Supplementary Material

S1. Probability distributions

In this section we summarize some basic properties of the multivariate normal distribution, which is used many times in the development of the variational Bayesian treatment of ERP-DCM, and the log-normal distribution, which is used as marginal distribution of the error variance parameter in our formulation.

The multivariate normal distribution

For a random vector $x \in \mathbb{R}^d$, the multivariate normal distribution is defined by means of the probability density function

$$N(x;\mu,\Sigma) \coloneqq (2\pi)^{-\frac{d}{2}}|\Sigma|^{-\frac{1}{2}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right), \tag{1}$$

where $|\cdot|$ denotes the matrix determinant. The vector $\mu \in \mathbb{R}^d$ and the positive-definite matrix $\Sigma \in \mathbb{R}^{d\times d}$ are referred to as expectation and covariance parameters of the multivariate normal distribution and correspond to its first moment (expectation) and second-order central moment (covariance). The entropy of the multivariate normal distribution is given in terms of the covariance parameter as

$$\mathcal{H}\big(N(x;\mu,\Sigma)\big) = \frac{1}{2}\ln|\Sigma| + \frac{d}{2}\ln(2\pi e). \tag{2}$$

The log-normal distribution

For a scalar random variable x taking on strictly positive values the log-normal distribution (Johnson, Kotz, & Balakrishnan, 1994) is defined by means of the probability density function

$$LN(x;\mu,\sigma^2) \coloneqq (2\pi\sigma^2)^{-\frac{1}{2}}x^{-1}\exp\left(-\frac{1}{2\sigma^2}(\ln x-\mu)^2\right). \tag{3}$$

The parameters $\mu \in \mathbb{R}$ and $\sigma^2 > 0$ are commonly referred to as log-scale and shape parameters, respectively. The expectation and variance of x are given in terms of the parameters by

$$E(x) = \exp\left(\mu+\frac{1}{2}\sigma^2\right) \text{ and } V(x) = \exp(\sigma^2-1)\exp(2\mu+\sigma^2), \tag{4}$$

for $s \in \mathbb{Z}$, the sth moment of x is given by

$$E(x^s) = \exp\left(s\mu+\frac{1}{2}s^2\sigma^2\right), \tag{5}$$

and its entropy is given by

$$\mathcal{H}\big(LN(x;\mu,\sigma^2)\big) = \frac{1}{2}+\frac{1}{2}\ln(2\pi\sigma^2)+\mu \tag{6}$$

The log-normal distribution is defined such, that if the random variable x is distributed according to $LN(x;\mu,\sigma^2)$, then the random variable $\ln x$ is distributed according to the normal distribution $N(\ln(x);\mu,\sigma^2)$.

S2. Univariate delay differential equation conversion

To make the discussion of the conversion of delay differential equations to ordinary differential equations more transparent, we here consider the case of scalar-valued functions

$$x\colon \mathbb{R}\to\mathbb{R}, t\mapsto x(t) \text{ and } \varphi\colon \mathbb{R}\to\mathbb{R}, x\mapsto \varphi(x) \tag{1}$$

defining the one-dimensional delay differential equation

$$\dot{x}(t)=\varphi\big(x(t-\tau)\big) \tag{2}$$

with a given delay parameter $\tau\in\mathbb{R}$. To approximate (1) by an ordinary differential equation, we consider a first-order Taylor approximation of the concatenated function

$$(\varphi\circ x)(t)\colon \mathbb{R}\to\mathbb{R}, t\mapsto(\varphi\circ x)(t)\coloneqq\varphi\big(x(t)\big). \tag{3}$$

A first-order Taylor approximation of $\varphi\circ x$ at $t-\tau$ with expansion point t is given by

$$\varphi\big(x(t-\tau)\big)\approx\varphi\big(x(t)\big)-\left(\frac{d}{dt}\varphi\big(x(t)\big)\right)\tau=\varphi\big(x(t)\big)-\left(\frac{d}{dx}\varphi\big(x(t)\big)\frac{d}{dt}x(t)\right)\tau \tag{4}$$

where the equality follows with the chain rule of differentiation. With (2), we thus have

$$\dot{x}(t)\approx\varphi\big(x(t)\big)-\tau\frac{d}{dx}\varphi\big(x(t)\big)\dot{x}(t) \tag{5}$$

By treating the approximation in (5) as equality and rearranging, we obtain

$$\dot{x}(t)\left(1+\tau\frac{d}{dx}\varphi\big(x(t)\big)\right)=\varphi\big(x(t)\big)\Rightarrow\dot{x}(t)=\left(1+\tau\frac{d}{dx}\varphi\big(x(t)\big)\right)^{-1}\varphi\big(x(t)\big). \tag{6}$$

By defining

$$Q\coloneqq\left(1+\tau\frac{d}{dx}\varphi\big(x(t)\big)\right) \tag{7}$$

we can express (6) succinctly as

$$\dot{x}(t)=Q^{-1}\varphi\big(x(t)\big) \tag{8}$$

Finally, by defining

$$\psi:\mathbb{R}\to\mathbb{R}, x\mapsto\psi(x)\coloneqq Q^{-1}\varphi(x) \tag{9}$$

we have approximated the delay differential equation (2) by the ordinary differential equation

$$\dot{x}(t)=\psi(x(t)). \tag{10}$$

S3. Simulation Parameter Values

The parameter values of the table below were used for the simulation of the data discussed in Sections 4.2 and 4.3.

Interpretation	Parameter Value
Input function parameters	$\rho = (0.16, -0.22)^T$
Activation function parameters	$s = (-0.0009, 0.08)^T$
Forward Connectivity	$A^F = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$
Backward Connectivity	$A^B = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
Lateral Connectivity	$A^L = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
Input connectivity	$C \coloneqq (1,0)^T$
Excitatory Time Constants	$\tau_{ex} = (0.16,0.07)^T$
Inhibitory Time Constants	$\tau_{in} = (0.13, -0.82)^T$
Excitatory Receptor Density	$H_{ex} = (-0.11, 0.02)^T$
Inhibitory Receptor Density	$H_{in} = (-0.07, -0.17)^T$
Intrinsic Coupling	$\gamma = (-0.02, -0.07, 0.18, -0.17)^T$
Between-source delays	$\Delta = \begin{pmatrix} 0 & -0.52 \\ -0.6 & 0 \end{pmatrix}$
Dipole Positions	$\theta_p = (42, -31, 58, 54, -22, 18)^T$
Dipole Moments	$\theta_p = (0.02, -0.05, -0.04, 0.17, -0.50, -0.17)^T$
Neural State Projection	$\theta_j = (-0.1,0,0,0,0,0, -0.46,0,1)$

S4. Experimental Data Application Parameter Values

The parameter values of the table below were used for in the analysis of the experimental data reported in Section 4.3.

Interpretation	Parameter Value
Input function parameters	$\rho = (0,0)^T$
Activation function parameters	$s = (0,0)^T$
Forward Connectivity	$A^F = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$
Backward Connectivity	$A^B = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$
Lateral Connectivity	$A^L = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
Excitatory Time Constants	$\tau_{ex} = (0,0)^T$
Inhibitory Time Constants	$\tau_{in} = (0,0)^T$
Excitatory Receptor Density	$H_{ex} = (0,0)^T$
Inhibitory Receptor Density	$H_{in} = (0,0)^T$
Intrinsic Coupling	$\gamma = (0,0,0,0)^T$
Between-source delays	$\Delta = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
Dipole Positions	$\theta_p = (42, -31, 58, 54, -22, 18)^T$
Dipole Moments	$\theta_p = (10.6, -5.7, 5.4, 20.7, -24.5, -3.6)^T$
Neural State Projection	$\theta_j = (-0.29,0,0,0,0,0, -0.04,0,1)$

S5. Matlab Code

PDDE_1.m – Forward model and delay parameter matrix visualization (Figures 1 and 2)

```
function pdde_1_R1

% This function visualizes basic aspects of the ERP-DCM latent and
% forward model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
clc
close all

% invoke Fieldtrip
addpath(genpath(fullfile(pwd, 'Fieldtrip')))

% -------------------------------------------------------------------------
%                           Model Formulation
% -------------------------------------------------------------------------

% neural model structure and parameters
% -------------------------------------------------------------------------
% structural aspects and fixed parameters
m_f.f               = @spm_fx_erp;                                      % neural evolution function
m_f.ns              = 200;                                              % number of time bins
m_f.dt              = 0.001;                                            % time bin length (sec)
m_f.pst             = (0:m_f.ns-1)*m_f.dt;                              % peristimulus time (msec)
m_f.nt              = length(m_f.pst);                                  % number of time points
m_f.n               = 2;                                                % number of neural masses
m_f.m               = 9;                                                % number of neural states per neural mass
m_f.ons             = -10;                                              % system input onset (ms)
m_f.dur             = 1;                                                % duration (width) of the input function
m_f.x               = sparse(m_f.n,m_f.m);                              % initial condition
m_f.R               = [ 0.16 -0.22];                                    % input function parameters rho_1 and rho_2
m_f.S               = [-9e-4  0.08];                                    % static nonlinearity (activation function) parameters
m_f.T               = [ 0.16  0.13;  0.07 -0.82];                       % source-specific excitatory and inhibitory time constants
m_f.G               = [-0.11  0.45;  0.02 -0.63];                       % source-specific excitatory and inhibitory receotor densities
m_f.H               = [ 0.02 -0.07   0.18 -0.17];                       % source-independent intrinsic connectivity parameters
m_f.D               = [ 0    -0.52; -0.06  0   ];                       % between source delay parameters
m_f.A{1}            = [0 0; 0 0];                                       % forward  connectivity
m_f.A{2}            = [0 0; 0 0];                                       % backward connectivity
m_f.A{3}            = zeros(2,2);                                       % lateral connectivity
m_f.C               = [NaN NaN]';                                       % input connectivity

% forward model structure and parameters
% -------------------------------------------------------------------------
% load an existing dipole model
load(fullfile(cd, 'Data\SIM\dip_model_struct.mat'))
m_g.vol             = dip_model_struct.vol;                             % forward model specification
m_g.sens            = dip_model_struct.sens ;                           % sensor space specification
m_g.elab            = dip_model_struct.sens.label;                      % electrode labels
m_g.ne              = numel(m_g.elab);                                  % number of electrodes
m_g.X0              = zeros(m_f.ns*m_g.ne, 3*m_g.ne);                   % confound design matrix
m_g.Lpos            = [42 -31 58; 54 -22 18]';                          % dipole coordinate specifications
m_g.L               = [0.02 -0.05 -0.04; 0.17 -0.50 -0.17]';            % dipole moment specification
m_g.J               = [-0.1 0 0 0 0 0 -0.46 0 1];                       % state weighting specification

% -------------------------------------------------------------------------
%                           Visualization
% -------------------------------------------------------------------------
% initialize figure
fig = figure;
set(fig, 'Color', [1 1 1])

% canonical dipole
% -------------------------------------------------------------------------
% define parameters
pos         = m_g.Lpos(:,1);                                            % MNI position coordinates
mom         = [ 1;  1;   1];                                            % MNI moment coordinates
mf          = 4e1;                                                      % magnifying factor for dipole moment
mfname      = [pwd, '\Data\SIM\cortex_5124.surf.gii'];                  % cortical mesh filename
m           = struct(gifti(mfname));                                    % load cortical mesh using @gifti and @xmltree methods and convert to structure
m           = rmfield(m, 'mat');                                        % remove mat field for patch functionality

subplot(2,3,1)
hold on
patch(m,'facecolor', [.5 .5 .5], 'EdgeColor', 'none','FaceLighting','gouraud', 'visible', 'on', 'facealpha', 0.1); % plot mesh
plot3(pos(1),pos(2),pos(3), 'bo', 'MarkerSize',2, 'MarkerFaceColor', 'b');  % plot dipole location
pro = mf*diag(mom);                                                     % canonical axis projection of the dipole moment
for j = 1:3
    quiver3(pos(1),pos(2),pos(3),pro(1,j),pro(2,j),pro(3,j),'b:', 'LineWidth',2, 'ShowArrowHead', 'off'); % plot axis projections
end
quiver3(pos(1),pos(2),pos(3),mf*mom(1),mf*mom(2),mf*mom(3),'b-', 'LineWidth',2, 'ShowArrowHead', 'off'); % plot moment
grid on
axis equal
axis tight
xlabel('x', 'FontName', 'Calibri Light', 'FontSize', 22)
ylabel('y', 'FontName', 'Calibri Light', 'FontSize', 22)
zlabel('z', 'FontName', 'Calibri Light', 'Rotation',0,'FontSize', 22)
set(gca, 'FontName', 'Calibri Light', 'FontSize', 16)
view([115 10])

% plot canonical dipole leadfields
% -------------------------------------------------------------------------
nd          = size(m_g.L,2);                                            % number of dipoles
ne          = size(m_g.sens.chanpos,1);                                 % number of electrodes
dp          = 1e-3*m_g.Lpos ;                                           % dipole coordinates adjusted for fieldtrip
L_can  = NaN(ne,3,nd);                                                  % compute canonical dipole lead-field using Fieldtrip
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end
L_can = L_can*(10^1)^-round(log10(max(max(max(abs(L_can))))/8)/1);          % rescale lead-field according to SPM
subplot(2,3,2)
imagesc(L_can(:,:,1))
set(gca, 'xtick', 1:3, 'xticklabel',{'x', 'y', 'z'})
set(gca, 'FontName', 'Calibri Light', 'FontSize', 26)
ylabel('Electrodes', 'FontName', 'Calibri Light', 'FontSize', 20)

% plot canonical topologies
%-------------------------------------------------------------------------
dat.label               = m_g.elab';                                    % prepare fieldtrip parameter structure fields for plot data
dat.dimord              = 'chan_time';
dat.time                = 0;
dat.avg                 =  L_can(:,:,1)*mom(:,1);
cfg.layout              = 'ordered';                                    % prepare fieldtrip parameter structure fields for plot layout
lay                     = ft_prepare_layout(cfg,dat);
load([pwd, '\Data\SIM\coords_338_elecs.mat']);                          % obtain correct channel locations for fieldtrip
for n = 1:length(dat.label);                                            % match the labels in the of the current data
    actelec = cell2mat(dat.label(n,:));
    ind     = strmatch(actelec, Layout.Labels, 'exact');
    X(n)    = Layout.Coords(ind,1);
    Y(n)    = Layout.Coords(ind,2);
end
lay.pos(1:64,:)         = [X;Y]';
scaler                  =  0.6;
lay.pos(1:length(dat.label),:)  = [X;Y].*scaler;                        % specify spatial scaling of the plot
lay.width               = lay.width*scaler;
lay.height              = lay.height*scaler;
cfg.layout              = lay;
cfg.interactive         = 'no';
cfg.style               = 'straight';
cfg.comment             = 'no';
subplot(2,3,3)
ft_topoplotER(cfg, dat);                                                % topoplot

% latent variable evolution
% -------------------------------------------------------------------------
% evaluate latent data
theta               = [1 0];
[f_theta_f,u]       = erp_f(m_f,theta);
f_theta_f_s1        = f_theta_f(:,1:2:end-1);
tidx                = 43;                                               % time index for plotting
subplot(2,3,4)                                                          % visualize
[ax,h1,h2] = plotyy(m_f.pst,m_g.J*f_theta_f_s1',m_f.pst,f_theta_f_s1);
xlim([0 .15])
set(h1, 'Color', 'b')
set(h2, 'Color', [.7 .7 .7])
set(ax(1),'ycolor','b')
set(ax(1), 'FontName', 'Calibri Light', 'FontSize', 20)
set(ax(2), 'FontName', 'Calibri Light', 'FontSize', 20)
line([m_f.pst(tidx) m_f.pst(tidx)],get(ax(2),'YLim'),'Color',[0 0 0])
ylabel('a.u.', 'FontName', 'Calibri Light', 'FontSize', 20)
xlabel('Time [s]', 'FontName', 'Calibri Light', 'FontSize', 20)

% actual dipole
% -------------------------------------------------------------------------
% define parameters
pos         = m_g.Lpos(:,1);                                            % MNI position coordinates
mom         = m_g.L(:,1);                                               % MNI moment coordinates
mf          = 9e2;                                                      % magnifying factor for dipole moment
mfname      = [pwd, '\Data\SIM\cortex_5124.surf.gii'];                  % cortical mesh filename
m           = struct(gifti(mfname));                                    % load cortical mesh using @gifti and @xmltree methods and convert to structure
m           = rmfield(m, 'mat');                                        % remove mat field for patch functionality
subplot(2,3,5)
hold on
patch(m,'facecolor', [.5 .5 .5], 'EdgeColor', 'none','FaceLighting','gouraud', 'visible', 'on', 'facealpha', 0.1); % plot mesh
plot3(pos(1),pos(2),pos(3), 'bo', 'MarkerSize',2, 'MarkerFaceColor', 'b');
pro = mf*diag(mom);                                                     % canonical axis projection of the dipole moment
for j = 1:3
    quiver3(pos(1),pos(2),pos(3),pro(1,j),pro(2,j),pro(3,j),'b:', 'LineWidth',2, 'ShowArrowHead', 'off');  % plot axis projections
end
quiver3(pos(1),pos(2),pos(3),mf*mom(1),mf*mom(2),mf*mom(3),'b-', 'LineWidth',2, 'ShowArrowHead', 'off');% plot moment
grid on
axis equal
axis tight
xlabel('x', 'FontName', 'Calibri Light', 'FontSize', 22)
```

```
ylabel('y', 'FontName', 'Calibri Light', 'FontSize', 22)
zlabel('z', 'FontName', 'Calibri Light', 'Rotation',0,'FontSize', 22)
set(gca, 'FontName', 'Calibri Light', 'FontSize', 16)
view([41 14])

% plot actual topology
%--------------------------------------------------------------------------
L_dip = NaN(ne,nd);                                                         % evaluate channel predictions based on dipole moments
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*m_g.L(:,i);
end
g_theta_g = [];                                                             % evaluate g(\theta_g)
for i = 1:nd
    g_theta_g = [g_theta_g kron(m_g.J,L_dip(:,i))];
end
h_theta                 = g_theta_g*f_theta_f';                             % evaluate data
dat.label               = m_g.elab';                                        % prepare fieldtrip parameter structure fields for plot data
dat.dimord              = 'chan_time';
dat.time                = 0;
dat.avg                 = h_theta(:,tidx);
cfg.layout              = 'ordered';                                        % prepare fieldtrip parameter structure fields for plot layout
lay                     = ft_prepare_layout(cfg,dat);
load([pwd, '\Data\SIM\coords_338_elecs.mat']);                              % obtain correct channel locations for fieldtrip
for n = 1:length(dat.label);                                                % match the labels in the of the current MEEG object
    actelec = cell2mat(dat.label(n,:));
    ind     = strmatch(actelec, Layout.Labels, 'exact');
    X(n)    = Layout.Coords(ind,1);
    Y(n)    = Layout.Coords(ind,2);
end
lay.pos(1:64,:)         = [X;Y]';
scaler                  =  0.6;                                             % specify spatial scaling of the plot
lay.pos(1:length(dat.label),:)  = [X;Y]'.*scaler;
lay.width               = lay.width*scaler;
lay.height              = lay.height*scaler;
cfg.layout              = lay;
cfg.interactive         = 'no';
cfg.style               = 'straight';
cfg.comment             = 'no';
subplot(2,3,6)
ft_topoplotER(cfg, dat);                                                    % topoplot

% remove Fieldtrip
%--------------------------------------------------------------------------
rmpath(genpath(fullfile(pwd, 'Fieldtrip')))

end

% -------------------------------------------------------------------------
%                       Subfunctions
% -------------------------------------------------------------------------
function [h_theta_fg]   = erp_h(m_f,m_g,theta)

% This function evaluates the expectation parameter generating function
%
% h : (theta_f,theta_g) |-> h(theta_f,theta_g) := vec(g(theta_g)*f(theta_f)
%
% of the delay differential equation model for ERPs
%
%       Inputs
%           m_f         : latent neural model structure
%           m_g         : forward model structure
%           theta       : model parameters
%
%       Outputs
%           h_theta_fg  : evaluated function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluate component functions f and g
[f_theta_f, u]  = erp_f(m_f,theta);
g_theta_g       = erp_g(m_g);

% evaluate concatenated function h
h_theta_fg      = 1e4*spm_vec(g_theta_g*f_theta_f');

end

function [f_theta_f,u]  = erp_f(m_f,vartheta)

% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_gen_erp.m of
% the SPM12 distribution. Based on a given parameter setting this function
% generates an input time-course and  integrates the resulting delay
% differential equation system.
%
%       Inputs
%           m_f         : latent neural model structure and fixed parameters
%           theta       : free latent neural model parameter
%
%       Outputs
%           f_theta_f   : evaluated latent neural model
%           u           : evaluated system input function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.R             = m_f.R;
theta.S             = m_f.S;
theta.T             = m_f.T;
theta.G             = m_f.G;
theta.H             = m_f.H;
theta.D             = m_f.D;
theta.A{1}          = m_f.A{1};
theta.A{2}          = m_f.A{2};
theta.A{3}          = m_f.A{3};
theta.C             = m_f.C;

% free parameters
theta.C(1)          = vartheta(1);
theta.C(2)          = vartheta(2);

% evaluate peri-stimulus time input function
t                   = ((1:m_f.ns)*m_f.dt)*1000;
delay               = m_f.ons(1)+128*theta.R(1,1);
scale               = m_f.dur(1)*exp(theta.R(1,2));
u                   = 32*exp(-(t - delay).^2/(2*scale^2))';

% integrate systen
x                   = m_f.x                                                 ; % initial condition
f                   = m_f.f                                                 ; % function handle @spm_fx_erp
dt                  = m_f.dt                                                ; % integration time bin
[fx,dfdx,D]         = f(x,u(1),theta,m_f)                                   ; % dx(t)/dt and Jacobian df/dx and check for delay operator
p                   = max(abs(real(eig(full(dfdx)))))                       ; %
N                   = ceil(max(1, dt*p*2))                                  ; %
n                   = spm_length(x)                                         ; %
Q                   = (spm_expm(dt*D*dfdx/N) - speye(n,n))*spm_inv(dfdx)    ; %
v                   = spm_vec(x)                                            ; % initialize state
y                   = NaN(length(v), length(u))                             ; % initialize state time-course

% cycle over time-steps
for i = 1:size(u,1)

    % update dx = (expm(dt*J) - I)*inv(J)*f(x,u)
    for j = 1:N
        v = v + Q*f(v,u(i),theta,m_f);
    end
    y(:,i) = v;

end

% transpose
f_theta_f       = y';

end

function [f,J,Q]        = spm_fx_erp(x,u,P,M)

% state equations for a neural mass model of erps
% FORMAT [f,J,D] = spm_fx_erp(x,u,P,M)
% FORMAT [f,J]   = spm_fx_erp(x,u,P,M)
% FORMAT [f]     = spm_fx_erp(x,u,P,M)
% x      - state vector
%   x(:,1) - voltage (spiny stellate cells)
%   x(:,2) - voltage (pyramidal cells) +ve
%   x(:,3) - voltage (pyramidal cells) -ve
%   x(:,4) - current (spiny stellate cells)    depolarizing
%   x(:,5) - current (pyramidal cells)         depolarizing
%   x(:,6) - current (pyramidal cells)         hyperpolarizing
%   x(:,7) - voltage (inhibitory interneurons)
%   x(:,8) - current (inhibitory interneurons) depolarizing
%   x(:,9) - voltage (pyramidal cells)
%
% f       - dx(t)/dt  = f(x(t))
% J       - df(t)/dx(t)
% D       - delay operator dx(t)/dt = f(x(t - d))
%                                   = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
%  M.pF.E = [32 16 4];             % extrinsic rates (forward, backward, lateral)
%  M.pF.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2 g3, g4)
%  M.pF.D = [2 16];                % propogation delays (intrinsic, extrinsic)
%  M.pF.G = [4 32];                % receptor densities (excitatory, inhibitory)
%  M.pF.T = [8 16];                % synaptic constants (excitatory, inhibitory)
%  M.pF.R = [1 1/2];               % parameter of static nonlinearity
%
% _________________________________________________________________________
% David O, Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neuronal dynamics. NeuroImage 20: 1743-1755
% _________________________________________________________________________
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging

% Karl Friston
% $Id: spm_fx_erp.m 5369 2013-03-28 20:09:27Z karl $

% get dimensions and configure state variables
%--------------------------------------------------------------------------
n       = length(P.A{1});       % number of sources
x       = spm_unvec(x,M.x);     % neuronal states

% [default] fixed parameters
%--------------------------------------------------------------------------
E       = [1 1/2 1/8]*32;               % extrinsic rates (forward, backward, lateral)
```

```
G       = [1 4/5 1/4 1/4]*128;              % intrinsic rates (g1 g2 g3 g4)
D       = [2 16];                           % propagation delays (intrinsic, extrinsic)
H       = [4 32];                           % receptor densities (excitatory, inhibitory)
T       = [8 16];                           % synaptic constants (excitatory, inhibitory)
R       = [2 1]/3;                          % parameters of static nonlinearity

% test for free parameters on intrinsic connections
%--------------------------------------------------------------------------
G       = G.*exp(P.H);
G       = ones(n,1)*G;

% no exponential transforms to foster parameter identifiability
% -------------------------------------------------------------------------
A{1}    = P.A{1}*E(1);
A{2}    = P.A{2}*E(2);
A{3}    = P.A{3}*E(3);
C       = P.C;

% intrinsic connectivity and parameters
%--------------------------------------------------------------------------
Te      = T(1)/1000*exp(P.T(:,1));        % excitatory time constants
Ti      = T(2)/1000*exp(P.T(:,2));        % inhibitory time constants
He      = H(1)*exp(P.G(:,1));             % excitatory receptor density
Hi      = H(2)*exp(P.G(:,2));             % inhibitory receptor density

% pre-synaptic inputs: s(V)
%--------------------------------------------------------------------------
S       = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
%--------------------------------------------------------------------------
U       = C*u(:)*2;

% State: f(x)
%==========================================================================
f(:,1) = x(:,4)                                                             ; % voltage change (spiny stellate cells)
f(:,2) = x(:,5)                                                             ; % positive voltage change (pyramidal cells) +ve
f(:,3) = x(:,6)                                                             ; % negative voltage change (pyramidal cells) -ve
f(:,4) = (He.*((A{1} + A{3})*S(:,9) + G(:,1).*S(:,9) + U) - 2*x(:,4) - x(:,1)./Te)./Te      ; % current change (spiny stellate cells)    depolarizing
f(:,5) = (He.*((A{2} + A{3})*S(:,9) + G(:,2).*S(:,1)) - 2*x(:,5) - x(:,2)./Te)./Te         ; % current change (pyramidal cells)         depolarizing
f(:,6) = (Hi.*G(:,4).*S(:,7) - 2*x(:,6) - x(:,3)./Ti)./Ti                   ; % current change (pyramidal cells)         hyperpolarizing
f(:,7) = x(:,8)                                                             ; % voltage change (inhibitory interneurons)
f(:,8) = (He.*((A{2} + A{3})*S(:,9) + G(:,3).*S(:,9)) - 2*x(:,8) - x(:,7)./Te)./Te         ; % current change(inhibitory interneurons) depolarizing
f(:,9) = x(:,5) - x(:,6)                                                    ; % voltage (pyramidal cells)

% vectorize
f       = spm_vec(f);


% avoid infinite recursion of spm_diff
if nargout < 2
    return
end

% Jacobian
%==========================================================================
J  = spm_diff(M.f,x,u,P,M,1);

% delays
%==========================================================================
% Delay differential equations can be integrated efficiently (but
% approximately) by absorbing the delay operator into the Jacobian
%
%    dx(t)/dt     = f(x(t - d))
%                 = Q(d)f(x(t))
%
%    J(d)         = Q(d)df/dx
%--------------------------------------------------------------------------
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,n)).*De;                                  % remove diagonal entries
Di = (1 - speye(9,9)).*Di;                                  % remove diagonal entries
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D  = Di + De;

% -------------------------------------------------------------------------
%                     Delay parameter matrix Visualization
% -------------------------------------------------------------------------


% ERP-DCM ordering
% -------------------------------------------------------------------------
nu   = size(D,1);                                           % number of neural state variables
roDi = NaN(size(Di));                                       % re-ordered intrinsic  delays
roDe = NaN(size(De));                                       % re-ordered extrinsic delays
roD  = NaN(size(D));                                        % re-ordered combined delays
rotl   = cell(1,nu);                                        % re-ordered ticks label initialization
idx    = 1;                                                 % re-ordered tick label index

% reo-ordering hack
for sv = 1:9
    roDi(sv,:)   = [Di(sv*2-1,1:2:17) Di(sv*2-1,2:2:18)];
    roDi(sv+9,:) = [Di(sv*2,1:2:17)   Di(sv*2,2:2:18)  ];
    roDe(sv,:)   = [De(sv*2-1,1:2:17) De(sv*2-1,2:2:18)];
    roDe(sv+9,:) = [De(sv*2,1:2:17)   De(sv*2,2:2:18)  ];
    roD(sv,:)    = [D(sv*2-1,1:2:17)  D(sv*2-1,2:2:18) ];
    roD(sv+9,:)  = [D(sv*2,1:2:17)    D(sv*2,2:2:18)   ];
end

% create axes labels by cycling over state variables and sources
for i = 1:n
    for sv = 1:9
        rotl{idx} = ['x_' num2str(sv) '^{^' num2str(i) '^}'];
        idx = idx + 1;

    end
end

fig = figure;
set(fig, 'Color', [1 1 1])
subplot(1,3,1)
imagesc(roDi, [0 0.01])
axis square
axis off
colorbar
set(gca, 'FontSize',20, 'FontName', 'Calibri Light')

subplot(1,3,2)
imagesc(roDe, [0 0.01])
axis square
axis off
colorbar
set(gca, 'FontSize',20, 'FontName', 'Calibri Light')

subplot(1,3,3)
imagesc(roD, [0 0.01])
axis square
axis off
colorbar
set(gca, 'FontSize',20, 'FontName', 'Calibri Light')
pause

% Implement: dx(t)/dt = f(x(t - d)) = inv(1 + D.*dfdx)*f(x(t))
%                     = Q*f = Q*J*x(t)
%--------------------------------------------------------------------------
Q  = spm_inv(speye(length(J)) + D.*J);

end

function [g_theta_g]    = erp_g(m_g)

% This function evaluates the EEG forward model of the delay differential
% equation model for ERPs based on a structural formulation of the forward
% model and parameter setting
%
%   Inputs
%           m_g         : EEG forward model structure and fixed parameters
%           theta       : EEG lead-field free parameters

%    Outputs
%           g_theta_g   : evaluated EEG forward model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.Lpos  = m_g.Lpos;
theta.L     = m_g.L;
theta.J     = m_g.J;

% evaluate constants
nd          = size(theta.L,2)               ; % number of dipoles
ne          = size(m_g.sens.chanpos,1)      ; % number of electrodes
dp          = 1e-3*theta.Lpos               ; % dipole coordinates adjusted for fieldtrip

% compute canonical dipole lead-field using Fieldtrip
L_can  = NaN(ne,3,nd);
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end

% rescale lead-field according to SPM
L_can = L_can*(10^1)^-round(log10(max(max(max(abs(L_can))))/8)/1);

% evaluate channel predictions based on dipole moments
L_dip = NaN(ne,nd);
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*theta.L(:,i);
end

% evaluate g(\theta_g)
g_theta_g = [];
for i = 1:nd
    g_theta_g = [g_theta_g kron(theta.J,L_dip(:,i))];
end
end
```

