

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

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

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

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

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

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

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

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

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

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

and its entropy is given by

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

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

$$x: \mathbb{R} \rightarrow \mathbb{R}, t \mapsto x(t) \text{ and } \varphi: \mathbb{R} \rightarrow \mathbb{R}, x \mapsto \varphi(x) \quad (1)$$

defining the one-dimensional delay differential equation

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

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

$$(\varphi \circ x)(t): \mathbb{R} \rightarrow \mathbb{R}, t \mapsto (\varphi \circ x)(t) := \varphi(x(t)). \quad (3)$$

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

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

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

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

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

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

By defining

$$Q := \left(1 + \tau \frac{d}{dx} \varphi(x(t)) \right) \quad (7)$$

we can express (6) succinctly as

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

Finally, by defining

$$\psi: \mathbb{R} \rightarrow \mathbb{R}, x \mapsto \psi(x) := Q^{-1} \varphi(x) \quad (9)$$

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

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

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	$C := (1, 0)^T$
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	$\theta_j = (-0.1, 0, 0, 0, 0, -0.46, 0, 1)$

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	$\theta_j = (-0.29, 0, 0, 0, 0, -0.04, 0, 1)$

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 ESP-DCM latent and
% forward model
% Copyright (c) Dirk Ostwald
% -----
clear all

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

% ----- Model Formulation
% -----
% neural model structure and parameters
% -----
% structural aspects and fixed parameters
m.f.f = wgm_fm_exp; % neural evolution function
m.f.ms = 200; % number of time bins
m.f.dt = 0.001; % time bin length (sec)
m.f.pct = (0:m.f.ms-1)*m.f.dt; % peristimulus time (msec)
m.f.nt = length(m.f.pct); % number of time points
m.f.n = 21; % number of neural masses
m.f.m = 9; % number of neural states per neural mass
m.f.dur = -10; % system input onset (ms)
m.f.w = sparse(m.f.n,m.f.m); % initial condition
m.f.R = [ 0.18 -0.22]; % input function parameters rho_1 and rho_2
m.f.S = [-0.04 0.08]; % static nonlinearity (activation function) parameters
m.f.T = [ 0.18 0.13; 0.07 -0.82]; % source-specific excitatory and inhibitory time constants
m.f.D = [-0.11 0.45; 0.02 -0.63]; % source-specific excitatory and inhibitory receptor densities
m.f.H = [ 0.00 -0.07 0.18 -0.37]; % source-independent intrinsic connectivity parameters
m.f.A[1] = [ 0 -0.52; -0.06 0 ]; % forward connectivity
m.f.A[2] = [ 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(od, 'Data/EM/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.z0 = zeros(m.f.n,m.g.ne,3*m.g.ne); % confound design matrix
m.g.locs = [2 -31 58; 14 -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 -0.46 0 1]; % dipole weighting specification

% ----- Visualization
% -----
% initialise 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 = del; % magnifying factor for dipole moment
fname = [pwd, 'Data/EM/cortex_s124_surf.gii']; % cortical mesh filename
struct2ipf(fname); % load cortical mesh using ipfci and sumtree methods and convert to structure
m = mtfldim, 'mat'; % remove mat field for patch functionality

subplot(2,3,1)
hold on
patch(m, 'facecolor', [1.5 1.5 1.5], '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 = m*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),m*mom(1),m*mom(2),m*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([15 10]);

% plot canonical dipole leadfields
% -----
nd = size(m.g.L,2); % number of dipoles
ne = size(m.g.sens,2); % number of electrodes
dp = nd*m.g.Lpos; % dipole coordinates adjusted for fieldtrip
L_can = NaN(ne,3,nd); % compute canonical dipole lead-field using fieldtrip
for l = 1:nd
    L_can(:,l,:) = ft_compute_leadfield(dp(l,:), m.g.sens, m.g.vol);
end
L_can = L_can/(10^4)*round(log10(max(max(abs(L_can))))/8)/11; % rescale lead-field according to SPW
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_ave = L_can(:,1,1)*mom(1,1); % prepare fieldtrip parameter structure fields for plot layout
cfg_layout = 'overlaid';
lay = ft_prepare_layout(cfg, dat); % obtain correct channel locations for fieldtrip
load([pwd, 'Data/EM/cortex_s124_locs.mat']); % match the labels in the of the current data
for n = 1:length(dat_label)
    actelec = cellmat(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];
axcaler = 0.45; % specify spatial scaling of the plot
lay_pos(1:length(dat_label),:) = [X;Y]*axcaler;
lay_width = lay_width*axcaler;
lay_height = lay_height*axcaler;
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] = exp_f(m.f,theta);
f_theta_f_sl = f_theta_f_u(:,1:2*end-1);
tidx = 41; % time index for plotting
subplot(2,3,4)
[ax,li,li] = plotly(m.f.pct,m.g.J*f_theta_f_sl'*m.f.pct,f_theta_f_sl); % visualize
xlim([0 15])
set(h1, 'Color', 'b');
set(h2, 'Color', [1.7 1.7 1.7]);
set(ax(1), 'ymin', 0);
set(ax(1), 'FontName', 'Calibri Light', 'FontSize', 20)
set(ax(2), 'FontName', 'Calibri Light', 'FontSize', 20)
line(m.f.pct,tidx, m.f.pct(tidx),get(ax(2), 'Ymin'), '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 = del; % magnifying factor for dipole moment
fname = [pwd, 'Data/EM/cortex_s124_surf.gii']; % cortical mesh filename
struct2ipf(fname); % load cortical mesh using ipfci and sumtree methods and convert to structure
m = mtfldim, 'mat'; % remove mat field for patch functionality

subplot(2,3,5)
hold on
patch(m, 'facecolor', [1.5 1.5 1.5], 'EdgeColor', 'none', 'FaceLighting', ' gouraud', 'visible', 'on', 'facealpha', 0.1); % plot mesh
plot3(pos(1),pos(2),pos(3), 'bo', 'MarkerSize', 2, 'MarkerFaceColor', 'b'); % canonical axis projection of the dipole moment
pro = m*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),m*mom(1),m*mom(2),m*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)
xlabel('x', 'FontName', 'Calibri Light', 'Rotation', 0, 'FontSize', 22)
set(gca, 'FontName', 'Calibri Light', 'FontSize', 16)
view(45, 145)

% plot actual topology
%-----
L_dip = NaN(ind);
for i = 1:ind
    L_dip(i,1) = L_can(i,1)*m_g.L(i,1);
end
g_theta_g = [];
for i = 1:ind
    g_theta_g = [g_theta_g kron(m_g.J, L_dip(i,1))];
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_dmsord = 'dms_time';
dat_time = 0;
dat_xyz = h_theta(:,1:dx);
cfg_layout = 'ordered'; % prepare fieldtrip parameter structure fields for plot layout
lay = ft_prepare_layout(cfg_dat);
load(pwd, '\Data\BIM\coords_398_elec.mat'); % obtain correct channel locations for fieldtrip
for n = 1:length(dat_label)
    actlab = cell2mat(dat_label(n,:)); % match the labels in the of the current MEG object
    ind = strcmp(actlab, Layout_Labels, 'exact');
    X(n) = Layout_Coords(ind,1);
    Y(n) = Layout_Coords(ind,2);
end
lay_pos(1:64,:) = [X;Y];
scale = 0.4; % specify spatial scaling of the plot
lay_pos(1:length(dat_label),:) = [X;Y]*scale;
lay_width = lay_width*scale;
lay_height = lay_height*scale;
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] = exp_nm_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 EBP
%
% 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] = exp_nm_f_theta;
g_theta_g = exp_nm_g;
% evaluate concatenated function h
h_theta_fg = 1e4*spm_vec([g_theta_g; f_theta_f]);
end

function [f_theta_f, u] = exp_nm_f_vartheta
% This function integrates the neural evolution function of the delay
% differential equation model for EBP. It is based on spm_gen_exp.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_B = m_f.B;
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.ms)*m_f.dt)+1000;
delay = m_f.delay+28*theta_B(1,1);
scale = m_f.scale+1*exp(theta_B(1,1));
u = 32*exp(-(t - delay).^2/(2*scale^2));
% integrate system
%
% x : initial condition
% f : function handle spm_fm_exp
% dt : integration time bin
% [fx, dfdx, D] = [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))))); %
% M = cell2mat(1, dt*(1:1)); %
% n = spm_length(x); %
% Q = [spm_exp(dt)*dfdx/n - spye(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 = 1:1:size(u,1)
    % update dx = -expm(dt*N) - I*inv(J)*f(x,u)
    for j = 1:N
        v = v + Q*f(v,u(1), theta_m_f);
    end
    y(i,:) = v;
end
% transpose
f_theta_f = y';
end

function [f, J, Q] = spm_fm_exp(x, u, P, M)
% state equations for a neural mass model of exps
% FORMAT [f, J, Q] = spm_fm_exp(x, u, P, M)
% FORMAT [f, J] = spm_fm_exp(x, u, P, M)
% FORMAT [f] = spm_fm_exp(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))
%
% Prior fixed parameter scaling [Defaults]
%
% M.pP.E = [32 16 4]; % extrinsic rates (forward, backward, lateral)
% M.pP.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2, g3, g4)
% M.pP.D = [2 16]; % propagation delays (intrinsic, extrinsic)
% M.pP.C = [4 32]; % receptor densities (excitatory, inhibitory)
% M.pP.R = [8 16]; % synaptic constants (excitatory, inhibitory)
% M.pP.S = [1/2]; % parameter of static nonlinearity
%
% David O. Priston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neural dynamics. NeuroImage 20: 1743-1755
%
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging
%
% Karl Friston
% 21st spm_fm_exp.m 5369 2013-03-28 20:09:272 Karl S
%
% get dimensions and configure state variables
%-----
N = length(A[1]); % number of sources
x = spm_unvec(x, M.A); % neuronal states
% (default) fixed parameters
%-----
E = [1 1/2 1/8]*128; % extrinsic rates (forward, backward, lateral)

```

```

G = [1 4/5 1/4 1/4]*100; % 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]*R[1];
A[2] = P.A[2]*R[2];
A[3] = P.A[3]*R[3];
C = P.C;

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

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

% exogenous input
U = C*u(1)^2;

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

% vectorize
f = spm_vec(f);

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

% Jacobian
J = spm_diff(W,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) = 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
D = kron(ones(9,9),De);
Di = kron(Di,speye(n,n));
D = Di + De;

% Delay parameter matrix Visualization

% ERP-DCM ordering
n = 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
roT = cell(1,mu); % re-ordered ticks label initialization
idx = 1; % re-ordered tick label index

% re-ordering hack
for sv = 1:9
    roDI(sv,:) = [DI(sv*2-1,1:2:17) DI(sv*2-1,2:2:18)];
    roDI(sv+1,:) = [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+1,:) = [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+1,:) = [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
        roT{idx} = ['s_' num2str(sv) ' ' num2str(i) ''];
        idx = idx + 1;
    end
end

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

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

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

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

end

function [g_theta_g] = exp_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_champs,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 SNR
L_can = L_can*(10^4)^(round(log10(max(max(abs(L_can))))/8)/3));

% 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 ESP-DM toy example.
% Copyright (C) Dirk Ostwald
% -----
% ----- Initialization
% -----
clc
close all
stream = RandStream.getGlobalStream;reset(stream); % set the random number generator for reproducible results
% -----
% ----- 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 = sym_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_l_arg_y = y;
vFE_l_arg_h = h;
vFE_l_arg_m_theta = m_theta;
vFE_l_arg_sig_theta = sig_theta;
vFE_l_arg_sigma = sigma;
vFE_s_theta(1) = vFE_l(m_theta,s_theta,vFE_l_arg);
vFE_s_theta(2) = vFE_l(m_theta,s_theta_update ,vFE_l_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')
ylim([0 11])
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
% plot variational free energy descent
% -----
% ----- array initialization
vFE_s_theta = NaN(length(s_space_full),length(m_space_full));
% cycle over variational expectation space
for j = 1:length(s_space_full)
    for i = 1:length(m_space_full)
        vFE_s_theta(i,j) = vFE_l(m_space_full(j),s_space_full(i),vFE_l_arg);
    end
end
subplot(2,3,2)
hold on
surf(m_space_full, s_space_full, -vFE_s_theta, 'EdgeColor', 'None')
alpha(6)
plot3(m_theta, m_theta, [s_theta, s_theta_update], -vFE_s_theta, 'ko', 'MarkerFaceColor', 'k', 'MarkerSize', 10)
xlim([s_min_full s_max_full])
ylim([s_min_full s_max_full])
set(gca, 'Dir', 'normal')
xlabel('m_theta', 'FontSize', 26, 'FontName', 'Calibri Light')
ylabel('s_theta', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
zlabel('P', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')
view([116 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
scots = ['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_theta = linspace(s_min,s_max,s_res);
    m_theta = theta;
    vFE_l_arg_mu_theta = mu_theta;
    vFE_l_arg_sig_theta = sig_theta;
    % array 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_l(m_theta,s_space(i),vFE_l_arg);
    end
    % evaluate the variational variance update equation for the current choice of m_theta
    Jh_m_theta = sym_diff(h,m_theta,1);
    s_theta_update = 1/((1/sigma)*(Jh_m_theta'*Jh_m_theta)) + (1/sig_theta);
    P_s_theta_update = vFE_l(m_theta,s_theta_update,vFE_l_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', scots[s]);
    plot(s_theta_update, -P_s_theta_update, 'ko', 'MarkerSize', 10, 'MarkerFaceColor', scots[s])
end
% -----
% ----- 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
zho = 0.99; % backtracking constant zho
max_iter = 5; % maximal number of iterations
% -----
% ----- model estimation - iteration arrays
% -----
m_theta_n = NaN(p,max_iter); % variational expectation parameter
P_n = NaN(1,max_iter); % variational free energy
dP_n = NaN(1,max_iter); % gradient of the variational free energy wrt m_theta
d2P_n = NaN(1,max_iter); % Hessian of the variational free energy wrt m_theta
c1_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 = sym_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_L_arg.y = y;
VFE_L_arg.mu_theta = h;
VFE_L_arg.mu_theta = mu_theta;
VFE_L_arg.sig_theta = sig_theta;
VFE_L_arg.sigma = sigma;
F_n(i) = VFE_L(m_theta_n(i), s_theta, VFE_L_arg);
dF_n(i) = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta, VFE_L_arg), m_theta_n(i), 1)));
d2F_n(i) = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta, VFE_L_arg), m_theta_n(i), [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 = -d2F_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
d2F_n(i-1) = d2F_n(i-1) + max(0, delta - min(eig(d2F_n(i-1))))*eye(p);
% re-evaluate Newton search direction
p_i = -d2F_n(i-1)\dF_n(i-1);
end
% backtracking evaluation of the step length
t_i_n(i-1) = f_backtrack_VFE_L(m_theta_n(i-1), s_theta, VFE_L_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_L(m_theta_n(i), s_theta, VFE_L_arg);
dF_n(i) = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta, VFE_L_arg), m_theta_n(i), 1)));
d2F_n(i) = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta, VFE_L_arg), m_theta_n(i), [1 1]));)
end

