Qrad0 = Qheat; Qrad1 = Qheat; Qrad2 = Qheat; Qrad3 = Qheat;
I1 = zeros(Nx,Ny,'like',likeRef); I2 = I1; I3 = I1;
res_energy = zeros(size(TVol), 'like', likeRef);

needEmissivity = (w.I171>0) || (w.I193>0) || (w.I335>0) || (w.energy>0);
if needEmissivity
    % Field-aligned Spitzer conduction q = -kappa0 * T^(5/2) * (b̂·∇T) * b̂
    kappa0 = 1e-11; % W·m^-1·K^-7/2 (order of magnitude)
    [dTdx,dTdy,dTdz] = grad_mixed_SI(TVol, dx_m, dy_m, dz_m);

    Bnorm = sqrt(BxVol.^2 + ByVol.^2 + BzVol.^2 + epsv);
    bhx = BxVol ./ Bnorm; bhy = ByVol ./ Bnorm; bhz = BzVol ./ Bnorm;
    bdotGradT = bhx.*dTdx + bhy.*dTdy + bhz.*dTdz;
    qx = -kappa0 * TVol.^(5/2) .* bhx .* bdotGradT;
    qy = -kappa0 * TVol.^(5/2) .* bhy .* bdotGradT;
    qz = -kappa0 * TVol.^(5/2) .* bhz .* bdotGradT;

    % ∇·q in SI
    [dqxdx,~,~] = grad_mixed_SI(qx, dx_m, dy_m, dz_m);
    [~,dqydy,~] = grad_mixed_SI(qy, dx_m, dy_m, dz_m);
    [~,~,dqzdz] = grad_mixed_SI(qz, dx_m, dy_m, dz_m);
    div_q = dqxdx + dqydy + dqzdz;

    % phenomenological heating (logistic vs z in Mm)
    z_mm = zVol*opts.rangeN;
    Q0 = 1; z0 = 3.5; dzL = 0.8; scaleZ = 1;
```

```
    Qheat = Q0 ./ (1 + exp(-(scaleZ*z_mm - z0)/dzL));

    Qheat_joule = etaVol.*J2Vol;
    %max(J2Vol(:))

   % max(z_mm(:))

%    ii = find(isnan((z_mm)));
    %fprintf('zm length(ii) = %d\n', length(ii))

 %   ii = find(isnan((Qheat)));
    %fprintf('Qheat length(ii) = %d\n', length(ii))

    % simple radiative cooling + channel-like responses
    tau = 4;
    Qrad0 = TVol / tau;

    if w.I171>0 || w.energy>0
        Qrad1 = QradT(1, rhoVol, TVol);
        I1 = sum(Qrad1.* opts.dz, 3);

          %  ii = find(isnan(extractdata(Qrad1)));
    %fprintf('Qrad1 length(ii) = %d\n', length(ii))
    end
    if w.I193>0 || w.energy>0
        Qrad2 = QradT(2, rhoVol, TVol);
        I2 = sum(Qrad2.* opts.dz, 3);

        %    ii = find(isnan(extractdata(Qrad2)));
    %fprintf('Qrad2 length(ii) = %d\n', length(ii))
    end
    if w.I335>0 || w.energy>0
        Qrad3 = QradT(3, rhoVol, TVol);
        I3 = sum(Qrad3.* opts.dz, 3);

          %  ii = find(isnan(extractdata(Qrad3)));
    %fprintf('Qrad3 length(ii) = %d\n', length(ii))
    end

    if w.energy>0
        Qrad = Qrad0 + Qrad1 + Qrad2 + Qrad3;
        res_energy = div_q + Qheat + Qheat_joule - Qrad;

        %ii = find(isnan(extractdata(Qrad)));
        %fprintf('length(ii) = %d\n', length(ii))
    end
end


%ii = find(isnan(extractdata(divBVol)));
%fprintf('divB length(ii) = %d\n', length(ii))
%% --------------------------- slice constraints (k=ii1)
Bx1d = dlarray(cast(Bx1,'like',extractdata(likeRef)));
By1d = dlarray(cast(By1,'like',extractdata(likeRef)));
Bz1d = dlarray(cast(Bz1,'like',extractdata(likeRef)));
```

