# Implementing the Belief Propagation Algorithm in MATLAB

Björn S. Rüffer* Christopher M. Kellett*

Technical Report
Version as of November 13, 2008

We provide some example Matlab code as a supplement to the paper [6]. This technical report is **not** intended as a standalone introduction to the belief propagation algorithm, but instead only aims to provide some technical material, which didn't fit into the paper.

## 1 Introduction

For an excellent introduction and mathematical treatise of modern iterative decoding theory, we refer to [4]. Worth mentioning is also the survey paper [2] on factor graphs and the sum-product algorithm, the superclass that contains belief propagation. This current technical note provides Matlab code to implement the dynamical system formulation of the belief propagation algorithm and a few related concepts, as detailed in [6]. More conventional implementations —that is, from a coding perspective— exist and some are publicly available [3].

The Matlab code examples detailed in this report can be found, along with the most up-to-date version of this report itself, at [5].

Our presentation differs also in another aspect from the standard ones: Unlike the information theory convention, where messages and codewords are represented by row vectors, we throughout use column vectors as this is standard in dynamical systems. Of course this does not lead to differences other than representational ones.

This report is organized as follows: In Section 2 we give a simple example on how one can generate a very basic random parity-check matrix and compute a corresponding generator matrix. Section 3 details the channel transmission and Section 4 provides code to implement the belief propagation algorithm as a dynamical system. The output trajectories obtained using this Matlab code can then be plotted using the routine in

*School of Electrical Engineering and Computer Science, The University of Newcastle, Australia, Bjoern.Rueffer@Newcastle.edu.au, Chris.Kellett@newcastle.edu.au

Section 5. In Section 6 we provide a more advanced method for generating parity-check matrices with prescribed degree distribution.

## 2 Parity-Check and generator matrices

A parity-check matrix is any matrix $H \in \mathbb{F}_2^{m\times n}$. Throughout we assume that $n = m+k$, $n, m, k > 0$ and that $H$ has full rank, i.e., rank $m$. If $H$ does not have full rank, then rows can be removed until it does, thereby increasing $k$ and decreasing $m$ accordingly.

To generate a parity-check matrix for a repeat-$n$ code in canonical form, one could use the following Matlab statement:

```
n=10; m=n-1; H=spalloc(n-1,n,2*(n-1)); for i=1:n-1, H(i,i)=1; H(i,i+1)=1; end
```

A simple random parity-check matrix can be generated using the code in Listing 1, and code for generating more involved parity-check matrices is given in Section 6.

Using only Gauss elimination and possibly by swapping columns, $H$ can be brought into the form

$$QHP = \begin{bmatrix} I_m & A \end{bmatrix}, \tag{1}$$

where the invertible matrix $Q \in \mathbb{F}_2^{m\times m}$ encodes the steps of the Gauss elimination (swapping and adding rows of $H$), $P \in \mathbb{F}_2^{n\times n}$ is a permutation matrix to encode the swapping of columns in $H$, $A \in \mathbb{F}_2^{m\times k}$, and $I_m$ is the $m \times m$ identity matrix.

A generator matrix for $H$ is a matrix $G \in \mathbb{F}_2^{n\times k}$ such that $HG = 0$. According to (1) we can take $G$ to be

$$G = P \begin{bmatrix} A \\ I_k \end{bmatrix}, \tag{2}$$

since

$$QHG = \begin{bmatrix} I_m & A \end{bmatrix} P^{-1} \cdot P \begin{bmatrix} A \\ I_k \end{bmatrix} = 2A = 0 \quad (\text{in } \mathbb{F}_2^m).$$

Now $G$ maps *message vectors* $\mathbf{m} \in \mathbb{F}_2^k$ to *codewords* $c = G\mathbf{m} \in \mathbb{F}_2^n$, i.e., to elements of the null-space $\mathcal{C} = \mathcal{C}_H = \{x \in \mathbb{F}_2^n : Hx = 0\}$ of $H$, which is also termed the *set of codewords* or just the *code*.

The MATLAB code examples in Listings 1 and 2 can be used to generate a very basic parity-check matrix and to obtain a generator matrix from a given parity-check matrix.

