Appendix G - Matlab Software

Appendix G

Matlab Software

This appendix provides full listings of the matlab programs in this text. While these programs can be found online at the Cioffi website, it is helpful to have "hardcopy" listing in this file should access be limited. Readers can then retype, or cut and paste, these programs into their preferred use.

No representation is made here or anywhere as to the accuracy of programs, nor is indemnity for any reason in their use. This text focuses on educational uses for communication engineers attempting to understand and design their own systems, taking their own precautions in ensuring all representations they may subsequently make are accurate.

Appendix G sections successively capture matlab software from this text s successive chapters.

G.1 Chapter 1 Matlab Software listings

G.2 Chapter 2 Matlab Software listings

G.2.1 MAC matlab programs from Section 2.7

mu_mac.m Multiuser MAC

```
% function [Bu, GU, S0, MSWMFunb , B] = mu_bc(H, AU, Lyu , cb)
%-------------------------------------------------------------------------
% Inputs: Hu, AU , Usize, cb
% Outputs: Bu, Gunb, Wunb, S0, MSWMFunb
%
% H: noise-whitened BC matrix [H1 ; ... ; HU] (with actual noise, not wcn)
%    sum-Ly x Lx x N
% AU: Block-row square-root discrete modulators, [A1 ... AU]
%     Lx  x (U * Lx) x N
% Lyu: # of (output, Lyu) dimensions for each user U ... 1 in 1 x U row vector
% cb: = 1 if complex baseband or 2 if real baseband channel
%
% GU: unbiased precoder matrices: (Lx U) x (Lx U) x N
%     For each of U users, this is Lx x Lx matrix on each tone
% S0: sub-channel dimensional channel SNRs: (Lx U) x (Lx U) x N
% MSWMFunb: users' unbiased diagonal mean-squared whitened matched matrices
%    For each of U cells and Ntones, this is an Lx x Lyu  matrix
% Bu - users bits/symbol 1 x U
%    the user should recompute SNR if there is a cyclic prefix
% B - the user bit distributions (U x N) in cell array
%
%
%-------------------------------------------------------------------------
function [Bu, Gunb, S0, MSWMFunb, B] = mu_bc(H, AU, Lyu, cb)

%
[~ , U] = size(Lyu);
Bu=zeros(1,U);
[Ly,Lx,N]=size(H);

% Computing Ht: Ht = H*A

%-------------------------------------------------------------------------
S0=cell(U,N);
% b=cell(U,N);
for u=1:U
    for n=1:N
     S0{u,n} = zeros(Lx,Lx);
%     b{u,n}=0;
    end
end
MSWMFunb=cell(U,N);
Gunb=cell(U,N);

for u=1:U
    for n=1:N
      MSWMFunb{u,n}=zeros(Lx,Lyu(u));
      Gunb{u,n}=zeros(Lx,Lx*sum(Lyu));
    end
```

```
end

% Use compute GDFE

for u=1:U
  for n=1:N
   [~,Gunbtemp(:,:,n),~,SOtemp(:,:,n),MSWMFtemp(:,:,n)]= computeGDFE( ...
    H(sum(Lyu(1:u))-Lyu(u)+1:sum(Lyu(1:u)),:,n), AU(:,:,n),cb);
   Gunb{u,n}=Gunbtemp(Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),:,n);
   SO{u,n}=SOtemp(Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),...
       Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),n);
   MSWMFunb{u,n}=MSWMFtemp(Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),:,n);
   B{u,n}= (1/cb)*log2(det(SO{u,n}));
  end
   Bu(u)=sum([B{u,:}],2);
end
```

SWC.m and macmax.m Simultaneous Water-Filling for MAC

```
% function [Rxx, bsum , bsum_lin] = SWF(Eu, H, user_ind, Rnn, cb)
%
% Simultaneous water-filling MAC max rate sum (linear and nonlinear GDFE)
% The input is space-time domain h, and the user can specify a temporal
% block symbol size N (essentially an FFT size).
%
% Inputs:
% Eu  U x 1 energy/SAMPLE vector. Single scalar equal energy all users
%     any (N/N+nu) scaling should occur BEFORE input to this program.
% H   The FREQUENCY-DOMAIN Ly x sum(Lx(u)) x N MIMO channel for all users.
%     N is determined from size(H) where N = # tones
%     (equally spaced over (0,1/T) at N/T.
%     if time-domain h, H = 1/sqrt(N)*fft(h, N, 3)
% Lxu 1xU vector of each user's number of antennas
% Rnn The Ly x Ly x N noise-autocorrelation tensor (last index is per tone)
% cb  cb = 1 for complex, cb=2 for real baseband
%
% Outputs:
% Rxx A block-diagonal psd matrix with the input autocorrelation for each
%     user on each tone. Rxx has size (sum(Lx(u)) x sum(Lx(u)) x N .
%     sum trace(Rxx) over tones and spatial dimensions equal the Eu
% bsum the maximum rate sum.
% bsum  bsum_lin - the maximum sum rate with a linear receiver
%     b is an internal convergence sum rate value, not output
%
% This program is modified version of one originally supplied by student
% Chris Baca

function [Rxx, bsum, bsum_lin] = SWF(Eu, H, Lxu , Rnn, cb)
U = numel(Lxu);
[Ly, Lx_sum, N] = size(H);

if numel(Eu) == 1
    Eu = Eu*ones(U);
end
```

```
i = 0;
for n=1:N
Rxx(:,:,n) = diag(zeros(Lx_sum, N));
H(:,:,n)=inv(sqrtm(Rnn(:,:,n)))*H(:,:,n);
end
b = zeros(U, 1);

cum_index=1;
user_ind(1)=1;
for u=2:U
    cum_index = cum_index + Lxu(u);
    user_ind(u) = cum_index;
end

for u = 1:U

    st = user_ind(u);

    if u < U
        en = user_ind(u+1)-1;
    else
        en = Lx_sum;
    end

    for n = 1:N
        Rxx(st:en, st:en,n) = Eu(u)*eye(en-st+1);
    end
end

b = zeros(U);
while 1
    b_prev = b;

    for u = 1:U
        %new user
        st = user_ind(u);

            if u < U
                en = user_ind(u+1)-1;
            else
                en = Lx_sum;
            end
        M_u = zeros(en-st+1,en-st+1, N);
        g_u = [];
       Rnn_un=zeros(Ly,Ly,N);
        for n = 1:N
            %new tone
            Rnn_un(:,:,n) = eye(Ly);
            for v = 1:U
                if v ~= u
                    st = user_ind(v);
                    if v < U
                        en = user_ind(v+1)-1;
```

```
                    else
                        en = Lx_sum;
                    end

                    Hvn = H(:, st:en, n);
                    Rxxvn = Rxx(st:en, st:en, n);
                    Rnn_un(:,:,n) = Rnn_un(:,:,n) + Hvn*Rxxvn*Hvn';
                end
            end

            st = user_ind(u);

            if u < U
                en = user_ind(u+1)-1;
            else
                en = Lx_sum;
            end


            Hun = H(:, st:en, n);

            Hun_til = sqrtm(inv(Rnn_un(:,:,n)))*Hun;

            [F, D, M_n] = svd(Hun_til);
            s = svd(Hun_til);

            M_u(:,:,n) = M_n;
            g_u(:,n) = s.^2;

        end

        g_flat = reshape(g_u, [1, numel(g_u)]);
%        [g_sort, ind] = sort(g_flat, 'descend');
        [B,E,L_star] = waterfill_gn(g_flat,Eu(u),0, cb);

        L = numel(g_flat);
%        Etot = N*Eu(u);
%        j = L;

%        e = zeros(1,L);
%       size(E)
%        e(1:L_star)=E;

        e = reshape(E, [en-st+1, N]);
        b(u,1) = sum(B);

        for n = 1:N
            Rxx(st:en, st:en, n) = M_u(:,:,n)*diag(e(:,n))*M_u(:,:,n)';
        end
    end

    i = i+1;

    if norm(rms(b_prev-b)) <= 1e-5
```

```
        break
    end
    if i>1000
        break
    end
end
%b = b(:,1);
%Hcell = mat2cell(H,Ly,Lx_sum,ones(1,N));
%Hexpand = [blkdiag(Hcell{1,1,:})]
%Rcell = mat2cell(Rxx,Ly,Lx_sum,ones(1,N));
%Rcell = mat2cell(Rxx,Lx_sum,Lx_sum,ones(1,N));
%Rxxexpand = [blkdiag(Rcell{1,1,:})]
%bsum = log2(det(eye(Lx_sum*N)+Hexpand*Rxxexpand*Hexpand'));
%bsum = log2(det(eye(Ly*N)+Hexpand*Rxxexpand*Hexpand'));
bsum=0;
for n=1:N
    bsum=bsum+(1/cb)*log2(det(eye(Ly)+H(:,:,n)*Rxx(:,:,n)*H(:,:,n)'));
end
bsum=real(bsum);
bs=zeros(1,U);
bsum_lin=0;
for u=1:U
    indices=user_ind(u):user_ind(u)+Lxu(u)-1;
    for n=1:N
     bs(u)=bs(u)+(1/cb)*(log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)')) - log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)'- H(:,indices,n)*Rxx(indices, ...
       indices,n)*H(:,indices,n)')));
    end
    bsum_lin=bsum_lin+real(bs(u));
end

% function [Rxx, bsum , bsum_lin] = macmax(Esum, h, Lxu, N , cb)
%
% Simultaneous water-filling Esum MAC max rate sum (linear & nonlinear GDFE)
% The input is space-time domain h, and the user can specify a temporal
% block symbol size N (essentially an FFT size).
%
% This program uses the CVX package
%
% the inputs are:
% Esum The sum-user energy/SAMPLE scalar.
%      This will be increased by the number of tones N by this program.
%      Each user energy should be scaled by N/(N+nu)if there is cyclic prefix
%      This energy is the trace of the corresponding user Rxx (u)
%      The sum energy is compouted as the sum of the Eu components
%      internally.
% h    The TIME-DOMAIN Ly x sum(Lx(u)) x N channel for all users
% Lxu  The number of antennas for each user 1 x U
% N    The number of used tones (equally spaced over (0,1/T) at N/T.
% cb   cb = 1 for complex, cb=2 for real baseband
%
% the outputs are:
% Rxx A block-diagonal psd matrix with the input autocorrelation for each
```

```
%     user on each tone. Rxx has size (sum(Lx(u)) x sum(Lx(u)) x N .
%     sum trace(Rxx) over tones and spatial dimensions equal the Eu
% bsum the maximum rate sum.
% bsum  bsum_lin - the maximum sum rate with a linear receiver
%
%     b is an internal convergence (vector, rms) value, but not sum rate

function [Rxx, bsum, bsum_lin] = macmax(Esum, h, Lxu, N , cb)

H = fft(h, N, 3); % scaled so that N factor no longer needed in rate calculation
[Ly, Lx, ~] = size(H);
Esum=N*Esum;
if numel(Lxu) > 1
    U = numel(Lxu);
    if sum(Lxu)~=Lx
        error('mismatch between sum of Lxu and Lx');
    end
elseif (Lx/Lxu)~= floor(Lx/Lxu)
    error('invalid Lxu');
else
    U = Lx/Lxu;
    Lxu = Lxu*ones(1,U);
end

idx_end = cumsum(Lxu);
idx_start = [1, idx_end(1:end-1)+1];
idx_exp_end = N*idx_end;
idx_exp_start = [1, idx_exp_end(1:end-1)+1];

Hcell=mat2cell(H, Ly, Lxu, ones(1,N));
Hexpand = zeros(N*Ly,N*Lx);
for u = 1:U
    Hexpand(:,idx_exp_start(u):idx_exp_end(u)) = blkdiag(Hcell{1,u,:});
end

Lxu_max = max(Lxu);
cvx_begin quiet
cvx_solver mosek
    variable Rxxun(Lxu_max,Lxu_max,N,U) hermitian semidefinite
    Rxxun_cell = reshape(num2cell(Rxxun, [1,2,3]), 1, U);
    for u = 1:U
        Rxxun_cell{u} = Rxxun_cell{u}(1:Lxu(u), 1:Lxu(u),:);
        Rxxun_cell{u} = reshape(num2cell(Rxxun_cell{u},[1,2]), 1, N);
    end
    Rxxun_cell = [Rxxun_cell{:}];
    Rxx = blkdiag(Rxxun_cell{:});
    maximize (1/cb*log2(exp(1))*log_det(eye(N*Ly) + Hexpand*Rxx*Hexpand'))
    subject to
        trace(Rxx) <= Esum;
cvx_end

bsum=cvx_optval;

Rxxun_cell = reshape(num2cell(Rxxun, [1,2,3]), U, 1);
```

```
for u = 1:U
    Rxxun_cell{u} = Rxxun_cell{u}(1:Lxu(u), 1:Lxu(u),:);
    Rxxun_cell{u} = reshape(num2cell(Rxxun_cell{u},[1,2]), 1, N);
end
Rxxun_cell = vertcat(Rxxun_cell{:}); % U*N cell array

Rxx = zeros(Lx,Lx,N);
for n=1:N
    Rxx(:,:,n) = blkdiag(Rxxun_cell{:,n});
end

bs=zeros(1,U);
bsum_lin=0;
for u=1:U
    indices=idx_start(u):idx_end(u);
    for n=1:N
     bs(u)=bs(u)+(1/cb)*(log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)')) - log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)'- H(:,indices,n)*Rxx(indices, ...
       indices,n)*H(:,indices,n)')));
    end
    bsum_lin=bsum_lin+real(bs(u));
end
end
```

G.2.2 BC matlab programs from Section 2.8

mu_bc.m Multiuser BC

```
% function [Bu, GU, S0, MSWMFunb , B] = mu_bc(H, AU, Lyu , cb)
%-------------------------------------------------------------------------
% Inputs: Hu, AU , Usize, cb
% Outputs: Bu, Gunb, Wunb, S0, MSWMFunb
%
% H: noise-whitened BC matrix [H1 ; ... ; HU] (with actual noise, not wcn)
%    sum-Ly x Lx x N
% AU: Block-row square-root discrete modulators, [A1 ... AU]
%     Lx  x (U * Lx) x N
% Lyu: # of (output, Lyu) dimensions for each user U ... 1 in 1 x U row vector
% cb: = 1 if complex baseband or 2 if real baseband channel
%
% GU: unbiased precoder matrices: (Lx U) x (Lx U) x N
% S0: sub-channel dimensional channel SNRs: (Lx U) x (Lx U) x N
% MSWMFunb: users' unbiased diagonal mean-squared whitened matched matrices
%    (Lx U) x sum-Lyu
% Bu - users bits/symbol 1 x U
%    the user should recompute SNR if there is a cyclic prefix
% B - the user bit distributions
%
%
%-------------------------------------------------------------------------
function [Bu, Gunb, S0, MSWMFunb, B] = mu_bc(H, AU, Lyu, cb)

%
[~ , U] = size(Lyu);
```

```
Bu=zeros(1,U);
[Ly,Lx,N]=size(H);

% Computing Ht: Ht = H*A

%-------------------------------------------------------------------------
SO=cell(U,N);
% b=cell(U,N);
for u=1:U
    for n=1:N
     SO{u,n} = zeros(Lx,Lx);
%     b{u,n}=0;
    end
end
MSWMFunb=cell(U,N);
Gunb=cell(U,N);

for u=1:U
    for n=1:N
      MSWMFunb{u,n}=zeros(Lx,Lyu(u));
      Gunb{u,n}=zeros(Lx,Lx*sum(Lyu));
    end
end

% Use compute GDFE

for u=1:U
  for n=1:N
   [~,Gunbtemp(:,:,n),~,SOtemp(:,:,n),MSWMFtemp(:,:,n)]= computeGDFE( ...
    H(sum(Lyu(1:u))-Lyu(u)+1:sum(Lyu(1:u)),:,n), AU(:,:,n),cb);
   Gunb{u,n}=Gunbtemp(Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),:,n);
   SO{u,n}=SOtemp(Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),...
       Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),n);
   MSWMFunb{u,n}=MSWMFtemp(Lx*(sum(Lyu(1:u))-Lyu(u))+1:Lx*(sum(Lyu(1:u))),:,n);
   B{u,n}= (1/cb)*log2(det(SO{u,n}));
  end
   Bu(u)=sum([B{u,:}],2);
end
```

rq.m

```
function [R,Q,P] = rq(A, t)
%rq Triangular-orthogonal decomposition
%   [R,Q] = rq(A), where A is m-by-n, produces an m-by-n upper triangular
%   matrix R and an n-by-n unitary matrix Q so that A = R*Q'.
%
%   [R,Q] = rq(A,0) produces the "economy size" decomposigion.
%   If m<n, only the last m columns of R and Q are computed. If m>=n, this
%   is teh same as [R,Q] = rq(A).
%
%   [R,Q,P] = rq(A,_) also returns a permutation order P so that
%   A(P,:) = R*Q'.

reverse_idx_func = @(X) X(end:-1:1, end:-1:1);
```

```
if nargout == 2
    if nargin == 1
        [std_Q, std_R] = qr(reverse_idx_func(A)');
    else
        [std_Q, std_R] = qr(reverse_idx_func(A)', 0);
    end
    P = 0;
elseif nargout == 3
    if nargin == 1
        [std_Q, std_R, std_P] = qr(reverse_idx_func(A)');
        P = reverse_idx_func(std_P);
        [P,~] = find(P);
    else
        [std_Q, std_R, std_P] = qr(reverse_idx_func(A)', 0);
        size_P = max(std_P);
        P = (std_P'==1:size_P)';
        P = reverse_idx_func(P);
        [P,~] = find(P);
    end
    P = P';

end
Q = reverse_idx_func(std_Q);
R = reverse_idx_func(std_R)';

end
```

bcmax.m

```
% function [Rxx, Rwcn, bmax] = bcmax(iRxx, H, Lyu)
%
%  Uses cvx_wcnoise.m and rate-adaptive waterfill.m (Lagrange
%  Multiplier based)
%   Inputs:
%       - iRxx: initial input autocorrelation array, size is Lx x Lx x N.
%           Only the sum of traces matters, so can initialize to any valid
%           autocorrelation matrix Rxx to run wcnoise.
%           needs to include factor N/(N+nu) if nu ~= 0
%       - H: channel response, size is Ly x Lx x N, w/o sqrt(N)
%       normalization
%       - Lyu: array number of antennas at each user; scalar Lyu means same for all
%   Outputs:
%       - Rxx: optimized input autocorrelation Lx x Lx x N
%       - Rwcn: optimized worst-case noise autocorrelation, with white local noise
%               Ly x Ly x N
%               SO IF H is noise-whitened for Rnn, then actual noise is
%               Rwcn^(1/2)*Rnn*Rwcn^(*/2)
%       - b: maximum sum rate/real-dimension - user must mult by 2 for
%            complex case

function [Rxx, Rwcn, bmin] = bcmax(iRxx, H, Lyu)

    [Ly,Lx,N] = size(H);
    total_en = trace(sum(iRxx, 3));
```

```
    bmin = 0;
    ib=zeros(1,N);
    Rwcn = zeros(Ly,Ly,N);
    for n=1:N % worst-case noise independent over tones for BC
    [Rwcn(:,:,n), ib(n)] = cvx_wcnoise(iRxx(:,:,n), H(:,:,n), Lyu);
    end

    while (abs(sum(ib) - bmin) > 1e-5) %tolerance
        % uncomment the following two lines to see how the loop progresses
        %bmax
        bmin = sum(ib);
     % Vector Coding Gains for each tone
        M=zeros(Lx,Lx,N);
        gains = zeros(Lx,N);
        for n=1:N
            [V,D,~]=svd(Rwcn(:,:,n));
            sqD = sqrt(D).*(D>1e-6);
            invsqRwcn = V*pinv(sqD)*V';
            Htil = invsqRwcn*H(:,:,n);
            [~, g, M(:,:,n)] = svd(Htil);
            g=diag(g);
            gains(1:length(g),n)=g.^2;
        end
     % Water-filling step
        En = waterfill(total_en, reshape(gains',N*Lx,1), 1);
        %bvec=0.5*log2(ones(N*Lx,1)+En.*reshape(gains',N*Lx,1))';
        %bmax = real(sum(bvec));
        En=reshape(En,N,Lx)';
     % update worst-case-noise step
        for n=1:N
            newiRxx = M(:,:,n)*diag(En(:,n))*M(:,:,n)';
            iRxx(:,:,n) = (iRxx(:,:,n)+newiRxx)/2;

            %iRxx(:,:,n) = M(:,:,n)*diag(En(:,n))*M(:,:,n)';
           [Rwcn(:,:,n), ib(:,n)] = cvx_wcnoise(iRxx(:,:,n), H(:,:,n), Lyu);
        ib(:,n)=real(ib(:,n));
        end
    end
    Rxx = iRxx;
    bmin = sum(ib);
end
```

Worst-Case Noise program wcnoise.m and cvx_wcnoise.m

```
% function [Rwcn, bsum] = wcnoise(Rxx, H, Ly, dual_gap, nerr)
%
% inputs
% H is U*Ly by Lx, where
%       Ly is the (constant) number of antennas/receiver,
%       Lx is the number of transmit antennas, and
%       U is the number of users. H can be a complex matrix
% Rxx is the Lx by Lx input (nonsingular) autocorrelation matrix
%       which can be complex (and Hermitian!)
% dual_gap is the duality gap, defaulting to 1e-6 in wcnoise
```

```
% nerr is Newton's method acceptable error, defaulting to 1e-4 in wcnoise
%
% outputs
% Rwcn is the U*Ly by U*Ly worst-case-noise autocorrelation matrix.
% bsum is the rate-sum/real-dimension.
%
% I = 0.5 * log(det(H*Rxx*H'+Rwcn)/det(Rwcn)), Rwcn has Ly x Ly diagonal blocks
%     that are each equal to an identity matrix
%
function [Rwcn, bsum] = wcnoise(Rxx, H, Ly, dual_gap, nerr)

switch nargin
    case 3
        dual_gap = 1e-6;
        nerr = 1e-4;
    case 4
        nerr = 1e-4;
end

[n,m] = size(H);
K = n / Ly;

% make sure the input Rxx is strickly hermitian
Rxx = (real(Rxx)+real(Rxx'))/2 + sqrt(-1) * (imag(Rxx)+imag(Rxx'))/2;
for i = 1:m
  Rxx(i,i) = real(Rxx(i,i));
end

if min(eig(Rxx)) < 0
  fprintf('Input Rxx is not positive semi-definite! The answers may be wrong')
end

count = 1;

A = zeros(n,n,n*n);
for i = 1:n
   A(i,i,count) = 1;
   count = count+1;
end



for i = 2:n
   for j = 1:i - 1
      A(i,j,count) = 1;
      A(j,i,count) = 1;
      count = count+1;
   end
end

complexI = sqrt(-1);

for i = 2:n
   for j = 1:i - 1
```

```
      A(i,j,count) = complexI;
      A(j,i,count) = -complexI;
      count = count+1;
   end
end


map = zeros(n,n);
for i = 1:K
   map((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = ones(Ly,Ly);
end



NT_max_it = 1000;                                % Maximum number of Newton's
                                                 % method iterations
%dual_gap = 1e-6;
mu = 10;                                         % step size for t
alpha = 0.001;                                   % back tracking line search parameters
beta = 0.5;

count = 1;
%n%nerr = 1e-4;                                     % acceptable error for inner loop
                                                 % Newton's method

v_0 = zeros(n*n,1);                              % Strictly feasible point;
v_0(1:n) = 0.5 *  ones(n,1);
v = v_0;
t = 1;
l_v = 1;                                         % lambda(v) for newton's method termination



while (1+n*n)/t > dual_gap
   t =  t * mu;
   l_v = 1;
   count = 1;
   while l_v > nerr && count < NT_max_it

      f_val = 0;                                 % calculating function value
      Rwcn = zeros(n,n);
      Rzprime = zeros(n,n);

      for i = 1:n*n                              % computing Rz
         Rwcn = Rwcn + v(i) * A(:,:,i);
      end

      for i = 1:K
          Rzprime((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = Rwcn((i-1) * Ly + 1:i * Ly,(i-1) *
      end
      % the real operation is not needed below as determinant of a Hermitian
      % matrix is always real, it is included for avoiding numerical issues
      f_val = t * log(real(det(H * Rxx * H' + Rwcn))) - (t + 1) * log(real(det(Rwcn))) - log(real(de
```

```
S = inv(H * Rxx * H' + Rwcn);
Q = inv(eye(n) - Rzprime);
Rz_inv = inv(Rwcn);

g = zeros(n*n,1);
h = zeros(n*n,n*n);

for i = 1:n*n
   g(i) = t * trace(A(:,:,i) * S) - (t + 1) * trace(A(:,:,i) * Rz_inv)...
        + (sum(sum(A(:,:,i) .* map)) ~= 0) * trace(A(:,:,i) * Q) ;  % gradient
end

% make sure g is always real (gradient of a real function with respect to real variables)
g = real(g);

for i = 1:n*n
   for j = 1:n*n
      h(i,j) = -t * trace(A(:,:,i) * S * A(:,:,j) * S) + (t + 1) * trace(A(:,:,i) * Rz_inv * A
               + (sum(sum(A(:,:,i) .* map)) ~= 0) * (sum(sum(A(:,:,j) .* map)) ~= 0) * trace(A
   end
end

% make sure h is always real (gradient of a real gradient values)
% make sure he is symmetric
h = (h+h')/2;
h = real(h);



dv = real(-h\g);                          % search direction

v_min = min(abs(dv));

s = 1;                                    % checking v = v+s*dx feasible
                                          % and also back tracking algorithm

while 1
   v_new = v + s * dv;

   Rz_new = zeros(n,n);

   for i = 1:n*n
      Rz_new = Rz_new + v_new(i) * A(:,:,i);
   end

   for i = 1:K
       Rzprime((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = Rz_new((i-1) * Ly + 1:i * Ly,(i-
   end

   f_new = t * log(real(det(H * Rxx * H' + Rz_new))) - (t + 1) * log(real(det(Rz_new))) - log(


   feas_check = 1;
```

```
        if real(eig(Rz_new)) > zeros(n,1)
            feas_check = 1;
        else
            feas_check = 0;
        end
        if real(eig(eye(n) - Rzprime)) > zeros(n,1)
            feas_check = 1;
        else
            feas_check = 0;
        end

        if feas_check == 1
          feas_check = feas_check * (f_new < f_val + alpha * s * g' * dv);
        end

        if feas_check == 1
          break
        end

        s = s * beta;

        if s < 1e-30 * v_min
          % s is too small, break the while loop
          s = 0;
          break;
        end

      end

      v = v + s * dv;                          % update v
      l_v = -g'*dv ;                           % lambda(v)^2 for Newton's method

      if s > 0
        % valid step size returned by the line search algorithm above
        count = count + 1;                     % number of Newtons method iterations
      else
        % s = 0 and we could not find a feasible newton step, skip this iteration and go to a larger
        count = NT_max_it;
      end
   end
   if l_v >= nerr
     l_v
     fprintf('Inner loop Newton method failed to converge for ')
     t
   end

end

Rwcn = zeros(n,n);

for i = 1:n*n
   Rwcn = Rwcn + v(i) * A(:,:,i);
end
```

```
for i = 1:K
    Rwcn((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = eye(Ly);
end
bsum = 0.5 * log2(real(det(H * Rxx * H' + Rwcn)/det(Rwcn)));
```

cvx_wcnoise.m This program requires some explanation of the equivalent reformation the worst-case noise criteria so that CVX's facilities apply.

The basic WCN problem is (defining $\widetilde{H} \triangleq H \cdot R_{\boldsymbol{xx}}^{1/2}$ for any square root matrix of $R_{\boldsymbol{xx}}$)

$$P1: \text{maximize} \quad \frac{1}{2}\log_2 \left| I - \widetilde{H}^* \cdot \left(R_{\boldsymbol{nn}} + \widetilde{H} \cdot \widetilde{H}^* \right)^+ \cdot \widetilde{H} \right| \tag{G.1}$$

$$s.t. \quad R_{\boldsymbol{nn}}(u) = I_{L_{y,u}} \ , \ \forall \ u \in \boldsymbol{U} \tag{G.2}$$

$$R_{\boldsymbol{nn}} \succeq 0 \ . \tag{G.3}$$

The space of positive semidefinite matrices is $\mathbb{S}_+^{L_x}$ and has a matrix $Z \in \mathbb{S}_+^{L_x}$ such that P1 is equivalent to:

$$P2: \text{maximize} \quad \frac{1}{2}\log_2 |I - Z| \tag{G.4}$$

$$s.t. \quad R_{\boldsymbol{nn}}(u) = I_{L_{y,u}} \ , \ \forall \ u \in \boldsymbol{U} \tag{G.5}$$

$$R_{\boldsymbol{nn}} \succeq 0 \tag{G.6}$$

$$Z \succeq I - \widetilde{H}^* \cdot \left(R_{\boldsymbol{nn}} + \widetilde{H} \cdot \widetilde{H}^* \right)^+ \cdot \widetilde{H} \ . \tag{G.7}$$

Essentially maximizing $|I - Z|$ is like minimizing Z for a positive semidefinite matrix and the lower bound on Z in the constraints prevents A from going below the value that makes the expression equal to the mutual information's determinant. The program cvx_wcnoise.m uses the following equivalent formation:

$$P3: \text{maximize} \quad \frac{1}{2}\log_2 |I - Z| \tag{G.8}$$

$$s.t. \quad R_{\boldsymbol{nn}}(u) = I_{L_{y,u}} \ , \ \forall \ u \in \boldsymbol{U} \tag{G.9}$$

$$\begin{bmatrix} R_{\boldsymbol{nn}} + \widetilde{H} \cdot \widetilde{H}^* & \widetilde{H} \\ \widetilde{H}^* & Z \end{bmatrix} \succeq 0 \tag{G.10}$$

$$R_{\boldsymbol{nn}} \succeq 0 \tag{G.11}$$

$$Z \succeq 0 \ . \tag{G.12}$$

The program also constraints the imaginary part of $\mathcal{S}_{wcn}$ to be zero to help with numerical issues.

```
% function [Rnn, sumRatebar, S1, S2, S3, S4] = cvx_wcnoise(Rxx, H, Lyu)
%
%cvx_wcnoise This function computes the worst-case noise for any given input
%autocorrelation Rxx and channel matrix.
%   Arguments:
%       - Rxx: input autocorrelation, size(Lx, Lx)
%       - H: channel response, size (Ly, Lx)
%       - Lyu: number of antennas at each user, scalar/vector of length U
%   Outputs:
%       - Rnn: worst-case noise autocorrelation, with white local noise
%       - sumRatebar: maximum sum rate/real-dimension
%       - S1 is the lagrange multiplier for the real part of Rnn diagonal
%       elements.  Zero values indicate secondary users.
%       - S2 is the imaginary part
%       - S3 is for the positive semidefinite constraint on Rwcn
%       - S4 is for a larger Schur compliment used in the optimization
```

```
%
function [Rnn, sumRatebar, S1, S2, S3, S4] = cvx_wcnoise(Rxx, H, Lyu)
[Ly,Lx] = size(H);
if length(Lyu) == 1
    U = Ly/Lyu;
    Lyus = ones(1,U)*Lyu;
else
    U = length(Lyu);
    Lyus = Lyu;
end
cum_Lyu = cumsum(Lyus);
cum_Lyu = [0, cum_Lyu(1:end-1)];
us = 1:U;

[Q,D,~]=svd(Rxx);
rD = rank(D);
sqD = sqrt(diag(D));
sqD(rD+1:end) = 0;
Htilde = H*Q*diag(sqD);
cvx_begin sdp quiet
cvx_solver mosek
cvx_precision high
variable Rnn(Ly, Ly) hermitian
variable Z(Lx, Lx) hermitian semidefinite
dual variable S1{U}
dual variable S2{U}
dual variable S3
dual variable S4
maximize log_det(eye(Lx) - Z)
subject to
for u = us
    S1{u}: eye(Lyus(u))==real(Rnn(cum_Lyu(u) + (1:Lyus(u)), cum_Lyu(u) + (1:Lyus(u))));
    S2{u}: 0==imag(Rnn(cum_Lyu(u) + (1:Lyus(u)), cum_Lyu(u) + (1:Lyus(u))));
end
S3: Rnn == hermitian_semidefinite(Ly);
S4: [Rnn + Htilde*Htilde', Htilde; Htilde', Z] == hermitian_semidefinite(Ly+Lx);
cvx_end

sumRatebar = -0.5*log2(exp(1))*cvx_optval;
```

waterfill.m

```
% function [En] = waterfill(total_en, gn, gap)
%
% Waterfill for any set of channel gains, just produces energy
%
% Water accepts the channel gains as input, and finds water-fill solution
% for the given (input) total energy, with codes of gap "gap."
% The gap is specified as linear (so not in dB).
% The program uses any gain set and does not compute an SNR, nor does it
% know the original of the gains (so no guard-band penalty is presumed)
%
% Inputs
%       total_en is the total energy/symbol for all dimensions
```

```
%       gn is a vector containing all the gains (energy xfer)
%       gap is the LINEAR gap of the code, so gap =1 means 0 dB
% Output
%       En is the set of energies for the original dimensions

function [En] = waterfill(total_en, gn, gap)
    Ntot = numel(gn);
    Ex_bar = total_en/Ntot;
    [gn_sorted, Index]=sort(gn, 'descend');  % sort gain, and get Index
    [~,InvIndex] = sort(Index);
    % Ntot = numel(gn);
    num_zero_gn = length(find(gn_sorted == 0)); %number of zero gain subchannels
    Nstar= Ntot - num_zero_gn;
    % Number of used channels,
    % start from Ntot - (number of zero gain subchannels)

    while(1)
        K=1/Nstar*(Ntot*Ex_bar+gap*sum(1./gn_sorted(1:Nstar)));
        En_min=K-gap/gn_sorted(Nstar); % En_min occurs in the worst channel
        if (En_min<0)
            Nstar=Nstar-1;  % If negative En, continue with less channels
        else
            break;       % If all En positive, done.
        end
    end

    En=K-gap./gn_sorted(1:Nstar);  % Calculate En
    En = [En; zeros(Ntot-Nstar, 1)];        % Pad zeros to revert to unsorted order
    En = En(InvIndex);
end
```

