```
function [s] = generateOptimizedLFM(simParams, s_init)

% Number of samples per pulse
nPulse = length(simParams.t);

% Form binary spectral mask
G = abs(simParams.spectral_template);
wF = fftshift(G <= 0.9);

% Percent power scaling for inequality constraint
gamma = (sum(G<=0)/length(G))/(nPulse)*(10^(6/10));

% Parse initialization waveform
if nargin < 2 || isempty(s_init)
    % No initial waveform provided -> default to LFM
    s_init = generateLFM(simParams);
end
s_init = [real(s_init); imag(s_init)];

% Define autocorrelation mask
wsl = computeCorrelationMask(s_init, simParams.nfft);

% Define general optimization parameters
optParams = struct();
optParams.outerIter = 200;
optParams.innerIter = 20;
optParams.p = 8;
optParams.gamma = gamma;
optParams.nPulse = nPulse;
optParams.nfft = simParams.nfft;
optParams.wsl = wsl;
optParams.wF = wF;
optParams.alpha = 5e-5;
optParams.rho_init = 0.1;
optParams.eta_init = 0.1;
optParams.tol = 1e-4;

% Call Augmented Lagrange Method wrapper
[x_opt, hist] = ALM(s_init, optParams);

% Plot convergence curves
plotConvergence(hist);

% Reconstruct waveform from real and imaginary
s = x_opt(1:nPulse) + 1i*x_opt(nPulse+1:end);

% Normalize pulse to unit energy
s = s./sqrt(sum(abs(s).^2));



%% Objective and Constraint Functions

function [J] = Jx(x, optParams)

    nPulse = optParams.nPulse;
    nfft = optParams.nfft;
    wsl = optParams.wsl;
```

```
59      p = optParams.p;
60
61      % Reconstruct waveform from real and imaginary
62      s = x(1:nPulse) + 1i*x(nPulse+1:end);
63
64      % Evaluate cost function
65      ccorr = ifft( fft(s, nfft) .* conj(fft(s, nfft)));
66      J = log(sum(abs(wsl.*ccorr).^p));
67
68  return
69
70  function [VJ] = VJx(x, optParams)
71
72      nfft = optParams.nfft;
73      wsl = optParams.wsl;
74      p = optParams.p;
75      nPulse = optParams.nPulse;
76
77      % Reconstruct waveform from real and imaginary
78      s = x(1:nPulse) + 1i.*x(nPulse+1:end);
79
80      % Evaluate cost function
81      ccorr = ifft(fft(s, nfft) .* conj(fft(s, nfft)));
82      Jsl = sum(abs(wsl.*ccorr).^p);
83
84      % Redundant pre-calculations
85      temp = (abs(ccorr).^(p-2)).*ccorr;
86      tempsl = fft(wsl.*temp);
87
88      % CELSI gradient
89      S = fft(s, nfft);
90      c = ifft(S .* tempsl);
91      g_s = p * c(1:nPulse);
92      g_re = 2 * real(g_s) / Jsl;
93      g_im = 2 * imag(g_s) / Jsl;
94      VJ = [g_re; g_im];
95
96  return
97
98  function [H] = Hx(x, optParams)
99
100     nPulse = optParams.nPulse;
101
102     % Reconstruct complex waveform
103     s = x(1:nPulse) + 1i*x(nPulse+1:end);
104
105     % Constant-envelope constraint
106     H = abs(s).^2 - 1;
107
108 return
109
110 function VH = JHx(x, optParams)
111
112     nPulse = optParams.nPulse;
113
114     % reconstruct complex vector
115     s = x(1:nPulse) + 1i*x(nPulse+1:end);
116
```

```
    % allocate Jacobian: N constraints × 2N variables
    VH = zeros(nPulse, 2*nPulse);

    % fill block diagonal
    for k = 1:nPulse
        VH(k, k)         = 2*real(s(k));
        VH(k, k+nPulse)  = 2*imag(s(k));
    end

return

function G = Gx(x, optParams)

    nPulse = optParams.nPulse;
    nfft   = optParams.nfft;
    wF     = optParams.wF;
    gamma  = optParams.gamma;

    % Reconstruct complex waveform
    s = x(1:nPulse) + 1i*x(nPulse+1:end);

    % Compute inequality constraint value
    Sf = wF .* fft(s, nfft);
    G = sum(abs(Sf).^2) - gamma * sum(abs(s).^2);

return

function VG = VGx(x, optParams)

    % numerical epsilon
    eps = 1e-7;

    % output gradient
    VG = zeros(length(x),1);

    % wrapper to evaluate inequality constraint
    f = @(z) Gx(z, optParams);

    % finite-difference loop
    for k = 1:length(x)
        e = zeros(length(x),1);