PDDE_2.m – First toy model (Figures 3 and 4)

```
function pdde_2_R1

% This function implements the estimation of an ERP-DCM toy example.
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------

% -------------------------------------------------------------------------
%                             Initialization
% -------------------------------------------------------------------------
clc
close all
stream = RandStream.getGlobalStream;reset(stream);                          % set the random number generator for reproducible restults

% -------------------------------------------------------------------------
%                 1D parameter space & variance known model
% -------------------------------------------------------------------------
h                       = @(theta_f) h_fun(theta_f);                        % function handle specification h : R   -> R^n
theta                   = 2;                                                % true, but unknown, parameter
sigma                   = 10;                                               % true, known, variance parameter
p                       = length(theta);                                    % number of parameters
h_theta                 = h(theta);                                         % evaluate h
y                       = mvnrnd(h_theta,sigma*eye(length(h_theta)))';      % sample data for all simulations

% -------------------------------------------------------------------------
%                   Estimation of variational variance
% -------------------------------------------------------------------------
fprintf('Estimation of variational variance for theta only\n')

% prior formulation
% -------------------------------------------------------------------------
mu_theta                = 0;                                                % prior parameter settings
sig_theta               = 1e3;                                              % prior parameter settings

% estimate variational variance
% -------------------------------------------------------------------------
% set estimated variational expectation to prior parameter
m_theta                 = mu_theta;
s_theta                 = sig_theta;

% evaluate the variational variance update equation for the current choice of m_theta
Jh_m_theta              = spm_diff(h,m_theta,1);
s_theta_update          = 1/(((1/sigma)*(Jh_m_theta'*Jh_m_theta)) + (1/sig_theta));

% evaluate variational free energy before and after s_theta update
vFE_1_arg.y             = y;
vFE_1_arg.h             = h;
vFE_1_arg.mu_theta      = mu_theta;
vFE_1_arg.sig_theta     = sig_theta;
vFE_1_arg.sigma         = sigma;
vFE_s_theta(1)          = vFE_1(m_theta,s_theta,vFE_1_arg);
vFE_s_theta(2)          = vFE_1(m_theta,s_theta_update ,vFE_1_arg);

% visualization
% -------------------------------------------------------------------------
fig = figure;
set(fig, 'Color', [1 1 1])

% plot the data,function h, and MAP fit
subplot(2,3,1)
hold on
plot(1:10, h_theta)
plot(1:10, y, 'ko', 'MarkerSize', 6)
xlabel('x', 'FontSize', 26, 'FontName', 'Calibri Light')
xlim([0 11])
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% plot variational free energy descent
% -------------------------------------------------------------------------
m_min_full              = -0.2;
m_max_full              =  0.2;
m_res_full              = 1e2;
m_space_full            = linspace(m_min_full,m_max_full,m_res_full);
s_min_full              = 1e-6;
s_max_full              = 2e3;
s_res_full              = 1e2;
s_space_full            = linspace(s_min_full,s_max_full,s_res_full);

% arrray initialization
vFE_s_m_theta           = NaN(length(s_space_full),length(m_space_full));

% cycle over variational expectation space
for i = 1:length(s_space_full)
    for j = 1:length(m_space_full)
        vFE_s_m_theta(i,j) = vFE_1(m_space_full(j),s_space_full(i),vFE_1_arg);
    end
end

subplot(2,3,2)
hold on
surf(m_space_full, s_space_full, -vFE_s_m_theta, 'EdgeColor', 'None')
alpha(.6)
plot3([mu_theta mu_theta], [s_theta s_theta_update], -vFE_s_theta, 'ko--', 'MarkerFaceColor', 'k', 'MarkerSize', 10)
xlim([m_min_full m_max_full])
ylim([s_min_full s_max_full])
set(gca, 'YDir', 'normal')
xlabel('m_\theta', 'FontSize', 26, 'FontName', 'Calibri Light')
ylabel('s_\theta^2', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
zlabel('F', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')
view([-136 38])
grid on

% evaluate the free energy as a function of the variational variance prior
% -------------------------------------------------------------------------
% prior variational variance parameter settings
sig_theta_all = [1e-5 1e-4 1e3];

% plotting colors
scols         = {'r', 'b', 'g'};

% cycle over variance parameter settings
for s = 1:3

    % define current prior/marginal parameters
    mu_theta            = 2;
    sig_theta           = sig_theta_all(s);
    s_min               = 0;
    s_max               = 1.2e-4;
    s_res               = 1e2;
    s_space             = linspace(s_min,s_max,s_res);
    m_theta             = theta;
    vFE_1_arg.mu_theta  = mu_theta;
    vFE_1_arg.sig_theta = sig_theta;

    % arrray initialization
    vFE_s_theta         = NaN(1,length(s_space));

    % cycle over variational expectation space
    for i = 1:length(s_space)
        vFE_s_theta(i) = vFE_1(m_theta,s_space(i),vFE_1_arg);
    end

    % evaluate the variational variance update equation for the current choice of m_theta
    Jh_m_theta          = spm_diff(h,m_theta,1);
    s_theta_update      = 1/(((1/sigma)*(Jh_m_theta'*Jh_m_theta)) + (1/sig_theta));
    F_s_theta_update    = vFE_1(m_theta,s_theta_update,vFE_1_arg);

    % visualization of analytical estimation
    % -------------------------------------------------------------------------
    % plot free energy as function of the variational variance, and its update
    subplot(2,3,3)
    hold on
    lines{s} = plot(s_space , -vFE_s_theta , 'Color', scols{s});
    plot(s_theta_update     , -F_s_theta_update , 'ko', 'MarkerSize', 10, 'MarkerFaceColor', scols{s})

end

xlim([s_min s_max])
xlabel('s_\theta^2', 'FontSize', 26, 'FontName', 'Calibri Light')
ylabel('F', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
legend(lines, '\sigma_\theta^2 = 10^{-4}','\sigma_\theta^2 = 10^{-3}','\sigma_\theta^2 = 10^{3}', 'Location', 'NorthEast')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')


% -------------------------------------------------------------------------
%                   Estimation of variational mean
% -------------------------------------------------------------------------
fprintf('Estimation of variational mean for theta only\n')

% prior formulation
% -------------------------------------------------------------------------
mu_theta                = 0;                                                % prior parameter settings
sig_theta               = 1e3;                                              % prior parameter settings

% backtracking Newton line search with Hessian modification parameters
% -------------------------------------------------------------------------
delta                   = sqrt(2*eps);                                      % Hessian modification constant delta
c                       = 1e-4;                                             % backtracking constant c
rho                     = 0.99;                                             % backtracking constant rho
max_iter                = 5;                                                % maximal number of iterations

% model estimation - iteration arrays
% -------------------------------------------------------------------------
m_theta_n               = NaN(p,max_iter);                                  % variational expectation parameter
F_n                     = NaN(1,max_iter);                                  % variational free energy
dF_n                    = NaN(1,max_iter);                                  % gradient of the variational free energy wrt m_theta
ddF_n                   = NaN(1,max_iter);                                  % Hessian of the variational free energy wrt m_theta
t_i_n                   = NaN(1,max_iter);                                  % backtracking step size

% model estimation - iteration 0
% -------------------------------------------------------------------------
m_theta_n(1)            = mu_theta;                                         % initialization of the variational parameters to prior parameters
Jh_m_theta              = spm_diff(h,m_theta,1);                            % evaluate constant variational variance
s_theta                 = 1/(((1/sigma)*(Jh_m_theta'*Jh_m_theta)) + (1/sig_theta));
```

```
% evaluation of the variational free energy, its gradient and Hessian
vFE_1_arg.y             = y;
vFE_1_arg.h             = h;
vFE_1_arg.mu_theta      = mu_theta;
vFE_1_arg.sig_theta     = sig_theta;
vFE_1_arg.sigma         = sigma;
F_n(1)                  = vFE_1(m_theta_n(1),s_theta, vFE_1_arg);
dF_n(1)                 = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta,vFE_1_arg), m_theta_n(1),1)));
ddF_n(1)                = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta,vFE_1_arg), m_theta_n(1),[1 1])));

% Newton iterations 1,2,3,...
% -------------------------------------------------------------------------
for i = 2:max_iter

    % backtracking Newton line search with Hessian modification for
    % variational expectation
    % ---------------------------------------------------------------------
    fprintf('Fixed-form VB Newton Iteration %4.0f \n', i-1)

    % Newton search direction
    p_i                 = -ddF_n(i-1)\dF_n(i-1);

    % Hessian modification if p_i is not a descent direction
    if ~(p_i'*dF_n(i-1) < 0)

       % diagonal Hessian modification based on spectral decomposition
       ddF_n(i-1) = ddF_n(i-1) + max(0,(delta - min(eig(ddF_n(i-1)))))*eye(p);

       % re-evaluate Newton search direction
       p_i      = -ddF_n(i-1)\dF_n(i-1);

    end

    % backtracking evaluation of the step length
    t_i_n(i-1)          = f_backtrack_vFE_1(m_theta_n(i-1), s_theta, vFE_1_arg, dF_n(i-1), p_i, c, rho);

    % perform Newton update
    m_theta_n(i)        = m_theta_n(i-1) + t_i_n(i-1)*p_i;

    % evaluation of the variational free function value, gradient, and Hessian at updated location
    F_n(i)              = vFE_1(m_theta_n(i),s_theta, vFE_1_arg);
    dF_n(i)             = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta,vFE_1_arg), m_theta_n(i),1)));
    ddF_n(i)            = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta,vFE_1_arg), m_theta_n(i),[1 1])));

end


% plot the the MAP fit
% -------------------------------------------------------------------------
h_m_theta_MAP  = h(m_theta_n(end));
subplot(2,3,1)
plot(1:10, h_m_theta_MAP, 'k--')
legend('h(\theta)', 'y', 'h(\theta_{MAP})', 'Location', 'NorthWest')

% visualize Newton descent on variational free energy surface
% -------------------------------------------------------------------------
m_min_full              = -.2;
m_max_full              = 2.4;
m_res_full              = 1e2;
m_space_full            = linspace(m_min_full,m_max_full,m_res_full);
s_min_full              = 7e-5;
s_max_full              = 9e-5;
s_res_full              = 1e2;
s_space_full            = linspace(s_min_full,s_max_full,s_res_full);

% arrray initialization
vFE_s_m_theta           = NaN(length(s_space_full),length(m_space_full));

% cycle over variational expectation space
for i = 1:length(s_space_full)
    for j = 1:length(m_space_full)
        vFE_s_m_theta(i,j) = vFE_1(m_space_full(j),s_space_full(i),vFE_1_arg);
    end
end

subplot(2,3,4)
hold on
surf(m_space_full, s_space_full, -vFE_s_m_theta, 'EdgeColor', 'None')
alpha(.6)
plot3(m_theta_n, s_theta*ones(length(m_theta_n)), -F_n, 'ko--', 'MarkerFaceColor', 'k', 'MarkerSize', 8)
xlim([m_min_full m_max_full])
ylim([s_min_full s_max_full])
view([140 44])
grid on
set(gca, 'YDir', 'normal')
xlabel('m_\theta', 'FontSize', 26, 'FontName', 'Calibri Light')
ylabel('s_\theta^2', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
zlabel('F', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% plot free energy and derivatives as function of variational expectation
% -------------------------------------------------------------------------
sig_theta_all           = [1e-3 1.4e-3];                                    % prior parameter settings
mu_theta                = 0;                                                % prior parameter settings

for s = 1:length(sig_theta_all)

    % set prior parameters
    vFE_1_arg.mu_theta  = mu_theta;
    vFE_1_arg.sig_theta = sig_theta_all(s);

    % evaluate constant variational variance
    Jh_m_theta          = spm_diff(h,m_theta,1);
    s_theta             = 1/(((1/sigma)*(Jh_m_theta'*Jh_m_theta)) + (1/sig_theta));
    m_min               = -.5;
    m_max               = 2.1;
    m_res               = 1e2;
    m_space             = linspace(m_min,m_max,m_res);

    % arrray initialization
    vFE_m_theta         = NaN(1,length(m_space));
    dvFE_m_theta_n      = NaN(1,length(m_space));
    ddvFE_m_theta_n     = NaN(1,length(m_space));

    % cycle over variational expectation space
    for i = 1:length(m_space)
        vFE_m_theta(i)      = vFE_1(m_space(i),s_theta,vFE_1_arg);
        dvFE_m_theta_n(i)   = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta,vFE_1_arg), m_space(i),1)));
        ddvFE_m_theta_n(i)  = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta,vFE_1_arg), m_space(i),[1 1])));
    end

    % visualize
    subplot(2,3,5+s-1)
    hold on
    plot(m_space,-vFE_m_theta               , 'k')
    plot(m_space,-dvFE_m_theta_n            , 'b')
    plot(m_space,-ddvFE_m_theta_n           , 'r')
    plot(m_space,zeros(1,length(m_space))   , 'k--')
    xlabel('m_\theta', 'FontSize', 26, 'FontName', 'Calibri Light')
    ylim([min(-vFE_m_theta) max(-ddvFE_m_theta_n)])
    xlim([m_min m_max])
    legend('F', 'dF', 'd^2F', 'Location', 'NorthEast')
    set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
end

return
% -------------------------------------------------------------------------
%                 Estimation of variational mean & variance
% -------------------------------------------------------------------------
fprintf('Joint estimation of variational variance and mean for theta\n')

% prior formulation
% -------------------------------------------------------------------------
mu_theta            = 0;                                                    % prior parameter settings
sig_theta           = 1e3;                                                  % prior parameter settings
p                   = length(mu_theta);                                     % number of parameters

% fixed-form variational Bayes Newton algorithm parameters
% -------------------------------------------------------------------------
max_i               = 5;
max_j               = 5;                                                    % maximal number of Newton iterations for m_theta optimization
delta               = sqrt(2*eps);                                          % Hessian modification constant delta
c                   = 1e-4;                                                 % backtracking constant c
rho                 = 0.99;                                                 % backtracking constant rho

% model estimation - iteration arrays
% -------------------------------------------------------------------------
m_theta_i           = NaN(1,max_i);                                         % variational expectation parameter
s_theta_i           = NaN(1,max_i);                                         % variational variance parameter
F_i                 = NaN(1,max_i);                                         % variational free energy

% model estimation - iteration 0
% -------------------------------------------------------------------------
% initialization of the variational parameters to prior parameters
m_theta_i(1)        = mu_theta;
s_theta_i(1)        = sig_theta;
vFE_1_arg.y         = y;
vFE_1_arg.h         = h;
vFE_1_arg.mu_theta  = mu_theta;
vFE_1_arg.sig_theta = sig_theta;
vFE_1_arg.sigma     = sigma;
F_i(1)              = vFE_1(m_theta_i(1),s_theta_i(1), vFE_1_arg);

% iterations 1,2,3,...
% -------------------------------------------------------------------------
for i = 2:max_i

    fprintf('Fixed-form VB Newton Iteration %4.0f \n', i-1)

    % analytical update of variational variance parameter
    % ---------------------------------------------------------------------
    % evaluate the variational variance update equation for the current choice of m_theta
    fprintf('   Estimating q(m_theta) variance    ... ')
    Jh_m_theta          = spm_diff(h,m_theta_i(i-1),1);
    s_theta_i(i)        = inv(((1/sigma)*(Jh_m_theta'*Jh_m_theta)) + (1/sig_theta));
    fprintf('finished. \n')

    % evaluation of the variational free function derivatives
    dF                  = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta_i(i),vFE_1_arg), m_theta_i(i-1),1)));
    ddF                 = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta_i(i),vFE_1_arg), m_theta_i(i-1),[1 1])));

    % Newton iterations 1,2,3,...
    % ---------------------------------------------------------------------
```

```
    fprintf('   Estimating q(m_theta) expectation ... ')

    % initialize m_theta
    m_theta      = m_theta_i(i-1);

    % iterations
    for j = 2:max_j

        % backtracking Newton line search with Hessian modification for
        % variational expectation
        % ------------------------------------------------------------------
        % Newton search direction
        p_j      = -ddF\dF;

        % Hessian modification if p_i is not a descent direction
        if ~(p_j'*dF < 0)

           % diagonal Hessian modification based on spectral decomposition
           ddF  = ddF + max(0,(delta - min(eig(ddF))))*eye(p);

           % re-evaluate Newton search direction
           p_j  = -ddF\dF;

        end

        % backtracking evaluation of the step length
        t_j      = f_backtrack_vFE_1(m_theta, s_theta_i(i), vFE_1_arg, dF, p_j, c, rho);

        % perform Newton update
        m_theta = m_theta + t_j*p_j;

        % evaluate negative variational free energy and its derivatives
        dF       = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta_i(i),vFE_1_arg), m_theta,1)));
        ddF      = full(spm_cat(spm_diff(@(m_theta) vFE_1(m_theta,s_theta_i(i),vFE_1_arg), m_theta,[1 1])));


    end
    fprintf('finished. \n')

    % record m_theta and variational free energy update
    m_theta_i(i) = m_theta;
    F_i(i)       = vFE_1(m_theta_i(i), s_theta_i(i), vFE_1_arg);

end

% visualization
% -------------------------------------------------------------------------
fig = figure;
set(fig, 'Color', [1 1 1])

% visualization of prior and approximated posterior over theta
theta_min    = -1;
theta_max    =  4;
theta_res    = 1e4;
theta_space = linspace(theta_min, theta_max, theta_res);

subplot(2,3,1)
hold on
[ax, line1, line2] = plotyy(theta_space, pdf('Normal', theta_space, mu_theta , sqrt(sig_theta)), theta_space, pdf('Normal', theta_space, m_theta_i(end), sqrt(s_theta_i(end))));
xlim([theta_min theta_max])
xlabel('\theta', 'FontSize', 26, 'FontName', 'Calibri Light')
legend('p(\theta)', 'q(\theta)')
set(ax, 'FontSize', 20, 'FontName', 'Calibri Light')

%  Visualization - Iteration 0
% -------------------------------------------------------------------------
m_min_full             = -3;
m_max_full             = 3;
m_res_full             = 1e2;
m_space_full           = linspace(m_min_full,m_max_full,m_res_full);
s_min_full             = 1e-3;
s_max_full             = 1.5e3;
s_res_full             = 1e2;
s_space_full           = linspace(s_min_full,s_max_full,s_res_full);
vFE_1_arg.y            = y;
vFE_1_arg.h            = h;
vFE_1_arg.mu_theta     = mu_theta;
vFE_1_arg.sig_theta    = sig_theta;
vFE_1_arg.sigma        = sigma;

% arrray initialization
vFE_s_m_theta            = NaN(length(s_space_full),length(m_space_full));

% cycle over variational expectation space
for i = 1:length(s_space_full)
    for j = 1:length(m_space_full)
        vFE_s_m_theta(i,j) = vFE_1(m_space_full(j),s_space_full(i),vFE_1_arg);
    end
end

subplot(2,3,2)
hold on
surf(m_space_full, s_space_full,  -vFE_s_m_theta, 'EdgeColor', 'None')
alpha(.6)
plot3(m_theta_i(1), s_theta_i(1), -F_i(1), 'ko', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
xlabel('m_\theta', 'FontSize', 26, 'FontName', 'Calibri Light')
ylabel('s_\theta^2', 'FontSize', 26, 'FontName', 'Calibri Light')
grid on
xlim([m_min_full m_max_full])
ylim([s_min_full s_max_full])
zlim( [-3.5e3 0])
caxis([-3.5e3 0])
view([143 12])
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')


% depict the global minimum of the variational free energy
% -------------------------------------------------------------------------
m_min_full             = 2.00;
m_max_full             = 2.06;
m_res_full             = 1e2;
m_space_full           = linspace(m_min_full,m_max_full,m_res_full);
s_min_full             = 1e-6;
s_max_full             = 4e-4;
s_res_full             = 1e2;
s_space_full           = linspace(s_min_full,s_max_full,s_res_full);
vFE_1_arg.y            = y;
vFE_1_arg.h            = h;
vFE_1_arg.mu_theta     = mu_theta;
vFE_1_arg.sig_theta    = sig_theta;
vFE_1_arg.sigma        = sigma;

% arrray initialization
vFE_s_m_theta          = NaN(length(s_space_full),length(m_space_full));

% cycle over variational expectation space
for i = 1:length(s_space_full)
    for j = 1:length(m_space_full)
        vFE_s_m_theta(i,j) = vFE_1(m_space_full(j),s_space_full(i),vFE_1_arg);
    end
end

% plot variational free energy as function of both variational parameters
subplot(2,3,3)
hold on
surf(m_space_full, s_space_full,      -vFE_s_m_theta, 'EdgeColor', 'None')
plot3(m_theta_i(end), s_theta_i(end), -F_i(end), 'ko', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
alpha(.6)
grid on
xlim([m_min_full m_max_full])
ylim([s_min_full s_max_full])
xlabel('m_\theta', 'FontSize', 26, 'FontName', 'Calibri Light')
ylabel('s_\theta^2', 'FontSize', 26, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
view([135 12])

% -------------------------------------------------------------------------
%                 Variational Bayesian estimation of sigma^2
% -------------------------------------------------------------------------
fprintf('Estimation of variational shape and scale sigma^2\n')
mu_theta             = 2.031;                                   % prior expectation
sig_theta_all        = [7.4015e-05 2e-3];                       % prior variance
mu_sigsqr            = 0.2;                                     % prior shape parameter
si_sigsqr            = 4;                                       % prior scale parameter

% cycle over uncertainty setting for theta
% -------------------------------------------------------------------------
for s = 1:length(sig_theta_all)

    % set theta uncertainty
    sig_theta            = sig_theta_all(s);

    % algoritm parameters and arrays
    % ---------------------------------------------------------------------
    delta                = sqrt(2*eps);                         % Hessian modification constant delta
    c                    = 1e-4;                                % backtracking constant c
    rho                  = 0.99;                                % backtracking constant rho
    max_iter             = 10;                                  % maximal number of iterations
    ms_n                 = NaN(2,max_iter);                     % variational expectation parameter
    F_n                  = NaN(1,max_iter);                     % variational free energy

    % model estimation - iteration 0
    % ---------------------------------------------------------------------
    % initialization of the variational parameters to prior parameters
    ms_n(:,1)            = [mu_sigsqr; si_sigsqr];

    % evaluation of the variational free energy, its gradient and Hessian
    vFE_arg.y            = y;
    vFE_arg.h            = h;
    vFE_arg.m_theta      = mu_theta;
    vFE_arg.s_theta      = sig_theta;
    vFE_arg.mu_sigsqr    = mu_sigsqr;
    vFE_arg.si_sigsqr    = mu_sigsqr;
    F_n(1)               = vFE_2(ms_n(:,1),vFE_arg);
    dF                   = full(spm_cat(spm_diff(@(ms_n) vFE_2(ms_n,vFE_arg), ms_n(:,1),1)))';
    ddF                  = full(spm_cat(spm_diff(@(ms_n) vFE_2(ms_n,vFE_arg), ms_n(:,1),[1 1])'));

    % Newton iterations 1,2,3,...
    % ---------------------------------------------------------------------
    for i = 2:max_iter

        % backtracking Newton line search with Hessian modification for
        % variational expectation
```

```
        % ------------------------------------------------------------------
        fprintf('Fixed-form VB Newton Iteration %4.0f \n', i-1)

        % Newton search direction
        p_i                 = -ddF\dF;

        % Hessian modification if p_i is not a descent direction
        if ~(p_i'*dF < 0)

           % diagonal Hessian modification based on spectral decomposition
           ddF      = ddF + max(0,(delta - min(eig(ddF))))*eye(2);

           % re-evaluate Newton search direction
           p_i      = -ddF\dF;

        end

        % backtracking evaluation of the step length
        t_i                 = f_backtrack_vFE_2(ms_n(:,i-1), vFE_arg, dF, p_i, c, rho);

        % perform Newton update
        ms_n(:,i)           = ms_n(:,i-1) + t_i*p_i;

        % evaluation of the variational free function value, gradient, and Hessian at updated location
        F_n(i)              = vFE_2(ms_n(:,i),vFE_arg);
        dF                  = full(spm_cat(spm_diff(@(ms_n) vFE_2(ms_n,vFE_arg), ms_n(:,i),1)))';
        ddF                 = full(spm_cat(spm_diff(@(ms_n) vFE_2(ms_n,vFE_arg), ms_n(:,i),[1 1])'));

    end

    % prior and variational distribution visualization
    % ------------------------------------------------------------------------
    sigsqr_x   = linspace(1e-3 ,2e1 ,1e2);                                  % RV support
    subplot(2,3,4+s-1)
    hold on
    plot(sigsqr_x, lognpdf(sigsqr_x, mu_sigsqr  , sqrt(si_sigsqr))    , 'b')
    plot(sigsqr_x, lognpdf(sigsqr_x, ms_n(1,end), sqrt(ms_n(2,end)))  , 'r')
    plot(exp(mu_sigsqr), 0, 'o', 'MarkerFaceColor' , 'b','MarkerSize', 10)
    plot(exp(ms_n(1,end)), 0, 'o', 'MarkerFaceColor' , 'r', 'MarkerSize', 10)
    xlabel( '\sigma^2', 'FontSize', 20, 'FontName', 'Calibri Light')
    legend(['p(\sigma^2), H(p(\sigma^2)) = ' num2str(0.5 + log(2*pi*si_sigsqr)   + mu_sigsqr,2)  ], ...
           ['q(\sigma^2), H(q(\sigma^2)) = ' num2str(0.5 + log(2*pi*ms_n(2,end)) + ms_n(1,end),2)])
    set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

end


% variational free energy and Newton iteration visualization
% ------------------------------------------------------------------------
m_sigsqr            = linspace(-1.5 , 7   ,1e2);                        % scale variational parameter
s_sigsqr            = linspace(1e-2 ,1e-1,1e2);                         % shape variational parameter

% evaluate variational free energy  surface
FE_all = NaN(length(m_sigsqr), length(s_sigsqr));
for i = 1:length(m_sigsqr)
    for j = 1:length(s_sigsqr)
        FE_all(i,j) = vFE_2([m_sigsqr(i) s_sigsqr(j)]', vFE_arg);
    end
end

subplot(2,3,6)
hold on
surf(s_sigsqr, m_sigsqr, -FE_all, 'EdgeColor', 'None')
alpha(.6)
plot3(ms_n(2,:), ms_n(1,:), -F_n, 'ko--', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
xlim([s_sigsqr(1) s_sigsqr(end)])
ylim([m_sigsqr(1) m_sigsqr(end)])
ylabel('m_{\sigma^2}' ,  'FontName','Calibri Light', 'FontSize', 26)
xlabel('s_{\sigma^2}' ,  'FontName','Calibri Light', 'FontSize', 26)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
view([41 22])
grid on

end
% ------------------------------------------------------------------------
%                           Subfunctions
% ------------------------------------------------------------------------
function [hx]           = h_fun(theta)

% This function generates a "latent time series" of 11 data points given by
%
%             h : R -> R^10, theta |-> vec([1:10].^theta)
%
%   Inputs
%           theta   : scalar
%
%   Outputs
%           hx      : 10 x 1  array
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% compute output argument
hx  = ((1:10).^theta)';

end

function [vFE]          = vFE_1(m_theta,s_theta,vFE_1_arg)

% This function evaluates the variational free energy for the special case
% of a 1-dimensional parameter space and a known variance parameter
%
%   Inputs
%       m_theta    : 1 x 1 scalar  - variational expectation
%       s_theta    : 1 x 1 scalar  - variational variance
%       vFE_1_arg  : structure with fields
%           .y                     : n x 1 array   - data
%           .f                     : string        - function handle
%           .mu_theta              : 1 x 1 scalar  - prior expectation
%           .sigma_theta           : 1 x 1 scalar  - prior variance
%           .sigma                 : 1 x 1 scalar  - likelihood variance
%
%   Outputs
%           vFE                    : 1 x 1 scalar  - negative variational free energy
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% unpack input structure
y                       = vFE_1_arg.y;
h                       = vFE_1_arg.h;
mu_theta                = vFE_1_arg.mu_theta;
sig_theta               = vFE_1_arg.sig_theta;
sigma                   = vFE_1_arg.sigma;

% evaluation of the number data points and parameters
n                       = size(y,1);
p                       = size(m_theta,1);

% evalutate f(m_theta(i)) and Jf(m_theta_i)
h_m_theta               = h(m_theta);
Jh_m_theta              = spm_diff(h,m_theta,1);

% evaluation of the variational free energy value
T1                      = -(n/2)*log(2*pi)    - (n/2)*log(sigma)       - (1/2)*((1/sigma)*(y - h_m_theta)'*(y - h_m_theta) + (1/sigma)*(Jh_m_theta'*Jh_m_theta)*s_theta);
T2                      = -(p/2)*log(2*pi)    - (1/2)*log(sig_theta)   - ((1/2)*(1/sig_theta))*((m_theta - mu_theta)^2)    - (1/2)*(s_theta/sig_theta);
T3                      =  (1/2)*log(s_theta) + (p/2)*log(2*pi*exp(1));

% evaluate the negative free energy
vFE                     = -(T1 + T2 + T3);

end

function [t_k]          = f_backtrack_vFE_1(x_k, s_theta, vFE_1_arg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE_1
%
%   Inputs
%       x_k         : 1 x 1 array of function argument
%       s_theta     : 1 x 1 array of additional function argument
%       vFE_1_arg   : additional input argument structure for the function f_name
%       df_x_k      : n x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE_1(x_k, s_theta, vFE_1_arg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE_1(x_k + t_k*p_k, s_theta, vFE_1_arg);


% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE_1(x_k+t_k*p_k, s_theta, vFE_1_arg);
end
end

function [FE]           = vFE_2(ms_sigsqr,vFE_arg)

% This function evaluates the negative variational free energy function for
% the case of the estimation of the variance parameter in a linear
% regression model by means of a fixed-form variational approach with log
% normal prior and variational distributions
%
%   Inputs
%       ms_sigsqr   : 2 x 1 log-scale and shape parameters
%           vFE_arg     : structure with additional inputs with fields
%               .y              : n x 1 data vector
%               .mu_sigsqr      : scalar log-scale prior parameter
%               .si_sigsqr      : scalar shape prior parameter
%
%   Outputs
%           FE          : scalar negative variational free energy
```

```
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------------
% unpack and summarize input parameters
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
m_theta         = vFE_arg.m_theta;
s_theta         = vFE_arg.s_theta;
mu_sigsqr       = vFE_arg.mu_sigsqr;
si_sigsqr       = vFE_arg.si_sigsqr;
y               = vFE_arg.y;
h               = vFE_arg.h;

% evaluation of the number data points and parameters
n               = size(y,1);

% evalutate f(m_theta(i)) and Jf(m_theta_i)
h_m_theta       = h(m_theta);
Jh_m_theta      = spm_diff(h,m_theta,1);

% evaluate relevant variational free enerby terms
T1          = -(n/2)*log(2*pi)                  - (n/2)*m_sigsqr   - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2          = -(1/2)*log(2*pi*si_sigsqr) -  m_sigsqr       - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T3          =  (1/2) + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr;

% evaluate negative variational free energy
FE          = -(T1 + T2 + T3);

end

function [t_k]          = f_backtrack_vFE_2(x_k, vFE_arg, df_x_k, p_k, c ,rho)

% This function evaluates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k         : 1 x 1 array of function argument
%       vFE_arg     : additional variational energy input arguments
%       df_x_k      : n x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE_2(x_k,vFE_arg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE_2(x_k + t_k*p_k, vFE_arg);

% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || x_k(2) + t_k*p_k(2) < 0;
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE_2(x_k+t_k*p_k, vFE_arg);
end
end
```

