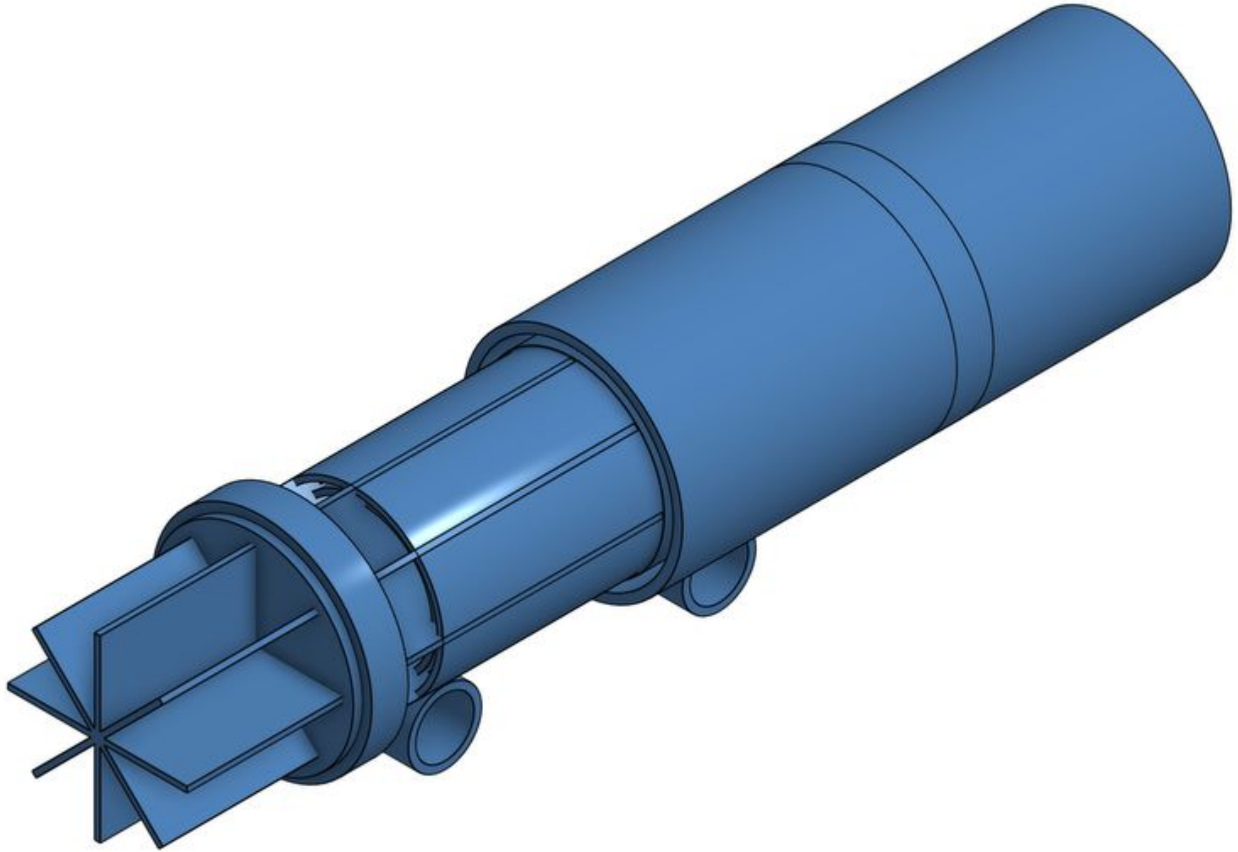
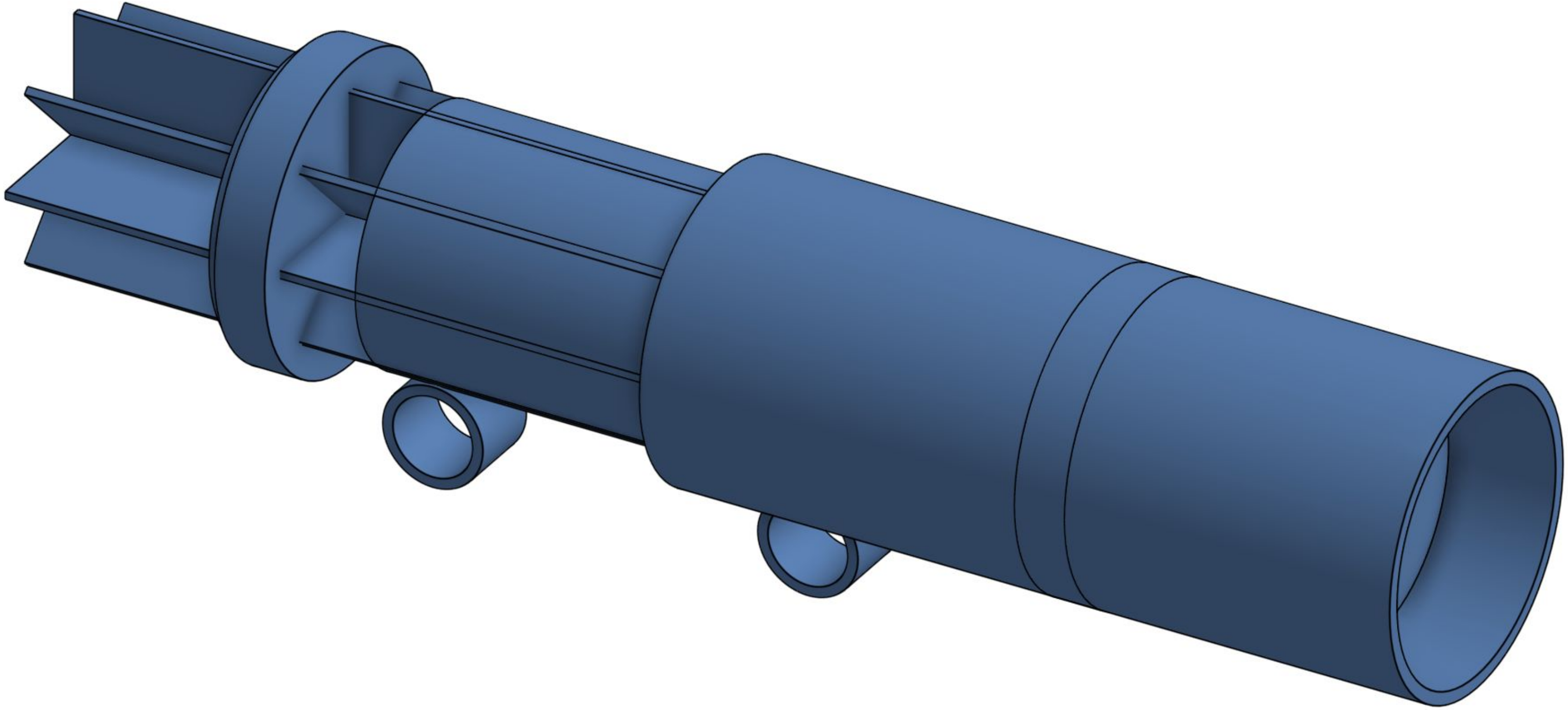
Vascura HydroNet: Supplementals

CAD Model, Prototype, and PIXGNN MATLAB Code





```matlab
function run_Chemical_System_MultiInverse
   % RUN_CHEMICAL_SYSTEM_MULTIINVERSE
   % Robust Inverse PINN for Coupled Chemo-Hydrodynamic System

   clc; clear; close all;

   %% 1. CONFIGURATION
   config.N_grid = 100;
   config.T_final = 0.5;
   config.MaxEpochs = 10000;  % "Wayyy more" epochs
   config.lr_Net = 0.001;     % Adam LR for Network
   config.lr_Params = 0.01;   % Boosted LR for Parameters

   % --- TRUE PARAMETERS (Ground Truth) ---
   p_true.alpha = 0.01;       % Target
   p_true.gD2   = 1.0;        % Target (Heavy contaminant)
   p_true.beta2 = 1.0;        % Target (Reactive)

   % --- FIXED INFRASTRUCTURE PARAMETERS ---
   p_true.D1    = 0.5;
   p_true.delta = 0.1;
   p_true.nu    = 0.02;
   p_true.Ds    = 0.01;
   p_true.beta1 = 2.0;
   p_true.Di    = 0.01;
   p_true.gamma = 0.5;
   p_true.Dr    = 0.01;

   %% 2. DATA GENERATION (GROUND TRUTH)
   fprintf('Step 1: Simulating Ground Truth (Method of Lines)...\n');

   x = linspace(0, 1, config.N_grid)';
   dx = x(2) - x(1);
   t_span = linspace(0, config.T_final, 60);   % 60 Time snapshots

   % Initial Conditions
   U0 = zeros(5*config.N_grid, 1);
   idx_S = (2*config.N_grid+1):3*config.N_grid;
   U0(idx_S) = 1.0;

   opts = odeset('RelTol', 1e-5, 'AbsTol', 1e-5);
   [~, U_sol] = ode15s(@(t, u) rhs_pde(t, u, x, dx,
                config.N_grid, p_true), t_span, U0, opts);
   % Extract Raw Data