% plot the the MAP fit
% -----
h_m_theta_MAP = h(m_theta_map);
subplot(2,3,1)
plot(h(m_theta_MAP), 'y', 'h(theta_MAP)', 'Location', 'NorthWest')
% visualize Newton descent on variational free energy surface
% -----
m_min_full = 1;
m_max_full = 2.4;
m_res_full = 1e-3;
m_space_full = linspace(m_min_full, m_max_full, m_res_full);
s_min_full = 1;
s_max_full = 9e-5;
s_res_full = 1e-2;
s_space_full = linspace(s_min_full, s_max_full, s_res_full);
% array initialization
VFE_s_m_theta = NaN(length(s_space_full), length(m_space_full));
% cycle over variational expectation space
for i = 1:length(m_space_full)
for j = 1:length(s_space_full)
VFE_s_m_theta(i,j) = VFE_L(m_space_full(j), s_space_full(i), VFE_L_arg);
end
end
subplot(2,3,4)
hold on
surf(m_space_full, s_space_full, ~VFE_s_m_theta, 'EdgeColor', 'None')
alpha(6)
plot(m_theta_n, s_theta*ones(length(m_theta_n)), 'P_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', 'FontSize', 26, 'FontName', 'Calibri Light', 'Rotation', 0)
xlabel('P', '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_L_arg.mu_theta = mu_theta;
VFE_L_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 = 1e-2;
m_space = linspace(m_min, m_max, m_res);
% array initialization
VFE_m_theta = NaN(1, length(m_space));
dVFE_m_theta_n = NaN(1, length(m_space));
d2VFE_m_theta_n = NaN(1, length(m_space));
% cycle over variational expectation space
for i = 1:length(m_space)
VFE_m_theta(i) = VFE_L(m_space(i), s_theta, VFE_L_arg);
dVFE_m_theta_n(i) = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta, VFE_L_arg), m_space(i), 1)));
d2VFE_m_theta_n(i) = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta, VFE_L_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, 'k')
plot(m_space, ~d2VFE_m_theta_n, 'k')
plot(m_space, zeros(1, length(m_space)), 'k')
xlabel('m_theta', 'FontSize', 26, 'FontName', 'Calibri Light')
ylim([m_min ~VFE_m_theta] max(~d2VFE_m_theta_n))
xlim([m_min m_max])
legend('P', 'dP', 'd2P', 'Location', 'NorthWest')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
end
return
% -----
% Estimation of variational mean & variance
% -----
fprintf('Joint estimation of variational variance and mean for theta(s)')
% 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; % 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_L_arg.y = y;
VFE_L_arg.mu_theta = h;
VFE_L_arg.sig_theta = sig_theta;
VFE_L_arg.sigma = sigma;
F_i(1) = VFE_L(m_theta_i(1), s_theta_i(1), VFE_L_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 for the current choice of m_theta
fprintf(' Estimating @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(' [fixed \n')
% evaluation of the variational free function derivatives
dF = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta_i(i), VFE_L_arg), m_theta_i(i-1), 1)));
d2F = full(spm_cat(spm_diff(@(m_theta) VFE_L(m_theta, s_theta_i(i), VFE_L_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_1(1-1);
% iterations
for j = 2:max_j
% backtracking Newton line search with Hessian modification for
% variational expectation
% Newton search direction
p_j = -dSP/dF;
% Hessian modification if p_j is not a descent direction
if -(p_j)'*dF < 0
% diagonal Hessian modification based on spectral decomposition
dSP = dSP + max(0, delta - min(eig(dSP)))'*eye(p);
% re-evaluate Newton search direction
p_j = -dSP/dF;
end
% backtracking evaluation of the step length
t_j = f_backtrack_VFE_1(m_theta, s_theta_1(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_1(i), VFE_1_arg), m_theta, 1)));
dSP = full(spm_cat(spm_diff(@(m_theta) VFE_1(m_theta, s_theta_1(i), VFE_1_arg), m_theta, 1 1)));
end
fprintf(' finished. \n')
% record m_theta and variational free energy update
m_theta_1(i) = m_theta;
F_1(i) = VFE_1(m_theta_1(i), s_theta_1(i), VFE_1_arg);
end
% visualization
% -----
fig = figure;
set(fig, 'Color', [1 1 1]);
% visualization of prior and approximated posterior over theta
theta_min = -2;
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_1(end), sqrt(s_theta_1(end))));
xlim([theta_min theta_max])
xlabel(m_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 = 1e4;
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;
% array 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(4)
plot3(m_theta_1(i), s_theta_1(i), -F_1(i), 'ko', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
xlabel(m_theta, 'FontSize', 26, 'FontName', 'Calibri Light')
ylabel(s_theta_1, '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])
axis([-3.5e3 0])
view([15 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 = 2.04;
m_res_full = 1e2;
m_space_full = linspace(m_min_full, m_max_full, m_res_full);
s_min_full = -1e-4;
s_max_full = 1e-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;
% array 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')
alpha(4)
plot3(m_theta_1(end), s_theta_1(end), -F_1(end), 'ko', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
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_1, 'FontSize', 26, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
view([15 12])
% -----
% Variational Bayesian estimation of sigma^2
% -----
fprintf(' Estimation of variational shape and scale sigma^2 \n')
mu_theta = 2.021; % prior expectation
sig_theta_all = [7.4015e-05 2e-3]; % prior variance
mu_sigqr = 0.2; % prior shape parameter
s1_sigqr = 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);
% algorithm 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
mu_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
mu_n(1,:) = [mu_sigqr; s1_sigqr];
% evaluation of the variational free energy, its gradient and Hessian
VFE_arg.y = y;
VFE_arg.h = h;
VFE_arg.mu_theta = mu_theta;
VFE_arg.s_theta = s_theta;
VFE_arg.mu_sigqr = mu_sigqr;
VFE_arg.s1_sigqr = s1_sigqr;
VFE_arg.F_n = VFE_1(mu_n(1,1), VFE_arg);
dF = full(spm_cat(spm_diff(@(mu_n) VFE_2(mu_n, VFE_arg), mu_n(1,1), 1)));
dSP = full(spm_cat(spm_diff(@(mu_n) VFE_2(mu_n, VFE_arg), mu_n(1,1), 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 = -dSP/dP;
% Hessian modification if p_i is not a descent direction
if ~(p_i' * dP < 0)
% diagonal Hessian modification based on spectral decomposition
dSP = dSP + max(0, delta - min(sig(dSP))) * eye(2);
% re-evaluate Newton search direction
p_i = -dSP/dP;
end
% backtracking evaluation of the step length
t_i = f_backtrack_VFE_2(mu_n(i-1), VFE_arg, dP, p_i, c, rho);
% perform Newton update
mu_n(i) = mu_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(mu_n(i), VFE_arg);
dP = -full_spe_cat(spe_diff(@(mu_n) VFE_2(mu_n, VFE_arg), mu_n(i-1), 1));
dSP = -full_spe_cat(spe_diff(@(mu_n) VFE_2(mu_n, VFE_arg), mu_n(i-1), 1, 1));
end

% prior and variational distribution Visualization
%-----
sigqr_1 = linspace(1e-3, 2e1, 1e2); % RV support
subplot(2,3,4)
hold on
plot(sigqr_x, logpdf(sigqr_x, mu_sigqr, sqrt(sigqr_1)) , 'b')
plot(sigqr_x, logpdf(sigqr_x, mu_n(1, end), sqrt(mu_n(2, end))) , 'r')
plot(exp(mu_sigqr), 0, 'o', 'MarkerFaceColor', 'b', 'MarkerSize', 10)
plot(exp(mu_n(1, end)), 0, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 10)
xlabel('sigma^2', 'FontName', 'Calibri Light', 'FontSize', 26)
legend(['\mu_sigqr', 'RV(\sigma^2)'], 'numstr(0.5 + log(2)*pi*\mu_sigqr) + mu_sigqr, 2), ...
['\mu_n(1, end)', 'RV(\sigma^2)'], 'numstr(0.5 + log(2)*pi*\mu_n(2, end)) + mu_n(1, end), 2))
set(gca, 'FontName', 'Calibri Light', 'FontSize', 20)
end

% variational free energy and Newton iteration Visualization
%-----
mu_sigqr = linspace(1e-3, 7, 1e2); % scale variational parameter
s_sigqr = linspace(1e-2, 1e-1, 1e2); % shape variational parameter
% evaluate variational free energy surface
FE_all = NaN(length(mu_sigqr), length(s_sigqr));
for j = 1:length(s_sigqr)
FE_all(i, j) = VFE_3(mu_sigqr(i), s_sigqr(j)), VFE_arg;
end

subplot(2,3,6)
hold on
surf(mu_sigqr, s_sigqr, FE_all, 'EdgeColor', 'None')
alpha(6)
plot(mu_n(2, 1), mu_n(1, 1), -P_n, 'k--', 'MarkerFaceColor', 'k', 'MarkerSize', 12)
xlim([mu_sigqr(1) mu_sigqr(end)])
ylim([s_sigqr(1) s_sigqr(end)])
xlabel('mu_sigqr', 'FontName', 'Calibri Light', 'FontSize', 26)
ylabel('s_sigqr', 'FontName', 'Calibri Light', 'FontSize', 26)
xlabel('mu_n(1, end)', 'FontName', 'Calibri Light', 'FontSize', 26)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
view([1 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 [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_3(mu_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
% mu_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 : handle - function handle
% mu_theta : 1 x 1 scalar - prior expectation
% s_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(mu_theta, 1);
% evaluate f(mu_theta) and Jf(mu_theta, 1)
h_mu_theta = h(mu_theta);
Jh_mu_theta = spe_diff(h_mu_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_mu_theta) * (y - h_mu_theta) + (1/sigma) * (Jh_mu_theta)' * Jh_mu_theta * s_theta);
T2 = -(p/2) * log(2*pi) - (1/2) * log(sig_theta) - ((1/2) * (1/sig_theta) * (mu_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 evaluates 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 + t_k * p_k) 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(mu_sigqr, 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
% mu_sigqr : 2 x 1 log-scale and shape parameters
% VFE_arg : structure with additional inputs with fields
% .y : n x 1 data vector
% .mu_sigqr : scalar log-scale prior parameter
% .s_sigqr : scalar shape prior parameter
%
% Outputs
% FE : scalar negative variational free energy

```

```

% Copyright (C) Dirk Ostwald
% -----
% unpack and summarise input parameters
m_sigqr = m_sigqr(1);
s_sigqr = m_sigqr(2);
m_theta = VFE_arg.m_theta;
s_theta = VFE_arg.s_theta;
mu_sigqr = VFE_arg.mu_sigqr;
si_sigqr = VFE_arg.si_sigqr;
y = VFE_arg.y;
h = VFE_arg.h;

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

% evaluate f(m_theta(1)) and df(m_theta,1)
h_m_theta = h_m_theta;
Jh_m_theta = spm_diff(h_m_theta,1);

% evaluate relevant variational free energy terms
T2 = -(n/2)*log(2*pi) - (n/2)*m_sigqr - (1/2)*exp(-m_sigqr + s*m_sigqr)*(y - h_m_theta)^(y - h_m_theta) + trace((Jh_m_theta*(Jh_m_theta)*m_theta));
T2 = -(1/2)*log(2*pi*si_sigqr) - m_sigqr - (1/2)*(1/si_sigqr)*(s_sigqr + m_sigqr - mu_sigqr)^2;
T3 = (1/2) + (1/2)*log(2*pi*s_sigqr) + m_sigqr;

% evaluate negative variational free energy
FE = -(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_m1
% This function implements the estimation of an RSP-DM toy-model for
% unknown variance parameter sigma^2.
%
% Copyright (c) Dirk Ostwald
% -----
%
% ----- Initialization
% -----
clear
stream = RandStream.getGlobalStream/reset(stream); % set the random number generator for reproducible results
% 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
sigqr = 1; % true, but unknown, variance parameter
h_theta = h(theta); % evaluate h
y = mvnrnd(h_theta, sigqr*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 = 1e-3;
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 expectation parameter theta
vfearg.sig_theta = [1e6 0 1e-3]; % prior variance parameter theta
vfearg.mu_sigqr = 4.6; % prior shape parameter
vfearg.si_sigqr = 1e-3; % prior scale parameter
mu_sigqr = [vfearg.mu_sigqr vfearg.si_sigqr]; % variational shape and scale parameter settings
case 2
vfearg.mu_theta = [0 5]'; % prior expectation parameter theta
vfearg.sig_theta = [1e6 0 1e-3]; % prior variance parameter theta
vfearg.mu_sigqr = 4.6; % prior shape parameter
vfearg.si_sigqr = 1e-3; % prior scale parameter
mu_sigqr = [vfearg.mu_sigqr vfearg.si_sigqr]; % variational shape and scale parameter settings
case 3
vfearg.mu_theta = [0 5]'; % prior expectation parameter theta
vfearg.sig_theta = [1e6 0 1e-3]; % prior variance parameter theta
vfearg.mu_sigqr = 1e-3; % prior shape parameter
vfearg.si_sigqr = 9.2; % prior scale parameter
mu_sigqr = [vfearg.mu_sigqr vfearg.si_sigqr]; % variational shape and scale parameter settings
end
% variational variance parameter evaluation
Jh_m_theta = spm_diff(h, vfearg.mu_theta, 1);
s_theta = inv((exp(-mu_sigqr(1)+(1/2)*mu_sigqr(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, mu_sigqr, 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, [t-1]*4+1);
imagesc(m_theta_f_space, m_theta_g_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 (mu_sigqr, si_sigqr) optimization
delta = 1e-4; % Heaviside 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 expectation parameter theta
sig_theta = [1e6 0 1e-3]; % prior variance parameter theta
mu_si_sigqr = [4.6; 1e-3]; % prior shape and scale parameter setting \sigma^2
case 2
mu_theta = [0 5]'; % prior expectation parameter theta
sig_theta = [1e6 0 1e6]; % prior variance parameter theta
mu_si_sigqr = [4.6; 1e-3]; % prior shape and scale parameter setting \sigma^2
case 3
mu_theta = [0 5]'; % prior expectation parameter theta
sig_theta = [1e6 0 1e6]; % prior variance parameter theta
mu_si_sigqr = [2.3; 1e-0]; % prior shape and scale parameter setting \sigma^2
end
% model estimation - iteration 0
% initialization of the variational parameters to prior parameters
m_theta = mu_theta;
s_theta = sig_theta;
mu_sigqr = mu_si_sigqr;
vfearg.y = y;
vfearg.h = h;
vfearg.mu_theta = mu_theta;
vfearg.sig_theta = sig_theta;
vfearg.mu_sigqr = mu_si_sigqr(1);
vfearg.si_sigqr = mu_si_sigqr(2);
% initialization and initial evaluation of the negative free energy
F = NaN(max_i);
F(1) = VFE(m_theta, s_theta, mu_sigqr, vfearg);
% VBNE iterations 1,2,3,...
% -----
for i = 2:max_i
% User update
% -----
fprintf('Fixed-Form VB Newton Iteration %d.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(-mu_sigqr(1)+(1/2)*mu_sigqr(2))*(Jh_m_theta'*Jh_m_theta + inv(sig_theta))));
% Newton line search for theta_f variational expectation parameter
% -----
% iteration strage
m_theta_j = NaN(p, max_j);
F_m_theta_j = NaN(1, max_j);
% iterand and objective function initialization
m_theta_j(1, 1) = m_theta;
F_m_theta_j(1) = VFE(m_theta, s_theta, mu_sigqr, vfearg);
% derivatives initialization
DF_m_theta = full(spm_cat(spm_diff(@(m_theta) VFE(m_theta, s_theta, mu_sigqr, vfearg), m_theta, 1)));
dDF_m_theta = full(spm_cat(spm_diff(@(m_theta) VFE(m_theta, s_theta, mu_sigqr, vfearg), m_theta, [1 1])));
% Newton iterations 1,2,3,...
for j = 2:max_j
% Newton search direction
p_j = -dDF_m_theta/VF_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
dF_m_theta = dF_m_theta + max(0,(delta - min(eig(dF_m_theta))))*eye(p);
% re-evaluate Newton search direction
p_j = -dF_m_theta/dF_m_theta;
end
% backtracking evaluation of the step length
t_j = backtrack_VFE_m_theta(m_theta, s_theta, m_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, m_sigsqr, vfearg), m_theta, 1)));
dF_m_theta = full(spm_cat(spm_diff(@(m_theta) VFE(m_theta, s_theta, m_sigsqr, vfearg), m_theta, [1 1])));
% record updates
m_theta_j(1,j) = m_theta;
F_m_theta_j(j) = VFE(m_theta,s_theta, m_sigsqr, vfearg);
end