G.3 Chapter 3 Matlab Software listings

The dfecolor Program

```
function [dfseSNR,w_t,opt_delay]=dfsecolorsnr(l,h,nff,nbb,delay,Ex,noise);
function [dfseSNR,w_t,opt_delay]=dfsecolorsnr(l,h,nff,nbb,delay,Ex,noise);
% ------------------------------------------------------------
% [dfseSNR,w_t] = dfecolor(l,p,nff,nbb,delay,Ex,noise);
%
% INPUTS
% l     = oversampling factor
% h     = pulse response, oversampled at l (size)
% nff   = number of feed-forward taps
% nbb   = number of feedback taps
% delay = delay of system <= nff+length of p - 2 - nbb
%         if delay = -1, then choose best delay
% Ex    = average energy of signals
% noise = noise autocorrelation vector (size l*nff)
%   so white noise is [1 zeros(l*nff-1)]
% NOTE: noise is assumed to be stationary
%
% OUTPUTS
% dfseSNR = equalizer SNR, unbiased in dB
% w_t = equalizer coefficients [w -b]
% opt_delay = optimal delay found if delay =-1 option used.
%             otherwise, returns delay value passed to function
% created 4/96;
% ------------------------------------------------------------

size = length(h);
nu = ceil(size/l)-1;
h = [h zeros(1,(nu+1)*l-size)];

% error check
if nff<=0
  error('number of feed-forward taps > 0');
end
if delay > (nff+nu-1-nbb)
  error('delay must be <=  (nff+(length of p)-2-nbb)');
elseif delay < -1
  error('delay must be >= -1');
elseif delay == -1
    delay = 0:1:nff+nu-1-nbb;
end

%form htmp = [h_0 h_1 ... h_nu] where h_i=[h(i*l) h(i*l-1)... h((i-1)*l+1)
htmp(1:l,1) = [h(1); zeros(l-1,1)];
for i=1:nu
  htmp(1:l,i+1) = conj((h(i*l+1:-1:(i-1)*l+2))');
end

%form matrix H, vector channel matrix
H = zeros(nff*l+nbb,nff+nu);
for i=1:nff,
  H(((i-1)*l+1):(i*l),i:(i+nu)) = htmp;
```

```
end

%precompute Rn matrix - constant for all delays
Rn  = zeros(nff*l+nbb);
Rn(1:nff*l,1:nff*l) = l*toeplitz(noise);

dfseSNR = -100;
H_init = H;
%loop over all possible delays
for d = delay,
    H = H_init;
    H(nff*l+1:nff*l+nbb,d+2:d+1+nbb) = eye(nbb);
    %P
    temp= zeros(1,nff+nu);
    temp(d+1)=1;
    %construct matrices
    Ry  = Ex*H*H' + Rn;
    Rxy = Ex*temp*H';
    new_w_t = Rxy*inv(Ry);
    sigma_dfse = Ex - real(new_w_t*Rxy');
    new_dfseSNR = 10*log10(Ex/sigma_dfse - 1);
    %save setting of this delay if best performance so far
    if new_dfseSNR >= dfseSNR
        w_t = new_w_t;
        dfseSNR = new_dfseSNR;
        opt_delay = d;
    end
end
```

The dfeRAKE.m program

```
% function [dfseSNR,W,b]=dfsecolorsnr(l,p,nff,nbb,delay,Ex,noise);
% DFE design program for RAKE receiver
%
%
% Inputs
% l     = oversampling factor
% L       = No. of fingers in RAKE
% p     = pulse response matrix, oversampled at l (size),
%         each row corresponding to a diversity path
% nff   = number of feedforward taps for each RAKE finger
% nbb   = number of feedback taps
% delay = delay of system <= nff+length of p - 2 - nbb
%         if delay = -1, then choose best delay
% Ex    = average energy of signals
% noise = noise autocorrelation vector (size L x l*nff)
% NOTE: noise is assumed to be stationary, but may be spatially correlated
%
% outputs:
% dfseSNR = equalizer SNR, unbiased in dB
% ------------------------------
function [dfseSNR,W,b]=dfsecolorsnr(l,p,nff,nbb,delay,Ex,noise);

siz = size(p,2);
```

```
L=size(p,1);
nu = ceil(siz/l)-1;
p = [p zeros(L,(nu+1)*l-siz)];


% error check
if nff<=0
  error('number of feedforward taps must be > 0');
end
if delay > (nff+nu-1-nbb)
  error('delay must be <=  (nff+(length of p)-2-nbb)');
end
if delay < -1
  error('delay must be >= 0');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%if length(noise) ~= L*l*nff
%   error('Length of noise autocorrelation vector must be L x(l*nff');
%end
if size(noise,2) ~= l*nff | size(noise,1) ~= L
   error('Size of noise autocorrelation matrix must be L x l*nff ');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%form ptmp = [p_0 p_1 ... p_nu] where p_i=[p(i*l) p(i*l-1)... p((i-1)*l+1)
for m=1:L
   ptmp((m-1)*l+1:m*l,1) = [p(m,1); zeros((l-1),1)];
end
for k=1:nu
   for m=1:L
      ptmp((m-1)*l+1:m*l,k+1) = conj((p(m,k*l+1:-1:(k-1)*l+2))');
   end
end
ptmp;


% form matrix P, vector channel matrix
P = zeros(nff*l*L+nbb,nff+nu);


%First construct the P matrix as in MMSE-LE
for k=1:nff,
  P(((k-1)*l*L+1):(k*l*L),k:(k+nu)) = ptmp;
end

%Add in part needed for the feedback
P(nff*l*L+1:nff*l*L+nbb,delay+2:delay+1+nbb) = eye(nbb);
temp= zeros(1,nff+nu);
temp(delay+1)=1;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Rn  = zeros(nff*l*L+nbb);

for i = 1:L
    n_t = toeplitz(noise(i,:));

    for j = 1:l:l*nff
        for k = 1:l:l*nff
            Rn((i-1)*l+(j-1)*L+1:i*l+(j-1)*L, (i-1)*l+(k-1)*L+1:i*l+(k-1)*L) = n_t(j:j+l-1,k:k+l-1);
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Ex*P*P';
Ry  = Ex*P*P' + Rn;
Rxy = Ex*temp*P';
IRy=inv(Ry);
w_t = Rxy*IRy;
%Reshape the w_t matrix into the RAKE filter bank and feedback matrices
ww=reshape(w_t(1:nff*l*L),l*L,nff);
for m=1:L
   W(m,:)=reshape(ww((m-1)*l+1:m*l,1:nff),1,l*nff);
end
b=-w_t(nff*l*L+1:nff*l*L+nbb);
sigma_dfse = Ex - w_t*Rxy';
dfseSNR = 10*log10(Ex/sigma_dfse - 1);
```

G.4 Chapter 4 Matlab Software listings

G.4.1 waterfill_gn.m program

```
% function [bn, en, Nstar] = waterfill_gn(gn, E_bar, gap, cb)
%
% This function allows the energy budget to be divided by the total number
% of dimensions (unlike waterfill, for which only the total energy is
% specified).  THIS FUNCTION ACCEPTS ENERGY/INPUT DIMENSION NOT TOTAL
% ENERGY.
%
% So, for instance, if there are (N+ nu) temporal dimensions
% with only N carrying energy, the normalization in construction E_bar
% would divide by N+nu.  Instead if spatial with Lx dimensions, E_bar would
% divide by Lx.  Basically, the program user of waterfill_gn decides this
% normalization.
%
% INPUT
% gn is the channel gain (a row vector).
% E_bar is the normalized power constraint (E_total / Ntot)
% gap is the gap in dB
% cb = 1 for complex bb and cb=2 for real bb (if not given, assumes cb=2)
%
% OUTPUT
% en is the energy in the nth subchannel; for complex baseband divide by 2
% bn is the bit in the nth subchannel; for complex baseband multiply by 2
% Nstar is the number of energized dimensions
%
% dB into normal scale
function [bn, en , Nstar] = waterfill_gn(gn, E_bar, gap , cb)
gap = 10^(gap/10);
if nargin < 4
    cb=2;
end

[col, Ntot] = size(gn);

if col ~= 1
    error = 1
    return;
end

% initialization
en = zeros(1, Ntot);
bn = zeros(1, Ntot);

%%%%%%%%%%%%%%%%%%%%%%%%
% Now do waterfilling %
%%%%%%%%%%%%%%%%%%%%%%%%

%sort
[gn_sorted, Index] = sort(gn);   % sort gain, and get Index

gn_sorted = fliplr(gn_sorted);   % flip left/right to get the largest
```

```
                                % gain in leftside
Index = fliplr(Index);          % also flip index

num_zero_gn = length(find(gn_sorted == 0));
Nstar = Ntot - num_zero_gn;     % number of zero gain subchannels

                               % Number of used channels,
                               % start from Ntot - (number of zero gain subchannels)

while(1)
                                % The K calculation has been modified in order to
                                % accomodate the size of the number
    K = 1/Nstar * (Ntot * E_bar + gap * sum(1 ./ gn_sorted(1:Nstar)));
  En_min = K - gap/gn_sorted(Nstar);
                                % En_min occurs in the worst channel
  if (En_min<0)
     Nstar = Nstar - 1;   % If negative En, continue with less channels
  else
     break;               % If all En positive, done.
  end
end

En = K - gap./gn_sorted(1:Nstar); % Calculate En
Bn =(1/cb) * log2(K * gn_sorted(1:Nstar)/gap);  % Calculate bn

bn(Index(1:Nstar))=Bn; % return values in original index
en(Index(1:Nstar))=En; % return values in original index
```

G.4.2 DMTra.m program

```
% function [gn,en_bar,bn_bar,Nstar,b_bar,SNRdmt]=DMTra(h,NoisePSD,Ex_bar,N,gap)
%
% CAUTION - the user must know if the input channel h is complex baseband
% or real baseband in interpretting the outputs, as per comments below
%
% INPUTS
% h - the complex or real baseband sampled (temporal) pulse response
% NoisePSD and Ex_bar - noise power and energy/dimension or cpx-sample.
%    These two parameters need to be consistent in terms of # dimensions.
% N is the DFT size, N>2, and is "Nbar" for complex channels and N
%     for real channels
% The sampling rate (on each of Inphase or Quad for complex) is (N+v)/T
%    where 1/T is DMT symbol rate and nu = length(p) - 1
% gap is the gap in dB
%
% OUTPUTS
% gn is vector of channel gains or magnitude-squared |P|^2 / NoisePSD
% En - vector of energy allocation.
%      Per real dimension for real channels;
%      Per tone for complex channel.
%    For real channels, the upper image frequencies duplicate the lower
%      frequencies.
%    For complex channels, each En output is per that tone (= cmplx dim)
% bn_bar is the vector of bit/dim for all subchannels
```

```
%    For complex channels, double bn_bar for bits/tone.
%    For real channels, the upper image frequencies duplicate lower
%      frequencies, but bn_bar remains bits per real dimension.
% Nstar is the number of used input-size DFT (real or complex) dimensions
%     (2*Nstar is number of real dimensions for complex chan)
% b_bar is the number of bits per (real) dimension
%     (so 2*(N+v)*b_bar is total number of bits/symbol for a complex chan )
% SNRdmt is the equivalent DMT "geometric" SNR (complex or real) in dB
%
function [gn,en_bar,bn_bar,Nstar,b_bar,SNRdmt]=DMTra(h,NoisePSD,Ex_bar,N,gap)

Noise_var=NoisePSD;
gap=10^(gap/10);
nu=length(h)-1;

% initialization
en=zeros(1,N);
bn=zeros(1,N);
gn=zeros(1,N);
Hn = zeros(1,N);

% subchannel center frequencies
f=0:1/N:1-1/N;

% find Hn vector
for i=1:length(h)
Hn=Hn+h(i)*exp(j*2*pi*f*(i-1));
end

% find gn vector
gn=abs(Hn).^2/Noise_var;

%%%%%%%%%%%%%%%%%%%%%%%%
% Waterfilling for DMT %
%%%%%%%%%%%%%%%%%%%%%%%%

%sort
[gn_sorted, Index]=sort(gn);  % sort gain, and get Index

gn_sorted = fliplr(gn_sorted);% flip left/right to get the largest
                              % gain in leftside
Index = fliplr(Index);        % also flip index


num_zero_gn = length(find(gn_sorted == 0)); %number of zero-gain subchannels
Nstar=N - num_zero_gn;
  % Number of used channels,
  % start from N - (number of zero gain subchannels)

while(1)
  K=1/Nstar*(N*Ex_bar+gap*sum(1./gn_sorted(1:Nstar)));
  En_min=K-gap/gn_sorted(Nstar); % En_min occurs in the worst channel
  if (En_min<0)
     Nstar=Nstar-1;  % If negative En, continue with less channels
```

```
  else
     break;       % If all En positive, done.
  end
end

En=K-gap./gn_sorted(1:Nstar);  % Calculate En
Bn=log2(K*gn_sorted(1:Nstar)/gap);  % Calculate bn

bn(Index(1:Nstar))=Bn; % return values in original index
en(Index(1:Nstar))=En; % return values in original index

% Since channel is even, need to display
                                % only half of result
en_bar=en(1:N);
bn_bar=0.5*bn(1:N);

% calculate b_bar
b_bar=1/(N+nu)*(0.5*sum(bn));
SNRdmt=10*log10(gap*(2^(2*b_bar)-1));
```

G.4.3 DMTma.m program

```
% function [gn,En,bn,b_bar,SNRdmt]=DMTLCra(h,NoisePSD,Ex_bar,N,gap)
%
%
% Levin Campello's Method with DMT - REAL BASEBAND ONLY, allows PAM on
% Nyquist and DC and QAM on others.  Beta is forced to 1.
%
% Inputs
% h is the pulse response(an nu is set to length(p) - 1
% NoisePSD is the noise PSD on same scale as Ex_bar
% Ex_bar is the normalized energy
% N is the total number of real/complex subchannels, N>2
% gap is the gap in dB
%
% Outputs
% gn is the vector of channel gains (DC to Nyquist)
% En is the vector energy distribution from DC to Nyquist
% bn is the vector bit distribution from DC to Nyquist
% b_bar is the number of bits per dimension in the DMT symbol
%
% The first and last bins are PAM; the rest are QAM.

function [gn,En,bn,b_bar,SNRdmt]=DMTLCra(h,NoisePSD,Ex_bar,N,gap)
Noise_var=NoisePSD;
gap=10^(gap/10);
nu=length(h)-1;

% initialization
En=zeros(1,N/2+1);
bn=zeros(1,N/2+1);
gn=zeros(1,N/2+1);
Hn = zeros(1,N/2+1);
decision_table=zeros(1,N/2+1);
```

```
% subchannel center frequencies
f=0:1/N:1/2;

% find Hn vector
for i=1:length(h)
Hn=Hn+h(i)*exp(j*2*pi*f*(i-1));
end

% find gn vector
gn=abs(Hn).^2/Noise_var;

%debugging purpose
%plot(gn)

%%%%%%%%%%%%%%%%%%%%%%%%
% Levin Campello Loading %
%%%%%%%%%%%%%%%%%%%%%%%%

%initialization

%used energy so far
E_so_far=0;
%decision table - QAM and PAM
decision_table(2:N/2)=2*gap./gn(2:N/2);
% Gap formula incremental energies.
if gn(1) ~= 0
    decision_table(1)=3*gap/gn(1);
else
    decision_table(1)=inf;
end
if gn(N/2+1) ~=0
    decision_table(N/2+1)=3*gap/gn(N/2+1);
else
    decision_table(N/2+1)=inf;
end

%decision_table: debugging purpose

while(1)

    [y,index]=min(decision_table);
    E_so_far=E_so_far+y;

    if E_so_far > Ex_bar*N

        break;

    else

        En(index)=En(index)+y;
        bn(index)=bn(index)+1;

if (index ==1 | index == N/2+1)
```

```
  decision_table(index)=4*decision_table(index);
else
  decision_table(index)=2*decision_table(index);
end

    end

end

% calculate b_bar
b_bar=1/(N+nu)*(sum(bn));
SNRdmt=10*log10(gap*(2^(2*b_bar)-1));
```

G.4.4 Levin Campello Loading Programs

Levin Campello LC.m (Section 4.3)

```
% function [gn,En,bn,b_bar,SNRdmt]=DMTLCra(h,NoisePSD,Ex_bar,N,gap)
%
%
% Levin Campello's Method with DMT - REAL BASEBAND ONLY, allows PAM on
% Nyquist and DC and QAM on others.  Beta is forced to 1.
%
% Inputs
% h is the pulse response(an nu is set to length(p) - 1
% NoisePSD is the noise PSD on same scale as Ex_bar
% Ex_bar is the normalized energy
% N is the total number of real/complex subchannels, N>2
% gap is the gap in dB
%
% Outputs
% gn is the vector of channel gains (DC to Nyquist)
% En is the vector energy distribution from DC to Nyquist
% bn is the vector bit distribution from DC to Nyquist
% b_bar is the number of bits per dimension in the DMT symbol
%
% The first and last bins are PAM; the rest are QAM.

function [gn,En,bn,b_bar,SNRdmt]=DMTLCra(h,NoisePSD,Ex_bar,N,gap)
Noise_var=NoisePSD;
gap=10^(gap/10);
nu=length(h)-1;

% initialization
En=zeros(1,N/2+1);
bn=zeros(1,N/2+1);
gn=zeros(1,N/2+1);
Hn = zeros(1,N/2+1);
decision_table=zeros(1,N/2+1);

% subchannel center frequencies
f=0:1/N:1/2;

% find Hn vector
```

```
for i=1:length(h)
Hn=Hn+h(i)*exp(j*2*pi*f*(i-1));
end

% find gn vector
gn=abs(Hn).^2/Noise_var;

%debugging purpose
%plot(gn)

%%%%%%%%%%%%%%%%%%%%%%%%
% Levin Campello Loading %
%%%%%%%%%%%%%%%%%%%%%%%%

%initialization

%used energy so far
E_so_far=0;
%decision table - QAM and PAM
decision_table(2:N/2)=2*gap./gn(2:N/2);
% Gap formula incremental energies.
if gn(1) ~= 0
    decision_table(1)=3*gap/gn(1);
else
    decision_table(1)=inf;
end
if gn(N/2+1) ~=0
    decision_table(N/2+1)=3*gap/gn(N/2+1);
else
    decision_table(N/2+1)=inf;
end

%decision_table: debugging purpose

while(1)

    [y,index]=min(decision_table);
    E_so_far=E_so_far+y;

    if E_so_far > Ex_bar*N

        break;

    else

        En(index)=En(index)+y;
        bn(index)=bn(index)+1;

if (index ==1 | index == N/2+1)
  decision_table(index)=4*decision_table(index);
else
  decision_table(index)=2*decision_table(index);
end
```

```
    end

end

% calculate b_bar
b_bar=1/(N+nu)*(sum(bn));
SNRdmt=10*log10(gap*(2^(2*b_bar)-1));
```

Margin Adaptive Levin Campello MALC.m

```
% function [gn,En,bn,b_bar_check,margin]=MALC(h,SNRmfb,Ex_bar,b_bar,Ntot,gap)
%
% Margin Adaptive Levin Campello Loading
%
% h is the pulse response
% SNRmfb is the SNRmfb, so Ex_bar*norm(p)^2/sigma^2, in dB
% Ex_bar is the normalized energy
% Ntot is the total number of real subchannels, Ntot>2
%   Any guard periods must be addressed by program user
% gap is the gap in dB
% b_bar is the bit rate
%
% gn is channel gain
% En is the energy in the nth subchannel (PAM or QAM)
% bn is the bit in the nth subchannel (PAM or QAM)
% b_bar_check is the bit rate for checking - this should be equal to b_bar
% margin is the margin (in dB)
%
% The first bin and the last bin is PAM, the rest of them are QAM.

function [gn,En,bn,b_bar_check,margin]=MALC(h,SNRmfb,Ex_bar,b_bar,Ntot,gap)
Noise_var=Ex_bar*(norm(h)^2)/(10^(SNRmfb/10));
gap=10^(gap/10);

% initialization
En=zeros(1,Ntot/2+1);
bn=zeros(1,Ntot/2+1);
gn=zeros(1,Ntot/2+1);
Hn = zeros(1,Ntot/2+1);
decision_table=zeros(1,Ntot/2+1);

% subchannel center frequencies
f=0:1/Ntot:1/2;

% find Hn vector
for i=1:length(h)
Hn=Hn+h(i)*exp(j*2*pi*f*(i-1));
        % This value will be different depending if p represents
        % p(1) + p(2)*D^-1 + ....  or p(1) + p(2)*D^+1....,
        % but we'll get same gn, thus same waterfilling result.
        % (Note that both have the same magnitude response!)
end

% find gn vector
```

```
gn=abs(Hn).^2/Noise_var;

%debugging purpose
%plot(gn)

%%%%%%%%%%%%%%%%%%%%%%%
% Now do LC %
%%%%%%%%%%%%%%%%%%%%%%%

%initialization

%used energy so far
E_so_far=0;
%decision table - QAM and PAM
for n=2:Ntot/2
    if gn(n) ~= 0
        decision_table(n)=2*gap./gn(n);
    else
        decision_table(n)=inf;
    end
end
if gn(1) ~= 0
    decision_table(1)=3*gap/gn(1);
else
    decision_table(1)=inf;
end
if gn(Ntot/2+1) ~=0
    decision_table(Ntot/2+1)=3*gap/gn(Ntot/2+1);
else
    decision_table(Ntot/2+1)=inf;
end

%decision_table: debugging purpose

while(1)

    [y,index]=min(decision_table);
    E_so_far=E_so_far+y;

    if sum(bn) >= Ntot*b_bar

        break;

    else

        En(index)=En(index)+y;
        bn(index)=bn(index)+1;

if (index ==1 | index == Ntot/2+1)
  decision_table(index)=4*decision_table(index);
else
  decision_table(index)=2*decision_table(index);
end
```

```
    end

end

% calculate b_bar
b_bar_check=1/Ntot*(sum(bn));

% check margin
margin=10*log10(Ntot*Ex_bar/sum(En));
```

Levin Campello Rate Adaptive DMT Loading

```
% function [gn,En,bn,b_bar,SNRdmt]=DMTLCra(P,NoisePSD,Ex_bar,N,gap)
%
%
% Levin Campello's Method with DMT - REAL BASEBAND ONLY, allows PAM on
% Nyquist and DC and QAM on others.  Beta is forced to 1.
%
% Inputs
% p is the pulse response(an nu is set to length(p) - 1
% NoisePSD is the noise PSD on same scale as Ex_bar
% Ex_bar is the normalized energy
% N is the total number of real/complex subchannels, N>2
% gap is the gap in dB
%
% Outputs
% gn is the vector of channel gains (DC to Nyquist)
% En is the vector energy distribution from DC to Nyquist
% bn is the vector bit distribution from DC to Nyquist
% b_bar is the number of bits per dimension in the DMT symbol
%
% The first and last bins are PAM; the rest are QAM.

function [gn,En,bn,b_bar,SNRdmt]=DMTLCra(p,NoisePSD,Ex_bar,N,gap)
Noise_var=NoisePSD;
gap=10^(gap/10);
nu=length(p)-1;

% initialization
En=zeros(1,N/2+1);
bn=zeros(1,N/2+1);
gn=zeros(1,N/2+1);
Hn = zeros(1,N/2+1);
decision_table=zeros(1,N/2+1);

% subchannel center frequencies
f=0:1/N:1/2;

% find Hn vector
for i=1:length(p)
Hn=Hn+p(i)*exp(j*2*pi*f*(i-1));
end

% find gn vector
```

```
gn=abs(Hn).^2/Noise_var;

%debugging purpose
%plot(gn)

%%%%%%%%%%%%%%%%%%%%%%%
% Levin Campello Loading %
%%%%%%%%%%%%%%%%%%%%%%%

%initialization

%used energy so far
E_so_far=0;
%decision table - QAM and PAM
decision_table(2:N/2)=2*gap./gn(2:N/2);
% Gap formula incremental energies.
if gn(1) ~= 0
    decision_table(1)=3*gap/gn(1);
else
    decision_table(1)=inf;
end
if gn(N/2+1) ~=0
    decision_table(N/2+1)=3*gap/gn(N/2+1);
else
    decision_table(N/2+1)=inf;
end

%decision_table: debugging purpose

while(1)

    [y,index]=min(decision_table);
    E_so_far=E_so_far+y;

    if E_so_far > Ex_bar*N

        break;

    else

        En(index)=En(index)+y;
        bn(index)=bn(index)+1;

if (index ==1 | index == N/2+1)
  decision_table(index)=4*decision_table(index);
else
  decision_table(index)=2*decision_table(index);
end

    end

end

% calculate b_bar
```

```
b_bar=1/(N+nu)*(sum(bn));
SNRdmt=10*log10(gap*(2^(2*b_bar)-1));
```

G.4.5 TEQ Program

```
% [w, b, SNRmfb, SNRdmt_teq, SNRdmt, Nstar]=teq(h,L,nu,N,noise,Ex_bar,gap,cb)
%
% programmed by J.M. Cioffi
% Calculates a time domain equalizer and calculates the SNR for an infinite
% length DMT system to compare
%
% Inputs
% ------
% h: channel impulse response
% (FIR channels: p= 1+0.5D+2.3D^2->p=[1 0.5 2.3]
% IIR channels:  p=1/(2.4-0.5D)-> p=[2.4 -0.5])
% L: number of equalizer taps (should be < N/2)
% nu: target channel pulse response length < size(h)-1
% N:  FFT size for eventual computation of SNRdmt_teq
% noise: noise variance per sample
% Ex_bar: Input energy per sample
% cb = 2 for real, 1 for complex
%
% Outputs
% -------
% w: TEQ equalizer
% b: MMSE target channel shaping function (length=nu+1)
% SNRmfb: matched filter bound SNR
% SNRdmt_teq: DMT SNR for this equalized channel for N input (dB scale)
% SNRdmt: DMT SNR for the unequalized first nu+1 samples of the channel
%         Nbig=10000 (dB scale)
% Nopt:  Number of used dimensions
%
% Remarks
% -------
% calls "DMTra.m" and "waterfill_gn" functions
%
% ************************************************


function [w, b, SNRmfb, SNRdmt_teq, SNRdmt, Nopt]=teq(h,L,nu,N,noise,Ex_bar,gap,cb)
%   initialization
    norm_h=norm(h);
    [~ , size_h]=size(h);
    if nu > size_h - 2
        'error - nu < size of h - 1'
        return
    elseif L > N/2-1
        'error L > N/2-1'
    else
    C=[h(1) ; zeros(L-1,1)];
    R=[h,zeros(1,L-1)];
    H=toeplitz(C,R);
```

```
    btemp=zeros(L,nu+1);
    wtemp=zeros(L,L);
    mmsetemp=zeros(1,L);

%   find the MMSE's for all delays
    for delay = 1:L
    ryy=Ex_bar*H*H'+noise*eye(L,L);
    rxy=Ex_bar*[zeros(nu+1,delay) eye(nu+1) zeros(nu+1,L+size_h-2-nu-delay)]*H';
    rle=eye(nu+1)*Ex_bar-rxy*inv(ryy)*rxy';
    [v d]=eig(rle);
    [mmsetemp(delay) , index] = min(diag(d));
    btemp(delay,:)=norm_h*v(:,index)';
    wtemp(delay,:)=btemp(delay,:)*rxy*inv(ryy);
    mmsetemp(delay)=d(index,index)*norm_h^2;
    end

%   extract TEQ for this best delay
    [mmse_opt , index_opt] = min(mmsetemp);
    b=btemp(index_opt,:);
    w=wtemp(index_opt,:);
    eqh=conv(w,h);

% compute performance indications
    norm_c=norm(eqh(index_opt+1:index_opt+nu+1));
    error=[eqh(1:index_opt),zeros(1,nu+1),eqh(index_opt+nu+2:end)];
    Re=conv(error,conj(error(end:-1:1)));
    isi_energy=norm(error)^2;
    noise_energy=norm(w)^2*noise;
    Rw=noise*conv(w,conj(w(end:-1:1)));
    distortion=isi_energy*Ex_bar+noise_energy;
    Rb=conv(b,conj(b(end:-1:1)));

    SNRmfb=norm_h^2*Ex_bar/noise;
    SNRmfb=10*log10(SNRmfb);
    ReF=abs(fft(Re,N));
    RwF=abs(fft(Rw,N));
    RbF=abs(fft(Rb,N));
    gn=RbF./(ReF+RwF);
    Nbig=10000;
    fixfac=(Nbig/(Nbig + nu))*((N+nu)/N);
    [bn, en, Nopt] = waterfill_gn(gn,Ex_bar, gap, cb);
    bsumbar=sum(bn)/(N+nu);
    SNRdmt_teq=10*log10(2^(cb*bsumbar)-1);

    %SNRdmt for an infinite length DMT without the equalizer but taking the
    %first nu+1 samples of the channel
    [gn,en_bar,bn_bar,Nstar,b_bar,SNRdmt]=DMTra(h, noise, fixfac*Ex_bar,10000,gap);

    end
```