   Raw_C = U_sol(:, 1:config.N_grid)';
   Raw_U = U_sol(:, config.N_grid+1:2*config.N_grid)';
   Raw_S = U_sol(:, 2*config.N_grid+1:3*config.N_grid)';
```

```matlab
   Raw_I = U_sol(:, 3*config.N_grid+1:4*config.N_grid)';
   Raw_R = U_sol(:, 4*config.N_grid+1:5*config.N_grid)';

   %% 3. PRE-PROCESSING & NORMALIZATION
   fprintf('Step 2: Normalizing Data for PINN...\n');

   % 1. Input Normalization (Map x,t to [-1, 1])
   [T_mesh, X_mesh] = meshgrid(t_span, x);
   X_norm = 2 * X_mesh - 1;       % [0,1] -> [-1,1]
   T_norm = 2 * (T_mesh/config.T_final) - 1;

   dlX = dlarray(X_norm(:)',  'CB');
   dlT = dlarray(T_norm(:)',  'CB');

   % 2. Output Normalization (Z-Score)
   Data_Raw = [Raw_C(:)'; Raw_U(:)'; Raw_S(:)';
Raw_I(:)'; Raw_R(:)'];
   mu = mean(Data_Raw, 2);
   sig = std(Data_Raw, 0, 2) + 1e-6;   % Avoid div by zero

   Data_Norm = (Data_Raw - mu) ./ sig;
   dlTargets = dlarray(Data_Norm,  'CB');

   % Store normalization constants for Physics Loss
   stats.mu = dlarray(mu);
   stats.sig = dlarray(sig);
   stats.T_scale = config.T_final / 2;   % dt/dT_norm
   stats.X_scale = 1.0 / 2;              % dx/dX_norm

   %% 4. NETWORK & PARAMETER SETUP

   % Neural Network (Classic PINN Architecture)
   layers = [
      featureInputLayer(2,  'Name', 'in')
      fullyConnectedLayer(64,  'Name', 'fc1')
      tanhLayer('Name', 'act1')
      fullyConnectedLayer(64,  'Name', 'fc2')
      tanhLayer('Name', 'act2')
      fullyConnectedLayer(64,  'Name', 'fc3')
      tanhLayer('Name', 'act3')
```

```matlab
      fullyConnectedLayer(64,  'Name', 'fc4')