        e(k) = 1;
        VG(k) = (f(x + eps*e) - f(x - eps*e)) / (2*eps);
    end

return

function [x, hist] = ALM(x0, optParams)

    % ALM  Augmented Lagrangian Method

    % Unpack optimization settings
    alpha      = optParams.alpha;       % primal gradient step
    rho        = optParams.rho_init;    % equality penalty
    eta        = optParams.eta_init;    % inequality penalty
    maxOuter   = optParams.outerIter;   % outer AL iterations
    maxInner   = optParams.innerIter;   % max inner GD iterations
    tol        = optParams.tol;         % outer stopping tolerance
```

```
    nPulse     = optParams.nPulse;

    % Inner-loop tolerance
    if isfield(optParams, 'innerTol')
        innerTol = optParams.innerTol;
    else
        innerTol = 1e-3;
    end

    % Initialize primal and dual variables
    x      = x0(:);
    lambda = zeros(nPulse,1);
    mu     = 0;

    % History storage
    hist.obj     = zeros(maxOuter,1);
    hist.hnorm   = zeros(maxOuter,1);
    hist.gval    = zeros(maxOuter,1);
    hist.gradNorm = zeros(maxOuter,1);
    hist.rho     = zeros(maxOuter,1);
    hist.eta     = zeros(maxOuter,1);

    % 0. OUTER ALM LOOP
    for k = 1:maxOuter

        % 1. INNER LOOP
        for it = 1:maxInner

            % Objective gradient
            gJ = VJx(x, optParams);

            % Equality constraint + Jacobian
            h  = Hx(x, optParams);
            Jh = JHx(x, optParams);
            v  = lambda + rho*h;
            grad_eq = Jh' * v;

            % Inequality constraint + gradient
            g  = Gx(x, optParams);
            gG = VGx(x, optParams);
            t  = max(0, g);
            grad_ineq = (mu + eta*t) * gG;

            % Full AL gradient
            grad_AL = gJ + grad_eq + grad_ineq;

            % Gradient descent step
            x_new = x - alpha * grad_AL;

            % Inner stopping: small step or gradient
            if it == 1
                grad0_norm = norm(grad_AL);
            end

            step_norm = norm(x_new - x);
            grad_norm = norm(grad_AL);

            x = x_new;
```

```
233
234             if (grad_norm <= innerTol * (1 + grad0_norm)) ...
235                     || (step_norm <= innerTol * (1 + norm(x)))
236                 break;
237             end
238         end
239
240         % 2. RE-EVALUATE AT CURRENT x FOR DUAL UPDATES & LOGGING
241         Jval = Jx(x, optParams);
242         h    = Hx(x, optParams);
243         g    = Gx(x, optParams);
244
245         % 3. DUAL UPDATES
246         lambda = lambda + rho * h;
247         mu     = max(0, mu + eta * g);
248
249         % 4. LOG HISTORY
250         hist.obj(k)      = Jval;
251         hist.hnorm(k)    = norm(h);
252         hist.gval(k)     = g;
253         hist.gradNorm(k) = grad_norm;
254         hist.rho(k)      = rho;
255         hist.eta(k)      = eta;
256
257         fprintf('Outer %3d: Obj = %.4f, ||h|| = %.3e, g = %.3e, |grad| = %.3e\n', k, Jval,
    norm(h), g, grad_norm);
258
259         % 5. OUTER STOPPING CRITERIA
260         if (norm(h) < tol) && (g <= tol)
261             fprintf('ALM converged successfully.\n');
262             hist.obj      = hist.obj(1:k);
263             hist.hnorm    = hist.hnorm(1:k);
264             hist.gval     = hist.gval(1:k);
265             hist.gradNorm = hist.gradNorm(1:k);
266             hist.rho      = hist.rho(1:k);
267             hist.eta      = hist.eta(1:k);
268             return;
269         end
270
271     end
272
273     fprintf('ALM reached maximum outer iterations.\n');
274     hist.obj      = hist.obj(1:end);
275     hist.hnorm    = hist.hnorm(1:end);
276     hist.gval     = hist.gval(1:end);
277     hist.gradNorm = hist.gradNorm(1:end);
278
279 return
```