PDDE_3.m – Second toy model (Figures 5 and 6)

```
function pdde_3_R1

% This function implements the estimation of an ERP-DCM toy-model for
% unknow variance parameter sigma^2.
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------

% -------------------------------------------------------------------------
%                           Initialization
% -------------------------------------------------------------------------
clc
close all
stream  = RandStream.getGlobalStream;reset(stream);                         % set the random number generator for reproducible restuts
                                                                            % plot intermediate results
% -------------------------------------------------------------------------
%                   Model Formulation and Data Sampling
% -------------------------------------------------------------------------
theta                   = [2 5]';                                           % true, but unknown, parameter
p                       = length(theta);                                    % cardinality of theta
h                       = @(theta) h_fun(theta);                            % function handle specification h : R^2 -> R^10
sigsqr                  = 1e2;                                              % true, but unknoknown, variance parameter
h_theta                 = h(theta);                                         % evaluate h
y                       = mvnrnd(h_theta,sigsqr*eye(length(h_theta)))';     % sample data

% -------------------------------------------------------------------------
%                   Free Energy surface visualization
% -------------------------------------------------------------------------
% m_theta space
m_theta_f_min           = 1.5;
m_theta_f_max           = 2.3;
m_theta_f_res           = 1e1;
m_theta_f_space         = linspace(m_theta_f_min,m_theta_f_max,m_theta_f_res);
m_theta_g_min           = 3;
m_theta_g_max           = 7;
m_theta_g_res           = 1e1;
m_theta_g_space         = linspace(m_theta_g_min,m_theta_g_max,m_theta_g_res);

% initialize figure
fig = figure;
set(fig, 'Color', [1 1 1])
titles = {'T1', 'T2', 'T3', 'vFE'};

% cycle over prior settings
% -------------------------------------------------------------------------
for s = 1:3

    % free energy arguments
    vfearg.y                = y;                                            % data
    vfearg.h                = h;                                            % model function

    switch s
        case 1
            vfearg.mu_theta         = [0 0]';                               % prior exectation parameter theta
            vfearg.sig_theta        = [1e6 0;0 1e6];                        % prior variance   parameter theta
            vfearg.mu_sigsqr        = 4.6;                                  % prior shape      parameter
            vfearg.si_sigsqr        = 1e-3;                                 % prior scale      parameter
            ms_sigsqr               = [vfearg.mu_sigsqr;vfearg.si_sigsqr];  % variational shape and scale parameter settings
        case 2
            vfearg.mu_theta         = [0 5]';                               % prior exectation parameter theta
            vfearg.sig_theta        = [1e6 0;0 1e-3];                       % prior variance   parameter theta
            vfearg.mu_sigsqr        = 4.6;                                  % prior shape      parameter
            vfearg.si_sigsqr        = 1e-3;                                 % prior scale      parameter
            ms_sigsqr               = [vfearg.mu_sigsqr;vfearg.si_sigsqr];  % variational shape and scale parameter settings
        case 3
            vfearg.mu_theta         = [0 5]';                               % prior exectation parameter theta
            vfearg.sig_theta        = [1e6 0;0 1e-3];                       % prior variance   parameter theta
            vfearg.mu_sigsqr        = 1e-3;                                 % prior shape      parameter
            vfearg.si_sigsqr        = 9.2;                                  % prior scale      parameter
            ms_sigsqr               = [vfearg.mu_sigsqr;vfearg.si_sigsqr];  % variational shape and scale parameter settings
    end

    % variational variance paramete evaluation
    Jh_m_theta  = spm_diff(h,vfearg.mu_theta,1);
    s_theta     = inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (inv(vfearg.sig_theta))));

    % variational free energy terms evaluation
    vFE_terms_m_theta       = NaN(m_theta_f_res,m_theta_g_res,4);
    for i = 1:m_theta_f_res
        for j = 1:m_theta_g_res
            [vFE_terms_all]   = vFE_terms([m_theta_f_space(i) m_theta_g_space(j)]',s_theta, ms_sigsqr, vfearg);
            vFE_terms_m_theta(i,j,1)    = vFE_terms_all.T1;
            vFE_terms_m_theta(i,j,2)    = vFE_terms_all.T2;
            vFE_terms_m_theta(i,j,3)    = vFE_terms_all.T3;
            vFE_terms_m_theta(i,j,4)    = vFE_terms_all.vFE;
        end
    end

    % visualization
    for t = 1:size(vFE_terms_m_theta,3)
        subplot(3,4,(s-1)*4+t)
        imagesc(m_theta_g_space, m_theta_f_space,vFE_terms_m_theta(:,:,t))
        title(titles{t}, 'FontName', 'Calibri Light', 'FontSize', 26, 'FontWeight', 'Normal')
        set(gca, 'FontName', 'Calibri Light', 'FontSize', 20)
        axis square
        colorbar
    end
end

% -------------------------------------------------------------------------
%                           Model inversion
% -------------------------------------------------------------------------
% initialize figure
fig = figure;
set(fig, 'Color', [1 1 1])

% fixed_form variational Bayes Newton algorithm parameters
% -------------------------------------------------------------------------
max_i                   = 10;                                               % maximal number of fixed-form VB Newton algorithm
max_j                   = 10;                                               % maximal number of Newton iterations for m_theta optimization
max_k                   = 10;                                               % maximal number of Newton iterations for (m_sigsqr, s_sigssqr) optimization
delta                   = sqrt(eps);                                        % Hessian modification constant
c                       = 1e-4;                                             % backtracking constant c
rho                     = 0.99;                                             % backtracking constant rho

% cycle over prior distributions (= probabilistic models)
% -------------------------------------------------------------------------
for sim = 1:3

switch sim
    case 1
        mu_theta        = [0 5]';                                           % prior exectation parameter theta
        sig_theta       = [1e6 0;0 1e-3];                                   % prior variance   parameter theta
        mu_si_sigsqr    = [4.6;1e-3];                                       % prior shape and scale parameter setting \sigma^2
    case 2
        mu_theta        = [0 5]';                                           % prior exectation parameter theta
        sig_theta       = [1e6 0;0 1e6];                                    % prior variance   parameter theta
        mu_si_sigsqr    = [4.6;1e-3];                                       % prior shape and scale parameter setting \sigma^2
    case 3
        mu_theta        = [0 5]';                                           % prior exectation parameter theta
        sig_theta       = [1e6 0;0 1e6];                                    % prior variance   parameter theta
        mu_si_sigsqr    = [2.3;1e-0];                                       % prior shape and scale parameter setting \sigma^2    end
end

% model estimation - iteration 0
% -------------------------------------------------------------------------
% initialization of the variational parameters to prior parameters
m_theta                 = mu_theta;
s_theta                 = sig_theta;
ms_sigsqr               = mu_si_sigsqr;
vfearg.y                = y;
vfearg.h                = h;
vfearg.mu_theta         = mu_theta;
vfearg.sig_theta        = sig_theta;
vfearg.mu_sigsqr        = mu_si_sigsqr(1);
vfearg.si_sigsqr        = mu_si_sigsqr(2);

% initialization and initial evaluation of the negative free energy
F                       = NaN(max_i);
F(1)                    = vFE(m_theta, s_theta, ms_sigsqr, vfearg);

% VBNL iterations 1,2,3,...
% -------------------------------------------------------------------------
for i = 2:max_i

    % User update
    % ---------------------------------------------------------------------
    fprintf('Fixed-form VB Newton Iteration %4.0f ... ', i-1)

    % Analytical update for theta_f variational variance parameter
    % ---------------------------------------------------------------------
    % evaluate the variational variance update equation for the current choice of m_theta
    Jh_m_theta      = spm_diff(h,m_theta,1);
    s_theta         = inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (inv(sig_theta))));

    % Newton line search for theta_f variational expectation parameter
    % ---------------------------------------------------------------------
    % iteration arrays
    m_theta_j       = NaN(p,max_j);
    F_m_theta_j     = NaN(1,max_j);

    % iterand and  objective function initialization
    m_theta_j(:,1)  = m_theta;
    F_m_theta_j(1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % deriatives initialization
    dF_m_theta      = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, 1)'));
    ddF_m_theta     = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, [1 1])'));

    % Newton iterations 1,2,3,...
    for j = 2:max_j

        % Newton search direction
        p_j = -ddF_m_theta\dF_m_theta;
```

```
        % Hessian modification if p_j is not a descent direction
        if ~(p_j'*dF_m_theta < 0)

            % diagonal Hessian modification based on spectral decomposition
            ddF_m_theta = ddF_m_theta + max(0,(delta - min(eig(ddF_m_theta))))*eye(p);

            % re-evaluate Newton search direction
            p_j       = -ddF_m_theta\dF_m_theta;
        end

        % backtracking evaluation of the step length
        t_j             = backtrack_vFE_m_theta(m_theta, s_theta, ms_sigsqr, vfearg, dF_m_theta, p_j,c,rho);

        % perform parameter update
        m_theta         = m_theta + t_j*p_j;

        % update remaining entries in free energy arrays
        dF_m_theta      = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, 1)'));
        ddF_m_theta     = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, [1 1])'));

        % record updates
        m_theta_j(:,j)    = m_theta;
        F_m_theta_j(j)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    end


    % Newton line search for sigma^2 variational parameters
    % -------------------------------------------------------------------
    % iteration arrays
    ms_sigsqr_k         = NaN(2,max_k);
    F_ms_sigsqr_k       = NaN(1,max_j);

    % iterand and  objective function initialization
    ms_sigsqr_k(:,1)    = ms_sigsqr;
    F_ms_sigsqr_k(1)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
    dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
    ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

    % Newton iterations 1,2,3,...
    for k = 2:max_k

        % Newton search direction
        p_k = -ddF_ms_sigsqr\dF_ms_sigsqr;

        % Hessian modification if p_j is not a descent direction
        if ~(p_k'*dF_ms_sigsqr < 0)

            % diagonal Hessian modification based on spectral decomposition
            ddF_ms_sigsqr = ddF_ms_sigsqr + max(0,(delta - min(eig(ddF_ms_sigsqr))))*eye(2);

            % re-evaluate Newton search direction
            p_k  = -ddF_ms_sigsqr\dF_ms_sigsqr;
        end

        % backtracking evaluation of the step length
        t_k = backtrack_vFE_ms_sigsqr(m_theta, s_theta, ms_sigsqr, vfearg, dF_ms_sigsqr, p_k,c,rho);

        % perform Newton update
        ms_sigsqr = ms_sigsqr + t_k*p_k;

        % update remaining entries in free energy arrays
        dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
        ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

        % record updates
        ms_sigsqr_k(:,k)    = ms_sigsqr;
        F_ms_sigsqr_k(k)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    end


    % free energy update
    % -------------------------------------------------------------------
    F(i) = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % user update
    % -------------------------------------------------------------------
    fprintf('vFE %6.3f  dvFE %6.3f \n', F(i), F(i) - F(i-1))

end

% -------------------------------------------------------------------------
%                             Visualization
% -------------------------------------------------------------------------

% evaluate the MAP fit
h_m_theta_MAP       = h_fun(m_theta);

% RV support sets
sigsqr_x            = linspace(1e-3,2e2    , 1e4);
theta_f_x           = linspace(  -.5,  2.5 , 1e4);
theta_g_x           = linspace(    3,  7   , 1e4);

% data, function h, and MAP fit
subplot(3,5,(sim-1)*5 + 1)
hold on
plot(1:10, h_theta )
plot(1:10, y , 'ko'    , 'MarkerSize', 6)
plot(1:10, h_m_theta_MAP , 'k--')
xlabel('x', 'FontSize', 22, 'FontName', 'Calibri Light')
if sim == 1
    legend('h(\theta)', 'y', 'h(m_{\theta}^{MAP})', 'Location', 'NorthWest')
end
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')

subplot(3,5,(sim-1)*5 + 2)
hold on
plot(theta_f_x, pdf('Normal', theta_f_x, mu_theta(1) , sqrt(sig_theta(1))), 'b');
plot(theta_f_x, pdf('Normal', theta_f_x, m_theta(1)  , sqrt(s_theta(1,1))), 'r--');
plot(mu_theta(1)        , 0, 'o', 'MarkerFaceColor' , 'b', 'MarkerSize', 8)
plot(m_theta(1)         , 0, 'o', 'MarkerFaceColor' , 'r', 'MarkerSize', 8);
xlabel('\theta_f', 'FontSize', 22, 'FontName', 'Calibri Light')
if sim == 1
    legend('p(\theta_f)', 'q(\theta_f)',  'Location', 'NorthWest')
end
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')
xlim([theta_f_x(1) theta_f_x(end)])

subplot(3,5,(sim-1)*5 + 3)
hold on
plot(theta_g_x, pdf('Normal', theta_g_x, mu_theta(2) , sqrt(sig_theta(2,2))), 'b');
plot(theta_g_x, pdf('Normal', theta_g_x, m_theta(2)  , sqrt(s_theta(2,2)))  , 'r--');
plot(mu_theta(2)        , 0, 'o', 'MarkerFaceColor' , 'b', 'MarkerSize', 8)
plot(m_theta(2)         , 0, 'o', 'MarkerFaceColor' , 'r', 'MarkerSize', 8);
xlabel('\theta_g', 'FontSize', 22, 'FontName', 'Calibri Light')
if sim == 1
    legend('p(\theta_g)', 'q(\theta_g)',  'Location', 'NorthEast')
end
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')
xlim([theta_g_x(1) theta_g_x(end)])


% sigma^2 prior and posterior
subplot(3,5,(sim-1)*5 + 4)
hold on
plot(sigsqr_x, lognpdf(sigsqr_x, mu_si_sigsqr(1) , sqrt(mu_si_sigsqr(2)))    , 'b')
plot(sigsqr_x, lognpdf(sigsqr_x, ms_sigsqr(1)    , sqrt(ms_sigsqr(2)))      , 'r--')
plot(exp(mu_si_sigsqr(1) + .5*mu_si_sigsqr(2))   , 0, 'o', 'MarkerFaceColor' , 'b', 'MarkerSize', 10)
plot(exp(ms_sigsqr(1)    + .5*ms_sigsqr(2))      , 0, 'o', 'MarkerFaceColor' , 'r', 'MarkerSize', 10)
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')
if sim == 1
    legend('p(\sigma^2)', 'q(\sigma^2)',  'Location', 'NorthEast')
end
xlabel('\sigma^2', 'FontSize', 22, 'FontName', 'Calibri Light')
xlim([0 sigsqr_x(end)])

% variational free energy
subplot(3,5,(sim-1)*5 + 5)
hold on
plot(-F(2:end), 'o-')
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')
xlim([0 max_i])
title('F(m_\theta^{(i)}, \Sigma_\theta^{(i)}, m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)})', 'FontName', 'Calibri Light', 'FontWeight', 'Normal')
xlabel('i', 'FontSize', 22, 'FontName', 'Calibri Light')

end
end

% -------------------------------------------------------------------------
%                             Subfunctions
% -------------------------------------------------------------------------
function [f_theta_f]    = f_fun(theta_f)

% This function evaluates the ERP-DCM toy-model function
%
%             f : R -> R^10, theta_f |-> vec([1:10].^theta_f)
%
%   Inputs
%          theta_f     : scalar
%
%   Outputs
%          f_theta_f   : 10 x 1  array
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% compute output argument
f_theta_f  = ((1:10).^theta_f)';

end

function [g_theta_g]    = g_fun(theta_g)

% This function evaluates the ERP-DCM toy-model function
%
%           g : R -> R, theta_g |-> theta_g
%
%   Inputs
%          theta_g     : scalar
%
```

```
%   Outputs
%           g_theta_g   : scalar
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% compute output argument
g_theta_g = theta_g;

end

function [h_theta]      = h_fun(theta)

% This function evaluates the ERP-DCM expectation parameter generating
% function
%
%     h : (theta) |-> h(theta) := g(theta_g)*f(theta_f)
%
%      Inputs
%          theta       : 2 x 1 function parameter
%
%      Outputs
%          h_theta     : evaluated function
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% subfunction parameters
theta_f         = theta(1);
theta_g         = theta(2);

% evaluate component functions f and g
f_theta_f       = f_fun(theta_f);
g_theta_g       = g_fun(theta_g);

% evaluate concatenated function h
h_theta         = g_theta_g*f_theta_f;

end

function [vFE]          = vFE(m_theta,s_theta, ms_sigsqr, vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : 2 x 1 variational expectation for theta
%       s_theta    : 2 x 2 variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters   for sigma^2
%       vfearg     : additional argument structure with fields
%           .y                  : n x 1 array   - data
%           .h                  : string        - function handle
%           .mu_theta           : scalar prior expectation for theta
%           .sig_theta          : scalar prior variance for theta
%           .mu_sigsqr          : scalar prior scale parameter for sigma^2
%           .si_sigsqr          : scalar prior shape parameter for sigma^2
%   Outputs
%           vFE                 : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
h               = vfearg.h;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta_f(i)) and J^h(m_theta_f(i))
h_m_theta       = h(m_theta);
Jh_m_theta      = spm_diff(h,m_theta,1);

% evaluation of the variational free energy value
T1              = -(n/2)*log(2*pi)              - (n/2)*m_sigsqr            - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2              = -(p/2)*log(2*pi)              - (1/2)*log(det(sig_theta))  - (1/2)*(m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
T3              = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                     - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T4              =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
T5              =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                                  ;

% evaluate the negative free energy
vFE             = -(T1 + T2 + T3 + T4 + T5);

end

function [vFE]  = vFE_terms(m_theta,s_theta, ms_sigsqr, vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : 2 x 1 variational expectation for theta
%       s_theta    : 2 x 2 variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters   for sigma^2
%       vfearg     : additional argument structure with fields
%           .y                  : n x 1 array   - data
%           .h                  : string        - function handle
%           .mu_theta           : scalar prior expectation for theta
%           .sig_theta          : scalar prior variance for theta
%           .mu_sigsqr          : scalar prior scale parameter for sigma^2
%           .si_sigsqr          : scalar prior shape parameter for sigma^2
%   Outputs
%           vFE                 : structure with fields
%               .T1A                : scalar accuracy term A
%               .T1B                : scalar accuracy term B
%               .T2                 : scalar prior influence m_theta
%               .T3                 : scalar prior influence ms_sigsqr
%               .T4                 : scalar entropy q(theta)
%               .T5                 : scalar entropy q(sigma^2)
%               .vFE                : scalar variational free energy
%
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
h               = vfearg.h;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta_f(i)) and J^h(m_theta_f(i))
h_m_theta       = h(m_theta);
Jh_m_theta      = spm_diff(h,m_theta,1);

% evaluation of the variational free energy value
vFE.T1          = -(n/2)*log(2*pi)  - (n/2)*m_sigsqr   - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta));
vFE.T2          = -(1/2)*exp(-m_sigsqr +.5*s_sigsqr)*(trace((Jh_m_theta'*Jh_m_theta)*s_theta));
vFE.T3          = -(p/2)*log(2*pi)              - (1/2)*log(det(sig_theta))  - (1/2)*(m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
vFE.T4          = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                     - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
vFE.T5          =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
vFE.T6          =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                                  ;

% evaluate the positive free energy
vFE.vFE         = vFE.T1 + vFE.T2 + vFE.T3 + vFE.T4 + vFE.T5 + vFE.T6;

end

function [t_k]          = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsqr, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k         : p x 1     array of function argument
%       s_theta     : p x p     array of additional vFE arguments
%       ms_sigsqr   : 2 x 1     array of additional vFE arguments
%       vfearg      : additional input for vFE
%       df_x_k      :p_f x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(x_k, s_theta, ms_sigsqr, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(x_k + t_k*p_k, s_theta, ms_sigsqr, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(x_k+t_k*p_k, s_theta, ms_sigsqr, vfearg);
end
end

function [t_k]          = backtrack_vFE_ms_sigsqr(m_theta, s_theta, x_k, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       m_theta     : p x 1   array of additional vFE arguments
%       s_theta     : p x p array of additional vFE arguments
```

```
%       x_k         : 2 x 1     array of function arguments
%       vfearg      : additional input for vFE
%       df_x_k      : 2 x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(m_theta, s_theta, x_k, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(m_theta, s_theta, x_k + t_k*p_k, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)|| x_k(2) + t_k*p_k(2) < 0;
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(m_theta, s_theta, x_k+t_k*p_k, vfearg);
end
end
```

PDDE_4.m – ERP-DCM input connectivity estimation (Figures 7 and 8)

```
function pdde_4_R1

% This function implements the estimation of latent input connectivity in
% a basic delay differential equation ERP model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
clc
close all

% set the random number generator for reproducible sampling results
reset(RandStream.getGlobalStream)

% invoke Fieldtrip
addpath(genpath(fullfile(pwd, 'Fieldtrip')))

% -------------------------------------------------------------------------
%                           Model Formulation
% -------------------------------------------------------------------------

% neural model structure and parameters
% -------------------------------------------------------------------------
% structural aspects and fixed parameters
m_f.f               = @spm_fx_erp;                                      % neural evolution function
m_f.ns              = 200;                                              % number of time bins
m_f.dt              = 0.001;                                            % time bin length (sec)
m_f.pst             = (0:m_f.ns-1)*m_f.dt;                              % peristimulus time (msec)
m_f.nt              = length(m_f.pst);                                  % number of time points
m_f.n               = 2;                                                % number of neural masses
m_f.m               = 9;                                                % number of neural states per neural mass
m_f.ons             = -10;                                              % system input onset (ms)
m_f.dur             = 1;                                                % duration (width) of the input function
m_f.x               = sparse(m_f.n,m_f.m);                              % initial condition
m_f.R               = [ 0.16 -0.22];                                    % input function parameters rho_1 and rho_2
m_f.S               = [-9e-4  0.08];                                    % static nonlinearity (activation function) parameters
m_f.T               = [ 0.16  0.13;  0.07 -0.82];                       % source-specific excitatory and inhibitory time constants
m_f.G               = [-0.11  0.45;  0.02 -0.63];                       % source-specific excitatory and inhibitory receotor densities
m_f.H               = [ 0.02 -0.07   0.18 -0.17];                       % source-independent intrinsic connectivity parameters
m_f.D               = [ 0    -0.52; -0.06  0   ];                       % between source delay parameters
m_f.A{1}            = [0 0; 1 0];                                       % forward  connectivity
m_f.A{2}            = [0 0; 0 0];                                       % backward connectivity
m_f.A{3}            = zeros(2,2);                                       % lateral connectivity
m_f.C               = [NaN NaN]';                                       % input connectivity

% forward model structure and parameters
% -------------------------------------------------------------------------
% load an existing dipole model
load(fullfile(cd, 'Data\SIM\dip_model_struct.mat'))
m_g.vol             = dip_model_struct.vol;                             % forward model specification
m_g.sens            = dip_model_struct.sens ;                           % sensor space specification
m_g.elab            = dip_model_struct.sens.label;                      % electrode labels
m_g.ne              = numel(m_g.elab);                                  % number of electrodes
m_g.X0              = zeros(m_f.ns*m_g.ne, 3*m_g.ne);                   % confound design matrix
m_g.Lpos            = [42 -31 58; 54 -22 18]';                          % dipole coordinate specifications
m_g.L               = [0.02 -0.05 -0.04; 0.17 -0.50 -0.17]';            % dipole moment specification
m_g.J               = [-0.1 0 0 0 0 0 -0.46 0 1];                       % state weighting specification

% likelihood model parameters
% -------------------------------------------------------------------------
n                   = m_f.nt*m_g.ne;                                    % number of data points

% prior formulation
% -------------------------------------------------------------------------
mu_theta            = [0 0]';                                           % prior expectation parameter setting theta
sig_theta           = [1e3 0; 0 1e3];                                   % prior variancen parameter setting theta
mu_si_sigsqr        = [4.6;1];                                          % prior shape and scale parameter setting \sigma^2

% fixed form variational Bayes Newton algorithm parameters
% -------------------------------------------------------------------------
max_i               = 5;                                                % maximal number of fixed-form VB Newton algorithm
max_j               = 8;                                                % maximal number of Newton iterations for m_theta optimization
max_k               = 8;                                                % maximal number of Newton iterations for (m_sigsqr, s_sigssqr) optimization
delta               = sqrt(eps);                                        % Hessian modification constant
c                   = 1e-4;                                             % backtracking constant c
rho                 = 0.9;                                              % backtracking constant rho
vareps              = 1e0;                                              % gradient norm convergence criterion
vardelta            = 1e-3;                                             % parameter norm change convergence criterion

% -------------------------------------------------------------------------
%                           Model Realization
% -------------------------------------------------------------------------
theta               = [1 0]';                                           % true, but unknown, input connectivity
sigsqr              = 1e-1;                                             % true, but unknown, data variance
p                   = length(theta);                                    % number of parameters
h_theta_fg          = erp_h(m_f,m_g,theta);                             % data expectation
y                   = mvnrnd(h_theta_fg, sigsqr*eye(n))';               % data sampling

% -------------------------------------------------------------------------
%                           Data Visualization
% -------------------------------------------------------------------------
% initialize figure
fig = figure;
set(fig, 'Color', [1 1 1])

% evaluate latent data
[f_theta_f_u]       = erp_f(m_f,theta);
f_theta_f_s1        = f_theta_f(:,1:2:end-1);
f_theta_f_s2        = f_theta_f(:,2:2:end);

% Input function u
% -------------------------------------------------------------------------
subplot(2,3,1)
plot(m_f.pst,u)
xlim([m_f.pst(1) m_f.pst(end)])
title('u', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% latent variable evolution
% -------------------------------------------------------------------------
subplot(2,3,2)
plot(m_f.pst,f_theta_f_s1)
xlim([m_f.pst(1) m_f.pst(end)])
title('x^{(1)}', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-10 40])

subplot(2,3,5)
plot(m_f.pst,f_theta_f_s2)
xlim([m_f.pst(1) m_f.pst(end)])
title('x^{(2)}', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-1 5])

% observed variables evolution
% -------------------------------------------------------------------------
subplot(2,3,3)
imagesc(m_f.pst, 1:numel(m_g.elab),spm_unvec(y, NaN(m_g.ne,m_f.nt)))
xlim([m_f.pst(1) m_f.pst(end)])
cb = colorbar;
title('y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel(cb, '\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('Electrodes', 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% selected electrodes
subplot(2,3,6)
eoi     = [47 49:51];
eoilab      = m_g.elab(eoi);
data        = spm_unvec(y, NaN(m_g.ne,m_f.nt));
plot(m_f.pst,data(eoi,:))
xlim([m_f.pst(1) m_f.pst(end)])
title('y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
legend(eoilab)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% -------------------------------------------------------------------------
%                           VBNL iteration 0
% -------------------------------------------------------------------------
% initialization of the variational parameters to prior parameters
m_theta             = mu_theta;
s_theta             = sig_theta;
ms_sigsqr           = mu_si_sigsqr;
vfearg.y            = y;
vfearg.m_f          = m_f;
vfearg.m_g          = m_g;
vfearg.mu_theta     = mu_theta;
vfearg.sig_theta    = sig_theta;
vfearg.mu_sigsqr    = mu_si_sigsqr(1);
vfearg.si_sigsqr    = mu_si_sigsqr(2);
vfearg.theta        = theta;

% parameter iteration arrays
s_theta_i           = NaN(p,p,max_i-1);
F_s_theta_i         = NaN(1,max_i-1);
m_theta_j           = NaN(p,max_j,max_i-1);
F_m_theta_j         = NaN(max_j,max_i-1);
ms_sigsqr_k         = NaN(2,max_k,max_i-1);
F_ms_sigsqr_k       = NaN(max_k,max_i-1);

% initialization and initial evaluation of the negative free energy
F                   = NaN(max_i-1);
F(1)                = vFE(m_theta, s_theta, ms_sigsqr, vfearg);

% -------------------------------------------------------------------------
%                           VBNL iterations 1,2,3,...
% -------------------------------------------------------------------------
for i = 2:max_i
```

```
    % User update
    % --------------------------------------------------------------------
    fprintf('Fixed-form VB Newton Iteration %4.0f ... \n', i-1)

    % Analytical update for theta_f variational variance parameter
    % --------------------------------------------------------------------
    % evaluate the variational variance update equation for the current choice of m_theta
    fprintf('   Estimating q(m_theta) variance ... ')
    Jh_m_theta          = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g, spm_unvec(m_theta, theta)),m_theta,1)));
    s_theta             = spm_inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (spm_inv(sig_theta))));

    % save iterand and objective function
    s_theta_i(:,:,i-1)  = s_theta;
    F_s_theta_i(i-1)    =  vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % inform user
    fprintf('finished. \n')

    % Newton line search for theta_f variational expectation parameter
    % --------------------------------------------------------------------
    fprintf('   Estimating q(m_theta) expectation ... ')

    % save iterand and objective function
    m_theta_j(:,1,i-1)  = m_theta;
    F_m_theta_j(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % numerical gradient and Hessian of the negative free energy
    dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
    ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_m_theta) >= vareps
        for j = 2:max_j

            % gradient search direction
            p_j = -ddF_m_theta\dF_m_theta;

            % Hessian modification if p_j is not a descent direction
            if ~(p_j'*dF_m_theta < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_m_theta = ddF_m_theta + max(0,(delta - min(eig(ddF_m_theta)))*eye(p);

                % re-evaluate Newton search direction
                p_j = -ddF_m_theta\dF_m_theta;
            end

            % backtracking evaluation of the step length
            t_j                 = backtrack_vFE_m_theta(m_theta, s_theta, ms_sigsqr, vfearg, dF_m_theta, p_j,c,rho);

            % perform parameter update
            m_theta             = m_theta + t_j*p_j;

            % numerical gradient and Hessian of the negative free energy
            dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
            ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

            % record updates
            m_theta_j(:,j,i-1)    = m_theta;
            F_m_theta_j(j,i-1)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_m_theta) < vareps || norm(m_theta_j(:,j,i-1) - m_theta_j(:,j-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished.\n')

    % Newton line search for sigma^2 variational parameters
    % --------------------------------------------------------------------
    fprintf('   Estimating q(sigma^2) parameters  ... ')

    % save iterand and objective function
    ms_sigsqr_k(:,1,i-1)  = ms_sigsqr;
    F_ms_sigsqr_k(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
    dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
    ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_ms_sigsqr) > vareps
        for k = 2:max_k

            % Newton search direction
            p_k = -ddF_ms_sigsqr\dF_ms_sigsqr;

            % Hessian modification if p_j is not a descent direction
            if ~(p_k'*dF_ms_sigsqr < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_ms_sigsqr = ddF_ms_sigsqr + max(0,(delta - min(eig(ddF_ms_sigsqr)))*eye(2);

                % re-evaluate Newton search direction
                p_k  = -ddF_ms_sigsqr\dF_ms_sigsqr;
            end

            % backtracking evaluation of the step length
            t_k = backtrack_vFE_ms_sigsqr(m_theta, s_theta, ms_sigsqr, vfearg, dF_ms_sigsqr, p_k,c,rho);

            % perform Newton update
            ms_sigsqr           = ms_sigsqr + t_k*p_k;

            % update remaining entries in free energy arrays
            dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
            ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

            % record updates
            ms_sigsqr_k(:,k,i-1)  = ms_sigsqr;
            F_ms_sigsqr_k(k,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_ms_sigsqr) < vareps || norm(ms_sigsqr_k(:,k,i-1) - ms_sigsqr_k(:,k-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished. \n')

    % free energy update
    % --------------------------------------------------------------------
    F(i) = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % user update
    % --------------------------------------------------------------------
    fprintf('vFE %6.3f  dvFE %6.3f \n', F(i), F(i) - F(i-1))
end

% --------------------------------------------------------------------
%                       Algorithm Visualization
% --------------------------------------------------------------------
% initialize figure
h = figure;
set(h, 'Color', [1 1 1])

% m_theta prior and approximate posterior
% --------------------------------------------------------------------
res         = 2e2;
c_1_min     = -2;
c_1_max     =  2;
c_1_res     = res;
c_1         = linspace(c_1_min, c_1_max, c_1_res);
c_2_min     = -2;
c_2_max     =  2;
c_2_res     = res;
c_2         = linspace(c_2_min, c_2_max, c_2_res);

% prior p(c)
p_c = NaN(length(c_1),length(c_2));
for i = 1:length(c_1)
    for j = 1:length(c_2)
        p_c(i,j) = mvnpdf(mu_theta',[c_1(i) c_2(j)],sig_theta);
    end
end
subplot(2,3,1)
hold on
imagesc(c_2,c_1,p_c)
plot(theta(2), theta(1),  'k+', 'MarkerSize', 8, 'MarkerFaceColor', 'k')
set(gca, 'FontSize', 12, 'FontName', 'Calibri Light')
ylabel('c_1', 'FontName', 'Calibri Light', 'FontSize', 24, 'Rotation', 0)
xlabel('c_2', 'FontName', 'Calibri Light', 'FontSize', 24)
title('p(c)', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
axis([c_2(1) c_2(end) c_1(1) c_1(end)])
colorbar
axis square
set(gca, 'FontSize', 20)

% approximate posterior q(c)
c_1_min     = .999;
c_1_max     = 1.004;
c_1_res     = res;
c_1         = linspace(c_1_min, c_1_max, c_1_res);
c_2_min     = -.001;
c_2_max     =  .0005;
c_2_res     = res;
c_2         = linspace(c_2_min, c_2_max, c_2_res);
q_c         = NaN(length(c_1),length(c_2));
for i = 1:length(c_1)
    for j = 1:length(c_2)
        q_c(i,j) = mvnpdf(m_theta',[c_1(i) c_2(j)],s_theta);
    end
end
subplot(2,3,2)
hold on
imagesc(c_2, c_1, q_c)
plot(theta(2), theta(1),  'k+', 'MarkerSize', 8, 'MarkerFaceColor', 'k')
set(gca, 'FontSize', 12, 'FontName', 'Calibri Light')
xlabel('c_2', 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'ytick', [.999 1.000 1.001 1.002 1.003])
```

```
title('q(C)', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
set(gca, 'FontSize', 20)
colorbar
axis([c_2(1) c_2(end) c_1(1) c_1(end)])
axis square

% sigma^2 prior and posterior
% -------------------------------------------------------------------------
sigsqr_pri      = linspace(1e-3,6e2,2e2);
p_sigsqr_pri    = lognpdf(sigsqr_pri, mu_si_sigsqr(1) , sqrt(mu_si_sigsqr(2)));
sigsqr_pos      = linspace(1e-6,0.2,1e2);
p_sigsqr_pos    = lognpdf(sigsqr_pos, ms_sigsqr(1) , sqrt(ms_sigsqr(2)));

subplot(2,3,3)
hold on
line(sigsqr_pri,p_sigsqr_pri, 'Color','b')
ax1             = gca; % current axes
ax1.XColor      = 'b';
ax1.YColor      = 'b';
ax1.FontName    = 'Calibri Light';
ax1.FontSize    = 20;
ax1_pos         = ax1.Position;
xlabel('\sigma^2', 'FontName', 'Calibri Light', 'FontSize', 24, 'Color', 'k')
ax2             = axes('Position',ax1_pos,'XAxisLocation','top','YAxisLocation','right','Color','None', 'FontName', 'Calibri Light', 'FontSize', 20);
ax2.XColor      = 'r';
ax2.YColor      = 'r';
line(sigsqr_pos,p_sigsqr_pos,'Parent',ax2, 'Color','r')


% m_theta variational free energy ascent on first VBNL iteration
% -------------------------------------------------------------------------
res             = 3e1;
m_theta_1_max   = 1.5;
m_theta_1_min   = -.5;
m_theta_1_res   = res;
m_theta_1_space  = linspace(m_theta_1_min,m_theta_1_max,m_theta_1_res);
m_theta_2_max   = 0.2;
m_theta_2_min   = -.2;
m_theta_2_res   = res;
m_theta_2_space  = linspace(m_theta_2_min,m_theta_2_max,m_theta_2_res);
vFE_m_theta     = NaN(res,res);
for i = 1:res
    for j = 1:res
        vFE_m_theta(i,j) = vFE([m_theta_1_space(i) m_theta_2_space(j)]', s_theta_i(:,:,1), mu_si_sigsqr, vfearg);
    end
end
subplot(2,3,4)
hold on
surf(m_theta_2_space, m_theta_1_space, -vFE_m_theta, 'EdgeColor', 'None')
plot3(m_theta_j(2,:,1), m_theta_j(1,:,1),  -F_m_theta_j(:,1), 'ko-', 'MarkerFaceColor' ,'k')
axis tight
alpha(.6)
view([-30 60])
xlabel('m_{\theta_2}', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('m_{\theta_1}', 'FontSize', 24, 'FontName', 'Calibri Light')
title('m_\theta^{(i)}', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')


% ms_sigsqr variational free energy ascent on first VBNL iteration
% -------------------------------------------------------------------------
m_sigsqr_max    = 5;
m_sigsqr_min    = -4;
m_sigsqr_res    = res;
m_sigsqr_space  = linspace(m_sigsqr_min,m_sigsqr_max,m_sigsqr_res);
s_sigsqr_max    = 1e-6;
s_sigsqr_min    = 1;
s_sigsqr_res    = res;
s_sigsqr_space  = linspace(s_sigsqr_min,s_sigsqr_max,s_sigsqr_res);
vFE_ms_sigsqr   = NaN(res,res);
for i = 1:res
    for j = 1:res
        vFE_ms_sigsqr(i,j) = vFE(m_theta_j(:,2,1), s_theta_i(:,:,1), [m_sigsqr_space(i) s_sigsqr_space(j)]', vfearg);
    end
end
subplot(2,3,5)
hold on
surf(s_sigsqr_space, m_sigsqr_space, -vFE_ms_sigsqr, 'EdgeColor', 'None')
plot3(ms_sigsqr_k(2,:,1), ms_sigsqr_k(1,:,1),  -F_ms_sigsqr_k(:,1), 'ko-', 'MarkerFaceColor' ,'k')
axis tight
alpha(.6)
view([-45 77])
xlabel('s_{\sigma^2}', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('m_{\sigma^2}', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
title('m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)}', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 24)

% variational free energy
subplot(2,3,6)
hold on
plot(-F(2:end), 'ko-', 'MarkerFaceColor', 'k')
xlim([0 max_i])
title('F(m_\theta^{(i)}, \Sigma_\theta^{(i)}, m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)})', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 20)
xlabel('i', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')


% remove Fieldtrip
% -------------------------------------------------------------------------
rmpath(genpath(fullfile(pwd, 'Fieldtrip')))

end

% -------------------------------------------------------------------------
%                               Subfunctions
% -------------------------------------------------------------------------
function [h_theta_fg]   = erp_h(m_f,m_g,theta)

% This function evaluates the expectation parameter generating function
%
% h : (theta_f,theta_g) |-> h(theta_f,theta_g) := vec(g(theta_g)*f(theta_f)
%
% of the delay differential equation model for ERPs
%
%       Inputs
%           m_f         : latent neural model structure
%           m_g         : forward model structure
%           theta       : model parameters
%
%       Outputs
%           h_theta_fg  : evaluated function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluate component functions f and g
[f_theta_f, u]  = erp_f(m_f,theta);
g_theta_g       = erp_g(m_g);

% evaluate concatenated function h
h_theta_fg      = 1e4*spm_vec(g_theta_g*f_theta_f');

end

function [f_theta_f,u]  = erp_f(m_f,vartheta)

% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_gen_erp.m of
% the SPM12 distribution. Based on a given parameter setting this function
% generates an input time-course and  integrates the resulting delay
% differential equation system.
%
%       Inputs
%           m_f         : latent neural model structure and fixed parameters
%           theta       : free latent neural model parameter
%
%       Outputs
%           f_theta_f   : evaluated latent neural model
%           u           : evaluated system input function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.R             = m_f.R;
theta.S             = m_f.S;
theta.T             = m_f.T;
theta.G             = m_f.G;
theta.H             = m_f.H;
theta.D             = m_f.D;
theta.A{1}          = m_f.A{1};
theta.A{2}          = m_f.A{2};
theta.A{3}          = m_f.A{3};
theta.C             = m_f.C;

% free parameters
theta.C(1)          = vartheta(1);
theta.C(2)          = vartheta(2);

% evaluate peri-stimulus time input function
t                   = ((1:m_f.ns)*m_f.dt)*1000;
delay               = m_f.ons(1)+128*theta.R(1,1);
scale               = m_f.dur(1)*exp(theta.R(1,2));
u                   = 32*exp(-(t - delay).^2/(2*scale^2))';

% integrate systen
x                   = m_f.x                                             ; % initial condition
f                   = m_f.f                                             ; % function handle @spm_fx_erp
dt                  = m_f.dt                                            ; % integration time bin
[fx,dfdx,D]         = f(x,u(1),theta,m_f)                             ; % dx(t)/dt and Jacobian df/dx and check for delay operator
p                   = max(abs(real(eig(full(dfdx)))))                   ; %
N                   = ceil(max(1, dt*p*2))                              ; %
n                   = spm_length(x)                                     ; %
Q                   = (spm_expm(dt*D*dfdx/N) - speye(n,n))*spm_inv(dfdx)    ; %
v                   = spm_vec(x)                                        ; % initialize state
y                   = NaN(length(v), length(u))                         ; % initialize state time-course

% cycle over time-steps
for i = 1:size(u,1)

    % update dx = (expm(dt*J) - I)*inv(J)*f(x,u)
    for j = 1:N
        v = v + Q*f(v,u(i),theta,m_f);
    end
    y(:,i) = v;
```

```
end

% transpose
f_theta_f      = y';

end

function [f,J,Q]        = spm_fx_erp(x,u,P,M)

% state equations for a neural mass model of erps
% FORMAT [f,J,D] = spm_fx_erp(x,u,P,M)
% FORMAT [f,J]   = spm_fx_erp(x,u,P,M)
% FORMAT [f]     = spm_fx_erp(x,u,P,M)
% x      - state vector
%   x(:,1) - voltage (spiny stellate cells)
%   x(:,2) - voltage (pyramidal cells) +ve
%   x(:,3) - voltage (pyramidal cells) -ve
%   x(:,4) - current (spiny stellate cells)    depolarizing
%   x(:,5) - current (pyramidal cells)         depolarizing
%   x(:,6) - current (pyramidal cells)         hyperpolarizing
%   x(:,7) - voltage (inhibitory interneurons)
%   x(:,8) - current (inhibitory interneurons) depolarizing
%   x(:,9) - voltage (pyramidal cells)
%
% f      - dx(t)/dt  = f(x(t))
% J      - df(t)/dx(t)
% D      - delay operator dx(t)/dt = f(x(t - d))
%                                  = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
%  M.pF.E = [32 16 4];          % extrinsic rates (forward, backward, lateral)
%  M.pF.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2 g3, g4)
%  M.pF.D = [2 16];             % propogation delays (intrinsic, extrinsic)
%  M.pF.G = [4 32];             % receptor densities (excitatory, inhibitory)
%  M.pF.T = [8 16];             % synaptic constants (excitatory, inhibitory)
%  M.pF.R = [1 1/2];            % parameter of static nonlinearity
%
%__________________________________________________________________________
% David O, Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neuronal dynamics. NeuroImage 20: 1743-1755
%__________________________________________________________________________
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging

% Karl Friston
% $Id: spm_fx_erp.m 5369 2013-03-28 20:09:27Z karl $

% get dimensions and configure state variables
%--------------------------------------------------------------------------
n       = length(P.A{1});         % number of sources
x       = spm_unvec(x,M.x);       % neuronal states

% [default] fixed parameters
%--------------------------------------------------------------------------
E       = [1 1/2 1/8]*32;                 % extrinsic rates (forward, backward, lateral)
G       = [1 4/5 1/4 1/4]*128;            % intrinsic rates (g1 g2 g3 g4)
D       = [2 16];                         % propagation delays (intrinsic, extrinsic)
H       = [4 32];                         % receptor densities (excitatory, inhibitory)
T       = [8 16];                         % synaptic constants (excitatory, inhibitory)
R       = [2 1/3];                        % parameters of static nonlinearity

% test for free parameters on intrinsic connections
%--------------------------------------------------------------------------
G       = G.*exp(P.H);
G       = ones(n,1)*G;

% no exponential transforms to foster parameter identifiability
% -------------------------------------------------------------------------
A{1}    = P.A{1}*E(1);
A{2}    = P.A{2}*E(2);
A{3}    = P.A{3}*E(3);
C       = P.C;

% intrinsic connectivity and parameters
%--------------------------------------------------------------------------
Te      = T(1)/1000*exp(P.T(:,1));        % excitatory time constants
Ti      = T(2)/1000*exp(P.T(:,2));        % inhibitory time constants
He      = H(1)*exp(P.G(:,1));             % excitatory receptor density
Hi      = H(2)*exp(P.G(:,2));             % inhibitory receptor density

% pre-synaptic inputs: s(V)
%--------------------------------------------------------------------------
S       = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
%--------------------------------------------------------------------------
U       = C*u(:)*2;

% State: f(x)
%==========================================================================
f(:,1) = x(:,4)                                                          ; % voltage change (spiny stellate cells)
f(:,2) = x(:,5)                                                          ; % positive voltage change (pyramidal cells) +ve
f(:,3) = x(:,6)                                                          ; % negative voltage change (pyramidal cells) -ve
f(:,4) = (He.*((A{1} + A{3})*S(:,9) + G(:,1).*S(:,9) + U) - 2*x(:,4) - x(:,1)./Te)./Te   ; % current change (spiny stellate cells)    depolarizing
f(:,5) = (He.*((A{2} + A{3})*S(:,9) + G(:,2).*S(:,1)) - 2*x(:,5) - x(:,2)./Te)./Te      ; % current change (pyramidal cells)         depolarizing
f(:,6) = (Hi.*G(:,4).*S(:,7) - 2*x(:,6) - x(:,3)./Ti)./Ti                ; % current change (pyramidal cells)         hyperpolarizing
f(:,7) = x(:,8)                                                          ; % voltage change (inhibitory interneurons)
f(:,8) = (He.*((A{2} + A{3})*S(:,9) + G(:,3).*S(:,9)) - 2*x(:,8) - x(:,7)./Te)./Te      ; % current change(inhibitory interneurons) depolarizing
f(:,9) = x(:,5) - x(:,6)                                                 ; % voltage (pyramidal cells)


% vectorize
f       = spm_vec(f);

% avoid infinite recursion of spm_diff
if nargout < 2
    return
end

% Jacobian
%==========================================================================
J  = spm_diff(M.f,x,u,P,M,1);

% delays
%==========================================================================
% Delay differential equations can be integrated efficiently (but
% approximately) by absorbing the delay operator into the Jacobian
%
%    dx(t)/dt      = f(x(t - d))
%                  = Q(d)f(x(t))
%
%    J(d)          = Q(d)df/dx
%--------------------------------------------------------------------------
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,n)).*De;
Di = (1 - speye(9,9)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D  = Di + De;

% Implement: dx(t)/dt = f(x(t - d)) = inv(1 + D.*dfdx)*f(x(t))
%                     = Q*f = Q*J*x(t)
%--------------------------------------------------------------------------
Q  = spm_inv(speye(length(J)) + D.*J);

end

function [g_theta_g]    = erp_g(m_g)

% This function evaluates the EEG forward model of the delay differential
% equation model for ERPs based on a structural formulation of the forward
% model and parameter setting
%
%   Inputs
%           m_g          : EEG forward model structure and fixed parameters
%           theta      : EEG lead-field free parameters

%    Outputs
%           g_theta_g   : evaluated EEG forward model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.Lpos  = m_g.Lpos;
theta.L     = m_g.L;
theta.J     = m_g.J;

% evaluate constants
nd          = size(theta.L,2)                ; % number of dipoles
ne          = size(m_g.sens.chanpos,1)       ; % number of electrodes
dp          = 1e-3*theta.Lpos                ; % dipole coordinates adjusted for fieldtrip

% compute canonical dipole lead-field using Fieldtrip
L_can  = NaN(ne,3,nd);
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end

% rescale lead-field according to SPM
L_can = L_can*(10^1)^-round(log10(max(max(abs(L_can))))/8)/1);

% evaluate channel predictions based on dipole moments
L_dip = NaN(ne,nd);
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*theta.L(:,i);
end

% evaluate g(\theta_g)
g_theta_g = [];
for i = 1:nd
    g_theta_g = [g_theta_g kron(theta.J,L_dip(:,i))];
end
end

function [vFE]          = vFE(m_theta,s_theta,ms_sigsqr,vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : p x 1 variational expectation for theta
```

```
%       s_theta    : p x p variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters   for sigma^2
%       vfearg     : additional argument structure with fields
%           .y              : n x 1 array   - data
%           .h              : string        - function handle
%           .mu_theta       : scalar prior expectation for theta
%           .sig_theta      : scalar prior variance for theta
%           .mu_sigsqr      : scalar prior scale parameter for sigma^2
%           .si_sigsqr      : scalar prior shape parameter for sigma^2
%           .theta          : true, but unknown, parameter structure
%   Outputs
%           vFE             : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
m_f             = vfearg.m_f;
m_g             = vfearg.m_g;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;
theta           = vfearg.theta;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta) and J^h(m_theta)
h_m_theta       = erp_h(m_f,m_g,spm_unvec(m_theta, theta));
Jh_m_theta      = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g,spm_unvec(m_theta, theta)),m_theta,1)));


% evaluation of the variational free energy value
T1              = -(n/2)*log(2*pi)          - (n/2)*m_sigsqr            - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2              = -(p/2)*log(2*pi)          - (1/2)*log(det(sig_theta))  - (1/2)*((m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
T3              = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                 - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T4              =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
T5              =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                                  ;

% evaluate the negative free energy
vFE             = -(T1 + T2 + T3 + T4 + T5);


end

function [t_k]          = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsqr, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k             : p x 1     array of function argument
%       s_theta         : p x p     array of additional vFE arguments
%       ms_sigsqr       : 2 x 1     array of additional vFE arguments
%       vfearg          : additional input for vFE
%       df_x_k          :p_f x 1 array of function gradient at x_k
%       p_k             : search direction
%       c               : scalar constant in ]0,1[
%       rho             : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(x_k, s_theta, ms_sigsqr, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(x_k + t_k*p_k, s_theta, ms_sigsqr, vfearg);

% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(x_k+t_k*p_k, s_theta, ms_sigsqr, vfearg);
end
end

function [t_k]          = backtrack_vFE_ms_sigsqr(m_theta, s_theta, x_k, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       m_theta     : p x 1   array of additional vFE arguments
%       s_theta     : p x p array of additional vFE arguments
%       x_k         : 2 x 1     array of function arguments
%       vfearg      : additional input for vFE
%       df_x_k      : 2 x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(m_theta, s_theta, x_k, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(m_theta, s_theta, x_k + t_k*p_k, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)|| x_k(2) + t_k*p_k(2) < 0;
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(m_theta, s_theta, x_k+t_k*p_k, vfearg);
end
```