Listing 1: A crude way to obtain a simple parity-check matrix, by just specifying the dimensions $m$ and $n$ and a density of non-zero elements in $H$ of at least $d \in (0, 1)$.

```
function H = generate_H(m,n,d)
% H = GENERATE_H (m,n,d)
%
% generate a m by n parity check matrix, where the density is
% influenced by the parameter d (between zero and one)

H=sparse(m,n);
H=mod(H,2);
```

```
while not(all(sum(H,1)>=2) && all(sum(H,2)>=2)),
   H=H+abs(sprand(m,n,d))>0;
   H=mod(H,2);
end
```

Listing 2: A code example to construct a generator matrix for a given parity-check matrix. Some consistency checks (e.g., to see if $n > m$) are omitted.

```
function [G]=generatormatrix(H);
% function [G]=generatormatrix(H); given a sparse parity check matrix
% H compute a generator matrix G

Hp=H;
[m,n]=size(Hp);

%suppose m<n!

colperm=1:n;

for j=1:m,
    % find row to put as new row j
    i=min(find(Hp(j:m,j)));
    if isempty(i);
        % do some column swapping!
        k=min(max(find(Hp(j,:)),j));
        if isempty(k),
            disp(['problem in row' num2str(j,0)]);
            continue;
        end
        temp = Hp(:,j);
        Hp(:,j)=Hp(:,k);
        Hp(:,k)=temp;
        temp=colperm(k);
        colperm(k)=colperm(j);
        colperm(j)=temp;
    end
    % swap rows
    % adjust indices!
    i=i+j-1;
    if (i~=j),
        temp = Hp(j,:);
        Hp(j,:)=Hp(i,:);
        Hp(i,:)=temp;
    end % if
    % clear out rest of column
    K= find(Hp(:,j));
    K= K(find(K~=j));
    if ~ isempty(K),
```

```
41            t1=full(Hp(j,:));
42            for k=K',
43                t2=full(Hp(k,:));
44                temp=xor(t1,t2);
45                Hp(k,:)=sparse(temp);
46            end
47        end
48    end % for
49
50    % now Hp = [Id_m A]
51    A = Hp(:,m+1:n);
52
53    %compute G
54    [b,invperm]=sort(colperm);
55    G = [A; speye(n-m)];
56    G=G(invperm,:);
57    % consistency check: mod(H*G,2) should give all-zero matrix
58
59    end % function
```

# 3 Transmission trough an AWGN channel

Now that we can generate codewords for given messages, we still have to map these to channel symbols, transmit via an Additive White Gaussian Noise (AWGN) channel and compute log-likelihood ratios (LLRs) afterwards. This is the aim of this section.

An AWGN channel takes as input a real number $x$ and outputs $x+z$, where $z$ is drawn from a $\mathcal{N}(0,\sigma^2)$ normal distribution. Here $\sigma$ is a channel parameter. Transmission of a vector $x \in \mathbb{R}^n$ means to consecutively transmit its components. It is assumed that the noise samples affecting each component are all independent.

## 3.1 Binary phase shift keying (BPSK)

A given codeword $x \in \mathcal{C} \subset \mathbb{F}_2^n$ can be mapped to a vector $\tilde{x} \in \mathbb{R}^n$ via

$$x_i \mapsto \tilde{x}_i = (-1)^{x_i}\,. \tag{3}$$

This procedure is commonly referred to as BPSK.

## 3.2 The transmission step

We have to fix a channel parameter $\sigma > 0$. In Matlab, the transmission step is now very easy, we just use the assignment $y =$ *tilde_x* + **randn**(*n*,1)∗*sigma*^2, to compute the vector of received channel symbols $y \in \mathbb{R}^n$.

### 3.3 Computing LLRs

The log-likelihood ratios for each bit are given by

$$u_i = \log \frac{p_{Y_i|X_i}(y_i|0)}{p_{Y_i|X_i}(y_i|1)}\,, \tag{4}$$

where $p_{Y_i|X_i}$ is the density of the conditional probability $(x_i, y_i) \mapsto P(Y_i \leq y_i | X_i = x_i)$.

To compute $u_i$, we actually have to know $\sigma$, or at least make a guess $\hat{\sigma}$ about $\sigma$. The guessing step, which we will omit here, is called *estimation*. For simplicity we assume that the receiver knows $\sigma$.

Substituting the density formulas for the $\mathcal{N}(1, \sigma^2)$ and $\mathcal{N}(-1, \sigma^2)$ distributions into (4) we obtain

$$u_i = \frac{4y_i}{2\sigma^2}\,. \tag{5}$$

So at this stage we can encode a $k$-bit message to an $n$-bit codeword, transmit it through a noisy channel (using BPSK) and compute a-priori log-likelihood ratios.