% Newton line search for sigma^2 variational parameters
% -----
% iteration arrays
m_sigsqr_k = NaN(2,max_k);
F_m_sigsqr_k = NaN(1,max_j);
% iterand and objective function initialization
m_sigsqr_k(1) = m_sigsqr;
F_m_sigsqr_k(1) = VFE(m_theta,s_theta, m_sigsqr, vfearg);
% evaluation of the current negative free energy gradient and Hessian wrt m_sigsqr
dF_m_sigsqr = full(spm_cat(spm_diff(@(m_sigsqr) VFE(m_theta,s_theta, m_sigsqr, vfearg), m_sigsqr, 1)));
dF_m_sigsqr = full(spm_cat(spm_diff(@(m_sigsqr) VFE(m_theta,s_theta, m_sigsqr, vfearg), m_sigsqr, [1 1])));
% Newton iterations 1,2,3,...
for k = 2:max_k
% Newton search direction
p_k = -dF_m_sigsqr/dF_m_sigsqr;
% Hessian modification if p_k is not a descent direction
if ~(p_k'*dF_m_sigsqr < 0)
% diagonal Hessian modification based on spectral decomposition
dF_m_sigsqr = dF_m_sigsqr + max(0,(delta - min(eig(dF_m_sigsqr))))*eye(2);
% re-evaluate Newton search direction
p_k = -dF_m_sigsqr/dF_m_sigsqr;
end
% backtracking evaluation of the step length
t_k = backtrack_VFE_m_sigsqr(m_theta, s_theta, m_sigsqr, vfearg, dF_m_sigsqr, p_k,c,rho);
% perform Newton update
m_sigsqr = m_sigsqr + t_k*p_k;
% update remaining entries in free energy arrays
dF_m_sigsqr = full(spm_cat(spm_diff(@(m_sigsqr) VFE(m_theta,s_theta, m_sigsqr, vfearg), m_sigsqr, 1)));
dF_m_sigsqr = full(spm_cat(spm_diff(@(m_sigsqr) VFE(m_theta,s_theta, m_sigsqr, vfearg), m_sigsqr, [1 1])));
% record updates
m_sigsqr_k(1:k) = m_sigsqr;
F_m_sigsqr_k(k) = VFE(m_theta,s_theta, m_sigsqr, vfearg);
end

% free energy update
% -----
F(1) = VFE(m_theta,s_theta, m_sigsqr, vfearg);
% user update
% -----
fprintf('VFE %6.3f dVFE %6.3f \n', F(1), F(1) - F(1-1))
end

% -----
% Visualisation
% -----
% 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 MDP fit
subplot(3,5,(aim-1)*5 + 1)
hold on
plot(1:10, h_theta)
plot(1:10, h_m_theta_MAP, 'MarkerSize', 6)
xlabel('x', 'FontSize', 22, 'FontName', 'Calibri Light')
if aim == 1
legend('h(theta)', 'y', 'h_m_theta_MAP'), 'Location', 'NorthWest'
end
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')
subplot(3,5,(aim-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, mu_theta(1), sqrt(s_theta(1,2))), 'r--');
plot(mu_theta(1), 0, 'o', 'MarkerFaceColor', 'b', 'MarkerSize', 8)
plot(mu_theta(1), 0, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8);
xlabel('\theta_f', 'FontSize', 22, 'FontName', 'Calibri Light')
if aim == 2
legend('p(\theta_f)', 'q(\theta_f)'), 'Location', 'NorthWest'
end
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')
xlim([theta_f_at(1) theta_f_x(end)])
subplot(3,5,(aim-1)*5 + 3)
hold on
plot(theta_g_x, pdf(Normal, theta_g_x, mu_theta(2), sqrt(sig_theta(2))), 'b');
plot(theta_g_x, pdf(Normal, theta_g_x, mu_theta(2), sqrt(s_theta(2,2))), 'r--');
plot(mu_theta(2), 0, 'o', 'MarkerFaceColor', 'b', 'MarkerSize', 8)
plot(mu_theta(2), 0, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 8);
xlabel('\theta_g', 'FontSize', 22, 'FontName', 'Calibri Light')
if aim == 1
legend('p(\theta_g)', 'q(\theta_g)'), 'Location', 'NorthWest'
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,(aim-1)*5 + 4)
hold on
plot(sigsqr_x, logpdf(sigsqr_x, mu_s1_sigsqr(1), sqrt(mu_s1_sigsqr(2))), 'b');
plot(sigsqr_x, logpdf(sigsqr_x, m_sigsqr(1), sqrt(m_sigsqr(2))), 'r--');
plot(exp(mu_s1_sigsqr(1) + 5*mu_s1_sigsqr(2)), 0, 'o', 'MarkerFaceColor', 'b', 'MarkerSize', 10)
plot(exp(mu_sigsqr(1) + 5*m_sigsqr(2)), 0, 'o', 'MarkerFaceColor', 'r', 'MarkerSize', 10)
set(gca, 'FontSize', 16, 'FontName', 'Calibri Light')
if aim == 1
legend('p(\sigma^2)', 'q(\sigma^2)'), 'Location', 'NorthWest'
end
xlabel('\sigma^2', 'FontSize', 22, 'FontName', 'Calibri Light')
xlim([sigsqr_x(end)])

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

% -----
% Subfunctions
% -----
function [f_theta_f] = f_fun(theta_f)
% This function evaluates the ERD-DM 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 ERD-DM 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 ESP-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_sigsgqr,vfearg)
% This function evaluates the variational free energy for ESP-DCM toy
% problem
%
% Inputs
% m_theta : 2 x 1 variational expectation for theta
% s_theta : 2 x 2 variational variance for theta
% ms_sigsgqr : 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
% .ms_sigsgqr : scalar prior scale parameter for sigma^2
% .sl_sigsgqr : scalar prior shape parameter for sigma^2
%
% Outputs
% VFE : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
% -----
% unpack input structure
m_sigsgqr = ms_sigsgqr(1);
s_sigsgqr = ms_sigsgqr(2);
y = vfearg.y;
h = vfearg.h;
mu_theta = vfearg.mu_theta;
sig_theta = vfearg.sig_theta;
ms_sigsgqr = vfearg.ms_sigsgqr;
sl_sigsgqr = vfearg.sl_sigsgqr;
% evaluation of the number data points and parameters
n = size(y,1);
p = size(m_theta,1);
% evaluate h(m_theta,f()) and J^h(m_theta,f())
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_sigsgqr - (1/2)*exp(-m_sigsgqr + 5*s_sigsgqr)*(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)*(1/s_sigsgqr)*(m_sigsgqr - m_theta)*(m_sigsgqr - m_theta) + trace(sig_theta*s_theta);
T3 = -(1/2)*log(2*pi)*sl_sigsgqr - m_sigsgqr - (1/2)*(1/sl_sigsgqr)*(s_sigsgqr + m_sigsgqr - ms_sigsgqr)^2;
T4 = (1/2)*log(det(s_theta)) + (p/2)*log(2*pi)*exp(1);
T5 = (1/2)*(1/2)*log(2*pi)*m_sigsgqr + m_sigsgqr;
% evaluate the negative free energy
VFE = -(T1 + T2 + T3 + T4 + T5);
end

function [VFE] = vfe_terms(m_theta,s_theta,ms_sigsgqr,vfearg)
% This function evaluates the variational free energy for ESP-DCM toy
% problem
%
% Inputs
% m_theta : 2 x 1 variational expectation for theta
% s_theta : 2 x 2 variational variance for theta
% ms_sigsgqr : 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
% .ms_sigsgqr : scalar prior scale parameter for sigma^2
% .sl_sigsgqr : 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_sigsgqr
% .T4 : scalar entropy of theta
% .T5 : scalar entropy of sigma^2
% .VFE : scalar variational free energy
%
% Copyright (C) Dirk Ostwald
% -----
% unpack input structure
m_sigsgqr = ms_sigsgqr(1);
s_sigsgqr = ms_sigsgqr(2);
y = vfearg.y;
h = vfearg.h;
mu_theta = vfearg.mu_theta;
sig_theta = vfearg.sig_theta;
ms_sigsgqr = vfearg.ms_sigsgqr;
sl_sigsgqr = vfearg.sl_sigsgqr;
% evaluation of the number data points and parameters
n = size(y,1);
p = size(m_theta,1);
% evaluate h(m_theta,f()) and J^h(m_theta,f())
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_sigsgqr - (1/2)*exp(-m_sigsgqr + 5*s_sigsgqr)*(y - h_m_theta)*(y - h_m_theta);
VFE_T2 = -(1/2)*exp(-m_sigsgqr + 5*s_sigsgqr)*trace((Jh_m_theta'*Jh_m_theta)*s_theta);
VFE_T3 = -(p/2)*log(2*pi) - (1/2)*log(det(sig_theta)) - (1/2)*(1/s_theta)*(m_theta - mu_theta)*(m_theta - mu_theta) + trace(sig_theta*s_theta);
VFE_T4 = -(1/2)*log(2*pi)*sl_sigsgqr - m_sigsgqr - (1/2)*(1/sl_sigsgqr)*(s_sigsgqr + m_sigsgqr - ms_sigsgqr)^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)*m_sigsgqr + m_sigsgqr;
% 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_sigsgqr,vfearg,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 : p x 1 array of function argument
% s_theta : p x p array of additional VFE arguments
% ms_sigsgqr : 2 x 1 array of additional VFE arguments
% vfearg : additional input for VFE
% df_x_k : [p,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(x_k,s_theta,ms_sigsgqr,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_sigsgqr,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_sigsgqr,vfearg);
end
end

function [t_k] = backtrack_vfe_ms_sigsgqr(m_theta,s_theta,x_k,vfearg,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
% 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
% vfeary   : 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, a_theta, x_k, vfeary);
% 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, a_theta, x_k + t_k*p_k, vfeary);
% 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, a_theta, x_k+t_k*p_k, vfeary);
end
end

```


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

```

function pdde_4.m

% This function implements the estimation of latent input connectivity in
% a basic delay differential equation ERP model
% Copyright (C) Dirk Ostwald
% -----
% 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_fm_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; % poststimulus time (base)
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.ims = -10; % system input onset (ms)
m_f.dur = 1; % duration (width) of the input function
m_f.x = spzeros(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 = [0.02 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 receptor 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.A1 = [0 0 0]; % forward connectivity
m_f.A2 = [0 0 0]; % backward connectivity
m_f.A3 = zeros(2,2); % lateral connectivity
m_f.C = NaN*NaN'; % input connectivity

% forward model structure and parameters
% -----
% load an existing dipole model
load(fullfile(pwd, 'Data\BEM(dip_model_struct.mat)'))
% forward model specification
m_g.vol = dip_model_struct.vol; % sensor space specification
m_g.sens = dip_model_struct.sens; % electrode labels
m_g.elab = dip_model_struct.sens_label; % number of electrodes
m_g.X0 = zeros(m_f.ns*m_g.ne, 3*m_g.ne); % confound design matrix
m_g.ipos = [2 41 54 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 -0.46 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 variances parameter setting theta
mu_sigsqr = [4e-11]; % 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 mu_theta optimization
max_k = 8; % maximal number of Newton iterations for (mu_sigsqr, s_sigsqr) optimization
delta = 1e-10; % Heaviside modification constant
c = 1e-4; % backtracking constant c
rho = 0.9; % backtracking constant rho
varsqr = 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-11; % true, but unknown, data variance
p = length(theta); % number of parameters
h_theta_fg = exp_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_s1] = exp_f(m_f.theta);
f_theta_f_s1 = f_theta_f_s1(1:2:end-1);
f_theta_f_s2 = f_theta_f_s1(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('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('f(1)', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('f(1)', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-1 0.1])

subplot(2,3,3)
plot(m_f.pst,f_theta_f_s2)
xlim(m_f.pst(1),m_f.pst(end));
title('f(2)', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('f(2)', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-1 0.1])

% observed variables evolution
% -----
subplot(2,3,3)
imagesc(m_f.pst, 1:numel(m_g.elab),spm_uvvec(y, NaN(m_g.ne,m_f.nt)))
xlim(m_f.pst(1),m_f.pst(end));
cb = colorbar;
title('V', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel(cb, 'UV', '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_uvvec(y, NaN(m_g.ne,m_f.nt));
plot(m_f.pst,data(eoi,:))
xlim(m_f.pst(1),m_f.pst(end));
title('V', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
legend(eoiLab)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('UV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% -----
% VBML Iteration 0
% -----
% initialization of the variational parameters to prior parameters
mu_theta = mu_theta;
s_theta = sig_theta;
mu_sigsqr = mu_s1_sigsqr;
vfearg_y = y;
vfearg_m_f = m_f;
vfearg_m_g = m_g;
vfearg_mu_theta = mu_theta;
vfearg_s_theta = s_theta;
vfearg_mu_sigsqr = mu_s1_sigsqr(1);
vfearg_s1_sigsqr = mu_s1_sigsqr(2);
vfearg_theta = theta;

% parameter iteration arrays
s_theta_1 = NaN(p,max_i-1);
mu_theta_1 = NaN(1,max_i-1);
h_theta_1 = NaN(m_g.ne,max_i-1);
mu_sigsqr_k = NaN(2,max_k,max_i-1);
F = NaN(max_i-1);

% initialization and initial evaluation of the negative free energy
F(1) = -vF(mu_theta, s_theta, mu_sigsqr, vfearg);

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

```

```

% User update
fprintf('Fixed-form VB Newton Iteration %d of ... \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) exp_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 + (spm_inv(sig_theta))));

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

% numerical gradient and Hessian of the negative free energy
DP_m_theta = full(spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, 1));
dDP_m_theta = full(spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, 1));

% Newton iterations 1,2,3,...
if norm(DP_m_theta) > vareps
    for j = 2:max_j
        % gradient search direction
        p_j = -dDP_m_theta/dF_m_theta;

        % Hessian modification if p_j is not a descent direction
        if -(p_j'*DP_m_theta < 0)
            % diagonal Hessian modification based on spectral decomposition
            dDP_m_theta = dDP_m_theta + max(0, (delta - min(eig(dDP_m_theta))))*eye(2);

            % re-evaluate Newton search direction
            p_j = -dDP_m_theta/dF_m_theta;
        end

        % backtracking evaluation of the step length
        t_j = backtrack_vFE(m_theta, s_theta, ms_sigsqr, vfearg, DP_m_theta, p_j, c_rbo);

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

        % numerical gradient and Hessian of the negative free energy
        DP_m_theta = full(spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, 1));
        dDP_m_theta = full(spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, ms_sigsqr, vfearg), m_theta, 1));

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

        if norm(DP_m_theta) < vareps || norm(m_theta_j(j,1:i-1) - m_theta_j(j-1,1: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(i,1:i-1) = ms_sigsqr;
F_ms_sigsqr_k(i,1:i-1) = vFE(m_theta, s_theta, ms_sigsqr, vfearg);

% evaluation of the current negative free energy gradient and Hessian wrt ms_sigsqr
DP_ms_sigsqr = full(spm_cat(spm_diff(@ms_sigsqr) vFE(m_theta, s_theta, ms_sigsqr, vfearg), ms_sigsqr, 1));
dDP_ms_sigsqr = full(spm_cat(spm_diff(@ms_sigsqr) vFE(m_theta, s_theta, ms_sigsqr, vfearg), ms_sigsqr, 1));

% Newton iterations 1,2,3,...
if norm(DP_ms_sigsqr) > vareps
    for k = 2:max_k
        % Newton search direction
        p_k = -dDP_ms_sigsqr/dF_ms_sigsqr;

        % Hessian modification if p_k is not a descent direction
        if -(p_k'*DP_ms_sigsqr < 0)
            % diagonal Hessian modification based on spectral decomposition
            dDP_ms_sigsqr = dDP_ms_sigsqr + max(0, (delta - min(eig(dDP_ms_sigsqr))))*eye(2);

            % re-evaluate Newton search direction
            p_k = -dDP_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, DP_ms_sigsqr, p_k, c_rbo);

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

        % update remaining entries in free energy arrays
        DP_ms_sigsqr = full(spm_cat(spm_diff(@ms_sigsqr) vFE(m_theta, s_theta, ms_sigsqr, vfearg), ms_sigsqr, 1));
        dDP_ms_sigsqr = full(spm_cat(spm_diff(@ms_sigsqr) vFE(m_theta, s_theta, ms_sigsqr, vfearg), ms_sigsqr, 1));

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

        if norm(DP_ms_sigsqr) < vareps || norm(ms_sigsqr_k(k,1:i-1) - ms_sigsqr_k(k-1,1: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 Visualisation
% -----
% initialize figure
h = figure;
set(h, 'Color', [1 1 1]);

% m_theta prior and approximate posterior
% -----
res = 20;
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) = wppdf(m_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) = wppdf(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, 'yticks', (1.999 1.000 1.001 1.002 1.003))

```

```

title('C', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
set(gca, 'FontSize', 20)
colorbar;
axis([c(1) c(2) end] c(1) c(1) end))
axis square