      tanhLayer('Name', 'act4')
      fullyConnectedLayer(5,  'Name', 'out')
   ];
   net = dlnetwork(layers);

   % Unknown Parameters (Initialized in Log-Space)
   % True: Alpha=0.01, gD2=1.0, Beta2=1.0
   % Guesses: Alpha=0.05, gD2=0.1, Beta2=0.1

   theta_Alpha = dlarray(log(0.05),   'CB');
   theta_gD2   = dlarray(log(0.1),    'CB');
   theta_Beta2 = dlarray(log(0.1),    'CB');

   %% 5. ROBUST TRAINING LOOP
   fprintf('Step 3: Starting Training (%d Epochs)...\n',
config.MaxEpochs);

   % Adam State Variables
   avgNet = []; sqNet = [];
   avgP_A = []; sqP_A = [];
   avgP_G = []; sqP_G = [];
   avgP_B = []; sqP_B = [];

   % History
   h_loss = []; h_alpha = []; h_gD2 = []; h_beta2 = [];

   % Real-time Plotting
   figure('Name', 'Inverse PINN Training', 'Color', 'w',
'Position', [100 100 1000 600]);
   tiledlayout(2,2);

   ax1 = nexttile; hL = semilogy(ax1, 0,0,   'k-',
'LineWidth', 1.5); title('Total Loss'); grid on;
   ax2 = nexttile; hold on; title('Alpha (Diffusion)');
yline(p_true.alpha, 'k--'); hP1 =
plot(0,0, 'r','LineWidth',2); grid on;
   ax3 = nexttile; hold on; title('gD2 (Gravity
Coupling)'); yline(p_true.gD2, 'k--'); hP2 =
plot(0,0, 'g','LineWidth',2); grid on;
   ax4 = nexttile; hold on; title('Beta2 (Reactivity)');
yline(p_true.beta2, 'k--'); hP3 =
plot(0,0, 'b','LineWidth',2); grid on;

   w_phys = 0.0;  % Start with 0 physics weight
```

```matlab
   start_tic = tic;

   for epoch = 1:config.MaxEpochs

      % 1. Compute Gradients
      [gradients_net, loss, loss_d, loss_p, g_th_Alpha,
g_th_gD2, g_th_Beta2] = dlfeval( ...