G.5 Chapter 5 Matlab software listings

G.5.1 From Subsection 5.1 - GDFE partitioning programs

fixmod.m

```
%
% function [Ct, Ot, Ruutt] = fixmod(H_NW, Ruu, C, tol)
%-------------------------------------------------------------------------
% fixmod removes the part of x that lies in H_NW's nullspace
%       x->xt, u->ut
% Inputs: H_NW, Ruu, C, tol
%     H_NW: Noise-whitened channel
%     Ruu: Autocorrelation matrix of u
%     C: discrete modulator matrix
%     tol: used in licols to determine rank of matrix Ctemp
%
% Outputs: Ct, Ot, Ruutt
%     Ct: new discrete modulator with components without H_NW's nullspace
%        components
%     Ot: Projection matrix of C2 onto C
%     Ruutt: new autocorrelation matrix for u
%
% This program calls licols
%-------------------------------------------------------------------------
function [Ct, Ot, Ruutt] = eliminate_x_nullspace(H_NW, Ruu, C, tol)

if nargin < 4
  tol = 1e-10;
end

% Step 1
[F, Lambda, M] = svd(H_NW);
% Step 2
Rho_H = rank(H_NW);
% Grabbing Rho_H vectors from M to form P_m
P_m = M(:, 1:Rho_H) * M(:, 1:Rho_H)';
% Step 3
Ctemp = P_m * C;
% Getting Ct from Ctemp
[Ct, Ct_indices] =licols(Ctemp, tol);

% Getting the indices in C2 (This can easily be O(n) instead of O(n^2)
C2_indices = 1:size(Ctemp, 2);
for Ct_index = Ct_indices
    C2_indices(C2_indices==Ct_index) = 0;
end

% Computing C2, Ot, and Ruutt
C2_indices = C2_indices(C2_indices~=0);
C2 = Ctemp(:, C2_indices);

% Step 4
Ot = inv(Ct' * Ct) * (Ct' * C2);
```

```
% Step 5
R11 = Ruu(Ct_indices, Ct_indices);
R12 = Ruu(Ct_indices, C2_indices);
R21 = Ruu(C2_indices, Ct_indices);
R22 = Ruu(C2_indices, C2_indices);

% Step 8
Ruutt = R11 + Ot * R21 + R12 * Ot' + Ot * R22 * Ot';
end
```

licols.m : The program lichols.m is a program from former student Ethan Liang that computes linearly independent columns of a matrix.

```
% function [Xsub,idx]=licols(X,tol,mode)
% Matt J (2021). Extract linearly independent subset of matrix columns
%(https://www.mathworks.com/matlabcentral/fileexchange/77437-extract-linearly-independent-subset-of-
%
% Significant modifications by Ethan Liang
%
%Extract a linearly independent set of columns of a given matrix X
%
%    [Xsub,idx]=licols(X)
%
%Inputs:
%
%  X: The given input matrix
%  tol: A rank estimation tolerance. Default=1e-10
%  mode: Rule for choice of columns
%          'desc': choose columns based on descending values of R
%          'LR': choose columns from left to right
%
%out:
%
% Xsub: The extracted columns of X
% idx:  The indices (into X) of the extracted columns
function [Xsub,idx]=licols(X,tol,mode)
     if ~nnz(X) %X has no non-zeros and hence no independent columns
         Xsub=[]; idx=[];
         return
     end

     if (nargin<3)
         mode = 'LR';
         if (nargin<2)
             tol=1e-10;
         end
     end



     if (strcmpi(mode, 'desc'))
         [Q, R, E] = qr(X,'vector');
         if ~isvector(R)
             diagr = abs(diag(R));
```

```
        else
            diagr = abs(R(1));
        end

          %Rank estimation
          r = find(diagr >= tol*diagr(1), 1, 'last'); %rank estimation
          idx=sort(E(1:r));
          Xsub=X(:,idx);
    elseif (strcmpi(mode, 'LR'))

        % Although the above code works, this selection prioritizes
        % selecting columns from left to right, which is more commonly
        % used in the examples in the textbook.
        [Q, R] = qr(X);
        if ~isvector(R)
            diagr = abs(diag(R));
        else
            diagr = abs(R(1));
        end

        idz = 1;
        idx = [];
        for idy = 1:size(X, 2)
            if (abs(R(idy, idz)) >= tol*diagr(1))
                idx = [idx, idy];
                idz = idz + 1;
            end
        end
        Xsub=X(:,idx);

    else
        error("Incorrect mode selected: " + string(mode));
    end
end
```

fixin.m

```
% function [At, OA, Ruupp] = fixin(Ruutt, Ct, tol)
%
%-------------------------------------------------------------------------
% xt->xp, ut->up (Reducing ranking of ut->up, possible b/c Rxxtt singlar)
% Inputs: Ruutt, Ct, tol
% Outputs: At, OA, Ruupp
%
% Ruutt: autocorrelation matrix of ut
% Ct: "nullspace of H"-adjusted discrete modulator matrix
% tol: used in licols to determine rank of matrix Ctemp
% At: "nullspace of H & Rxx singularity"-adjusted discrete modulator matrix
% OA: projection matrix of A2 onto A
% Ruupp: autocorrelation matrix of up
%-------------------------------------------------------------------------

function [At, OA, Ruupp] = fixin(Ruutt, Ct, tol)
if nargin < 3
```

```
  tol = 1e-10;
end

% Step 1
Rxxtt = Ct * Ruutt * Ct';
% This is necessary to get V orthonormal. eig in MATLAB only produces
% orthonormal V when Rxxtt is known to be Hermitian. Without following
% command, MATLAB doesn't recognize Rxxtt as Hermitian.
Rxxtt = 0.5 * (Rxxtt + Rxxtt');
[V, D] = eig(Rxxtt);

% Step 2
eig_indices = abs(diag(D)) > tol;
Pq = V(:, eig_indices) * V(:, eig_indices)';

% Step 3
Atemp = Pq * Ct;
[At, A_indices] =licols(Atemp, tol);


% Getting the indices in A2 (This can easily be O(n) instead of O(n^2)
A2_indices = 1:size(Atemp, 2);
for A_index = A_indices
    A2_indices(A2_indices==A_index) = 0;
end

% Computing A2, OA, and Ruutt
A2_indices = A2_indices(A2_indices~=0);
A2 = Atemp(:, A2_indices);

% Step 4
OA = inv(At' * At) * (At' * A2);

% Step 5
R11tt = Ruutt(A_indices, A_indices);
R12tt = Ruutt(A_indices, A2_indices);
R21tt = Ruutt(A2_indices, A_indices);
R22tt = Ruutt(A2_indices, A2_indices);

% Step 8
Ruupp = R11tt + OA * R21tt + R12tt * OA' + OA * R22tt * OA';

end
```

computeGDFE Program: This program was originally supplied by Ethan Liang, and then significantly modified by the instructor afterward.

```
% function [snrGDFEu, GU, WU, S0, MSWMFU, b, bbar, snrGLEu] = computeGDFE(H, A, cb, Lx)
%-------------------------------------------------------------------------
% Inputs
% H: noise-whitened channel, Ly x Lx (one tone)
% A: any Lx x Lx square root of input autocorrelation matrix
%       This can be generalized non-square square-root
% cb: =1 if H is complex baseband; =2 if H is real baseband
```

```
% Lx: optional input of Lx when not equal to size of A
%     this is used to compute bits/dimension properly
%
% Outputs
% GU: unbiased feedback matrix
% WU: unbiased feedforward linear equalizer
% S0: sub-channel channel gains
% MSWMFU: unbiased mean-squared whitened matched filter
% b: bit distribution vector
% bbar: number of bits/dimension (real if cb=2, complex if cb=1)
% snrGDFEu - unbiased SNR in dB; assumes size of R_sqrt input
%    the user should recompute SNR if there is a cyclic prefix
% b = bit distribution over symbol dimensions (real cb=2; complex cb=1)
% bbar is sum(b)/Lx so total bits/(real cb=2; cplx cb=1) dimension
% snrGLEu is the linear GLE SNR
%
% Note: The R_sqrt need not be square non-singular, as long as
% R_sqrt*R_sqrt' = input autocorrelation matrix Rxx, but SNRLEu is off
%Thanks to Ethan Liang, corrected/updated J. Cioffi
%
%------------------------------------------------------------------------
function [snrGDFEu, GU, WU, S0, MSWMFU, b , bbar, snrGLEu] = computeGDFE(H, A, cb, Lx)
%------------------------------------------------------------------------
% Computing Ht: Ht = H*A
Ht = H*A;
%------------------------------------------------------------------------

if (nargin<4)
  [~ , Lx , ~] = size(A);
end
[Ly,~,~]=size(H);

% Computing Rf, Rbinv, Gbar
Rf = Ht' * Ht;
Rbinv = Rf + eye(size(Rf));
Gbar = chol(Rbinv);
Ng=size(Gbar);
Ng=Ng(1);

% Computing the matrices of interest
G = inv(diag(diag(Gbar)))*Gbar;
S0 = diag(diag(Gbar))*diag(diag(Gbar));
% W = inv(S0)*inv(G');

GU = eye(size(G)) + S0*pinv(S0-eye(size(G)))*(G-eye(size(G)));
WU = pinv(S0-eye(size(G)))*inv(G');
MSWMFU = WU*Ht';
% Section to handle numerical is with large-dim determinants/geo-ave
if Ng < 10
   snrGDFEu = 10*log10(det(S0)^(1/(Lx))-1);
else
   snrGDFE=0;
   for n=1:Ng
    snrGDFE = snrGDFE + (10/(Lx))*log10(S0(n,n));
```

```
   end
   snrGDFEu=10*log10(10^(0.1*snrGDFE)-1);
end
b=(1/cb)*log2(diag(S0));
bbar=(1/Lx)*sum(b);
% This is linear equalizer computation
sGLE0=inv(diag(diag(inv(Rbinv))));
snrGLEu=10*log10(det(sGLE0)^(1/(Lx))-1);
```

G.5.2 From Subsection 5.4 - MU MAC programs

SWF.m program

```
% function [Rxx, bsum , bsum_lin] = SWF(Eu, H, Lxu, Rnn, cb)
%
% Simultaneous water-filling MAC max rate sum (linear and nonlinear GDFE)
% The input is space-time domain h, and the user can specify a temporal
% block symbol size N (essentially an FFT size).
%
% Inputs:
% Eu  U x 1 energy/SAMPLE vector. Single scalar equal energy all users
%     any (N/N+nu) scaling should occur BEFORE input to this program.
% H   The FREQUENCY-DOMAIN Ly x sum(Lx(u)) x N MIMO channel for all users.
%     N is determined from size(H) where N = # used tones
% Lxu 1xU vector of each user's number of antennas
% Rnn The Ly x Ly x N noise-autocorrelation tensor (last index is per tone)
% cb  cb = 1 for complex, cb=2 for real baseband
%       cb=2 corresponds to a frequency range at an sampling rate 1/T' of
%       [0, 1/2T'] while with cb=1, it is [0, 1/T'].  The Rnn entered for
%       these two situations may differ, depending on how H is computed.
%
% Outputs:
% Rxx A block-diagonal psd matrix with the input autocorrelation for each
%     user on each tone. Rxx has size (sum(Lx(u)) x sum(Lx(u)) x N .
%     sum trace(Rxx) over tones and spatial dimensions equal the Eu
% bsum the maximum rate sum.
% bsum  bsum_lin - the maximum sum rate with a linear receiver
%     b is an internal convergence sum rate value, not output
%
% This program significantly modifies one originally supplied by student
% Chris Baca

function [Rxx, bsum, bsum_lin, Bun , Eun] = SWF(Eu, H, Lxu , Rnn, cb)
U = numel(Lxu);
[Ly, Lx_sum, N] = size(H);

if numel(Eu) == 1
    Eu = Eu*ones(U);
end

i = 0;
for n=1:N
Rxx(:,:,n) = diag(zeros(Lx_sum, N));
H(:,:,n)=inv(sqrtm(Rnn(:,:,n)))*H(:,:,n);
end
```

```
b = zeros(U, 1);
bsum=sum(b);

cum_index=1;
user_ind(1)=1;
for u=2:U
    cum_index = cum_index + Lxu(u);
    user_ind(u) = cum_index;
end

for u = 1:U

    st = user_ind(u);

    if u < U
        en = user_ind(u+1)-1;
    else
        en = Lx_sum;
    end

    for n = 1:N
        Rxx(st:en, st:en,n) = Eu(u)*eye(en-st+1);
    end
end

%b = zeros(U);
while 1
    bsum_prev = bsum;
    b_prev = b;

    for u = 1:U
        %new user
        st = user_ind(u);

            if u < U
                en = user_ind(u+1)-1;
            else
                en = Lx_sum;
            end
        M_u = zeros(en-st+1,en-st+1, N);
        g_u = [];
       Rnn_un=zeros(Ly,Ly,N);
        for n = 1:N
            %new tone
            Rnn_un(:,:,n) = eye(Ly);
            for v = 1:U
                if v ~= u
                    st = user_ind(v);
                    if v < U
                        en = user_ind(v+1)-1;
                    else
                        en = Lx_sum;
                    end
```

```
                Hvn = H(:, st:en, n);
                Rxxvn = Rxx(st:en, st:en, n);
                Rnn_un(:,:,n) = Rnn_un(:,:,n) + Hvn*Rxxvn*Hvn';
            end
        end

        st = user_ind(u);

        if u < U
            en = user_ind(u+1)-1;
        else
            en = Lx_sum;
        end


        Hun = H(:, st:en, n);
        Hun_til = sqrtm(inv(Rnn_un(:,:,n)))*Hun;

        [F, D, M_n] = svd(Hun_til);
        s = svd(Hun_til);

        M_u(:,:,n) = M_n;
        g_u(:,n) = s.^2;

    end

    g_flat = reshape(g_u, [1, numel(g_u)]);
%        [g_sort, ind] = sort(g_flat, 'descend');

    [B(u,:),E(u,:),L_star] = waterfill_gn(g_flat,Eu(u),0, cb);

    L = numel(g_flat);
%       Etot = N*Eu(u);
%       j = L;

%       e = zeros(1,L);
%       size(E)
%       e(1:L_star)=E;

    e = reshape(E(u,:), [en-st+1, N]);
    b=sum(B(u,:));

    for n = 1:N
        Rxx(st:en, st:en, n) = M_u(:,:,n)*diag(e(:,n))*M_u(:,:,n)';
    end
end
    bsum=0;
    for n=1:N
    bsum=bsum+(1/cb)*log2(det(eye(Ly)+H(:,:,n)*Rxx(:,:,n)*H(:,:,n)'));
    end
    bsum=real(bsum);

i = i+1;
```

```
    if abs(bsum-bsum_prev) <= 1e-6
  % if norm(b-b_prev) <= 0
        break
    end
    if i>1000
        i
        break
    end
end
%b = b(:,1);
%Hcell = mat2cell(H,Ly,Lx_sum,ones(1,N));
%Hexpand = [blkdiag(Hcell{1,1,:})]
%Rcell = mat2cell(Rxx,Ly,Lx_sum,ones(1,N));
%Rcell = mat2cell(Rxx,Lx_sum,Lx_sum,ones(1,N));
%Rxxexpand = [blkdiag(Rcell{1,1,:})]
%bsum = log2(det(eye(Lx_sum*N)+Hexpand*Rxxexpand*Hexpand'));
%bsum = log2(det(eye(Ly*N)+Hexpand*Rxxexpand*Hexpand'));
%bsum=0;
%for n=1:N
%    bsum=bsum+(1/cb)*log2(det(eye(Ly)+H(:,:,n)*Rxx(:,:,n)*H(:,:,n)'));
%end
%bsum=real(bsum);
bs=zeros(1,U);
bsum_lin=0;
for u=1:U
    indices=user_ind(u):user_ind(u)+Lxu(u)-1;
    for n=1:N
     bs(u)=bs(u)+(1/cb)*(log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)')) - log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)'- H(:,indices,n)*Rxx(indices, ...
       indices,n)*H(:,indices,n)')));
    end
    bsum_lin=bsum_lin+real(bs(u));
end
```

maxRMAC_cvx.m

```
% function [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta, cb)
%
% maxRMAC_cvx Maximizes weighted rate sum subject to energy constraint, each
% user has ONLY ONE transmit antenna. It uses CVX and mosek. It only works
% for Lxu=1 on all users.
%
%   INPUTS:
%     H(:,u,n): Ly x U x N channel matrix. Ly = number of receiver antennas,
%     U = number of users and N = number of tones.
%
%     If the channel is real-bbd (cb=2), maxPMAC_cvx realizes user data rates
%     over all the tones (or equivalently positive and negative
%     freqs). This means the input H must be conjugate symmetric H_n =
%     H_{N-n}^*.  The program will reduce N by 2 and focus energy on the
%     lower half of frequencies. Thus N>1 must be even for real channels,
%     with a special exception made for N=1.
%
```

```
%     By constast if cb=1 (cplx bbd), maxRMAC_cvx need not have a conjugate
%     symmetric H and N is not reduced.
%
% Eu: 1 x U Energy constraint for each user as total energy per symbol.
%     For N=1 channel, the program adjusts energy to be per complex (cb=1)
%     symbol at the beginning and then restores at end.  cb=2 has no
%     change.  So for N=1, the input Eu should be u's energy/symbol.
%
% theta: weight on each user's rate, length-U vector.
%
% cb: =1 if H is complex baseband, and cb=2 if H is real baseband.
%
%
%   OUTPUTS:
% Eun:    U x N energy distribution. Eun(u,n) is user u's energy allocation
%         on tone n.
% w:      U x 1 Lagrangian multiplier w.r.t. energy constraints
% bun:    U by N bit distributions for all users.
% ************************************************************************
function [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta,cb)
[Ly, U, N] = size(H);
if N>1
    if cb==2
    H=H(:,:,1:N/2);
    N=N/2;
    end
else
    Eu=Eu/(3-cb);
end
Eu = reshape(Eu,[],1);
theta = reshape(theta,[],1);
[stheta, idx] = sort(theta, 'descend');
delta = -diff([stheta;0]);
sH = H(:,idx, :);
Hs = zeros(Ly, Ly, U, N);
for n=1:N
    for u=1:U
        Hs(:,:,u,n) = sH(:,u,n)*sH(:,u,n)';
    end
end
cvx_begin quiet
cvx_solver mosek
    variable Eun(U,N) nonnegative
    dual variable w
    expression r(U,N)
    S = cumsum(Hs.*repelem(reshape(Eun,1,1,U,N),Ly,Ly,1,1), 3);
    for u = 1:U
        for n = 1:N
            r(u,n) = log_det(S(:,:,u,n) + eye(Ly));
        end
    end
    maximize sum(delta'*r)
    subject to
        w: sum(Eun,2)<=Eu;
```

```
cvx_end

S = cumsum(Hs.*repelem(reshape(Eun,1,1,U,N),Ly,Ly,1,1), 3) + eye(Ly);
cumrate = zeros(U,N);
for u = 1:U
    for n = 1:N
        cumrate(u,n) = (1/cb)*real(log2(det(S(:,:,u,n))));
    end
end
bun(idx,:) = diff([zeros(1,N); cumrate]);
w(idx)=w;
if N == 1
    Eun=(3-cb)*Eun;
end
end
```

maxRMACMIMO.m

```
% function [Rxxs, Eun, w, bun] = maxRMACMIMO(H, Lxu, Eu, theta , cb)
%
% maxRMAC_vector_cvx Maximize weighted rate sum subject to energy
% constaint. This uses cvx and mosek.
%
%   INPUTS:
%     H(:,u,n): Ly x U x N channel matrix. Ly = number of receiver antennas,
%     U = number of users and N = number of tones.
%
%     If the channel is real-bbd (cb=2), maxPMAC_cvx realizes user data rates
%     over all the tones (or equivalently positive and negative
%     freqs). This means the input H must be conjugate symmetric H_n =
%     H_{N-n}^*.  The program will reduce N by 2 and focus energy on the
%     lower half of frequencies. Thus N>1 must be even for real channels,
%     with a special exception made for N=1.
%
%     By constast if cb=1 (cplx bbd), maxRMAC_cvx need not have a conjugate
%     symmetric H and N is not reduced.
%
% Eu: 1 x U Energy constraint for each user as total energy per symbol.
%     For N=1 channel, the program adjusts energy to be per complex (cb=1)
%     symbol at the beginning and then restores at end.  cb=2 has no
%     change.  So for N=1, the input Eu should be u's energy/symbol.
%
% theta: weight on each user's rate, length-U vector.
%
% cb: =1 if H is complex baseband, and cb=2 if H is real baseband.
%
%   OUTPUTS:
% Rxxs:  U-by-N cell array containing Rxx(u,n)'s if Lxu is a length-U
%        vector; or Lxu-by-Lxu-by-U-by-N tensor if Lxu is a scalar.
% Eun:    U x N energy distribution. Eun(u,n) is user u's energy allocation
%         on tone n.
% w:      U x 1 Lagrangian multiplier w.r.t. energy constraints
% bun:    U by N bit distributions for all users.
% bun: U by N bit distributions for all users.
```

```
% ************************************************************************
function [Rxxs, Eun, w, bun] = maxRMACMIMO(H, Lxu, Eu, theta, cb)

UNIFORM_FLAG = 0;
[Ly, ~, N] = size(H);
if N>1
    if cb==2
    H=H(:,:,1:N/2);
    N=N/2;
    end
else
    Eu=Eu/(3-cb);
    H = reshape(H,Ly,[],1);
end

Eu = reshape(Eu,[],1);
theta = reshape(theta,[],1);
[stheta, idx] = sort(theta, 'descend');
delta = -diff([stheta;0]);
U = length(Eu);
if length(Lxu) == 1
    Lxu = ones(1,U)*Lxu;
    UNIFORM_FLAG = 1;
end
Lxu_max = max(Lxu);
index_end = cumsum(Lxu);
index_start = [1,index_end(1:end-1)+1];
Lxu = Lxu(idx);
index_start = index_start(idx);
index_end = index_end(idx);
cvx_begin quiet
cvx_solver mosek
    variable rxxs(Lxu_max, Lxu_max, U, N) hermitian
    dual variable w
    expressions r(U,N) S(Ly, Ly) Eun(U,N)
    for n = 1:N
        S=eye(Ly);
        for u = 1:U
            S = S + H(:,index_start(u):index_end(u),n)...
                *rxxs(1:Lxu(u),1:Lxu(u),u,n)...
                *H(:,index_start(u):index_end(u),n)';
            r(u,n) = log_det(S);
            Eun(u,n) = trace(rxxs(1:Lxu(u),1:Lxu(u),u,n));
        end
    end
    maximize sum(delta'*r)
    subject to
        w: sum(Eun,2)<=Eu;
        for u = 1:U
            for n=1:N
                rxxs(1:Lxu(u),1:Lxu(u),u,n)==hermitian_semidefinite(Lxu(u));
            end
        end
cvx_end
```

```
cumrate = zeros(U,N);
for n = 1:N
    S=eye(Ly);
    for u = 1:U
        S = S + H(:,index_start(u):index_end(u),n)...
            *rxxs(1:Lxu(u),1:Lxu(u),u,n)...
            *H(:,index_start(u):index_end(u),n)';
        cumrate(u,n) = (1/cb)*real(log2(det(S)));
    end
end
bun(idx,:) = diff([zeros(1,N); cumrate]);
w(idx)=w;
if N == 1
    Eun=(3-cb)*Eun;
    rxxs=(3-cb)*rxxs;
end

if UNIFORM_FLAG
    Rxxs(:,:,idx,1:N) = rxxs;
    for u = 1:U
        for n = 1:N
            Eun(idx(u),n) = trace(Rxxs(:,:,u,n));
        end
    end
else
    Rxxs = cell(U,N);
    for u = 1:U
        for n = 1:N
            Rxxs{idx(u),n} = rxxs(1:Lxu(u),1:Lxu(u),u,n);
            Eun(idx(u),n) = trace(Rxxs{idx(u),n});
        end
    end
end

end
```

maxRMAC.m This program works with $L_{x,u} = 1$ and no CVX use.

```
 % function [Eun, w, bun] = maxRMAC(H, Eu, theta, cb)
%
% maxRMAC Maximizes weighted rate sum subject to energy constraint, each
% user has ONLY ONE transmit antenna (Lxu=1). It does not use CVX.
%
%   INPUTS:
%     H(:,u,n): Ly x U x N channel matrix. Ly = number of receiver antennas,
%     U = number of users and N = number of tones.
%
%     If the channel is real-bbd (cb=2), maxPMAC_cvx realizes user data rates
%     over all the tones (or equivalently positive and negative
%     freqs). This means the input H must be conjugate symmetric H_n =
%     H_{N-n}^*.  The program will reduce N by 2 and focus energy on the
%     lower half of frequencies. Thus N>1 must be even for real channels,
%     with a special exception made for N=1.
```

```
%
%     By constast if cb=1 (cplx bbd), maxRMAC_cvx need not have a conjugate
%     symmetric H and N is not reduced.
%
% Eu: 1 x U Energy constraint for each user as total energy per symbol.
%     For N=1 channel, the program adjusts energy to be per complex (cb=1)
%     symbol at the beginning and then restores at end.  cb=2 has no
%     change.  So for N=1, the input Eu should be u's energy/symbol.
%
% theta: weight on each user's rate, length-U vector.
%
% cb: =1 if H is complex baseband, and cb=2 if H is real baseband.
%
%
%   OUTPUTS:
% Eun:    U x N energy distribution. Eun(u,n) is user u's energy allocation
%         on tone n.
% w:      U x 1 Lagrangian multiplier w.r.t. energy constraints
% bun:    U by N bit distributions for all users.
%
% Subroutines called
%     startEllipseRate
%     Lag_dual_f_rate
%     minPtonerate
% *--------------------------------------------------------------------------------------
function [Eun, w, bun] = maxRMAC(H, Eu, theta, cb)

err = 1e-9;                             % error tolerance
count = 0;
[Ly, U, N] = size(H);
if N>1
    if cb==2
    H=H(:,:,1:N/2);
    N=N/2;
    end
else
    Eu=Eu/(3-cb);
end
Eu = reshape(Eu,[],1);
theta = reshape(theta,[],1);

[A, g] = startEllipseRate(H, Eu, theta); % starting ellipsoid
w = g;

while 1
                                        % Ellipsoid method starts here
    [f, bun, Eun] = Lag_dual_f_rate(H, theta, w, Eu , cb);
    g = Eu - sum(Eun,2);                  % sub-gradient

    if sqrt(g' * A * g) <= err          % stopping criteria
        break
    end
                                        % Updating the ellipsoid
    gt = g / sqrt(g' * A * g);
```

```
    w = w - 1 / (U + 1) * A * gt;
    A = U^2 / (U^2 - 1) * (A - 2 / (U + 1) * A * gt * gt' * A);

    ind = find(w < zeros(U,1));

    while ~isempty(ind)                 % This part is to make sure that w is feasible,
        g = zeros(U,1);                 % it was not covered in the lecture notes and you may skip
        g(ind(1)) = -1;
        gt = g / sqrt(g' * A * g);
        w = w - 1 / (U + 1) * A * gt;
        A = U^2 / (U^2 - 1) * (A - 2 / (U + 1) * A * gt * gt' * A);
        ind = find(w < zeros(U,1));
    end
    count = count+1;
end

bun = bun /log(2);                          % conversion from nuts to bits
if N == 1
    Eun=(3-cb)*Eun;
end
```

The subroutines called are

startEllipseRate.m

```
% function [A, g] = startEllipseRate(H, Eu, theta)
% This function initializes the ellipsoid method for rate maximization problem.
%
% function [A, g] = startEllipseRate(G, Eu, theta)
% H containes the channel matrices as is defined in maxRMAC.m and Eu is
% the U by 1 power contraint vector, theta is the U by 1 rate weights.
% For all rate maximization WF steps, decoding order is 1,2,...U.
%
% A is the matrix describing the staring ellipsoid and g is its center
% ************************************************************************
function [A, g] = startEllipseRate(H, Eu, theta)

[Ly, U, N] = size(H);

order = [1:U];                      % decoding order, arbitrary
tmax = zeros(U,1);

for u = 1:U
    bn = zeros(U,N);
    en = zeros(U,N);
    E = Eu;
    E(u) = E(u) * 0.9;
    for u1 = flipud(order)          % starting from last user.

        for index = 1:N
            Rnoise(:,:,index) = eye(Ly);
        end
        for u2 = flipud(order)      % Note that the users that have been decoded has zero E
            if u2 == u1,
            else
```

```
                for index = 1:N
                    Rnoise(:,:,index) = Rnoise(:,:,index) + en(u2,index) * ...
                                        H(:,u2,index) * H(:,u2,index)';
                end
            end
        end
        for index = 1:N
            h_temp = Rnoise(:,:,index)^(-1/2) * H(:,u1,index);
            g(u1,index) = svd(h_temp)^2;
        end

        [b_temp, E_temp] = waterfill_gn(g(u1,:), Eu(u1) / N, 0);

        en(u1,:) = E_temp;
        bn(u1,:) = b_temp;
    end
    tmax(u) = sum(theta .* sum(bn,2));    % \sum_n \sum_u \theta_u bn_u
end
A = eye(U) * sum(tmax.^2) / 4;        % an sphere centerced at tmax/2
g = tmax / 2;
```

Lag_dual_f_rate.m

```
% function [f, b, E] = Lag_dual_f_rate(G, theta, w, Eu , cb);
% this function computes the Lagrange dual function by solving the
% optimization problem (calling the function minPtone) on each tone.
% the inputs are:
% 1)  H, an Ly by U by N channel matrix. Ly is the number of receiver antennas,
%     U is the total number of users and N is the total number of tones.
%     H(:,:,n) is the channel matrix for all users on tone n
%     and H(:,u,n) is the channel for user u on tone n. In this code we assume each user
%     only has single transmit antenna, thus H(:,u,n) is a column vector.
% 2)  theta, a U by 1 vector containing the weights for the rates.
% 3)  w, a U by 1 vector containing the weights for each user's power.
% 4)  Eu, a U by 1 vector containing the power constraints for all the users
% 5)  cb = 1 for complex bb and cb=2 for real bb
%
% the outputs are:
% 1)  f, the Lagrange dual function value.
% 2)  b, a U by N vector containing the rates for all users and over all tones
%     that optimizes the Lagrangian. b(u,:) is the rate allocation for user
%     u over all tones.
% 3)  E, a U by N vector containing the powers for all users and over all tones
%     that optimizes the Lagrangian. E(u,:) is the power allocation for
%     user u over all tones.
% ************************************************************
function [f, b, E] = Lag_dual_f_rate(H, theta, w, Eu, cb);

[Ly, U, N] = size(H);
f = 0;
b = zeros(U,N);
E = zeros(U,N);

% Performing optimization over all N tones,
```

```
for i = 1:N
    [temp, b(:,i), E(:,i)] = minPtonerate(H(:,:,i), theta, w , cb);
    f = f + temp;
end
f = f + w' * Eu;
```

and

minPtonerate.m

```
% function [f, b, e] = minPtonerate(H, theta, w, cb)
% minPtone maximizes f = sum_{u=1}^U lambda_u * b_u - sum_{u=1}^U w_u * e_u
% subject to b \in C_g(H,e)
%
% the inputs are:
% 1)  H, an Ly by U channel matrix. Ly is the number of receiver antennas,
%     U is the total number of users.
%     H(:,u) is the channel for user u. In this code we assume each user
%     only has single transmit antenna, thus H(:,u) is a column vector.
% 2)  theta, a U by 1 vector containing the weights for the rates.
% 3)  w, a U by 1 vector containing the weights for each user's energy.
% 4)  cb=1 for complex bb and cb=2 for real bb
%
% the outputs are:
% 1)  f, the minimum value (or maximum value of the -1 * function).
% 2)  b, a U by 1 vector containing the rates for all users
%     that optimizes the given function.
% 3)  e, a U by 1 vector containing the energies for all users
%     that optimizes the given function.
function [f, b, e] = minPtonerate(H, theta, w, cb)

[Ly, U] = size(H);
[stheta, ind] = sort(-theta);
stheta = -stheta;
sH = H(:,ind);
sw = w(ind);

NT_max_it = 1000;                                % Maximum number of Newton's
                                                 % method iterations

dual_gap = 1e-6;
mu = 10;                                         % step size for t
alpha = 0.01;                                    % back tracking line search parameters
beta = 0.5;

count = 1;
nerr = 1e-5;                                     % acceptable error for inner loop
                                                 % Newton's method

e = ones(U,1);                                   % Strictly feasible point;

t = .1;
l_p = 1;                                         % for newton's method termination
```

```
while (1+U)/t > dual_gap
   t =  t * mu;
   l_p = 1;
   count = 1;
   while l_p > nerr & count < NT_max_it

      f_val = eval_f(t * stheta, sH, e, t * sw);
                                              % calculating function value

                                              % calculating the hessian and gradient
      [g, h] = Hessian(t * stheta, sH, e, t * sw);

      de = -real(h\g);                                % search direction

      l_p = g' * de;                          % theta(e)^2 for Newton's method

      s = 1;                                  % checking e = e+s*de feasible
                                              % and also back tracking algorithm

      e_new = e + s * de;



      if e_new > zeros(U,1)
          f_new = eval_f(t * stheta, sH, e_new, t * sw);
          feas_check = (f_new > f_val + alpha * s * g' * de);
      else
          feas_check = 0;
      end






      while ~feas_check
         s = s * beta;
         if s < 1e-40
             l_p = nerr/2;
             break
         end
         e_new = e + s * de;

         if e_new > zeros(U,1)
             f_new = eval_f(t * stheta, sH, e_new, t * sw);
             feas_check = (f_new > f_val + alpha * s * g' * de);
         else
             feas_check = 0;
         end

      end

      e = e + s * de;                         % update e
      count = count + 1;                      % number of Newtons method iterations
   end
```

```
end


M = eye(Ly) + sH(:,1) * sH(:,1)' * e(1);
b = zeros(U,1);
b(1) =  (1/cb) * real(log(det(M)));
for u = 2:U
    b(u) = -(1/cb) * real(log(det(M)));
    M = M + sH(:,u) * sH(:,u)' * e(u);
    b(u) = b(u) + (1/cb) * real(log(det(M)));
end


b(ind) = b;
e(ind) = e;
f = theta' * b - w' * e;
```