end

PDDE_5.m – ERP-DCM dipole moment estimation (Figure 9)

```
function pdde_5_R1

% This function implements the estimation of dipole moments in a basic
% delay differential equation ERP model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
clc
close all

% set the random number generator for reproducible sampling results
reset(RandStream.getGlobalStream)

% invoke Fieldtrip
addpath(genpath(fullfile(pwd, 'Fieldtrip')))

% -------------------------------------------------------------------------
%                           Model Formulation
% -------------------------------------------------------------------------

% neural model structure and parameters
% -------------------------------------------------------------------------
% structural aspects and fixed parameters
m_f.f               = @spm_fx_erp;                                  % neural evolution function
m_f.ns              = 200;                                          % number of time bins
m_f.dt              = 0.001;                                        % time bin length (sec)
m_f.pst             = (0:m_f.ns-1)*m_f.dt;                          % peristimulus time (msec)
m_f.nt              = length(m_f.pst);                              % number of time points
m_f.n               = 2;                                            % number of neural masses
m_f.m               = 9;                                            % number of neural states per neural mass
m_f.ons             = -10;                                          % system input onset (ms)
m_f.dur             = 1;                                            % duration (width) of the input function
m_f.x               = sparse(m_f.n,m_f.m);                          % initial condition
m_f.R               = [ 0.16 -0.22];                                % input function parameters rho_1 and rho_2
m_f.S               = [-9e-4  0.08];                                % static nonlinearity (activation function) parameters
m_f.T               = [ 0.16  0.13;  0.07 -0.82];                   % source-specific excitatory and inhibitory time constants
m_f.G               = [-0.11  0.45;  0.02 -0.63];                   % source-specific excitatory and inhibitory receotor densities
m_f.H               = [ 0.02 -0.07   0.18 -0.17];                   % source-independent intrinsic connectivity parameters
m_f.D               = [ 0    -0.52; -0.06  0   ];                   % between source delay parameters
m_f.A{1}            = [0 0; 1 0];                                   % forward  connectivity
m_f.A{2}            = zeros(2,2);                                   % backward connectivity
m_f.A{3}            = zeros(2,2);                                   % lateral connectivity
m_f.C               = [1 0]';                                       % input connectivity

% forward model structure and parameters
% -------------------------------------------------------------------------
% load an existing dipole model
load(fullfile(cd, 'Data', 'dip_model_struct.mat'))
m_g.vol             = dip_model_struct.vol;                         % forward model specification
m_g.sens            = dip_model_struct.sens ;                       % sensor space specification
m_g.elab            = dip_model_struct.sens.label;                  % electrode labels
m_g.ne              = numel(m_g.elab);                              % number of electrodes
m_g.X0              = zeros(m_f.ns*m_g.ne, 3*m_g.ne);               % confound design matrix
m_g.Lpos            = [42 -31 58; 54 -22 18]';                      % dipole coordinate specifications
m_g.L               = [NaN NaN NaN; 0.17 -0.50 -0.17]';             % dipole moment specification
m_g.J               = [-0.1 0 0 0 0 0 -0.46 0 1];                   % state weighting specification

% likelihood model parameters
% -------------------------------------------------------------------------
theta               = [0.02 -0.05 -0.04]';                          % dipole moments S1
p                   = length(theta);                                % number of parameters
sigsqr              = 1e-1;                                         % true, but unknown, data variance
n                   = m_f.nt*m_g.ne;                                % number of data points

% prior formulation
% -------------------------------------------------------------------------
mu_theta            = zeros(p,1);                                   % prior expectation parameter setting theta
sig_theta           = [1e3 0 0; 0 1e3 0; 0 0 1e3];                  % prior variancen parameter setting theta
mu_si_sigsqr        = [4.6;1];                                      % prior shape and scale parameter setting \sigma^2

% fixed form variational Bayes Newton algorithm parameters
% -------------------------------------------------------------------------
max_i               = 5;                                            % maximal number of fixed-form VB Newton algorithm
max_j               = 8;                                            % maximal number of Newton iterations for m_theta optimization
max_k               = 8;                                            % maximal number of Newton iterations for (m_sigsqr, s_sigssqr) optimization
delta               = sqrt(eps);                                    % Hessian modification constant
c                   = 1e-4;                                         % backtracking constant c
rho                 = 0.9;                                          % backtracking constant rho
vareps              = 1e0;                                          % gradient norm convergence criterion
vardelta            = 1e-3;                                         % parameter norm change convergence criterion

% -------------------------------------------------------------------------
%                           Model Realization
% -------------------------------------------------------------------------
h_theta_fg          = erp_h(m_f,m_g,theta);                         % data expectation
y                   = mvnrnd(h_theta_fg, sigsqr*eye(n))';           % data sampling

% -------------------------------------------------------------------------
%                           Data Visualization
% -------------------------------------------------------------------------
% initialize figure
fig = figure;
set(fig, 'Color', [1 1 1])

% evaluate latent data
[f_theta_f,u]       = erp_f(m_f);
f_theta_f_s1        = f_theta_f(:,1:2:end-1);
f_theta_f_s2        = f_theta_f(:,2:2:end);

% Input function u
% -------------------------------------------------------------------------
subplot(2,3,1)
plot(m_f.pst,u)
xlim([m_f.pst(1) m_f.pst(end)])
title('u', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% latent variable evolution
% -------------------------------------------------------------------------
subplot(2,3,2)
plot(m_f.pst,f_theta_f_s1)
xlim([m_f.pst(1) m_f.pst(end)])
title('x^{(1)}', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-10 40])

subplot(2,3,5)
plot(m_f.pst,f_theta_f_s2)
xlim([m_f.pst(1) m_f.pst(end)])
title('x^{(2)}', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-1 5])

% observed variables evolution
% -------------------------------------------------------------------------
subplot(2,3,3)
imagesc(m_f.pst, 1:numel(m_g.elab),spm_unvec(y, NaN(m_g.ne,m_f.nt)))
xlim([m_f.pst(1) m_f.pst(end)])
cb = colorbar;
title('Y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel(cb, '\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('Electrodes', 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% selected electrodes
subplot(2,3,6)
eoi         = [47 49:51];
eoilab      = m_g.elab(eoi);
data        = spm_unvec(y, NaN(m_g.ne,m_f.nt));
plot(m_f.pst,data(eoi,:))
xlim([m_f.pst(1) m_f.pst(end)])
title('Y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
legend(eoilab)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% -------------------------------------------------------------------------
%                           VBNL iteration 0
% -------------------------------------------------------------------------
% initialization of the variational parameters to prior parameters
m_theta             = mu_theta;
s_theta             = sig_theta;
ms_sigsqr           = mu_si_sigsqr;
vfearg.y            = y;
vfearg.m_f          = m_f;
vfearg.m_g          = m_g;
vfearg.mu_theta     = mu_theta;
vfearg.sig_theta    = sig_theta;
vfearg.mu_sigsqr    = mu_si_sigsqr(1);
vfearg.si_sigsqr    = mu_si_sigsqr(2);
vfearg.theta        = theta;

% parameter iteration arrays
s_theta_i           = NaN(p,p,max_i-1);
F_s_theta_i         = NaN(1,max_i-1);
m_theta_j           = NaN(p,max_j,max_i-1);
F_m_theta_j         = NaN(max_j,max_i-1);
ms_sigsqr_k         = NaN(2,max_k,max_i-1);
F_ms_sigsqr_k       = NaN(max_k,max_i-1);

% initialization and initial evaluation of the negative free energy
F                   = NaN(max_i-1);
F(1)                = vFE(m_theta, s_theta, ms_sigsqr, vfearg);


% VBNL iterations 1,2,3,...
```

```
% ------------------------------------------------------------------------
for i = 2:max_i

    % User update
    % ------------------------------------------------------------------------
    fprintf('Fixed-form VB Newton Iteration %4.0f ... \n', i-1)

    % Analytical update for theta_f variational variance parameter
    % ------------------------------------------------------------------------
    % evaluate the variational variance update equation for the current choice of m_theta
    fprintf('   Estimating q(m_theta) variance  ... ')
    Jh_m_theta          = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g, spm_unvec(m_theta, theta)),m_theta,1)));
    s_theta             = spm_inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (spm_inv(sig_theta))));

    % save iterand and objective function
    s_theta_i(:,:,i-1)  = s_theta;
    F_s_theta_i(i-1)    =  vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % inform user
    fprintf('finished. \n')

    % Newton line search for theta_f variational expectation parameter
    % ------------------------------------------------------------------------
    fprintf('   Estimating q(m_theta) expectation ... ')

    % save iterand and objective function
    m_theta_j(:,1,i-1)  = m_theta;
    F_m_theta_j(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % numerical gradient and Hessian of the negative free energy
    dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
    ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_m_theta) >= vareps
        for j = 2:max_j

            % gradient search direction
            p_j = -ddF_m_theta\dF_m_theta;

            % Hessian modification if p_j is not a descent direction
            if ~(p_j'*dF_m_theta < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_m_theta = ddF_m_theta + max(0,(delta - min(eig(ddF_m_theta)))*eye(p);

                % re-evaluate Newton search direction
                p_j = -ddF_m_theta\dF_m_theta;
            end

            % backtracking evaluation of the step length
            t_j             = backtrack_vFE_m_theta(m_theta, s_theta, ms_sigsqr, vfearg, dF_m_theta, p_j,c,rho);

            % perform parameter update
            m_theta         = m_theta + t_j*p_j;

            % numerical gradient and Hessian of the negative free energy
            dF_m_theta      = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
            ddF_m_theta     = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

            % record updates
            m_theta_j(:,j,i-1)    = m_theta;
            F_m_theta_j(j,i-1)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_m_theta) < vareps || norm(m_theta_j(:,j,i-1) - m_theta_j(:,j-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished.\n')

    % Newton line search for sigma^2 variational parameters
    % ------------------------------------------------------------------------
    fprintf('   Estimating q(sigma^2) parameters  ... ')

    % save iterand and objective function
    ms_sigsqr_k(:,1,i-1)  = ms_sigsqr;
    F_ms_sigsqr_k(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
    dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
    ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_ms_sigsqr) > vareps
        for k = 2:max_k

            % Newton search direction
            p_k = -ddF_ms_sigsqr\dF_ms_sigsqr;

            % Hessian modification if p_j is not a descent direction
            if ~(p_k'*dF_ms_sigsqr < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_ms_sigsqr = ddF_ms_sigsqr + max(0,(delta - min(eig(ddF_ms_sigsqr)))*eye(2);

                % re-evaluate Newton search direction
                p_k  = -ddF_ms_sigsqr\dF_ms_sigsqr;
            end

            % backtracking evaluation of the step length
            t_k = backtrack_vFE_ms_sigsqr(m_theta, s_theta, ms_sigsqr, vfearg, dF_ms_sigsqr, p_k,c,rho);

            % perform Newton update
            ms_sigsqr           = ms_sigsqr + t_k*p_k;

            % update remaining entries in free energy arrays
            dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
            ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

            % record updates
            ms_sigsqr_k(:,k,i-1)  = ms_sigsqr;
            F_ms_sigsqr_k(k,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_ms_sigsqr) < vareps || norm(ms_sigsqr_k(:,k,i-1) - ms_sigsqr_k(:,k-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished. \n')

    % free energy update
    % ------------------------------------------------------------------------
    F(i) = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % user update
    % ------------------------------------------------------------------------
    fprintf('vFE %6.3f  dvFE %6.3f \n', F(i), F(i) - F(i-1))
end

% ------------------------------------------------------------------------
%                       Algorithm Visualization
% ------------------------------------------------------------------------

% initialize figure
h = figure;
set(h, 'Color', [1 1 1])

% m_theta prior and approximate posterior
% ------------------------------------------------------------------------
subplot(2,3,1)
hold on
plot3(mu_theta(1),mu_theta(2),mu_theta(3), 'ko', 'MarkerFaceColor', 'k')
lambda = eig(sig_theta);
[x,y,z] = ellipsoid(mu_theta(1),mu_theta(2),mu_theta(3),2*sqrt(5.991*lambda(1)),2*sqrt(5.991*lambda(2)),2*sqrt(5.991*lambda(3)));
hsurf = surf(x,y,z, 'EdgeColor', 'None','FaceLighting','gouraud');
camlight headlight
set(hsurf,'FaceColor',[0 0 1],'FaceAlpha',0.3)
grid on
view([45 20])
axis square
xlabel('\theta_{m_{1}}^{1}', 'FontName', 'Calibri Light', 'FontSize', 20)
ylabel('\theta_{m_{2}}^{1}', 'FontName', 'Calibri Light', 'FontSize', 20)
zlabel('\theta_{m_{3}}^{1}', 'FontName', 'Calibri Light', 'FontSize', 20, 'Rotation', 0)
title(`p(\theta_m^{1})`, 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
set(gca, 'FontSize', 19, 'FontName', 'Calibri Light')

subplot(2,3,2)
hold on
plot3(m_theta(1),m_theta(2),m_theta(3), 'ko', 'MarkerFaceColor', 'k')
plot3(theta(1),theta(2),theta(3), 'k+', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
lambda = eig(s_theta);
[x,y,z] = ellipsoid(m_theta(1),m_theta(2),m_theta(3),2*sqrt(5.991*lambda(1)),2*sqrt(5.991*lambda(2)),2*sqrt(5.991*lambda(3)));
hsurf = surf(x,y,z, 'EdgeColor', 'None','FaceLighting','gouraud');
xlim([0.019  0.021])
ylim([-0.051 -0.0495])
zlim([-0.05 -0.03])
camlight headlight
set(hsurf,'FaceColor',[0 0 1],'FaceAlpha',0.3)
grid on
view([45 20])
axis square
set(gca, 'FontSize', 19, 'FontName', 'Calibri Light')
xlabel('\theta_{m_{1}}^{1}', 'FontName', 'Calibri Light', 'FontSize', 20)
ylabel('\theta_{m_{2}}^{1}', 'FontName', 'Calibri Light', 'FontSize', 20)
title(`q(\theta_m^{1})`, 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')

% sigma^2 prior and posterior
% ------------------------------------------------------------------------
sigsqr_pri      = linspace(1e-3,6e2,1e2);
p_sigsqr_pri    = lognpdf(sigsqr_pri, mu_si_sigsqr(1) , sqrt(ms_si_sigsqr(2)));
sigsqr_pos      = linspace(1e-6,0.2,1e2);
p_sigsqr_pos    = lognpdf(sigsqr_pos, ms_sigsqr(1) , sqrt(ms_sigsqr(2)));

subplot(2,3,3)
hold on
line(sigsqr_pri,p_sigsqr_pri, 'Color','b')
ax1             = gca; % current axes
ax1.XColor      = 'b';
```

```
ax1.YColor      = 'b';
ax1.FontName    = 'Calibri Light';
ax1.FontSize    = 20;
ax1_pos         = ax1.Position;
xlabel('\sigma^2', 'FontName', 'Calibri Light', 'FontSize', 24, 'Color', 'k')
ax2             = axes('Position',ax1_pos,'XAxisLocation','top','YAxisLocation','right','Color','None', 'FontName', 'Calibri Light', 'FontSize', 20);
ax2.XColor      = 'r';
ax2.YColor      = 'r';
line(sigsqr_pos,p_sigsqr_pos,'Parent',ax2, 'Color','r')

% m_theta variational free energy ascent on first VBNL iteration
% -------------------------------------------------------------------------
% Newton iteration 0
res             = 3e1;
m_theta_1_min   = -.02;
m_theta_1_max   =  .04;
m_theta_1_res   = res;
m_theta_1_space   = linspace(m_theta_1_min,m_theta_1_max,m_theta_1_res);
m_theta_2_min   = -.1;
m_theta_2_max   = .05;
m_theta_2_res   = res;
m_theta_2_space   = linspace(m_theta_2_min,m_theta_2_max,m_theta_2_res);
vFE_m_theta     = NaN(res,res);
for i = 1:res
    for j = 1:res
        vFE_m_theta(i,j) = vFE([m_theta_1_space(i) m_theta_2_space(j) m_theta_j(3,1,1)]', s_theta_i(:,:,1), mu_si_sigsqr, vfearg);
    end
end
subplot(2,3,4)
hold on
surf(m_theta_2_space, m_theta_1_space, -vFE_m_theta, 'EdgeColor', 'None')
plot3(m_theta_j(2,1,1), m_theta_j(1,1,1),  -F_m_theta_j(1,1), 'ko-', 'MarkerFaceColor' ,'k')
axis tight
alpha(.6)
view([-30 60])
xlabel('m_{\theta_2}', 'FontSize', 20, 'FontName', 'Calibri Light')
ylabel('m_{\theta_1}', 'FontSize', 20, 'FontName', 'Calibri Light')
title('m_\theta', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% Newton iteration 1
res             = 3e1;
m_theta_1_min   = -.02;
m_theta_1_max   =  .06;
m_theta_1_res   = res;
m_theta_1_space   = linspace(m_theta_1_min,m_theta_1_max,m_theta_1_res);
m_theta_2_min   = -.15;
m_theta_2_max   = .05;
m_theta_2_res   = res;
m_theta_2_space   = linspace(m_theta_2_min,m_theta_2_max,m_theta_2_res);
vFE_m_theta     = NaN(res,res);
for i = 1:res
    for j = 1:res
        vFE_m_theta(i,j) = vFE([m_theta_1_space(i) m_theta_2_space(j) m_theta_j(3,2,1)]', s_theta_i(:,:,1), mu_si_sigsqr, vfearg);
    end
end
subplot(2,3,5)
hold on
surf(m_theta_2_space, m_theta_1_space, -vFE_m_theta, 'EdgeColor', 'None')
plot3(m_theta_j(2,2,1), m_theta_j(1,2,1),  -F_m_theta_j(2,1), 'ko-', 'MarkerFaceColor' ,'k')
axis tight
alpha(.6)
view([-30 60])
xlabel('m_{\theta_2}', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('m_{\theta_1}', 'FontSize', 24, 'FontName', 'Calibri Light')
title('m_\theta', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% variational free energy
% -------------------------------------------------------------------------
subplot(2,3,6)
hold on
plot(-F(2:end), 'ko-', 'MarkerFaceColor', 'k')
xlim([0 max_i])
title('F(m_\theta^{(i)}, \Sigma_\theta^{(i)}, m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)})', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 20)
xlabel('i', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% remove Fieldtrip
% -------------------------------------------------------------------------
rmpath(genpath(fullfile(pwd, 'Fieldtrip')))

end

% -------------------------------------------------------------------------
%                           Subfunctions
% -------------------------------------------------------------------------
function [h_theta_fg]   = erp_h(m_f,m_g,theta)

% This function evaluates the expectation parameter generating function
%
% h : (theta_f,theta_g) |-> h(theta_f,theta_g) := vec(g(theta_g)*f(theta_f)
%
% of the delay differential equation model for ERPs
%
%       Inputs
%           m_f         : latent neural model structure
%           m_g         : forward model structure
%           theta       : model parameters
%
%       Outputs
%           h_theta_fg  : evaluated function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluate component functions f and g
[f_theta_f, u]  = erp_f(m_f);
g_theta_g       = erp_g(m_g,theta);

% evaluate concatenated function h
h_theta_fg      = 1e4*spm_vec(g_theta_g*f_theta_f');

end

function [f_theta_f,u]  = erp_f(m_f)

% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_gen_erp.m of
% the SPM12 distribution. Based on a given parameter setting this function
% generates an input time-course and  integrates the resulting delay
% differential equation system.
%
%       Inputs
%           m_f         : latent neural model structure and fixed parameters
%           theta       : free latent neural model parameter
%
%       Outputs
%           f_theta_f   : evaluated latent neural model
%           u           : evaluated system input function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.R             = m_f.R;
theta.S             = m_f.S;
theta.T             = m_f.T;
theta.G             = m_f.G;
theta.H             = m_f.H;
theta.D             = m_f.D;
theta.A{1}          = m_f.A{1};
theta.A{2}          = m_f.A{2};
theta.A{3}          = m_f.A{3};
theta.C             = m_f.C;

% evaluate peri-stimulus time input function
t                   = ((1:m_f.ns)*m_f.dt)*1000;
delay               = m_f.ons(1)+128*theta.R(1,1);
scale               = m_f.dur(1)*exp(theta.R(1,2));
u                   = 32*exp(-(t - delay).^2/(2*scale^2))';

% integrate systen
x                   = m_f.x                                             ; % initial condition
f                   = m_f.f                                             ; % function handle @spm_fx_erp
dt                  = m_f.dt                                            ; % integration time bin
[fx,dfdx,b]         = f(x,u(1),theta,m_f)                               ; % dx(t)/dt and Jacobian df/dx and check for delay operator
p                   = max(abs(real(eig(full(dfdx)))))                   ; %
N                   = ceil(max(1, dt*p*2))                              ; %
n                   = spm_length(x)                                     ; %
Q                   = (spm_expm(dt*D*dfdx/N) - speye(n,n))*spm_inv(dfdx)    ; %
v                   = spm_vec(x)                                        ; % initialize state
y                   = NaN(length(v), length(u))                         ; % initialize state time-course

% cycle over time-steps
for i = 1:size(u,1)

    % update dx = (expm(dt*J) - I)*inv(J)*f(x,u)
    for j = 1:N
        v = v + Q*f(v,u(i),theta,m_f);
    end
    y(:,i) = v;

end

% transpose
f_theta_f       = y';

end

function [f,J,Q]        = spm_fx_erp(x,u,P,M)

% state equations for a neural mass model of erps
% FORMAT [f,J,D] = spm_fx_erp(x,u,P,M)
% FORMAT [f,J]   = spm_fx_erp(x,u,P,M)
% FORMAT [f]     = spm_fx_erp(x,u,P,M)
% x     - state vector
%   x(:,1) - voltage (spiny stellate cells)
%   x(:,2) - voltage (pyramidal cells) +ve
%   x(:,3) - voltage (pyramidal cells) -ve
%   x(:,4) - current (spiny stellate cells)    depolarizing
%   x(:,5) - current (pyramidal cells)         depolarizing
%   x(:,6) - current (pyramidal cells)         hyperpolarizing
%   x(:,7) - voltage (inhibitory interneurons)
%   x(:,8) - current (inhibitory interneurons) depolarizing
```

```
%   x(:,9) - voltage (pyramidal cells)
%
% f         - dx(t)/dt  = f(x(t))
% J         - df(t)/dx(t)
% D         - delay operator dx(t)/dt = f(x(t - d))
%                                     = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
%  M.pF.E = [32 16 4];           % extrinsic rates (forward, backward, lateral)
%  M.pF.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2 g3, g4)
%  M.pF.D = [2 16];              % propogation delays (intrinsic, extrinsic)
%  M.pF.G = [4 32];              % receptor densities (excitatory, inhibitory)
%  M.pF.T = [8 16];              % synaptic constants (excitatory, inhibitory)
%  M.pF.R = [1 1/2];             % parameter of static nonlinearity
%
%__________________________________________________________________________
% David O, Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neuronal dynamics. NeuroImage 20: 1743-1755
%__________________________________________________________________________
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging

% Karl Friston
% $Id: spm_fx_erp.m 5369 2013-03-28 20:09:27Z karl $

% get dimensions and configure state variables
%--------------------------------------------------------------------------
n       = length(P.A{1});           % number of sources
x       = spm_unvec(x,M.x);         % neuronal states

% [default] fixed parameters
%--------------------------------------------------------------------------
E       = [1 1/2 1/8]*32;                % extrinsic rates (forward, backward, lateral)
G       = [1 4/5 1/4 1/4]*128;           % intrinsic rates (g1 g2 g3 g4)
D       = [2 16];                        % propogation delays (intrinsic, extrinsic)
H       = [4 32];                        % receptor densities (excitatory, inhibitory)
T       = [8 16];                        % synaptic constants (excitatory, inhibitory)
R       = [2 1]/3;                       % parameters of static nonlinearity

% test for free parameters on intrinsic connections
%--------------------------------------------------------------------------
G       = G.*exp(P.H);
G       = ones(n,1)*G;

% no exponential transforms to foster parameter identifiability
% -------------------------------------------------------------------------
A{1}    = P.A{1}*E(1);
A{2}    = P.A{2}*E(2);
A{3}    = P.A{3}*E(3);
C       = P.C;

% intrinsic connectivity and parameters
%--------------------------------------------------------------------------
Te      = T(1)/1000*exp(P.T(:,1));       % excitatory time constants
Ti      = T(2)/1000*exp(P.T(:,2));       % inhibitory time constants
He      = H(1)*exp(P.G(:,1));            % excitatory receptor density
Hi      = H(2)*exp(P.G(:,2));            % inhibitory receptor density

% pre-synaptic inputs: s(V)
%--------------------------------------------------------------------------
S       = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
%--------------------------------------------------------------------------
U       = C*u(:)*2;

% State: f(x)
%==========================================================================
f(:,1) = x(:,4)                                                 ; % voltage change (spiny stellate cells)
f(:,2) = x(:,5)                                                 ; % positive voltage change (pyramidal cells) +ve
f(:,3) = x(:,6)                                                 ; % negative voltage change (pyramidal cells) -ve
f(:,4) = (He.*((A{1} + A{3})*S(:,9) + G(:,1).*S(:,9) + U) - 2*x(:,4) - x(:,1)./Te)./Te  ; % current change (spiny stellate cells)     depolarizing
f(:,5) = (He.*((A{2} + A{3})*S(:,9) + G(:,2).*S(:,1)) - 2*x(:,5) - x(:,2)./Te)./Te      ; % current change (pyramidal cells)          depolarizing
f(:,6) = (Hi.*G(:,4).*S(:,7) - 2*x(:,6) - x(:,3)./Ti)./Ti                                ; % current change (pyramidal cells)          hyperpolarizing
f(:,7) = x(:,8)                                                 ; % voltage change (inhibitory interneurons)
f(:,8) = (He.*((A{2} + A{3})*S(:,9) + G(:,3).*S(:,9)) - 2*x(:,8) - x(:,7)./Te)./Te      ; % current change(inhibitory interneurons) depolarizing
f(:,9) = x(:,5) - x(:,6)                                        ; % voltage (pyramidal cells)


% vectorize
f       = spm_vec(f);

% avoid infinite recursion of spm_diff
if nargout < 2
    return
end

% Jacobian
%==========================================================================
J  = spm_diff(M.f,x,u,P,M,1);

% delays
%==========================================================================
% Delay differential equations can be integrated efficiently (but
% approximately) by absorbing the delay operator into the Jacobian
%
%    dx(t)/dt     = f(x(t - d))
%                 = Q(d)f(x(t))
%
%    J(d)         = Q(d)df/dx
%--------------------------------------------------------------------------
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,n)).*De;
Di = (1 - speye(9,9)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D  = Di + De;

% Implement: dx(t)/dt = f(x(t - d)) = inv(1 + D.*dfdx)*f(x(t))
%                     = Q*f = Q*J*x(t)
%--------------------------------------------------------------------------
Q  = spm_inv(speye(length(J)) + D.*J);

end

function [g_theta_g]    = erp_g(m_g, vartheta)

% This function evaluates the EEG forward model of the delay differential
% equation model for ERPs based on a structural formulation of the forward
% model and parameter setting
%
%   Inputs
%           m_g          : EEG forward model structure and fixed parameters
%           vartheta     : EEG lead-field free parameters

%    Outputs
%           g_theta_g    : evaluated EEG forward model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.Lpos  = m_g.Lpos;
theta.L     = m_g.L;
theta.J     = m_g.J;

% free parameters
theta.L(1,1) = vartheta(1);
theta.L(2,1) = vartheta(2);
theta.L(3,1) = vartheta(3);


% evaluate constants
nd          = size(theta.L,2)                                       ; % number of dipoles
ne          = size(m_g.sens.chanpos,1)                              ; % number of electrodes
dp          = 1e-3*theta.Lpos                                       ; % dipole coordinates adjusted for fieldtrip

% compute canonical dipole lead-field using Fieldtrip
L_can  = NaN(ne,3,nd);
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end

% rescale lead-field according to SPM
L_can = L_can*(10^1)^-round(log10(max(max(max(abs(L_can))))/8)/1);

% evaluate channel predictions based on dipole moments
L_dip = NaN(ne,nd);
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*theta.L(:,i);
end

% evaluate g(\theta_g)
g_theta_g     = kron(theta.J,L_dip);


end

function [vFE]          = vFE(m_theta,s_theta,ms_sigsqr,vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : p x 1 variational expectation for theta
%       s_theta    : p x p variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters  for sigma^2
%       vfearg     : additional argument structure with fields
%           .y              : n x 1 array   - data
%           .h              : string        - function handle
%           .mu_theta       : scalar prior expectation for theta
%           .sig_theta      : scalar prior variance for theta
%           .mu_sigsqr      : scalar prior scale parameter for sigma^2
%           .si_sigsqr      : scalar prior shape parameter for sigma^2
%           .theta          : true, but unknown, parameter structure
%   Outputs
%           vFE             : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
```

```
y               = vfearg.y;
m_f             = vfearg.m_f;
m_g             = vfearg.m_g;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;
theta           = vfearg.theta;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta) and J^h(m_theta)
h_m_theta       = erp_h(m_f,m_g, spm_unvec(m_theta, theta));
Jh_m_theta      = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g,spm_unvec(m_theta, theta)),m_theta,1)));


% evaluation of the variational free energy value
T1              = -(n/2)*log(2*pi)          - (n/2)*m_sigsqr            - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2              = -(p/2)*log(2*pi)          - (1/2)*log(det(sig_theta))  - (1/2)*((m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
T3              = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                 - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T4              =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
T5              =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                                                ;

% evaluate the negative free energy
vFE             = -(T1 + T2 + T3 + T4 + T5);


end

function [vFE]          = vFE_terms(m_theta,s_theta, ms_sigsqr, vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : 2 x 1 variational expectation for theta
%       s_theta    : 2 x 2 variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters   for sigma^2
%       vfearg     : additional argument structure with fields
%           .y              : n x 1 array   - data
%           .h              : string        - function handle
%           .mu_theta       : scalar prior expectation for theta
%           .sig_theta      : scalar prior variance for theta
%           .mu_sigsqr      : scalar prior scale parameter for sigma^2
%           .si_sigsqr      : scalar prior shape parameter for sigma^2
%   Outputs
%           vFE             : structure with fields
%               .T1A            : scalar accuracy term A
%               .T1B            : scalar accuracy term B
%               .T2             : scalar prior influence m_theta
%               .T3             : scalar prior influence ms_sigsqr
%               .T4             : scalar entropy q(theta)
%               .T5             : scalar entropy q(sigma^2)
%               .vFE            : scalar variational free energy
%
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
m_f             = vfearg.m_f;
m_g             = vfearg.m_g;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;
theta           = vfearg.theta;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta) and J^h(m_theta)
h_m_theta       = erp_h(m_f,m_g,spm_unvec(m_theta, theta));
Jh_m_theta      = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g,spm_unvec(m_theta, theta)),m_theta,1)));

% evaluation of the variational free energy value
vFE.T1          = -(n/2)*log(2*pi)  - (n/2)*m_sigsqr   - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta));
vFE.T2          = -(1/2)*exp(-m_sigsqr +.5*s_sigsqr)*(trace((Jh_m_theta'*Jh_m_theta)*s_theta));
vFE.T3          = -(p/2)*log(2*pi)          - (1/2)*log(det(sig_theta))  - (1/2)*((m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
vFE.T4          = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                 - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
vFE.T5          =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
vFE.T6          =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                                                ;

% evaluate the positive free energy
vFE.vFE         = vFE.T1 + vFE.T2 + vFE.T3 + vFE.T4 + vFE.T5 + vFE.T6;

end

function [t_k]          = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsqr, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k         : p x 1     array of function argument
%       s_theta     : p x p     array of additional vFE arguments
%       ms_sigsqr   : 2 x 1     array of additional vFE arguments
%       vfearg      : additional input for vFE
%       df_x_k      :p_f x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(x_k, s_theta, ms_sigsqr, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(x_k + t_k*p_k, s_theta, ms_sigsqr, vfearg);

% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(x_k+t_k*p_k, s_theta, ms_sigsqr, vfearg);
end
end

function [t_k]          = backtrack_vFE_ms_sigsqr(m_theta, s_theta, x_k, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       m_theta     : p x 1   array of additional vFE arguments
%       s_theta     : p x p array of additional vFE arguments
%       x_k         : 2 x 1     array of function arguments
%       vfearg      : additional input for vFE
%       df_x_k      : 2 x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(m_theta, s_theta, x_k, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(m_theta, s_theta, x_k + t_k*p_k, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)|| x_k(2) + t_k*p_k(2) < 0;
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(m_theta, s_theta, x_k+t_k*p_k, vfearg);
end
end
```