```
Ic3gd = dlarray(cast(Ic3g,'like',extractdata(likeRef)));
x3gd  = dlarray(cast(x3g, 'like',extractdata(likeRef)));
y3gd  = dlarray(cast(y3g, 'like',extractdata(likeRef)));
z3gd  = dlarray(cast(z3g, 'like',extractdata(likeRef)));

ii1 = min(3, Nz);  % safeguard
if w.Bx>0
    Bx1P = BVol(:,:,ii1);
    R6 = 10*( Bx1d(:)'-Bx1P(:)' );
else
    R6 = zeros(1,numel(Bx1d),'like',likeRef);
end
if w.By>0
    By1P = projectAngleToPi(BinclVol(:,:,ii1));
    R7 = (By1d(:)'-By1P(:)');
else
    R7 = zeros(1,numel(By1d),'like',likeRef);
end
if w.Bz>0
    Bz1P = projectAngleToPi(BazimVol(:,:,ii1));
    R8 = (Bz1P(:)'-Bz1d(:)');
else
    R8 = zeros(1,numel(Bz1d),'like',likeRef);
end
if w.Timg>0
    T1P = TVol(:,:,ii1);
    R9 = (T1P(:)' - Ic3gd(:)');
else
    R9 = zeros(1,numel(Ic3gd),'like',likeRef);
end
if w.energy>0
    R10 = res_energy(:)';
else
    R10 = zeros(1,numel(res_energy),'like',likeRef);
end

if w.I171>0, R3 = ( x3gd(:)'-I1(:)' ); else, R3 = zeros(1, numel(x3gd),
'like', likeRef); end
if w.I193>0, R4 = ( y3gd(:)' - I2(:)'); else, R4 = zeros(1, numel(y3gd),
'like', likeRef); end
if w.I335>0, R5 = ( z3gd(:)' - I3(:)'); else, R5 = zeros(1, numel(z3gd),
'like', likeRef); end

%% -------------------------- automatic normalization (EMA of RMS)
if opts.norm.enable
    [R0,  opts] = applyNorm('R0',  R0,  w.force>0,  opts);
    [R1,  opts] = applyNorm('R1',  R1,  w.divB>0,   opts);
    [R3,  opts] = applyNorm('R3',  R3,  w.I171>0,   opts);
    [R4,  opts] = applyNorm('R4',  R4,  w.I193>0,   opts);
    [R5,  opts] = applyNorm('R5',  R5,  w.I335>0,   opts);
    [R6,  opts] = applyNorm('R6',  R6,  w.Bx>0,     opts);
    [R7,  opts] = applyNorm('R7',  R7,  w.By>0,     opts);
    [R8,  opts] = applyNorm('R8',  R8,  w.Bz>0,     opts);
    [R9,  opts] = applyNorm('R9',  R9,  w.Timg>0,   opts);
    [R10, opts] = applyNorm('R10', R10, w.energy>0, opts);
```

```
end

%% --------------------------- losses
a = struct();
a.loss0  = w.force  * mse(R0,  zeros(size(R0),  'like', R0),
'DataFormat','CB');
a.loss1  = w.divB   * mse(R1,  zeros(size(R1),  'like', R1),
'DataFormat','CB');
a.loss3  = w.I171   * mse(R3,  zeros(size(R3),  'like', R3),
'DataFormat','CB');
a.loss4  = w.I193   * mse(R4,  zeros(size(R4),  'like', R4),
'DataFormat','CB');
a.loss5  = w.I335   * mse(R5,  zeros(size(R5),  'like', R5),
'DataFormat','CB');
a.loss6  = w.Bx     * mse(R6,  zeros(size(R6),  'like', R6),
'DataFormat','CB');
a.loss7  = w.By     * mse(R7,  zeros(size(R7),  'like', R7),
'DataFormat','CB');
a.loss8  = w.Bz     * mse(R8,  zeros(size(R8),  'like', R8),
'DataFormat','CB');
a.loss9  = w.Timg   * mse(R9,  zeros(size(R9),  'like', R9),
'DataFormat','CB');
a.loss10 = w.energy * mse(R10, zeros(size(R10), 'like', R10),
'DataFormat','CB');

loss = a.loss0 + a.loss1 + a.loss3 + a.loss4 + a.loss5 + ...
       a.loss6 + a.loss7 + a.loss8 + a.loss9 + a.loss10;