maxRESMAC_cvx.m

```
% function [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta, cb)
%
% maxRMAC_cvx Maximizes weighted rate sum subject to energy constraint, each
% user has ONLY ONE transmit antenna. It uses CVX and mosek. It only works
% for Lxu=1 on all users.
%
%   INPUTS:
% H(:,u,n): Ly x U x N channel matrix. Ly = number of receiver antennas,
%     U = number of users and N = number of tones.
%
%     If the channel is real-bbd (cb=2), maxPMAC_cvx realizes user data rates
%     over all the tones (or equivalently positive and negative
%     freqs). This means the input H must be conjugate symmetric H_n =
%     H_{N-n}^*.  The program will reduce N by 2 and focus energy on the
%     lower half of frequencies. Thus N>1 must be even for real channels.
%     For real N=1 channel, the program runs and will produce half the data
%     rates for the same complex N=1 channel.
%
%     By constast if cb=1 (cplx bbd), maxRMAC_cvx need not have a conjugate
%     symmetric H and N is not reduced.
%
% Eu: TOTAL SUM-ENERGY/symbol constraint for all user inputs.
%
% theta: weight on each user's rate, length-U vector.
%
% cb: =1 if H is complex baseband, and cb=2 if H is real baseband.
%
%   OUTPUTS:
% Eun:    U x N energy distribution. Eun(u,n) is user u's energy allocation
%         on tone n.
% w:   U x 1 Lagrangian multiplier w.r.t. energy constraints
% bun: U by N bit distributions for all users.
% ************************************************************************
function [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta,cb)
```

```
[Ly, U, N] = size(H);
if N>1
    if cb==2
    H=H(:,:,1:N/2);
    N=N/2;
    end
end
Eu = reshape(Eu,[],1);
theta = reshape(theta,[],1);
[stheta, idx] = sort(theta, 'descend');
delta = -diff([stheta;0]);
sH = H(:,idx, :);
Gs = zeros(Ly, Ly, U, N);
parfor n=1:N
    for u=1:U
        Gs(:,:,u,n) = sH(:,u,n)*sH(:,u,n)';
    end
end
cvx_begin quiet
cvx_solver mosek
    variable Eun(U,N) nonnegative
    dual variable w
    expression r(U,N)
    S = cumsum(Gs.*repelem(reshape(Eun,1,1,U,N),Ly,Ly,1,1), 3);
    for u = 1:U
        for n = 1:N
            r(u,n) = log_det(S(:,:,u,n) + eye(Ly));
        end
    end
    maximize sum(delta'*r)
    subject to
        w: sum(sum(Eun,2))<=Eu;
cvx_end

S = cumsum(Gs.*repelem(reshape(Eun,1,1,U,N),Ly,Ly,1,1), 3) + eye(Ly);
cumrate = zeros(U,N);
for u = 1:U
    for n = 1:N
        cumrate(u,n) = (1/cb)*real(log2(det(S(:,:,u,n))));
    end
end
bun(idx,:) = diff([zeros(1,N); cumrate]);
w(idx)=w;
end
```

maxRESMACMIMO.m

```
% function [Rxxs, Eun, w, bun] = maxRESMACMIMO(H, Lxu, Eu, theta , cb)
%
% maxRMAC_vector_cvx Maximize weighted rate sum subject to energy
% constaint. This uses cvx and mosek.
%
%   INPUTS:
% H(:,u,n): Ly x U x N channel matrix. Ly = number of receiver antennas,
```

```
%     U = number of users and N = number of tones.
%
%     If the channel is real-bbd (cb=2), maxPMAC_cvx realizes user data rates
%     over all the tones (or equivalently positive and negative
%     freqs). This means the input H must be conjugate symmetric H_n =
%     H_{N-n}^*.  The program will reduce N by 2 and focus energy on the
%     lower half of frequencies. Thus N>1 must be even for real channels.
%     For real N=1 channel, the program runs and will produce half the data
%     rates for the same complex N=1 channel.
%
%     By constast if cb=1 (cplx bbd), maxRMAC_cvx need not have a conjugate
%     symmetric H and N is not reduced.
%
% Lxu 1 x U vector of user number of antennas
%
% Eu: TOTAL SUM-ENERGY/symbol constraint for all user inputs.
%
% theta: weight on each user's rate, length-U vector.
%
% cb: =1 if H is complex baseband, and cb=2 if H is real baseband.
%   Outputs:
%   OUTPUTS:
% Rxxs:  U-by-N cell array containing Rxx(u,n)'s if Lxu is a length-U
%        vector; or Lxu-by-Lxu-by-U-by-N tensor if Lxu is a scalar.
% Eun:    U x N energy distribution. Eun(u,n) is user u's energy allocation
%         on tone n.
% w:   U x 1 Lagrangian multiplier w.r.t. energy constraints
% bun: U by N bit distributions for all users.
%       - Rxxs:
%       - Eun:  U-by-N matrix showing users' transmit energy on each tone
%       - w: U-by-1 Lagrangian multiplier w.r.t. energy constraints
%       - bun: U-by-N matrix showing users' rate on each tone
% ***********************************************************************
function [Rxxs, Eun, w, bun] = maxRESMACMIMO(H, Lxu, Eu, theta, cb)

UNIFORM_FLAG = 0;
[Ly, ~, N] = size(H);
if N>1
    if cb==2
    H=H(:,:,1:N/2);
    N=N/2;
    end
end
if N == 1
    H = reshape(H,Ly,[],1);
end
Eu = reshape(Eu,[],1);
theta = reshape(theta,[],1);
[stheta, idx] = sort(theta, 'descend');
delta = -diff([stheta;0]);
U = length(Eu);
if length(Lxu) == 1
    Lxu = ones(1,U)*Lxu;
    UNIFORM_FLAG = 1;
```

```
end
Lxu_max = max(Lxu);
index_end = cumsum(Lxu);
index_start = [1,index_end(1:end-1)+1];
Lxu = Lxu(idx);
index_start = index_start(idx);
index_end = index_end(idx);
cvx_begin quiet
cvx_solver mosek
    variable rxxs(Lxu_max, Lxu_max, U, N) hermitian
    dual variable w
    expressions r(U,N) S(Ly, Ly) Eun(U,N)
    for n = 1:N
        S=eye(Ly);
        for u = 1:U
            S = S + H(:,index_start(u):index_end(u),n)...
                *rxxs(1:Lxu(u),1:Lxu(u),u,n)...
                *H(:,index_start(u):index_end(u),n)';
            r(u,n) = log_det(S);
            Eun(u,n) = trace(rxxs(1:Lxu(u),1:Lxu(u),u,n));
        end
    end
    maximize sum(delta'*r)
    subject to
        w: sum(sum(Eun,2))<=Eu;
        for u = 1:U
            for n=1:N
                rxxs(1:Lxu(u),1:Lxu(u),u,n)==hermitian_semidefinite(Lxu(u));
            end
        end
cvx_end

cumrate = zeros(U,N);
for n = 1:N
    S=eye(Ly);
    for u = 1:U
        S = S + H(:,index_start(u):index_end(u),n)...
            *rxxs(1:Lxu(u),1:Lxu(u),u,n)...
            *H(:,index_start(u):index_end(u),n)';
        cumrate(u,n) = (1/cb)*real(log2(det(S)));
    end
end
bun(idx,:) = diff([zeros(1,N); cumrate]);
w(idx)=w;

if UNIFORM_FLAG
    Rxxs(:,:,idx,1:N) = rxxs;
    for u = 1:U
        for n = 1:N
            Eun(idx(u),n) = trace(Rxxs(:,:,u,n));
        end
    end
else
    Rxxs = cell(U,N);
```

```
    for u = 1:U
        for n = 1:N
            Rxxs{idx(u),n} = rxxs(1:Lxu(u),1:Lxu(u),u,n);
            Eun(idx(u),n) = trace(Rxxs{idx(u),n});
        end
    end
end
end
```

minPMAC.m and related programs

```
% function [Eun, theta, bun, FEAS_FLAG, bu_a, info] = minPMAC(H, bu, w, cb)
%
% This main function contains ellipsoid method part and calls direclty or
%   indirectly 6 other functions.  minPMAC uses no CVX.  Another routine
%   minPMAC_cvx uses cvx and usually runs longer.  However, these scalar
%   minPMAC programs assume each user has Lxu = 1;
%   There is a CVX-using program minPMACMIMO that allows variable Lxu.
%
% INPUTS:
% H(:,u,n): Ly x U x N channel matrix. Ly = number of receiver antennas,
%     U = number of users and N = number of tones.
%
%     If the channel is real-bbd (cb=2), minPMAC realizes user data rates
%     over the lower half of the tones (or equivalently, the positive
%     freqs), and directly corresponds to the input bu user-data rate vector.
%     N is the number of tones actually transmitted (so corresponds to a 2N
%     size FFT when cb=2).
%
%     By constast if cb=1 (cplx bbd), minPMAC realizes user data rates
%     over all tones, but uses the same core optmization, so the data rate
%     is halved internal to the program, realizing that half bu rate on the
%     lower tones, because the designer (cb=1) wants it over all tones.
%     N is the number of tones actually transmitted (so corresponds to a N
%     size full-complex FFT when cb=1).
%
% bu_min: U x 1 vector containing the target rates for all the users.
%
% w:  U x 1 vector containing the weights for each user's power.
%
% cb: =1 if H is complex baseband, and cb=2 if H is real baseband.
%
% OUTPUTS:
% Eun:    U by N energy distribution that minimizes the weighted-sum energy.
%       E(u,n) is user u's energy allocation on tone n.
% theta:the optimal U by 1 dual variable vector containing optimal weights
%     of rates. Theta determines the decoding order. Largest theta is
%     decoded last, and smallest first.
% bun,   U by N bit distributions for all users.
% FEAS_FLAG: indicator of achievability.
%           FEAS_FLAG=1 if the target is achieved by a single ordering;
%           FEAS_FLAG=2 if the target is achieved by time-sharing
% bu_a: U-by-1 vector showing achieved sum rate of each user.
% info: various length output depending on FEAS_FLAG
```

```
%           --if FEAS_FLAG=1: 1 x 4 cell array containing
%               {Rxxs, Eun, bun, theta} corresponds to the single vertex
%               there are no equal-theta user sets in this case
%           --if FEAS_FLAG=2: 1-x 6 cell array, with each row representing
%               a time-shared vertex {Rxxs, Eun, bun, theta, frac}
%               there are numclus equal-theta user sets in this case.
%
%  info's row entries in detail (one row for each vertex shared
%       - Eun:  U-by-N matrix showing users' transmit energy on each tone.
%           If infeasible, output 0.
%       - bun: U-by-N matrix showing users' rate on each tone. If
%           infeasible, output 0.
%       - theta: U-by-1 Lagrangian multiplier w.r.t. target rates
%       - order: produces the order from left(best) to right for vertex
%       - frac: fraction of dimensions for each vertex in time share (FF =2
%           ONLY)
%       - cluster: index (to which cluster the user belongs; 0 means no
%           cluster)
%
% Subroutines called directly are
%       startEllipse.m
%       Lag_dual_f.m
%   and indirectly
%       minPtone.m
%       Hessian.m
%       eval_f.m
%       fmwaterfill_gn.,
%
% *************************************************************************
function [Eun, theta, bun, FEAS_FLAG, bu_a, info] = minPMAC_new(H, bu_min, w, cb)

tstart=tic;
bu_min=1/(3-cb)*bu_min;
err=1e-9;
conv_tol = 1e-2;% error tolerance

count = 0;
[Ly, U, N] = size(H);
w = reshape(w, U, 1);
bu_min = reshape(bu_min, U, 1);
bun = zeros(U,N);
Eun = zeros(U,N);

[A, g, w] = startEllipse(H, bu_min, w);         % starting ellipsoid
theta = g;

bu_min = bu_min  * log(2);                       % conversion from bits to nuts

while 1
                                        % Ellipsoid method starts here
    [~, bun, Eun] = Lag_dual_f(H, theta, w, bu_min);
    g = sum(bun,2) - bu_min;                    % sub-gradient
```

```
    if sqrt(g' * A * g) <= err           % stopping criteria
        break
    end
                                         % Updating the ellipsoid
    tmp = A*g / sqrt(g' * A * g);
    theta = theta - 1 / (U + 1) * tmp;

    A = U^2 / (U^2 - 1) * (A - 2 / (U + 1) * (tmp * tmp'));

    ind = find(theta < zeros(U,1));

    while ~isempty(ind)                  % This part is to make sure that theta is feasible,
        g = zeros(U,1);                  % it was not covered in the lecture notes and you may skip
        g(ind(1)) = -1;
        tmp = A * g / sqrt(g' * A * g);
        theta = theta - 1 / (U + 1) * tmp;
        A = U^2 / (U^2 - 1) * (A - 2 / (U + 1) * (tmp * tmp'));
        ind = find(theta < zeros(U,1));
    end
    count = count+1;
end

bun = (3-cb)*bun /log(2);                              % conversion from nats to bits

  %--------------------------------
 bu_min = (3-cb)*bu_min /log(2);
 bu_a=sum(bun,2);


% --------- REORDER USERS ACCORDING TO THETA -------
% This ordering will need eventually to be reversed before returing from
% this function.
[theta , Itheta] = sort(theta, 'descend');
[~, Jtheta]=sort(Itheta);
% theta = theta(Jtheta) reverses this sort later.
bu_a=bu_a(Itheta);
bun=bun(Itheta,:);
Eun=Eun(Itheta,:);
bu_min=bu_min(Itheta);
H=H(:,Itheta,:);

% --------------  SIMPLE CASE OF NO EQUAL-THETA USERS -------------
% This next section is implemented only if there are no equal-theta users
% and then returns results.
if isempty(find(abs(diff(theta)) <= 1e-5*min(theta), 1))
    FEAS_FLAG = 1;
    bu_a=sum(bun,2);
    % restore the original order
    bu_a=bu_a(Jtheta);
    bu_v=bu_a';
    theta = theta(Jtheta);
    bun=bun(Jtheta,:);
    Eun=Eun(Jtheta,:);
    % make table and exit
```

```
    info = table(bu_v, {Eun}, {bun}, {theta}, {Itheta(end:-1:1)}); % detailed info of boundary verti
    info.Properties.VariableNames(2:end) = {'Eun' 'bun' 'theta', 'order'};
    toc(tstart)
    return  % ARRIVE HERE AND SIMPLE CASE IS DONE.PROGRAM OVER
end
%
%---------------------TWO OR MORE EQUAL-THETA USERS ------------------
% This point of program is only reached if there are equal theta values for
% two or more users.  A convex combination of those orders different rate
% vectors will implement vertex sharing.  The info table above does not
% exist if we are in this section.  So it will be initialized and then
% vertices added to it.

% Only need vertices corresponding to the number of equal-theta users,
% which number will be the sizeset+1 that this section generates.  Thus,
% there is no need for U! orders to be searched (which is usually a much
% larger number.

% ---------- ENUMERATION OF POSSIBLE ORDERS --------------
% there will be setsize+1 orders to check when this section completes as
% the top setsize+1 rows of the matrix order.
% invorder restores the original order

spots = diff([-theta', 0]) < 1e-7*norm(theta);
sizeclus = []; % size of each cluster
numclus = 0; % number of clusters (>1 equal-theta groups)
set_thetaeq = []; % index of the first entry of each cluster

i = 1;
while i <= U
    sizeset = 1;
    flagclus = 0;
    while spots(i) == 1
        sizeset = sizeset + 1;
        if ~flagclus % enter a new cluster, save the last user
            set_thetaeq = [set_thetaeq, i];
            flagclus = 1;
            numclus = numclus + 1;
        end
        i = i+1;
    end
    if sizeset > 1
        sizeclus = [sizeclus, sizeset];
    end
    i = i+1;
end

order=repmat(1:U,sum(sizeclus-1),1);
cumsize = cumsum([0,sizeclus-1]);

for jdx = 1:numclus
    Jright = eye(sizeclus(jdx));
    Jright = [Jright(:,end), Jright(:,1:end-1)];
    Jleft = Jright';
```

```
    u_range = set_thetaeq(jdx):set_thetaeq(jdx)+sizeclus(jdx)-1; % users in cluster jdx
    for i = 1:sizeclus(jdx)-1
        order(cumsize(jdx)+i, u_range) = u_range * Jleft^(i); % all permutated orders (excluding 1,2
    end
end


% ----------- FIND FIRST VERTEX & CREATE INFO TABLE ----------
bu_v = sum(bun,2)'; % 1xU
initialbu_v = bu_v;
initialbun = bun;
firstvertices = de2bi(0:2^U-1).*initialbu_v;

% The vertices will be stored in a table that is indexed by bu_v
initialinfo = table(bu_v, {Eun}, {bun}, {theta}, {U:-1:1}); % detailed info of boundary vertices
initialinfo.Properties.VariableNames(2:end) = {'Eun' 'bun' 'theta', 'order'};

% ----- For each cluster -----------
for jdx=1:numclus
    if jdx == 1
        Sinit = repmat(eye(Ly),1,1,N);
        cumrateinit = zeros(1, N);
        for n = 1:N
            for i = 1:set_thetaeq(1)-1
                Sinit(:,:,n) = Sinit(:,:,n) + H(:,i,n)*Eun(i,n)*H(:,i,n)';
            end
            cumrateinit(n) = (1/cb)*real(log2(det(Sinit(:,:,n))));
        end

    else
        for n = 1:N
            for i = set_thetaeq(jdx-1):set_thetaeq(jdx-1)+sizeclus(jdx)-1
                Sinit(:,:,n) = Sinit(:,:,n) + H(:,i,n)*Eun(i,n)*H(:,i,n)';
            end
            cumrateinit(n) = (1/cb)*real(log2(det(Sinit(:,:,n))));
        end
    end
    u_range = set_thetaeq(jdx):set_thetaeq(jdx)+sizeclus(jdx)-1;
    bu_v=initialbu_v;
    known_vertices = initialinfo; % detailed info of boundary vertices
    bd_vertices = bu_v(u_range);    % track critical boundary vertices
    vertices = de2bi(0:2^sizeclus(jdx)-1).*bd_vertices;
    if min(bd_vertices - bu_min(u_range)') >= -conv_tol % achievable w/o time share
        info = known_vertices;
        info.frac = 1;
        info.clusterID = jdx;
        if jdx == 1
            Big_info = info;
        else
            Big_info = [Big_info; info];
        end
        continue;
    end
    bd_V = 1;
```

```
    for idx = cumsize(jdx)+1:cumsize(jdx)+sizeclus(jdx)-1 % add more vertices in cluster jdx
        cumrate = zeros(sizeclus(jdx)+1, N);
        cumrate(1,:) = cumrateinit;
        for n = 1:N
            rel_idx = 2;
            S = Sinit(:,:,n);
            for u = u_range
                u_or = order(jdx, u);
                S = S + H(:,u_or,n)*Eun(u_or,n)*H(:,u_or,n)';
                cumrate(rel_idx,n) = (1/cb)*real(log2(det(S)));
                rel_idx = rel_idx + 1;
            end
        end
        bun = initialbun;
        bun(u_range,:) = diff(cumrate);
        bun(order(idx,:),:) = bun;
        bu_v = sum(bun, 2)';

        bd_V = bd_V + 1;
        bd_vertices_extend = [bd_vertices; bu_v(u_range)];
        known_vertices = [known_vertices; {bu_v, {Eun}, {bun}, {theta}, {order(idx,end:-1:1)}}];
        vertices = [vertices; de2bi(0:2^sizeclus(jdx)-1).*bu_v(u_range)];
        tess = convhulln(vertices);
        vertices = vertices(unique(tess),:);
        bd_vertices = intersect(bd_vertices_extend, vertices, 'rows', 'stable');
        tess = convhulln(vertices);
        % delete inner vertices
        if size(bd_vertices, 1) < bd_V
            to_remove = setdiff(bd_vertices_extend, bd_vertices, 'rows');
            known_vertices(ismember(known_vertices.bu_v(u_range), to_remove, 'rows'),:) = [];
            bd_V = size(bd_vertices, 1);
        end

        if inhull(bu_min(u_range)', vertices, tess, conv_tol) % bu_min achievable by time-share
            FEAS_FLAG = 2;
            frac = bd_vertices'\bu_min(u_range);
            a_v = find(frac >= err); % active vertices in time-share
            info = known_vertices(a_v, :);
            info.frac = frac(a_v);
            info.clusterID = ones(length(a_v), 1)*jdx;
            bu_a(u_range) = bd_vertices(a_v,:)'*frac(a_v);
            break;
        end
    end

    % write big info table
    if jdx == 1
        Big_info = info;
    else
        Big_info = [Big_info; info];
    end
end

info = Big_info;
```

```
bd_V = size(info.bu_v, 1);

% ----------- RESTORE ORIGINAL ORDER TO ALL ----------------
theta = theta(Jtheta);
temptheta = reshape(cell2mat(info.theta), U, bd_V);
temptheta = temptheta(Jtheta,:);
info.theta= mat2cell(temptheta', [ones(1,bd_V)] , U);
%
tempbun=reshape(cell2mat(info.bun),U,bd_V,N);
tempbun=tempbun(Jtheta,:,:);
tempbun=permute(tempbun,[2 1 3]);
info.bun=mat2cell(tempbun,[ones(1,bd_V)], U , N);
%
tempEun=reshape(cell2mat(info.Eun),U,bd_V,N);
tempEun=tempEun(Jtheta,:,:);
tempEun=permute(tempEun,[2 1 3]);
info.Eun=mat2cell(tempEun,[ones(1,bd_V)], U , N);
%
bu_a = bu_a(Jtheta);
info.bu_v=info.bu_v(:,Jtheta);
%
tmporder = cell2mat(info.order);
tmporder = Itheta(tmporder);
info.order = mat2cell(tmporder, ones(1,bd_V), U);
toc(tstart)
end
```

minPtone.m

```
% function [f, b, e] = minPtone(H, theta, w, N)
%
% minPtone maximizes f = sum_{u=1}^U lambda_u * b_u - sum_{u=1}^U w_u * e_u
% subject to b \in C_g(G,e); presumably for a single tone.
%
% Inputs:
% 1)  H, an Ly by U channel matrix. Ly is the number of receiver antennas,
%     U is the total number of users.
%     H(:,u) is the channel for user u. minPtone assumes each user
%     has only a single transmit antenna, thus H(:,u) is a column vector.
% 2)  theta, a U by 1 vector containing the weights for the rates.
% 3)  w, a U by 1 vector containing the weights for each user's power.
% 4)  N is needed to pass to subroutines so that PSD is increased at fixed
%     sampling rate relative to noise-whitening.
%
% Outputs:
% 1)  f, the minimum value (or maximum value of the -1 * function).
% 2)  b, a U by 1 vector containing the rates for all users
%     that optimizes the given function.
% 3)  e, a U by 1 vector containing the PSD for all users
%     that optimizes the given function.
%
% Subroutines called are eval_f , Hessian
function [f, b, e] = minPtone(H, theta, w)
```

```
[Ly, U] = size(H);
[stheta, ind] = sort(-theta);
stheta = -stheta;
sH = H(:,ind);
sw = w(ind);



NT_max_it = 1000;                              % Maximum number of Newton's
                                               % method iterations
dual_gap = 1e-6;
mu = 10;                                       % step size for t
alpha = 0.01;                                  % back tracking line search parameters
beta = 0.5;

count = 1;
nerr = 1e-5;                                   % acceptable error for inner loop
                                               % Newton's method

e = ones(U,1);                                 % Strictly feasible point;

t = .1;
l_p = 1;                                       % for newton's method termination



while (1+U)/t > dual_gap
   t =  t * mu;
   l_p = 1;
   count = 1;
   while l_p > nerr & count < NT_max_it

      f_val = eval_f(t * stheta, sH, e, t * sw);
                                               % calculating function value

                                               % calculating the hessian and gradient
      [g, h] = Hessian(t * stheta, sH, e, t * sw);

      de = -real(h\g);                                % search direction

      l_p = g' * de;                           % theta(e)^2 for Newton's method

      s = 1;                                   % checking e = e+s*de feasible
                                               % and also back tracking algorithm


      e_new = e + s * de;



      if e_new > zeros(U,1)
          f_new = eval_f(t * stheta, sH, e_new, t * sw);
          feas_check = (f_new > f_val + alpha * s * g' * de);
      else
```

```
          feas_check = 0;
      end

      while ~feas_check
         s = s * beta;
         if s < 1e-40
             l_p = nerr/2;
             break
         end
         e_new = e + s * de;

         if e_new > zeros(U,1)
             f_new = eval_f(t * stheta, sH, e_new, t * sw);
             feas_check = (f_new > f_val + alpha * s * g' * de);
         else
             feas_check = 0;
         end

      end

      e = e + s * de;                          % update e
      count = count + 1;                       % number of Newtons method iterations
   end

end


M = eye(Ly) + sH(:,1) * sH(:,1)' * e(1);
b = zeros(U,1);
b(1) =  0.5 * real(log(det(M)));
for u = 2:U
    b(u) = -0.5 * real(log(det(M)));
    M = M + sH(:,u) * sH(:,u)' * e(u);
    b(u) = b(u) + 0.5 * real(log(det(M)));
end


b(ind) = b;
e(ind) = e;
f = theta' * b - w' * e;
```

Lag_dual_f

```
% function [f, b, E] = Lag_dual_f(H, theta, w, bu);
%
% Lag_dual_f computes the Lagrange dual function by solving the
% optimization problem (calling the function minPtone) on each tone.
%
% Inputs:
% 1)  H, an Ly by U by N channel matrix. Ly is the number of receiver antennas,
%     U is the total number of users and N is the total number of tones.
%     H(:,:,n) is the channel matrix for all users on tone n
%     and H(:,u,n) is the channel for user u on tone n. Lag_dual_f assumes each user
%     only has single transmit antenna, thus H(:,u,n) is a column vector.
```

```
% 2)  theta, a U by 1 vector containing the weights for the rates.
% 3)  w, a U by 1 vector containing the weights for each user's power.
% 4)  bu, a U by 1 vector containing the target rates for all the users.

% Outputs:
% 1)  f, the Lagrange dual function value.
% 2)  b, a U by N vector containing the rates for all users and over all tones
%     that optimizes the Lagrangian. b(u,:) is the rate allocation for user
%     u over all tones.
% 3)  E, a U by N vector containing the powers for all users and over all tones
%     that optimizes the Lagrangian. E(u,:) is the power allocation for
%     user u over all tones.
%
% subroutines called: minPtone
function [f, b, E] = Lag_dual_f(H, theta, w, bu);

[Ly, U, N] = size(H);
f = 0;
b = zeros(U,N);
E = zeros(U,N);

% Performing optimization over all N tones,
for i = 1:N
    [temp, b(:,i), E(:,i)] = minPtone(H(:,:,i), theta, w);
    f = f + temp;
end

f = f - theta' * bu;
```

eval_f.m

```
% function f = eval_f(theta, H, e, w, N)
%
% eval_f evaluates the value of the function,
%
% f(e) = (theta_1 - theta_2)/2 * log det(I + H_1 * H_1' * e_1) +
% (theta_2 - theta_3)/2 * log det(I + H_1 * H_1' * e_1 + H_2 * H_2' * e_2) + ...
% (theta_{U-1} - theta_U)/2 * log det(I + H_1 * H_1' * e_1 + ... + H_{U-1} * H_{U-1}' * e_{U-1}) +
% theta_U/2 * log det(I + H_1 * G_1' * e_1 + ... + H_U * H_U' * e_U) - w^T * e + sum_{u=1}^U log(e_u
% theta should be in decreasing order, theta_1 >= theta_2 >= ... >=theta_U.
%
% the inputs are:
% 1)  theta, a U by 1 vector of weights for the rates.
% 2)  H, an Ly by U channel matrix. H(:,u) is the channel
%     vector for user u. Again each user has just one transmit antenna.
% 3)  e, a U by 1 vector containing each user's power.
% 4)  w, a U by 1 vector containing weights for each user's power
%
% the output is f the function value given above.
% no subroutines are called and this function does not need to know N
function f = eval_f(theta, H, e, w )

[Ly,U] = size(H);
```

```
theta = 0.5 * (theta - [theta(2:U); 0]);

M = zeros(Ly,Ly,U);                          % M(:,:,i) = (I + sum_{u=1}^i G_u * G_u^H * e_u)
                                            % M is computed recursively

M(:,:,1) = eye(Ly) + H(:,1) * H(:,1)' * e(1);
f = theta(1) * log(det(M(:,:,1))) + log(e(1)) - w(1) * e(1);
for u = 2:U
    M(:,:,u) = M(:,:,u-1) + H(:,u) * H(:,u)' * e(u);
    f = f + theta(u) * log(det(M(:,:,u))) + log(e(u)) -  w(u) * e(u);
end
```