% sigma^2 prior and posterior
% -----
sigqr_pri = linspace(1e-3, 6e2, 2e2);
p_sigqr_pri = logpdf(sigqr_pri, mu_si_sigqr(1), sqrt(mu_si_sigqr(2)));
sigqr_pos = linspace(1e-6, 0.2, 1e2);
p_sigqr_pos = logpdf(sigqr_pos, mu_sigqr(1), sqrt(mu_sigqr(2)));

subplot(2,3,3)
hold on
line(sigqr_pri, p_sigqr_pri, 'Color', 'b')
ax1.XColor = 'g'; % current axes
ax1.YColor = 'b';
ax1.FontName = 'Calibri Light';
ax1.FontSize = 20;
ax1_pos = ax2.Position;
xlabel('\sigma^2', 'FontName', 'Calibri Light', 'FontSize', 24, 'Color', 'k')
ax2.XColor = 'ax'; % axes('Position', ax2_pos, 'XAxisLocation', 'top', 'YAxisLocation', 'right', 'Color', 'None', 'FontName', 'Calibri Light', 'FontSize', 20);
ax2.YColor = 'r';
ax2.YColor = 'r';
line(sigqr_pos, p_sigqr_pos, 'Parent', ax2, 'Color', 'r')

% -----
% m_theta variational free energy ascent on first VBML iteration
% -----
res = 3e1;
m_theta_max = 1.5;
m_theta_min = 0.5;
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 = 0.1;
m_theta_2_res = res;
m_theta_2_space = linspace(m_theta_2_min, m_theta_2_max, m_theta_2_res);
VFR_m_theta = NaN(res, res);
for i = 1:res
    for j = 1:res
        VFR_m_theta(i, j) = VFR([m_theta_1_space(i) m_theta_2_space(j)], a_theta_1(i, 1), mu_si_sigqr, vfearg);
    end
end
subplot(2,3,4)
hold on
surf(m_theta_2_space, m_theta_1_space, -VFR_m_theta, 'EdgeColor', 'None')
plot3(m_theta_1(2:1), m_theta_1(1:1), -VFR_m_theta_1(1,1), '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', 'FontSize', 20, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% -----
% m_sigqr variational free energy ascent on first VBML iteration
% -----
m_sigqr_max = 5;
m_sigqr_min = 4;
m_sigqr_res = res;
m_sigqr_space = linspace(m_sigqr_min, m_sigqr_max, m_sigqr_res);
m_sigqr_max = 1e-6;
m_sigqr_min = 1;
m_sigqr_res = res;
m_sigqr_space = linspace(m_sigqr_min, m_sigqr_max, m_sigqr_res);
VFR_m_sigqr = NaN(res, res);
for i = 1:res
    for j = 1:res
        VFR_m_sigqr(i, j) = VFR([m_theta_1(2:1), m_theta_1(1:1)], [m_sigqr_space(i) m_sigqr_space(j)], vfearg);
    end
end
subplot(2,3,5)
hold on
surf(m_sigqr_space, m_sigqr_space, -VFR_m_sigqr, 'EdgeColor', 'None')
plot3(m_sigqr_k(2:1), m_sigqr_k(1:1), -VFR_m_sigqr_k(1,1), 'ko', 'MarkerFaceColor', 'k')
axis tight
alpha(6)
view([-45 77])
xlabel('m_sigqr', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('m_theta', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
title('m_sigqr', 'FontSize', 20, 'FontName', 'Calibri Light')

% -----
% variational free energy
subplot(2,3,6)
hold on
plot([0 2] end, 'ko', 'MarkerFaceColor', 'k')
line([0 max_i])
title('m_theta', 'FontSize', 24, 'FontName', 'Calibri Light')
xlabel('m_theta', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

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

% -----
% -----
% -----
function [h_theta_fg] = exp_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 BBPs
% 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] = exp_f(m_f, theta);
g_theta_g = exp_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] = exp_f(m_f, vartbeta)
% This function integrates the neural evolution function of the delay
% differential equation model for BBPs. It is based on spm_gen_exp.m of
% the SPM2 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.G = m_f.G;
theta.T = m_f.T;
theta.d = m_f.d;
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
scale = ((1/m_f.m) * m_f.dt) * 1000;
delay = m_f.delay(1) * theta.R(1,1);
scale = m_f.delay(2) * exp(theta.R(1,2));
u = 12 * exp(-t - delay) / (2 * scale^2);

% integrate system
x = m_f.x; % initial condition
dx = m_f.f; % function handle @spm_fx_exp
dt = m_f.dt; % integration time bin
[fx, dfdx, D] = [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)); %
m = spm_invg(h); %
Q = (spm_exp(dt * D * dfdx / N) - spm_eye(n,n)) * spm_inv(dfdx); %
y = spm_vec(x); % initialize state
y = NaN(length(y), length(u)); % initialize state time-course

% cycle over time-steps
for i = 1:size(u,1)
    % update dx = (exp(dt * Q) - I) * inv(J) * f(x,u)
    for j = 1:N
        v = v + Q * (v, u(i), theta, m_f);
    end
    v(i-1) = v;
end

```

```

end

% transpose
f_theta_f = Y';
end

function [F,J,Q] = spm_fx_esp(x,u,P,M)
% state equations for a neural mass model of erp
% FORMAT [F,J,Q] = spm_fx_esp(x,u,P,M)
% FORMAT [F,J] = spm_fx_esp(x,u,P,M)
% FORMAT [F] = spm_fx_esp(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) hyperpolarizing
% x(8) - current (inhibitory interneurons) depolarizing
% x(9) - voltage (pyramidal cells)
%
% f = dx(t)/dt = f(x(t))
% J = df(x)/dx(t)
% D = delay operator dx(t)/dt = f(x(t-d)) = D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
% M.pP.R = [32 16 4]; % intrinsic rates (forward, backward, lateral)
% M.pP.H = [1 4/8 1/4 1/4]*128; % intrinsic rates (g1, g2, g3, g4)
% M.pP.D = [2 16]; % propagation delays (intrinsic, extrinsic)
% M.pP.G = [4 32]; % receptor densities (excitatory, inhibitory)
% M.pP.T = [8 16]; % synaptic constants (excitatory, inhibitory)
% M.pP.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_esp.m 5369 2013-03-28 20:09:272 karl 8
%
% get dimensions and configure state variables
%-----
n = length(P.A{1}); % number of sources
x = spm_vecvec(x,M,n); % neuronal states
%-----
% [default] fixed parameters
%-----
E = [1 1/2 1/8 1/32]; % intrinsic rates (forward, backward, lateral)
G = [1 4/8 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(9,1)*G;
% no exponential transforms to foster parameter identifiability
%-----
A{1} = P.A{1}*R{1};
A{2} = P.A{2}*R{2};
A{3} = P.A{3}*R{3};
G = G.*C;
% intrinsic connectivity and parameters
%-----
% We = 1/(1000*exp(P.T{1})); % excitatory time constants
% TI = 2/(1000*exp(P.T{2})); % inhibitory time constants
% Ha = H{1}*exp(P.G{1}); % excitatory receptor density
% Hi = H{2}*exp(P.G{2}); % inhibitory receptor density
% pre-synaptic inputs: a(V)
%-----
S = 1./(1 + exp(-R{1}*x - R{2}))) - 1./(1 + exp(R{1}*R{2}));
% exogenous input
%-----
U = C*u{1}^2;
% State: f(x)
%-----
f(1) = x(1); % voltage change (spiny stellate cells)
f(2) = x(1); % positive voltage change (pyramidal cells) +ve
f(3) = x(1); % negative voltage change (pyramidal cells) -ve
f(4) = (Hs.*(A{1} + A{3})*S(1) + G(1,1)*S(1,9) + U) - 2*x(1,4) - x(1,1)/T(1,7); % current change (spiny stellate cells) depolarizing
f(5) = (Hs.*(A{2} + A{3})*S(2) + G(2,1)*S(1,1)) - 2*x(1,5) - x(1,2)/T(1,7); % current change (pyramidal cells) depolarizing
f(6) = (Hs.*(A{1} + A{3})*S(1) - 2*x(1,6) - x(1,3)/T(1,7)); % current change (pyramidal cells) hyperpolarizing
f(7) = x(1); % voltage change (inhibitory interneurons)
f(8) = (Hs.*(A{2} + A{3})*S(1,9) + G(1,3)*S(1,9)) - 2*x(1,8) - x(1,7)/T(1,7); % current change (inhibitory interneurons) depolarizing
f(9) = x(1); % voltage (pyramidal cells)
%-----
% vectorize
f = spm_vec(f);
% avoid infinite recursion of spm_diff
if nargin < 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))
%-----
J(d) = -Q(d)/df/dx
%-----
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - apeak(m,n)).*De;
Di = (1 - apeak(m,n)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,apeye(m,n));
D = De + Di;
% Implement: dx(t)/dt = f(x(t-d)) = inv(1 + D.*dfdx)*f(x(t))
%-----
C = C.*Q^T*x(t);
Q = spm_inv(apeye(length(D)) + D.*J);
end

function [g_theta_g] = exp_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.L = m_g.L;
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_channels,1); % number of electrodes
dp = ie.*theta.L; % dipole coordinates adjusted for fieldtrip
% compute canonical dipole lead-field using Fieldtrip
L_can = NaN(ne,nd);
for i = 1:nd
L_can(i,1:i) = ft_compute_leadfield(dp(i,:), m_g.sens, m_g.vol);
end
% rescale lead-field according to SPM
L_can = L_can/100; % 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
end
function [vFE] = vFE(m_theta,theta,m_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
% mu_sigqr : 2 x 1 Variational parameters for sigma^2
% vfearg : additional argument structure with fields
%
% y : 0 x 1 array - data
% h : string - function handle
% mu_theta : scalar prior expectation for theta
% sig_theta : scalar prior variance for theta
% mu_sigqr : scalar prior scale parameter for sigma^2
% si_sigqr : 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
mu_sigqr = mu_sigqr(1);
s_sigqr = mu_sigqr(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_sigqr = vfearg.mu_sigqr;
si_sigqr = vfearg.si_sigqr;
theta = vfearg.theta;

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

% evaluate h(mu_theta) and 2^h(mu_theta)
h_mu_theta = exp(h(m_f,m_g,spe_uvec(mu_theta, theta)));
Jh_mu_theta = full(spe_uvec(dif(h(mu_theta) exp(h(m_f,m_g,spe_uvec(mu_theta, theta)),mu_theta,1))););

% evaluation of the variational free energy value
T1 = -(n/2)*log(2*pi) - (1/2)*mu_sigqr - (1/2)*exp(-mu_sigqr + 5*mu_sigqr)*((y - h_mu_theta)**(y - h_mu_theta) + trace((Jh_mu_theta*(Jh_mu_theta)*s_theta));
T2 = -(p/2)*log(2*pi) - (1/2)*log(det(sig_theta)) - (1/2)*((mu_theta - mu_theta)**inv(sig_theta)*(mu_theta - mu_theta) + trace(sig_theta*s_theta));
T3 = -(1/2)*log(2*pi*si_sigqr) - mu_sigqr - (1/2)*(1/si_sigqr)*(mu_sigqr + mu_sigqr - mu_sigqr)^2);
T4 = (1/2)*log(det(s_theta)) + (p/2)*log(2*pi*exp(1));
T5 = (1/2) + (1/2)*log(2*pi*s_sigqr) + mu_sigqr;

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

end

function [t_k] = backtrack_vfe_mu_theta(x_k, s_theta, mu_sigqr, vfearg, 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 : p x 1 array of function argument
% s_theta : p x p array of additional VFE arguments
% mu_sigqr : 2 x 1 array of additional VFE arguments
% vfearg : additional input for VFE
% df_x_k : p 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, mu_sigqr, vfearg);
%
% initialise 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, mu_sigqr, 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, mu_sigqr, vfearg);
end

function [t_k] = backtrack_vfe_mu_sigqr(mu_theta, s_theta, x_k, vfearg, 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
% mu_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(mu_theta, s_theta, x_k, vfearg);
%
% initialise step-size
t_k = 1;
%
% evaluation of f(x_{k+1}) given the initial step size
f_x_k_p_1 = VFE(mu_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(mu_theta, s_theta, x_k+t_k*p_k, vfearg);
end
end

```

end

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

```

function pdde_5_k1
% 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,'globalstream')

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

% -----
% Model Formulation
% -----
% neural model structure and parameters
% -----
% structural aspects and fixed parameters
m_f.f = @spm_fm_erp; % neural evolution function
m_f.na = 200; % number of time bins
m_f.dt = 0.001; % time bin length (sec)
m_f.pst = (0:m_f.na-1)*m_f.dt; % poststimulus time (msec)
m_f.nt = length(m_f.pst); % number of time points
m_f.m = 2; % number of neural masses
m_f.ms = 9; % number of neural states per neural mass
m_f.ona = -10; % system input onset (ms)
m_f.dur = 1; % duration (width) of the input function
m_f.x = sparses(m_f.n,m_f.m); % initial condition
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 receptor densities
m_f.R = [0.02 -0.07; 0.18 -0.17]; % source-independent intrinsic connectivity parameters
m_f.D = [-0.82 -0.06 0 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(pwd, '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.no = numel(m_g.ne); % number of dipoles
m_g.lpos = [42 -31 58; 54 -22 18]; % dipole coordinate specifications
m_g.f = [NaN NaN NaN; 0.17 -0.30 -0.17]; % dipole moment specification
m_g.J = [-0.1 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
sigqr = 1e-1; % true, but unknown, data variance
n = m_f.nt*m_g.ne; % number of data points

% prior formulation
% -----
mu_theta = zeros(p,2); % prior expectation parameter setting theta
sig_theta = [1e-3 0 0 1e3 0; 0 0 1e3]; % prior variance parameter setting theta
m_s_sigqr = [4;6]; % 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
delta = sqrt(eps); % maximal number of Newton iterations for (m_sigqr, s_sigqr) optimization
c = 1e-4; % Hessian modification constant
rho = 0.9; % backtracking constant rho
varqps = 1e0; % gradient norm convergence criterion
varqbeta = 1e-3; % parameter norm change convergence criterion

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

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

% evaluate latent data
[f_theta_f_s1] = exp_f(m_f);
f_theta_f_s1 = f_theta_f_s1(1:12:end-1);
f_theta_f_s2 = f_theta_f_s1(1:2:2: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')
ylabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
xlabel('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('f_theta_f_s1', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
xlabel('u', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-10 40])

subplot(2,3,3)
plot(m_f.pst,f_theta_f_s2)
xlim(m_f.pst(1) m_f.pst(end))
title('f_theta_f_s2', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
xlabel('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('UMV', '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)
eol = [47 49:51];
eolab = m_g.elab(eol);
data = spm_unvec(y, NaN(m_g.ne,m_f.nt));
plot(m_f.pst,data(eol,:))
xlim(m_f.pst(1) m_f.pst(end))
title('y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
xlabel('UMV', 'Rotation', 0, 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% -----
% VBM Iteration 0
% -----
% initialization of the variational parameters to prior parameters
h_theta = mu_theta;
m_s_theta = sig_theta;
m_s_sigqr = mu_s_sigqr;
vfeary_y = y;
vfeary_m_f = m_f;
vfeary_m_g = m_g;
vfeary_h_theta = h_theta;
vfeary_sig_theta = sig_theta;
vfeary_mu_sigqr = mu_s_sigqr(1);
vfeary_s1_sigqr = mu_s_sigqr(2);
vfeary_theta = theta;

% parameter iteration arrays
s_theta_1 = NaN(1,p,max_i-1);
p_theta_1 = NaN(1,max_i-1);
m_s_theta_j = NaN(p,max_j,max_i-1);
p_m_theta_j = NaN(max_i,max_i-1);
mu_sigqr_k = NaN(2,max_k,max_i-1);
p_mu_s_sigqr_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, m_s_sigqr, vfeary);

