Appendix D

Simulation tools

The entire simulation was carried out with MATLAB/SIMULINK, a comfortable engineering tool for numerical calculations and simulations. The plant model was built using SIMULINK standard blocks from its library in a hierarchical manner as it is usual for SIMULINK. Separately, the control algorithm was coded in MATLAB for two reasons. Firstly, it seems to be more realistic to simulate a DSP by a function with the same inputs and outputs as in a real time application. Secondly, a sophisticated algorithm can be coded in MATLAB very easily, and can later on be translated into C-code with MATLAB functions. Therefore, the first section shows all SIMULINK block diagrams used, the second section presents the data file and all other functions which are part of the algorithm.

D.1 Simulink block diagrams

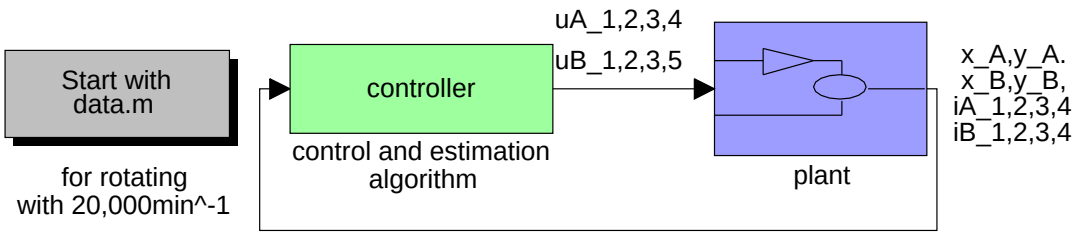

Figure D.1: Block diagram of the entire simulation loop with plant and the SIMULINK s-function "control and estimation algorithm".

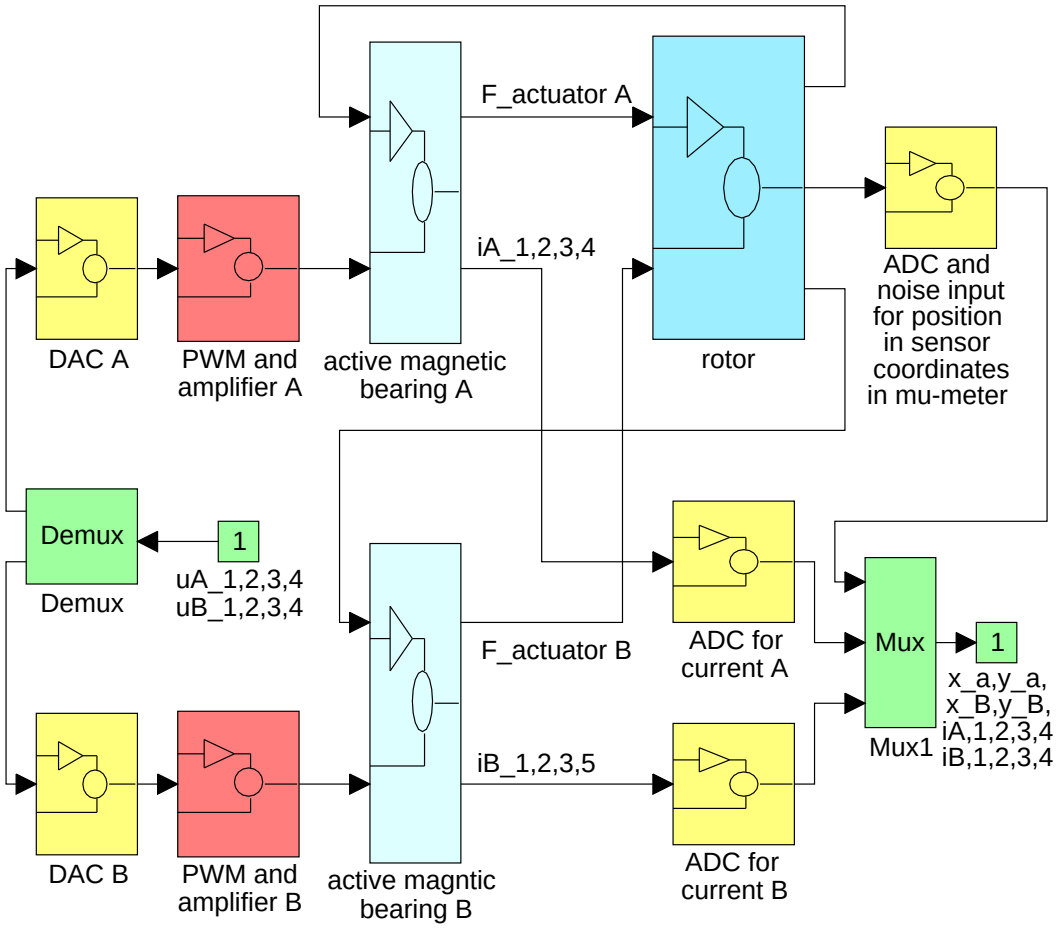

Figure D.2: SIMULINK block diagram of the plant with the elements controller, DAC, PWM with amplifier, actuators, rotor, and ADCs.

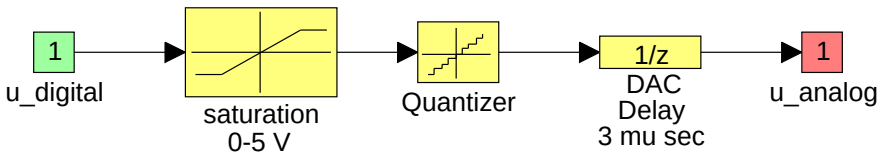

Figure D.3: One DAC for the output voltage as an example for all 8 channels.