Hessian.m

```
% function [g, H] = Hessian(theta, G, e, w)
%
% This function calculates the gradiant (g) and the Hessian (H) of the
% function f(e) = (theta_1 - theta_2)/2 * log det(I + G_1 * G_1^H * e_1) +
% (theta_2 - theta_3)/2 * log det(I + G_1 * G_1^H * e_1 + G_2 * G_2^H * e_2) + ...
% (theta_{U-1} - theta_U)/2 * log det(I + G_1 * G_1^H * e_1 + ... + G_{U-1} * G_{U-1}^H * e_{U-1}) +
% theta_U/2 * log det(I + G_1 * G_1^H * e_1 + ... + G_U * G_U^H * e_U) - w^T * e + sum_{u=1}^U log(e
% theta should be in decreasing order, theta_1 >= theta_2 >= ... >=theta_U.
%
% the inputs are:
% 1)  theta, a U by 1 vector of weights for the rates.
% 2)  G, an Ly by U channel matrix. G(:,u) is the channel
%     vector for user u. Again each user has just one transmit antenna.
% 3)  e, a U by 1 vector containing each user's power.
% 4)  w, a U by 1 vector containing weights for each user's power
%
% the outputs are:
% 1)  g, U by 1 gradiant vector.
% 2)  H, U by U Hessian matrix.
function [g, H] = Hessian(theta, G, e, w)
[Ly,U] = size(G);

theta = 0.5 * (theta - [theta(2:U); 0]);

M = zeros(Ly,Ly,U);                          % M(:,:,i) = (I + sum_{u=1}^i G_u * G_u^H * e_u)^{-1}
                                            % M is computed recursively using matrix inversion lemma

M(:,:,1) = eye(Ly) - G(:,1) * G(:,1)' * e(1) / (1 + e(1) * G(:,1)' * G(:,1));

for u = 2:U
    M(:,:,u) = M(:,:,u-1) - M(:,:,u-1) * G(:,u) * G(:,u)' * M(:,:,u-1) * e(u) / (1 + e(u) * G(:,u)'
end

g = zeros(U,1);

% g_u = sum_{j=u}^U theta_j * G_u * M_j * G_u^H - w_u + 1/e_u
for u = 1:U
    for j = u:U
        g(u) = g(u) + theta(j) * G(:,u)' * M(:,:,j) * G(:,u);
    end
```

```
end

g = g + 1./ e - w;

% H_{u,l} = sum_{j = max(u,l)}^U -theta_j * tr(G_u * G_u^H * M_j * G_l * G_l^H * M_j) - 1/e_u^2 * de

H = zeros(U,U);
for u = 1:U
    for l = 1:U
        for j = max(u,l):U
            H(u,l) = H(u,l) - theta(j) * trace(G(:,u) * G(:,u)' * M(:,:,j) * G(:,l) * G(:,l)' * M(:,
        end
    end
end
H = H - diag(1./(e.^2));
```

startEllipse.m

```
% function [A, g] = startEllipse(H, bu, w)%
% This function initializes the ellipsoid method for power minimization problem
% specifically for problem 13.11 in EE479 class.
%
% function [A, g] = startEllipse(H, bu)
% G containes the channel matrices as is defined in minPMAC.m and bu is
% the target U by 1 rate vector, w is the U by 1 power weights.
% For all fixed margin WF steps, decoding order is 1,2,...U.
%
% A is the matrix describing the staring ellipsoid and g is its center
% subroutine called fmwaterfill_gn

function [A, g] = startEllipse(H, bu, w)


[Ly, U, N] = size(H);

order = 1:U;                        % decoding order, arbitrary
tmax = zeros(U,1);
Rnoise = zeros(Ly,Ly,N);
g = zeros(U,N);

for u = 1:U
    en = zeros(U,N);
    b = bu;
    b(u) = b(u) + 1;
    for u1 = flipud(order)           % starting from last user.

        for index = 1:N
            Rnoise(:,:,index) = eye(Ly);
        end
        for u2 = flipud(order)       % Note that the users that have been decoded has zero E
            if u2 == u1
            else
                for index = 1:N
                    Rnoise(:,:,index) = Rnoise(:,:,index)/N + en(u2,index) * ...
```

```
                                        H(:,u2,index) * H(:,u2,index)';
                end
            end
        end
        for index = 1:N
            h_temp = sqrtm(Rnoise(:,:,index))\ H(:,u1,index);
            g(u1,index) = abs(svd(h_temp))^2;
        end
        [~, E_temp] = fmwaterfill_gn(g(u1,:), b(u1)/N, 0);
        en(u1,:) = E_temp;
    end
    tmax(u) = sum(w .* sum(en,2));    % \sum_n \sum_u \w_u En_u
end
A = eye(U) * sum(tmax.^2) / 4;        % an sphere centerced at tmax/2
g = tmax / 2;
```

fmwaterfill_gn.m

```
% function [bn, en] = fmwaterfill_gn(gn, b_bar, gap)
%
% INPUT
% gn is the channel gain (a row vector).
% b_bar is the target bit rate (b_total / Ntot)
% gap is the gap in dB
%
% OUTPUT
% en is the energy in the nth subchannel
% bn is the bit in the nth subchannel
%
% dB into normal scale
function [bn, en] = fmwaterfill_gn(gn, b_bar, gap)
gap = 10^(gap/10);

% Ntot

[col Ntot] = size(gn);

if col ~= 1
    error = 1
    return;
end

% initialization
en = zeros(1, Ntot);
bn = zeros(1, Ntot);

%%%%%%%%%%%%%%%%%%%%%%%%
% Now do waterfilling %
%%%%%%%%%%%%%%%%%%%%%%%%

%sort
[gn_sorted, Index] = sort(gn);   % sort gain, and get Index

gn_sorted = fliplr(gn_sorted);   % flip left/right to get the largest
```

```
                                % gain in leftside
Index = fliplr(Index);          % also flip index

num_zero_gn = length(find(gn_sorted == 0));
Nstar = Ntot - num_zero_gn;     % number of zero gain subchannels

                               % Number of used channels,
                               % start from Ntot - (number of zero gain subchannels)

while(1)
                                % The K calculation has been modified in order to
                                % accomodate the size of the number
    logK = log(gap) + 1/Nstar * ((Ntot) * b_bar * 2 * log(2) - sum(log(gn_sorted(1:Nstar))));
    K = exp(logK);
  En_min = K - gap/gn_sorted(Nstar);
                                % En_min occurs in the worst channel
  if (En_min<0)
     Nstar = Nstar - 1;   % If negative En, continue with less channels
  else
     break;               % If all En positive, done.
  end
end

En = K - gap./gn_sorted(1:Nstar); % Calculate En
Bn =.5 * log2(K * gn_sorted(1:Nstar)/gap);  % Calculate bn

bn(Index(1:Nstar))=Bn; % return values in original index
en(Index(1:Nstar))=En; % return values in original index
```

minPMACMIMO This is a more general minPMAC program that allows variable number of antennas and uses CVX.

```
% function [FEAS_FLAG, bu_a, info] = minPMACMIMO(H, Lxu, bu_min, w, cb)
%
% INPUTS:
%  H(:,u,n): Ly x U x N channel matrix. Ly = number of receiver antennas,
%            U = number of users and N = number of tones.
%
%     If the channel is real-bbd (cb=2), minPMAC realizes user data rates
%     over the lower half of the tones (or equivalently, the positive
%     freqs), and directly corresponds to the input bu_min user-data rate vector.
%
%     By constast if cb=1 (cplx bbd), minPMAC realizes user data rates over
%     all tones, but uses the same real-bbd core optmization that doubles the
%     number of tones with an equivalent-gain set, so this program halves
%     the data rate internally and realizes that half bu_min rate on the
%     lower tones, which is the input set for which results are reported.
%
% Lxu:  1 x 1 or 1 x U vector containing each users' number of
%       transmit antennas. If Lxu is 1 x 1, each user has Lxu antennas.
%
% bu_min: U x 1 vector containing the target rates for all the users.
%
% w:  U x 1 vector containing the weights for each user's energy.
```

```
%
% cb: =1 if H is complex baseband, and cb=2 if H is real baseband.
%   Outputs:
%       - FEAS_FLAG: indicator of achievability.
%           FEAS_FLAG=1 if the target is achieved by a single ordering;
%           FEAS_FLAG=2 if the target is achieved by time-sharing
%       - bu_a: U-by-1 vector showing achieved sum rate of each user.
%       - info: various length output depending on FEAS_FLAG
%           --if FEAS_FLAG=1: 1 x 4 cell array containing
%               {Rxxs, Eun, bun, theta} corresponds to the single vertex
%               there are no equal-theta user sets in this case
%           --if FEAS_FLAG=2: 1-x 6 cell array, with each row representing
%               a time-shared vertex {Rxxs, Eun, bun, theta, frac}
%               there are numclus equal-theta user sets in this case.
%
%      info's row entries in detail (one row for each vertex shared
%       - Rxxs: U-by-N cell array containing Rxx(u,n)'s if Lxu is a
%           length-U vector; or Lxu-by-Lxu-by-U-by-N tensor if Lxu is a
%           scalar. If the rate target is infeasible, output 0.
%       - Eun:  U-by-N matrix showing users' transmit energy on each tone.
%           If infeasible, output 0.
%       - bun: U-by-N matrix showing users' rate on each tone. If
%           infeasible, output 0.
%       - theta: U-by-1 Lagrangian multiplier w.r.t. target rates
%       - order: produces the order from left(best) to right for vertex
%       - frac: fraction of dimensions for each vertex in time share (FF =2
%           ONLY)
%       - cluster: index (to which cluster the user belongs; 0 means no
%           cluster)
%
% Functions called are
%       startEllipse_var_Lxu
%       minPtoneMIMO
% ***********************************************************************
function [FEAS_FLAG, bu_a, info] = minPMACMIMO(H, Lxu, bu_min, w, cb)
tstart=tic;
bu_min=1/(3-cb)*bu_min;
err=1e-9;
conv_tol = 1e-1;
[Ly, ~, N] = size(H);
U = length(w);
% Yun, I don't think these next 3 lines change anything?
  if N == 1
    H = reshape(H,Ly,[],1);
  end
w = reshape(w, U, 1);
bu_min = reshape(bu_min, U, 1);

% initialize
count = 0;
bun = zeros(U,N);
Eun = zeros(U,N);
Rxxs = cell(1,N);
[A, theta] = startEllipse_var_Lxu(H, Lxu, bu_min, w);
```

```
bu_min = bu_min*log(2);

  if length(Lxu) == 1
    Lxu = ones(1,U)*Lxu;
    UNIFORM_FLAG = 1;
  end

while 1
    % solve for each tone
    parfor tone = 1:N
        [~, bun(:,tone), Rxxs{tone}, Eun(:,tone)] = minPtoneMIMO(H(:,:,tone), Lxu, theta, w);
    end
    % update ellipsoid for theta
    g = sum(bun,2) - bu_min;
    % stopping criteria
    tmp_err = sqrt(g'*A*g);
    if tmp_err <= err
        break
    end
    gt = g/tmp_err;
    tmp = A*gt;
    theta = theta - 1/(U + 1)*tmp;
    A = U^2/(U^2 - 1)*(A - 2/(U + 1)*(tmp*tmp'));
    ind = find(theta < zeros(U,1));

    while ~isempty(ind) % This part is to make sure that theta is feasible,
        g = zeros(U,1);
        g(ind(1)) = -1;
        tmp = A * g / sqrt(g' * A * g);
        theta = theta - 1 / (U + 1) * tmp;
        A = U^2 / (U^2 - 1) * (A - 2 / (U + 1) * (tmp * tmp'));
        ind = find(theta < zeros(U,1));
    end
    count = count+1;
end

% if max(Lxu) == 1
%     Rxxs = Eun;
% end
bun = (3-cb)*bun /log(2);
bu_min = (3-cb)*bu_min /log(2);
bu_a=sum(bun,2);









% --------- REORDER USERS ACCORDING TO THETA -------
% This ordering will need eventually to be reversed before returing from
```

```
% this function.
[theta , Itheta] = sort(theta, 'descend');
[~, Jtheta]=sort(Itheta);
% theta = theta(Jtheta) reverses this sort later.
bu_a=bu_a(Itheta);
bun=bun(Itheta,:);
Eun=Eun(Itheta,:);
bu_min=bu_min(Itheta);
index_end = cumsum(Lxu);
index_start = [1, index_end(1:end-1)+1];
hidx = [];
for i = Itheta'
    hidx = [hidx, index_start(i):index_end(i)];
end
Lxu = Lxu(Itheta);
index_end = cumsum(Lxu);
index_start = [1, index_end(1:end-1)+1];
H=H(:,hidx,:);
for tone=1:N
    Rxxs{tone}= Rxxs{tone}(Itheta);
end

% --------------  SIMPLE CASE OF NO EQUAL-THETA USERS -------------
% This next section is implemented only if there are no equal-theta users
% and then returns results.
if isempty(find(abs(diff(theta)) <= 1e-5*min(theta), 1))
    FEAS_FLAG = 1;
    bu_a=sum(bun,2);
    % restore the original order
    bu_a=bu_a(Jtheta);
    bu_v=bu_a';
    theta = theta(Jtheta);
    bun=bun(Jtheta,:);
    Eun=Eun(Jtheta,:);
    for tone=1:N
       Rxxs{tone}= Rxxs{tone}(Jtheta);
    end
    % make table and exit
    info = table(bu_v, Rxxs, {Eun}, {bun}, {theta}, {Itheta(end:-1:1)}); % detailed info of boundary
    info.Properties.VariableNames(2:end) = {'Rxxs', 'Eun', 'bun', 'theta', 'order'};
    toc(tstart)
    return  % ARRIVE HERE AND SIMPLE CASE IS DONE.PROGRAM OVER
end
%
%--------------------TWO OR MORE EQUAL-THETA USERS ------------------
% This point of program is only reached if there are equal theta values for
% two or more users.  A convex combination of those orders different rate
% vectors will implement vertex sharing.  The info table above does not
% exist if we are in this section.  So it will be initialized and then
% vertices added to it.

% Only need vertices corresponding to the number of equal-theta users,
% which number will be the sizeset+1 that this section generates.  Thus,
% there is no need for U! orders to be searched (which is usually a much
```

```
% larger number.

% ----------- ENUMERATION OF POSSIBLE ORDERS ---------------
% there will be setsize+1 orders to check when this section completes as
% the top setsize+1 rows of the matrix order.
% invorder restores the original order

spots = diff([-theta', 0]) < 1e-7*norm(theta);
sizeclus = []; % size of each cluster
numclus = 0; % number of clusters (>1 equal-theta groups)
set_thetaeq = []; % index of the first entry of each cluster

i = 1;
while i <= U
    sizeset = 1;
    flagclus = 0;
    while spots(i) == 1
        sizeset = sizeset + 1;
        if ~flagclus % enter a new cluster, save the last user
            set_thetaeq = [set_thetaeq, i];
            flagclus = 1;
            numclus = numclus + 1;
        end
        i = i+1;
    end
    if sizeset > 1
        sizeclus = [sizeclus, sizeset];
    end
    i = i+1;
end

order=repmat(1:U,sum(sizeclus-1),1);
cumsize = cumsum([0,sizeclus-1]);

for jdx = 1:numclus
    Jright = eye(sizeclus(jdx));
    Jright = [Jright(:,end), Jright(:,1:end-1)];
    Jleft = Jright';
    u_range = set_thetaeq(jdx):set_thetaeq(jdx)+sizeclus(jdx)-1; % users in cluster jdx
    for i = 1:sizeclus(jdx)-1
        order(cumsize(jdx)+i, u_range) = u_range * Jleft^(i); % all permutated orders (excluding 1,2
    end
end


% ----------- FIND FIRST VERTEX & CREATE INFO TABLE ----------
bu_v = sum(bun,2)'; % 1xU
initialbu_v = bu_v;
initialbun = bun;
firstvertices = de2bi(0:2^U-1).*initialbu_v;

% The vertices will be stored in a table that is indexed by bu_v
initialinfo = table(bu_v, Rxxs, {Eun}, {bun}, {theta}, {U:-1:1}); % detailed info of boundary vertic
initialinfo.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta' 'order'};
```

```
% ----- For each cluster -----------
for jdx=1:numclus
    if jdx == 1
        Sinit = repmat(eye(Ly),1,1,N);
        cumrateinit = zeros(1, N);
        for n = 1:N
            for i = 1:set_thetaeq(1)-1
                Sinit(:,:,n) = Sinit(:,:,n) + H(:,index_start(i):index_end(i),n)*Rxxs{n}{i}*H(:,inde
            end
            cumrateinit(n) = (1/cb)*real(log2(det(Sinit(:,:,n))));
        end

    else
        for n = 1:N
            for i = set_thetaeq(jdx-1):set_thetaeq(jdx-1)+sizeclus(jdx)-1
                Sinit(:,:,n) = Sinit(:,:,n) + H(:,index_start(i):index_end(i),n)*Rxxs{n}{i}*H(:,inde
            end
            cumrateinit(n) = (1/cb)*real(log2(det(Sinit(:,:,n))));
        end
    end
    u_range = set_thetaeq(jdx):set_thetaeq(jdx)+sizeclus(jdx)-1;
    bu_v=initialbu_v;
    known_vertices = initialinfo; % detailed info of boundary vertices
    bd_vertices = bu_v(u_range);    % track critical boundary vertices
    vertices = de2bi(0:2^sizeclus(jdx)-1).*bd_vertices;
    if min(bd_vertices - bu_min(u_range)') >= -conv_tol % achievable w/o time share
        info = known_vertices;
        info.frac = 1;
        info.clusterID = jdx;
        if jdx == 1
            Big_info = info;
        else
            Big_info = [Big_info; info];
        end
        continue;
    end
    bd_V = 1;
    for idx = cumsize(jdx)+1:cumsize(jdx)+sizeclus(jdx)-1 % add more vertices in cluster jdx
        cumrate = zeros(sizeclus(jdx)+1, N);
        cumrate(1,:) = cumrateinit;
        for n = 1:N
            rel_idx = 2;
            S = Sinit(:,:,n);
            for u = u_range
                u_or = order(jdx, u);
                S = S + H(:,index_start(u_or):index_end(u_or),n)*Rxxs{n}{u_or}*H(:,index_start(u_or)
                cumrate(rel_idx,n) = (1/cb)*real(log2(det(S)));
                rel_idx = rel_idx + 1;
            end
        end
        bun = initialbun;
        bun(u_range,:) = diff(cumrate);
        bun(order(idx,:),:) = bun;
```

```
        bu_v = sum(bun, 2)';

        bd_V = bd_V + 1;
        bd_vertices_extend = [bd_vertices; bu_v(u_range)];
        known_vertices = [known_vertices; {bu_v, Rxxs, {Eun}, {bun}, {theta}, {order(idx,end:-1:1)}}
        vertices = [vertices; de2bi(0:2^sizeclus(jdx)-1).*bu_v(u_range)];
        tess = convhulln(vertices);
        vertices = vertices(unique(tess),:);
        bd_vertices = intersect(bd_vertices_extend, vertices, 'rows', 'stable');
        tess = convhulln(vertices);
        % delete inner vertices
        if size(bd_vertices, 1) < bd_V
            to_remove = setdiff(bd_vertices_extend, bd_vertices, 'rows');
            known_vertices(ismember(known_vertices.bu_v(u_range), to_remove, 'rows'),:) = [];
            bd_V = size(bd_vertices, 1);
        end

        if inhull(bu_min(u_range)', vertices, tess, conv_tol) % bu_min achievable by time-share
            FEAS_FLAG = 2;
            frac = bd_vertices'\bu_min(u_range);
            a_v = find(frac >= err); % active vertices in time-share
            info = known_vertices(a_v, :);
            info.frac = frac(a_v);
            info.clusterID = ones(length(a_v), 1)*jdx;
            bu_a(u_range) = bd_vertices(a_v,:)'*frac(a_v);
            break;
        end
    end

    % write big info table
    if jdx == 1
        Big_info = info;
    else
        Big_info = [Big_info; info];
    end
end

info = Big_info;
bd_V = size(info.bu_v, 1);

% ----------- RESTORE ORIGINAL ORDER TO ALL ----------------
theta = theta(Jtheta);
temptheta = reshape(cell2mat(info.theta), U, bd_V);
temptheta = temptheta(Jtheta,:);
info.theta= mat2cell(temptheta', [ones(1,bd_V)] , U);
%
tempbun=reshape(cell2mat(info.bun),U,bd_V,N);
tempbun=tempbun(Jtheta,:,:);
tempbun=permute(tempbun,[2 1 3]);
info.bun=mat2cell(tempbun,[ones(1,bd_V)], U , N);
%
tempEun=reshape(cell2mat(info.Eun),U,bd_V,N);
tempEun=tempEun(Jtheta,:,:);
tempEun=permute(tempEun,[2 1 3]);
```

```
info.Eun=mat2cell(tempEun,[ones(1,bd_V)], U , N);
%
bu_a = bu_a(Jtheta);
info.bu_v=info.bu_v(:,Jtheta);
%
tmporder = cell2mat(info.order);
tmporder = Itheta(tmporder);
info.order = mat2cell(tmporder, ones(1,bd_V), U);

for tone=1:N
    Rxxs{tone}= Rxxs{tone}(Jtheta);
end
for u=1:U
    for n=1:N
        info.Rxxs{n}{u}=Rxxs{n}{u};
    end
end
toc(tstart)
end
```

This CVX based program uses two modified versions of the suboutines above as

minPtoneMIMO.m

```
% function [f, bn, Rxxs, En] = minPtoneMIMO(H, Lxu, theta, w)
% minPtone_cvx_var_Lxu minimizes f = - sum_{u=1}^U theta_u * b_u +
% sum_{u=1}^U w_u * e_u
% subject to b \in C_g(H,e). This program uses cvx package and mosek solver
%
% the inputs are:
% 1)  H, an Ly by Lx channel matrix. Ly is the number of receiver antennas,
%     Lx is the total number of transmit antennas.
%     H(:,index_start(u):index_end(u)) is the channel for user u.
% 2)  Lxu, a scalar or a length-U vector containing the number of transmit
%     antennas of each user. If Lxu is a scalar, each user has Lxu antennas
% 3)  theta, a U by 1 vector containing the weights for the rates.
% 4)  w, a U by 1 vector containing the weights for each user's energy.
%
% the outputs are:
% 1)  f, -1 * minimum value (or maximum value of the -1 * function).
% 2)  bn, a U by 1 vector containing the rates for each user
%     that optimizes the given function.
% 3)  Rxxs, a 1 by U cell array containing the Rxx's for each user that
%     optimizes the given function
% 4)  En, a U by 1 vector containing the energy of each user
% ------------------------------------------------------------------
function [f, bn, Rxxs, En] = minPtoneMIMO(H, Lxu, theta, w)
[Ly, ~] = size(H);
U = length(theta);
if length(Lxu) == 1
    Lxu = ones(1,U)*Lxu;
end
Lxu_max = max(Lxu);
index_end = cumsum(Lxu);
index_start = [1,index_end(1:end-1)+1];
```

```
[stheta, order] = sort(theta, 'descend');
D = eye(U)+diag(-ones(U-1,1),1);
max_bound = max(-diff([stheta;0]));
if max_bound > 1e4
    stheta = stheta/max_bound*1e2;
    w = w/max_bound*1e2;
end
cvx_begin sdp quiet
    cvx_solver mosek
    variable Rxx(Lxu_max,Lxu_max, U) hermitian semidefinite
    expressions Ru(U) En(U) tmp(Ly, Ly)
    tmp = eye(Ly);
    for v = 1:U
        Hv = H(:,index_start(order(v)):index_end(order(v)));
        rxxv = Rxx(1:Lxu(order(v)),1:Lxu(order(v)),order(v));
        tmp = tmp + Hv*rxxv*Hv';
        Ru(v) = 0.5*log_det(tmp);
        En(order(v)) = trace(rxxv);
    end
    minimize (w'*En - Ru'*pos(D*stheta))
cvx_end

Rxxs = cell(U,1);
tmp = eye(Ly);
for v = 1:U
    Hv = H(:,index_start(order(v)):index_end(order(v)));
    rxxv = Rxx(1:Lxu(order(v)),1:Lxu(order(v)),order(v));
    tmp = tmp + Hv*rxxv*Hv';
    Ru(v) = 0.5*real(log(det(tmp)));
    Rxxs{order(v)} = rxxv;
    En(order(v)) = trace(rxxv);
end
bn = zeros(U,1);
bn(order) = diff([0;Ru]);
f = -cvx_optval;

end
```

and

startEllipse_var_Lxu.m

```
% function [A, g] = startEllipse_var_Lxu(H, Lxu, bu_min, w)
%
% This function initializes the ellipsoid method for power minimization
% problem for MAC, with variable number of transmit antennas at each user.
% Called by minPMAC_new program
%
% function [A, g] = startEllipse_var_Lxu(H, Lxu, bu_min, w)
% H containes the channel matrices as is defined in minPMAC.m, Lxu is
% either a scalar or a length-U vector indicating number of transmit
% antennas of each user. bu_min is the target U by 1 rate vector, w is the
% U by 1 energy weights.
% For all fixed margin WF steps, decoding order is 1,2,...U.
```

```
%
% A is the matrix describing the staring ellipsoid and g is its center
% ********************************************************************
function [A, g] = startEllipse_var_Lxu(H, Lxu, bu_min, w)


[Ly, ~, N] = size(H);
U = length(w);
if length(Lxu) == 1
    Lxu = ones(1,U)*Lxu;
end
index_end = cumsum(Lxu);
index_start = [1,index_end(1:end-1)+1];

order = 1:U;                        % decoding order, arbitrary
tmax = zeros(U,1);

for u = 1:U
    Rxxs = cell(U,N);
    for u1 = 1:U
        for tone = 1:N
            Rxxs{u1,tone} = zeros(Lxu(u1));
        end
    end
    b = bu_min;
    b(u) = b(u) + 1;
    en = zeros(U, N);
    for u1 = order
        M = cell(1,N);
        gs = zeros(Lxu(u1),N);
        for tone = 1:N
            Rnoise = eye(Ly) +...
                H(:,[1:index_start(u1)-1,index_end(u1)+1:end],tone)...
                *blkdiag(Rxxs{[1:u1-1,u1+1:U], tone})...
                *H(:,[1:index_start(u1)-1,index_end(u1)+1:end],tone)';
            [~,h_tmp,M{tone}] = svd(Rnoise^(-1/2)*H(:,index_start(u1):index_end(u1),tone),0);
            gs(1:length(diag(h_tmp)),tone) = diag(h_tmp).^2;
        end
        [~, e_tmp] = fmwaterfill_gn(reshape(gs,1,[]), b(u1)/N, 0);
        en_tmp = reshape(e_tmp, Lxu(u1), N);
        en(u1,:) = sum(en_tmp, 1);
        for tone = 1:N
            Rxxs{u1,tone} = M{tone}*diag(en_tmp(:,tone))*M{tone}';
        end
    end

    tmax(u) = w'* sum(en,2);
end

g = tmax / 2;
A = g'*g*eye(U);        % an sphere centerced at tmax/2
```

This reuses fmwaterfill_gn , but revises minPtone to minPtoneMIMO and startEllipse to startEllipse_var_Lxu

minPtoneMIMO_cvx.m

```
% function [f, bn, Rxxs, En] = minPtoneMIMO(H, Lxu, theta, w)
% minPtone_cvx_var_Lxu minimizes f = - sum_{u=1}^U theta_u * b_u +
% sum_{u=1}^U w_u * e_u
% subject to b \in C_g(H,e). This program uses cvx package and mosek solver
%
% the inputs are:
% 1)  H, an Ly by Lx channel matrix. Ly is the number of receiver antennas,
%     Lx is the total number of transmit antennas.
%     H(:,index_start(u):index_end(u)) is the channel for user u.
% 2)  Lxu, a scalar or a length-U vector containing the number of transmit
%     antennas of each user. If Lxu is a scalar, each user has Lxu antennas
% 3)  theta, a U by 1 vector containing the weights for the rates.
% 4)  w, a U by 1 vector containing the weights for each user's energy.
%
% the outputs are:
% 1)  f, -1 * minimum value (or maximum value of the -1 * function).
% 2)  bn, a U by 1 vector containing the rates for each user
%     that optimizes the given function.
% 3)  Rxxs, a 1 by U cell array containing the Rxx's for each user that
%     optimizes the given function
% 4)  En, a U by 1 vector containing the energy of each user
function [f, bn, Rxxs, En] = minPtoneMIMO(H, Lxu, theta, w)
[Ly, ~] = size(H);
U = length(theta);
if length(Lxu) == 1
    Lxu = ones(1,U)*Lxu;
end
Lxu_max = max(Lxu);
index_end = cumsum(Lxu);
index_start = [1,index_end(1:end-1)+1];

[stheta, order] = sort(theta, 'descend');
D = eye(U)+diag(-ones(U-1,1),1);
max_bound = max(-diff([stheta;0]));
if max_bound > 1e4
    stheta = stheta/max_bound*1e2;
    w = w/max_bound*1e2;
end
cvx_begin sdp quiet
    cvx_solver mosek
    variable Rxx(Lxu_max,Lxu_max, U) hermitian semidefinite
    expressions Ru(U) En(U) tmp(Ly, Ly)
    tmp = eye(Ly);
    for v = 1:U
        Hv = H(:,index_start(order(v)):index_end(order(v)));
        rxxv = Rxx(1:Lxu(order(v)),1:Lxu(order(v)),order(v));
        tmp = tmp + Hv*rxxv*Hv';
        Ru(v) = 0.5*log_det(tmp);
        En(order(v)) = trace(rxxv);
    end
    minimize (w'*En - Ru'*pos(D*stheta))
cvx_end
```

```
Rxxs = cell(U,1);
tmp = eye(Ly);
for v = 1:U
    Hv = H(:,index_start(order(v)):index_end(order(v)));
    rxxv = Rxx(1:Lxu(order(v)),1:Lxu(order(v)),order(v));
    tmp = tmp + Hv*rxxv*Hv';
    Ru(v) = 0.5*real(log(det(tmp)));
    Rxxs{order(v)} = rxxv;
    En(order(v)) = trace(rxxv);
end
bn = zeros(U,1);
bn(order) = diff([0;Ru]);
f = -cvx_optval;

end
```

subparagraphstartEllipse_var_Lxu.m

```
% function [A, g] = startEllipse_var_Lxu(H, Lxu, bu_min, w)
%
% This function initializes the ellipsoid method for power minimization
% problem for MAC, with variable number of transmit antennas at each user.
% Called by minPMAC_new program
%
% function [A, g] = startEllipse_var_Lxu(H, Lxu, bu_min, w)
% H containes the channel matrices as is defined in minPMAC.m, Lxu is
% either a scalar or a length-U vector indicating number of transmit
% antennas of each user. bu_min is the target U by 1 rate vector, w is the
% U by 1 energy weights.
% For all fixed margin WF steps, decoding order is 1,2,...U.
%
% A is the matrix describing the staring ellipsoid and g is its center
% ********************************************************************
function [A, g] = startEllipse_var_Lxu(H, Lxu, bu_min, w)


[Ly, ~, N] = size(H);
U = length(w);
if length(Lxu) == 1
    Lxu = ones(1,U)*Lxu;
end
index_end = cumsum(Lxu);
index_start = [1,index_end(1:end-1)+1];

order = 1:U;                        % decoding order, arbitrary
tmax = zeros(U,1);

for u = 1:U
    Rxxs = cell(U,N);
    for u1 = 1:U
        for tone = 1:N
            Rxxs{u1,tone} = zeros(Lxu(u1));
        end
    end
    b = bu_min;
```

```
    b(u) = b(u) + 1;
    en = zeros(U, N);
    for u1 = order
        M = cell(1,N);
        gs = zeros(Lxu(u1),N);
        for tone = 1:N
            Rnoise = eye(Ly) +...
                H(:,[1:index_start(u1)-1,index_end(u1)+1:end],tone)...
                *blkdiag(Rxxs{[1:u1-1,u1+1:U], tone})...
                *H(:,[1:index_start(u1)-1,index_end(u1)+1:end],tone)';
            [~,h_tmp,M{tone}] = svd(Rnoise^(-1/2)*H(:,index_start(u1):index_end(u1),tone),0);
            gs(1:length(diag(h_tmp)),tone) = diag(h_tmp).^2;
        end
        [~, e_tmp] = fmwaterfill_gn(reshape(gs,1,[]), b(u1)/N, 0);
        en_tmp = reshape(e_tmp, Lxu(u1), N);
        en(u1,:) = sum(en_tmp, 1);
        for tone = 1:N
            Rxxs{u1,tone} = M{tone}*diag(en_tmp(:,tone))*M{tone}';
        end
    end

    tmax(u) = w'* sum(en,2);
end

g = tmax / 2;
A = g'*g*eye(U);        % an sphere centerced at tmax/2
```