% VBM iterations 1,2,3,...

```

```

% -----
% -----
for i = 2:max_1
% User update
% -----
% 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 ... ')
[m_theta, ...] = full_spm_cat(spm_diff(@m_theta) spm_h(m_f, m_g, spm_unvec(m_theta, theta)), m_theta, 1);
m_theta = spm_inv(spm*(ms_sigqr(1)+1/2)*ms_sigqr(2))\([m_theta, theta] + (spm_inv(sig_theta)));
% save iterand and objective function
s_theta_1(i,1:2) = s_theta;
F_s_theta_1(i-1) = vFE(m_theta, s_theta, ms_sigqr, vfeary);
% inform user
fprintf('finished.\n');
% Newton line search for theta_f variational expectation parameter
% -----
% Estimating q(m_theta) expectation ... ')
% save iterand and objective function
m_theta_j(i,1:2) = m_theta;
F_m_theta_j(i,1:2) = vFE(m_theta, s_theta, ms_sigqr, vfeary);
% 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_sigqr, vfeary), m_theta, 1);
dDF_m_theta = full_spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, ms_sigqr, vfeary), m_theta, 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, s_theta, ms_sigqr, vfeary, 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_sigqr, vfeary), m_theta, 1);
dDF_m_theta = full_spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, ms_sigqr, vfeary), m_theta, 1);
% record update
m_theta_j(j,1:2) = m_theta;
F_m_theta_j(j,1:2) = vFE(m_theta, s_theta, ms_sigqr, vfeary);
if norm(DF_m_theta) < vareps || norm(m_theta_j(i,j,1:2) - m_theta_j(i,j-1,1:2)) < vardelta
break
end
end
% inform user
fprintf('finished.\n');
% Newton line search for sigma^2 variational parameters
% -----
% Estimating q(sigma^2) parameters ... ')
% save iterand and objective function
ms_sigqr_k(i,1:2) = ms_sigqr;
F_ms_sigqr_k(i,1:2) = vFE(m_theta, s_theta, ms_sigqr, vfeary);
% evaluation of the current negative free energy gradient and Hessian wrt ms_sigqr
DF_ms_sigqr = full_spm_cat(spm_diff(@ms_sigqr) vFE(m_theta, s_theta, ms_sigqr, vfeary), ms_sigqr, 1);
dDF_ms_sigqr = full_spm_cat(spm_diff(@ms_sigqr) vFE(m_theta, s_theta, ms_sigqr, vfeary), ms_sigqr, 1);
% Newton iterations 1,2,3,...
if norm(DF_ms_sigqr) > vareps
for k = 2:max_k
% Newton search direction
p_k = -dDF_ms_sigqr/dF_ms_sigqr;
% Hessian modification if p_k is not a descent direction
if -(p_k'*DF_ms_sigqr < 0)
% diagonal Hessian modification based on spectral decomposition
dDF_ms_sigqr = dDF_ms_sigqr + max(0, delta - min(eig(dDF_ms_sigqr))) * eye(2);
% re-evaluate Newton search direction
p_k = -dDF_ms_sigqr/dF_ms_sigqr;
end
% backtracking evaluation of the step length
t_k = backtrack_vFE_ms_sigqr(m_theta, s_theta, ms_sigqr, vfeary, DF_ms_sigqr, p_k, c_rho);
% perform Newton update
ms_sigqr = ms_sigqr + t_k*p_k;
% update remaining entries in free energy arrays
DF_ms_sigqr = full_spm_cat(spm_diff(@ms_sigqr) vFE(m_theta, s_theta, ms_sigqr, vfeary), ms_sigqr, 1);
dDF_ms_sigqr = full_spm_cat(spm_diff(@ms_sigqr) vFE(m_theta, s_theta, ms_sigqr, vfeary), ms_sigqr, 1);
% record update
ms_sigqr_k(k,1:2) = ms_sigqr;
F_ms_sigqr_k(k,1:2) = vFE(m_theta, s_theta, ms_sigqr, vfeary);
if norm(DF_ms_sigqr) < vareps || norm(ms_sigqr_k(i,k,1:2) - ms_sigqr_k(i,k-1,1:2)) < vardelta
break
end
end
% inform user
fprintf('finished.\n');
% free energy update
% -----
% F(i) = vFE(m_theta, s_theta, ms_sigqr, vfeary);
% user update
% -----
fprintf('vFE %6.3f dVFE %6.3f \n', F(i), F(i) - F(i-1));
end
% -----
% Algorithm Visualization
% -----
% initialise figure
h = figure;
set(h, 'Color', [1 1 1]);
% m_theta prior and approximate posterior
% -----
subplot(2,3,1)
hold on
plot(m_theta(1), m_theta(2), m_theta(3), 'ko', 'MarkerFaceColor', 'k')
lambda = eig(sig_theta);
[xy, 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(xy, 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);
title(p('theta_m_1[1]'), 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal');
set(gca, 'FontSize', 19, 'FontName', 'Calibri Light');
% -----
subplot(2,3,2)
hold on
plot(m_theta(1), m_theta(2), m_theta(3), 'ko', 'MarkerFaceColor', 'k')
lambda = eig(s_theta);
[xy, 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(xy, z, 'EdgeColor', 'None', 'FaceLighting', 'gouraud');
camlight headlight;
zlim([-0.05 0.02]);
xlim([-0.05 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);
zlabel('theta_m_3[1]', 'FontName', 'Calibri Light', 'FontSize', 20);
title(q('theta_m_1[1]'), 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal');
% -----
% sigma^2 prior and posterior
% -----
sigqr_pri = linspace(1e-3, 6e2, 1e2);
p_sigqr_pri = logpdf(sigqr_pri, ms_sigqr(1), sqrt(ms_sigqr(2)));
sigqr_pos = linspace(1e-4, 0.2, 1e2);
p_sigqr_pos = logpdf(sigqr_pos, ms_sigqr(1), sqrt(ms_sigqr(2)));
subplot(2,3,3)
hold on
line(sigqr_pri, p_sigqr_pri, 'Color', 'b');
axl = gca; % current axes
axl.XColor = 'b';

```

```

ax1.HColor = '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,'Position','top','Position','right','Color','none','FontName','Calibri Light','FontSize',20);
ax2.HColor = 'r';
line(sigqr_pos,p_sigqr_pos,'Parent',ax2,'Color','r');

% m_theta variational free energy ascent on first VBML 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_1(3,1,1) m_theta_1(1,1,1) mu_si_sigqr vtearg]);
    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_1(2,1,1),m_theta_1(1,1,1),VFE_m_theta_1(1,1),'k','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','FontSize','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_1(3,2,1) m_theta_1(1,1,1) mu_si_sigqr vtearg]);
    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_1(2,2,1),m_theta_1(1,2,1),VFE_m_theta_1(2,1),'k','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','FontSize','Normal','FontSize',24);
set(gca,'FontSize',20,'FontName','Calibri Light');

% variational free energy
subplot(2,3,6)
hold on
plot([2:end], 'k','MarkerFaceColor','k')
xlim([0 max(i)])
title('m_theta^{[i]}, sigma_theta^{[i]}, m[sigma^2]^{[i]}, m[sigma^2]^{[i]}','FontName','Calibri Light','FontSize','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] = exp_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 EBPB
% 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] = exp_f_m_f;
[G_theta_g] = exp_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] = exp_f_m_f
% This function integrates the neural evolution function of the delay
% differential equation model for EBPB. It is based on spm_gen_exp.m of
% the SPM2 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
u = ((1/m_f.se)*m_f.dt)*1000;
delay = m_f.delay*(1+2*theta.S(1,1));
scale = m_f.scale*(1+exp(theta.R(1,2)));
u = 12*exp(-(t - delay)./2/(2*scale^2));
% integrate system
x = m_f.x; % initial condition
dt = m_f.dt; % function handle spm_fx_exp
[t,dfdx,D] = [x,u(1),theta_m_f]; % integration time bin
D = max(abs(real(eig(dfdx))))); % dx(t)/dt and Jacobian df/dx and check for delay operator
N = ceil(max(1, dt*p^2)); %
M = spm_length(N); %
Q = [spm_exp(dt*D*dfdx/N) - speye(n,n)]*spm_inv(dfdx); %
Y = spm_vec(u); % initialise state
Y = NaN(length(Y), length(u)); % initialise state time-course

% cycle over time-steps
for i = 1:size(u,1)
    % update dx = expm(dt*Q) - I*inv(J)*f(x,u)
    for j = 1:N
        Y = Y + Q*(Y,u(i),theta_m_f);
    end
    Y(i) = Y;
end

% transouse
f_theta_f = Y;

end

function [f,T,D] = spm_fx_exp(x,u,p,M)
% state equations for a neural mass model of args
% FORMAT [f,T,D] = spm_fx_exp(x,u,p,M)
% PARAM [f] = spm_fx_exp(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(x)/dx(x)
% D - delay operator dx(t)/dt = f(x(t-d))
%
% Prior fixed parameter scaling (Defaults)
%
% M.p.R = [32 16 4]; % extrinsic rates (forward, backward, lateral)
% M.p.R = [1 4/8 1/4 1/4]*128; % intrinsic rates (q1, q2 q3, q4)
% M.p.D = [2 16]; % propagation delays (intrinsic, extrinsic)
% M.p.G = [4 32]; % receptor densities (excitatory, inhibitory)
% M.p.T = [8 16]; % synaptic constants (excitatory, inhibitory)
% M.p.R = [1 1/2]; % parameter of static nonlinearity
%
% David O. Priston K2 (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
% SID: spm_fm_exp.m 5369 2013-03-28 20:09:272 Karl S
%
% get dimensions and configure state variables
%-----
n = length(P_A[1]); % number of sources
N = spm_size(m,M,n); % neuronal states
%-----
% (default) fixed parameters
%-----
R = [1 1/2 1/8 1/32]; % extrinsic rates (forward, backward, lateral)
G = [1 4/8 1/4 1/4]*128; % intrinsic rates (q1, q2, q3, q4)
D = [2 16]; % propagation delays (intrinsic, extrinsic)
T = [4 32]; % receptor densities (excitatory, inhibitory)
% synaptic constants (excitatory, inhibitory)
R = [1 1/2]; % parameter of static nonlinearity
%-----
% test for free parameters on intrinsic connections
%-----
G = G.*exp(P_B);
G = ones(9,1)*G;
% no exponential transforms to foster parameter identifiability
%-----
A[1] = P.A[1]*R[1]; % excitatory time constants
A[2] = P.A[2]*R[2]; % inhibitory time constants
A[3] = P.A[3]*R[3]; % excitatory receptor density
G = P.G;
% intrinsic connectivity and parameters
%-----
T1 = T1/1000*exp(P_T1); % excitatory time constants
T2 = T2/1000*exp(P_T2); % 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]*y)) - 1./(1 + exp(R[1]*R[2]));
% exogenous input
%-----
U = C*u(1)*2;
% State: f(x)
%-----
f(1,1) = x(1,4); % voltage change (spiny stellate cells)
f(1,2) = x(1,5); % positive voltage change (pyramidal cells) -ve
f(1,3) = x(1,6); % negative voltage change (pyramidal cells) -ve
f(1,4) = (He.*([A[1] + A[3]*S(1,9) + G(1,1)*S(1,9) + U] - 2*x(1,4) - x(1,1)/T1)/T1; % current change (spiny stellate cells) depolarizing
f(1,5) = (Hi.*([A[2] + A[3]*S(1,9) + G(1,2)*S(1,9) - 2*x(1,5) - x(1,2)/T2)/T2; % current change (pyramidal cells) depolarizing
f(1,6) = (Hi.*([A[2] + A[3]*S(1,9) + G(1,2)*S(1,9) - 2*x(1,6) - x(1,3)/T1)/T1; % current change (pyramidal cells) hyperpolarizing
f(1,7) = x(1,8); % voltage change (inhibitory interneurons)
f(1,8) = (Hi.*([A[2] + A[3]*S(1,9) + G(1,3)*S(1,9) - 2*x(1,8) - x(1,7)/T2)/T2; % current change/inhibitory interneurons) depolarizing
f(1,9) = x(1,9) - x(1,6); % voltage (pyramidal cells)
% vectorize
f = spm_vec(f);
% avoid infinite recursion of spm_diff
if nargin < 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) = df(x(t))
%
% J(d) = -Q(d)/dx
%-----
De = D(2)*exp(P_D)/1000;
Dt = D(1)/1000;
De = (1 - spexp(-Dt))*De;
Dt = (1 - spexp(-Dt))*Dt;
De = kron(ones(9,9),De);
Dt = kron(Dt,spexp(-Dt));
D = Dt + De;
% Implement dx(t)/dt = f(x(t-d)) = inv(1 + D.*dfdx)*f(x(t))
% C = C.* exp(-Q*d);
%-----
Q = spm_inv(spexp(-length(D))*D.*J);
end
function [g_theta_g] = exp_g(m_g, vartheta)
% This function evaluates the EEG forward model of the delay differential
% equation model for ESPs 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_ipos = m_g.ipos;
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_ipos; % 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 SPW
L_can = L_can*(10^3)*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 = kron(theta_J,L_dip);
end
function [vFE] = vFE(m_theta_s_theta_ms_siggr_vFEarg)
% This function evaluates the variational free energy for ESP-DCM toy
% problem
%
% Inputs
% m_theta : p x 1 variational expectation for theta
% s_theta : p x p variational variance for theta
% ms_siggr : 2 x 1 variational parameters for sigma^2
% vFEarg : additional argument structure with fields
% 'y' : n x 1 array - data
% 'h' : string - function handle
% 'ms_theta' : scalar prior expectation for theta
% 'sig_theta' : scalar prior variance for theta
% 'ms_siggr' : scalar prior scale parameter for sigma^2
% 's_siggr' : scalar prior shape parameter for sigma^2
% 't_theta' : true, but unknown, parameter structure
%
% Outputs
% vFE : scalar negative variational free energy
%
% Copyright (C) Dirk Ostwald
%-----
% unpack input structure
ms_siggr = ms_siggr(1);
s_siggr = ms_siggr(2);

```

```

y = vfeary.y;
m_f = vfeary.m_f;
m_g = vfeary.m_g;
mu_theta = vfeary.mu_theta;
sig_theta = vfeary.sig_theta;
mu_sigqr = vfeary.mu_sigqr;
s1_sigqr = vfeary.s1_sigqr;
theta = vfeary.theta;

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

% evaluate h(mu_theta) and J^T h(mu_theta)
h_mu_theta = exp(h_m_f_m_g_spm_uvec(mu_theta, theta));
Jh_mu_theta = full(spm_cat(spm_diff(s(mu_theta)) exp(h_m_f_m_g_spm_uvec(mu_theta, theta)), mu_theta, 1));

% evaluation of the variational free energy value
T1 = -(n/2)*log(2*pi) - (n/2)*mu_sigqr - (1/2)*exp(-mu_sigqr + 5*s1_sigqr)*(y - h_mu_theta)**(y - h_mu_theta) + trace((Jh_mu_theta'*Jh_mu_theta)*mu_theta);
T2 = -(p/2)*log(2*pi) - (1/2)*log(det(sig_theta)) - (1/2)*(mu_theta - mu_theta)**inv(sig_theta)*(mu_theta - mu_theta) + trace(sig_theta*a_theta);
T3 = -(1/2)*log(2*pi*s1_sigqr) - mu_sigqr - (1/2)*(1/s1_sigqr)*(s1_sigqr + mu_sigqr - mu_sigqr**2);
T4 = (1/2)*log(det(a_theta)) + (p/2)*log(2*pi*exp(1));
T5 = (1/2) + (1/2)*log(2*pi*s1_sigqr) + mu_sigqr;

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

end

function [VFE] = vfe_terms(mu_theta, s_theta, mu_sigqr, vfeary)

% This function evaluates the variational free energy for ESP-DCM toy
% problem
%
% Inputs
% mu_theta : 2 x 1 variational expectation for theta
% s_theta : 2 x 2 variational variance for theta
% mu_sigqr : 2 x 1 variational parameters for sigma^2
% vfeary : additional argument structure with fields
% y : n x 1 array - data
% Jh : string - function handle
% mu_theta : scalar prior expectation for theta
% s1_theta : scalar prior variance for theta
% mu_sigqr : scalar prior scale parameter for sigma^2
% s1_sigqr : 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 mu_theta
% .T3 : scalar prior influence mu_sigqr
% .T4 : scalar entropy of theta
% .T5 : scalar entropy of sigma^2
% .VFE : scalar variational free energy
%
%
% Copyright (C) Dirk Ostwald
% -----
% Unpack input structure
mu_sigqr = mu_sigqr(1);
mu_sigqr = mu_sigqr(2);
y = vfeary.y;
m_f = vfeary.m_f;
m_g = vfeary.m_g;
mu_theta = vfeary.mu_theta;
sig_theta = vfeary.sig_theta;
mu_sigqr = vfeary.mu_sigqr;
s1_sigqr = vfeary.s1_sigqr;
theta = vfeary.theta;