## 4 The Belief Propagation (BP) algorithm

A detailed descriptionof BP and motivation why this algorithm should do what it does, can be found in [4]. The implementation presented here is based on the article [6], which also contains a rather condensed introduction to iterative decoding using belief propagation. Listing 3 sets up the matrices $B$ and $P$ as well as the structure needed for the operator $S$, which is given in Listing 4. Listing 5 implements the BP algorithm as a dynamical system. The *iterate_BP*($T$,$u$) takes a number of iterations $T$ to perform and a vector of input LLRs $u \in \mathbb{R}^n$ as arguments. The output is a real $n \times (T+1)$ matrix containing the trajectories of the output LLRs.

Listing 3: Initialization for the main program: Compute the matrices $B$,$P$ and the structure for the operator $S$, which is here also encoded via a matrix (S_)

```
function []=H2DS(H)
% H2DS (H)
%
% generate matrices for dynamical system associated to H, i.e., P,
% S_, and q, which are stored as global variables, as well as m and
% n, the dimensions of H

global B P S_ q m n

[m,n]=size(H);
q=nnz(H);

% calculate the amount of nonzero elements
% needed for these matrices:
```

```
P=spalloc(q,q,(sum(H,2)-1)' * sum(H,2));
S_=spalloc(q,q,(sum(H,1)-1) * sum(H,1)');

% find the matrix P
k=0;
for j=1:n,
   I=find(H(:,j));
   for x=1:length(I),
      for y=x+1:length(I),
         P(k+x,k+y)=1;
         P(k+y,k+x)=1;
      end
   end
   k=k+length(I);
end

% find S_ (structure for the nonlinearity S)
k=0;
for i=1:m,
   J=find(H(i,:));
   for x=1:length(J),
      for y=x+1:length(J),
         S_(k+x,k+y)=1;
         S_(k+y,k+x)=1;
      end
   end
   k=k+length(J);
end

% compute matrix B
B=spalloc(q,n,q);
b=[];
for k=1:m,
   b=[b find(H(k,:))];
end
B=sparse([1:q]',b',ones(q,1),q,n);
```

Listing 4: The implementation of the nonlinearity $S : \mathbb{R}^q \to \mathbb{R}^q$

```
function y = S(x)
% S - the nonlinearity in the BP feedback system
%
% y=S(x) is the vector obtained by applying the atanh
% formula to those indices of x corresponding to the ith
% row in the global matrix S_

global S_ q

y=ones(q,1);
for i=1:q,
```

```
    for j=find(S_(i,:)),
      y(i) = y(i) * tanh(x(j)/2);
    end
end
y=2*atanh(y);
```

Listing 5: A function to calculate output trajectories for the dynamical system mimicking BP, where $u \in \mathbb{R}^n$ is the vector of a-priori LLRs and $T \in \mathbb{N}$ is the number of iterations to make

```
function y = iterate_BP(T,u)
% y = iterate_BP(T,u) - This function implements the BP dynamical
%  system; the output trajectory is returned for final time T and
%  input u. The initial state is always zero.

global B P S_ q n m

x1_k=zeros(q,1);
x2_k=zeros(q,1);
x1_k_1=zeros(q,1);
x2_k_1=zeros(q,1);

y=zeros(n,T+1);
for t=1:T,
   x1_k_1=P*x2_k+B*u;
   x2_k_1=S(x1_k);
   y(:,t)=B' * x2_k + u;
   x1_k=x1_k_1;
   x2_k=x2_k_1;
end
y(:,T+1)=B' * x2_k + u;
```

## 5 Plotting output trajectories

To visualize the output of the function *iterate_BP*() from the previous section, we provide here a routine to generate a plot in the flavor of Figure 2 in [6], see Listing 6

Listing 6: Plotting output trajectories in color or monochrome

```
function plot_BP_output(y,mono,filename)
% plot_BP_output(y,mono,filename) - Plot a given output trajectory.
% If mono is supplied (regardless of its value), then plot in
% monochrome, otherwise in color. If in addition a filename is
% supplied, save the figure to that file (in EPS format). Saving
% to a file implies that the plot will be monochrome.

global n
```

```
T=size(y); T=T(2)-1;

clf;
if nargin>1,
  plot(0:T,y,'ko-') % monochrome
else
  plot(0:T,y,'o-') % color
end
grid on;
axis([0 T min(min(y))-.5 max(max(y))+.5])
LEGEND=[];
for k=1:n,
  LEGEND=[LEGEND; strcat('output',num2str(k))];
end
legend(LEGEND)
xlabel('time'); ylabel('LLR');

if nargin==3,
  if isstr(filename),
    saveas(gcf,filename,'eps');
  end
end
```

## 6 More parity-check matrices

In this section we first introduce so-called regular parity check matrices. These are a special case of parity-check matrices with a prescribed degree distribution pair. By the degree distribution we actually refer to the bipartite undirected graph defined by a parity-check matrix, the so-called *Tanner* or *factor graph*, cf. [4].