admMAC.m

```
% function [FEAS_FLAG, bu_a, info] = admMAC_cvx(H, Lxu, bu, Eu, cb)
%
% admMAC_rate_region determines whether the target rate vector bu is
% feasible for (noise-whitened) channel H and energy/symbol Eu via rate region
%
%   Input arguments:
%       - H: Ly-by-Lx-by-N channel matrix. H(:,:,n) denotes the channel for
%           the n-th tone.
%       - Lxu: number of transmit antennas of each user. It can be either a
%           scalar or a length-U vector. If it is a scalar, every user has
%           Lxu transmit antennas; otherwise user u has Lxu(u) transmit
%           antennas.
%       - bu: target rate of each user, length-U vector.
%       - Eu: Energy/symbol on each user, length-U vector.
%   Outputs:
%       - FEAS_FLAG: indicator of achievability. FEAS_FLAG=0 if the target
%           is not achievable; FEAS_FLAG=1 if the target is achievable by a
%           single ordering; FEAS_FLAG=2 if the target is achievable by
%           time-sharing
%       - bu_a: U-by-1 vector showing achieved sum rate of each user.
%       - info: various length output depending on FEAS_FLAG
%           --if FEAS_FLAG=0: empty
%           --if FEAS_FLAG=1: 1-by-5 cell array containing
%               {Rxxs, Eun, bun, theta, w} corresponds to the single vertex
%           --if FEAS_FLAG=2: v-by-7 cell array, with each row representing
```

```
%               a time-shared vertex {Rxxs, Eun, bun, theta, w, frac, cluster}
%
%      info's row entries in detail (one row for each vertex shared
%       - Rxxs: U-by-N cell array containing Rxx(u,n)'s if Lxu is a
%           length-U vector; or Lxu-by-Lxu-by-U-by-N tensor if Lxu is a
%           scalar. If the rate target is infeasible, output 0.
%       - Eun:  U-by-N matrix showing users' transmit energy on each tone.
%           If infeasible, output 0.
%       - bun: U-by-N matrix showing users' rate on each tone. If
%           infeasible, output 0.
%       - theta: U-by-1 Lagrangian multiplier w.r.t. target rates
%       - w: U-by-1 Lagrangian multiplier w.r.t. energy constraints
%       - order: U-by-1 user order
%       - frac: fraction of dimensions for each vertex in time share (FF =2
%           ONLY), per cluster
%       -- cluster 1 has largest common-theta, cluster 2 has second largest
%          common-theta group, and so on
%
%      Subroutines called
%       - maxRMAC_cvx
%       - maxRMACMIMO
%*************************************************************************
function [FEAS_FLAG, bu_a, info] = admMAC(H, Lxu, bu, Eu, cb)
tstart=tic;
[Ly, ~, N] = size(H);
Eu = reshape(Eu,[],1);
bu = reshape(bu,[],1);
%Rxxs = cell(1,N);
U = length(Eu);
bu_a=zeros(1,U);
theta = ones(U,1);
A = eye(U)*U;
count = 1;
% error tolerance
err = 1e-6;
% tolerance of convex hull boundary
conv_tol = 1e-6;
FEAS_FLAG=0;


if max(Lxu) == 1
    SCALAR_FLAG = 1;
    [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta, cb);
    Rxxstemp=cell(1,N);
    for tone=1:N
        Rxxstemp{tone} = num2cell(Eun(:,tone));
    end
else
    SCALAR_FLAG = 0;
    [Rxxs, Eun, w, bun] = maxRMACMIMO(H, Lxu, Eu, theta, cb);
    Rxxstemp=Rxxs;
end

% achieved users' rates
```

```
bu_v = sum(bun,2)';
g = bu_v' - bu;

if theta'*g < -err % infeasible criterion
    FEAS_FLAG = 0;
    bu_a = 0;
    info=[];
    return
end

if sqrt(g'*A*g) <= err || min(g) >= 0 % achievable by current vertex
    FEAS_FLAG = 1;
    bu_a = bu_v';
    info = table(bu_v, Rxxstemp, {Eun}, {bun}, {theta}, {w}); % detailed info of boundary vertices
info.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta' 'w'};
    return
end

% add the first vertex
known_vertices = table(bu_v, Rxxstemp, {Eun}, {bun}, {theta}, {w}); % detailed info of boundary vert
known_vertices.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta' 'w'};
bd_vertices = bu_v; % track critical boundary vertices
bd_V = 1; % track number of critical boundary vertices
vertices = de2bi(0:2^U-1).*bu_v; % all boundary vertices
tess = convhulln(vertices);

while 1
    count = count + 1;
    bu_min=bu;

    % update ellipsoid
    tmp = A*g/sqrt(g'*A*g);
    theta = theta - 1/(U + 1)*tmp;
    A = U^2/(U^2 - 1) * (A - 2/(U + 1)*(tmp*tmp'));
    ind = find(theta < zeros(U,1));
    while ~isempty(ind) % helps numerically, as thetas > 0
        g = zeros(U,1);
        g(ind(1)) = -1;
        tmp = A*g/sqrt(g'*A*g);
        theta = theta - 1/(U + 1)*tmp;
        A = U^2/(U^2 - 1) * (A - 2/(U + 1)*(tmp*tmp'));
        ind = find(theta < zeros(U,1));
    end

    % ------------------------ next vertex ----------
    % this next vertex must use the new theta's order

    % get the next vertex
    if SCALAR_FLAG
        [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta, cb);
        Rxxs=Eun;
        Rxxstemp=cell(1,N);
        for tone=1:N
            Rxxstemp{tone} = num2cell(Eun(:,tone));
```

```
        end
    else
%       [Rxxs, Eun, w, bun] = maxRMAC_vector_cvx(H, Lxu, Eu_now, theta , cb);
        [Rxxs, Eun, w, bun] = maxRMACMIMO(H, Lxu, Eu_now, theta , cb);
        Rxxstemp=Rxxs;
    end
    bu_v=sum(bun,2)';
    g = bu_v' - bu;
    bu_a=bu_v;

    if theta'*g < -err % infeasible criteria
        FEAS_FLAG = 0;
        bu_a = bu_v';
        info = {};
        return
    end

%     if sqrt(g'*A*g) <= err || min(g) >= 0 % achievable by current vertex
    if min(g) >= 0 % achievable by current vertex
        FEAS_FLAG = 1;
        bu_a = bu_v';

        info = table(bu_v, Rxxstemp, {Eun}, {bun}, {theta}, {w}); % detailed info of boundary vertic
        info.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta' 'w'};
        return
    end

    %---------------------TWO OR MORE EQUAL-THETA USERS ------------------
    % This point of while loop is only reached if there are equal-theta users.
    % This section then vertex shares for the equal-theta clusters.
    %
    % Only equal-theta users vertices' rate subsets are evaluated.
    % sizeset is the size of a cluster = # of equal-theta users
    % There is no need for U! orders to be searched (which is usually a much
    % larger number, just the sizeclus within each cluster, while sizeset+1 is
    % the total number of orders that are searched (over all clusters).
    %
    % order and invorder change with respect to Itheta and Jtheta ONLY within
    % the clusters so that different vertices can be shared to get the correct
    % data rate bu_min (bu input) instead of just sum(bu_min)
    % ------------------------------------------------------------------

    % form clusters with equal-theta users and check if each cluster's sum(bu_min) achieves the targ
    flagadd = 1; % whether the current vertex / cluster of vertex should be added
    [stheta , Itheta] = sort(theta, 'descend');
    [~, Jtheta]=sort(Itheta);
    delta = -diff([stheta;0]);
    spots = delta < max(err, 1e-2*norm(theta));
    spots(end) = 0;
%     spots = delta < 1e-2*norm(stheta); % whether a user (sorted by theta) is in the same cluster a



    sizeclus = []; % size of each cluster
```

```
numclus = 0; % number of clusters (>1 equal-theta groups)
set_thetaeq = []; % index of the first entry of each cluster

flagclus = 0;
for i = 1:U
    if spots(i) && ~flagclus % create a new set
        numclus = numclus + 1;
        set_thetaeq = [set_thetaeq, i];
        sizeclus = [sizeclus, 1];
        flagclus = 1;
    elseif spots(i) % add user to an existing set
        sizeclus(end) = sizeclus(end) + 1;
    elseif flagclus % last user to add
        sizeclus(end) = sizeclus(end) + 1;
        flagclus = 0;
        us = Itheta(set_thetaeq(end):set_thetaeq(end)+sizeclus(end)-1);
        if sum(bu_v(us) - bu(us)') < -err*length(us)
            flagadd = 0;
            break
        end
    else % user not in a cluster
        flagclus = 0;
        if bu_v(Itheta(i)) - bu(Itheta(i)) < -err
            flagadd = 0;
            break
        end
    end
end

if ~flagadd % continue to next iteration with update theta
    continue
end

% --------- REORDER USERS ACCORDING TO THETA -------
% This ordering will need eventually to be reversed before returing from
% this function.
% theta = theta(Jtheta) reverses this sort later.
bu_a = bu_a(Itheta);
bu_v = bu_v(Itheta);
bun = bun(Itheta,:);
Eun = Eun(Itheta,:);
bu_min = bu_min(Itheta);
index_end = cumsum(Lxu);
index_start = [1, index_end(1:end-1)+1];
hidx = [];
for i = Itheta'
    hidx = [hidx, index_start(i):index_end(i)];
end
Lxu = Lxu(Itheta);
index_end = cumsum(Lxu);
index_start = [1, index_end(1:end-1)+1];
H = H(:,hidx,:);
for tone = 1:N
    Rxxstemp{tone} = Rxxstemp{tone}(Itheta);
```

```
    end

    % ----------- CREATE INFO TABLE & ADD FIRST VERTEX ----------
    initialbu_v = bu_v; % save this value as it is needed if more than 1 cluster
    initialbun = bun;
    firstvertices = de2bi(0:2^U-1).*initialbu_v;

    bu_a=bu_v;
    initialinfo = table(bu_v, Rxxstemp, {Eun}, {bun}, {theta}, {U:-1:1}); % detailed info of boundar
    initialinfo.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta' 'order'};

%     info = table(bu_v, Rxxstemp, {Eun}, {bun}, {theta}, {U:-1:1}); % detailed info of boundary ver
%     info.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta', 'order'};

    % format the clusters and permuted orders
    order=repmat(1:U,sum(sizeclus-1),1);
    cumsize = cumsum([0,sizeclus-1]);
    for jdx = 1:numclus
        u_range = set_thetaeq(jdx):set_thetaeq(jdx)+sizeclus(jdx)-1; % users in cluster jdx
        for i = 1:sizeclus(jdx)-1
            order(cumsize(jdx)+i, u_range) = circshift(u_range, i); % all permutated orders (excludi
        end
    end

    % ----- For each cluster -----------
    for jdx=1:numclus
        FEAS_FLAG = 0;
        if jdx == 1
            Sinit = repmat(eye(Ly),1,1,N);
            cumrateinit = zeros(1, N);
            for n = 1:N
                for i = 1:set_thetaeq(1)-1
                    Sinit(:,:,n) = Sinit(:,:,n) + H(:,index_start(i):index_end(i),n)*Rxxstemp{n}{i}*
                end
                cumrateinit(n) = (1/cb)*real(log2(det(Sinit(:,:,n))));
            end

        else
            for n = 1:N
                for i = set_thetaeq(jdx-1):set_thetaeq(jdx-1)+sizeclus(jdx)-1
                    Sinit(:,:,n) = Sinit(:,:,n) + H(:,index_start(i):index_end(i),n)*Rxxstemp{n}{i}*
                end
                cumrateinit(n) = (1/cb)*real(log2(det(Sinit(:,:,n))));
            end
        end
        u_range = set_thetaeq(jdx):set_thetaeq(jdx)+sizeclus(jdx)-1;
        bu_v=initialbu_v;
        known_vertices = initialinfo; % detailed info of boundary vertices
        bd_vertices = bu_v(u_range);    % track critical boundary vertices
        vertices = de2bi(0:2^sizeclus(jdx)-1).*bd_vertices;
        if min(bd_vertices - bu_min(u_range)') >= -conv_tol % achievable w/o time share
            info = known_vertices;
            info.frac = 1;
            info.clusterID = jdx;
```

```
    if jdx == 1
        Big_info = info;
    else
        Big_info = [Big_info; info];
    end
    continue;
end
bd_V = 1;
for idx = cumsize(jdx)+1:cumsize(jdx)+sizeclus(jdx)-1 % add more vertices in cluster jdx
    cumrate = zeros(sizeclus(jdx)+1, N);
    cumrate(1,:) = cumrateinit;
    for n = 1:N
        rel_idx = 2;
        S = Sinit(:,:,n);
        for u = u_range
            u_or = order(jdx, u);
            S = S + H(:,index_start(u_or):index_end(u_or),n)*Rxxstemp{n}{u_or}*H(:,index_sta
            cumrate(rel_idx,n) = (1/cb)*real(log2(det(S)));
            rel_idx = rel_idx + 1;
        end
    end
    bun = initialbun;
    bun(u_range,:) = diff(cumrate);
    bun(order(idx,:),:) = bun;
    bu_v = sum(bun, 2)';

    bd_V = bd_V + 1;
    bd_vertices_extend = [bd_vertices; bu_v(u_range)];
    known_vertices = [known_vertices; {bu_v, Rxxstemp, {Eun}, {bun}, {theta}, {order(idx,end
    vertices = [vertices; de2bi(0:2^sizeclus(jdx)-1).*bu_v(u_range)];
    tess = convhulln(vertices);
    vertices = vertices(unique(tess),:);
    bd_vertices = intersect(bd_vertices_extend, vertices, 'rows', 'stable');
    tess = convhulln(vertices);
    % delete inner vertices
    if size(bd_vertices, 1) < bd_V
        to_remove = setdiff(bd_vertices_extend, bd_vertices, 'rows');
        known_vertices(ismember(known_vertices.bu_v(u_range), to_remove, 'rows'),:) = [];
        bd_V = size(bd_vertices, 1);
    end

    if inhull(bu_min(u_range)', vertices, tess, conv_tol) % bu_min achievable by time-share
        FEAS_FLAG = 2;
        frac = bd_vertices'\bu_min(u_range); % TODO: change this
        a_v = find(frac >= err); % active vertices in time-share
        info = known_vertices(a_v, :);
        info.frac = frac(a_v);
        info.frac = info.frac/sum(info.frac);
        info.clusterID = ones(length(a_v), 1)*jdx;
        bu_a(u_range) = bd_vertices(a_v,:)'*info.frac;
        break;
    end
end
if FEAS_FLAG == 0
```

```
            break;
        end

        % write big info table
        if jdx == 1
            Big_info = info;
        else
            Big_info = [Big_info; info];
        end
    end
    if FEAS_FLAG ~= 0
        break;
    end
end


info = Big_info;
bd_V = size(info.bu_v, 1);

% ----------- RESTORE ORIGINAL ORDER TO ALL ----------------
theta = theta(Jtheta);
temptheta = reshape(cell2mat(info.theta), U, bd_V);
temptheta = temptheta(Jtheta,:);
info.theta= mat2cell(temptheta', [ones(1,bd_V)] , U);
%
tempbun=reshape(cell2mat(info.bun),U,bd_V,N);
tempbun=tempbun(Jtheta,:,:);
tempbun=permute(tempbun,[2 1 3]);
info.bun=mat2cell(tempbun,[ones(1,bd_V)], U , N);
%
tempEun=reshape(cell2mat(info.Eun),U,bd_V,N);
tempEun=tempEun(Jtheta,:,:);
tempEun=permute(tempEun,[2 1 3]);
info.Eun=mat2cell(tempEun,[ones(1,bd_V)], U , N);
%
bu_a = bu_a(Jtheta);
info.bu_v=info.bu_v(:,Jtheta);
%
tmporder = cell2mat(info.order);
tmporder = Itheta(tmporder);
info.order = mat2cell(tmporder, ones(1,bd_V), U);

for tone=1:N
    Rxxstemp{tone}= Rxxstemp{tone}(Jtheta);
end
for u=1:U
    for n=1:N
        info.Rxxs{n}{u}=Rxxstemp{n}{u};
    end
end
toc(tstart)
end
```

```
% function [FEAS_FLAG, bu_a, info] = admMAC_rate_region(H, Lxu, bu, Eu)
%
% admMAC_rate_region determines whether the target rate vector bu is
% feasible for (noise-whitened) channel H and energy/symbol Eu via rate region
%
%   Input arguments:
%       - H: Ly-by-Lx-by-N channel matrix. H(:,:,n) denotes the channel for
%           the n-th tone.
%       - Lxu: number of transmit antennas of each user. It can be either a
%           scalar or a length-U vector. If it is a scalar, every user has
%           Lxu transmit antennas; otherwise user u has Lxu(u) transmit
%           antennas.
%       - bu: target rate of each user, length-U vector.
%       - Eu: Energy/symbol on each user, length-U vector.
%   Outputs:
%       - FEAS_FLAG: indicator of achievability. FEAS_FLAG=0 if the target
%           is not achievable; FEAS_FLAG=1 if the target is achievable by a
%           single ordering; FEAS_FLAG=2 if the target is achievable by
%           time-sharing
%       - bu_a: U-by-1 vector showing achieved sum rate of each user.
%       - info: various length output depending on FEAS_FLAG
%           --if FEAS_FLAG=0: empty
%           --if FEAS_FLAG=1: 1-by-5 cell array containing
%               {Rxxs, Eun, bun, theta, w} corresponds to the single vertex
%           --if FEAS_FLAG=2: v-by-6 cell array, with each row representing
%               a time-shared vertex {fraction, Rxxs, Eun, bun, theta, w}
%
%       - Rxxs: U-by-N cell array containing Rxx(u,n)'s if Lxu is a
%           length-U vector; or Lxu-by-Lxu-by-U-by-N tensor if Lxu is a
%           scalar. If the rate target is infeasible, output 0.
%       - Eun:  U-by-N matrix showing users' transmit power on each tone.
%           If infeasible, output 0.
%       - bun: U-by-N matrix showing users' rate on each tone. If
%           infeasible, output 0.
%       - theta: U-by-1 Lagrangian multiplier w.r.t. target rates
%       - w: U-by-1 Lagrangian multiplier w.r.t. power constraints
function [FEAS_FLAG, bu_a, info] = admMAC_rate_region(H, Lxu, bu, Eu)
[Ly, ~, N] = size(H);
if N == 1
    H = reshape(H,Ly,[],1);
end
Eu = reshape(Eu,[],1);
bu = reshape(bu,[],1);
U = length(Eu);
theta = ones(U,1);
A = eye(U)*U;
count = 1
% error tolerance
err = 1e-6;
% tolerance of convex hull boundary
conv_tol = 1e-6;


if max(Lxu) == 1
```

```
    SCALAR_FLAG = 1;
    [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta);
    Rxxs = Eun;
else
    SCALAR_FLAG = 0;
    [Rxxs, Eun, w, bun] = maxRMAC_vector_cvx(H, Lxu, Eu, theta);
end

% achieved users' rates
bu_v = sum(bun,2)';
g = bu_v' - bu;

if theta'*g < -err % infeasible criterion
    FEAS_FLAG = 0;
    bu_a = bu_v';
    info=[];
    return
end

if sqrt(g'*A*g) <= err || min(g) >= 0 % achievable by current vertex
    FEAS_FLAG = 1;
    bu_a = bu_v';
    info = table(bu_v, {Rxxs}, {Eun}, {bun}, {theta}, {w}); % detailed info of boundary vertices
info.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta' 'w'};
    return
end

% add the first vertex
known_vertices = table(bu_v, {Rxxs}, {Eun}, {bun}, {theta}, {w}); % detailed info of boundary vertic
known_vertices.Properties.VariableNames(2:end) = {'Rxxs' 'Eun' 'bun' 'theta' 'w'};
bd_vertices = bu_v; % track critical boundary vertices
bd_V = 1; % track number of critical boundary vertices
vertices = de2bi(0:2^U-1).*bu_v; % all boundary vertices
tess = convhulln(vertices);

while 1
    count = count + 1

    % update ellipsoid
    tmp = A*g/sqrt(g'*A*g);
    theta = theta - 1/(U + 1)*tmp;
    A = U^2/(U^2 - 1) * (A - 2/(U + 1)*(tmp*tmp'));
    ind = find(theta < zeros(U,1));
    while ~isempty(ind)
        g = zeros(U,1);
        g(ind(1)) = -1;
        tmp = A*g/sqrt(g'*A*g);
        theta = theta - 1/(U + 1)*tmp;
        A = U^2/(U^2 - 1) * (A - 2/(U + 1)*(tmp*tmp'));
        ind = find(theta < zeros(U,1));
    end

    % get the next vertex
    if SCALAR_FLAG
```

```
    [Eun, w, bun] = maxRMAC_cvx(H, Eu, theta);
    Rxxs = Eun;
else
    [Rxxs, Eun, w, bun] = maxRMAC_vector_cvx(H, Lxu, Eu, theta);
end
% achieved users' rates
bu_v = sum(bun,2)';
g = bu_v' - bu;

if theta'*g < -err % infeasible criteria
    FEAS_FLAG = 0;
    bu_a = bu_v';
    info = {};
    break
end

if sqrt(g'*A*g) <= err || min(g) >= 0 % achievable by current vertex
    FEAS_FLAG = 1;
    bu_a = bu_v';
    info = {Rxxs, Eun, bun, theta, w};
    break
end

if ~inhull(bu_v, vertices, tess, conv_tol)
    % add new vertices and build new convex hull
    known_vertices = [known_vertices; {bu_v, {Rxxs}, {Eun}, {bun}, {theta}, {w}}];
    bd_vertices_extend = [bd_vertices; bu_v];
    bd_V = bd_V + 1;
    new_vs = de2bi(0:2^U-1).*bu_v;
    vertices = [vertices;new_vs];
    tess = convhulln(vertices);
    % keep only the vertices at the boundary of convex hull
    vertices = vertices(unique(tess),:);
    bd_vertices = intersect(bd_vertices_extend, vertices, 'rows');
    tess = convhulln(vertices);
    % delete inner vertices
    if size(bd_vertices, 1) < bd_V
        to_remove = setdiff(bd_vertices_extend, bd_vertices, 'rows');
        known_vertices(ismember(known_vertices.bu_v, to_remove, 'rows'),:) = [];
        bd_V = size(bd_vertices, 1);
    end
    if inhull(bu', vertices, tess, conv_tol) % achievable by time-share
        FEAS_FLAG = 2;
        cvx_begin quiet  % compute time-share combination
            variable frac(bd_V) nonnegative
            minimize norm(frac,1)
            subject to
                sum(frac) <= 1;
                bd_vertices'*frac == bu;
        cvx_end
        active_idx = find(frac>=err);
        a_v = bd_vertices(active_idx,:);
        a_frac = frac(active_idx);
        [tab_pos,frac_loc] = ismember(known_vertices.bu_v, a_v,'rows');
```

```
            info = known_vertices(tab_pos,:);
            info.frac = a_frac(frac_loc);
            bu_a = bu;
            break
        end
    end
end
```

G.5.3 Section 5.5 Duality Programs and BC programs

mac2bc.m

```
% function Rxxb = mac2bc(Rxxm, Hmac, N)
%
% This function converts U users' MAC autcorrelation matrices to dual BC
% autocorrelation matrices. These output matrices achieve the same set of
% rates in the dual BC by a lossless precoder. The user should order the
% inputs Rxxm and Hmac (see below) so that they have the desired order
% already.
%
% Inputs
% Rxxm - an Lx x Lx x U array of users autocorrelation matrices
%        corresponding to [U:-1:1]=order
% Hmac - the Ly x Lx x U MAC channel for with users listed U...1, from left
%
% Output
% Rxxb  - Ly  Ly x U dual BC's output autocorrelation matrix, which has the
%         left user in order reversed (order *J) at top/left in Rxxb.
%
% ------------------------------------------------------------------------

function Rxxb = mac2bc(Rxxm, Hmac)

[Ly, Lx, U] = size(Hmac);

% These two following lines just set index u to same as textbook
% because matlab has reversed order with respect to textbook
%
Hmac = Hmac(:,:,[U:-1:1]); %reversing to matlab's order
Rxxm = Rxxm(:,:,[U:-1:1]); %reversing to matlab's order
%J=hankel([zeros(Ly-1,1) ; 1],[1 zeros(1,Ly-1)])
%
% the following algorithm uses u like in the textbook
Rxxbtot = zeros(Ly,Ly);

% Gtot is the transmit covariance matrix of the BC

B = zeros(Ly,Ly,U);
A = zeros(Lx,Lx,U);

% A and B matrices are the Rtildenoise for BC and MAC respectively


B(:,:,U) = eye(Ly);
```

```
for k = U:-1:2
   B(:,:,k-1) = B(:,:,k) + Hmac(:,:,k)*Rxxm(:,:,k)*Hmac(:,:,k)';
end

A(:,:,1) = eye(Lx);

for k = 1:U
   temp_A = inv(sqrtm(A(:,:,k))); %sqrtm provides transpose of text's matrix sq root
   temp_B = inv(sqrtm(B(:,:,k)));

   [F L M] = svd(temp_B' * Hmac(:,:,k) * temp_A', 'econ');
   Rxxb(:,:,k) = temp_B' * F * M' * sqrtm(A(:,:,k)) * Rxxm(:,:,k) * sqrtm(A(:,:,k))' * M * F' * temp
   %Rxxb(:,:,k) = temp_B * F * M' * sqrtm(A(:,:,k))' * Rxxm(:,:,k) * sqrtm(A(:,:,k)) * M * F' * temp

   Rxxbtot = Rxxbtot + Rxxb(:,:,k);

   if k~=U
       % the Jy matrix is not necessary because Rxxbtot is already in that
       % order corresponding to this reversal in text
      A(:,:,k+1) = eye(Lx) + Hmac(:,:,k+1)'* Rxxbtot * Hmac(:,:,k+1);
   end
end
```

bc2mac.m

```
% function Rxxm = bc2mac(Rxxb, Hmac)
%
% This function converts the BC autocorrelation matrices set in Rxxb to the
% corresponding set of autocorrelation matrices to Rxxm of a dual MAC.
%
% The command Hbc=conj(permute(Hmac(:,:,end:-1:1),[2 1 3])), where
% generates the Hbc if needed for any reason.
%
% U = number of users, which is obtained from the 3rd dimension of the input
% tensor Hmac.  Lx and Ly correspond to this (dual) MAC that has Ly receive
% antennas and Lx transmit antennas per user.  Variable Lxu (so Lyu on Rxxb)
% needs to find maximum over u, and extend all by zero columns in Hmac and
% zeroed rows and columns in Rxxb.
%
% The output Rxxm user order is reversed restored) by this program.
%
% Inputs
%
% Rxxb is an Ly by Ly by U array containing the BC input Rxx(u) matrices.
%   where Rxxb(:,:,u) is the autocorrelation matrix for BC user u.
%   The overall autocorrelation matrix Rxxbsum is sum(Rxxb,3), but not
%   output.
%
% Hmac is an Ly x Lx x U MAC channel matrix that for which the dual BC is
% computed internally. Hmacu(:,:,u) is user u's noise-whitened
%   equivalent channel matrix.
%
% Output
%
```

```
% Rxxm contains the MAC autocorrelation matrices, which are Ly x Ly x U.
%    Rxxm(:,:,u) is user u's aucorrelation matrix, reversed in order from
%    the meaning of u in the input Rxxb.
%
% ------------------------------------------------------------------
function Rxxm = bc2mac(Rxxb, Hmac)

[Ly, ~, U] = size(Hmac); % temp,1,2 help in debugging, could be ~
%[Ly, temp1 , temp2 ]=size(Rxxb);
H=conj( permute( Hmac(:,:,end:-1:1) , [2 1  3] )  ); % computes dual BC
[Lx,~,~]=size(H);

% Rxxbsum is the transmit covariance matrix of the BC

B = zeros(Ly,Ly,U); % MAC's noise plus crosstalk
A = zeros(Lx,Lx,U); % BC's noise plus crosstalk
Rxxm = zeros(Ly,Ly,U);
Rxxbsum=zeros(Lx,Lx);

% construct the BC autocorrelation matrices cummulative sums
A(:,:,1) = eye(Lx);
for u = 1:U-1
   Rxxbsum=Rxxbsum+Rxxb(:,:,u);
   A(:,:,u+1) = eye(Lx) + H(:,:,u+1)*Rxxbsum*H(:,:,u+1)';
end

B(:,:,U) = eye(Ly);
Hmac=Hmac(:,:,U:-1:1); % align Hmac order with matlab

for u = U:-1:1;
   temp_A = pinv(sqrtm(A(:,:,u))); % sqrtm provides congugate transpose of text's square root
   temp_B = pinv(sqrtm(B(:,:,u)));

   [F L M] = svd(temp_B' * Hmac(:,:,u) * temp_A', 'econ'); %Hcheck inside svd

   Rxxm(:,:,u) = temp_A * M * F' * sqrtm(B(:,:,u))' * Rxxb(:,:,u) * sqrtm(B(:,:,u)) * F * M' * temp_
  % Rxxm(:,:,u) = temp_A * M * F' * sqrtm(B(:,:,u)) * Rxxb(:,:,u) * sqrtm(B(:,:,u)) * F * M' * temp_
   if u~=1
      B(:,:,u-1) = B(:,:,u) + Hmac(:,:,u)* Rxxm(:,:,u) * Hmac(:,:,u)';
   end
end
Rxxm(:,:,U:-1:1) = Rxxm; % restoring user U at top
```

Maximum E-sum MAC sum rate program macmax.m

```
% function [Rxx, bsum , bsum_lin] = macmax(Esum, h, Lxu, N , cb)
%
% Simultaneous water-filling Esum MAC max rate sum (linear & nonlinear GDFE)
% The input is space-time domain h, and the user can specify a temporal
% block symbol size N (essentially an FFT size).
%
% This program uses the CVX package
%
% the inputs are:
```

```
% Esum The sum-user energy/SAMPLE scalar in time-domain.
%      This will be increased by cb*N by this program.
%      Each user energy should be scaled by N/(N+nu)if there is cyclic prefix
%      This energy is the trace of the corresponding user Rxx (u)
%      The sum energy is computed as the sum of the Eu components
%      internally.
% h    The TIME-DOMAIN Ly x sum(Lx(u)) x N channel for all users
% Lxu  The number of antennas for each user 1 x U
% N    The FFT size (equally spaced over (0,1/T) at 1/(NT).
% cb   cb = 1 for complex, cb=2 for real baseband
%
% the outputs are:
% Rxx A block-diagonal psd matrix with the input autocorrelation for each
%     user on each tone. Rxx has size (sum(Lx(u)) x sum(Lx(u)) x N .
%     sum trace(Rxx) over tones and spatial dimensions equal the Eu
%     FREQUENCY DOMAIN
% bsum the maximum rate sum, when cb=2, this is effectively over lower half
% of tones, or equivalently the 1/2*log2 form of data rates are summed
% bsum  bsum_lin - the maximum sum rate with a linear receiver
%
%     b is an internal convergence (vector, rms) value, but not sum rate

function [Rxx, bsum, bsum_lin] = macmax(Esum, h, Lxu, N , cb)

H = fft(h, N, 3); % scaled so that N factor no longer needed in rate calculation
[Ly, Lx, ~] = size(H);
Esum=cb*N*Esum;
if numel(Lxu) > 1
    U = numel(Lxu);
    if sum(Lxu)~=Lx
        error('mismatch between sum of Lxu and Lx');
    end
elseif (Lx/Lxu)~= floor(Lx/Lxu)
    error('invalid Lxu');
else
    U = Lx/Lxu;
    Lxu = Lxu*ones(1,U);
end

idx_end = cumsum(Lxu);
idx_start = [1, idx_end(1:end-1)+1];
idx_exp_end = N*idx_end;
idx_exp_start = [1, idx_exp_end(1:end-1)+1];

Hcell=mat2cell(H, Ly, Lxu, ones(1,N));
Hexpand = zeros(N*Ly,N*Lx);
for u = 1:U
    Hexpand(:,idx_exp_start(u):idx_exp_end(u)) = blkdiag(Hcell{1,u,:});
end

Lxu_max = max(Lxu);
cvx_begin quiet
cvx_solver mosek
    variable Rxxun(Lxu_max,Lxu_max,N,U) hermitian semidefinite
```

```
    Rxxun_cell = reshape(num2cell(Rxxun, [1,2,3]), 1, U);
    for u = 1:U
        Rxxun_cell{u} = Rxxun_cell{u}(1:Lxu(u), 1:Lxu(u),:);
        Rxxun_cell{u} = reshape(num2cell(Rxxun_cell{u},[1,2]), 1, N);
    end
    Rxxun_cell = [Rxxun_cell{:}];
    Rxx = blkdiag(Rxxun_cell{:});
    maximize (1/cb*log2(exp(1))*log_det(eye(N*Ly) + Hexpand*Rxx*Hexpand'))
    subject to
        trace(Rxx) <= Esum;
cvx_end

bsum=cvx_optval;

Rxxun_cell = reshape(num2cell(Rxxun, [1,2,3]), U, 1);
for u = 1:U
    Rxxun_cell{u} = Rxxun_cell{u}(1:Lxu(u), 1:Lxu(u),:);
    Rxxun_cell{u} = reshape(num2cell(Rxxun_cell{u},[1,2]), 1, N);
end
Rxxun_cell = vertcat(Rxxun_cell{:}); % U*N cell array

Rxx = zeros(Lx,Lx,N);
for n=1:N
    Rxx(:,:,n) = blkdiag(Rxxun_cell{:,n});
end

bs=zeros(1,U);
bsum_lin=0;
for u=1:U
    indices=idx_start(u):idx_end(u);
    for n=1:N
     bs(u)=bs(u)+(1/cb)*(log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)')) - log2(det(eye(Ly)+H(:,:,n)*Rxx(:, ...
       :,n)*H(:,:,n)'- H(:,indices,n)*Rxx(indices, ...
       indices,n)*H(:,indices,n)')));
    end
    bsum_lin=bsum_lin+real(bs(u));
end
end
```

```
% function [Rwcn, bsum] = wcnoise(Rxx, H, Ly, dual_gap, nerr)
%
% inputs
% H is U*Ly by Lx, where
%       Ly is the number of antennas/receiver,
%       Lx is the number of transmit antennas, and
%       U is the number of users.
% Rxx is the Lx by Lx input autocorrelation matrix.
% dual_gap is the duality gap, defaulting to 1e-6 in wcnoise
% nerr is Newton's method acceptable error, defaulting to 1e-4 in wcnoise
%
% outputs
```

```
% Rwcn is the U*Ly by U*Ly worst-case-noise autocorrelation matrix.
% bsum is the rate-sum/real-dimension.
%
% I = 0.5 * log(det(H*Rxx*H'+Rwcn)/det(Rwcn)), Rwcn has Ly x Ly diagonal blocks
%     that are each equal to an identity matrix
%
function [Rwcn, bsum] = wcnoise(Rxx, H, Ly, dual_gap, nerr)

switch nargin
    case 3
        dual_gap = 1e-6;
        nerr = 1e-4;
    case 4
        nerr = 1e-4;
end

[n,m] = size(H);
K = n / Ly;

A = zeros(n,n,n*(n+1)/2);
for i = 1:n
   A(i,i,i) = 1;
end

count = n+1;

for i = 2:n
   for j = 1:i - 1
      A(i,j,count) = 1;
      A(j,i,count) = 1;
      count = count+1;
   end
end

map = zeros(n,n);
for i = 1:K
   map((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = ones(Ly,Ly);
end




NT_max_it = 1000;                              % Maximum number of Newton's
                                               % method iterations

%dual_gap = 1e-6;
mu = 10;                                       % step size for t
alpha = 0.001;                                 % back tracking line search parameters
beta = 0.5;

count = 1;
%n%nerr = 1e-4;                                   % acceptable error for inner loop
                                               % Newton's method


v_0 = zeros(n*(n+1)/2,1);                      % Strictly feasible point;
```

```
v_0(1:n) = 0.5 *  ones(n,1);
v = v_0;
t =1;
l_v = 1;                                        % lambda(v) for newton's method termination



while (1+n)/t > dual_gap
   t =  t * mu;
   l_v = 1;
   count = 1;
   while l_v > nerr & count < NT_max_it

      f_val = 0;                                % calculating function value
      Rwcn = zeros(n,n);
      Rzprime = zeros(n,n);

      for i = 1:n*(n+1)/2                       % computing Rz
         Rwcn = Rwcn + v(i) * A(:,:,i);
      end

      for i = 1:K
          Rzprime((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = Rwcn((i-1) * Ly + 1:i * Ly,(i-1) *
      end

      f_val = t * log(det(H * Rxx * H' + Rwcn)) - (t + 1) * log(det(Rwcn)) - log(det(eye(n) - Rzprim

      S = inv(H * Rxx * H' + Rwcn);
      Q = inv(eye(n) - Rzprime);
      Rz_inv = inv(Rwcn);
      g = zeros(n*(n+1)/2,1);
      h = zeros(n*(n+1)/2,n*(n+1)/2);

      for i = 1:n*(n+1)/2
         g(i) = t * trace(A(:,:,i) * S) - (t + 1) * trace(A(:,:,i) * Rz_inv)...
              + (sum(sum(A(:,:,i) .* map)) ~= 0) * trace(A(:,:,i) * Q) ;  % gradient
      end


      for i = 1:n*(n+1)/2
         for j = 1:n*(n+1)/2
            h(i,j) = -t * trace(A(:,:,i) * S * A(:,:,j) * S) + (t + 1) * trace(A(:,:,i) * Rz_inv * A
                     +(sum(sum(A(:,:,i) .* map)) ~= 0) * (sum(sum(A(:,:,j) .* map)) ~= 0) * trace(A(
         end
      end


      dv = -h\g;                                % search direction


      s = 1;                                    % checking v = v+s*dx feasible
                                                % and also back tracking algorithm

      v_new = v + s * dv;
```

```
f_new = 0;

Rz_new = zeros(n,n);

for i = 1:n*(n+1)/2
   Rz_new = Rz_new + v_new(i) * A(:,:,i);
end

for i = 1:K
    Rzprime((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = Rz_new((i-1) * Ly + 1:i * Ly,(i-1)
end

f_new = t * log(det(H * Rxx * H' + Rz_new)) - (t + 1) * log(det(Rz_new)) - log(det(eye(n) - Rz



feas_check = 1;
if real(eig(Rz_new)) > zeros(n,1)
    feas_check = 1;
else
    feas_check = 0;
end

if real(eig(eye(n) - Rzprime)) > zeros(n,1)
    feas_check = 1;
else
    feas_check = 0;
end

feas_check = feas_check * (f_new < f_val + alpha * s * g' * dv);


while feas_check ~= 1
   s = s * beta ;

   v_new = v + s * dv;
   f_new = 0;

   Rz_new = zeros(n,n);

   for i = 1:n*(n+1)/2
      Rz_new = Rz_new + v_new(i) * A(:,:,i);
   end

   for i = 1:K
       Rzprime((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = Rz_new((i-1) * Ly + 1:i * Ly,(i-
   end

   f_new = t * log(det(H * Rxx * H' + Rz_new)) - (t + 1) * log(det(Rz_new)) - log(det(eye(n) -



   feas_check = 1;
```

```
        if real(eig(Rz_new)) > zeros(n,1)
            feas_check = 1;
        else
            feas_check = 0;
        end
        if real(eig(eye(n) - Rzprime)) > zeros(n,1)
            feas_check = 1;
        else
            feas_check = 0;
        end


        feas_check = feas_check * (f_new < f_val + alpha * s * g' * dv);



    end

    v = v + s * dv;                          % update v
    l_v = -g'*dv ;                            % lambda(v)^2 for Newton's method
    count = count + 1;                       % number of Newtons method iterations
  end

end

Rwcn = zeros(n,n);

for i = 1:n*(n+1)/2
  Rwcn = Rwcn + v(i) * A(:,:,i);
end

for i = 1:K
   Rwcn((i-1) * Ly + 1:i * Ly,(i-1) * Ly + 1:i * Ly) = eye(Ly);
end
bsum = 0.5 * log2(det(H * Rxx * H' + Rwcn)/det(Rwcn));
```