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

% evaluate h(mu_theta) and J^T h(mu_theta)
h_mu_theta = exp(h_m_f_m_g_spm_uvec(mu_theta, theta));
Jh_mu_theta = full(spm_cat(spm_diff(s(mu_theta)) exp(h_m_f_m_g_spm_uvec(mu_theta, theta)), mu_theta, 1));

% evaluation of the variational free energy value
VFE_T1 = -(n/2)*log(2*pi) - (n/2)*mu_sigqr - (1/2)*exp(-mu_sigqr + 5*s1_sigqr)*(y - h_mu_theta)**(y - h_mu_theta);
VFE_T2 = -(1/2)*exp(-mu_sigqr + 5*s1_sigqr)*trace((Jh_mu_theta'*Jh_mu_theta)*mu_theta);
VFE_T3 = -(p/2)*log(2*pi) - (1/2)*log(det(sig_theta)) - (1/2)*(mu_theta - mu_theta)**inv(sig_theta)*(mu_theta - mu_theta) + trace(sig_theta*a_theta);
VFE_T4 = -(1/2)*log(2*pi*s1_sigqr) - mu_sigqr - (1/2)*(1/s1_sigqr)*(s1_sigqr + mu_sigqr - mu_sigqr**2);
VFE_T5 = (1/2)*log(det(a_theta)) + (p/2)*log(2*pi*exp(1));
VFE_T6 = (1/2) + (1/2)*log(2*pi*s1_sigqr) + mu_sigqr;

% 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_mu_theta(x_k, s_theta, mu_sigqr, vfeary, 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 : p x 1 array of function argument
% s_theta : p x p array of additional vfe arguments
% mu_sigqr : 2 x 1 array of additional vfe arguments
% vfeary : 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, mu_sigqr, vfeary);

% initialise 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, mu_sigqr, vfeary);

% 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, mu_sigqr, vfeary);
end

end

function [t_k] = backtrack_vfe_mu_sigqr(mu_theta, s_theta, x_k, vfeary, 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
% mu_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
% vfeary : 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(mu_theta, s_theta, x_k, vfeary);

% initialise step-size
t_k = 1;

% evaluation of f(x_k+1) given the initial step size
f_x_k_p_1 = vfe(mu_theta, s_theta, x_k + t_k*p_k, vfeary);

% 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(mu_theta, s_theta, x_k + t_k*p_k, vfeary);
end

end

```

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

```

function pdde_6_01
% This function implements the estimation of forward connectivity and
% dipole moments in a basic delay differential equation ERP model
%
% Copyright (C) Dirk Ostwald
%
% -----
% 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,'globalstream')

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

% -----
% Model Formulation
% -----
% neural model structure and parameters
% -----
% structural aspects and fixed parameters
m_f_exp = @expm_exp; % neural evolution function
m_f_nse = 200; % number of time bins
m_f_dt = 0.001; % time bin length (sec)
m_f_pst = (0:m_f_nse-1)*m_f_dt; % poststimulus time (msec)
m_f_nt = length(m_f_pst); % number of time points
m_f_nm = 2; % number of neural masses
m_f_fm = 9; % number of neural states per neural mass
m_f_om = -10; % system input onset (ms)
m_f_dur = 1; % duration (width) of the input function
m_f_x = sparse(m_f_nm,m_f_nm); % initial condition
m_f_s = [-9e4 0.08]; % input function parameters rho_1 and rho_2
m_f_T = [0.16 0.13; 0.07 -0.82]; % static nonlinearity (activation function) parameters
m_f_G = [-0.11 0.45; 0.02 -0.63]; % source-specific excitatory and inhibitory time constants
m_f_H = [0.02 -0.07; 0.18 -0.17]; % source-specific excitatory and inhibitory receptor densities
m_f_D = [0 -0.52; -0.56 0]; % source-independent intrinsic connectivity parameters
m_f_A[1] = [0 0; NaN 0]; % between source delay parameters
m_f_A[2] = zeros(2,2); % forward connectivity
m_f_A[3] = zeros(2,2); % backward connectivity
m_f_C = [1 0]; % lateral connectivity
m_f_C = [1 0]; % input connectivity

% forward model structure and parameters
% -----
% load an existing dipole model
load(fullfile(pwd, 'data', 'dip_model_struct.mat'))
m_g_vol = dip_model_struct.vol; % forward model specification
m_g_sens = sensor_space_specification; % sensor space specification
m_g_elab = dip_model_struct.sens.label; % electrode labels
m_g_nme = numel(m_g_elab); % number of electrodes
m_g_co = zeros(m_nm,m_g_nm,3*m_g_nme); % confound design matrix
m_g_ipos = [42 -31 58; 54 -22 18]; % dipole coordinate specifications
m_g_e = NaN*NaN(0,17,-0.50,-0.17); % dipole moment specification
m_g_J = [-0.1 0 0 0 0 0 -0.46 0]; % 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
sigpr = 1e-1; % true, but unknown, data variance
n = m_f_nt*m_g_nme; % number of data points

% prior formulation
% -----
mu_theta = zeros(p,1); % prior expectation parameter setting theta
sig_theta = eye(p)*1e3; % prior variances parameter setting theta
mu_sigsqr = [0,]; % 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 mu_theta optimization
max_k = 10; % maximal number of Newton iterations for mu_sigsqr, mu_sigsqr optimization
delta = sqrt(eps); % Heunian modification constant
c = 1e-4; % backtracking constant c
rho = 0.1; % backtracking constant rho
varage = 1e-1; % gradient norm convergence criterion
vardelta = 1e-3; % parameter norm change convergence criterion

% -----
% Model Realization
% -----
h_theta_fg = exp_h(m_f,m_g,theta); % data expectation
y = mvnrnd(h_theta_fg, sigpr*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_s1] = exp_f(m_f,theta);
f_theta_f_s1 = f_theta_f_s1(1:9,:);
f_theta_f_s2 = f_theta_f_s1(10:16,:);

% 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('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('x^{(1)}', 'FontSize', 24, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')
ylim([-10 40])

subplot(2,3,3)
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('x^{(2)}', '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_nm,m_f_nt)))
xlim(m_f_pst(1),m_f_pst(end))
cb = colorbar;
title('y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
ylabel('y', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')
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);
data = spm_unvec(y, NaN(m_g_nm,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(eoi)
xlabel('Time [s]', 'FontSize', 24, 'FontName', 'Calibri Light')
ylabel('y', 'FontName', 'Calibri Light', 'FontSize', 24)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

% -----
% VBM: Iteration 0
% -----
% initialization of the variational parameters to prior parameters
mu_theta = mu_theta;
sig_theta = sig_theta;
mu_sigsqr = mu_sigsqr;
vfeag_y = y;
vfeag_m_f = m_f;
vfeag_mu_theta = mu_theta;
vfeag_sig_theta = sig_theta;
vfeag_mu_sigsqr = mu_sigsqr(1);
vfeag_sigsqr = mu_sigsqr(2);
vfeag_theta = theta;

% parameter iteration arrays
s_theta_s = NaN(p,max_i-1);
F_s_theta_s = NaN(i,max_i-1);
u_theta_s = NaN(p,max_j,max_i-2);

```

```

F_m_theta_j = NaN(max_j,max_i-1);
ms_sigqr_k = NaN(2,max_k,max_i-1);
F_m_sigqr_k = NaN(max_k,max_i-1);
% initialisation and initial evaluation of the negative free energy
F = NaN(max_i,1);
F(1) = VFE(m_theta, s_theta, ms_sigqr, vfearg);
% -----
% VML iterations 1,2,3,...
% -----
for i = 2:max_i
% User update
% -----
fprintf('Fixed-form VB Newton Iteration %d.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) exp_h(m_f,m_g,spm_vec(m_theta),theta)),m_theta,1));
s_theta = spm_inv((exp(-ms_sigqr(1)+(1/2)*ms_sigqr(2))*(JH_m_theta'*JH_m_theta) + (spm_inv(sig_theta))));
% save iterand and objective function
s_theta_i(i,:,:) = s_theta;
F_m_theta_i(i,:) = VFE(m_theta,s_theta,ms_sigqr,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(i,:,:) = m_theta;
F_m_theta_j(i,:) = VFE(m_theta,s_theta,ms_sigqr,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_sigqr,vfearg),m_theta,1)));
d2F_m_theta = full(spm_cat(spm_diff(@(m_theta) VFE(m_theta,s_theta,ms_sigqr,vfearg),m_theta,1)));
% Newton iterations 1,2,3,...
if norm(dF_m_theta) > vareps
for j = 2:max_j
% gradient search direction
p_j = -d2F_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
d2F_m_theta = d2F_m_theta + max(0,delta - min(eig(d2F_m_theta))))*eye(p);
% re-evaluate Newton search direction
p_j = -d2F_m_theta\dF_m_theta;
end
% backtracking evaluation of the step length
t_j = backtrack_VFE_m_theta(m_theta,s_theta,ms_sigqr,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_sigqr,vfearg),m_theta,1)));
d2F_m_theta = full(spm_cat(spm_diff(@(m_theta) VFE(m_theta,s_theta,ms_sigqr,vfearg),m_theta,1)));
% record updates
m_theta_j(i,j,:) = m_theta;
F_m_theta_j(i,j,:) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
if norm(dF_m_theta*p_j) < vareps || norm(m_theta_j(i,j,:)) - m_theta_j(i,j-1,:) < vardelta
break
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_sigqr_k(i,:,:) = ms_sigqr;
F_ms_sigqr_k(i,:) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
% evaluation of the current negative free energy gradient and Hessian wrt ms_sigqr
dF_ms_sigqr = full(spm_cat(spm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,1)));
d2F_ms_sigqr = full(spm_cat(spm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,1)));
% Newton iterations 1,2,3,...
if norm(dF_ms_sigqr) > vareps
for k = 2:max_k
% Newton search direction
p_k = -d2F_ms_sigqr\dF_ms_sigqr;
% Hessian modification if p_k is not a descent direction
if ~(p_k'*dF_ms_sigqr < 0)
% diagonal Hessian modification based on spectral decomposition
d2F_ms_sigqr = d2F_ms_sigqr + max(0,delta - min(eig(d2F_ms_sigqr))))*eye(2);
% re-evaluate Newton search direction
p_k = -d2F_ms_sigqr\dF_ms_sigqr;
end
% backtracking evaluation of the step length
t_k = backtrack_VFE_ms_sigqr(m_theta,s_theta,ms_sigqr,vfearg,dF_ms_sigqr,p_k,c,rho);
% perform Newton update
ms_sigqr = ms_sigqr + t_k*p_k;
% update remaining entries in free energy arrays
dF_ms_sigqr = full(spm_cat(spm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,1)));
d2F_ms_sigqr = full(spm_cat(spm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,1)));
% record updates
ms_sigqr_k(i,k,:) = ms_sigqr;
F_ms_sigqr_k(i,k,:) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
if norm(dF_ms_sigqr) < vareps || norm(ms_sigqr_k(i,k,:)) - ms_sigqr_k(i,k-1,:) < vardelta
break
end
end
% Inform user
fprintf(' finished. \n')
% free energy update
% -----
F(i) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
% user update
% -----
fprintf('VFE %d.3f dVFE %d.3f \n', F(i), F(i) - F(i-1))
end
% -----
% Algorithm Visualization
% -----
% initialise figure
h = figure;
set(h,'Color',[1 1 1])
% m_theta prior and approximate posterior
% -----
% visualisation 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,lines,line2] = ploty(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));
[xy,z] = allipos(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(xy,z,'EdgeColor','none','FaceLighting','gouraud');
colormap(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)
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(mu_theta(2),mu_theta(3),mu_theta(4),'ko','MarkerFaceColor','k')
lambda = eig(s_theta(2:4,2:4));
[xy,z] = allipos(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(xy,z,'EdgeColor','none','FaceLighting','gouraud');
colormap(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)
title('p(theta_2,theta_3,theta_4)','FontName','Calibri Light','FontSize',24,'FontWeight','Normal')
set(gca,'FontSize',20,'FontName','Calibri Light')

```

```

xlim([-0.05 -0.01])
camlight headlight
set(haur, '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)
xlabel('theta_4', 'FontName', 'Calibri Light', 'FontSize', 24, 'Rotation', 0)
title('u(theta_2, theta_3, theta_4)', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight', 'Normal')

% sigma^2 prior and posterior
%-----
sigqr_pri = invspecie(6, 30, 1e3);
p_sigqr_pri = logpdf(sigqr_pri, mu_sl_sigqr(1), sqrt(mu_sl_sigqr(2)));
sigqr_pos = invspecie(6, 6, 2, 1e3);
p_sigqr_pos = logpdf(sigqr_pos, mu_sigqr(1), sqrt(mu_sigqr(2)));
subplot(2,3,4)
hold on
line(sigqr_pri, p_sigqr_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', 'r');
ax2 = axes('Position', ax1_pos, 'XAxisLocation', 'top', 'YAxisLocation', 'right', 'Color', 'None', 'FontName', 'Calibri Light', 'FontSize', 20);
ax2.XColor = 'r';
ax2.YColor = 'r';
line(sigqr_pos, p_sigqr_pos, 'Parent', ax2, 'Color', 'r')

% variational free energy
%-----
subplot(2,3,[5 6])
hold on
for i = 1:max(1,1)
    plot([p_mu_theta_2(i), 'b-', 'MarkerFaceColor', 'b')
    plot([p_mu_sigqr_A(i), 'r-', 'MarkerFaceColor', 'r')
end
legend('mu(theta)', 'mu(sigma^2)', 'u(sigma^2)')
ylim([-4e3 4e3])
xlim([0.5 0.51])
xlabel('Sub-loop Iteration', 'FontSize', 24, 'FontName', 'Calibri Light')
title('mu(theta) [i], u(sigma^2) [i], u(sigma^2) [i]', 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 20)
set(gca, 'tick', 1:8)
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light')

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

end

%-----
% Subfunctions
function [h_theta_fg] = exp_fm_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 EPDs
%
% 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] = exp_fm_f(theta);
g_theta_g = exp_fm_g(theta);
% evaluate concatenated function h
h_theta_fg = 1e4*spm_vec(g_theta_g*f_theta_f);
end

function [f_theta_f, u] = exp_fm_f(theta)
% This function integrates the neural evolution function of the delay
% differential equation model for EPDs. It is based on spm_gen_exp_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.B = m_f.B;
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 = ((theta_f.mf.mf.dt)*100);
delay = m_f.oms(1)+128*theta.R(1,1);
scale = m_f.dur(1)*exp(theta.R(1,2));
u = 32*exp(-t - delay)*2*(scale^2)';
% integrate system
x = m_f.x; % initial condition
f = m_f.f; % function handles spm_fm_exp
dt = m_f.dt; % integration time bin
[fx, dfdx, D] = [f(x,u), theta, m_f]; % dx(t)/dt and Jacobian df/dx and check for delay operator
D = NaN(size(real(eig(fx)))); %
M = cell(max(1, dt)*2); %
n = spm_nepochs; %
Q = [spm_expn(dt*D*dfdx/n) - spm_eye(n,n)*spm_inv(dfdx)]; %
v = spm_vec(u); % 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*n) - 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
end
si(:,s) = s+ns*9*ns;
% create output
f_theta_f = [];
for s = 1:ns
    [f_theta_f; [f_theta_f; y(si(:,s),:)]];
end
function [f, J, Q] = spm_fm_exp(x, u, P, M)
% state equations for a neural mass model of erps
% FOMMAT [f, J, D] = spm_fm_exp(x, u, P, M)
% FOMMAT [f, J] = spm_fm_exp(x, u, P, M)
% FOMMAT [f] = spm_fm_exp(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 : -dt(t)/dt(t)
% D : delay operator dx(t)/dt = f(x(t-d))
%
% Prior fixed parameter scaling (Defaults)
%
% M.pP.R = [32 16 4]; % extrinsic rates (forward, backward, lateral)
% M.pP.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (sl, st, sp, go)
% M.pP.D = [2 16]; % propagation delays (intrinsic, extrinsic)
% M.pP.O = [4 22]; % receptor densities (excitatory, inhibitory)
% M.pP.T = [8 16]; % synaptic constants (excitatory, inhibitory)
% M.pP.R = [1 1/2]; % parameter of static nonlinearity
%

```

```

% David O. Friston EJ (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_fm_exp.m 5369 2013-03-28 20:09:272 Karl S

% get dimensions and configure state variables
%-----
n = length(P_A(1)); % number of sources
M = spm_vecvec(M,A); % neuronal states