PDDE_6.m – ERP-DCM forward connectivity and dipole moment estimation (Figure 10)

```
function pdde_6_R1

% This function implements the estimation of forward connectivity and
% dipole moments in a basic delay differential equation ERP model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
clc
close all

% turn of matrix inversion warning - these cases result in extremely small
% step sizes, which in turn result in algorithm convergence.
warning off

% set the random number generator for reproducible sampling results
reset(RandStream.getGlobalStream)

% invoke Fieldtrip
addpath(genpath(fullfile(pwd, 'Fieldtrip')))

% -------------------------------------------------------------------------
%                           Model Formulation
% -------------------------------------------------------------------------

% neural model structure and parameters
% -------------------------------------------------------------------------
% structural aspects and fixed parameters
m_f.f                   = @spm_fx_erp;                                  % neural evolution function
m_f.ns                  = 200;                                          % number of time bins
m_f.dt                  = 0.001;                                        % time bin length (sec)
m_f.pst                 = (0:m_f.ns-1)*m_f.dt;                          % peristimulus time (msec)
m_f.nt                  = length(m_f.pst);                              % number of time points
m_f.n                   = 2;                                            % number of neural masses
m_f.m                   = 9;                                            % number of neural states per neural mass
m_f.ons                 = -10;                                          % system input onset (ms)
m_f.dur                 = 1;                                            % duration (width) of the input function
m_f.x                   = sparse(m_f.n,m_f.m);                          % initial condition
m_f.R                   = [ 0.16 -0.22];                                % input function parameters rho_1 and rho_2
m_f.S                   = [-9e-4  0.08];                                % static nonlinearity (activation function) parameters
m_f.T                   = [ 0.16  0.13;  0.07 -0.82];                   % source-specific excitatory and inhibitory time constants
m_f.G                   = [-0.11  0.45;  0.02 -0.63];                   % source-specific excitatory and inhibitory receotor densities
m_f.H                   = [ 0.02 -0.07   0.18 -0.17];                   % source-independent intrinsic connectivity parameters
m_f.D                   = [ 0    -0.52; -0.06  0   ];                   % between source delay parameters
m_f.A{1}                = [0 0; NaN 0];                                 % forward  connectivity
m_f.A{2}                = zeros(2,2);                                   % backward connectivity
m_f.A{3}                = zeros(2,2);                                   % lateral connectivity
m_f.C                   = [1 0]';                                       % input connectivity

% forward model structure and parameters
% -------------------------------------------------------------------------
% load an existing dipole model
load(fullfile(cd, 'Data', 'dip_model_struct.mat'))
m_g.vol                 = dip_model_struct.vol;                         % forward model specification
m_g.sens                = dip_model_struct.sens ;                       % sensor space specification
m_g.elab                = dip_model_struct.sens.label;                  % electrode labels
m_g.ne                  = numel(m_g.elab);                              % number of electrodes
m_g.X0                  = zeros(m_f.ns*m_g.ne, 3*m_g.ne);               % confound design matrix
m_g.Lpos                = [42 -31 58; 54 -22 18]';                      % dipole coordinate specifications
m_g.L                   = [NaN NaN NaN; 0.17 -0.50 -0.17]';             % dipole moment specification
m_g.J                   = [-0.1 0 0 0 0 0 0 -0.46 0 1];                 % state weighting specification

% likelihood model parameters
% -------------------------------------------------------------------------
theta                   = [1 0.02 -0.05 -0.04]';                        % forward connectivity S1 -> S2, dipole moments S1
p                       = length(theta);                                % number of free parameters
sigsqr                  = 1e-1;                                         % true, but unknown, data variance
n                       = m_f.nt*m_g.ne;                                % number of data points

% prior formulation
% -------------------------------------------------------------------------
mu_theta                = zeros(p,1);                                   % prior expectation parameter setting theta
sig_theta               = eye(p)*1e3;                                   % prior variancen parameter setting theta
mu_si_sigsqr            = [0;.1];                                       % prior shape and scale parameter setting \sigma^2


% fixed form variational Bayes Newton algorithm parameters
% -------------------------------------------------------------------------
max_i                   = 5;                                            % maximal number of fixed-form VB Newton algorithm
max_j                   = 8;                                            % maximal number of Newton iterations for m_theta optimization
max_k                   = 8;                                            % maximal number of Newton iterations for (m_sigsqr, s_sigssqr) optimization
delta                   = sqrt(eps);                                    % Hessian modification constant
c                       = 1e-4;                                         % backtracking constant c
rho                     = 0.1;                                          % backtracking constant rho
vareps                  = 1e-1;                                         % gradient norm convergence criterion
vardelta                = 1e-3;                                         % parameter norm change convergence criterion

% -------------------------------------------------------------------------
%                           Model Realization
% -------------------------------------------------------------------------
h_theta_fg              = erp_h(m_f,m_g,theta);                         % data expectation
y                       = mvnrnd(h_theta_fg, sigsqr*eye(n))';           % data sampling

save y y

return

% -------------------------------------------------------------------------
%                           Data Visualization
% -------------------------------------------------------------------------
% initialize figure
fig = figure;
set(fig, 'Color', [1 1 1])

% evaluate latent data
[f_theta_f_u]           = erp_f(m_f,theta);
f_theta_f_s1            = f_theta_f(1:9  ,:);
f_theta_f_s2            = f_theta_f(10:18,:);

% Input function u
% -------------------------------------------------------------------------
subplot(2,3,1)
plot(m_f.pst,u)
xlim([m_f.pst(1) m_f.pst(end)])
title('u', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% latent variable evolution
% -------------------------------------------------------------------------
subplot(2,3,2)
plot(m_f.pst,f_theta_f_s1)
xlim([m_f.pst(1) m_f.pst(end)])
title('x^{(1)}', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-10 40])

subplot(2,3,5)
plot(m_f.pst,f_theta_f_s2)
xlim([m_f.pst(1) m_f.pst(end)])
title('x^{(2)}', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-1 5])

% observed variables evolution
% -------------------------------------------------------------------------
subplot(2,3,3)
imagesc(m_f.pst, 1:numel(m_g.elab),spm_unvec(y, NaN(m_g.ne,m_f.nt)))
xlim([m_f.pst(1) m_f.pst(end)])
cb = colorbar;
title('y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel(cb, '\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('Electrodes', 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% selected electrodes
subplot(2,3,6)
eoi     = [47 49:51];
eoilab      = m_g.elab(eoi);
data        = spm_unvec(y, NaN(m_g.ne,m_f.nt));
plot(m_f.pst,data(eoi,:))
xlim([m_f.pst(1) m_f.pst(end)])
title('y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
legend(eoilab)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% -------------------------------------------------------------------------
%                           VBNL iteration 0
% -------------------------------------------------------------------------
% initialization of the variational parameters to prior parameters
m_theta                 = mu_theta;
s_theta                 = sig_theta;
ms_sigsqr               = mu_si_sigsqr;
vfearg.y                = y;
vfearg.m_f              = m_f;
vfearg.m_g              = m_g;
vfearg.mu_theta         = mu_theta;
vfearg.sig_theta        = sig_theta;
vfearg.mu_sigsqr        = mu_si_sigsqr(1);
vfearg.si_sigsqr        = mu_si_sigsqr(2);
vfearg.theta            = theta;

% parameter iteration arrays
s_theta_i               = NaN(p,p,max_i-1);
F_s_theta_i             = NaN(1,max_i-1);
m_theta_j               = NaN(p,max_j,max_i-1);
```

```
F_m_theta_j         = NaN(max_j,max_i-1);
ms_sigsqr_k         = NaN(2,max_k,max_i-1);
F_ms_sigsqr_k       = NaN(max_k,max_i-1);

% initialization and initial evaluation of the negative free energy
F                   = NaN(max_i-1);
F(1)                = vFE(m_theta, s_theta, ms_sigsqr, vfearg);

% ------------------------------------------------------------------------
%                         VBNL iterations 1,2,3,...
% ------------------------------------------------------------------------
for i = 2:max_i

    % User update
    % --------------------------------------------------------------------
    fprintf('Fixed-form VB Newton Iteration %4.0f ... \n', i-1)

    % Analytical update for theta_f variational variance parameter
    % --------------------------------------------------------------------
    % evaluate the variational variance update equation for the current choice of m_theta
    fprintf('   Estimating q(m_theta) variance   ... ')
    Jh_m_theta          = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g, spm_unvec(m_theta, theta)),m_theta,1)));
    s_theta             = spm_inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (spm_inv(sig_theta))));

    % save iterand and objective function
    s_theta_i(:,:,i-1)  = s_theta;
    F_s_theta_i(i-1)    =  vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % inform user
    fprintf('finished. \n')

    % Newton line search for theta_f variational expectation parameter
    % --------------------------------------------------------------------
    fprintf('   Estimating q(m_theta) expectation ... ')

    % save iterand and objective function
    m_theta_j(:,1,i-1)  = m_theta;
    F_m_theta_j(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % numerical gradient and Hessian of the negative free energy
    dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
    ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_m_theta) >= vareps
        for j = 2:max_j

            % gradient search direction
            p_j = -ddF_m_theta\dF_m_theta;

            % Hessian modification if p_j is not a descent direction
            if ~(p_j'*dF_m_theta < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_m_theta = ddF_m_theta + max(0,(delta - min(eig(ddF_m_theta)))*eye(p);

                % re-evaluate Newton search direction
                p_j = -ddF_m_theta\dF_m_theta;
            end

            % backtracking evaluation of the step length
            t_j                 = backtrack_vFE_m_theta(m_theta, s_theta, ms_sigsqr, vfearg, dF_m_theta, p_j,c,rho);

            % perform parameter update
            m_theta             = m_theta + t_j*p_j;

            % numerical gradient and Hessian of the negative free energy
            dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
            ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

            % record updates
            m_theta_j(:,j,i-1)    = m_theta;
            F_m_theta_j(j,i-1)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_m_theta'*p_j) < vareps || norm(m_theta_j(:,j,i-1) - m_theta_j(:,j-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished.\n')

    % Newton line search for sigma^2 variational parameters
    % --------------------------------------------------------------------
    fprintf('   Estimating q(sigma^2) parameters  ... ')

    % save iterand and objective function
    ms_sigsqr_k(:,1,i-1)  = ms_sigsqr;
    F_ms_sigsqr_k(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
    dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
    ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_ms_sigsqr) > vareps
        for k = 2:max_k

            % Newton search direction
            p_k = -ddF_ms_sigsqr\dF_ms_sigsqr;

            % Hessian modification if p_j is not a descent direction
            if ~(p_k'*dF_ms_sigsqr < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_ms_sigsqr = ddF_ms_sigsqr + max(0,(delta - min(eig(ddF_ms_sigsqr))))*eye(2);

                % re-evaluate Newton search direction
                p_k  = -ddF_ms_sigsqr\dF_ms_sigsqr;
            end

            % backtracking evaluation of the step length
            t_k = backtrack_vFE_ms_sigsqr(m_theta, s_theta, ms_sigsqr, vfearg, dF_ms_sigsqr, p_k,c,rho);

            % perform Newton update
            ms_sigsqr           = ms_sigsqr + t_k*p_k;

            % update remaining entries in free energy arrays
            dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
            ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

            % record updates
            ms_sigsqr_k(:,k,i-1)  = ms_sigsqr;
            F_ms_sigsqr_k(k,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_ms_sigsqr) < vareps || norm(ms_sigsqr_k(:,k,i-1) - ms_sigsqr_k(:,k-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished. \n')

    % free energy update
    % --------------------------------------------------------------------
    F(i) = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % user update
    % --------------------------------------------------------------------
    fprintf('vFE %6.3f  dvFE %6.3f \n', F(i), F(i) - F(i-1))
end

% ------------------------------------------------------------------------
%                       Algorithm Visualization
% ------------------------------------------------------------------------
% initialize figure
h = figure;
set(h, 'Color', [1 1 1])

% m_theta prior and approximate posterior
% ------------------------------------------------------------------------
% visualization of prior and approximated posterior over theta(1)
theta_min   = -1;
theta_max   =  4;
theta_res   = 1e4;
theta_space = linspace(theta_min, theta_max, theta_res);

subplot(2,3,1)
hold on
[ax,line1,line2] = plotyy(theta_space, pdf('Normal', theta_space, mu_theta(1) , sqrt(sig_theta(1,1))), theta_space, pdf('Normal', theta_space, m_theta(1), sqrt(s_theta(1,1))));
xlim([theta_min theta_max])
xlabel('\theta_1', 'FontSize', 24, 'FontName', 'Calibri Light')
legend('p(\theta_1)', 'q(\theta_1)')
set(ax, 'FontSize', 20, 'FontName', 'Calibri Light')

subplot(2,3,2)
hold on
plot3(mu_theta(2),mu_theta(3),mu_theta(4), 'ko', 'MarkerFaceColor', 'k')
lambda = eig(sig_theta(2:4,2:4));
[x,y,z] = ellipsoid(mu_theta(2),mu_theta(3),mu_theta(4),2*sqrt(5.991*lambda(1)),2*sqrt(5.991*lambda(2)),2*sqrt(5.991*lambda(3)));
hsurf = surf(x,y,z, 'EdgeColor', 'None','FaceLighting','gouraud');
camlight headlight
set(hsurf,'FaceColor',[0 0 1],'FaceAlpha',0.3)
grid on
view([45 20])
axis square
xlabel('\theta_2', 'FontName', 'Calibri Light', 'FontSize', 24)
ylabel('\theta_3', 'FontName', 'Calibri Light', 'FontSize', 24)
zlabel('\theta_4', 'FontName', 'Calibri Light', 'FontSize', 24, 'Rotation', 0)
title('p(\theta_2,\theta_3,\theta_4)', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

subplot(2,3,3)
hold on
plot3(m_theta(2),m_theta(3),m_theta(4), 'ko', 'MarkerFaceColor', 'k')
plot3(theta(2),theta(3),theta(4), 'k+', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
lambda = eig(s_theta(2:4,2:4));
[x,y,z] = ellipsoid(m_theta(2),m_theta(3),m_theta(4),2*sqrt(5.991*lambda(1)),2*sqrt(5.991*lambda(2)),2*sqrt(5.991*lambda(3)));
hsurf = surf(x,y,z, 'EdgeColor', 'None','FaceLighting','gouraud');
xlim([0.019  0.021])
ylim([-0.051 -0.0495])
```

```
zlim([-0.05 -0.03])
camlight headlight
set(hsurf,'FaceColor',[0 0 1],'FaceAlpha',0.3)
grid on
view([45 20])
axis square
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
xlabel('\theta_2', 'FontName', 'Calibri Light', 'FontSize', 24)
ylabel('\theta_3', 'FontName', 'Calibri Light', 'FontSize', 24)
zlabel('\theta_4', 'FontName', 'Calibri Light', 'FontSize', 24, 'Rotation', 0)
title('q(\theta_2,\theta_3,\theta_4)', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')

% sigma^2 prior and posterior
% ------------------------------------------------------------------------
sigsqr_pri      = linspace(1e-6,3e0,1e3);
p_sigsqr_pri    = lognpdf(sigsqr_pri, mu_si_sigsqr(1) , sqrt(mu_si_sigsqr(2)));
sigsqr_pos      = linspace(1e-6,0.2,1e3);
p_sigsqr_pos    = lognpdf(sigsqr_pos, ms_sigsqr(1) , sqrt(ms_sigsqr(2)));

subplot(2,3,4)
hold on
line(sigsqr_pri,p_sigsqr_pri, 'Color','b')
ax1             = gca; % current axes
ax1.XColor      = 'b';
ax1.YColor      = 'b';
ax1.FontName    = 'Calibri Light';
ax1.FontSize    = 20;
ax1_pos         = ax1.Position;
xlabel('\sigma^2', 'FontName', 'Calibri Light', 'FontSize', 24, 'Color', 'k')
ax2             = axes('Position',ax1_pos,'XAxisLocation','top','YAxisLocation','right','Color','None', 'FontName', 'Calibri Light', 'FontSize', 20);
ax2.XColor      = 'r';
ax2.YColor      = 'r';
line(sigsqr_pos,p_sigsqr_pos,'Parent',ax2, 'Color','r')

% variational free energy
% ------------------------------------------------------------------------
subplot(2,3,[5 6])
hold on
for i = 1:max_i-1
    plot(-F_m_theta_j(:,i)  , 'bo-', 'MarkerFaceColor', 'b')
    plot(-F_ms_sigsqr_K(:,i), 'ro-', 'MarkerFaceColor', 'r')
end
legend('m_{\theta}', '(m_{\sigma^2}, s_{\sigma^2})')
ylim([-14e3 -4e3])
xlim([0.5 8.5])
xlabel('Sub-Loop Iteration', 'FontSize', 24, 'FontName', 'Calibri Light')
title('F(m_\theta^{(i)}, \Sigma_\theta^{(i)}, m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)})', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 20)
set(gca, 'xtick', 1:8)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% remove Fieldtrip
% ------------------------------------------------------------------------
rmpath(genpath(fullfile(pwd, 'Fieldtrip')))

end

% ------------------------------------------------------------------------
%                               Subfunctions
% ------------------------------------------------------------------------
function [h_theta_fg]   = erp_h(m_f,m_g,theta)

% This function evaluates the expectation parameter generating function
%
% h : (theta_f,theta_g) |-> h(theta_f,theta_g) := vec(g(theta_g)*f(theta_f)
%
% of the delay differential equation model for ERPs
%
%       Inputs
%           m_f         : latent neural model structure
%           m_g         : forward model structure
%           theta       : model parameters
%
%       Outputs
%           h_theta_fg  : evaluated function
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluate component functions f and g
[f_theta_f, u]  = erp_f(m_f,theta);
g_theta_g       = erp_g(m_g,theta);

% evaluate concatenated function h
h_theta_fg      = 1e4*spm_vec(g_theta_g*f_theta_f);

end

function [f_theta_f,u]  = erp_f(m_f, vartheta)

% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_gen_erp.m of
% the SPM12 distribution. Based on a given parameter setting this function
% generates an input time-course and  integrates the resulting delay
% differential equation system.
%
%       Inputs
%           m_f         : latent neural model structure and fixed parameters
%           theta       : free latent neural model parameter
%
%       Outputs
%           f_theta_f   : evaluated latent neural model
%           u           : evaluated system input function
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% constant parameters
theta.R             = m_f.R;
theta.S             = m_f.S;
theta.T             = m_f.T;
theta.G             = m_f.G;
theta.H             = m_f.H;
theta.D             = m_f.D;
theta.A{1}          = m_f.A{1};
theta.A{2}          = m_f.A{2};
theta.A{3}          = m_f.A{3};
theta.C             = m_f.C;

% free parameters
theta.A{1}(2,1)     = vartheta(1);

% evaluate peri-stimulus time input function
t                   = ((1:m_f.ns)*m_f.dt)*1000;
delay               = m_f.ons(1)+128*theta.R(1,1);
scale               = m_f.dur(1)*exp(theta.R(1,2));
u                   = 32*exp(-(t - delay).^2/(2*scale^2))';

% integrate systen
x                   = m_f.x                                             ; % initial condition
f                   = m_f.f                                             ; % function handle @spm_fx_erp
dt                  = m_f.dt                                            ; % integration time bin
[fx,dfdx,D]         = f(x,u(1),theta,m_f)                               ; % dx(t)/dt and Jacobian df/dx and check for delay operator
p                   = max(abs(real(eig(full(dfdx)))))                   ; %
N                   = ceil(max(1, dt*p*2))                              ; %
n                   = spm_length(x)                                     ; %
Q                   = (spm_expm(dt*D*dfdx/N) - speye(n,n))*spm_inv(dfdx)  ; %
v                   = spm_vec(x)                                        ; % initialize state
y                   = NaN(length(v), length(u))                         ; % initialize state time-course

% cycle over time-steps
for i = 1:size(u,1)

    % update dx = (expm(dt*J) - I)*inv(J)*f(x,u)
    for j = 1:N
        v = v + Q*f(v,u(i),theta,m_f);
    end
    y(:,i) = v;

end

% re-order according to sources
ns                  = size(theta.C,1)                                   ; % number of sources
si                  = NaN(9,ns)                                         ; % source specific state indices
for s = 1:ns
    si(:,s) = s:ns:9*ns;
end

% create output
f_theta_f   = [];
for s = 1:ns
    f_theta_f  = [f_theta_f; y(si(:,s),:)];
end
end

function [f,J,Q]        = spm_fx_erp(x,u,P,M)

% state equations for a neural mass model of erps
% FORMAT [f,J,D] = spm_fx_erp(x,u,P,M)
% FORMAT [f,J]   = spm_fx_erp(x,u,P,M)
% FORMAT [f]     = spm_fx_erp(x,u,P,M)
% x     - state vector
%   x(:,1) - voltage (spiny stellate cells)
%   x(:,2) - voltage (pyramidal cells) +ve
%   x(:,3) - voltage (pyramidal cells) -ve
%   x(:,4) - current (spiny stellate cells)    depolarizing
%   x(:,5) - current (pyramidal cells)         depolarizing
%   x(:,6) - current (pyramidal cells)         hyperpolarizing
%   x(:,7) - voltage (inhibitory interneurons)
%   x(:,8) - current (inhibitory interneurons) depolarizing
%   x(:,9) - voltage (pyramidal cells)
%
% f       - dx(t)/dt  = f(x(t))
% J       - df(t)/dx(t)
% D       - delay operator dx(t)/dt = f(x(t - d))
%                                   = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
%  M.pF.E = [32 16 4];            % extrinsic rates (forward, backward, lateral)
%  M.pF.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2 g3, g4)
%  M.pF.D = [2 16];               % propogation delays (intrinsic, extrinsic)
%  M.pF.G = [4 32];               % receptor densities (excitatory, inhibitory)
%  M.pF.T = [8 16];               % synaptic constants (excitatory, inhibitory)
%  M.pF.R = [1 1/2];              % parameter of static nonlinearity
%
```

```
%_______________________________________________________________________
% David O, Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neuronal dynamics. NeuroImage 20: 1743-1755
%_______________________________________________________________________
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging

% Karl Friston
% $Id: spm_fx_erp.m 5369 2013-03-28 20:09:27Z karl $

% get dimensions and configure state variables
%-----------------------------------------------------------------------
n       = length(P.A{1});       % number of sources
x       = spm_unvec(x,M.x);     % neuronal states

% [default] fixed parameters
%-----------------------------------------------------------------------
E       = [1 1/2 1/8]*32;               % extrinsic rates (forward, backward, lateral)
G       = [1 4/5 1/4 1/4]*128;          % intrinsic rates (g1 g2 g3 g4)
D       = [2 16];                       % propogation delays (intrinsic, extrinsic)
H       = [4 32];                       % receptor densities (excitatory, inhibitory)
T       = [8 16];                       % synaptic constants (excitatory, inhibitory)
R       = [2 1]/3;                      % parameters of static nonlinearity

% test for free parameters on intrinsic connections
%-----------------------------------------------------------------------
G       = G.*exp(P.H);
G       = ones(n,1)*G;

% no exponential transforms to foster parameter identifiability
% ----------------------------------------------------------------------
A{1}    = P.A{1}*E(1);
A{2}    = P.A{2}*E(2);
A{3}    = P.A{3}*E(3);
C       = P.C;

% intrinsic connectivity and parameters
%-----------------------------------------------------------------------
Te      = T(1)/1000*exp(P.T(:,1));      % excitatory time constants
Ti      = T(2)/1000*exp(P.T(:,2));      % inhibitory time constants
He      = H(1)*exp(P.G(:,1));           % excitatory receptor density
Hi      = H(2)*exp(P.G(:,2));           % inhibitory receptor density

% pre-synaptic inputs: s(V)
%-----------------------------------------------------------------------
S       = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
%-----------------------------------------------------------------------
U       = C*u(:)*2;

% State: f(x)
%=======================================================================
f(:,1) = x(:,4)                                                                 ; % voltage change (spiny stellate cells)
f(:,2) = x(:,5)                                                                 ; % positive voltage change (pyramidal cells) +ve
f(:,3) = x(:,6)                                                                 ; % negative voltage change (pyramidal cells) -ve
f(:,4) = (He.*((A{1} + A{3})*S(:,9) + G(:,1).*S(:,9) + U) - 2*x(:,4) - x(:,1)./Te)./Te    ; % current change (spiny stellate cells)    depolarizing
f(:,5) = (He.*((A{2} + A{3})*S(:,9) + G(:,2).*S(:,1)) - 2*x(:,5) - x(:,2)./Te)./Te        ; % current change (pyramidal cells)          depolarizing
f(:,6) = (Hi.*G(:,4).*S(:,7) - 2*x(:,6) - x(:,3)./Ti)./Ti                       ; % current change (pyramidal cells)          hyperpolarizing
f(:,7) = x(:,8)                                                                 ; % voltage change (inhibitory interneurons)
f(:,8) = (He.*((A{2} + A{3})*S(:,9) + G(:,3).*S(:,9)) - 2*x(:,8) - x(:,7)./Te)./Te          ; % current change(inhibitory interneurons) depolarizing
f(:,9) = x(:,5) - x(:,6)                                                        ; % voltage (pyramidal cells)


% vectorize
f      = spm_vec(f);

% avoid infinite recursion of spm_diff
if nargout < 2
    return
end

% Jacobian
%=======================================================================
J  = spm_diff(M.f,x,u,P,M,1);

% delays
%=======================================================================
% Delay differential equations can be integrated efficiently (but
% approximately) by absorbing the delay operator into the Jacobian
%
%    dx(t)/dt     = f(x(t - d))
%                 = Q(d)f(x(t))
%
%    J(d)         = Q(d)df/dx
%-----------------------------------------------------------------------
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,n)).*De;
Di = (1 - speye(9,9)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D  = Di + De;

% Implement: dx(t)/dt = f(x(t - d)) = inv(1 + D.*dfdx)*f(x(t))
%                     = Q*f = Q*J*x(t)
%-----------------------------------------------------------------------
Q  = spm_inv(speye(length(J)) + D.*J);

end

function [g_theta_g]    = erp_g(m_g, vartheta)

% This function evaluates the EEG forward model of the delay differential
% equation model for ERPs based on a structural formulation of the forward
% model and parameter setting
%
%   Inputs
%           m_g         : EEG forward model structure and fixed parameters
%           vartheta    : EEG lead-field free parameters
%
%    Outputs
%           g_theta_g   : evaluated EEG forward model
%
% Copyright (C) Dirk Ostwald
%-----------------------------------------------------------------------
% constant parameters
theta.Lpos  = m_g.Lpos;
theta.L     = m_g.L;
theta.J     = m_g.J;

% free parameters
theta.L(1,1) = vartheta(2);
theta.L(2,1) = vartheta(3);
theta.L(3,1) = vartheta(4);

% evaluate constants
nd          = size(theta.L,2);                                          % number of dipoles
ne          = size(m_g.sens.chanpos,1);                                 % number of electrodes
dp          = 1e-3*theta.Lpos;                                          % dipole coordinates adjusted for fieldtrip

% compute canonical dipole lead-field using Fieldtrip
L_can  = NaN(ne,3,nd);
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end

% rescale lead-field according to SPM
L_can = L_can*(10^1)^-round(log10(max(max(max(abs(L_can))))/8)/1);

% evaluate channel predictions based on dipole moments
L_dip = NaN(ne,nd);
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*theta.L(:,i);
end

% evaluate g(\theta_g)
g_theta_g = [];
for i = 1:nd
    g_theta_g = [g_theta_g kron(theta.J,L_dip(:,i))];
end
end

function [vFE]          = vFE(m_theta,s_theta,ms_sigsqr,vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : p x 1 variational expectation for theta
%       s_theta    : p x p variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters   for sigma^2
%       vfearg     : additional argument structure with fields
%           .y                  : n x 1 array   - data
%           .h                  : string        - function handle
%           .mu_theta           : scalar prior expectation for theta
%           .sig_theta          : scalar prior variance for theta
%           .mu_sigsqr          : scalar prior scale parameter for sigma^2
%           .si_sigsqr          : scalar prior shape parameter for sigma^2
%           .theta              : true, but unknown, parameter structure
%   Outputs
%           vFE                 : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% ----------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
m_f             = vfearg.m_f;
m_g             = vfearg.m_g;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;
theta           = vfearg.theta;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta) and J^h(m_theta)
h_m_theta       = erp_h(m_f,m_g, spm_unvec(m_theta, theta));
Jh_m_theta      = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g,spm_unvec(m_theta, theta)),m_theta,1)));
```

```
% evaluation of the variational free energy value
T1              = -(n/2)*log(2*pi)          - (n/2)*m_sigsqr            - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2              = -(p/2)*log(2*pi)          - (1/2)*log(det(sig_theta))  - (1/2)*((m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
T3              = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                 - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T4              =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
T5              =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                                  ;

% evaluate the negative free energy
vFE             = -(T1 + T2 + T3 + T4 + T5);

end

function [t_k]          = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsqr, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k         : p x 1     array of function argument
%       s_theta     : p x p     array of additional vFE arguments
%       ms_sigsqr   : 2 x 1     array of additional vFE arguments
%       vfearg      : additional input for vFE
%       df_x_k       :p_f x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(x_k, s_theta, ms_sigsqr, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(x_k + t_k*p_k, s_theta, ms_sigsqr, vfearg);

% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(x_k+t_k*p_k, s_theta, ms_sigsqr, vfearg);
end
end

function [t_k]          = backtrack_vFE_ms_sigsqr(m_theta, s_theta, x_k, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       m_theta     : p x 1   array of additional vFE arguments
%       s_theta     : p x p array of additional vFE arguments
%       x_k         : 2 x 1     array of function arguments
%       vfearg      : additional input for vFE
%       df_x_k      : 2 x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(m_theta, s_theta, x_k, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(m_theta, s_theta, x_k + t_k*p_k, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)|| x_k(2) + t_k*p_k(2) < 0;
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(m_theta, s_theta, x_k+t_k*p_k, vfearg);
end
end
```