cvx_wcnoise.m

```
function [Rnn, sumRatebar] = cvx_wcnoise(Rxx, H, Lyu)
%cvx_wcnoise This function computes the worst-case noise for a given input
%autocorrelation Rxx and channel matrix.
%   Arguments:
%       - Rxx: input autocorrelation, size(Lx, Lx)
%       - H: channel response, size (Ly, Lx)
%       - Lyu: number of antennas at each user, scalar/vector of length U
%   Outputs:
%       - Rnn: worst-case noise autocorrelation, with white local noise
%       - sumRatebar: maximum sum rate/real-dimension

eps_margin = 1e-4;
[Ly,Lx] = size(H);
if length(Lyu) == 1
    U = Ly/Lyu;
    Lyus = ones(1,U)*Lyu;
```

```
else
    U = length(Lyu);
    Lyus = Lyu;
end
cum_Lyu = cumsum(Lyus);
cum_Lyu = [0, cum_Lyu(1:end-1)];
us = 1:U;

Htilde = H*sqrtm(Rxx);
cvx_begin sdp quiet
% cvx_solver mosek
variable Rnn(Ly, Ly) semidefinite
variable Z(Lx, Lx) semidefinite
maximize log_det(eye(Lx) - Z)
subject to
for u = us
    Rnn(cum_Lyu(u) + (1:Lyus(u)), cum_Lyu(u) + (1:Lyus(u))) == eye(Lyus(u));
end
Rnn - eps_margin*eye(Ly) == semidefinite(Ly);
[Rnn + Htilde*Htilde', Htilde; Htilde', Z] == semidefinite(Ly+Lx);
cvx_end

sumRatebar = -0.5*log2(exp(1))*cvx_optval;
```

bcmax.m

```
% function [Rxx, Rwcn, bmax] = bcmax(iRxx, H, Lyu)
%
%  Uses cvx_wcnoise.m and rate-adaptive waterfill.m (Lagrange
%  Multiplier based)
%   Arguments:
%       - iRxx: initial input autocorrelation array, size is Lx x Lx x N.
%           Only the sum of traces matters, so can initialize to any valid
%           autocorrelation matrix Rxx to run wcnoise.
%           needs to include factor N/(N+nu) if nu ~= 0
%       - H: channel response, size is Ly x Lx x N, w/o sqrt(N)
%       normalization
%       - Lyu: number of antennas at each user
%             can create variable-u by just using dummy zero rows in H for
%             all output receivers that have less than max (=Lyu input)
%   Outputs:
%       - Rxx: optimized input autocorrelation
%       - Rwcn: optimized worst-case noise autocorrelation, with white local noise
%               SO IF H is noise-whitened for Rnn, then actual noise is
%               Rwcn^(1/2)*Rnn*Rwcn^(*/2)
%       - b: maximum sum rate/real-dimension - user must mult by 2 for
%            complex case

function [Rxx, Rwcn, bmin] = bcmax(iRxx, H, Lyu)

    [Ly,Lx,N] = size(H);
    total_en = trace(sum(iRxx, 3));
    bmin = 0;
```

```
    ib=zeros(1,N);
    Rwcn = zeros(Ly,Ly,N);
    for n=1:N % worst-case noise independent over tones for BC
    [Rwcn(:,:,n), ib(n)] = cvx_wcnoise(iRxx(:,:,n), H(:,:,n), Lyu);
    end

    while (abs(sum(ib) - bmin) > 1e-5) %tolerance
        % uncomment the following two lines to see how the loop progresses
        %bmax
        bmin = sum(ib);
     % Vector Coding Gains for each tone
        M=zeros(Lx,Lx,N);
        gains = zeros(Lx,N);
        for n=1:N
            [V,D,~]=svd(Rwcn(:,:,n));
            sqD = sqrt(D).*(D>1e-6);
            invsqRwcn = V*pinv(sqD)*V';
            Htil = invsqRwcn*H(:,:,n);
            [~, g, M(:,:,n)] = svd(Htil);
            g=diag(g);
            gains(1:length(g),n)=g.^2;
        end
     % Water-filling step
        En = waterfill(total_en, reshape(gains',N*Lx,1), 1);
        %bvec=0.5*log2(ones(N*Lx,1)+En.*reshape(gains',N*Lx,1))';
        %bmax = real(sum(bvec));
        En=reshape(En,N,Lx)';
     % update worst-case-noise step
        for n=1:N
            newiRxx = M(:,:,n)*diag(En(:,n))*M(:,:,n)';
            iRxx(:,:,n) = (iRxx(:,:,n)+newiRxx)/2;

            %iRxx(:,:,n) = M(:,:,n)*diag(En(:,n))*M(:,:,n)';
           [Rwcn(:,:,n), ib(:,n)] = cvx_wcnoise(iRxx(:,:,n), H(:,:,n), Lyu);
        ib(:,n)=real(ib(:,n));
        end
    end
    Rxx = iRxx;
    bmin = sum(ib);
end
```

G.5.4 From Subsection 5.6 matlab programs

osb.m and related program listings

```
% function [S1, S2, b1, b2] = osb(Hmag_sq, No, E, theta, mask, ...
% gap, bitcap, cb)
%
% This is the main program call to osb.  It sets up calculation of w1
% and calls a subroutine that in turn sets up w2, which in turn calls
% another routine that does the main crosstalk calculations.
%
% Inputs
%
% Hmag_sq is a N x 2 x 2 where N is FFT size. N inferred from this.
```

```
% No      is a 1 x U white-noise power spectra density matrix.
%    If Hmag_sq is complex BB, then No should be the one-sided PSD.
% E       is a 1 x U energy vector.
% theta   is a 1 x U user-rate weighting vector.
% mask    is an N x U PSD maximum allowed.
% gap     is the (non-dB) linear gap (so 1 if 0 dB gap).
% bitcap  is a 1 x U maximum number of bits allowed per tone.
% cb      is 2 for real baseband and 1 for cplex bband
%
% Outputs
%
% S1      is user 1's Nx1 PSD
% S2      is user 2's Nx1 PSD
% b1      is user 1's Nx1 bit distribution
% b2      is user 2's Nx1 bit distribution
%
% calls optimize_l2.m, which calls optimize_s.m
% ------------------------------------------------------------

function [S1, S2, b1, b2] = osb(Hmag_sq, No, E, theta, mask, gap, bitcap,cb)

N = size(Hmag_sq,1);

b1 = zeros(N,1); b2 = zeros(N,1);
S1 = zeros(N,1); S2 = zeros(N,1);

w_1_l = 0; % lower limit on Lagrange for energy weight
w_1_u = 1; % upper limit on Lagrange for energy weight
%
% update OSB, get new upper limit in next call
[S1, S2, b1, b2] = optimize_l2(Hmag_sq, No, E(2), theta, w_1_u, mask, gap, bitcap, cb);

while sum(S1) > E(1) % too much energy
    w_1_u = 2*w_1_u; % reduce upper limmit
    [S1, S2, b1, b2] = optimize_l2(Hmag_sq, No, E(2), theta, w_1_u, mask, gap, bitcap, cb);
end

err = 1e-1;
% err = 1;
while (w_1_u - w_1_l)>err | sum(S1)>E(1) % too much energy
    lambda_1 = (w_1_u + w_1_l)/2;
    [S1, S2, b1, b2] = optimize_l2(Hmag_sq, No, E(2), theta, lambda_1, mask, gap, bitcap,cb);
    if (sum(S1) - E(1)) >= 0
         w_1_l = lambda_1; % increase lower limit
    else
         w_1_u = lambda_1; % decrease upper limit
    end
end

% function [S1, S2, b1, b2] = optimize_l2(Hmag_sq, No, E2, theta, ...
%     lambda_1, mask, gap, bitcap,cb)
%
% OSB finds weight w2 for USER 2
% A. Chowdhery ~2010 ; Updated by J. Cioffi in 2024.  It presently handles
```

```
% only 2 users, so U=2.  E(2) is held constant, while E(1) varies.
%
% Inputs
%
% Hmag_sq is a N x 2 x 2 where N is FFT size. N inferred from this.
% No      is a 1 x U white-noise power spectra density matrix.
%    If Hmag_sq is complex BB, then No should be the one-sided PSD.
% E2      is user 2's energy.
% theta   is a 1 x U user-rate weighting vector.
% w_1     is user 1's energy weight.
% mask    is an N x U PSD maximum allowed.
% gap     is the (non-dB) linear gap (so 1 if 0 dB gap).
% bitcap  is a 1 x U maximum number of bits allowed per tone.
% cb      is 2 for real baseband and 1 for cplex bband
%
% Outputs
%
% S1      is user 1's Nx1 PSD
% S2      is user 2's Nx1 PSD
% b1      is user 1's Nx1 bit distribution
% b2      is user 2's Nx1 bit distribution
%
% ------------------------------------------------------------

function [S1, S2, b1, b2] = optimize_l2(Hmag_sq, No, E2, theta, lambda_1, mask, gap, bitcap,cb)

N = size(Hmag_sq,1);

b1 = zeros(N,1); b2 = zeros(N,1);
S1 = zeros(N,1); S2 = zeros(N,1);

w_2_l = 0; % lower limit on Lagrange for energy constraint
w_2_u = 1e1; % upper limit on Lagrange for energy constraint

[S1, S2, b1, b2] = optimize_s(Hmag_sq, No, theta, lambda_1, w_2_u, mask, gap, bitcap,cb);

while sum(S2) > E2 % too much energy on user 2
    w_2_u = 2 * w_2_u;
    [S1, S2, b1, b2] = optimize_s(Hmag_sq, No, theta, lambda_1, w_2_u, mask, gap, bitcap,cb);
end

err = 1e-1;
while (w_2_u - w_2_l)>err | sum(S2)>E2
    lambda_2 = (w_2_u + w_2_l)/2;

    [S1, S2, b1, b2] = optimize_s(Hmag_sq, No, theta, lambda_1, lambda_2, mask, gap, bitcap, cb);

    if (sum(S2) - E2) >= 0
         w_2_l = lambda_2; % increase lower limit
    else
         w_2_u = lambda_2; % decrease upper limit
    end
end
```

```
% function [S1, S2, b1, b2] = optimize_s(Hmag_sq, No, w, ...
%          lambda_1, lambda_2, mask, gap, bitcap, cb)
%
% OSB tonal Lagrangian tests values 0:bitcap for users 1 and 2.
% A. Chowdhery ~2010 ; Updated by J. Cioffi in 2024. This is the OSB
% computational subroutine. Again, original by A. Chowdhery
% and ppdated by J. Cioffi in 2024.  This program presently handles
% only 2 users, so U=2.
%
% Inputs
%
% Hmag_sq is a N x 2 x 2 where N is FFT size. N inferred from this.
% No      is a 1 x U white-noise power spectra density matrix.
%    If Hmag_sq is complex BB, then No should be the one-sided PSD.
% E2      is user 2's energy.
% theta   is a 1 x U user-rate weighting vector.
% w_1      is user 1's energy weighting factor.
% w_2      is user 2's energy weighting factor.
% mask    is an N x U PSD maximum allowed.
% gap     is the (non-dB) linear gap (so 1 if 0 dB gap).
% bitcap  is a 1 x U maximum number of bits allowed per tone.
% cb      is 2 for real baseband and 1 for cplex bband
%
% Outputs
%
% S1      is user 1's Nx1 PSD
% S2      is user 2's Nx1 PSD
% b1      is user 1's Nx1 bit distribution
% b2      is user 2's Nx1 bit distribution
%
% ------------------------------------------------------------
function [S1, S2, b1, b2] = optimize_s(Hmag_sq, No, theta, w_1, w_2, mask, gap, bitcap, cb)

N = size(Hmag_sq,1);
S1 = zeros(N,1); S2 = zeros(N,1);
b1 = zeros(N,1); b2 = zeros(N,1);

for n = 1:N
    max_value = -inf;
    for i1 = 0:bitcap(1)
        for i2 = 0:bitcap(2)
            % next 3 lines essentially invert a matrix to find the psds
            D = Hmag_sq(n,1,1)*Hmag_sq(n,2,2)-gap^2*(2^i1-1)*(2^i2-1)*Hmag_sq(n,1,2)*Hmag_sq(n,2,1);
            N1 = gap*(2^i1-1)*(gap*(2^i2-1)*Hmag_sq(n,2,1)*No(n,2)+Hmag_sq(n,2,2)*No(n,1));
            N2 = gap*(2^i2-1)*(gap*(2^i1-1)*Hmag_sq(n,1,2)*No(n,1)+Hmag_sq(n,1,1)*No(n,2));
            s(1) = N1/D; s(2) = N2/D;
            Lk = theta*i1 + (1-theta)*i2 - w_1*s(1) - w_2*s(2);
            if Lk > max_value & s(1)<=mask(n,1) & s(2)<=mask(n,2) & s(1)>=0 & s(2)>=0
                max_value = Lk;
                b1(n) = i1/cb; b2(n) = i2/cb;
                S1(n) = s(1); S2(n) = s(2);
            end
        end
    end
```

```
end

%fprintf('w = %3.2f, l1 = %5.1e, l2 = %5.1e, b1 = %4d, b2 = %4d, E1 = %5.4f, E2 = %5.4f \n',...
 %   theta, lambda_1, lambda_2, sum(b1), sum(b2), sum(S1), sum(S2) );
%drawnow;

return;
```

wci.m - Worst Case IC interference with distributed management - M. Brady

```
function [Rate Y X int_profile]=wci(H,P,Sigma,Gap)
%function [Rate Y X int_profile]=wci(H,P,Sigma,Gap)
%Compute the worst-case interference for IC user 1
%Inputs: H is a UxUxN matrix of channel gains
%           where H(m,p,n) is channel from user p into user m on tone n
%        P is a Ux1 vector of power constraints.
%        Sigma is a Nx1 vector of AWGN noises (per tone noise)
%        Gap is the Gap-to-Capacity in LINEAR scale (not dB)
%Outputs:   Rate is the guaranteeable rate under WCI
%           Y is the worst-interference inducing power allocations
%           X is the victim modem response to the worst-interference
%           int_profile is WCI interference profile
%Restrictions:     *H of user 1 should not be zero for all tones
%                  *H must have at least 2 users
%                  *P must have each element strictly positive
%rev mhbrady 9/23/05

%Test the input dimensionality
Rate = 0;
Y = 0;
X = 0;
int_profile = 0;

TOL = 1e-20;  % Channel gains less than this are treated as 0

try
    TONESRAW = size(H,3);

    if size(Sigma) ~= [TONESRAW 1]
        disp('Invalid input dimensions for Sigma')
        return
    end
    if or(prod(size(Gap)) ~= 1,Gap < 1)
        disp('Invalid Gap')
        return
    end
    if or(size(P,1) ~= size(H,1), size(P,2) ~= 1)
        disp('Invalid input dimensions of P')
        return
    end

    if size(H,1)~=size(H,2)
        disp('Invalid input dimensions. H must be square (per tone)')
        return
```

```
    end


catch
    disp('Invalid input dimensions, or not enough inputs.')
    return
end


%Get power gains
Hsq = H .^ 2;
%Initialize interference structure
A = zeros(size(H,1),size(H,3));
Hsq_new = zeros(size(H,1),size(H,2));
Sigma_new = [];
%Extract TONESRAW where victim has gain of < TOL
pos = 1;
ontones = [];
for n=1:TONESRAW
    if(H(1,1,n) >= TOL)
        Hsq_new(:,:,pos) = Hsq(:,:,n);
        Sigma_new(pos,1) = Sigma(n);
        ontones = [ontones n];
        pos = pos+1;
    end
end

TONES = size(Hsq_new,3);

%Do normalization
normlz = Hsq_new(1,1,:);
normlz = reshape(normlz,prod(size(normlz)),1);
for intr=2:size(H,1)
    Htemp = Hsq_new(1,intr,:);
    Htemp = reshape(Htemp, prod(size(Htemp)),1);
    A(intr,:) = Gap*(Htemp ./ normlz)' ;
end

Sigma_norm = Gap*(Sigma_new ./ normlz)';

CAP = 2*repmat(P,1,TONES);
ABSTOL = .000001;

Atilde{1} = A;
for u=2:size(H,1)
    Atilde{u} = 0*A;
end

w = zeros(size(H,1),1);
w(1) = 1;

Sigma_norm2 = [Sigma_norm'...
    repmat(Sigma_norm',1,size(H,1)-1)];
```

```
[Xback, Yback, LB_cp]=linw(Atilde,Sigma_norm2',P,CAP,w',ABSTOL,0);

Rate = LB_cp;
Yb = Yback(2:size(Yback,1),:)';
Xb = Xback(1,:)';

%Do the writeback
int_profile2 = (normlz' / Gap) .* (sum(A .* Yback,1)+Sigma_norm);
int_profile2 = int_profile2';
int_profile = zeros(size(H,3),1);
int_profile(ontones) = int_profile2;

Y = zeros(size(H,3),size(H,1)-1);
X = zeros(size(H,3),1);

Y(ontones,:) = Yb;
X(ontones,:) = Xb;
```

G.6 Matlab Programs for spectra calculations in Chapter 6

G.6.1 Rectangular (REC) CPM Spectra Programs

There are 4 programs used:

1. Phase_nREC computes the phase pulse response for a rectangular frequency pulse response. (Since the integral is trivial, no Freq program is provided.)
2. PSum_nREC is a program that computes the product-and-sum term that is an intermediate result in Appendix A's spectrum calculations. This function is typically integrated by another function (Matlab "handle") call and so separated to assist debugging and keep program run time bounded.
3. R_nREC computes the autocorrelation function for positive autocorrelation lags by integration of the function in PSum_nREC up to $(\nu + 1)T$. The function repeats and scales by the constant $C(h, M)$ in a geometric progression for all subsequent symbol intervals.
4. S_nREC computes the desired power spectral density from R_nREC exploiting the geometric progression as well as using the FFT function to approximate a continuous integral.

These programs' source-code listings appear in order here.

```
function phase = Phase_nREC(t,T,n)
%
%       J. Cioffi - November 2017
%   Phase_nREC computes the phase-pulse-response value at time t for nu-REC
%      t is time index
%      n=nu is the number of non-0 symbols in the frequency response
%      T is the symbol period.
%
%      nREC should be linearly growing from 0 to 1/2nT, holding at 1/2 t>nT
%      phase is the output phase

l=size(t);
j=l(1);
l=l(2);
for i1=1:j
for i2=1:l
if t(i1,i2) < 0
    phase(i1,i2) = 0;
elseif t(i1,i2) > n*T
    phase(i1,i2) = 1/2;
else
    phase(i1,i2)=(1/(2*n*T))*t(i1,i2);
end
end
end

% ---------------------------------------

function sump = PSum_nREC(t,tau,T,n,h,M,m)

%       J. Cioffi - November 2017 (used in computing autocorrelation)
%
%  PSum_nREC computes the sum of exponents in the Aulin/Sundberg formula
%  for nu-REC continuous phase modulation.  PSum_nREC calls Phase_nREC above.
%
```

```
% sump is the sum of exponents term in the autocorrelation integral
%      t is time index
%      tau is autocorrelation lag over the interval 0 to T (really tauprime)
%      n=nu is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      indexj is the product index
%      m+1 is the upper index on the product (outer function that uses this one)
%           tau = tauprime + mT
%

lt=size(t);
l=lt(2);
sumptemp=zeros(m+1+n,l);
sump=zeros(1,l);
for index=1:m+1+n % index is offset by n=nu from the formula
    sumptemp(index,:)=(1/M)*sum(exp(kron(i*2*pi*h*([-M+1:2:M-1]'),Phase_nREC(t+(tau-(index-n-m)*T)*o
          - Phase_nREC(t-(index-n)*T*ones(1,l),T,n))),1);
end
temp=prod(sumptemp([1:m+1+n],:),1);
for indexb=1:l
        sump(indexb)=real(temp(1,indexb));
end

% ------------------------------------------------

function autoR = R_nREC(tau,T,n,h,M)
%
%       J. Cioffi - November 2017
%
%   R_nREC computes the autocorrelation function evalualted for any
%   positive time arg
%   R_nREC calls Psum_nREC through the integral function of matlab
%      tau is time lag (which is positive)
%      tauprime is an internal variable representing the fractional value
%      of tau (tau = tauprime + mT), which must be non-negative
%      n is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%
%     autoR is the output for lag tau - calls PSum_nREC

tauprime=tau-floor((1/T)*tau);
m=floor((1/T)*tau);
l=size(tau);
l=l(2);
autoR=zeros(1,l);
for index=1:l
autoR(index)=(1/T)*integral(@(t) PSum_nREC(t,tauprime(index),T,n,h,M,m(index)),0,T);
end

% -------------------------------------
```

```
function spec = S_nREC(T,n,h,M)
%
%           J. Cioffi - November 2017
%
%   S_nREC computes the power spectrum for REC autocorrelation
%   To avoid multiple nested integral-function calls, the FFT is used
%   instead
%       The exponential decay of the function in nT to nT+1 for all subsequent
%       symbol periods is used to compute those contributions
%      n is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      this program assumes that sampling interval is .01*T
%      this program computes 10k point FFT, so frequency range is 0:100/T
%     spec is the output

%   an FFT is used instead of a continuous integral to compute spectrum
%      nREC is called to compute the first portion of auto
%      correlation
%
lr1=(n)*T*100;
lr2=T*100;
R=R_nREC([0:(lr1+lr2)]*.01*T,T,n,h,M);
R1=R(1:lr1+1);
R2=R(lr1+2:lr1+lr2+1);
temp =[R1 R2];
for indexa = 2:50-n
    r([1:100])=R2*(C_alpha(h,M)^(indexa-1));
    temp=[temp , r([1:100])];
end
R=[temp, temp(5000:-1:2)];
spec=.01*fft(R,10000);
```

G.6.2 Raised-Cosine (RC) CPM Spectra Programs

There are 4 programs used:

1. Phase_nRC computes the phase pulse response for a raised-cosine pulse response. (Since the integral is trivial, no Freq program is provided.)

2. PSum_nRC is a program that computes the product-and-sum term that is an intermediate result in Appendix A's spectrum calculations. This function is typically integrated by another function (Matlab "handle") call and so separated to assist debugging and keep program run time bounded.

3. R_nRC computes the autocorrelation function for positive autocorrelation lags by integration of the function in PSum_nRC up to $(\nu+1)T$. The function repeats and scales by the constant $C_\alpha(M)$ in a geometric progression for all subsequent symbol intervals.

4. S_nRC computes the desired power spectral density from R_nRC exploiting the geometric progression as well as using the FFT function to approximate a continuous integral.

These RC programs' source-code listings appear in order here.

```
function phase = Phase_nRC(t,T,n)
%
% Phase_nRC compuotes the phase-pulse-response value at time t for nRC
%      t is time index
%      n is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      phase is the output.
%
%      Phase should be raised cosine from 0 to 1/2nT, holding at 1/2 t>nT
%      phase is the output phase

l=size(t);
l=l(2);
phase=zeros(1,l);
for index=1:l
if t(index) < 0
    phase(index) = 0;
elseif t(index) > n*T
    phase(index) = 1/2;
else
    phase(index)=(1/(2*n*T)*t(index))-(1/(4*pi))*sin(2*pi*t(index)/(n*T));
end
end

% ---------------------------------------

function sump = PSum_nRC(t,tau,T,n,h,M,m)
% Psum_nRC computes the sum of exponents term in the spectrum integral for nRC
%      t is time index
%      tau is offset over the interval 0 to nT
%      n=nu is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      indexj is the product index
%      m+1 is the upper index on the product (outer function that uses this one)
%   Psum_nRC calls Phase_nRC
%     sump is the output

lt=size(t);
l=lt(2);
sumptemp=zeros(m+1+n,l);
sump=zeros(1,l);
for index=1:m+1+n
    sumptemp(index,:)=(1/M)*sum(exp(kron(i*2*pi*h*([-M+1:2:M-1]'),Phase_nRC(t+(tau -(index-n-m)*T)*o
        - Phase_nRC(t-(index-n)*T*ones(1,l),T,n))),1);
end
temp=prod(sumptemp([1:m+1+n],:),1);
for indexb=1:l
        sump(indexb)=real(temp(1,indexb));
end

function autoR = R_nRC(tau,T,n,h,M)
```

```
%   R_nRC computes the RC autocorrelation function evalualted for any positive time arg
%      tau is time lag (which is positive)
%      tauprime is an internal variable representing the fractional value
%      of tau (tau = tauprime + mT), which must be non-negative
%      n is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%
%     autoR is the output for lag tau - this calls PSum_nRC

tauprime=tau-floor((1/T)*tau);
m=floor((1/T)*tau);
l=size(tau);
l=l(2);
autoR=zeros(1,l);
for index=1:l
autoR(index)=(1/T)*integral(@(t) PSum_nRC(t,tauprime(index),T,n,h,M,m(index)),0,T);
end

function spec = S_nRC(T,n,h,M)
% S_nREC computes the power spectrum for RC autocorrelation and calls R_nRC once.
%   To avoid multiple nested integral-function calls, the FFT is used
%   instead
%       The exponential decay of the function in nT to nT+1 for all subsequent
%       symbol periods is used to compute those contributions
%      n is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      this program assumes that sampling interval is .01*T
%      this program computes 10k point FFT, so frequency range is 0:100/T
%
%     spec is the power spectrum
lr1=(n)*T*100;
lr2=T*100;

R=R_nRC([0:(lr1+lr2)]*.01*T,T,n,h,M);
R1=R(1:lr1+1);
R2=R(lr1+2:lr1+lr2+1);
temp =[R1 R2];
for indexa = 2:50-n
    r([1:100])=R2*(C_alpha(h,M)^(indexa-1));
    temp=[temp , r([1:100])];
end
R=[temp, temp(5000:-1:2)];
spec=.01*fft(R,10000);
```

G.6.3 Spectrally Raised-Cosine (SRC) CPM Spectra Programs

There are 6 programs used:

1. Freq_nSRC computes the frequency pulse response for a spectrally raised-cosine. It's integration to a pulse response is non trivial.

- Causality is ensured by looking back 3ν symbol periods in time. This author found this necessary to get accurate results.

2. Phase_nSRC computes the phase pulse response via integration of Freq_nSRC.

3. Phase_nSRCinterp uses results from several computed and stored versions of calls to Phase_nSRC to create a continuous function in matlab that can be integrated but avoiding multiple embedded definite integrals in Matlab, which this author found to create long run times and be prone to numerical problems.

4. PSum_nRC is a program that computes the product-and-sum term that is an intermediate result in Appendix A's spectrum calculations. This function is typically integrated by another function (Matlab "handle") call and so separated to assist debugging and keep program run time bounded. In effect the function using the interpolated SRC call within will be integrated (once). This function extends the causality look back/forward to 6ν symbol periods, which the author found necessary for accurate results.

5. R_nSRC computes the autocorrelation function for positive autocorrelation lags by integration of the function in PSum_nSRC up to $(\nu + 1)T$, where ν includes the 6ν extension. The function repeats and scales by the constant $C_\alpha(M)$ in a geometric progression for all subsequent symbol intervals.

6. S_nSRC computes the desired power spectral density from R_nSRC exploiting the geometric progression as well as using the FFT function to approximate a continuous integral.