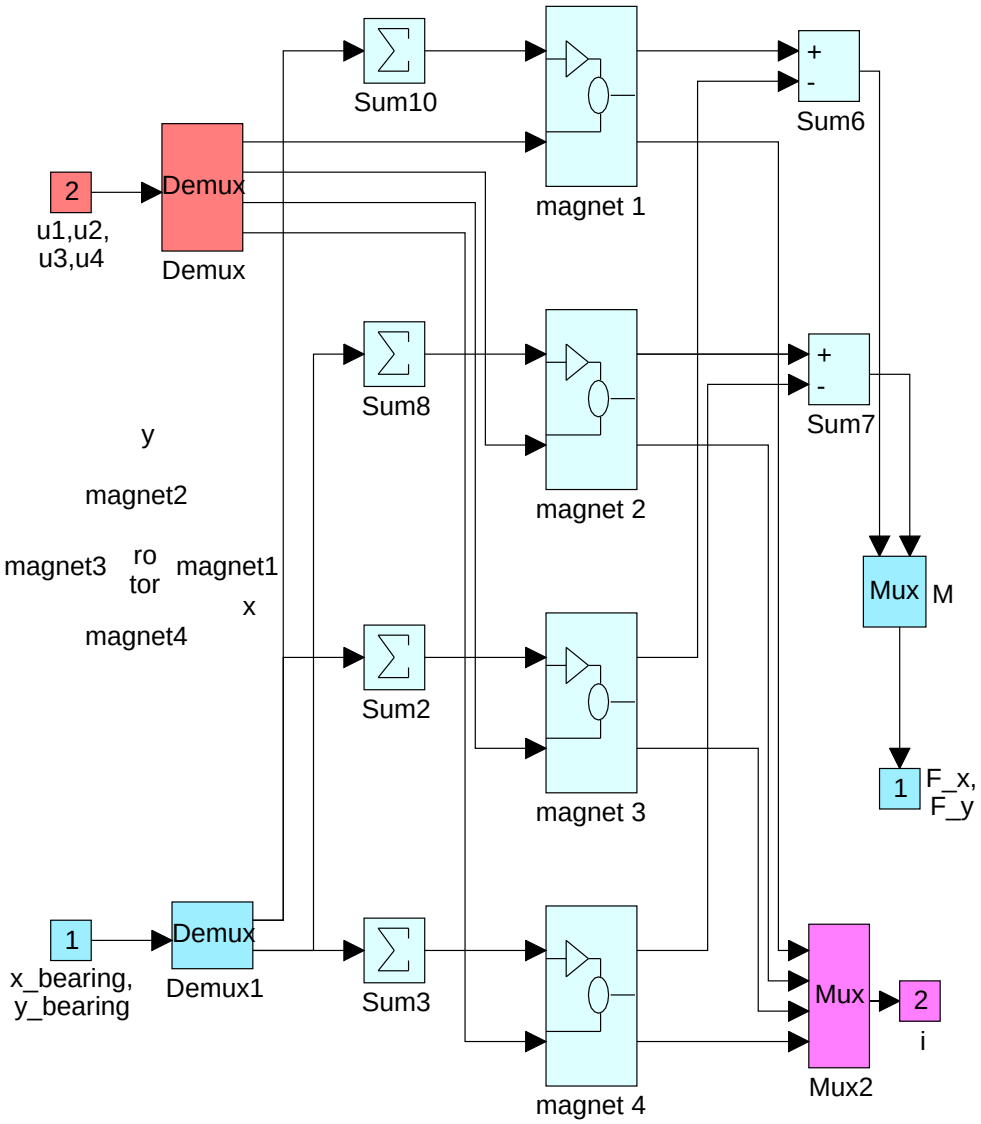

Figure D.4: An active magnetic bearing with four magnets.

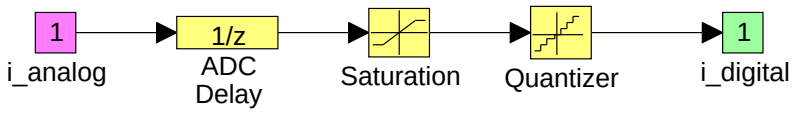

Figure D.5: One ADC for the current sampling as an example for all 8 channels.

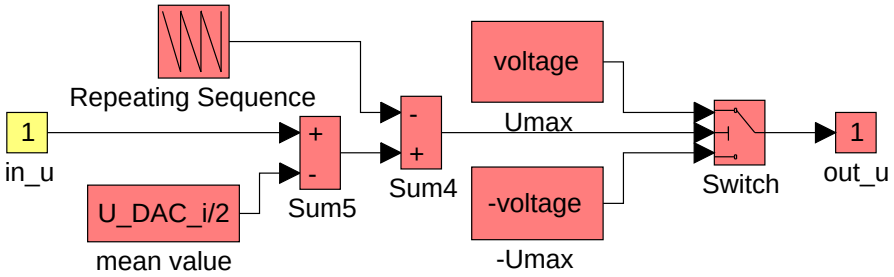

Figure D.6: A sample PWM with switching amplifier.

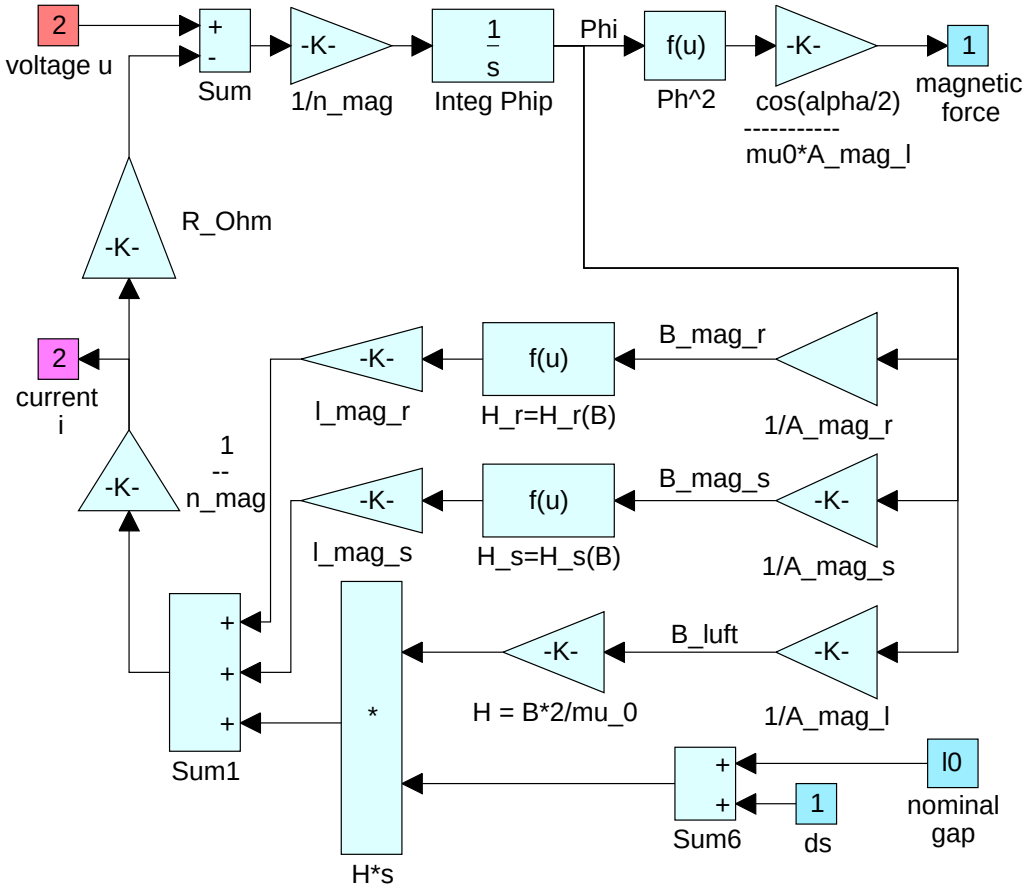

Figure D.7: The SIMULINK implementation of a magnet.