         @modelLoss, net, dlX, dlT, dlTargets,
theta_Alpha, theta_gD2, theta_Beta2, p_true, stats,
w_phys);

      % 2. Dynamic Physics Weighting (Safety Gate)
      loss_val = extractdata(loss_d);
      if w_phys == 0
         if loss_val < 0.05  % Initial fit achieved
            w_phys = 0.1;
            fprintf('>>> Shape Learned! Activating
Physics.\n');
         end
      else
         % Ramp up physics weight slowly to 1.0
         w_phys = min(1.0, w_phys * 1.002);
      end

      % 3. Update Network (Adam)
      [net, avgNet, sqNet] = adamupdate(net,
gradients_net, avgNet, sqNet, epoch, config.lr_Net);

      % 4. Update Parameters (Adam + Boost)
      % Only update parameters if physics is active!
      if w_phys > 0
         [theta_Alpha, avgP_A, sqP_A] =
adamupdate(theta_Alpha, g_th_Alpha, avgP_A, sqP_A,
epoch, config.lr_Params);
         [theta_gD2,   avgP_G, sqP_G] =
adamupdate(theta_gD2,   g_th_gD2,   avgP_G, sqP_G,
epoch, config.lr_Params);
         [theta_Beta2, avgP_B, sqP_B] =
adamupdate(theta_Beta2, g_th_Beta2, avgP_B, sqP_B,
epoch, config.lr_Params);
      End
      % 5. Logging
      if mod(epoch, 50) == 0
         % Convert log-params back to physical
         curr_Alpha = extractdata(exp(theta_Alpha));
         curr_gD2   = extractdata(exp(theta_gD2));
         curr_Beta2 = extractdata(exp(theta_Beta2));

         h_loss(end+1) = extractdata(loss);
         h_alpha(end+1) = curr_Alpha;
         h_gD2(end+1)   = curr_gD2;
         h_beta2(end+1) = curr_Beta2;
```

```
            % Update Plots
            set(hL, 'XData', 1:length(h_loss), 'YData',
h_loss);
            set(hP1, 'XData', 1:length(h_alpha), 'YData',
h_alpha);
            set(hP2, 'XData', 1:length(h_gD2), 'YData',
h_gD2);
            set(hP3, 'XData', 1:length(h_beta2), 'YData',
h_beta2);
            drawnow limitrate;

            if mod(epoch, 500) == 0
                fprintf('Ep %d | Loss: %.4f (Phys: %.4f)
| Alph: %.3f | gD2: %.3f | B2: %.3f\n', ...