% [default] fixed parameters
%-----
P = [1 1/2 1/8]*2; % extrinsic rates (forward, backward, lateral)
Q = [1 4/5 1/4 1/4]*100; % intrinsic rates (SI sp1 sp2)
D = [2 16]; % propagation delays (intrinsic, extrinsic)
R = [4 32]; % receptor densities (excitatory, inhibitory)
T = [18 16]; % synaptic constants (excitatory, inhibitory)
R = [2 1]*3; % parameters of static nonlinearity

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

% no exponential transforms to foster parameter identifiability
%-----
A(1) = P_A(1)*R(1); % excitatory time constants
A(2) = P_A(2)*R(2); % inhibitory time constants
A(3) = P_A(3)*R(3); % excitatory receptor density
G = P.*C; % inhibitory receptor density

% intrinsic connectivity and parameters
%-----
Tn = T/(1000*exp(P.T(1:3))); % excitatory time constants
Ti = T/(1000*exp(P.T(4:6))); % inhibitory time constants
Ha = H/(1*exp(P.D(-1:1))); % excitatory receptor density
Hi = H/(1*exp(P.D(-2:2))); % 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 = G*(u(1)+2);

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

% vectorize
f = spm_vec(f);

% avoid infinite recursion of spm_diff
if nargin < 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) = -Q(d)df/dx
%-----
De = D(2).*exp(P.D)/1000;
Di = D(1)/1000;
De = (1 - speye(n,1)).*De;
Di = (1 - speye(9,1)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,speye(n,1));
D = Di + De;

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

end

function [g_theta_g] = exp_g(m_g, vartheta)
% This function evaluates the EEG forward model of the delay differential
% equation model for EEPs 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_lpss = m_g.lpss;
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
ns = size(theta_lpss.champs,1); % number of electrodes
dp = 1e-3*theta_lpss; % dipole coordinates adjusted for fieldtrip

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

% compute lead-field according to SNM
L_can = L_can*(10^11)*round(log10(max(max(abs(L_can))))/8)/1;

% evaluate channel predictions based on dipole moments
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_L, L_dip(i,:))];
end
end

function [VFE] = vFE(m_theta, theta_ms_siggr, vfeary)
% This function evaluates the variational free energy for EEP-DCM toy
% problem
%-----
% Inputs
% m_theta : p x 1 variational expectation for theta
% theta_ms_siggr : p x p variational variance for theta
% vfeary : 2 x 1 variational parameters for sigma^2
% vfeary : 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_siggr : scalar prior scale parameter for sigma^2
% sl_siggr : 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
mu_siggr = mu_siggr(1);
sl_siggr = sl_siggr(2);
y = vfeary.y;
mu_f = vfeary.mu_f;
m_g = vfeary.m_g;
mu_theta = vfeary.mu_theta;
sig_theta = vfeary.sig_theta;
mu_siggr = vfeary.mu_siggr;
sl_siggr = vfeary.sl_siggr;
theta = vfeary.theta;

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

% evaluate h(m_theta) and J^2 h(m_theta)
h_m_theta = exp(h(m_f_m_g, spm_vecvec(m_theta, theta)));
Jh_m_theta = full(sgm_sar(spm_diff(h_m_theta) exp_Dir_f_m_g_spm_vecvec(m_theta, theta)), m_theta, 1));

```

```

% evaluation of the variational free energy value
T1 = -(p/2)*log(2*pi) - (1/2)*m_sigsqr - (1/2)*exp(-m_sigsqr)*(y - h_m_theta)**(y - h_m_theta) + trace((Zh_m_theta**Zh_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*\*s_sigsqr) - m_sigsqr - (1/2)*(1/s_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);
% evaluate the negative free energy
VFE = -(T1 + T2 + T3 + T4 + T5);
end

function [t_k] = backtrack_VFE_s_theta(x_k, s_theta, m_sigsqr, vfeary, 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 : p x 1 array of function argument
% s_theta : p x p array of additional VFE arguments
% m_sigsqr : 2 x 1 array of additional VFE arguments
% vfeary : additional input for VFE
% df_x_k : [p,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, m_sigsqr, vfeary);
% initialise 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, m_sigsqr, vfeary);
% 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, m_sigsqr, vfeary);
end
end

function [t_k] = backtrack_VFE_m_sigsqr(m_theta, s_theta, x_k, vfeary, 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
% 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
% vfeary : 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, vfeary);
% initialise 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, vfeary);
% 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, vfeary);
end
end

```

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

```

function pdde_7_m
% This function implements the estimation of condition-specific latent
% input connectivity in a basic delay differential equation ERP model
% Copyright (C) Dirk Ostwald
% -----
% close all
close all

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

% invoke Fieldtrip
sdspath(getpath(fullfile(pwd, 'Fieldtrip')))

% -----
% Model Formulation
% -----
% neural model structure and parameters
% -----
% structural aspects and fixed parameters
m_f.f = @sgm_fm_exp; % neural evolution function
m_f.m = 10; % number of time bins
m_f.dt = 0.002; % time bin length (sec)
m_f.pst = [0;m_f.m-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 = 8; % 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.c = sparsel(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.G = [-8e4 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 receptor 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.K = [0 1]; % design matrix
m_f.A[1] = [0 1]; % forward connectivity
m_f.A[2] = [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(pwd, 'Data\EMG', 'dip_model_struct.mat'))
% -----
% forward model specification
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.elab; % electrode labels
m_g.ne = numel(m_g.elab); % number of electrodes
m_g.d0 = dip_model_struct.m_g.m_g; % confound design matrix
m_g.tpos = [42 31 58 54 -22 18]; % dipole coordinate specifications
m_g.L = [0.02 -0.05 -0.04 0.17 -0.55 -0.17]; % dipole moment specification
m_g.W = [-0.1 0 0 0 0 -0.46 0 1]; % state weighting specification

% likelihood model parameters
% -----
% prior formulation
mu_theta = 0; % prior expectation parameter setting theta
sig_theta = 1e3; % prior variance parameter setting theta
mu_sl_sigsqr = [4.6:1]; % prior shape and scale parameter setting 'sigma^2'

% fixed form variational Bayes Newton algorithm parameters
% -----
max_l = 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_sigsqr optimization
delta = sqrt(eps); % Hessian modification constant
c = 1e-4; % backtracking constant c
rho = 0.9; % backtracking constant rho
varags = 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
mcond = size(m_f.K,1); % number of conditions
sigsqr = 1e-1; % true, but unknown, data variance
h_theta_fg = exp(h_theta_fg, sigsqr*eye(mcond)); % data expectation
y = mvnrnd(h_theta_fg, sigsqr*eye(mcond)); % 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)]);
[h_theta_fg,u] = spm_fm_f(theta);

% cycle over conditions
for cond = 1:size(m_f.K,1)
% -----
% separate latent between sources
f_theta_fg_1 = f_theta_fg([cond]:(1+cond));
f_theta_fg_2 = f_theta_fg([cond]:(10+cond));

% condition specific latent variable evolution
subplot(3,4,(cond-1)*4+1)
plot(m_f.pst,f_theta_fg_1)
axis([m_f.pst(1) m_f.pst(end)])
title('f', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 22, 'FontName', 'Calibri Light')
ylabel('u', 'FontSize', 22, 'FontName', 'Calibri Light')
set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')
ylim([0 60])

subplot(3,4,(cond-1)*4+2)
plot(m_f.pst,f_theta_fg_2)
axis([m_f.pst(1) m_f.pst(end)])
title('f', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
xlabel('Time [s]', 'FontSize', 22, 'FontName', 'Calibri Light')
ylabel('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)
hsvsps(m_f.pst, 1:numel(m_g.elab),yc([cond]))
axis([m_f.pst(1) m_f.pst(end)])
cb = colorbar;
title('y', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
ylabel('muV', 'Notation', 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)
eui = [47 49 51];
eolab = m_g.elab(eui);
data = yc([cond]);
plot(m_f.pst,data(eui,:))
axis([m_f.pst(1) m_f.pst(end)])
title('y', 'FontName', 'Calibri Light', 'FontSize', 22, 'FontWeight', 'Normal')
legend(eolab)
xlabel('Time [s]', 'FontSize', 22, 'FontName', 'Calibri Light')
ylabel('muV', 'Notation', 0, 'FontName', 'Calibri Light', 'FontSize', 20)
set(gca, 'FontSize', 18, 'FontName', 'Calibri Light')
end

% -----
% VBM Iteration 0
% -----
% initialization of the variational parameters to prior parameters
h_theta = mu_theta;
m_sigsqr = sig_theta;
vfeary_y = y;
vfeary_m_f = m_f;
vfeary_m_g = m_g;
vfeary_h_theta = h_theta;
vfeary_sig_theta = sig_theta;
vfeary_mu_sigsqr = mu_sl_sigsqr(1);
vfeary_sl_sigsqr = mu_sl_sigsqr(2);
vfeary_theta = theta;

% parameter iteration arrays
h_theta_1 = NaN(1,p,max_l-1);
p_h_theta_1 = NaN(1,max_l-1);
h_theta_j = NaN(p,max_j,max_l-1);
p_h_theta_j = NaN(p,max_j,max_l-1);
mu_sigsqr_k = NaN(2,max_k,max_l-1);
p_mu_sigsqr_k = NaN(max_k,max_l-1);

% initialization and initial evaluation of the negative free energy
F = NaN(max_l-1);
F(1) = vFE(h_theta, h_theta, mu_sigsqr, vfeary);

% -----
% VBM Iterations 1,2,3,...

```



```

% -----
for i = 2:max_i
% User update
% -----
% 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 ... ')
[m_theta, s_theta] = full(spm_cat(spm_diff(@m_theta) exp_h(m_f, m_g, spm_unvec(m_theta, theta)), m_theta, 1));
s_theta = spm_inv(exp(-m_sigqr(1)+1/2)*m_sigqr(2))/(CH_m_theta*CH_m_theta + (spm_inv(sig_theta)));
% save iterand and objective function
s_theta_i(i,1:2) = s_theta;
F_m_theta_i(i-1) = vFE(m_theta, s_theta, m_sigqr, vfeqr);
% inform user
fprintf('finished.\n');
% Newton line search for theta_f variational expectation parameter
% -----
% save iterand and objective function
m_theta_j(i,1:2) = m_theta;
F_m_theta_j(i,1:2) = vFE(m_theta, s_theta, m_sigqr, vfeqr);
% numerical gradient and Hessian of the negative free energy
DF_m_theta = full(spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, m_sigqr, vfeqr), m_theta, 1));
DDF_m_theta = full(spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, m_sigqr, vfeqr), m_theta, 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, s_theta, m_sigqr, vfeqr, 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, m_sigqr, vfeqr), m_theta, 1));
        DDF_m_theta = full(spm_cat(spm_diff(@m_theta) vFE(m_theta, s_theta, m_sigqr, vfeqr), m_theta, 1));
        % record updates
        m_theta_j(i,j,1:2) = m_theta;
        F_m_theta_j(i,j,1:2) = vFE(m_theta, s_theta, m_sigqr, vfeqr);
        if norm(DF_m_theta) < vareps || norm(m_theta_j(i,j,1:2) - m_theta_j(i,j-1,1:2)) < vardelta
            break
        end
    end
end
% inform user
fprintf('finished.\n');
% Newton line search for sigma^2 variational parameters
% -----
% save iterand and objective function
m_sigqr_k(i,1:2) = m_sigqr;
F_m_sigqr_k(i,1:2) = vFE(m_theta, s_theta, m_sigqr, vfeqr);
% evaluation of the current negative free energy gradient and Hessian wrt m_sigqr
DF_m_sigqr = full(spm_cat(spm_diff(@m_sigqr) vFE(m_theta, s_theta, m_sigqr, vfeqr), m_sigqr, 1));
DDF_m_sigqr = full(spm_cat(spm_diff(@m_sigqr) vFE(m_theta, s_theta, m_sigqr, vfeqr), m_sigqr, 1));
% Newton iterations 1,2,3,...
if norm(DF_m_sigqr) > vareps
    for k = 2:max_k
        % Newton search direction
        p_k = -DDF_m_sigqr\DF_m_sigqr;
        % Hessian modification if p_k is not a descent direction
        if -(p_k'*DF_m_sigqr < 0)
            % diagonal Hessian modification based on spectral decomposition
            dDF_m_sigqr = DDF_m_sigqr + max(0, delta - min(eig(dDF_m_sigqr))) * eye(2);
            % re-evaluate Newton search direction
            p_k = -DDF_m_sigqr\DF_m_sigqr;
        end
        % backtracking evaluation of the step length
        t_k = backtrack_vFE(m_theta, s_theta, m_sigqr, vfeqr, DF_m_sigqr, p_k, c.rho);
        % perform Newton update
        m_sigqr = m_sigqr + t_k*p_k;
        % update remaining entries in free energy arrays
        DF_m_sigqr = full(spm_cat(spm_diff(@m_sigqr) vFE(m_theta, s_theta, m_sigqr, vfeqr), m_sigqr, 1));
        DDF_m_sigqr = full(spm_cat(spm_diff(@m_sigqr) vFE(m_theta, s_theta, m_sigqr, vfeqr), m_sigqr, 1));
        % record updates
        m_sigqr_k(i,k,1:2) = m_sigqr;
        F_m_sigqr_k(i,k,1:2) = vFE(m_theta, s_theta, m_sigqr, vfeqr);
        if norm(DF_m_sigqr) < vareps || norm(m_sigqr_k(i,k,1:2) - m_sigqr_k(i,k-1,1:2)) < vardelta
            break
        end
    end
end
% inform user
fprintf('finished.\n');
% free energy update
% -----
F(i) = vFE(m_theta, s_theta, m_sigqr, vfeqr);
% 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_k, s_theta_f_k] = linspace(-5,1.5,164);
hold on
plot(theta_f_k, pdf(Normal, theta_f_k, m_theta, sqrt(s_theta)), 'r');
plot(theta_f_k, pdf(Normal, theta_f_k, m_theta, sqrt(s_theta)), 'b');
plot(m_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', 'NorthEast');
set(gca, 'FontSize', 20, 'FontName', 'Calibri Light');
xlim([theta_f_min theta_f_max]);
% sigma^2 prior and posterior
% -----
subplot(3,4,10)
[sigqr_k, s_sigqr_k] = linspace(1e-3,1,164);
hold on
plot(sigqr_k, logpdf(sigqr_k, m_sigqr(1), sqrt(m_sigqr(2))), 'r');
plot(sigqr_k, logpdf(sigqr_k, m_sigqr(1), sqrt(m_sigqr(2))), 'b');
plot(exp(m_sigqr(1) + 5*m_sigqr(2)), 0, 'o', 'MarkerFaceColor', 'b', 'MarkerSize', 8);
plot(exp(m_sigqr(1) + 5*m_sigqr(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 sigqr_max]);
% m_theta variational free energy ascent on first VBNI 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 l = 1:res
    vFE_m_theta(l) = vFE(m_theta_space(l), s_theta_i(i,1:2), m_sigqr, vfeqr);
end
hold on
plot(m_theta_space, vFE_m_theta, 'b');
plot(m_theta_space, vFE_m_theta, 'w', 'MarkerFaceColor', 'w', 'MarkerSize', 8);
xlabel('m_theta', 'FontSize', 22, 'FontName', 'Calibri Light');
legend('');
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_1])

```

```

title ['m_theta' [1]], '\sigma_theta' [1], m ['sigma^2' [1]], s_1 ['sigma^2' [1]], 'FontName', 'Calibri Light', 'FontSize', 16)
xlabel('t', '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] = exp_nm_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 EBPd
%
% 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] = exp_fm_g_theta;
g_theta_g = exp_fm_g;

% evaluate condition specific concatenated function h
% -----
h_theta_fg = cell2i, 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] = exp_fm_g_theta

% This function integrates the neural evolution function of the delay
% differential equation model for EBPd. It is based on spm_gen_exp_m of
% the SPM2 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.B = m_f.B;
theta.T = m_f.T;
theta.G = m_f.G;
theta.H = m_f.H;
theta.D = m_f.D;
theta.A = m_f.A;
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.ms)*m_f.dt)*1000;
delay = m_f.com(1)+28*theta.R(1,1);
scale = m_f.dsr(1)*exp(theta.B(1,2));
u = 32*exp(-t - delay).^2/(2*scale^2);

% initialise 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[3] = m_f.A[3] + theta.X(cond)*10 0; theta.B 0;