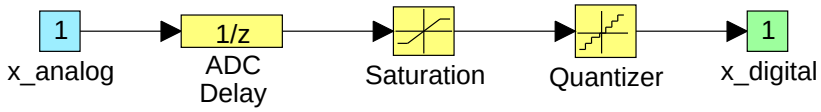

Figure D.8: One ADC for the position sampling as example for all 8 channels.

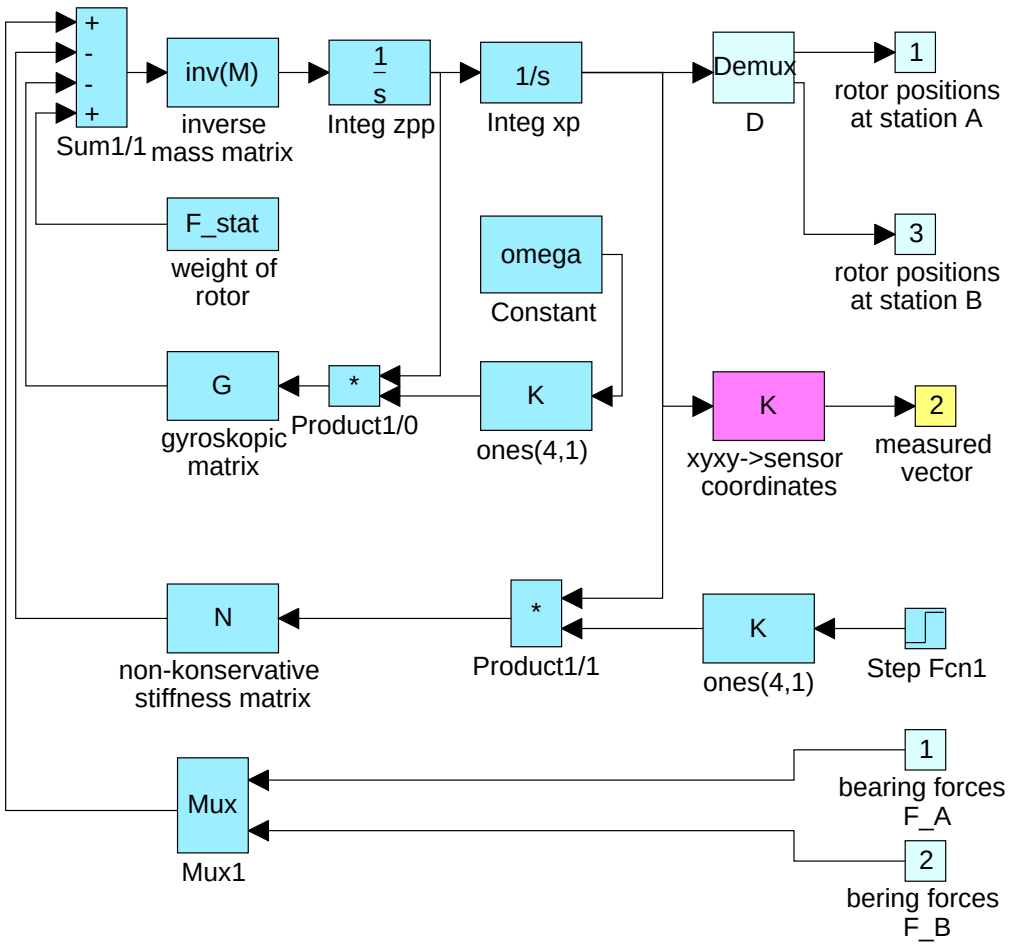

Figure D.9: SIMULINK model of the rigid rotor.

D.2 MATLAB code

All computations necessary for the model identification adaptive control algorithm were coded in the MATLAB programming language for the reasons mentioned above. Of course, the resulting code is not very efficient with respect to computation time and DSP resources, because the readability is more important during the developing stage and computation time is no issue within a simulation. For a real implementation in C or assembler code, the presented code has to be optimised.