PDDE_7.m – ERP-DCM estimation of condition-specific effects (Figure 11)

```
function pdde_7_R1

% This function implements the estimation of condition-specific latent
% input connectivity in a basic delay differential equation ERP model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
clc
close all

% set the random number generator for reproducible sampling results
reset(RandStream.getGlobalStream)

% invoke Fieldtrip
addpath(genpath(fullfile(pwd, 'Fieldtrip')))

% -------------------------------------------------------------------------
%                             Model Formulation
% -------------------------------------------------------------------------
% neural model structure and parameters
% -------------------------------------------------------------------------
% structural aspects and fixed parameters
m_f.f               = @spm_fx_erp;                                      % neural evolution function
m_f.ns              = 100;                                              % number of time bins
m_f.dt              = 0.002;                                            % time bin length (sec)
m_f.pst             = (0:m_f.ns-1)*m_f.dt;                              % peristimulus time (msec)
m_f.nt              = length(m_f.pst);                                  % number of time points
m_f.n               = 2;                                                % number of neural masses
m_f.m               = 9;                                                % number of neural states per neural mass
m_f.ons             = -10;                                              % system input onset (ms)
m_f.dur             = 1;                                                % duration (width) of the input function
m_f.x               = sparse(m_f.n,m_f.m);                              % initial condition
m_f.R               = [ 0.16 -0.22];                                    % input function parameters rho_1 and rho_2
m_f.S               = [-9e-4  0.08];                                    % static nonlinearity (activation function) parameters
m_f.T               = [ 0.16  0.13;  0.07 -0.82];                       % source-specific excitatory and inhibitory time constants
m_f.G               = [-0.11  0.45;  0.02 -0.63];                       % source-specific excitatory and inhibitory receotor densities
m_f.H               = [ 0.02 -0.07   0.18 -0.17];                       % source-independent intrinsic connectivity parameters
m_f.D               = [ 0    -0.52; -0.06  0   ];                       % between source delay parameters
m_f.X               = [0 1]';                                           % design matrix
m_f.A{1}            = [0 0; 1 0];                                       % forward connectivity
m_f.A{2}            = [0 0; 0 0];                                       % backward connectivity
m_f.A{3}            = zeros(2,2);                                       % lateral connectivity
m_f.B               = NaN;                                              % condition-specific effect
m_f.C               = [1 0]';                                           % input connectivity

% forward model structure and parameters
% -------------------------------------------------------------------------
% load an existing dipole model
load(fullfile(cd, 'Data\SIM', 'dip_model_struct.mat'))
m_g.vol             = dip_model_struct.vol;                             % forward model specification
m_g.sens            = dip_model_struct.sens ;                           % sensor space specification
m_g.elab            = dip_model_struct.sens.label;                      % electrode labels
m_g.ne              = numel(m_g.elab);                                  % number of electrodes
m_g.X0              = zeros(m_f.ns*m_g.ne, 3*m_g.ne);                   % confound design matrix
m_g.Lpos            = [42 -31 58; 54 -22 10]';                          % dipole coordinate specifications
m_g.L               = [0.02 -0.05 -0.04; 0.17 -0.50 -0.17]';            % dipole moment specification
m_g.J               = [-0.1 0 0 0 0 0 -0.46 0 1];                       % state weighting specification

% likelihood model parameters
% -------------------------------------------------------------------------
n                   = m_f.nt*m_g.ne;                                    % number of data points

% prior formulation
% -------------------------------------------------------------------------
mu_theta            = 0;                                                % prior expectation parameter setting theta
sig_theta           = 1e3;                                              % prior variance parameter setting theta
mu_si_sigsqr        = [4.6;1];                                          % prior shape and scale parameter setting \sigma^2

% fixed form variational Bayes Newton algorithm parameters
% -------------------------------------------------------------------------
max_i               = 5;                                                % maximal number of fixed-form VB Newton algorithm
max_j               = 8;                                                % maximal number of Newton iterations for m_theta optimization
max_k               = 8;                                                % maximal number of Newton iterations for (m_sigsqr, s_sigssqr) optimization
delta               = sqrt(eps);                                        % Hessian modification constant
c                   = 1e-4;                                             % backtracking constant c
rho                 = 0.9;                                              % backtracking constant rho
vareps              = 1e2;                                              % gradient norm convergence criterion
vardelta            = 1e-3;                                             % parameter norm change convergence criterion

% -------------------------------------------------------------------------
%                             Model Realization
% -------------------------------------------------------------------------
theta               = 1;                                                % true, but unknown, condition specific effect
p                   = length(theta);                                    % number of parameters
ncond               = size(m_f.X,1);                                    % number of conditions
sigsqr              = 1e-1;                                             % true, but unknown, data variance
h_theta_fg          = @erp_h(m_f,m_g,theta);                            % data expectation
y                   = mvnrnd(h_theta_fg, sigsqr*eye(n*ncond))';         % data sampling

% -------------------------------------------------------------------------
%                           Data Visualization
% -------------------------------------------------------------------------
% initialize figure
fig = figure;
set(fig, 'Color', [1 1 1])

% evaluate cell array data form for plotting
% -------------------------------------------------------------------------
yc                  = spm_unvec(y,{NaN(m_g.ne,m_f.nt) NaN(m_g.ne,m_f.nt)});
[f_theta_f,u]   = erp_f(m_f,theta);

% cycle over conditions
for cond = 1:size(m_f.X,1)

    % separate latent between sources
    % ---------------------------------------------------------------------
    f_theta_f_s1        = f_theta_f{cond}(1:9  ,:);
    f_theta_f_s2        = f_theta_f{cond}(10:18,:);

    % condition specific latent variable evolution
    % ---------------------------------------------------------------------
    subplot(3,4,(cond-1)*4+1)
    plot(m_f.pst,f_theta_f_s1)
    xlim([m_f.pst(1) m_f.pst(end)])
    title('x^{(1)}', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
    xlabel('Time [s]', 'FontSize', 22, 'FontName', 'Calibri Light')
    ylabel('a.u.', 'FontSize', 22, 'FontName', 'Calibri Light')
    set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')
    ylim([-10 40])

    subplot(3,4,(cond-1)*4+2)
    plot(m_f.pst,f_theta_f_s2)
    xlim([m_f.pst(1) m_f.pst(end)])
    title('x^{(2)}', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
    xlabel('Time [s]', 'FontSize', 22, 'FontName', 'Calibri Light')
    ylabel('a.u.', 'FontSize', 22, 'FontName', 'Calibri Light')
    set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')
    ylim([-1 5])

    % condition specific observed variable evolution
    % ---------------------------------------------------------------------
    subplot(3,4,(cond-1)*4+3)
    imagesc(m_f.pst, 1:numel(m_g.elab),yc{cond})
    xlim([m_f.pst(1) m_f.pst(end)])
    cb = colorbar;
    title('Y', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
    ylabel(cb, '\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 18)
    xlabel('Time [s]', 'FontSize', 22, 'FontName', 'Calibri Light')
    ylabel('Electrodes', 'FontName', 'Calibri Light', 'FontSize', 22)
    set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')

    % selected electrodes
    subplot(3,4,(cond-1)*4+4)
    eoi         = [47 49:51];
    eoilab      = m_g.elab(eoi);
    data        = yc{cond};
    plot(m_f.pst,data(eoi,:))
    xlim([m_f.pst(1) m_f.pst(end)])
    title('Y', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
    legend(eoilab)
    xlabel('Time [s]', 'FontSize', 22, 'FontName', 'Calibri Light')
    ylabel('\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 20)
    set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')

end

% -------------------------------------------------------------------------
%                             VBNL iteration 0
% -------------------------------------------------------------------------
% initialization of the variational parameters to prior parameters
m_theta             = mu_theta;
s_theta             = sig_theta;
ms_sigsqr           = mu_si_sigsqr;
vfearg.y            = y;
vfearg.m_f          = m_f;
vfearg.m_g          = m_g;
vfearg.mu_theta     = mu_theta;
vfearg.sig_theta    = sig_theta;
vfearg.mu_sigsqr    = mu_si_sigsqr(1);
vfearg.si_sigsqr    = mu_si_sigsqr(2);
vfearg.theta        = theta;

% parameter iteration arrays
s_theta_i           = NaN(p,p,max_i-1);
F_s_theta_i         = NaN(1,max_i-1);
m_theta_j           = NaN(p,max_j,max_i-1);
F_m_theta_j         = NaN(max_j,max_i-1);
ms_sigsqr_k         = NaN(2,max_k,max_i-1);
F_ms_sigsqr_k       = NaN(max_k,max_i-1);

% initialization and initial evaluation of the negative free energy
F                   = NaN(max_i-1);
F(1)                = vFE(m_theta, s_theta, ms_sigsqr, vfearg);

% -------------------------------------------------------------------------
%                       VBNL iterations 1,2,3,...
```

```
% ------------------------------------------------------------------------
for i = 2:max_i

    % User update
    % ------------------------------------------------------------------------
    fprintf('Fixed-form VB Newton Iteration %4.0f ... \n', i-1)

    % Analytical update for theta_f variational variance parameter
    % ------------------------------------------------------------------------
    % evaluate the variational variance update equation for the current choice of m_theta
    fprintf('   Estimating q(m_theta) variance  ... ')
    Jh_m_theta          = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g, spm_unvec(m_theta, theta)),m_theta,1)));
    s_theta             = spm_inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (spm_inv(sig_theta))));

    % save iterand and objective function
    s_theta_i(:,:,i-1)  = s_theta;
    F_s_theta_i(i-1)    =  vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % inform user
    fprintf('finished. \n')

    % Newton line search for theta_f variational expectation parameter
    % ------------------------------------------------------------------------
    fprintf('   Estimating q(m_theta) expectation ... ')

    % save iterand and objective function
    m_theta_j(:,1,i-1)  = m_theta;
    F_m_theta_j(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % numerical gradient and Hessian of the negative free energy
    dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
    ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_m_theta) >= vareps
        for j = 2:max_j

            % gradient search direction
            p_j = -ddF_m_theta\dF_m_theta;

            % Hessian modification if p_j is not a descent direction
            if ~(p_j'*dF_m_theta < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_m_theta = ddF_m_theta + max(0,(delta - min(eig(ddF_m_theta))))*eye(p);

                % re-evaluate Newton search direction
                p_j = -ddF_m_theta\dF_m_theta;
            end

            % backtracking evaluation of the step length
            t_j             = backtrack_vFE_m_theta(m_theta, s_theta, ms_sigsqr, vfearg, dF_m_theta, p_j,c,rho);

            % perform parameter update
            m_theta         = m_theta + t_j*p_j;

            % numerical gradient and Hessian of the negative free energy using long-step spm_diff variant
            dF_m_theta      = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
            ddF_m_theta     = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

            % record updates
            m_theta_j(:,j,i-1)    = m_theta;
            F_m_theta_j(j,i-1)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_m_theta) < vareps || norm(m_theta_j(:,j,i-1) - m_theta_j(:,j-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished.\n')

    % Newton line search for sigma^2 variational parameters
    % ------------------------------------------------------------------------
    fprintf('   Estimating q(sigma^2) parameters  ... ')

    % save iterand and objective function
    ms_sigsqr_k(:,1,i-1)  = ms_sigsqr;
    F_ms_sigsqr_k(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
    dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
    ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_ms_sigsqr) > vareps
        for k = 2:max_k

            % Newton search direction
            p_k = -ddF_ms_sigsqr\dF_ms_sigsqr;

            % Hessian modification if p_j is not a descent direction
            if ~(p_k'*dF_ms_sigsqr < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_ms_sigsqr = ddF_ms_sigsqr + max(0,(delta - min(eig(ddF_ms_sigsqr))))*eye(2);

                % re-evaluate Newton search direction
                p_k  = -ddF_ms_sigsqr\dF_ms_sigsqr;
            end

            % backtracking evaluation of the step length
            t_k = backtrack_vFE_ms_sigsqr(m_theta, s_theta, ms_sigsqr, vfearg, dF_ms_sigsqr, p_k,c,rho);

            % perform Newton update
            ms_sigsqr           = ms_sigsqr + t_k*p_k;

            % update remaining entries in free energy arrays
            dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
            ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

            % record updates
            ms_sigsqr_k(:,k,i-1)  = ms_sigsqr;
            F_ms_sigsqr_k(k,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_ms_sigsqr) < vareps || norm(ms_sigsqr_k(:,k,i-1) - ms_sigsqr_k(:,k-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished. \n')

    % free energy update
    % ------------------------------------------------------------------------
    F(i) = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % user update
    % ------------------------------------------------------------------------
    fprintf('vFE %6.3f  dvFE %6.3f \n', F(i), F(i) - F(i-1))
end

% ------------------------------------------------------------------------
%                           Algorithm Visualization
% ------------------------------------------------------------------------

% m_theta prior and approximate posterior
% ------------------------------------------------------------------------
subplot(3,4,9)
theta_f_x = linspace(-.5,1.5,1e4);
hold on
plot(theta_f_x, pdf('Normal', theta_f_x, m_theta  , sqrt(s_theta))  , 'r-');
plot(theta_f_x, pdf('Normal', theta_f_x, mu_theta , sqrt(sig_theta)), 'b');
plot(mu_theta       , 0, 'o', 'MarkerFaceColor' , 'b', 'MarkerSize', 8)
plot(m_theta        , 0, 'o', 'MarkerFaceColor' , 'r', 'MarkerSize', 8);
xlabel('\theta', 'FontSize', 22, 'FontName', 'Calibri Light')
legend('q(\theta)', 'p(\theta)', 'Location', 'NorthWest')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
xlim([theta_f_x(1) theta_f_x(end)])

% sigma^2 prior and posterior
% ------------------------------------------------------------------------
subplot(3,4,10)
sigsqr_x = linspace(1e-3,1,1e4);
hold on
plot(sigsqr_x, lognpdf(sigsqr_x, ms_sigsqr(1)    , sqrt(ms_sigsqr(2)))       , 'r')
plot(sigsqr_x, lognpdf(sigsqr_x, mu_si_sigsqr(1) , sqrt(mu_si_sigsqr(2)))    , 'b')
plot(exp(mu_si_sigsqr(1) + .5*mu_si_sigsqr(2))   , 0, 'o', 'MarkerFaceColor' , 'b', 'MarkerSize', 8)
plot(exp(ms_sigsqr(1)      + .5*ms_sigsqr(2))       , 0, 'o', 'MarkerFaceColor' , 'r', 'MarkerSize', 8)
xlabel('\sigma^2', 'FontSize', 22, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
legend('q(\sigma^2)', 'p(\sigma^2)', 'Location', 'NorthEast')
xlim([0 sigsqr_x(end)])


% m_theta variational free energy ascent on first VBNL iteration
% ------------------------------------------------------------------------
subplot(3,4,11)
res             = 3e1;
m_theta_min     = -.5;
m_theta_max     =  2;
m_theta_res     = res;
m_theta_space   = linspace(m_theta_min,m_theta_max,m_theta_res);
vFE_m_theta     = NaN(1,res);
for i = 1:res
    vFE_m_theta(i) = vFE(m_theta_space(i), s_theta_i(:,:,1), mu_si_sigsqr, vfearg);
end
hold on
plot(m_theta_space, -vFE_m_theta, 'b')
plot(m_theta_j(1,1:3,1), -F_m_theta_j(1:3,1), 'o', 'MarkerFaceColor', 'w', 'MarkerSize', 8)
xlabel('m_\theta', 'FontSize', 22, 'FontName', 'Calibri Light')
legend('F')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
xlim([m_theta_min m_theta_max])

% variational free energy
% ------------------------------------------------------------------------
subplot(3,4,12)
hold on
plot(-F(2:end), 'ko-', 'MarkerFaceColor', 'k')
xlim([0 max_i])
```

```
title('F(m_\theta^{(i)}, \Sigma_\theta^{(i)}, m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)})', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 16)
xlabel('i', 'FontSize', 22, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% remove Fieldtrip
% -------------------------------------------------------------------------
rmpath(genpath(fullfile(pwd, 'Fieldtrip')))

end

% -------------------------------------------------------------------------
%                           Subfunctions
% -------------------------------------------------------------------------
function [h_theta_fg]   = erp_h(m_f,m_g,theta)

% This function evaluates the expectation parameter generating function
%
% h : (theta_f,theta_g) |-> h(theta_f,theta_g) := vec(g(theta_g)*f(theta_f)
%
% of the delay differential equation model for ERPs
%
%       Inputs
%           m_f         : latent neural model structure
%           m_g         : forward model structure
%           theta       : model parameters
%
%       Outputs
%           h_theta_fg  : evaluated function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluate component functions f and g
[f_theta_f, u]  = erp_f(m_f,theta);
g_theta_g       = erp_g(m_g);

% evaluate condition specific concatenated function h
% -------------------------------------------------------------------------
h_theta_fg = cell(1, numel(f_theta_f));
for cond = 1:numel(f_theta_f)
    h_theta_fg{cond} = 1e4*spm_vec(g_theta_g*f_theta_f{cond});
end

% vectorize over conditions
h_theta_fg = spm_vec(h_theta_fg);

end

function [f_theta_f,u]  = erp_f(m_f,vartheta)

% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_gen_erp.m of
% the SPM12 distribution. Based on a given parameter setting this function
% generates an input time-course and  integrates the resulting delay
% differential equation system.
%
%       Inputs
%           m_f         : latent neural model structure and fixed parameters
%           theta       : free latent neural model parameter
%
%       Outputs
%           f_theta_f   : evaluated latent neural model
%           u           : evaluated system input function
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.R             = m_f.R;
theta.S             = m_f.S;
theta.T             = m_f.T;
theta.G             = m_f.G;
theta.H             = m_f.H;
theta.D             = m_f.D;
theta.X             = m_f.X;
theta.A{2}          = m_f.A{2};
theta.A{3}          = m_f.A{3};
theta.B             = m_f.B;
theta.C             = m_f.C;

% free parameter
theta.B             = vartheta;

% evaluate peri-stimulus time input function
t                   = ((1:m_f.ns)*m_f.dt)*1000;
delay               = m_f.ons(1)+128*theta.R(1,1);
scale               = m_f.dur(1)*exp(theta.R(1,2));
u                   = 32*exp(-(t - delay).^2/(2*scale^2))';

% initialize output array
f_theta_f           = cell(1,size(theta.X,1));

% cycle over conditions
% -------------------------------------------------------------------------
for cond = 1:size(theta.X,1)

    % condition specific forward connectivity matrix
    theta.A{1}      = m_f.A{1} + theta.X(cond).*[0 0; theta.B 0];

    % integrate systen
    x               = m_f.x                                             ; % initial condition
    f               = m_f.f                                             ; % function handle @spm_fx_erp
    dt              = m_f.dt                                            ; % integration time bin
    [fx,dfdx,D]     = f(x,u(1),theta,m_f)                               ; % dx(t)/dt and Jacobian df/dx and check for delay operator
    p               = max(abs(real(eig(full(dfdx)))))                   ; %
    N               = ceil(max(1, dt*p*2))                              ; %
    n               = spm_length(x)                                     ; %
    Q               = {spm_expm(dt*D*dfdx/N) - speye(n,n)}*spm_inv(dfdx) ; %
    v               = spm_vec(x)                                        ; % initialize state
    y               = NaN(length(v), length(u))                         ; % initialize state time-course

    % cycle over time-steps
    for i = 1:size(u,1)

        % update dx = (expm(dt*J) - I)*inv(J)*f(x,u)
        for j = 1:N
            v = v + Q*f(v,u(i),theta,m_f);
        end
        y(:,i) = v;

    end

    % re-order according to sources
    ns              = size(theta.C,1)                                   ; % number of sources
    si              = NaN(9,ns)                                         ; % source specific state indices
    for s = 1:ns
        si(:,s) = s:ns:9*ns;
    end

    % create output
    f_theta_f{cond}   = [];
    for s = 1:ns
        f_theta_f{cond}  = [f_theta_f{cond}; y(si(:,s),:)];
    end

end
end

function [f,J,Q]        = spm_fx_erp(x,u,P,M)

% state equations for a neural mass model of erps
% FORMAT [f,J,D] = spm_fx_erp(x,u,P,M)
% FORMAT [f,J]   = spm_fx_erp(x,u,P,M)
% FORMAT [f]     = spm_fx_erp(x,u,P,M)
% x      - state vector
%   x(:,1) - voltage (spiny stellate cells)
%   x(:,2) - voltage (pyramidal cells) +ve
%   x(:,3) - voltage (pyramidal cells) -ve
%   x(:,4) - current (spiny stellate cells)    depolarizing
%   x(:,5) - current (pyramidal cells)         depolarizing
%   x(:,6) - current (pyramidal cells)         hyperpolarizing
%   x(:,7) - voltage (inhibitory interneurons)
%   x(:,8) - current (inhibitory interneurons) depolarizing
%   x(:,9) - voltage (pyramidal cells)
%
% f        - dx(t)/dt  = f(x(t))
% J        - df(t)/dx(t)
% D        - delay operator dx(t)/dt = f(x(t - d))
%                                    = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
%  M.pF.E = [32 16 4];             % extrinsic rates (forward, backward, lateral)
%  M.pF.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2 g3, g4)
%  M.pF.D = [2 16];                % propogation delays (intrinsic, extrinsic)
%  M.pF.G = [4 32];                % receptor densities (excitatory, inhibitory)
%  M.pF.T = [8 16];                % synaptic constants (excitatory, inhibitory)
%  M.pF.R = [1 1/2];               % parameter of static nonlinearity
%
% _________________________________________________________________________
% David O, Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neuronal dynamics. NeuroImage 20: 1743-1755
% _________________________________________________________________________
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging

% Karl Friston
% $Id: spm_fx_erp.m 5369 2013-03-28 20:09:27Z karl $

% get dimensions and configure state variables
%--------------------------------------------------------------------------
n       = length(P.A{1});                   % number of sources
x       = spm_unvec(x,M.x);                 % neuronal states

% [default] fixed parameters
%--------------------------------------------------------------------------
E       = [1 1/2 1/8]*32;                   % extrinsic rates (forward, backward, lateral)
G       = [1 4/5 1/4 1/4]*128;              % intrinsic rates (g1 g2 g3 g4)
D       = [2 16];                           % propagation delays (intrinsic, extrinsic)
H       = [4 32];                           % receptor densities (excitatory, inhibitory)
T       = [8 16];                           % synaptic constants (excitatory, inhibitory)
R       = [2 1]/3;                          % parameters of static nonlinearity

% test for free parameters on intrinsic connections
%--------------------------------------------------------------------------
G       = G.*exp(P.H);
```

```
G       = ones(n,1)*G;

% no exponential transforms to foster parameter identifiability
% -------------------------------------------------------------------------
A{1}    = P.A{1}*E(1);
A{2}    = P.A{2}*E(2);
A{3}    = P.A{3}*E(3);
C       = P.C;

% intrinsic connectivity and parameters
%--------------------------------------------------------------------------
Te      = T(1)/1000*exp(P.T(:,1));      % excitatory time constants
Ti      = T(2)/1000*exp(P.T(:,2));      % inhibitory time constants
He      = H(1)*exp(P.G(:,1));           % excitatory receptor density
Hi      = H(2)*exp(P.G(:,2));           % inhibitory receptor density

% pre-synaptic inputs: s(V)
%--------------------------------------------------------------------------
S       = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
%--------------------------------------------------------------------------
U       = C*u(:)*2;

% State: f(x)
%==========================================================================
f(:,1) = x(:,4)                                                             ; % voltage change (spiny stellate cells)
f(:,2) = x(:,5)                                                             ; % positive voltage change (pyramidal cells) +ve
f(:,3) = x(:,6)                                                             ; % negative voltage change (pyramidal cells) -ve
f(:,4) = (He.*((A{1} + A{3})*S(:,9) + G(:,1).*S(:,9) + U) - 2*x(:,4) - x(:,1)./Te)./Te     ; % current change (spiny stellate cells)    depolarizing
f(:,5) = (He.*((A{2} + A{3})*S(:,9) + G(:,2).*S(:,1)) - 2*x(:,5) - x(:,2)./Te)./Te        ; % current change (pyramidal cells)          depolarizing
f(:,6) = (Hi.*G(:,4).*S(:,7) - 2*x(:,6) - x(:,3)./Ti)./Ti                   ; % current change (pyramidal cells)          hyperpolarizing
f(:,7) = x(:,8)                                                             ; % voltage change (inhibitory interneurons)
f(:,8) = (He.*((A{2} + A{3})*S(:,9) + G(:,3).*S(:,9)) - 2*x(:,8) - x(:,7)./Te)./Te         ; % current change(inhibitory interneurons) depolarizing
f(:,9) = x(:,5) - x(:,6)                                                    ; % voltage (pyramidal cells)


% vectorize
f      = spm_vec(f);

% avoid infinite recursion of spm_diff
if nargout < 2
    return
end

% Jacobian
%==========================================================================
J  = spm_diff(M.f,x,u,P,M,1);

% delays
%==========================================================================
% Delay differential equations can be integrated efficiently (but
% approximately) by absorbing the delay operator into the Jacobian
%
%    dx(t)/dt     = f(x(t - d))
%                 = Q(d)f(x(t))
%
%    J(d)         = Q(d)df/dx
%--------------------------------------------------------------------------
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,n)).*De;
Di = (1 - speye(9,9)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D  = Di + De;

% Implement: dx(t)/dt = f(x(t - d)) = inv(1 + D.*dfdx)*f(x(t))
%                     = Q*f = Q*J*x(t)
%--------------------------------------------------------------------------
Q  = spm_inv(speye(length(J)) + D.*J);

end

function [g_theta_g]    = erp_g(m_g)

% This function evaluates the EEG forward model of the delay differential
% equation model for ERPs based on a structural formulation of the forward
% model and parameter setting
%
%   Inputs
%           m_g         : EEG forward model structure and fixed parameters
%           theta     : EEG lead-field free parameters

%    Outputs
%           g_theta_g   : evaluated EEG forward model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.Lpos  = m_g.Lpos;
theta.L     = m_g.L;
theta.J     = m_g.J;

% evaluate constants
nd          = size(theta.L,2)               ; % number of dipoles
ne          = size(m_g.sens.chanpos,1)      ; % number of electrodes
dp          = 1e-3*theta.Lpos               ; % dipole coordinates adjusted for fieldtrip

% compute canonical dipole lead-field using Fieldtrip
L_can  = NaN(ne,3,nd);
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end

% rescale lead-field according to SPM
L_can = L_can*(10^1)^-round(log10(max(max(abs(L_can))))/8)/1);

% evaluate channel predictions based on dipole moments
L_dip = NaN(ne,nd);
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*theta.L(:,i);
end

% evaluate g(\theta_g)
g_theta_g = [];
for i = 1:nd
    g_theta_g = [g_theta_g kron(theta.J,L_dip(:,i))];
end
end

function [vFE]          = vFE(m_theta,s_theta,ms_sigsqr,vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : p x 1 variational expectation for theta
%       s_theta    : p x p variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters   for sigma^2
%       vfearg     : additional argument structure with fields
%           .y                  : n x 1 array    - data
%           .h                  : string         - function handle
%           .mu_theta           : scalar prior expectation for theta
%           .sig_theta          : scalar prior variance for theta
%           .mu_sigsqr          : scalar prior scale parameter for sigma^2
%           .si_sigsqr          : scalar prior shape parameter for sigma^2
%           .theta              : true, but unknown, parameter structure
%   Outputs
%       vFE                 : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
m_f             = vfearg.m_f;
m_g             = vfearg.m_g;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;
theta           = vfearg.theta;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta) and J^h(m_theta)
h_m_theta       = erp_h(m_f,m_g,spm_unvec(m_theta, theta));
Jh_m_theta      = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g,spm_unvec(m_theta, theta)),m_theta,1)));

% evaluation of the variational free energy value
T1              = - (n/2)*log(2*pi)             - (n/2)*m_sigsqr            - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2              = - (p/2)*log(2*pi)             - (1/2)*log(det(sig_theta))  - (1/2)*((m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
T3              = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                     - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T4              =  (1/2)*log(det(s_theta))    + (p/2)*log(2*pi*exp(1));
T5              =  (1/2)                      + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                         ;

% evaluate the negative free energy
vFE             = -(T1 + T2 + T3 + T4 + T5);


end

function [t_k]          = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsqr, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k         : p x 1     array of function argument
%       s_theta     : p x p     array of additional vFE arguments
%       ms_sigsqr   : 2 x 1     array of additional vFE arguments
%       vfearg      : additional input for vFE
%       df_x_k      :p_f x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
```

```
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(x_k, s_theta, ms_sigsqr, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(x_k + t_k*p_k, s_theta, ms_sigsqr, vfearg);

% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(x_k+t_k*p_k, s_theta, ms_sigsqr, vfearg);
end
end

function [t_k]          = backtrack_vFE_ms_sigsqr(m_theta, s_theta, x_k, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       m_theta     : p x 1   array of additional vFE arguments
%       s_theta     : p x p array of additional vFE arguments
%       x_k         : 2 x 1     array of function arguments
%       vfearg      : additional input for vFE
%       df_x_k      : 2 x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(m_theta, s_theta, x_k, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(m_theta, s_theta, x_k + t_k*p_k, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)|| x_k(2) + t_k*p_k(2) < 0;
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(m_theta, s_theta, x_k+t_k*p_k, vfearg);
end
end
```