                    epoch, extractdata(loss),
extractdata(loss_p), curr_Alpha, curr_gD2, curr_Beta2);
            end
        end
    end

    fprintf('\n--- Final Results ---\n');
    fprintf('Alpha:  Pred: %.4f | True: %.4f\n',
extractdata(exp(theta_Alpha)), p_true.alpha);
    fprintf('gD2:    Pred: %.4f | True: %.4f\n',
extractdata(exp(theta_gD2)), p_true.gD2);
    fprintf('Beta2:  Pred: %.4f | True: %.4f\n',
extractdata(exp(theta_Beta2)), p_true.beta2);
    toc(start_tic);
End
%% --- HELPER: GROUND TRUTH ODE ---
function dydt = rhs_pde(t, u, x, dx, Nx, p)
    % Unpack
    C = u(1:Nx); U = u(Nx+1:2*Nx); S = u(2*Nx+1:3*Nx);
    I = u(3*Nx+1:4*Nx); R = u(4*Nx+1:5*Nx);

    % Inlet Boundary (Fat Pulse)
    C(1) = exp(-5.0 * (t - 0.1).^2);
    S(1) = 1.0; U(1) = 0.0; I(1) = 0.0; R(1) = 0.0;

    dCdt = zeros(Nx,1); dUdt = zeros(Nx,1); dSdt =
zeros(Nx,1);
    dIdt = zeros(Nx,1); dRdt = zeros(Nx,1);

    for i = 2:Nx-1
        % Central Differences
        C_xx = (C(i+1)-2*C(i)+C(i-1))/dx^2;
        U_xx = (U(i+1)-2*U(i)+U(i-1))/dx^2;
        S_xx = (S(i+1)-2*S(i)+S(i-1))/dx^2;
        I_xx = (I(i+1)-2*I(i)+I(i-1))/dx^2;
        R_xx = (R(i+1)-2*R(i)+R(i-1))/dx^2;

        % Upwind Advection
        if U(i) >= 0
            C_x = (C(i)-C(i-1))/dx; U_x =
(U(i)-U(i-1))/dx;
            S_x = (S(i)-S(i-1))/dx; I_x =
(I(i)-I(i-1))/dx; R_x = (R(i)-R(i-1))/dx;
        else
            C_x = (C(i+1)-C(i))/dx; U_x =
(U(i+1)-U(i))/dx;
            S_x = (S(i+1)-S(i))/dx; I_x =
(I(i+1)-I(i))/dx; R_x = (R(i+1)-R(i))/dx;
        end

        % Reaction Terms
        Rx_C = p.D1*I(i) - p.delta*C(i);
        Rx_U = p.gD2*C(i);
        Rx_S = -p.beta1*S(i)*I(i) - p.beta2*S(i)*C(i);
        Rx_I = p.beta1*S(i)*I(i) + p.beta2*S(i)*C(i) -
p.gamma*I(i);
        Rx_R = p.gamma*I(i);

        dCdt(i) = p.alpha*C_xx - U(i)*C_x + Rx_C;
        dUdt(i) = p.nu*U_xx    - U(i)*U_x + Rx_U;

        dSdt(i) = p.Ds*S_xx    - U(i)*S_x + Rx_S;
        dIdt(i) = p.Di*I_xx    - U(i)*I_x + Rx_I;
        dRdt(i) = p.Dr*R_xx    - U(i)*R_x + Rx_R;
    end

    dCdt(Nx)=dCdt(Nx-1); dUdt(Nx)=dUdt(Nx-1);
dSdt(Nx)=dSdt(Nx-1);
    dIdt(Nx)=dIdt(Nx-1); dRdt(Nx)=dRdt(Nx-1);
    dydt = [dCdt; dUdt; dSdt; dIdt; dRdt];
end
%% --- HELPER: PINN LOSS FUNCTION ---
function [gNet, loss, loss_data, loss_phys, g_th_Alpha,
g_th_gD2, g_th_Beta2] = modelLoss(net, dlX, dlT,
dlTargets, th_Alpha, th_gD2, th_Beta2, p_fixed, stats,
w_phys)

    % 1. Forward Pass
    out_norm = forward(net, [dlT; dlX]);

    % 2. Data Loss
    loss_data = mse(out_norm, dlTargets);

    % 3. Physics Loss
    if w_phys > 0
        % Convert Log-Params to Physical Space
        Alpha = exp(th_Alpha);
        gD2   = exp(th_gD2);
        Beta2 = exp(th_Beta2);

        % Extract Normalization Stats
        mu = stats.mu;
        sig = stats.sig;
        dt_dNorm = stats.T_scale;
        dx_dNorm = stats.X_scale;

        % Automatic Differentiation (Gradient w.r.t
Inputs T_norm, X_norm)
        % out_norm(1) is C, (2) is U, etc.