In fact, it is mostly the degree distribution pair, that determines how the majority of the possible choices of parity-check matrices for that pair and a given code length perform [4, p.94], at least for very large block length.

One distinguishes the *factor* or *check node* distribution $\rho$ and the *variable* or *bit node* distribution $\lambda$. Together these form a degree distribution pair $(\lambda, \rho)$, given by the polynomials

$$\lambda(x) = \sum_i \lambda_i x^{i-1} \tag{6}$$

$$\rho(x) = \sum_i \rho_i x^{i-1}, \tag{7}$$

cf. [4, p.79], where $\lambda_i$ is the *fraction of edges* that connect to a variable node of degree $i$ and $\rho_i$ is the fraction of edges that connect to a check node of degree $i$.

## 6.1 Regular parity-check matrices

Now for a given pair of positive integers $(l, r)$, a *regular* $(l, r)$-code or a regular $(l, r)$ parity check matrix has $\lambda_l = 1$, $\lambda_i = 0$ for $i \neq l$ and $\rho_r = 1$ and $\rho_i = 0$ for $i \neq r$. That is, the degree is the same for all variable nodes, namely $l$, and the same for all check nodes, namely $r$.

The interesting thing about regular codes is, that for a fixed pair $(l, r)$ the number $q$ of non-zero entries in $H$ scales linearly with $n$. Note that $m$ must satisfy $l \cdot n = r \cdot m$, so that $m$ is determined by the triple $(l, r, n)$.

## 6.2 Capacity achieving degree distributions

It has been shown, that certain degree distribution pairs lead to codes with rates performing extremely close to Shannon's channel capacity, the theoretical limit. Such degree distributions can be found in [1]. There a code length of $n = 10^7$ has been used, with a randomly constructed parity-check matrix (avoiding short cycles). At a maximal variable degree of 200 and an average check node degree of 12, it could be shown that the code performed within 0.04 dB of the Shannon limit at a bit error rate of $10^-6$. It is reported that in this setting for a successful decoding an average of 800–1100 iterations were needed.

Please note that the provided Matlab code has only been tested with much shorter block length, e.g., $n = 100$. To efficiently simulate large block-lengths, some numerical improvements should be taken, as indicated in [1].

## 6.3 Generating parity-check matrices with pre-described degree distributions

In Listing 7 we provide an example Matlab program to generate a parity-check matrix for a given degree distribution pair.