PDDE_8.m – ERP-DCM model recovery (Figure 12)

```
function pdde_8_R1

% This function implements a basic Bayesian model selection example in the
% context of the estimation of latent input connectivity in a basic delay
% differential equation ERP model. Specifically, in a 2 x 2 design, ERP
% activity is simulated in a two-source model based on
%
%   (1) Input to source  1 only
%   (2) Input to sources 1 and 2
%
% The generated data are analyzed with probabilistic models allowing for
% estimation of
%
%   (1) Input to source  1 only
%   (2) Input to sources 1 and 2
%
% by means of tight and loose prior variances for prior expectations of
% zero. The resulting log marginal likelihood approximations are assessed
% for 2^4 realizations of the approach.
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
clc
close all

% set the random number generator to a random state
rng('shuffle')

% invoke Fieldtrip
addpath(genpath(fullfile(pwd, 'Fieldtrip')))

% simulation parameters
nsim                = 2^4;                                              % number of model realizations per generative model
max_i               = 5;                                                % maximal number of fixed-form VB Newton algorithm
max_j               = 8;                                                % maximal number of Newton iterations for m_theta optimization
max_k               = 8;                                                % maximal number of Newton iterations for (m_sigsqr, s_sigssqr) optimization
F_all               = NaN(4,max_i, nsim);                               % free energy array for simulation scenarios

% -------------------------------------------------------------------------
%                           Model Formulation
% -------------------------------------------------------------------------

% neural model structure and parameters
% -------------------------------------------------------------------------
% structural aspects and fixed parameters
m_f.f               = @spm_fx_erp;                                      % neural evolution function
m_f.ns              = 200;                                              % number of time bins
m_f.dt              = 0.001;                                            % time bin length (sec)
m_f.pst             = (0:m_f.ns-1)*m_f.dt;                              % peristimulus time (msec)
m_f.nt              = length(m_f.pst);                                  % number of time points
m_f.n               = 2;                                                % number of neural masses
m_f.m               = 9;                                                % number of neural states per neural mass
m_f.ons             = -10;                                              % system input onset (ms)
m_f.dur             = 1;                                                % duration (width) of the input function
m_f.x               = sparse(m_f.n,m_f.m);                              % initial condition
m_f.R               = [ 0.16 -0.22];                                    % input function parameters rho_1 and rho_2
m_f.S               = [-9e-4  0.08];                                    % static nonlinearity (activation function) parameters
m_f.T               = [ 0.16  0.13;  0.07 -0.82];                       % source-specific excitatory and inhibitory time constants
m_f.G               = [-0.11  0.45;  0.02 -0.63];                       % source-specific excitatory and inhibitory receotor densities
m_f.H               = [ 0.02 -0.07   0.18 -0.17];                       % source-independent intrinsic connectivity parameters
m_f.D               = [ 0    -0.52; -0.06  0   ];                       % between source delay parameters
m_f.A{1}            = [0 0; 1 0];                                       % forward  connectivity
m_f.A{2}            = [0 1; 0 0];                                       % backward connectivity
m_f.A{3}            = zeros(2,2);                                       % lateral connectivity
m_f.C               = [NaN NaN]';                                       % input connectivity

% forward model structure and parameters
% -------------------------------------------------------------------------
% load an existing dipole model
load(fullfile(cd, 'Data/SIM', 'dip_model_struct.mat'))
m_g.vol             = dip_model_struct.vol;                             % forward model specification
m_g.sens            = dip_model_struct.sens ;                           % sensor space specification
m_g.elab            = dip_model_struct.sens.label;                      % electrode labels
m_g.ne              = numel(m_g.elab);                                  % number of electrodes
m_g.X0              = zeros(m_f.ns*m_g.ne, 3*m_g.ne);                   % confound design matrix
m_g.Lpos            = [42 -31 58; 54 -22 18]';                          % dipole coordinate specifications
m_g.L               = [0.02 -0.05 -0.04; 0.17 -0.50 -0.17]';            % dipole moment specification
m_g.J               = [-0.1 0 0 0 0 0 -0.46 0 1];                       % state weighting specification

% likelihood model parameters
% -------------------------------------------------------------------------
n                   = m_f.nt*m_g.ne;                                    % number of data points

% prior formulation
% -------------------------------------------------------------------------
mu_theta            = [0 0]';                                           % prior expectation parameter setting theta
mu_si_sigsqr        = [4.6;1];                                          % prior shape and scale parameter setting \sigma^2

% fixed form variational Bayes Newton algorithm parameters
% -------------------------------------------------------------------------
delta               = sqrt(eps);                                        % Hessian modification constant
c                   = 1e-4;                                             % backtracking constant c
rho                 = 0.9;                                              % backtracking constant rho
vareps              = 1e0;                                              % gradient norm convergence criterion
vardelta            = 1e-3;                                             % parameter norm change convergence criterion

% -------------------------------------------------------------------------
%                       Cycle over  simulations
% -------------------------------------------------------------------------
for s = 1:nsim

    % free energy array index
    idx = 1;


    % ---------------------------------------------------------------------
    %                   Cycle over generative models
    % ---------------------------------------------------------------------
    for mgen = 1:2

        % true, but unknown, input connectivity
        switch mgen
            case 1
                theta  = [1 0]';
            case 2
                theta = [1 1]';
        end

        % -----------------------------------------------------------------
        %                       Model Realization
        % -----------------------------------------------------------------
        sigsqr              = 1e-1;                                         % true, but unknown, data variance
        p                   = length(theta);                                % number of parameters
        h_theta_fg          = erp_h(m_f,m_g,theta);                         % data expectation
        y                   = mvnrnd(h_theta_fg, sigsqr*eye(n))';           % data sampling

        % -----------------------------------------------------------------
        %                     Cycle over inversion models
        % -----------------------------------------------------------------
        % prior variance parameter setting theta
        for minv = 1:2
            switch minv
                case 1
                    sig_theta   = [1e3 0; 0 1e-3];
                case 2
                    sig_theta   = [1e3 0; 0 1e3];
            end
            % -------------------------------------------------------------
            %                         VBNL iteration 0
            % -------------------------------------------------------------

            % initialization of the variational parameters to prior parameters
            m_theta             = mu_theta;
            s_theta             = sig_theta;
            ms_sigsqr           = mu_si_sigsqr;
            vfearg.y            = y;
            vfearg.m_f          = m_f;
            vfearg.m_g          = m_g;
            vfearg.mu_theta     = mu_theta;
            vfearg.sig_theta    = sig_theta;
            vfearg.mu_sigsqr    = mu_si_sigsqr(1);
            vfearg.si_sigsqr    = mu_si_sigsqr(2);
            vfearg.theta        = theta;

            % parameter iteration arrays
            s_theta_i           = NaN(p,p,max_i-1);
            F_s_theta_i         = NaN(1,max_i-1);
            m_theta_j           = NaN(p,max_j,max_i-1);
            F_m_theta_j         = NaN(max_j,max_i-1);
            ms_sigsqr_k         = NaN(2,max_k,max_i-1);
            F_ms_sigsqr_k       = NaN(max_k,max_i-1);

            % initialization and initial evaluation of the negative free energy
            F                   = NaN(1,max_i-1);
            F(1)                = vFE(m_theta, s_theta, ms_sigsqr, vfearg);

            % -------------------------------------------------------------
            %                         VBNL iterations 1,2,3,...
            % -------------------------------------------------------------
            for i = 2:max_i

                % User update
                % ---------------------------------------------------------
                fprintf('Fixed-form VB Newton Iteration %4.0f ... \n', i-1)

                % Analytical update for theta_f variational variance parameter
                % ---------------------------------------------------------
                % evaluate the variational variance update equation for the current choice of m_theta
                fprintf('   Estimating q(m_theta) variance ... ')
                Jh_m_theta          = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g, spm_unvec(m_theta, theta)),m_theta,1)));
                s_theta             = spm_inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (spm_inv(sig_theta))));

                % save iterand and objective function
                s_theta_i(:,:,i-1)  = s_theta;
                F_s_theta_i(i-1)    =  vFE(m_theta,s_theta, ms_sigsqr, vfearg);
```

```
                % inform user
                fprintf('finished. \n')

                % Newton line search for theta_f variational expectation parameter
                % -------------------------------------------------------------
                fprintf('   Estimating q(m_theta) expectation ... ')

                % save iterand and objective function
                m_theta_j(:,1,i-1)  = m_theta;
                F_m_theta_j(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

                % numerical gradient and Hessian of the negative free energy
                dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
                ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

                % Newton iterations 1,2,3,...
                if norm(dF_m_theta) >= vareps
                    for j = 2:max_j

                        % gradient search direction
                        p_j = -ddF_m_theta\dF_m_theta;

                        % Hessian modification if p_j is not a descent direction
                        if ~(p_j'*dF_m_theta < 0)

                            % diagonal Hessian modification based on spectral decomposition
                            ddF_m_theta = ddF_m_theta + max(0,(delta - min(eig(ddF_m_theta))))*eye(p);

                            % re-evaluate Newton search direction
                            p_j = -ddF_m_theta\dF_m_theta;
                        end

                        % backtracking evaluation of the step length
                        t_j                 = backtrack_vFE_m_theta(m_theta, s_theta, ms_sigsqr, vfearg, dF_m_theta, p_j,c,rho);

                        % perform parameter update
                        m_theta             = m_theta + t_j*p_j;

                        % numerical gradient and Hessian of the negative free energy
                        dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
                        ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

                        % record updates
                        m_theta_j(:,j,i-1)    = m_theta;
                        F_m_theta_j(j,i-1)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

                        if norm(dF_m_theta) < vareps || norm(m_theta_j(:,j,i-1) - m_theta_j(:,j-1,i-1)) < vardelta
                            break
                        end
                    end
                end

                % inform user
                fprintf('finished.\n')

                % Newton line search for sigma^2 variational parameters
                % -------------------------------------------------------------
                fprintf('   Estimating q(sigma^2) parameters  ... ')

                % save iterand and objective function
                ms_sigsqr_k(:,1,i-1)  = ms_sigsqr;
                F_ms_sigsqr_k(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

                % evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
                dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
                ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

                % Newton iterations 1,2,3,...
                if norm(dF_ms_sigsqr) > vareps
                    for k = 2:max_k

                        % Newton search direction
                        p_k = -ddF_ms_sigsqr\dF_ms_sigsqr;

                        % Hessian modification if p_j is not a descent direction
                        if ~(p_k'*dF_ms_sigsqr < 0)

                            % diagonal Hessian modification based on spectral decomposition
                            ddF_ms_sigsqr = ddF_ms_sigsqr + max(0,(delta - min(eig(ddF_ms_sigsqr))))*eye(2);

                            % re-evaluate Newton search direction
                            p_k  = -ddF_ms_sigsqr\dF_ms_sigsqr;
                        end

                        % backtracking evaluation of the step length
                        t_k = backtrack_vFE_ms_sigsqr(m_theta, s_theta, ms_sigsqr, vfearg, dF_ms_sigsqr, p_k,c,rho);

                        % perform Newton update
                        ms_sigsqr           = ms_sigsqr + t_k*p_k;

                        % update remaining entries in free energy arrays
                        dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
                        ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

                        % record updates
                        ms_sigsqr_k(:,k,i-1)  = ms_sigsqr;
                        F_ms_sigsqr_k(k,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

                        if norm(dF_ms_sigsqr) < vareps || norm(ms_sigsqr_k(:,k,i-1) - ms_sigsqr_k(:,k-1,i-1)) < vardelta
                            break
                        end
                    end
                end

                % inform user
                fprintf('finished. \n')

                % free energy update
                % -------------------------------------------------------------
                F(i) = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

                % user update
                % -------------------------------------------------------------
                fprintf('vFE %6.3f  dvFE %6.3f \n', F(i), F(i) - F(i-1))
            end

            % save free energy evolution
            % -------------------------------------------------------------
            F_all(idx, :, s) = F;
            save([cd '\data\F_all.mat'], 'F_all')
            idx = idx + 1;
        end
    end
end

% -------------------------------------------------------------------------
%                               Visualization
% -------------------------------------------------------------------------
fig = figure;
set(fig, 'Color', [1 1 1])

% input architecture/function u
% -------------------------------------------------------------------------
[f_theta_f,u]       = erp_f(m_f,theta);
subplot(2,3,1)
plot(m_f.pst,u)
xlim([m_f.pst(1) m_f.pst(end)])
title('u', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

subplot(2,3,4)
plot(m_f.pst,u)
xlim([m_f.pst(1) m_f.pst(end)])
title('u', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('a.u.', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% evaluate log model evidence approximation
% -------------------------------------------------------------------------
% serial generative model
subplot(2,3,2)
hold on
bar(1:2, -mean(F_all(1:2,end,:),3),.6, 'FaceColor', [.8 .8 .8])
for i = 1:nsim
    plot(1:2, -F_all(1:2,end,i), 'k^-', 'MarkerFaceColor', 'w', 'MarkerSize', 4)
end
ylim([-3600 -3300])
xlim([0 3])
title('Serial Architecture', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
set(gca, 'FontName', 'Calibri Light', 'xtick', 1:2, 'xticklabel', {'M_1', 'M_2'}, 'FontSize', 26)

% parallel generative model
subplot(2,3,5)
hold on
bar(1:2, -mean(F_all(3:4,end,:),3),.6, 'FaceColor', [.8 .8 .8])
for i = 1:nsim
    plot(1:2, -F_all(3:4,end,i), 'k^-', 'MarkerFaceColor', 'w', 'MarkerSize', 4)
end
xlim([0 3])
ylim([-4200 -3100])
title('Parallel Architecture', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
set(gca, 'FontName', 'Calibri Light', 'xtick', 1:2, 'xticklabel', {'M_1', 'M_2'}, 'FontSize', 26)


% evaluate convergence
% -------------------------------------------------------------------------
% serial generative model
subplot(2,3,3)
hold on
plot(-mean(F_all(1,2:end,:),3), 'ko-', 'MarkerFaceColor', 'k')
plot(-mean(F_all(2,2:end,:),3), 'o-', 'Color', [.6 .6 .6],'MarkerFaceColor', [.6 .6 .6])
title('F(m_\theta^{(i)}, \Sigma_\theta^{(i)}, m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)})', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 22)
xlabel('i', 'FontSize', 24, 'FontName', 'Calibri Light')
legend('M_1', 'M_2', 'Location', 'SouthEast')
set(gca, 'FontName', 'Calibri Light', 'FontSize', 24)
xlim([0 5])

subplot(2,3,6)
hold on
plot(-mean(F_all(4,2:end,:),3), 'ko-', 'MarkerFaceColor', 'k')
```

```
plot(-mean(F_all(3,2:end,:),3), 'o-', 'Color', [.6 .6 .6],'MarkerFaceColor', [.6 .6 .6])
xlabel('i', 'FontSize', 24, 'FontName', 'Calibri Light')
title('F(m_\theta^{(i)}, \Sigma_\theta^{(i)}, m_{\sigma^2}^{(i)}, s_{\sigma^2}^{(i)})', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 22)
legend('M_1', 'M_1', 'Location', 'SouthEast')
set(gca, 'FontName', 'Calibri Light', 'FontSize', 24)
xlim([0 5])

end

% ------------------------------------------------------------------------
%                           Subfunctions
% ------------------------------------------------------------------------
function [h_theta_fg]   = erp_h(m_f,m_g,theta)

% This function evaluates the expectation parameter generating function
%
% h : (theta_f,theta_g) |-> h(theta_f,theta_g) := vec(g(theta_g)*f(theta_f)
%
% of the delay differential equation model for ERPs
%
%       Inputs
%           m_f         : latent neural model structure
%           m_g         : forward model structure
%           theta       : model parameters
%
%       Outputs
%           h_theta_fg  : evaluated function
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluate component functions f and g
[f_theta_f, u]  = erp_f(m_f,theta);
g_theta_g       = erp_g(m_g);

% evaluate concatenated function h
h_theta_fg      = 1e4*spm_vec(g_theta_g*f_theta_f');

end

function [f_theta_f,u]  = erp_f(m_f,vartheta)

% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_gen_erp.m of
% the SPM12 distribution. Based on a given parameter setting this function
% generates an input time-course and  integrates the resulting delay
% differential equation system.
%
%       Inputs
%           m_f         : latent neural model structure and fixed parameters
%           theta       : free latent neural model parameter
%
%       Outputs
%           f_theta_f   : evaluated latent neural model
%           u           : evaluated system input function
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% constant parameters
theta.R             = m_f.R;
theta.S             = m_f.S;
theta.T             = m_f.T;
theta.G             = m_f.G;
theta.H             = m_f.H;
theta.D             = m_f.D;
theta.A{1}          = m_f.A{1};
theta.A{2}          = m_f.A{2};
theta.A{3}          = m_f.A{3};
theta.C             = m_f.C;

% free parameters
theta.C(1)          = vartheta(1);
theta.C(2)          = vartheta(2);

% evaluate peri-stimulus time input function
t                   = ((1:m_f.ns)*m_f.dt)*1000;
delay               = m_f.ons(1)+128*theta.R(1,1);
scale               = m_f.dur(1)*exp(theta.R(1,2));
u                   = 32*exp(-(t - delay).^2/(2*scale^2))';

% integrate systen
x                   = m_f.x                                             ; % initial condition
f                   = m_f.f                                             ; % function handle @spm_fx_erp
dt                  = m_f.dt                                            ; % integration time bin
[fx,dfdx,D]         = f(x,u(1),theta,m_f)                               ; % dx(t)/dt and Jacobian df/dx and check for delay operator
p                   = max(abs(real(eig(full(dfdx)))))                   ; %
N                   = ceil(max(1, dt*p*2))                              ; %
n                   = spm_length(x)                                     ; %
Q                   = (spm_expm(dt*D*dfdx/N) - speye(n,n))*spm_inv(dfdx)    ; % initialize state
v                   = spm_vec(x)                                        ; % initialize state time-course
y                   = NaN(length(v), length(u))

% cycle over time-steps
for i = 1:size(u,1)

    % update dx = (expm(dt*J) - I)*inv(J)*f(x,u)
    for j = 1:N
        v = v + Q*f(v,u(i),theta,m_f);
    end
    y(:,i) = v;

end

% transpose
f_theta_f       = y';

end

function [f,J,Q]        = spm_fx_erp(x,u,P,M)

% state equations for a neural mass model of erps
% FORMAT [f,J,D] = spm_fx_erp(x,u,P,M)
% FORMAT [f,J]   = spm_fx_erp(x,u,P,M)
% FORMAT [f]     = spm_fx_erp(x,u,P,M)
% x     - state vector
%   x(:,1) - voltage (spiny stellate cells)
%   x(:,2) - voltage (pyramidal cells) +ve
%   x(:,3) - voltage (pyramidal cells) -ve
%   x(:,4) - current (spiny stellate cells)    depolarizing
%   x(:,5) - current (pyramidal cells)         depolarizing
%   x(:,6) - current (pyramidal cells)         hyperpolarizing
%   x(:,7) - voltage (inhibitory interneurons)
%   x(:,8) - current (inhibitory interneurons) depolarizing
%   x(:,9) - voltage (pyramidal cells)
%
% f       - dx(t)/dt  = f(x(t))
% J       - df(t)/dx(t)
% D       - delay operator dx(t)/dt = f(x(t - d))
%                                   = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
%  M.pF.E = [32 16 4];          % extrinsic rates (forward, backward, lateral)
%  M.pF.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2 g3, g4)
%  M.pF.D = [2 16];             % propogation delays (intrinsic, extrinsic)
%  M.pF.G = [4 32];             % receptor densities (excitatory, inhibitory)
%  M.pF.T = [8 16];             % synaptic constants (excitatory, inhibitory)
%  M.pF.R = [1 1/2];            % parameter of static nonlinearity
%
% _______________________________________________________________________
% David O, Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neuronal dynamics. NeuroImage 20: 1743-1755
% _______________________________________________________________________
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging

% Karl Friston
% $Id: spm_fx_erp.m 5369 2013-03-28 20:09:27Z karl $

% get dimensions and configure state variables
%--------------------------------------------------------------------------
n       = length(P.A{1});           % number of sources
x       = spm_unvec(x,M.x);         % neuronal states

% [default] fixed parameters
%--------------------------------------------------------------------------
E       = [1 1/2 1/8]*32;                   % extrinsic rates (forward, backward, lateral)
G       = [1 4/5 1/4 1/4]*128;              % intrinsic rates (g1 g2 g3 g4)
D       = [2 16];                           % propagation delays (intrinsic, extrinsic)
H       = [4 32];                           % receptor densities (excitatory, inhibitory)
T       = [8 16];                           % synaptic constants (excitatory, inhibitory)
R       = [2 1]/3;                          % parameters of static nonlinearity

% test for free parameters on intrinsic connections
%--------------------------------------------------------------------------
G       = G.*exp(P.H);
G       = ones(n,1)*G;

% no exponential transforms to foster parameter identifiability
% -------------------------------------------------------------------------
A{1}    = P.A{1}*E(1);
A{2}    = P.A{2}*E(2);
A{3}    = P.A{3}*E(3);
C       = P.C;

% intrinsic connectivity and parameters
%--------------------------------------------------------------------------
Te      = T(1)/1000*exp(P.T(:,1));        % excitatory time constants
Ti      = T(2)/1000*exp(P.T(:,2));        % inhibitory time constants
He      = H(1)*exp(P.G(:,1));             % excitatory receptor density
Hi      = H(2)*exp(P.G(:,2));             % inhibitory receptor density

% pre-synaptic inputs: s(V)
%--------------------------------------------------------------------------
S       = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
%--------------------------------------------------------------------------
U       = C*u(:)*2;

% State: f(x)
%--------------------------------------------------------------------------
f(:,1) = x(:,4)                                                         ; % voltage change (spiny stellate cells)
f(:,2) = x(:,5)                                                         ; % positive voltage change (pyramidal cells) +ve
f(:,3) = x(:,6)                                                         ; % negative voltage change (pyramidal cells) -ve
```

```
f(:,4) = (He.*((A{1} + A{3})*S(:,9) + G(:,1).*S(:,9) + U) - 2*x(:,4) - x(:,1)./Te)./Te      ; % current change (spiny stellate cells)    depolarizing
f(:,5) = (He.*((A{2} + A{3})*S(:,9) + G(:,2).*S(:,1)) - 2*x(:,5) - x(:,2)./Te)./Te         ; % current change (pyramidal cells)          depolarizing
f(:,6) = (Hi.*G(:,4).*S(:,7) - 2*x(:,6) - x(:,3)./Ti)./Ti                                  ; % current change (pyramidal cells)          hyperpolarizing
f(:,7) = x(:,8)                                                                            ; % voltage change (inhibitory interneurons)
f(:,8) = (He.*((A{2} + A{3})*S(:,9) + G(:,3).*S(:,9)) - 2*x(:,8) - x(:,7)./Te)./Te         ; % current change(inhibitory interneurons) depolarizing
f(:,9) = x(:,5) - x(:,6)                                                                   ; % voltage (pyramidal cells)


% vectorize
f       = spm_vec(f);

% avoid infinite recursion of spm_diff
if nargout < 2
    return
end

% Jacobian
%==========================================================================
J  = spm_diff(M.f,x,u,P,M,1);

% delays
%==========================================================================
% Delay differential equations can be integrated efficiently (but
% approximately) by absorbing the delay operator into the Jacobian
%
%    dx(t)/dt      = f(x(t - d))
%                  = Q(d)f(x(t))
%
%    J(d)          = Q(d)df/dx
%--------------------------------------------------------------------------
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,n)).*De;
Di = (1 - speye(9,9)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D  = Di + De;

% Implement: dx(t)/dt = f(x(t - d)) = inv(1 + D.*dfdx)*f(x(t))
%                     = Q*f = Q*J*x(t)
%--------------------------------------------------------------------------
Q  = spm_inv(speye(length(J)) + D.*J);

end

function [g_theta_g]    = erp_g(m_g)

% This function evaluates the EEG forward model of the delay differential
% equation model for ERPs based on a structural formulation of the forward
% model and parameter setting
%
%   Inputs
%          m_g         : EEG forward model structure and fixed parameters
%          theta       : EEG lead-field free parameters

%    Outputs
%          g_theta_g   : evaluated EEG forward model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.Lpos  = m_g.Lpos;
theta.L     = m_g.L;
theta.J     = m_g.J;

% evaluate constants
nd          = size(theta.L,2)                 ; % number of dipoles
ne          = size(m_g.sens.chanpos,1)        ; % number of electrodes
dp          = 1e-3*theta.Lpos                 ; % dipole coordinates adjusted for fieldtrip

% compute canonical dipole lead-field using Fieldtrip
L_can  = NaN(ne,3,nd);
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end

% rescale lead-field according to SPM
L_can = L_can*(10^1)^-round(log10(max(max(max(abs(L_can))))/8)/1);

% evaluate channel predictions based on dipole moments
L_dip = NaN(ne,nd);
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*theta.L(:,i);
end

% evaluate g(\theta_g)
g_theta_g     = kron(theta.J,L_dip);


end

function [vFE]          = vFE(m_theta,s_theta,ms_sigsqr,vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : p x 1 variational expectation for theta
%       s_theta    : p x p variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters  for sigma^2
%       vfearg     : additional argument structure with fields
%           .y              : n x 1 array   - data
%           .h              : string        - function handle
%           .mu_theta       : scalar prior expectation for theta
%           .sig_theta      : scalar prior variance for theta
%           .mu_sigsqr      : scalar prior scale parameter for sigma^2
%           .si_sigsqr      : scalar prior shape parameter for sigma^2
%           .theta          : true, but unknown, parameter structure
%   Outputs
%           vFE             : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
m_f             = vfearg.m_f;
m_g             = vfearg.m_g;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;
theta           = vfearg.theta;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evaluate h(m_theta) and J^h(m_theta)
h_m_theta       = erp_h(m_f,m_g,spm_unvec(m_theta, theta));
Jh_m_theta      = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g,spm_unvec(m_theta, theta)),m_theta,1)));


% evaluation of the variational free energy value
T1              = -(n/2)*log(2*pi)           - (n/2)*m_sigsqr              - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2              = -(p/2)*log(2*pi)           - (1/2)*log(det(sig_theta))  - (1/2)*((m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
T3              = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                   - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T4              =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
T5              =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                    ;

% evaluate the negative free energy
vFE             = -(T1 + T2 + T3 + T4 + T5);


end

function [t_k]          = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsqr, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k         : p x 1     array of function argument
%       s_theta     : p x p     array of additional vFE arguments
%       ms_sigsqr   : 2 x 1     array of additional vFE arguments
%       vfearg      : additional input for vFE
%       df_x_k       :p_f x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(x_k, s_theta, ms_sigsqr, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(x_k + t_k*p_k, s_theta, ms_sigsqr, vfearg);

% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(x_k+t_k*p_k, s_theta, ms_sigsqr, vfearg);
end
end

function [t_k]          = backtrack_vFE_ms_sigsqr(m_theta, s_theta, x_k, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       m_theta     : p x 1   array of additional vFE arguments
%       s_theta     : p x p array of additional vFE arguments
%       x_k         : 2 x 1     array of function arguments
%       vfearg      : additional input for vFE
```

```
%       df_x_k      : 2 x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% ------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(m_theta, s_theta, x_k, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(m_theta, s_theta, x_k + t_k*p_k, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)|| x_k(2) + t_k*p_k(2) < 0;
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(m_theta, s_theta, x_k+t_k*p_k, vfearg);
end
end
```