        % -- Helper for Derivatives --
        % We calculate d(Output_Norm)/d(Input_Norm) then
scale to physical

        % C derivatives
        gC = dlgradient(sum(out_norm(1,:)), {dlT, dlX},
'EnableHigherDerivatives', true);
        Ct_n = gC{1}; Cx_n = gC{2};
        Cxx_n = dlgradient(sum(Cx_n), dlX,
'EnableHigherDerivatives', true);

        gU = dlgradient(sum(out_norm(2,:)), {dlT, dlX},
'EnableHigherDerivatives', true);
        Ut_n = gU{1}; Ux_n = gU{2};
        Uxx_n = dlgradient(sum(Ux_n), dlX,
'EnableHigherDerivatives', true);

        % S derivatives
        gS = dlgradient(sum(out_norm(3,:)), {dlT, dlX},
'EnableHigherDerivatives', true);
        St_n = gS{1}; Sx_n = gS{2};
        Sxx_n = dlgradient(sum(Sx_n), dlX,
'EnableHigherDerivatives', true);
        % I derivatives
        gI = dlgradient(sum(out_norm(4,:)), {dlT, dlX},
'EnableHigherDerivatives', true);
        It_n = gI{1}; Ix_n = gI{2};
        Ixx_n = dlgradient(sum(Ix_n), dlX,
'EnableHigherDerivatives', true);
        % R derivatives
        gR = dlgradient(sum(out_norm(5,:)), {dlT, dlX},
'EnableHigherDerivatives', true);
        Rt_n = gR{1}; Rx_n = gR{2};
        Rxx_n = dlgradient(sum(Rx_n), dlX,
'EnableHigherDerivatives', true);

        C = out_norm(1,:) .* sig(1) + mu(1);
        U = out_norm(2,:) .* sig(2) + mu(2);
        S = out_norm(3,:) .* sig(3) + mu(3);
        I = out_norm(4,:) .* sig(4) + mu(4);
        R = out_norm(5,:) .* sig(5) + mu(5);

        Ct = Ct_n .* (sig(1)/dt_dNorm); Cx
= Cx_n .* (sig(1)/dx_dNorm); Cxx = Cxx_n .*
(sig(1)/dx_dNorm^2);
        Ut = Ut_n .* (sig(2)/dt_dNorm); Ux =
Ux_n .* (sig(2)/dx_dNorm); Uxx = Uxx_n .*
(sig(2)/dx_dNorm^2);
        St = St_n .* (sig(3)/dt_dNorm); Sx =
Sx_n .* (sig(3)/dx_dNorm); Sxx = Sxx_n .*
(sig(3)/dx_dNorm^2);
        It = It_n .* (sig(4)/dt_dNorm); Ix =
Ix_n .* (sig(4)/dx_dNorm); Ixx = Ixx_n .*
(sig(4)/dx_dNorm^2);
        Rt = Rt_n .* (sig(5)/dt_dNorm); Rx =
Rx_n .* (sig(5)/dx_dNorm); Rxx = Rxx_n .*
(sig(5)/dx_dNorm^2);

        % -- Compute Residuals (Physical
Space) --
        res_C = Ct + U.*Cx - Alpha.*Cxx -
p_fixed.D1.*I + p_fixed.delta.*C;
        res_U = Ut + U.*Ux - p_fixed.nu.*Uxx
- gD2.*C;
        res_S = St + U.*Sx - p_fixed.Ds.*Sxx
+ p_fixed.beta1.*S.*I + Beta2.*S.*C;
        res_I = It + U.*Ix - p_fixed.Di.*Ixx
- p_fixed.beta1.*S.*I - Beta2.*S.*C +
p_fixed.gamma.*I;
        res_R = Rt + U.*Rx - p_fixed.Dr.*Rxx
- p_fixed.gamma.*I;

        % This is a crucial step for stable
training.
        loss_phys = mean( (res_C/sig(1)).^2 +
(res_U/sig(2)).^2 + (res_S/sig(3)).^2 +   ...
                          (res_I/sig(4)).^2 +
(res_R/sig(5)).^2 );
    else
        loss_phys = dlarray(0);
        g_th_Alpha = dlarray(0);
        g_th_gD2   = dlarray(0);
        g_th_Beta2 = dlarray(0);
    end

    % Total Loss
    loss = loss_data + w_phys * loss_phys;

    % Gradients
    gNet = dlgradient(loss, net.Learnables);

    if w_phys > 0
        g_th_Alpha = dlgradient(loss,
th_Alpha);
        g_th_gD2   = dlgradient(loss,
th_gD2);
        g_th_Beta2 = dlgradient(loss,
th_Beta2);
    end
end
```