D.2.1 Data file

The following data file contains all information necessary for a simulation run. Within this file the current controller is defined, the coefficients for the active magnetic bearing are calculated and the linear state space model is derived in controller canonical form.

```
%
%       data.m
%
%       date file used for rotor simulation

%
%       general
%
fprintf('\n\n Loading data.m');
%
IDACflag   = 1;    % flag for adaptive control
integflag  = 0;    % flag for integrative feedback
excitflag  = 1;    % flag for excitation
loadflag   = 0;    % flag for additional load
weightflag = 1;    % flag for rotor weight
rotflag    = 1;    % flag for rotation
setflag    = 0;    % flag for set point

t0    = 0;         % [s] simulation start time
tterm = 1E-2;      % [s] simulation termination time
tstep = 1E-1;      % [s] time of step

%
% Integration
%
Tint  = 1E-6;      % [s] integration time
Tcomp = 4E-6;      % [s] computation delay
Tsamp = 1E-4;      % [s] sample time

%
% globals
%
global invTl2s R_max n_in n_out order index k_m % model
global Ahat Bhat Chat Ac0 Bc0 Cc0               % estimated model
global Kt Kw Ki Khat                            % controller
global U Y Xhat Yhat Ui u_act e_act epsilon     % controller states
```

```
global par d_par Dp0 Dp Pp Up Dpn Upn Psi Wk    % estimation algorithm
global k_rho rho rho_0 rho_inf maxnoise         % forgetting factor control
global sigmaN_1 sigmaN_2 delta N_1 N_2          % forgetting factor control
global threshold minvar maxvar                  % forgetting factor control
global trigger Dp0
global i0A i0B u0A u0B nn_i U_DAC_i n_mag       % current controller
global I_max U_max u_act e_act                  % current controller
global IDACflag        integflag setflag        % general
global U_s X_s Y_s                              % save variables
global sigma_s delta_s rho_s                    % save variables
global par_s det_s Kt_s Ki_s                    % save variables
global u_s i_s t_s t0 tterm tstep               % save variables

%
%        system modelling
%
% electro mechanical part
%
mu_0    = 4*pi*1E-7;     % [Vs/Am] permeability iv vacuum
mu_r_s  = 2000;          % [-]     relative stator permeability
mu_r_r  = 1000;          % [-]     relative rotor permeability
l_mag_s = 84*1E-3;       % [m]     length of magnetic path in stator
l_mag_r = 20*1E-3;       % [m]     length of magnetic path in rotor
A_mag_s = 700*1E-6;      % [m^2]   cross section area of length of
%                            magnetic path in stator
A_mag_r = 420*1E-6;      % [m^2]   cross section area of length of
%                            magnetic path in rotor
B_max_s = 1.3;           % [T]     maximal induction in stator
B_max_r = 1.95;          % [T]     maximal induction in rotor
A_mag_l = A_mag_s;       % [m^2]   cross section area of length of
%                            magnetic path in air
alpha   = pi/4;          % [rad]   angle between pole shoes
R_Ohm   = 0.8;           % [Ohm]   ohmic resistance
N_mag   = 2*65;          % [-]     number of windings = 2 poles
n_mag   = 4;             % [-]     number of magnets
l0      = 0.5*1E-3;      % [m]     nominal air gap
i00     = 4;             % [A]     bias current
u0      = i00*R_Ohm;     % [V]     bias voltage
I_max   = 8;             % [A]     maximum current

%
% constants for active magnetic bearing
% according to chapter 2
%
R_mag0   = ( l_mag_s/(mu_r_s*A_mag_s) ...
         + l_mag_r/(mu_r_r*A_mag_r) ...
         + 2*l0/A_mag_s)/mu_0;          % magnetic reluctance
L_mag0   = N_mag^2/R_mag0;              % inductivity
Phi00    = N_mag*i00/R_mag0;            % magnetic flux for bias current
Kphipphi =  R_Ohm*R_mag0/(N_mag^2);
Kphipl   = -2*R_Ohm*Phi00/((N_mag^2)*mu_0*A_mag_l);
Kphipu   =  1/N_mag;
KFphi    =  2*Phi00*cos(alpha/2)/(mu_0*A_mag_l);
Kiphi    =  R_mag0/N_mag;
Kil      =  2*Phi00/(N_mag*mu_0*A_mag_l);
```

```
KiFc     =  2*cos(alpha/2)*Phi00*N_mag/(A_mag_l*mu_0*R_mag0);
KlFc     = -4*cos(alpha/2)/R_mag0*(Phi00/(A_mag_l*mu_0))^2;

K_i = 2*KiFc;           % current stiffness
K_s = 2*KlFc;           % position stiffness

%
%  switching amplifier and current controller
%
TsADC_i  = 3E-6;               % [s] conversion delay for ADC for current
TsDAC_i  = 3E-6;               % [s] conversion delay for DAC for voltage
Tswitch  = 1/60000;            % [s] switching time
Kswitch  = 1;                  % [-] amplification
ADCbit_i =  9;                 % [-] bit-resolution for ADC for current
DACbit_i = 11;                 % [-] bit-resolution for DAC for voltage
U_ADC_i  = 8;                  % [A] input range for current
U_DAC_i  = 5;                  % [V] output range for voltage
voltage  = 75;                 % [V] switching voltage
K_a      = 2*voltage/U_DAC_i;
n_m      = Kiphi*Kphipu;
d_m      = [1 Kphipphi];
K_c      = 3.5;
n_i      = K_c*[1 Kphipphi];     % numerator for PI-controller
d_i      = [1 0];                % denominator for PI-controller
[nn_i,dd_i] = ...
c2dm(n_i,d_i,Tsamp,'tustin');    % discrete time PI controller
U_max    = 75;                  % maximum voltage for anti windup

n_s = K_a*R_mag0/N_mag^2;
d_s = [1 0];
n_h0 = 1;
d_h0 = [Tsamp/2 1];
n_o = n_s*n_h0;
d_o = conv(d_s,d_h0);
T_tot = Tsamp;

%
% measurement in mu-meter
%
k_m      = 1E6;         % [-]   measurement gain
R_max    = 250E-6*k_m;  % [mum]   Maximum displacement in mu-meter
ADCbit_x = 10;          % [-]   bit-resolution for ADC for position
TsADC_x  = 3E-6;        % [s]   conversion delay for ADC for position
maxnoise = 1;           % [mum]   measurement noise in mu-meter

%
% mechanical part
%
m_r =  28.768;          % [kg]    rotor mass
Iar =  0.8632;          % [kgm2]  axial moment of inertia for rotor
Ipr =  0.02188;         % [kgm2]  polar moment of inertia for rotor
a_s =  0.23877;         % [m]   distance to bearing A from centre of gravity
b_s = -0.24123;         % [m]   distance to bearing B from centre of gravity
c_s =  0.18977;         % [m]   distance to sensor A from centre of gravity
d_s = -0.19233;         % [m]   distance to sensor B from centre of gravity
```

```
if rotflag == 1,
   omega = 20000*2*pi/60;         % [rad/s] rotor speed
else
   omega = 0;                     % [rad/s]
end

%
% rotor weight
%
g   = 9.81;
mg0 = m_r*g;                                  % [kg] rotor weight
if weightflag == 1,
   F_stat =  -mg0/(a_s-b_s)*[0 -b_s 0 a_s];   % [N]  static force
else
   F_stat =  [0 0 0 0];                       % [N]  static force
end
%
if loadflag == 1,
   F_load =  [100 0 100 0];                   % [N]  disturbance force
else
   F_load =  [0 0 0 0];                       % [N]  disturbance force zero
end

%
% step of non conservative stiffness parameter
%
if excitflag == 1,
   excitation = 1E7;   % [N/m] non conservative stiffness parameter
else
   excitation = 0;     % [N/m] non conservative stiffness parameter = zero
end
n  =  -0.1;            % [m]   distance from centre of gravity
n_0 = 0;               % [N/m] quantity for design

%
% system matrices
%
Ks0 = diag([K_s,K_s,K_s,K_s]);  % position stiffness

Ki0 = diag([K_i,K_i,K_i,K_i]);  % current stiffness

M0  = diag([Iar,m_r,Iar,m_r]);  % mass matrix in inertia coordinates

G0  = [ 0  0 Ipr 0;             % gyroscopic matrix in inertia coordinates
        0  0  0  0;
      -Ipr 0  0  0;
        0  0  0  0];

Nn  = [  0    -1   ;            % matrix of non conservative stiffness
         1     0  ];            % at plane N'

%
% transformation matrices
%
Tl = [a_s 1  0  0;              % x(B) = Tl x(0)T
```

```
       0  0 a_s 1;              %
      b_s 1  0  0;              %
       0  0 b_s 1];
%
Ts = [c_s 1  0  0;              % x(S) = Tl x(0)T
       0  0 c_s 1;              %
      d_s 1  0  0;              %
       0  0 d_s 1];
%

Tn0n = [n 1 0 0;                %
        0 0 n 1];               % x(N) = Tn0n x(0)
                                % f(0) = Tnn0 f(N) = Tn0n' f(N)
Tn0nT = Tn0n';

%
invTl  = inv(Tl);
invTs  = inv(Ts);
invTlT = inv(Tl)';
Tl2s   = Ts*invTl;              % x(S) = Tl2s x(B)
                                %
invTl2s = inv(Tl2s);            %

M0    = invTl'*M0*invTl;        % mass matrix in bearing coordinates
invM0 = inv(M0);                % inverse mass matrix in bearing coordinates

G0 = invTl'*G0*invTl;           % gyroscopic matrix in bearing coordinates

N1 = Tn0n'*Nn*Tn0n;             % matrix of non conservative stiffness
N0 = invTl'*N1*invTl;           % in bearing coordinates

Omegadesign=0;

A0 = [              zeros(4)                          eye(4);
         -invM0*(Ks0+n_0*N0)   -invM0*G0*Omegadesign];

B0 = [             zeros(4)    ;
             invM0*Ki0   ];

C0 = k_m*[eye(4,4) zeros(4,4)];                    % output in mu-meter !!

D0 = zeros(4,4);

index = [2 2 2 2];
dof = max(size(M0));
order = max(size(A0));
[n_out,n_in] = size(D0);
[A0d,B0d,C0d,D0d] = c2dm(A0,B0,C0,D0,Tsamp,'zoh');  % Discretise
[Ac0,Bc0,Cc0,Tc0] = ctrlform(A0d,B0d,C0d,index);    % Transformation

%
% initial values for simulation
%
X0 = zeros(order,1);                % initial state
i0A = [i00-F_stat(1)/K_i; ...
```

```
       i00-F_stat(2)/K_i; ...
       i00+F_stat(1)/K_i; ...
       i00+F_stat(2)/K_i];          % static current bearing A
i0B = [i00-F_stat(3)/K_i; ...
       i00-F_stat(4)/K_i; ...
       i00+F_stat(3)/K_i; ...
       i00+F_stat(4)/K_i];          % static current bearing B
%
u0A  = i0A*R_Ohm/K_a;               % static voltage bearing A
u0B  = i0B*R_Ohm/K_a;               % static voltage bearing B
Phi0A = N_mag*i0A/R_mag0;           % static flux bearing A
Phi0B = N_mag*i0B/R_mag0;           % static flux bearing B
%
fprintf('\n\n Done. \n\n');
%
```