PDDE_9.m – ERP-DCM experimental application (Figure 13)

```
function pdde_9_R1

% This function implements a basic application of the PDDE framework to a
% somatosensory evoked potential recording. Specifically, it estimates the
% connectivity structure of a two-source neurophysiologial model.
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
clc
close all

% -------------------------------------------------------------------------
%                           EEG Data Loading
% -------------------------------------------------------------------------
spm_dir                 = 'D:\Users\Dirk\Projects\THEORY_Delay_Differentials_ERP\Revision\Matlab\SPM12';
fname                   = fullfile(pwd,'Data\EEG','Grand_Mean_all_all.mat');% source directory
twidx                   = 21:41;                                            % peristimulus time indices of a 0-100 ms time window
sf                      = 1e3;                                              % data scale factor to avoid numerical overflow
addpath(genpath(spm_dir));                                                  % invoke SPM
D                       = spm_eeg_load(fname);
y                       = sf*spm_vec(D(:,twidx,:));
pst                     = D.time(twidx);                                    % peristimulus time
dt                      = 1/D.fsample;                                      % time bin length
rmpath(genpath(spm_dir));

% -------------------------------------------------------------------------
%                           Model Formulation
% -------------------------------------------------------------------------

% neural model structure and parameters
% -------------------------------------------------------------------------
% structural aspects and fixed parameters
m_f.f                   = @spm_fx_erp;                                      % neural evolution function
m_f.ns                  = length(pst);                                      % number of time bins
m_f.dt                  = dt;                                               % time bin length (sec)
m_f.pst                 = pst;                                              % peristimulus time (msec)
m_f.nt                  = length(m_f.pst);                                  % number of time points
m_f.n                   = 2;                                                % number of neural masses
m_f.m                   = 9;                                                % number of neural states per neural mass
m_f.ons                 = 30;                                               % system input onset (ms)
m_f.dur                 = 10;                                               % duration (width) of the input function
m_f.x                   = sparse(m_f.n,m_f.m);                              % initial condition
m_f.R                   = zeros(1,2);                                       % input function parameters rho_1 and rho_2
m_f.S                   = zeros(1,2);                                       % static nonlinearity (activation function) parameters
m_f.T                   = zeros(2,2);                                       % source-specific excitatory and inhibitory time constants
m_f.G                   = zeros(2,2);                                       % source-specific excitatory and inhibitory receptor densities
m_f.H                   = zeros(1,4);                                       % source-independent intrinsic connectivity parameters
m_f.D                   = zeros(2,2);                                       % between source delay parameters
m_f.A{1}                = [0 0; 1 0];                                       % forward  connectivity
m_f.A{2}                = [0 1;0 0];                                        % backward connectivity
m_f.A{3}                = zeros(2,2);                                       % lateral connectivity
m_f.C                   = [NaN NaN]';                                       % input connectivity


% forward model structure and parameters
% -------------------------------------------------------------------------
load(fullfile(cd, 'Data/SIM', 'dip_model_struct.mat'))
m_g.vol                 = dip_model_struct.vol;                             % forward model specification
m_g.sens                = dip_model_struct.sens ;                           % sensor space specification
m_g.elab                = dip_model_struct.sens.label;                      % electrode labels
m_g.ne                  = numel(m_g.elab);                                  % number of electrodes
m_g.X0                  = zeros(m_f.ns*m_g.ne,3*m_g.ne);                    % confound design matrix
m_g.Lpos                = [42 54; -31 -22; 58 18];                          % dipole coordinate specifications
m_g.L                   = [10.64 20.67; -5.65  -24.53; 5.38  -3.62];        % dipole moment specification
m_g.J                   = [0.291 0 0 0 0 0 0.0351 0 1.0000];                % state weighting specification

% likelihood model parameters
% -------------------------------------------------------------------------
p                       = 2;                                                % number of parameters to estimate

% -------------------------------------------------------------------------
%                           VBNL initialization
% -------------------------------------------------------------------------
% invoke Fieldtrip
addpath(genpath(fullfile(pwd, 'Fieldtrip')))
max_i                   = 12;                                                % maximal number of fixed-form VB Newton algorithm
max_j                   = 8;                                                % maximal number of Newton iterations for m_theta optimization
max_k                   = 8;                                                % maximal number of Newton iterations for (m_sigsqr, s_sigssqr) optimization
delta                   = sqrt(eps);                                        % Hessian modification constant
c                       = 1e-4;                                             % backtracking constant c
rho                     = 0.9;                                              % backtracking constant rho
vareps                  = 1e0;                                              % gradient norm convergence criterion
vardelta                = 1e-3;                                             % parameter norm change convergence criterion

% model-indepedent prior formulation
% -------------------------------------------------------------------------
mu_theta                = zeros(p,1);                                       % prior expectation parameter setting theta
mu_si_sigsqr            = [1e2;1e1];                                        % prior shape and scale parameter setting \sigma^2

% model evaluation arrays
% -------------------------------------------------------------------------
nm                      = 2;                                                % number of models to test
m_theta_m               = NaN(p,nm);                                        % theta variational expectations
s_theta_m               = NaN(p,p,nm);                                      % theta variational covariances
ms_sigsqr_m             = NaN(2,nm);                                        % sigma^2 variational parameters
F_m                     = NaN(nm,max_i);                                    % variational free energy evolutions

% -------------------------------------------------------------------------
%                           Cycle over probabilistic models
% -------------------------------------------------------------------------
for m = 1:nm

    % model-dependent prior covariance matrix
    switch m

        % loose prior on c_1, tight prior on c_2
        case 1
            sig_theta           = [1e6 0; 0 1e-6];                          % prior covariance parameter setting theta

        % loose prior on c_1 and c_2
        case 2
            sig_theta           = 1e6*eye(p);                               % prior covariance parameter setting theta
    end

    % -------------------------------------------------------------------------
    %                           VBNL iteration 0
    % -------------------------------------------------------------------------
    % initialization of the variational parameters to prior parameters
    theta                   = mu_theta;
    m_theta                 = mu_theta;
    s_theta                 = sig_theta;
    ms_sigsqr               = mu_si_sigsqr;
    vfearg.y                = y;
    vfearg.m_f              = m_f;
    vfearg.m_g              = m_g;
    vfearg.mu_theta         = mu_theta;
    vfearg.sig_theta        = sig_theta;
    vfearg.mu_sigsqr        = mu_si_sigsqr(1);
    vfearg.si_sigsqr        = mu_si_sigsqr(2);
    vfearg.theta            = theta;

    % parameter iteration arrays
    s_theta_i               = NaN(p,p,max_i-1);
    F_s_theta_i             = NaN(1,max_i-1);
    m_theta_j               = NaN(p,max_j,max_i-1);
    F_m_theta_j             = NaN(max_j,max_i-1);
    ms_sigsqr_k             = NaN(2,max_k,max_i-1);
    F_ms_sigsqr_k           = NaN(max_k,max_i-1);

    % initialization and initial evaluation of the negative free energy
    F                       = NaN(1,max_i-1);
    F(1)                    = vFE(m_theta, s_theta, ms_sigsqr, vfearg);

    % -------------------------------------------------------------------------
    %                           VBNL iterations 1,2,3,...
    % -------------------------------------------------------------------------
    for i = 2:max_i

        % User update
        % -----------------------------------------------------------------
        fprintf('Fixed-form VB Newton Iteration %4.0f ... \n', i-1)

        % Analytical update for theta_f variational variance parameter
        % -----------------------------------------------------------------
        % evaluate the variational variance update equation for the current choice of m_theta
        fprintf('   Estimating q(m_theta) variance ... ')
        Jh_m_theta          = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g, spm_unvec(m_theta, theta)),m_theta,1)));
        s_theta             = spm_inv((exp(-ms_sigsqr(1)+(1/2)*ms_sigsqr(2))*(Jh_m_theta'*Jh_m_theta) + (spm_inv(sig_theta))));

        % save iterand and objective function
        s_theta_i(:,:,i-1)  = s_theta;
        F_s_theta_i(i-1)    =  vFE(m_theta,s_theta, ms_sigsqr, vfearg);

        % inform user
        fprintf('finished. \n')

        % Newton line search for theta_f variational expectation parameter
        % -----------------------------------------------------------------
        fprintf('   Estimating q(m_theta) expectation ... ')

        % save iterand and objective function
        m_theta_j(:,1,i-1)  = m_theta;
        F_m_theta_j(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

        % numerical gradient and Hessian of the negative free energy
        dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
        ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

        % Newton iterations 1,2,3,...
        if norm(dF_m_theta) >= vareps
            for j = 2:max_j

                % gradient search direction
                p_j = -ddF_m_theta\dF_m_theta;

                % Hessian modification if p_j is not a descent direction
```

```
            if -(p_j'*dF_m_theta < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_m_theta = ddF_m_theta + max(0,(delta - min(eig(ddF_m_theta))))*eye(p);

                % re-evaluate Newton search direction
                p_j = -ddF_m_theta\dF_m_theta;
            end

            % backtracking evaluation of the step length
            t_j                 = backtrack_vFE_m_theta(m_theta, s_theta, ms_sigsqr, vfearg, dF_m_theta, p_j,c,rho);

            % perform parameter update
            m_theta             = m_theta + t_j*p_j;

            % numerical gradient and Hessian of the negative free energy
            dF_m_theta          = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,1)'));
            ddF_m_theta         = full(spm_cat(spm_diff(@(m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg),m_theta,[1 1])'));

            % record updates
            m_theta_j(:,j,i-1)    = m_theta;
            F_m_theta_j(j,i-1)    = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_m_theta) < vareps || norm(m_theta_j(:,j,i-1) - m_theta_j(:,j-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished.\n')

    % Newton line search for sigma^2 variational parameters
    % -------------------------------------------------------------------
    fprintf('   Estimating q(sigma^2) parameters  ... ')

    % save iterand and objective function
    ms_sigsqr_k(:,1,i-1)  = ms_sigsqr;
    F_ms_sigsqr_k(1,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
    dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
    ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

    % Newton iterations 1,2,3,...
    if norm(dF_ms_sigsqr) > vareps
        for k = 2:max_k

            % Newton search direction
            p_k = -ddF_ms_sigsqr\dF_ms_sigsqr;

            % Hessian modification if p_j is not a descent direction
            if -(p_k'*dF_ms_sigsqr < 0)

                % diagonal Hessian modification based on spectral decomposition
                ddF_ms_sigsqr = ddF_ms_sigsqr + max(0,(delta - min(eig(ddF_ms_sigsqr))))*eye(2);

                % re-evaluate Newton search direction
                p_k  = -ddF_ms_sigsqr\dF_ms_sigsqr;
            end

            % backtracking evaluation of the step length
            t_k = backtrack_vFE_ms_sigsqr(m_theta, s_theta, ms_sigsqr, vfearg, dF_ms_sigsqr, p_k,c,rho);

            % perform Newton update
            ms_sigsqr           = ms_sigsqr + t_k*p_k;

            % update remaining entries in free energy arrays
            dF_ms_sigsqr        = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr,1)'));
            ddF_ms_sigsqr       = full(spm_cat(spm_diff(@(ms_sigsqr) vFE(m_theta,s_theta, ms_sigsqr, vfearg), ms_sigsqr, [1 1])'));

            % record updates
            ms_sigsqr_k(:,k,i-1)  = ms_sigsqr;
            F_ms_sigsqr_k(k,i-1)  = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

            if norm(dF_ms_sigsqr) < vareps || norm(ms_sigsqr_k(:,k,i-1) - ms_sigsqr_k(:,k-1,i-1)) < vardelta
                break
            end
        end
    end

    % inform user
    fprintf('finished. \n')

    % free energy update
    % -------------------------------------------------------------------
    F(i) = vFE(m_theta,s_theta, ms_sigsqr, vfearg);

    % user update
    % -------------------------------------------------------------------
    fprintf('vFE %6.3f  dvFE %6.3f \n', F(i), F(i) - F(i-1))
end

% save model evaluation values
% -------------------------------------------------------------------
m_theta_m(:,m)      = m_theta;                                      % theta variational expectations
s_theta_m(:,:,m)    = s_theta;                                      % theta variational covariances
ms_sigsqr_m(:,m)    = ms_sigsqr;                                    % sigma^2 variational parameters
F_m(m,:)            = F;                                            % variational free energy evolutions

end

% -------------------------------------------------------------------
%                       Visualization
% -------------------------------------------------------------------

% Data
% -------------------------------------------------------------------
fig = figure;
set(fig,'Color',[1 1 1])
subplot(2,3,1)
y_eeg = spm_unvec(y, NaN(m_g.ne,m_f.nt))/sf;
imagesc(m_f.pst, 1:numel(m_g.elab),y_eeg, [-1 1])
xlim([m_f.pst(1) m_f.pst(end)])
cb = colorbar;
title('Y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel(cb, '\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('Electrodes', 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% MAP estimate based predicted data
% -------------------------------------------------------------------
% cycle over models
for m = 1:nm
    h_theta_map = erp_h(m_f,m_g, m_theta_m(:,m));                   % data expectation
    y_map = spm_unvec(h_theta_map, NaN(m_g.ne,m_f.nt))/sf;
    subplot(2,3,m+1)
    imagesc(m_f.pst, 1:numel(m_g.elab),y_map,[-1 1])
    xlim([m_f.pst(1) m_f.pst(end)])
    cb = colorbar;
    title(['M_' num2str(m) ' Y MAP'], 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
    ylabel(cb, '\muV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
    xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
    ylabel('Electrodes', 'FontName', 'Calibri Light', 'FontSize', 24)
    set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
end

% theta_f estimates
% -------------------------------------------------------------------
subplot(2,3,4)
hold on
res         = 2e2;
c_1_min     = -.01;
c_1_max     = .3;
c_1_res     = res;
c_1         = linspace(c_1_min, c_1_max, c_1_res);
c_2_min     = -.01;
c_2_max     = .4;
c_2_res     = res;
c_2         = linspace(c_2_min, c_2_max, c_2_res);

% posterior approximation q(c)
q_c = NaN(length(c_1),length(c_2));
for i = 1:length(c_1)
    for j = 1:length(c_2)
        q_c(i,j) = (mvnpdf(m_theta_m(:,1)',[c_1(i) c_2(j)],s_theta_m(:,:,1)) + mvnpdf(m_theta_m(:,2)',[c_1(i) c_2(j)],s_theta_m(:,:,2)))/2;
    end
end
imagesc(c_2,c_1,q_c)
set(gca, 'YDir', 'Normal')
text(m_theta_m(2,1)+.013,m_theta_m(1,1),'Q(C) M_1', 'Color', [1 1 1], 'FontSize', 22, 'FontName', 'Calibri Light')
text(m_theta_m(2,2)-0.13,m_theta_m(1,2),'q(C) M_2', 'Color', [1 1 1], 'FontSize', 22, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
xlim([c_2_min c_2_max])
ylim([c_1_min c_1_max])
xlabel('c_2', 'FontName', 'Calibri Light', 'FontSize', 24)
ylabel('c_1', 'FontName', 'Calibri Light', 'FontSize', 24, 'Rotation', 0)

% sigma^2 estimates
% -------------------------------------------------------------------
subplot(2,3,5)
hold on
sigsqr_x = linspace(1e4,5e4,1e5);
cols = {'b', 'r'};
for m = 1:nm
    plot(sigsqr_x, lognpdf(sigsqr_x, ms_sigsqr_m(1,m), sqrt(ms_sigsqr_m(2,m))), cols{m})
end
set(gca, 'FontSize', 24, 'FontName', 'Calibri Light')
legend('q(\sigma^2) M_1 ',  'q(\sigma^2) M_2', 'Location', 'NorthEast')
xlabel('\sigma^2', 'FontSize',24, 'FontName', 'Calibri Light')

% variational free energy
% -------------------------------------------------------------------
subplot(2,3,6)
bar(1:2, -F_m(:,end),.6, 'FaceColor', [.8 .8 .8])
ylim([-1.05e4  -.95e4])
xlim([0 3])
title('F', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
set(gca, 'FontName', 'Calibri Light', 'xtick', 1:2, 'xticklabel', {'M_1', 'M_2'}, 'FontSize', 26)

end
```

```
% ---------------------------------------------------------------------
%                           Subfunctions
% ---------------------------------------------------------------------
function [h_theta_fg]   = erp_h(m_f,m_g,vartheta)

% This function evaluates the expectation parameter generating function
%
% h : (theta_f,theta_g) |-> h(theta_f,theta_g) := vec(g(theta_g)*f(theta_f)
%
% of the delay differential equation model for ERPs
%
%       Inputs
%           m_f         : latent neural model structure
%           m_g         : forward model structure
%           theta       : model parameters
%
%       Outputs
%           h_theta_fg  : evaluated function
%
% Copyright (C) Dirk Ostwald
% ---------------------------------------------------------------------
% evaluate component functions f and g
[f_theta_f, u]  = erp_f(m_f, vartheta);
g_theta_g       = erp_g(m_g);

% evaluate concatenated function h
h_theta_fg      = spm_vec(g_theta_g*f_theta_f');

end

function [f_theta_f,u]  = erp_f(m_f,vartheta)

% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_gen_erp.m of
% the SPM12 distribution. Based on a given parameter setting this function
% generates an input time-course and  integrates the resulting delay
% differential equation system.
%
%       Inputs
%           m_f         : latent neural model structure and fixed parameters
%           theta       : free latent neural model parameter
%
%       Outputs
%           f_theta_f   : evaluated latent neural model
%           u           : evaluated system input function
%
% Copyright (C) Dirk Ostwald
% ---------------------------------------------------------------------
% constant parameters
theta.R             = m_f.R;
theta.S             = m_f.S;
theta.T             = m_f.T;
theta.G             = m_f.G;
theta.H             = m_f.H;
theta.D             = m_f.D;
theta.A{1}          = m_f.A{1};
theta.A{2}          = m_f.A{2};
theta.A{3}          = m_f.A{3};
theta.C             = m_f.C;

% free parameters
theta.C             = vartheta;

% evaluate peri-stimulus time input function
t                   = ((1:m_f.ns)*m_f.dt)*1000;
delay               = m_f.ons(1)+128*theta.R(1,1);
scale               = m_f.dur(1)*exp(theta.R(1,2));
u                   = 32*exp(-(t-delay).^2/(2*scale^2))';

% integrate systen
x                   = m_f.x                                                 ; % initial condition
f                   = m_f.f                                                 ; % function handle @spm_fx_erp
dt                  = m_f.dt                                                ; % integration time bin
[fx,dfdx,D]         = f(x,u(1),theta,m_f)                                   ; % dx(t)/dt and Jacobian df/dx and check for delay operator
p                   = max(abs(real(eig(full(dfdx)))))                       ; %
N                   = ceil(max(1, dt*p*2))                                  ; %
n                   = spm_length(x)                                         ; %
Q                   = (spm_expm(dt*D*dfdx/N) - speye(n,n))*spm_inv(dfdx)    ; %
v                   = spm_vec(x)                                            ; % initialize state
y                   = NaN(length(v), length(u))                             ; % initialize state time-course

% cycle over time-steps
for i = 1:size(u,1)

    % update dx = (expm(dt*J) - I)*inv(J)*f(x,u)
    for j = 1:N
        v = v + Q*f(v,u(i),theta,m_f);
    end
    y(:,i) = v;

end

% transpose
f_theta_f       = y';

end

function [f,J,Q]        = spm_fx_erp(x,u,P,M)

% state equations for a neural mass model of erps
% FORMAT [f,J,D] = spm_fx_erp(x,u,P,M)
% FORMAT [f,J]   = spm_fx_erp(x,u,P,M)
% FORMAT [f]     = spm_fx_erp(x,u,P,M)
% x      - state vector
%   x(:,1) - voltage (spiny stellate cells)
%   x(:,2) - voltage (pyramidal cells) +ve
%   x(:,3) - voltage (pyramidal cells) -ve
%   x(:,4) - current (spiny stellate cells)    depolarizing
%   x(:,5) - current (pyramidal cells)         depolarizing
%   x(:,6) - current (pyramidal cells)         hyperpolarizing
%   x(:,7) - voltage (inhibitory interneurons)
%   x(:,8) - current (inhibitory interneurons) depolarizing
%   x(:,9) - voltage (pyramidal cells)
%
% f        - dx(t)/dt  = f(x(t))
% J        - df(t)/dx(t)
% D        - delay operator dx(t)/dt = f(x(t - d))
%                                    = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
%  M.pF.E = [32 16 4];          % extrinsic rates (forward, backward, lateral)
%  M.pF.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2 g3, g4)
%  M.pF.D = [2 16];             % propogation delays (intrinsic, extrinsic)
%  M.pF.G = [4 32];             % receptor densities (excitatory, inhibitory)
%  M.pF.T = [8 16];             % synaptic constants (excitatory, inhibitory)
%  M.pF.R = [1 1/2];            % parameter of static nonlinearity
%
% _____________________________________________________________________
% David O, Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neuronal dynamics. NeuroImage 20: 1743-1755
% _____________________________________________________________________
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging

% Karl Friston
% $Id: spm_fx_erp.m 5369 2013-03-28 20:09:27Z karl $

% get dimensions and configure state variables
%--------------------------------------------------------------------------
n       = length(P.A{1});       % number of sources
x       = spm_unvec(x,M.x);     % neuronal states

% [default] fixed parameters
%--------------------------------------------------------------------------
E       = [1 1/2 1/8]'32;               % extrinsic rates (forward, backward, lateral)
G       = [1 4/5 1/4 1/4]*128;          % intrinsic rates (g1 g2 g3 g4)
D       = [2 16];                       % propagation delays (intrinsic, extrinsic)
H       = [4 32];                       % receptor densities (excitatory, inhibitory)
T       = [8 16];                       % synaptic constants (excitatory, inhibitory)
R       = [1 1/2]/3;                    % parameters of static nonlinearity

% test for free parameters on intrinsic connections
%--------------------------------------------------------------------------
G       = G.*exp(P.H);
G       = ones(n,1)*G;

% no exponential transforms to foster parameter identifiability
% -------------------------------------------------------------------------
A{1}    = P.A{1}*E(1);
A{2}    = P.A{2}*E(2);
A{3}    = P.A{3}*E(3);
C       = P.C;

% intrinsic connectivity and parameters
%--------------------------------------------------------------------------
Te      = T(1)/1000*exp(P.T(:,1));      % excitatory time constants
Ti      = T(2)/1000*exp(P.T(:,2));      % inhibitory time constants
He      = H(1)*exp(P.G(:,1));           % excitatory receptor density
Hi      = H(2)*exp(P.G(:,2));           % inhibitory receptor density

% pre-synaptic inputs: s(V)
%--------------------------------------------------------------------------
S       = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
%--------------------------------------------------------------------------
U       = C*u(:)*2;

% State: f(x)
%==========================================================================
f(:,1) = x(:,4)                                                             ; % voltage change (spiny stellate cells)
f(:,2) = x(:,5)                                                             ; % positive voltage change (pyramidal cells) +ve
f(:,3) = x(:,6)                                                             ; % negative voltage change (pyramidal cells) -ve
f(:,4) = (He.*((A{1} + A{3})*S(:,9) + G(:,1).*S(:,9) + U) - 2*x(:,4) - x(:,1)./Te)./Te      ; % current change (spiny stellate cells)    depolarizing
f(:,5) = (He.*((A{2} + A{3})*S(:,9) + G(:,2).*S(:,1)) - 2*x(:,5) - x(:,2)./Te)./Te          ; % current change (pyramidal cells)          depolarizing
f(:,6) = (Hi.*G(:,4).*S(:,7) - 2*x(:,6) - x(:,3)./Ti)./Ti                   ; % current change (pyramidal cells)          hyperpolarizing
f(:,7) = x(:,8)                                                             ; % voltage change (inhibitory interneurons)
f(:,8) = (He.*((A{2} + A{3})*S(:,9) + G(:,3).*S(:,9)) - 2*x(:,8) - x(:,7)./Te)./Te          ; % current change(inhibitory interneurons) depolarizing
f(:,9) = x(:,5) - x(:,6)                                                    ; % voltage (pyramidal cells)


% vectorize
f       = spm_vec(f);
```

```
% avoid infinite recursion of spm_diff
if nargout < 2
    return
end

% Jacobian
%==========================================================================
J  = spm_diff(M.f,x,u,P,M,1);

% delays
%==========================================================================
% Delay differential equations can be integrated efficiently (but
% approximately) by absorbing the delay operator into the Jacobian
%
%    dx(t)/dt     = f(x(t - d))
%                 = Q(d)f(x(t))
%
%    J(d)         = Q(d)df/dx
%--------------------------------------------------------------------------
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,n)).*De;
Di = (1 - speye(9,9)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D  = Di + De;

% Implement: dx(t)/dt = f(x(t - d)) = inv(1 + D.*dfdx)*f(x(t))
%                     = Q*f = Q*J*x(t)
%--------------------------------------------------------------------------
Q  = spm_inv(speye(length(J)) + D.*J);

end

function [g_theta_g]    = erp_g(m_g)

% This function evaluates the EEG forward model of the delay differential
% equation model for ERPs based on a structural formulation of the forward
% model and parameter setting
%
%   Inputs
%       m_g         : EEG forward model structure and fixed parameters
%    Outputs
%          g_theta_g   : evaluated EEG forward model
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% constant parameters
theta.Lpos  = m_g.Lpos;
theta.L     = m_g.L;
theta.J     = m_g.J;

% evaluate constants
nd          = size(theta.L,2)                ; % number of dipoles
ne          = size(m_g.sens.chanpos,1)       ; % number of electrodes
dp          = 1e-3*theta.Lpos                ; % dipole coordinates adjusted for fieldtrip

% compute canonical dipole lead-field using Fieldtrip
L_can  = NaN(ne,3,nd);
for i  = 1:nd
    L_can(:,:,i) = ft_compute_leadfield(dp(:,i)', m_g.sens, m_g.vol);
end

% evaluate channel predictions based on dipole moments
L_dip = NaN(ne,nd);
for i = 1:nd
    L_dip(:,i) = L_can(:,:,i)*theta.L(:,i);
end

% evaluate g(\theta_g)
g_theta_g     = kron(theta.J,L_dip);

end

function [vFE]          = vFE(m_theta,s_theta,ms_sigsqr,vfearg)

% This function evaluates the variational free energy for ERP-DCM toy
% problem
%
%   Inputs
%       m_theta    : p x 1 variational expectation for theta
%       s_theta    : p x p variational variance    for theta
%       ms_sigsqr  : 2 x 1 variational parameters   for sigma^2
%       vfearg     : additional argument structure with fields
%           .y                  : n x 1 array   - data
%           .h                  : string        - function handle
%           .mu_theta           : scalar prior expectation for theta
%           .sig_theta          : scalar prior variance for theta
%           .mu_sigsqr          : scalar prior scale parameter for sigma^2
%           .si_sigsqr          : scalar prior shape parameter for sigma^2
%           .theta              : true, but unknown, parameter structure
%   Outputs
%           vFE                 : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% unpack input structure
m_sigsqr        = ms_sigsqr(1);
s_sigsqr        = ms_sigsqr(2);
y               = vfearg.y;
m_f             = vfearg.m_f;
m_g             = vfearg.m_g;
mu_theta        = vfearg.mu_theta;
sig_theta       = vfearg.sig_theta;
mu_sigsqr       = vfearg.mu_sigsqr;
si_sigsqr       = vfearg.si_sigsqr;
theta           = vfearg.theta;

% evaluation of the number data points and parameters
n               = size(y,1);
p               = size(m_theta,1);

% evalutate h(m_theta) and J^h(m_theta)
h_m_theta       = erp_h(m_f,m_g,spm_unvec(m_theta, theta));
Jh_m_theta      = full(spm_cat(spm_diff(@(m_theta) erp_h(m_f,m_g,spm_unvec(m_theta, theta)),m_theta,1)));


% evaluation of the variational free energy value
T1              = -(n/2)*log(2*pi)              - (n/2)*m_sigsqr            - (1/2)*exp(-m_sigsqr +.5*s_sigsqr)*((y - h_m_theta)'*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta));
T2              = -(p/2)*log(2*pi)              - (1/2)*log(det(sig_theta))  - (1/2)*((m_theta - mu_theta)'*inv(sig_theta)*(m_theta - mu_theta) + trace(sig_theta\s_theta));
T3              = -(1/2)*log(2*pi*si_sigsqr) - m_sigsqr                     - (1/2)*(1/si_sigsqr)*(s_sigsqr + (m_sigsqr - mu_sigsqr)^2);
T4              =  (1/2)*log(det(s_theta))   + (p/2)*log(2*pi*exp(1));
T5              =  (1/2)                     + (1/2)*log(2*pi*s_sigsqr) + m_sigsqr                                                                                  ;

% evaluate the negative free energy
vFE             = -(T1 + T2 + T3 + T4 + T5);


end

function [t_k]          = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsqr, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       x_k             : p x 1     array of function argument
%       s_theta         : p x p     array of additional vFE arguments
%       ms_sigsqr   : 2 x 1     array of additional vFE arguments
%       vfearg      : additional input for vFE
%       df_x_k       :p_f x 1 array of function gradient at x_k
%       p_k             : search direction
%       c               : scalar constant in ]0,1[
%       rho             : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(x_k, s_theta, ms_sigsqr, vfearg);

% initialize step-size
t_k             = 1;

% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(x_k + t_k*p_k, s_theta, ms_sigsqr, vfearg);

% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k)
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(x_k+t_k*p_k, s_theta, ms_sigsqr, vfearg);
end
end

function [t_k]          = backtrack_vFE_ms_sigsqr(m_theta, s_theta, x_k, vfearg, df_x_k, p_k,c,rho)

% This function evalutates the backtracking step-size of a Newton search
% for the variational expectation of the free energy function vFE
%
%   Inputs
%       m_theta     : p x 1   array of additional vFE arguments
%       s_theta     : p x p array of additional vFE arguments
%       x_k         : 2 x 1     array of function arguments
%       vfearg      : additional input for vFE
%       df_x_k      : 2 x 1 array of function gradient at x_k
%       p_k         : search direction
%       c           : scalar constant in ]0,1[
%       rho         : scalar constant in ]0,1[
%
%   Output
%       t_k     : backtracking/Armijo step-size
%
% Copyright (C) Dirk Ostwald
% -------------------------------------------------------------------------
% evaluation of f(x_k)
f_x_k           = vFE(m_theta, s_theta, x_k, vfearg);

% initialize step-size
t_k             = 1;
```

```
% evaluation of f(x_(k+1)) given the initial step size
f_x_k_p_1       =  vFE(m_theta, s_theta, x_k + t_k*p_k, vfearg);

% check sufficient increase of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*t_k*(df_x_k'*p_k) || isnan(f_x_k_p_1)|| x_k(2) + t_k*p_k(2) < 0
    t_k         = t_k*rho;
    f_x_k_p_1   = vFE(m_theta, s_theta, x_k+t_k*p_k, vfearg);
end
end
```