%% --------------------------- backprop
gradients = dlgradient(loss, net.Learnables);

%% Save

if ~opts.save.enable
    return;
else
    % ----- salvamento opcional -----
    varsToSave = struct( ...
        'xVol', xVol, 'yVol', yVol, 'zVol', zVol, ...
        'BxVol', BxVol, 'ByVol', ByVol, 'BzVol', BzVol, ...
        'Qheat', Qheat, 'Qrad0', Qrad0, 'Qrad1', Qrad1, ...
        'Qrad2', Qrad2, 'Qrad3', Qrad3, ...
        'Qheat_joule', exist('Qheat_joule','var')*Qheat_joule, ...
        'pVol', pVol, 'TVol', TVol, 'etaVol', etaVol, ...
        'JxVol', exist('JxVol','var')*JxVol, ...
        'JyVol', exist('JyVol','var')*JyVol, ...
        'JzVol', exist('JzVol','var')*JzVol, ...
        'J2Vol', exist('J2Vol','var')*J2Vol, ...
        'divBVol', divBVol);

    saveToFits(varsToSave, opts);
end


end % main
```

```
% ==================== helpers ====================
function A = castLike(A, ref)
% Convert A to same precision/device as ref (dlarray).
    ur = extractdata(ref);
    if isa(ur,'gpuArray'), A = gpuArray(A); ur = gather(ur); end
    if isa(ur,'single'), A = single(A); else, A = double(A); end
end

function [Rscaled, opts] = applyNorm(name, R, enabled, opts)
if ~enabled, Rscaled = R; return; end
rmsNow = sqrt(mean(R.^2, 'all'));
rmsNow = max(1e-12, double(gather(extractdata(rmsNow))));
if ~isfield(opts.norm.stats, name) || isempty(opts.norm.stats.(name))
    st = struct('ema', rmsNow, 't', 0);
else
    st = opts.norm.stats.(name);
end
if st.t < opts.norm.freezeIters
    beta = opts.norm.beta;
    st.ema = beta*st.ema + (1-beta)*rmsNow;
    st.t   = st.t + 1;
end
emaHat = st.ema / max(1e-12, (1 - opts.norm.beta^max(1,st.t)));
if st.t <= opts.norm.warmup
    scale = 1.0;
else
    scale = 1 / max(emaHat, 1e-12);
    scale = min(max(scale, opts.norm.clip(1)), opts.norm.clip(2));
end
Rscaled = R .* scale;
opts.norm.stats.(name) = st;
end


% function [Bx, By, Bz] = hmi_vector_to_cartesian(B, incl, azim)
% % Convert (B magnitude, inclination, azimuth) to Cartesian (radians).
% % Convention: incl=0 => +z; incl=pi/2 in the plane; azim measured from +x
toward +y.
%     Bx = B .* sin(incl) .* cos(azim);
%     By = B .* sin(incl) .* sin(azim);
%     Bz = B .* cos(incl);
% end

% function A = projectAngleToPi(A)
% % Wrap angle to (-pi, pi]
%     A = mod(A + pi, 2*pi) - pi;
% end

% function Q = QradT(chan, rho, T)
% % Toy radiative loss per channel: Q ~ rho^2 * R_chan(T) with a logT-Gaussian
response.
% % 1->171 Å (~0.8 MK), 2->193 Å (~1.5 MK), 3->335 Å (~2.5 MK)
%     logT = log10(max(T,1e3));
%     switch chan
```

```
%         case 1, mu=5.9; sigma=0.15;
%         case 2, mu=6.2; sigma=0.15;
%         case 3, mu=6.4; sigma=0.15;
%         otherwise, mu=6.2; sigma=0.2;
%     end
%     R = exp(-0.5*((logT-mu)/sigma).^2);
%     Lambda0 = 1e-31; % W·m^3 (placeholder scale)
%     Q = Lambda0 * (rho.^2) .* R;  % W/m^3
% end
```