D.2.2 Estimation algorithm

The state space estimation algorithm and all control algorithms are contained in one function. This function simulates all tasks of a DSP with the measured values for the rotor position and all coil currents as input from the ADCs, and the control voltage for all DACs as output, respectively. This function is called at each sample interval T_s.

```
function [sys, x0] = controller(t,x,u,flag,Tsamp)
%
%       function [sys, x0] = controller(t,x,u,flag,Tsamp)
%
%       model identification state space adaptive control

%
% global definitions
%
global invTl2s R_max n_in n_out order index k_m % model
global Ahat Bhat Chat Ac0 Bc0 Cc0               % estimated model
global Kt Kw Ki Khat                            % controller
global U Y Xhat Yhat Ui u_act e_act epsilon     % controller states
global par d_par Dp Dp0 Up Dpn Upn Psi Wk       % estimation algorithm
global k_rho rho rho_0 rho_inf maxnoise         % forgetting factor control
global sigmaN_1 sigmaN_2 delta N_1 N_2          % forgetting factor control
global threshold minvar maxvar                  % forgetting factor control
global i0A i0B u0A u0B nn_i U_DAC_i n_mag       % current controller
global I_max U_max u_act e_act                  % current controller
global trigger Dp0
global IDACflag        integflag setflag        % general
global U_s X_s Y_s                              % save variables
global sigma_s delta_s rho_s                    % save variables
global par_s det_s Kt_s Ki_s                    % save variables
global u_s i_s t_s t0 tterm tstep               % save variables

%
% flag = 0 --> Return sizes of parameters and initial conditions
%
if flag == 0,
```

```
%
% initialisation of state space controller
%
   Ahat = Ac0;                                  % initial system matrix
   Bhat = Bc0;                                  % initial controller matrix
   Chat = Cc0;                                  % initial measurement matrix
   Khat = obscalc(Ahat,Bhat,Chat,[index],'n'); % initial Kalman matrix
   if integflag == 1
      [Kt,Ki] = ...
      ctrlcalci(Ahat,Bhat,Chat,index,'y');     % initial controller
      Ui = zeros(n_out,1);                      % initial output
   else
      [Kt,Kw] = ...
      ctrlcalc(Ahat,Bhat,Chat,index,'y');      % initial controller
   end
   U = zeros(n_in,1);                           % initial input
   Y = zeros(n_out,1);                          % initial output
   Xhat = zeros(order,1);                       % initial estimated state
   Yhat = Chat*Xhat;                            % initial estimated output
   epsilon = Y - Yhat;                          % initial estimation error
   u_act = [u0A; u0B];                          % initial control voltage
   e_act = zeros(n_mag*2,1);                    % initial current error

%
%  forgetting factor control
%
   N_1 = 16;                            % short memory
   N_2 = 16^2;                          % long memory
   sigmaN_1 = sqrt(2)*maxnoise^2;       % initial variance with short memory
   sigmaN_2 = sqrt(2)*maxnoise^2;
   threshold = 0.2;                     % trigger value
   rho_0 = 0.999;                       % rho_0
   rho_inf = 1;                         % rho infinity
   rho = rho_inf;                       % initial rho
   delta = 0;                           % initial delta
   k_rho = 0.99;

%
%  parameter estimation
%
   par = [
                  Ahat(2,:)';
                  Ahat(4,:)';
                  Ahat(6,:)';
                  Ahat(8,:)';
                  Chat(:);
                  Khat(:);
                  ];                        % initial parameter vector
   d_par = max(size(par));                  % number of parameters
   Dpmin = [
                  ones(32,1)*1E-8;          % initial covariance
                  ones(32,1)*1E-8;          % diagonal matrix replaced
                  ones(32,1)*1E-8;          % by vector
                  ];
```

```
   Dp = diag(Dpmin);
   Dp0 = Dp;
   Up = eye(d_par,d_par);
   Dpn = eye(d_par,d_par);
   Upn = eye(d_par,d_par);
   minvar = 1E-12;                          % minimal covariance
   maxvar = 1E-6;                           % maximal covariance
   Psi = zeros(n_out,d_par)';               % initial gradient
   Wk = zeros(order,d_par);                 % initial Wk

%
% save variables initialisation
%
   n_s = ceil((tterm-t0)/Tsamp);
   U_s = zeros(n_in,n_s);
   X_s = zeros(order,n_s);
   Y_s = zeros(n_out,n_s);
   if IDACflag == 1,
     sigma_s = zeros(2,n_s);
     delta_s = zeros(1,n_s);
     rho_s = zeros(1,n_s);
     par_s = zeros(d_par,n_s);
     det_s = zeros(d_par,n_s);
     Kt_s = zeros(n_in*order,n_s);
     if integflag == 1
         Ki_s = zeros(n_out*n_in,n_s);
     end
   end
   i_s = zeros(n_mag*2,n_s);
   u_s = zeros(n_mag*2,n_s);
   t_s = zeros(1,n_s);

%
%  verbose output
%
   if IDACflag == 1,
      fprintf('\n  Starting estimation with %2.0f parameters \n',d_par);
   end
   if integflag == 1
      fprintf('\n  Starting controller with integrative feedback \n')
   else
      fprintf('\n  Starting state space controller \n')
   end

%
%  simulation
%
   set_param('rotor','Stop time','tterm');        % stop simulation
   sys = [0,1,2*n_mag,n_out+2*n_mag,0,0];
   x0 = [0];
%
%-----------------------------------------------------------------------
% flag = 2 --> real time hit
%
elseif flag == 2,
```

```
        sample_hit = (abs(rem(1E6*t,1E6*Tsamp)) < Tsamp/1E6);
        if sample_hit
%        fprintf('Flag 2 %e\n',t)
        counter = x(1) + 1;

%
%        input of measured data and transformation
%
  Ys = u(1:n_out);                    % read rotor position in sensor
                                      % coordinates in mu-meter
  Y = invTl2s*Ys;                     % transformation to bearing
                                      % coordinates
  if max(abs(Y))>R_max,               % check whether rotor is out of
                                      % range
    set_param('rotor','Stop time',t); % stop simulation
    fprintf('t = %g, ...
        Rotor touched bearing, ...
        emergency stop\n',t)
    return;
  end
  I_ist = u(n_out+1:n_out+2*n_mag);    % read currents for all magnets

%
%        Set point error
%
   if setflag == 1