Listing 7: Generate a parity-check matrix with prescribed degree distribution pair.

```
function [H,final_column_weights,final_row_weights] = ...
    MacKayNealCreateCode(n,r,v,h)
% ------------------------------------------------------------------------
% MacKay Neal algorithm for constructing parity-check matrix
%
% Date created: 3 November 2008
%
% Inputs: code length (n), code rate (r), column weight polynomial
%         (v), row weight polynomial (h)
%
% Outputs: n(1-r) x n parity-check matrix (H), generated column
%          and row weight polynomials
%
% ------------------------------------------------------------------------
% Sample parameters for rate-1/2, length 100, (3,6)-regular code
% n = 100;                        % Code length
% r = 0.5;                        % Code rate
```

```
% v = [0 0 1.0];                    % Column distributions as polynomial
% h = [0 0 0 0 0 1.0];               % Row distribution as polynomial
% ------------------------------------------------------------------------

% Initialisation
m = floor(n*(1-r));
H = zeros([m,n]);
alpha = [];                 % alpha will contain the column weight for
                            % each column
for i = 1:length(v)
    for j = 1:(floor(v(i)*n)) % always underfill and then add extras
                              % later
        alpha = [alpha i];
    end
end
while (length(alpha) ~= n) % fill out alpha to the appropriate length
    alpha = [alpha i];
end

beta = [];          % beta will contain the row weight for each row
for i=1:length(h)
    for j=1:(floor(h(i)*m)) % always underfill and then add extras
                            % later
        beta = [beta,i];
    end
end
while (length(beta) ~= m)  % fill out beta to the appropriate
                           % length
    beta = [beta i];
end

% Construction
for i = 1:n
    % construct column i
    c = [];
    beta_temp = beta;
    for j = 1:alpha(i)
        temp_row = randint(1,1,[1,m]);
     % We rule out choosing the same row twice for one column by
     % indicating a selection in beta_temp with a -10.  We also
     % select a row that has yet to equal its desired row weight IF
     % POSSIBLE.  However, since we insist on getting the correct
     % column weight, we will end up with some rows having one too
     % many entries.  C'est la guerre.  The actual row weights thus
     % constructed are calculated below.  You should check that
     % they're not too far off...
        while (((beta_temp(temp_row) == 0) && ...
                (max(beta_temp) > 0)) || ...
               ((beta_temp(temp_row) <= -1)))
```

```
            temp_row = mod(temp_row+1,m)+1;
        end
        c = [c temp_row];
        beta_temp(temp_row) = -10;
    end

    % decrement entries in beta
    for k = 1:length(c)
        beta(c(k)) = beta(c(k))-1;
    end

    % populate H
    for j = 1:alpha(i)
        H(c(j),i) = 1;
    end
end

% Calculate actual column distribution
column_weights = H'*ones(m,1);
for i = 1:max(column_weights)
    count = 0;
    for j = 1:length(column_weights)
        if (column_weights(j)==i)
            count = count + 1;
        end
    end
    final_column_weights(i) = count/length(column_weights);
end

% Calculate actual row weights
row_weights = H*ones(n,1);
for i = 1:max(row_weights)
    count = 0;
    for j = 1:length(row_weights)
        if (row_weights(j)==i)
            count = count + 1;
        end
    end
    final_row_weights(i) = count/length(row_weights);
end
```

## 7 Example

In Listing 8 we show some exemplary use of the provided Matlab functions. Suppose we would like to generate a simple random $10 \times 15$ parity-check matrix, see what it looks like, and compute a generator matrix. Here we have $n = 15$, $m = 10$, and hence $k = 5$.

In Figure 7 we see the structure of these two matrices.

Now let us generate a random message vector $\mathbf{m} \in \mathbb{F}_2^n$, encode this message to a codeword $x$, and transmit it via an AWGN channel with standard deviation $\sigma = 0.5$.

Now we could base a *hard decision* on this vector $u$ and see if that yields a codeword, i.e., in Matlab we could try ~**any**(*mod*(*H*∗(*double*(*u*<0)),2)), which should return 1 if we have a codeword. We may find at this stage that we do not have found a valid codeword yet. So let us perform belief propagation decoding, using the dynamical system implementation given in Section 4.

Listing 8: Generating a parity-check and a generator matrix as in Section 7. Exercise the encoding, channel transmission and decoding.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Setup a repeat 10-code.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

n=10; m=n-1; H=spalloc(n-1,n,2*(n-1));
for i=1:n-1,
  H(i,i)=1;
  H(i,i+1)=1;
end
G=generatormatrix(H);
spy(H)                          % see what H looks like
spy(G)                          % see what G looks like
H2DS(H); % setup for the BP algorithm

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Alternatively: setup a random length-15 code of dimension 5.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

H=generate_H(10,15,.05); % 15-10 is 5 as we all know
%%
% at this stage H might not have full rank, in that case we just
% repeat this step
%%
global H
spy(H)                          % see what H looks like
saveas(gcf,'H.eps','eps');
G=generatormatrix(H);
spy(G)                          % see what G looks like
saveas(gcf,'G.eps','eps');
H2DS(H); % setup for the BP algorithm