These SRC programs' source-code listings appear in order here.

```
function freq = Freq_nSRC(t,T,n,alpha)
% Freq_nSRC computes the TIME-DOMAIN freq-pulse-response
% value of spectrally raised cosine at time t
%      t is time index
%      n is the width of the main lobe
%      3n symbols delay is introduced to ensure causality (with small
%      truncation/windowing error)
%      T is the symbol period.
%      alpha is the roll-off factor (0 <= alpha <= 1)
%

l = size(t);
l=l(2);
t=t-(3*n)*T*ones(1,l);
plot(t)
freq=(1/(n*T))*sinc((2/(n*T))*t);
for k=1:l
    if t(k) == (n*T)/(4*alpha)
    freq(k) = freq(k)*pi*.25;
    elseif t(k) == -(n*T)/(4*alpha)
        freq(k) = freq(k)*pi*.25;
    elseif (t(k) < -3*n*T) | (t(k) > 3*n*T)
            freq(k)=0;
    else
    freq(k)=freq(k)*cos(((2*alpha*pi)/(n*T))*t(k))/(1-((16*alpha*alpha)/(n*n*T*T))*t(k)*t(k));
    end
end

function phase = Phase_nSRC(t,T,n,alpha)
```

```
%
%  Phase_nSRC computes the TIME-DOMAIN phase pulse response from the freq pulse response
%      tau is time variable input to the function Freq_nSRaiseCos
%      integrates freq-pulse in time from negative infinity to time t
%      t is the function input - phase is a function of t


l=size(t);
l=l(2);
phasearray=zeros(1,l);
for index=1:l
    phasearray(index)= integral(@(u)Freq_nSRC(u,T,n,alpha),-50*T,t(index));
end
phase=phasearray;

function phase = Phase_nSRCInterp(t,T,n,basephase)
%
% This function interpolates the phase pulse response from the samples in basephase
% basephase is a 501 point file of samples of the output of
%     Phase_nSRC, which itself looks back 3*n symbol periods
%     the call that generated basephase is then
%       basephase = Phase_nSRC([0:2000]*.1*T;
%     sampling is thus presumed at T/10 in basephase
%     basephase presumes an "n" so this function is called with
%     basephaseSRC(n,:), so basephaseSRC is typically 15 x 1001 array.
%
%     The value of THE SAME n is provided to PSum_nSRC so that 3*n is
%     implemented.
%  to generate basephase, generate the command below 15 times n=1:15
%  basephaseSRChalf(n,:)=Phase_nSRC(t,T,n,alpha); with alpha =0.5 for half
%  SEE COMMENTED CALLS BELOW
%  alpha is the excess bandwidth
% n is the number of symbol periods in the partial response
% T is the symbol period for basephase

l=size(t);
l=l(2);
phase=spline([0:1000]*.1,basephase,t);
low=floor(t*10/T);
index=zeros(1,l);
for k=1:l
%    index(k)=low(k)+40+n*10;
index(k)=low(k);
    if index(k) < 1
        phase(k)=0;
    elseif index(k) > 999 % 260+n*10
        phase(k)=0.5;
    end
end

function sump = PSum_nSRC(t,tau,T,n,h,M,m,basephase)
% phase is the sum of exponents term in the SRC spectrum integral
%      t is time index
%      tau is offset over the interval 0 to T, so really tauprime
```

```
%      n is the width of main spectrum lobe in symbol periods
%      SRC causality forces n--> 6*n rougly
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      indexj is the product index
%      m+1 is the upper index on the product (outer function that uses this one)
%
%     PSum_nSRC is an intermediate term in the spectrum calculation for
%     nSRC
%     sump is the output

lt=size(t);
l=lt(2);
sumptemp=zeros(m+1+6*n,l);
sump=zeros(1,l);
%n=n+10;
for index=1:m+1+6*n
    sumptemp(index,:)=(1/M)*sum(exp(kron(i*2*pi*h*([-M+1:2:M-1]'),Phase_nSRCInterp(t+(tau-(index-(6*
        - Phase_nSRCInterp(t-(index-(6*n))*T*ones(1,l),T,n,basephase))),1);
end
temp=prod(sumptemp([1:m+1+6*n],:),1);
for indexb=1:l
    sump(indexb)=real(temp(1,indexb));
end

function autoR = R_nSRC(tau,T,n,h,M,basephase)
%
%   R_nSRC computes the SRCautocorrelation function evalualted for any
%       positive time arg
%      tau is time lag (which is positive) and should not exceed (n+1)T
%      tauprime is an internal variable representing the fractional value
%      of tau (tau = tauprime + mT), which must be non-negative
%      n is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      alpha is roll-off factor (between 0 and 1)
%      h is the index of modulation
%      M is the number of messages
%   R_nSRC calls PSum_nSRC.
%
%     autoR is the output for lag tau

tauprime=tau-floor((1/T)*tau);
m=floor((1/T)*tau);
l=size(tau);
l=l(2);
autoR=zeros(1,l);
for index=1:l
   if tau(index) > (6*n)*T
      autoR(index)=0;
   else
autoR(index)=(1/T)*integral(@(t) PSum_nSRC(t,tauprime(index),T,n,h,M,m(index),basephase),0,T);
   end
end
```

```
function spec = S_nSRC(T,n,h,M,basephase)
%
% S_nSRC computes the power spectrum for REC autocorrelation
%   To avoid multiple nested integral-function calls, the FFT is used
%   instead
%       The exponential decay of the function in nT to nT+1 for all subsequent
%       symbol periods is used to compute those contributions
%       The functions called extend n to 6n to ensure near causality
%      n is the number of symbol non-0 symbols in the frequency response
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      this program assumes that sampling interval is .01*T
%      this program computes 10k point FFT, so frequency range is 0:100/T

%   an FFT is used instead of a continuous integral to compute spectrum
%      nREC is called to compute the successive portions of auto
%      correlation
lr1=(3*n)*T*100;
lr2=T*100;
R=R_nSRC([0:(lr1+lr2)]*.01*T,T,n,h,M,basephase);
R1=R(1:lr1+1);
R2=R(lr1+2:lr1+lr2+1);
temp =[R1 R2];
r([1:100])=R2;
for indexa = 2:50-(3*n)
    r([1:100])=r([1:100])*C_alpha(h,M);
    temp=[temp , r([1:100])];
end
R=[temp, temp(5000:-1:2)];
spec=.01*fft(R,10000);
```

G.6.4 Gaussian Minimum-Shift Keying (GMSK) CPM Spectra Programs

There are 6 programs used:

1. Freq_GMSK computes the frequency pulse response for a spectrally raised-cosine. It's integration to a pulse response is non trivial.
    - Causality is ensured by looking back $1/B$ symbol periods in time. This author found this necessary to get accurate results.
2. Phase_GMSK computes the phase pulse response via integration of Freq_GMSK.
3. Phase_GMSKinterp uses results from several computed and stored versions of calls to Phase_GMSK to create a continuous function in matlab that can be integrated but avoiding multiple embedded definite integrals in Matlab, which this author found to create long run times and be prone to numerical problems.
4. PSum_GMSK is a program that computes the product-and-sum term that is an intermediate result in Appendix A's spectrum calculations. This function is typically integrated by another function (Matlab "handle") call and so separated to assist debugging and keep program run time bounded. In effect the function using the interpolated GMSK call within will be integrated (once). This function extends the causality look back/forward to $1/B + 10$ symbol periods, which the author found necessary for accurate results.

5. R_GMSK computes the autocorrelation function for positive autocorrelation lags by integration of the function in PSum_GMSK up to $(\nu + 1)T$ wherre ν includes the $1/B + 10$ extention. The function repeats and scales by the constant $C_\alpha(M)$ in a geometric progression for all subsequent symbol intervals.

6. S_nGMSK computes the desired power spectral density from R_GMSK exploiting the geometric progression as well as using the FFT function to approximate a continuous integral.

These SRC programs' source-code listings appear in order here.

```
function freq = Freq_GMSK(t,T,B)
%    Freq_GMSK computes is the TIME-DOMAIN freq-pulse-response
%    value of spectrally raised cosine at time t
%      t is time indexT
%      T is the symbol period.
%      B is the time multiplier
%

l = size(t);
l=l(2);
if B > 1
t=t-T*ones(1,l);
else
    t=t-(T/B)*ones(1,l);
end
const=2*pi*B/sqrt(log(2));
freq=(1/(2*T))*(q(const*(t-T/2))-q(const*(t+T/2)));

--------------------------------

function phase = Phase_GMSK(t,T,B)
%
% Phase_GMSK computes the TIME-DOMAIN phase pulse response from the freq pulse response
%      tau is time variable input to the function Freq_nSRaiseCos
%      T is the symbol period
%      B is a scaling parameter that can be thought of roughly as 1/n
%      integrates freq-pulse in time from negative infinity to time t
%      t is the function input - phase is a function of t


l=size(t);
l=l(2);
phasearray=zeros(1,l);
for index=1:l
    phasearray(index)= integral(@(u)Freq_GMSK(u,T,B),-inf,t(index)); %50*T,t(index));
end
phase=phasearray;

-----------------------------------

function phase = Phase_GMSKInterp(t,T,B,basephase)
%
% Interpolates the phase pulse response from the samples in basephase
% basephase is a 501 point file of samples of the output of
```

```
%     Phase_GMSK, which itself looks back T/B symbol periods
%     where BT is the characterizing product and assumed of the form 1/integer
%       basephase = Phase_GMSK([0:200]*.1*T;
%     sampling is thus presumed at T/10 in basephase
%     basephase presumes an "1/int =B" so this function is called with
%     basephaseSRC(1/B,:), so basephaseSRC is really 10 x 201 array.
%
%     The value of THE SAME n (or B) is provided to PSum_GMSK, along with h
%     most GMSK use h=1/2.
%     basephase was generated with teh genGMSK function to avoid
%     computing it multiple times, and this is assumed 201 points at T/10
%     T is the symbol period for basephase

l=size(t);
l=l(2);
phase=spline([0:200]*.1,basephase,t);
low=floor(t*10/T);
index=zeros(1,l);
for k=1:l
index(k)=low(k);
    if index(k) < 1
        phase(k)=0;
    elseif index(k) > 199
        phase(k)=0.5;
    end
end

----------------------------

function sump = PSum_nSRC(t,tau,T,B,h,M,m,basephase)
% PSum computes the sum of exponents term in the spectrum integral
%      t is time index
%      tau is offset over the interval 0 to T, so really tauprime
%      B the GMSK paramter (assumed of form 1/integer)
%     To maintain causality, time is extened by 1/B+10
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      indexj is the product index
%      m+1 is the upper index on the product (outer function that uses this one)
%
%     PSum_nSRC is an intermediate term in the spectrum calculation for
%     nSRC
%     sump is the output

lt=size(t);
l=lt(2);
sumptemp=zeros(m+1+(1/B)+10,l);
sump=zeros(1,l);
%n=n+10;
for index=1:m+1+1/B+10
    sumptemp(index,:)=(1/M)*sum(exp(kron(i*2*pi*h*([-M+1:2:M-1]'),Phase_GMSKInterp(t+(tau-(index-(1/
        - Phase_GMSKInterp(t-(index-(1/B+10))*T*ones(1,l),T,B,basephase))),1);
end
```

```
temp=prod(sumptemp([1:m+1+(1/B)+10],:),1);
for indexb=1:l
    sump(indexb)=real(temp(1,indexb));
end

---------------------------
function autoR = R_GMSK(tau,T,B,h,M,basephase)
%
%   R_GMSK computes the autocorrelation function evalualted for any positive time arg
%      tau is time lag (which is positive) and should not exceed (n+1)T
%      tauprime is an internal variable representing the fractional value
%      of tau (tau = tauprime + mT), which must be non-negative
%      n is the number of symbol non-0 symbols in the frequency response
%      B is a time spreading factor roughly equivalent to 1/nu
%     To maintain causality, time is extened by 1/B+10
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%
%     autoR is the output for lag tau

tauprime=tau-floor((1/T)*tau);
m=floor((1/T)*tau);
l=size(tau);
l=l(2);
autoR=zeros(1,l);
for index=1:l
   if tau(index) > (10+1/B)*T
      autoR(index)=0;
   else
autoR(index)=(1/T)*integral(@(t) PSum_GMSK(t,tauprime(index),T,B,h,M,m(index),basephase),0,T);
   end
end

-----------------------

function spec = S_GMSK(T,B,h,M,basephase)
%
%  S_GMSK computes the power spectrum for GMSK autocorrelation
%   To avoid multiple nested integral-function calls, the FFT is used
%   instead
%       The exponential decay of the function in nT to nT+1 for all subsequent
%       symbol periods is used to compute those contributions
%       The functions called extend n to 6n to ensure near causality
%      B = 1/integer index
%      To maintain causality, time is extened by 1/B+10
%      T is the symbol period.
%      h is the index of modulation
%      M is the number of messages
%      this program assumes that sampling interval is .01*T
%      this program computes 10k point FFT, so frequency range is 0:100/T

%   an FFT is used instead of a continuous integral to compute spectrum
%      nREC is called to compute the successive portions of auto
```

```
%      correlation
lr1=(1/B)*T*100;
lr2=T*100;
R=R_GMSK([0:(lr1+lr2)]*.01*T,T,B,h,M,basephase);
R1=R(1:lr1+1);
R2=R(lr1+2:lr1+lr2+1);
temp =[R1 R2];
r([1:100])=R2;
for indexa = 2:50-(1/B)
    r([1:100])=r([1:100])*C_alpha(h,M);
    temp=[temp , r([1:100])];
end
R=[temp, temp(5000:-1:2)];
spec=.01*fft(R,10000);
```

G.7 Chapter 7 Matlab Software listings

G.7.1 MLSD Programs

mlsd1D.m

```
% function mh = MLSD(y,v,b,Sk, Yk, Xk)
%
% MLSD using VA for input trellis
% Written by Ghazi Al-Rawi
%   Updated significantly by J. Cioffi, 2023
% ********************************************************************
% INPUTS
% y  channel output sequence
%       1 x K complex (usually real) number for partial response
%          K is number of input bits in Xk
%       n x K integer if BSC with convolutional code
%   v  constraint length (log2 of the number of states)
%   b  number of bits per subsymbol
%   Sk 2^v x 2^b previous-state description matrix
%       e.g., EPR4's H=[1 1 -1 -1] has (binary) nextstate trellis
%       nextState = [1 2; 3 4; 5 6; 7 8; 1 2; 3 4; 5 6; 7 8];
%       so Sk = [1 5; 1 5; 2 6; 2 6; 3 7; 3 7; 4 8; 4 8];
%   Yk 2^v x 2^b noiseless 1D trellis output corresponding to Sk, so EPR4
%       Yk = [0 -2; 2 0; 2 0; 4 2; -2 -4; 0 -2; 0 -2; 2 0];
%       for 4-state convolutional code
%       Yk= [ 0 3 ; 2 1 ; 3 0 ; 1 2]
%   Xk b x 2^v input vector, so EPR4 could be
%       Xk = [0 1 0 1 0 1 0 1];
%   e  if e=1, then euclidean distance, otherwise hamming distance (xor)
%
% OUTPUT
% mh is the detected message sequence
%
% ********************************************************************

function mh = MLSD(y,v,b,Sk, Yk, Xk, e)

M = 2^b;

C = [0 1e12*ones(1, M^v-1)]; % State metric, initialized so as to start at state 0
D=[];
Paths=[];
C_next=[];

for k=1:length(y),
   i=1;
   for j=1:M^v,
      for b=1:M
          if e == 1
           D(i) = norm(Yk(j, b)-y(k))^2;
          else
           D(i)= sum(xor(dec2bin(Yk(j,b)),dec2bin(y(k))));
          end
         i=i+1;
```

```
      end
   end

   i=1;
   for j=1:M^v,
      minCost=1e12;
      for b=1:M
         cost = C(Sk(j, b))+D(i);
         i=i+1;
         if (cost<minCost)
            minCost = cost;
            Paths(j, k) = Sk(j, b);
         end
      end
      C_next(j)=minCost;
   end

   if (mod(k,1000)==0)
      C_next = C_next - min(C_next); % Normalization
   end

   C = C_next;
end

% Find the best survivor path at k=length(y)
[minCost, I] = min(C);

if (length(I)>1)
   warning('Warning: There are more than one path with equal cost at state k=length(y)!');
end

% We will pick the first
sp=I(1);

xh=[];
for k=length(y):-1:1,
   xh(k) = Xk(sp);
   sp = Paths(sp, k);
end

mh = xh;
```

G.7.2 APP Programs

These programs are from the matlab extra software pages where various engineers contribute.

BCJR_conv.m

```
% function BCJR_conv(y,trellis,sigma)
%                                        BCJR_conv Decoder
%
% This algorithm is reserved to the implementation of the Bahl, Cocke, Jelinek and Raviv (BCJR)
% algorithm. This function takes as input the channel output (corrupted
% data) and the a priori prob (we will set it to 1/2) and returns as output
```

```
% the APP Log Liklihood Ratio (LLR) for every data input. It is usually called a
% Soft Input Soft Output (SISO) decoder. It can be applied to any code
% having a finite state machine, in our case we will use it for rate-1/n convolutional codes.
%
%                                                K. Elkhalil, SUP'COM Tunisia
%
% ************************************************************************%
function LLR=BCJR_conv(y,trellis,sigma)
N=length(y); % y is the channel output
n=log2(trellis.numOutputSymbols);
k=log2(trellis.numInputSymbols); % k=1
R=k/n; % coding rate, R=1/n
LLR=zeros(1,N*R);
Pap=0.5; % The a priori probability.
%********* Computing gamma for all states at each time ***************
gamma=zeros(N*R,trellis.numStates,trellis.numStates); % we suppose that the first state is the 0 sta
    for k=1:N*R
        for s=1:trellis.numStates
            for ss=1:trellis.numStates
            [msg,in]=ismember(ss-1,trellis.nextStates(s,:));
            if msg==1
                gamma(k,ss,s)=0.5*((1/sqrt(2*pi*sigma^2))^n)*exp(-sum((y(n*k-n+1:n*k)-(1-2*binary(tr
            end
            end
        end
    end
%************** alpha recursions*******************
alpha=zeros(N*R+1,trellis.numStates);
alpha(1,1)=1;
    for k=2:N*R+1
        for ss=1:trellis.numStates
            for s=1:trellis.numStates
                alpha(k,ss)=alpha(k,ss)+gamma(k-1,ss,s)*alpha(k-1,s) ;
            end
        end
        alpha(k,:)=alpha(k,:)/sum(alpha(k,:)); % Normalization
    end
%************** beta recursions*******************
beta=zeros(N*R+1,trellis.numStates);
beta(N*R+1,:)=alpha(N*R+1,:);
    for k=N*R+1:-1:2
        for ss=1:trellis.numStates
            for s=1:trellis.numStates
                beta(k-1,ss)=beta(k-1,ss)+gamma(k-1,s,ss)*beta(k,s) ;
            end
        end
        beta(k-1,:)=beta(k-1,:)/sum(beta(k-1,:)) ; % Normalization
    end
%%%%%%%%%%%%%%%%%%% Computing the LLRs %%%%%%%%%%%%%%%
    for k=1:N*R
        up=0;
        down=0;
         for s=1:trellis.numStates
            for ss=1:trellis.numStates
```

```
            [msg,in]=ismember(ss-1,trellis.nextStates(s,:));
            if (msg==1 && in==1) % input=0
               up=up+alpha(k,s)*gamma(k,ss,s)*beta(k+1,ss);
            else if (msg==1 && in==2) % input=1
                    down=down+alpha(k,s)*gamma(k,ss,s)*beta(k+1,ss);
                end
            end
            end
         end
        LLR(k)=log(up/down);
    end
end
```

binary.m : This program is called by BCJR_conv.m .

```
% function binary(x,size)
function bin=binary(x,size)
i=max_bin2(x);
bin=zeros(1,size);
m=x ;
while m>0
m=m-2^i ;
bin(i+1)=1 ;
i=max_bin2(m) ;
end
z=bin ;
for jj=1:length(bin)
    z(jj)=bin(length(bin)-jj+1) ;
end
bin=z ;
end
```

max_bin2.m : This program is called by BCJR_conv.m .

```
function i=max_bin2(x)
y=2^0 ;
i=0;
if x==0
    i=0 ;
else
while x >= y
    y=y*2 ;
    i=i+1;
end
 i=i-1 ;
end%% 2^i=<x<=2^i+1 ;
end
```

G.8 Programs from Chapter 8

G.8.1 Trellis plotting program

This program extracted from matlab's extra program pages that are contributed by various engineers.

plotnextstates.m

```
% function plotnextstates(nextStates)
% Utility function to display nextStates matrix for Convolutional Encoder
% with Uncoded Bits and Feedback example.

%   Copyright 2003-2011 The MathWorks, Inc. $Revision: 1.1.6.2 $  $Date:
%   2011/12/11 07:38:29 $
function plotnextstates(nextStates)
%-- Create a new figure to plot NextStates
figure;

%-- Declare and initialize intermediate variables
[numStates,numIn] = size(nextStates);
yplot = NaN*ones(3*numIn,numStates);
yplot([1:3:end],:) = ones(numIn,1)*[0:numStates-1];
yplot([2:3:end],:) = nextStates';
xplot = repmat([1 numStates NaN]',numIn,numStates);

%-- Plot NextStates matrix
plot(xplot,yplot,'o-','MarkerSize',6, ...
    'MarkerEdgeColor','k','MarkerFaceColor','k');
grid on;
title('Plot of {\itNextStates} Matrix');
xlabel('State Transition');
set(gca,'yTick',[0:numStates-1],'YDir','Reverse',...
    'YTickLabel',...
    [dec2base([0:numStates-1],2) repmat('/',numStates,1) dec2base([0:numStates-1],8)], ...
    'YLim', [-1 numStates], 'Xlim',[0 numStates+1],'xTick',[1 numStates], ...
    'xTickLabel','N|N+1','Position',[.11 .1 .78 .8]);
ylabel('Initial State (Binary/Octal representation)',...
    'HorizontalAlignment','center','VerticalAlignment','Bottom');

axes('yTick',get(gca,'YTick'), ...
    'YAxisLocation','right','Color','none','XTick',[],'YLim',get(gca,'YLim'), ...
    'YTickLabel', get(gca,'YTickLabel'), ...
    'YDir','Reverse', 'Position',get(gca,'Position'));
ylabel('Final State (Binary/Octal representation)', ...
    'HorizontalAlignment','center','VerticalAlignment','Top');

%[EOF]\begin{verbatim}
```

G.8.2 LDPC Code Programs

get_h_matrix.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% function [H_no_dep H] = get_h_matrix(p,tr,tc,first_1);
%% Generate LDPC H Matrix Uses Generic-LDPC Method As Per Cioffi's Class Notes
```

```
%% Example: to Generate (529,462) code, p=23, rw=23, cw=3, first_1=2
%%          H = get_h_matrix(23,23,3,2),
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of input variables
%% p       :  Prime number of the size of base matrix of size p-by-p
%% tr      :  Row weight = # of base matrices (or 1's) /row, equivalent to K
%% tc      :  Col weight = # of base matrices (or 1's) per column,eq to J
%% first_1:  Set to 2 in generic LDPC code, so right shift by first_1-1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of output variables
%% H_no_dep  : the parity check matrix with no dependent rows
%% H         : without removing the dependent rows
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EE379, Chien-Hsin Lee, first version 06/2006, edits by J. Cioffi since
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [H_no_dep , H] = get_h_matrix(p,tr,tc,first_1_start);
% generate base matrix
a = eye(p,p);
a = [a(first_1_start:p,:);a(1:first_1_start-1,:)];

% generate H matrix
H = [];
for row = 1:tc
    current_row = [];
    for cl = 1:tr
        current_row = [current_row,a^((row-1)*(cl-1))];
    end
    H = [H;current_row];
end

% remove dependent rows
% this need to be update with mod2 operation
[Q R] = qr(H);
H_no_dep = [];
for i = 1:p*tc
    if( abs(R(i,i)) > 1E-9)
        H_no_dep = [H_no_dep;H(i,:)];
    end
end
```

systematic.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% function [H_no_dep H] = get_h_matrix(p,tr,tc,first_1);
%% Generate LDPC H Matrix Uses Generic-LDPC Method As Per Cioffi's Class Notes
%% Example: to Generate (529,462) code, p=23, rw=23, cw=3, first_1=2
%%          H = get_h_matrix(23,23,3,2),
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of input variables
%% p       :  Prime number of the size of base matrix of size p-by-p
%% tr      :  Row weight = # of base matrices (or 1's) /row, equivalent to K
%% tc      :  Col weight = # of base matrices (or 1's) per column,eq to J
%% first_1:  Set to 2 in generic LDPC code, so right shift by first_1-1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% Definition of output variables
%% H_no_dep  : the parity check matrix with no dependent rows
%% H         : without removing the dependent rows
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EE379, Chien-Hsin Lee, first version 06/2006, edits by J. Cioffi since
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [H_no_dep , H] = get_h_matrix(p,tr,tc,first_1_start);
% generate base matrix
a = eye(p,p);
a = [a(first_1_start:p,:);a(1:first_1_start-1,:)];

% generate H matrix
H = [];
for row = 1:tc
    current_row = [];
    for cl = 1:tr
        current_row = [current_row,a^((row-1)*(cl-1))];
    end
    H = [H;current_row];
end

% remove dependent rows
% this need to be update with mod2 operation
[Q R] = qr(H);
H_no_dep = [];
for i = 1:p*tc
    if( abs(R(i,i)) > 1E-9)
        H_no_dep = [H_no_dep;H(i,:)];
    end
end
```

nearestNds.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% function [Nd , dmin] = Neighbors(G);
%% This routine counts neighbors at distances dmin:dmin+19 for binary code G
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of input and internal variables
%% G       : generator matrix
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of output variables
%% Nd      : Number of codewords at distances dmin to dmin+19
%% dmin    : minimum Hamming distance
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EE379A, J. Cioffi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Nd, dmin] = neighbors(G);
Nd=zeros(1,20);
dmin = min(sum(G,2));
[limit,~]=size(G);
 for d=dmin:dmin+19
  for n=1:limit
   if sum(G(n,:)) == d
   Nd(d-dmin+1) = Nd(d-dmin+1) + 1;
```

```
   end
  end
 end
end
```

G.8.3 LDPC Encoder and Decoder Programs

encoder.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% [rx_bit, coded_bit, message_bit] = encoder(SNR,G,random,message_bit)
%% LDPC Encoder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of input variables
%% SNR         : Receiver SNR in dB
%% G           : Generator matrix k x n
%% random      : set to 1 for encoder to generate k random input bits
%% message_bit : message_bit to be encoded - should be k bits long 1 x k
%%             : if random is 1, the input message_bit is ignored
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of output variables
%% rx_bit      : coded_bit + AWGN
%% coded_bit   : encoded bit
%% message_bit : message bit to be encoded
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EE379, 06/2006, Chien-Hsin Lee , subsequent edits J. Cioffi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [rx_bit, coded_bit, message_bit] = encoder(SNR,G,random,message_bit)

[k,n]      = size(G);
if(random == 1)
    message_bit = round(rand(1,k));
end

noise        = randn(1,n)/sqrt(10^(SNR/10));
coded_bit    = mod(message_bit*G,2);
modulated_bit= coded_bit*2-1;
rx_bit       = modulated_bit + noise; %% sending -1 only
```

ldpc_decoderspa.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% function [decoded_bits llr_bits iter]=ldpc_decoder_spa(H,ybits,max_iter,var,fast)
%% SPA LDPC Decoder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of input variables
%% H        : parity check matrix
%% ybits    : received signal y
%% max_iter : maximun number of iterations
%% var      : noise variance (constraint decoders must know this)
%% fast     : 1: stop at good parity, 0: run till maximum iteration.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definition of output variables
%% decoded_bit : decoded bits
%% llr_bit     : llr of each input bit
```

```
%% iter        : number of iteration used
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EE379, 06/2006, Chien-Hsin Lee, subsequent edits J. Cioffi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [decoded_bits,llr_bits, iter ] = ldpc_decoder_spa(H,ybits,max_iter,var,fast)
[row col] = size(H);
iter      = 0;
pass      = 0;

llr_int   = zeros(1,col);    % intrinsic
llr_b2c   = zeros(row,col);  % bit node to check node
llr_c2b   = zeros(row,col);  % check node to bit node
tr        = (-1).^mod(sum(H')',2);

%% intrinsic bit probobility
for i = 1:col
    llr_int(i) = ((ybits(i)+1)^2 - (ybits(i)-1)^2)/2/var;
end
llr_bits        = llr_int;
decoded_bits(i) = (sign(llr_bits(i))+1)/2;

while(iter < max_iter && (pass ~= 1 || fast == 0) )
iter = iter + 1;
%% update bit note to check node LLR
for j = 1:col
        row_ptr  = find(H(:,j));
        for i = 1:length(row_ptr)
            llr_temp =llr_int(j);
            for k = 1:length(row_ptr)
                if( k ~= i)
                    llr_temp = llr_temp + llr_c2b(row_ptr(k),j);
                end
            end
            llr_b2c(row_ptr(i),j) = llr_temp;
        end
end

%% update probibilty of check equation
for i = 1:row
        col_ptr = find(H(i,:));
        for j = 1:length(col_ptr);
            X = 1;
            for k = 1:length(col_ptr);
                if( k ~= j)
                    X = X*tanh(llr_b2c(i,col_ptr(k))/2);
                end
            end
            llr_c2b(i,col_ptr(j)) = tr(i)*2*atanh(X);
        end
end

    %% check result
    for i=1:col
        llr_bits(i)     = llr_int(i)+sum(llr_c2b(:,i));
```

```
        decoded_bits(i) = (sign(llr_bits(i))+1)/2;
    end
    pass    = 1-sum(mod(decoded_bits*H',2));
end %% while()
```

Routine to call decoder ldpc_main.m The program main calls the decoder

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This is the main LDPC simulation bench
%% This sets up to run the (529,462) code
%% This will take about 10 hrs to run
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% EE379, Chien-Hsin Lee, 06/2006 ; consequent edits J. Cioffi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Define LDPC Code Here
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
prime      = 23;
col_weight = 3;
row_weight = 23;
H       = get_h_matrix(prime,row_weight,col_weight,2);
[Hs Gs] = systematic(H);
[r n]   = size(Hs);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Define Simulation Condition
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
iteration   = 100;              %% max number of iteration
block_error = 10;               %% number of block error for each sample point
SNR         = [6:0.5:8];        %% SNR points
var         = 1./10.^(SNR/10); %% variance of noise
random      = 1;                %% 1:generate message bit for simulation
fast        = 0;                %% 1:stop when parity is right

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Initial Program Variables
%% You should not need to touch this
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
steps         = length(SNR);
num_bit_err   = zeros(1,steps);
num_block_err = zeros(1,steps);
num_block     = zeros(1,steps);
iter          = zeros(1,steps);
fail_coded_bits   = [];
fail_receive_bits = [];
fail_decoded_bits = [];
randn('seed',12345678);
rand('seed',98765432);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Main Routine
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1:steps
```

```
    t1             = clock;
    while(num_block_err(i) < block_error)
        [rx_bit coded_bit message_bit] = encoder(SNR(i),Gs,random,zeros(1,n-r));
        [decode_bits llr iter_temp]    = ldpc_decoder_spa(Hs,rx_bit,iteration,var(i),fast);
        num_block(i) = num_block(i) + 1;
        iter(i)      = iter(i) + iter_temp;
        current_bit_err = sum(abs(decode_bits - coded_bit));
        if(current_bit_err ~= 0)
           num_bit_err(i)     = num_bit_err(i) + current_bit_err;
           num_block_err(i)   = num_block_err(i) + 1;
           fail_coded_bits    = [fail_coded_bits;  coded_bit];
           fail_receive_bits  = [fail_receive_bits;rx_bit];
           fail_decoded_bits  = [fail_decoded_bits;decode_bits];

           %% display current progress also dump the file
           t2 = clock;
           run_time(i) = etime(t2,t1);
           dlmwrite('result.txt', [SNR(1:i); run_time(1:i); num_bit_err(1:i); num_block(1:i); iter(1
           dlmwrite('fail_coded_bits.txt', fail_coded_bits, ' ');
           dlmwrite('fail_receive_bits.txt', fail_receive_bits, ' ');
           dlmwrite('fail_decoded_bits,txt', fail_decoded_bits, ' ');
           flg = sprintf('SNR= %0.2d; rec_block= %d; block_err= %d; bit_err= %d; run time= %0.3g; it
                         SNR(i),num_block(i),num_block_err(i), num_bit_err(i), run_time(i), iter(i)
           disp(flg);
        end
    end
end

plot(SNR(1:steps),log10(num_bit_err(1:steps)./num_block(1:steps)./n), ...
     SNR(1:steps),log10(num_block_err(1:steps)./num_block(1:steps)) )
legend('BER','BLER');
grid;
```

Bibliography

[1] Thomas Kailath, Ali H. Sayed, and Babak Hassibi. *"Linear Estimation"*. Prentice Hall, USA, 2000.

[2] Lasha Ephremidze. *"An Elementary Proof of the Ploynomial Matrix Spectral Factorization Theorem"*. Proceedings of the Royal Society of Edinburgh Section A: Mathematics, DOI: https://doi.org/10.1017/S0308210512001552Published online by Cambridge University Press: 24 July 2014. AMS Subject Classification (2010): 47A68.

1

2

3

4

5

6

7

8