     if t > tstep,
        W = 1E-6*k_m*[100 100 100 100]';
     else
        W = [0 0 0 0]';
     end
   else
     W = [0 0 0 0]';
   end

%
%        Prediction error
%
  epsilon = Y - Yhat;

%
%  Start estimation algorithm
%

if IDACflag == 1,
%
%        Two estimates for the sum of the variances
%
   sigmaN_1 = (N_1-2)*sigmaN_1/(N_1-1) + ...
                   epsilon'*epsilon/n_out/N_1;     % short memory
   sigmaN_2 = (N_2-2)*sigmaN_2/(N_2-1) + ...
                   epsilon'*epsilon/n_out/N_2;     % long memory

%
```

```
%       Calculation of trigger value deltahat(k)
%
   deltan = (sigmaN_1-sigmaN_2)/sigmaN_2;
   delta = (N_1-2)*delta/(N_1-1) + (deltan-delta)/N_1;

%
%       Compute forgetting factor rho(k)
%
   triggern = sign(delta-threshold);
   if triggern>0,
     if sign(triggern - trigger)>0
       fprintf('t = %1.6f, Parameter change triggered ... \n',t)
%       fprintf('t = %g, Add Dp0 to Dp \n',t)
%       Dp = Dp + Dp0;
     end
     rhon = rho_0;
     fprintf('t = %g, Reset rho to %f\n',t,rhon)
     burst = noise(4,1,100);
    else
     burst = zeros(4,1);
     rhon = k_rho*rho + rho_inf*(1-k_rho);
%     fprintf('t = %1.6f, rho = %f \n',t,rhon)
   end

%
%       covariance update using Bierman's factorisation
%
   for k = 1:n_out,
      f = Up'*Psi(:,k);
      g = Dp*f;
      beta(1) = rhon;
      for j = 1:d_par,
         beta(j+1) = beta(j) + f(j)*g(j);
         Dpn(j,j) = Dp(j,j)*beta(j)/beta(j+1)/rhon;
         Dpn(j) = max([minvar  min([ Dpn(j) maxvar]) ]);
         mu(j) = -f(j)/beta(j);
         nu(j) = g(j);
         for i = 1:j-1,
            Upn(i,j) = Up(i,j) + nu(i)*mu(j);
            nu(i) = nu(i) + Up(i,j)*nu(j);
         end
      end
      L(:,k) = (nu/beta(d_par+1))';
      Dp = Dpn;
      Up = Upn;
   end

%
%       Model update
%
   parn = par + L*epsilon;
   Ahat(2,:) = parn(1:8)';
   Ahat(4,:) = parn(9:16)';
   Ahat(6,:) = parn(17:24)';
   Ahat(8,:) = parn(25:32)';
```

```
   Chat(:)   = parn(33:64)';
   Khat(:)   = parn(65:96)';

%
%       Stability check of predictor
%
   eigenvalues = abs(eig(Ahat -Khat*Chat));
   evil = find(eigenvalues>=1);
   if evil == [],                % no eigenvalue outside the unit circle
      par = parn;                % parameter update
%      Dp = Dpn;
   else
      fprintf('t = %g, Predictor unstable, no update\n',t)
      Ahat(2,:) = par(1:8)';
      Ahat(4,:) = par(9:16)';
      Ahat(6,:) = par(17:24)';
      Ahat(8,:) = par(25:32)';
      Chat(:)   = par(33:64)';
      Khat(:)   = par(65:96)';
   end

%
%       New controller
%
   if integflag == 1
     [Kt,Ki] = ctrlcalci(Ahat,Bhat,Chat,index,'y');
   else
     [Kt,Kw] = ctrlcalc(Ahat,Bhat,Chat,index,'y');
   end

%
%       State space controller with or without integrative feedback
%
   if integflag == 1
     E    =  W + burst - Y;
     U    = -Kt*Xhat + Ui;
     Uin  = Ui + Ki*E;
   else
     U    = -Kt*Xhat + Kw*(W + burst);
   end

%
%       New state estimation
%
  Xhatn = Ahat*Xhat + Bhat*U + Khat*epsilon;
  Yhatn = Chat*Xhatn;

%
%       New Gradient Matrix
%
   [Mk,Vk] = MkVk_ACK(Xhat,epsilon);          % Mk(x(k)) and Vk(x(k))
   Wkn = (Ahat-Khat*Chat)*Wk + Mk + Khat*Vk;  % Wk(k+1)
   [Mkn,Vkn] = MkVk_ACK(Xhatn,epsilon);       % Mk(x(k+1)) and Vk(x(k+1))
   Psin = Wkn'*Chat' + Vkn';                  % Psi(k+1)
```

```
%
% end estimation algorithm

else        % no adaptive control

%
%        State space controller with or without integrative feedback
%
   if integflag == 1
     E    = W - Y;
     U    = -Kt*Xhat + Ui;
     Uin  = Ui + Ki*E;
   else
     U    = -Kt*Xhat + Kw*W;
   end

%
%       State estimation for state space controller
%
  Xhatn = Ahat*Xhat + Bhat*U + Khat*epsilon;
  Yhatn = Chat*Xhatn;


end         % if IDACflag == 1,

%
%  Necessary for step response
%
%if t < 10*Tsamp,
%  U = zeros(4,1);
%else
%  U = [0.25 0.5 0.75 1]';
%end

%
%  Necessary for frequency response
%
%U = [0.25 0.5 0.75 1]'*sin(2*pi*1000*t);

%
% setpoints for each current control loop
%
   I_sit(1,1) = i0A(1)+U(1);
   I_sit(2,1) = i0A(2)+U(2);
   I_sit(3,1) = i0A(3)-U(1);
   I_sit(4,1) = i0A(4)-U(2);
   I_sit(5,1) = i0B(1)+U(3);
   I_sit(6,1) = i0B(2)+U(4);
   I_sit(7,1) = i0B(3)-U(3);
   I_sit(8,1) = i0B(4)-U(4);

%
%  Necessary for step response
%
%if t < 10*Tsamp,
```

```
%  I_sit = 4*ones(8,1);
%else
%  I_sit = [4.25 4.5 5 5 3 3.5 3.9 4]';
%end

%
%  Necessary for frequency response
%
%  I_sit = 4*ones(8,1) + [0.25 0.5 1 1 1 0.5 0.9 1]'*sin(2*pi*10*t);

%
%       PI-current controller
%
   for k = 1:2*n_mag,
       I_sit(k,1) = max([ 0  min([ I_max  I_sit(k,1) ]) ]);
                               % Saturation i
       e_new(k,1) = I_sit(k,1)-I_ist(k,1);
       u_new(k,1) = u_act(k,1) + nn_i(1)*e_new(k,1) + ...
                               nn_i(2)*e_act(k,1);        % PI-controller
       u_new(k,1) = max([ -U_max  min([ U_max  u_new(k,1) ]) ]);
                               % Anti windup
   end

%
% memory shift
%
   Xhat  = Xhatn;          % estimated rotor states
   Yhat  = Yhatn;          % estimated rotor states
   u_act = u_new;          % old control voltage
   e_act = e_new;          % old current error
   if integflag == 1
     Ui  = Uin;
   end
   Wk = Wkn;
   Psi = Psin;
   trigger = triggern;
   rho = rhon;

%
% save variables
%
   U_s(:,counter) = U;
   X_s(:,counter) = Xhat;
   Y_s(:,counter) = Y;
   if IDACflag == 1,
      sigma_s(:,counter) = [sigmaN_1; sigmaN_2];
      delta_s(:,counter) = delta;
      rho_s(:,counter) = rho;

      par_s(:,counter) = par;
      det_s(:,counter) = diag(Dp);
      Kt_s(:,counter) = Kt(:);
      if integflag == 1
          Ki_s(:,counter) = Ki(:);
      end
```

```
   end
   i_s(:,counter) = I_ist;
   u_s(:,counter) = u_new;
   t_s(:,counter) = t;

%
% simulink system
%
   sys = [counter];

%       end of sample_hit loop
        else
                sys = x;
        end

%
%-----------------------------------------------------------------------
% flag = 3 --> Return state vector
%
elseif flag == 3
%
   sys = [u_act+U_DAC_i/2];        % shift voltage to DAC range

%
%-----------------------------------------------------------------------
% flag = 4 --> Return next sample hit
%
elseif flag == 4
%
        sys=ceil(t/Tsamp+Tsamp/1e8)*Tsamp;
%
end
```