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

```
%
% Now generate, encode and channel code a random message
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[m,n]=size(H); k=n-m;
message=double(rand(k,1)>0.5); % a k-vector of 0s and 1s
x=G*message; % x is now an n-vector of 0s and 1s (a codeword)
x_tilde=bpsk(x); % make that 1s and -1s (codeword in channel symbols)
iscodeword(hard_decision(x_tilde)) % everything OK up to here?


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Simulate transmission through an AWGN channel. Actually, we
% repeat as often as needed until the channel corrupts the
% transmitted codeword, so that we will have the opportunity to
% employ the iterative decoder.
%
%
% REPEAT FROM HERE TO VIEW PLOTS OF DIFFERENT TRAJECTORIES
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


sigma=0.5; % the channel parameter

% generate a channel output which at first is not a codeword:
y=x_tilde+randn(n,1)*sigma^2; % the channel adds the noise
TRIES=100; t=0;
while iscodeword(hard_decision(y)) && t<TRIES,
   y=x_tilde+randn(n,1)*sigma;
   t=t+1;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Compute the a-priori log-likelihood ratios (u) and perfrom T
% iterations of the BP algorithm (actually, in our implementation,
% we do T HALF-iterations.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

T=50; % (half-)iterations for BP to perform

u=4*y/(2*sigma^2); % compute a-posteriori LLRs
if ~any(mod(H*double(y<0),2)),
  'y already represents a codeword'
```

```
87    clf;
88  else
89    'performing BP'
90    Y=iterate_BP(T,u); % that means T (half-)iterations
91    for k=1:T,
92      if ~any(mod(H*double(Y(:,k)<0),2)),
93        strcat('found a codeword at iteration _',num2str(k))
94        break;
95      end
96    end
97    plot(0:T,Y,'o-')
98  end
```

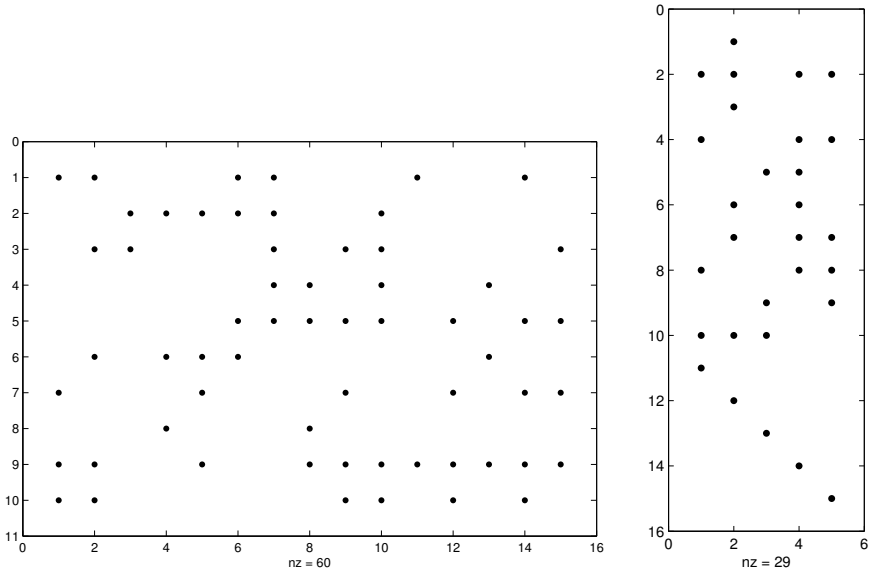


Figure 1: The graphical output of Listing 8: $H$ on the left and $G$ on the right.

## 8 Remarks on numerical precision

In the graphical output generated in the previous section we observe that some trajectories seem to cease to exist and then possibly reappear later on. This effect is caused by round-off errors due to the numerical precision of the computations in Matlab. For very large arguments the tanh-function returns 1, and atanh(1) = $\infty$. In theory though, finite arguments always give finite outputs, but number representation limits Matlab in actually producing accurate results after sometimes only a few iterations. Countermeasures employed, e.g., in [3], include to apply some saturation function after each iteration to the log-likelihood ratios, so that they do not grow unboundedly.

## References


[1] S.-Y. Chung, G. D. Forney, T. J. Richardson, and R. L. Urbanke. On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit. *IEEE Communications Letters*, 5(2):58–60, Feb. 2001.

[2] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[3] R. M. Neal. Software for Low Density Parity Check (LDPC) codes. online, 2001. http://www.cs.toronto.edu/ radford/software-online.html.

[4] T. Richardson and R. Urbanke. *Modern Coding Theory*. Cambridge University Press, New York, 2008.

[5] B. S. Rüffer and C. M. Kellett. Example Matlab code to implement Belief Propagation as a Dynamical System, Nov. 2008. Available at http://sigpromu.org/systemanalysis.

[6] B. S. Rüffer, C. M. Kellett, P. M. Dower, and S. R. Weller. Belief Propagation as a Dynamical System: The Linear Case and Open Problems. *Submitted*, Nov. 2008. Preprint available at http://sigpromu.org/systemanalysis/publications.html.