D.2.3 Controller calculation

The automated controller computation based on the identified model is carried out in a special function for a controller with or without an integrative feedback.

D.2.4 Controller calculation for P-structure

```
function [Kt,Kw] = ctrlcalc(A,B,C,N,flag);
%
%       function [Kt,Kw] = ctrlcalc(A,B,C,N,flag);
%
%       State space controller calculation,
%       A,B,C must be in controller canonical form
%       If flag == 'n', the system is transformed
%       N is the vector of controller index N = [n1 n2 ...]

%
%       Poles at z1/2 = 0.9691,  or
%             at s1/2 = -313.5,  (corresponds to sqrt(K_s/(m_r/2)))
```

```
%
z = 0.9691;
P = [1 -2*z z^2];
mui = 2;                % ni = 2
%
if flag == 'n',
  [A,B,C,T] = ctrlform(A,B,C,N);
end
N_N = max(size(N));     % number of second order sub-systems
%
for k = 1:N_N,
  mui = N(k);
  ii  = sum(N(1:k))-N(k)+1;
  iii = sum(N(1:k));
  XX = [ zeros(mui-1,mui-1)   -eye(mui-1,mui-1)  ; ...
                      P(mui+1:-1:2)              ];
  IABk(ii:iii,ii:iii) = XX + eye(mui,mui);
  %
  for l = 1:N_N,
    muj = N(l);
    jj  = sum(N(1:l))-muj+1;
    jjj = sum(N(1:l));
    if l == k,
      Kt(k,jj:jjj) = P(muj+1:-1:2) + A(iii,jj:jjj);
    else
      Kt(k,jj:jjj) = A(iii,jj:jjj);
    end
  end
  %
end
%
Kw = inv(C*inv(IABk)*B);
if flag == 'n',
  Kt = Kt*inv(T);
end
```

D.2.5 Controller calculation for PI-structure

```
function [Kt,Ki] = ctrlcalci(A,B,C,N,flag);
%
%       function [Kt,Ki] = ctrlcalci(A,B,C,N,flag);
%
%       State space controller calculation with additional
%       integrative feedback,
%       A,B,C must be in controller canonical form
%       If flag == 'n', the system is transformed
%       N is the vector of controller index N = [n1 n2 ...]

%
%       Poles at z1/2/3 = 0.9691,  or
%             at s1/2/3 = -313.5,  (corresponds to sqrt(K_s/(m_r/2)))
%
z = 0.9691;
Ps = [1 -3*z 3*z^2 -z^3];
```

```
ps1 = Ps(2);
ps2 = Ps(3);
ps3 = Ps(4);
mui = 2;                % ni = 2
%
if flag == 'n',
  [A,B,C,T] = ctrlform(A,B,C,N);
end
N_N = max(size(N));               % number of second order sub-systems
%
Sr = zeros(2*N_N,N_N);
for k = 1:N_N,
  Sr(2*k-1:2*k,k) = [1 1]';
end
N01 = C*Sr;                       % N0(z=1)
Ps1 = sum(Ps)*eye(N_N,N_N);       % P_Tilde(z=1)
%
Ki = Ps1*inv(N01);                % Gain Ki
%
for k = 1:N_N,
  mui = N(k);
  p1 = ps1 + 1;
  p2 = Ki(k,:)*C(:,2*k-1) - ps3;
  P = [1 p1 p2];
  ii  = sum(N(1:k))-N(k)+1;
  iii = sum(N(1:k));
  %
  for l = 1:N_N,
    muj = N(l);
    jj  = sum(N(1:l))-muj+1;
    jjj = sum(N(1:l));
    if l == k,
      Kt(k,jj:jjj) = P(muj+1:-1:2) + A(iii,jj:jjj);
    else
      Kt(k,jj:jjj) = A(iii,jj:jjj);
    end
  end
  %
end
%
if flag == 'n',
  Kt = Kt*inv(T);
end
```

D.2.6 Helper applications

In order to compute the state space representation in controller canonical form, the transformation derived in Appendix A was implemented in a MATLAB function. The same is true for the observer canonical form.

Transformation to controller canonical form

```
function [Ac, Bc, Cc, Gamma] = ctrlform(A,B,C,N)
%
%       function [Ac, Bc, Cc, Gamma] = ctrlform(A,B,C,N)
%
%       Transformation into controller canonical form
%       system matrices A,B,C and structural indices N = [n1,n2 ..]

%
%       general definitions
%
m = max(size(N));
n = max(size(A));
b = zeros(m,n);
Tc = [];
for k = 1:m,
  b(k,sum(N(1:k))) = 1;
  for l = 1:N(k),
    Tc = [Tc, A^(l-1)*B(:,k)];
  end
end
%
Gamma = [];
for k = 1:m,
  gamma = inv(Tc')*b(k,:)';
  for l = 1:N(k),
    Gamma = [Gamma; gamma'*A^(l-1)];
  end
end
%
Ac = Gamma*A*inv(Gamma);
Bc = Gamma*B;
Cc = C*inv(Gamma);
```

Transformation to observer cannonical form

```
function [Ao, Bo, Co, Gamma] = obsform(A,B,C,N)
%
%       function [Ao, Bo, Co, Gamma] = obsform(A,B,C,N)
%
%       Transformation into observer canonical form
%       system matrices A,B,C and structural indices N = [n1,n2 ..]

%
%       general definitions
%
m = max(size(N));
n = max(size(A));
c = zeros(m,n);
To = [];
for k = 1:m,
  c(k,sum(N(1:k))) = 1;
  for l = 1:N(k),
```

```
    To = [To; C(k,:)*A^(l-1)];
  end
end
%
Gamma = [];
for k = 1:m,
  gamma = inv(To)*c(k,:)';
  for l = 1:N(k),
    Gamma = [Gamma, A^(l-1)*gamma];
  end
end
%
Ao = inv(Gamma)*A*Gamma;
Bo = inv(Gamma)*B;
Co = C*Gamma;
```

Computation of an observer

```
function K = obscalc(A,B,C,N,flag);
%
%        function K = obscalc(A,B,C,N,flag)
%
%        State space observer calculation,
%        A,B,C must be in observer canonical form
%        If flag == 'n', the system is transformed
%        N is the vector of controller N = [n1 n2 ...]

%
%        general definitions
%
z = 0.90;                % all poles equal
P = [1 -2*z z^2];
muj = 2;                % nj = 2
%
if flag == 'n',
  [A,B,C,T] = obsform(A,B,C,N);
end
N_N = max(size(N));
%
for k = 1:N_N,
  muj = N(k);
  jj  = sum(N(1:k))-N(k)+1;
  jjj = sum(N(1:k));
  %
  for l = 1:N_N,
    mui = N(l);
    ii  = sum(N(1:l))-mui+1;
    iii = sum(N(1:l));
    if l == k,
      K(ii:iii,k) = P(mui+1:-1:2)' + A(ii:iii,jjj);
    else
      K(ii:iii,k) = A(ii:iii,jjj);
    end
  end
```

```
  %
end
%
if flag == 'n',
  K = T*K;
end
```

Computation of the matrices $\mathbf{M}_k$ and $\mathbf{V}_k$

Note that the following implementation is not very efficient, but easy. In a real implementation this problem would be solved in a more sophisticated way.

```
function [Mk, Vk] = MkVk(Xhat,epsilon)
%
%       function MkVk(Xhat,epsilon)
%
%       Compute Mk and Vk depending on Xhat and epsilon

Mk = [ zeros(1,8)   zeros(1,8)   zeros(1,8)   zeros(1,8)  zeros(1,32) ...
       epsilon'     zeros(1,28)  ; ...
         Xhat'      zeros(1,8)   zeros(1,8)   zeros(1,8)  zeros(1,32) ...
       zeros(1,4)    epsilon'    zeros(1,24)  ; ...
       zeros(1,8)   zeros(1,8)   zeros(1,8)   zeros(1,8)  zeros(1,32) ...
       zeros(1,8)    epsilon'    zeros(1,20)  ; ...
       zeros(1,8)     Xhat'      zeros(1,8)   zeros(1,8)  zeros(1,32) ...
       zeros(1,12)   epsilon'    zeros(1,16)  ; ...
       zeros(1,8)   zeros(1,8)   zeros(1,8)   zeros(1,8)  zeros(1,32) ...
       zeros(1,16)   epsilon'    zeros(1,12)  ; ...
       zeros(1,8)   zeros(1,8)     Xhat'      zeros(1,8)  zeros(1,32) ...
       zeros(1,20)   epsilon'    zeros(1,8)   ; ...
       zeros(1,8)   zeros(1,8)   zeros(1,8)   zeros(1,8)  zeros(1,32) ...
       zeros(1,24)   epsilon'    zeros(1,4)   ; ...
       zeros(1,8)   zeros(1,8)   zeros(1,8)     Xhat'     zeros(1,32) ...
       zeros(1,28)   epsilon'    ];
%
Vk = [ zeros(1,32)    Xhat'     zeros(1,8)  zeros(1,8)  zeros(1,8)  ...
       zeros(1,32)  ; ...
       zeros(1,32)  zeros(1,8)    Xhat'     zeros(1,8)  zeros(1,8)  ...
       zeros(1,32)  ; ...
       zeros(1,32)  zeros(1,8)  zeros(1,8)    Xhat'zeros(1,8)  ...
       zeros(1,32)  ; ...
       zeros(1,32)  zeros(1,8)  zeros(1,8)  zeros(1,8)    Xhat'     ...
       zeros(1,32)  ];
%
```