% integrate system
% -----
%   f      : initial condition
%   dt     : integration time step
%   [fx, dfdx, D] = [f(x,u), theta, m_f] : dx(t)/dt and Jacobian df/dx and check for delay operator
%   p      : maximum number of iterations
%   N      : cell(max(1, dt*p^2))
%   n      : spm_length(u)
%   Q      : (spm_exp(dt*D)+dfdx/N) - spm_eye(n,n)*spm_inv(dfdx)
%   y      : spm_vec(x, length(u))
%   u      : MAM(length(u), length(u))
%   y      : initialise state
%   u      : initialise state time-course

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

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

end

% re-order according to sources
ns = size(theta.C, 1);
si = MAM(9, ns);
for s = 1:ns
    si(i, s) = s*ns+9*s;
end

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

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

% state equations for a neural mass model of erps
% FORMAT [f, J, Q] = spm_exp(x, u, P, M)
% FORMAT [f, J] = spm_exp(x, u, P, M)
% FORMAT [Q] = spm_exp(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))
% Q - D(d)*f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
% M.pP.E = [32 16 4]; % extrinsic rates (forward, backward, lateral)
% M.pP.R = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2, g3, g4)
% M.pP.D = [2 16]; % propagation delays (intrinsic, extrinsic)
% M.pP.G = [4 32]; % receptor densities (excitatory, inhibitory)
% M.pP.T = [8 16]; % synaptic constants (excitatory, inhibitory)
% M.pP.R = [1 1/2]; % parameter of static nonlinearity
%
% David O. Friston KJ (2003) A neural mass model for MEG/EEG: coupling and
% neural dynamics. NeuroImage 20: 133-152
%
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging
%
% Karl Friston
% SID: spm_exp.m 5369 2013-03-28 20:09:272 Karl S

% get dimensions and configure state variables
N = length(D.A[1]); % number of sources
x = spm_unvec(x, M.X); % neuronal states

% [default] fixed parameters
% -----
R = [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.B);

```

```

G = ones(n,1)*G;
% no exponential transforms to foster parameter identifiability
%-----
A[1] = P.A[1]*R(1);
A[2] = P.A[2]*R(2);
A[3] = P.A[3]*R(3);
C = P.C;
% intrinsic connectivity and parameters
%-----
Te = T(1)/1000*exp(P.T(1,1)); % excitatory time constants
Ti = T(2)/1000*exp(P.T(1,2)); % inhibitory time constants
He = H(1)*exp(P.O(1,1)); % excitatory receptor density
Hi = H(2)*exp(P.O(1,2)); % inhibitory receptor density
% pre-synaptic input: s(V)
S = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));
% exogenous input
%-----
U = C*ui(1)*2;
% State: f(x)
%-----
f(1,1) = M(1,4); % voltage change (spiny stellate cells)
f(1,2) = M(1,5); % positive voltage change (pyramidal cells) +ve
f(1,3) = M(1,6); % negative voltage change (pyramidal cells) -ve
f(1,4) = (He.*(A[1] + A[3])*S(1,9) + G(1,1)*S(1,9) + U) - 2*(1,4) - X(1,1)/Te; % current change (spiny stellate cells) depolarizing
f(1,5) = (He.*(A[2] + A[3])*S(1,9) + G(1,2)*S(1,9)) - 2*(1,5) - X(1,2)/Te; % current change (pyramidal cells) depolarizing
f(1,6) = (Hi.*(M(1,4) - S(1,7)) - 2*(1,6) - X(1,3)/Ti; % current change (pyramidal cells) hyperpolarizing
f(1,7) = M(1,8); % voltage change (inhibitory interneurons) depolarizing
f(1,8) = (He.*(A[2] + A[3])*S(1,9) + G(1,3)*S(1,9) - 2*(1,8) - X(1,7)/Te; % current change (inhibitory interneurons) depolarizing
f(1,9) = X(1,5) - X(1,6); % voltage (pyramidal cells)
%-----
% vectorize
f = spm_vec(f);
% avoid infinite recursion of spm_diff
if nargin < 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) = -Q(d)f(x(t))
%-----
J(d) = -Q(d)df/dx
%-----
De = D(2).*exp(P.D)/1000;
De = D(1)/1000;
Di = (1 - splaye(n,n)).*De;
Di = (1 - splaye(9,9)).*Di;
De = kron(ones(9,9),Di);
Di = kron(Di,splaye(n,n));
D = Di + De;
% Implement: dact/dt = f(x(t - d)) = inv(1 + D.*dtx)*f(x(t))
%-----
Q = spm_inv(splaye(length(J)) + D.*J);
end
function [g_theta_g] = exp_g(m_g)
% This function evaluates the EED forward model of the delay differential
% equation model for EEPs based on a structural formulation of the forward
% model and parameter setting
%-----
% Inputs
% m_g : EED forward model structure and fixed parameters
% theta : EED lead-field free parameters
%-----
% Outputs
% g_theta_g : evaluated EED 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_champs,1) % number of electrodes
dp = 1e-3*theta.Lpos % dipole coordinates adjusted for fieldtrip
% compute canonical dipole lead-field using fieldtrip
Lcan = hem(m_g.nd);
for l = 1:nd
    Lcan(:,l,1) = ft_compute_leadfield(dp(:,l), m_g.sens, m_g.vol);
end
% rescale lead-field according to EPM
Lcan = Lcan*(10^4)*round(log10(max(abs(Lcan))))/8)/1;
% evaluate channel predictions based on dipole moments
L_dip = hem(m_g.nd);
for l = 1:nd
    L_dip(:,l) = Lcan(:,l,1)*theta.L(:,l);
end
% evaluate g(theta_g)
g_theta_g = [];
for l = 1:nd
    g_theta_g = [g_theta_g kron(theta.J,L_dip(:,l))];
end
end
function [VFE] = vFE(m_theta_s_theta_ms_sigsgqr,Vfeary)
% This function evaluates the variational free energy for EEP-DCM toy
% problem
%-----
% Inputs
% m_theta : p x l variational expectation for theta
% s_theta : p x p variational variance for theta
% ms_sigsgqr : 2 x l variational parameters for sigma^2
% vfeary : additional argument structure with fields
% y : n x l array - data
% m : string - function handle
% m_theta : scalar prior expectation for theta
% sig_theta : scalar prior variance for theta
% ms_sigsgqr : scalar prior scale parameter for sigma^2
% sl_sigsgqr : 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_sigsgqr = ms_sigsgqr(1);
s_sigsgqr = ms_sigsgqr(2);
y = vfeary.y;
m_f = vfeary.m_f;
m_g = vfeary.m_g;
m_theta = vfeary.m_theta;
sig_theta = vfeary.sig_theta;
ms_sigsgqr = vfeary.ms_sigsgqr;
sl_sigsgqr = vfeary.sl_sigsgqr;
theta = vfeary.theta;
% evaluation of the number data points and parameters
n = size(y,1);
p = size(m_theta,1);
% evaluate h(m_theta) and v(m_theta)
%-----
h_m_theta = exp(h(m_f,m_g,spm_unvec(m_theta, theta)));
Jh_m_theta = full(spm_cat(spm_diff(h(m_theta), exp(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) - (1/2)*m_sigsgqr - (1/2)*exp(-m_sigsgqr + s_theta_sigsgqr)*(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(s_theta*(mu_theta)*s_theta);
T3 = -(1/2)*log(2*pi)*sl_sigsgqr - m_sigsgqr - (1/2)*(1/sl_sigsgqr)*(m_sigsgqr + m_sigsgqr - mu_sigsgqr)^2;
T4 = (1/2)*log(det(s_theta)) + (p/2)*log(2*pi*exp(1));
T5 = (1/2) + (1/2)*log(2*pi*s_sigsgqr) + m_sigsgqr;
% evaluate the negative free energy
VFE = -(T1 + T2 + T3 + T4 + T5);
end
function [t_k] = backtrack_vFE_m_theta(x_k, s_theta, ms_sigsgqr, vfeary, 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 : p x l array of function argument
% m_theta : p x p array of additional vFE arguments
% ms_sigsgqr : 2 x l array of additional vFE arguments
% vfeary : additional input for vFE
% df_x_k : p x l 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, m_sigsqr, vfearg);
% initialise 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, m_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, m_sigsqr, vfearg);
end
end

function [t_k] = backtrack_vfe_m_sigsqr(m_theta, s_theta, x_k, vfearg, 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
%   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);
% initialise 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_m1
% 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     = 8; % maximal number of fixed-form VB Newton algorithm
max_k     = 8; % maximal number of Newton iterations for m_theta 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_fm_erp; % neural evolution function
m_f.m     = 200; % number of time bins
m_f.dt    = 0.001; % time bin length (sec)
m_f.pct   = (0:m_f.m-1)*m_f.dt; % peristimulus time (msec)
m_f.mt    = length(m_f.pct); % number of time points
m_f.n     = 2; % number of neural masses
m_f.m     = 1; % number of neural masses per neural mass
m_f.ons   = -10; % system input onset (ms)
m_f.dur   = 1; % duration (width) of the input function
m_f.c     = sparsel(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     = [8e-4 0.08]; % static nonlinearity (activation function) parameters
m_f.T     = [ 0.14 0.13; 0.07 -0.82]; % source-specific excitatory and inhibitory time constants
m_f.G     = [-1.11 0.45; 0.52 -0.43]; % source-specific excitatory and inhibitory receptor densities
m_f.H     = [ 0.02 -0.07 0.18 -0.17]; % source-independent intrinsic connectivity parameters
m_f.D     = [ 0 -0.22 -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(1,2); % lateral connectivity
m_f.C     = [NaN NaN]'; % input connectivity
% -----
% forward model structure and parameters
% -----
% load an existing dipole model
load(fullfile(od, '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.labels; % electrode labels
m_g.ne    = numel(m_g.elab); % number of electrodes
m_g.MD    = zeros(m_g.ne,m_g.ne, 3*m_g.ne); % confound design matrix
m_g.ipos  = [42 -31 58; 54 -22 18]'; % dipole coordinate specifications
m_g.L     = [0.02 -0.05 -0.04; 0.17 -0.05 -0.17]'; % dipole moment specification
m_g.W     = [-0.21 0 0 0 0 0 -0.46 0 1]; % state weighting specification
% -----
% likelihood model parameters
% -----
D         = m_f.MD*m_g.W; % number of data points
% -----
% prior formulation
% -----
m_theta   = [ 0 0]; % prior expectation parameter setting theta
m_theta_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
targsps  = 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
% -----
sigsgqr  = 1e-1; % true, but unknown, data variance
h_theta_fg = exp(h_theta_si_sigsqr); % number of parameters
y         = mvnrnd(h_theta_fg, sigsgqr*eye(n)); % data observation
% -----
% 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

% -----
% VBMN iteration 0
% -----
% initialization of the variational parameters to prior parameters
m_theta     = m_theta;
s_theta     = sig_theta;
m_theta_si_sigsqr = m_theta_si_sigsqr;
vfearg_y    = y;
vfearg_m_f  = m_f;
vfearg_m_g  = m_g;
vfearg_m_theta = m_theta;
vfearg_sig_theta = sig_theta;
vfearg_m_sigsqr = m_theta_si_sigsqr(1);
vfearg_si_sigsqr = m_theta_si_sigsqr(2);
vfearg_theta = theta;

% parameter iteration arrays
s_theta_i    = NaN(1,p,max_i-1);
F_s_theta_i = NaN(1,max_i-1);
m_theta_i    = NaN(1,max_i,max_i-1);
F_m_theta_i  = NaN(max_i,max_i-1);
m_theta_si_sigsqr_i = NaN(2,max_k,max_i-1);
F_m_sigsqr_i = NaN(max_k,max_i-1);

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

% -----
% VBMN 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\n');
Jh_m_theta = full(spm_cat(spm_diff(@(m_theta) exp(h(m_f,m_g, m_theta))))(m_theta,1));
s_theta     = spm_inv((spm_cat(spm_diff(@(m_theta) exp(h(m_f,m_g, m_theta))))(m_theta,1)))\Jh_m_theta);
% save iterand and objective function
% -----
s_theta_i(i-1:i-1) = s_theta;
F_s_theta_i(i-1) = vFE(m_theta, s_theta, m_theta_si_sigsqr, vfearg);

```

```

% inform user
fprintf('finished.\n')
% Newton line search for theta_f variational expectation parameter
% -----
fprintf(' Estimating q(theta) expectation ... ')
% save iterand and objective function
m_theta_j(1,1:1) = m_theta;
F_m_theta_j(1,1:1) = VFE(m_theta,s_theta,ms_sigqr,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_sigqr,vfearg),m_theta,1));
d2F_m_theta = full(spm_cat(spm_diff(m_theta) VFE(m_theta,s_theta,ms_sigqr,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 = -dF_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
            dF_m_theta = dF_m_theta + max(0,(delta - min(eig(dF_m_theta))))*eye(p);
        % re-evaluate Newton search direction
        p_j = -dF_m_theta/dF_m_theta;
        end
        % backtracking evaluation of the step length
        t_j = backtrack_VFE(m_theta,s_theta,ms_sigqr,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_sigqr,vfearg),m_theta,1));
        d2F_m_theta = full(spm_cat(spm_diff(m_theta) VFE(m_theta,s_theta,ms_sigqr,vfearg),m_theta,1 1));
        % record updates
        m_theta_j(1,j,1:1) = m_theta;
        F_m_theta_j(1,j,1:1) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
        if norm(dF_m_theta) < vareps || norm(m_theta_j(1,j,1:1) - m_theta_j(1,j-1,1:1)) < 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_sigqr_k(1,1:1) = ms_sigqr;
F_ms_sigqr_k(1,1:1) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
% evaluation of the current negative free energy gradient and Hessian wrt ms_sigqr
dF_ms_sigqr = full(spm_cat(spm_diff(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,1));
d2F_ms_sigqr = full(spm_cat(spm_diff(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,[1 1]));
% Newton iterations 1,2,3,...
if norm(dF_ms_sigqr) > vareps
    for k = 2:max_k
        % Newton search direction
        p_k = -dF_ms_sigqr/dF_ms_sigqr;
        % Hessian modification if p_k is not a descent direction
        if -(p_k*dF_ms_sigqr < 0)
            % diagonal Hessian modification based on spectral decomposition
            dF_ms_sigqr = dF_ms_sigqr + max(0,(delta - min(eig(dF_ms_sigqr))))*eye(2);
        % re-evaluate Newton search direction
        p_k = -dF_ms_sigqr/dF_ms_sigqr;
        end
        % backtracking evaluation of the step length
        t_k = backtrack_VFE_ms_sigqr(m_theta,s_theta,ms_sigqr,vfearg,dF_ms_sigqr,p_k,c.rho);
        % perform Newton update
        ms_sigqr = ms_sigqr + t_k*p_k;
        % update remaining entries in free energy arrays
        dF_ms_sigqr = full(spm_cat(spm_diff(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,1));
        d2F_ms_sigqr = full(spm_cat(spm_diff(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg),ms_sigqr,[1 1]));
        % record updates
        ms_sigqr_k(k,1:1) = ms_sigqr;
        F_ms_sigqr_k(k,1:1) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
        if norm(dF_ms_sigqr) < vareps || norm(ms_sigqr_k(k,1:1) - ms_sigqr_k(k-1,1:1)) < vardelta
            break
        end
    end
end
% inform user
fprintf('finished.\n')
% free energy update
% -----
F(i) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
% user update
% -----
fprintf('VFE %6.3F dVFE %6.3F Va', F(i), F(i) - F(i-1))
end
% save free energy evolution
% -----
F_all(idx, 1:1) = F;
save('data_F_all.mat', 'F_all')
idx = idx + 1;
end
end
% -----
% Visualization
% -----
fig = figure;
set(fig, 'Color', [1 1 1])
% input architecture/function u
% -----
[u_theta_f,u] = exp_fm_f_theta;
subplot(2,3,1)
plot(m_f_pat,u)
xlim([u_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_pat,u)
xlim([u_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,:),:)).6, 'FaceColor', [.8 .8 .8])
for l = 1:nsim
    plot(1:2,-F_all(1:2,end,l), 'k-', 'MarkerFaceColor', 'w', 'MarkerSize', 4)
end
ylim([-3000 -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,:),:)).6, 'FaceColor', [.8 .8 .8])
for l = 1:nsim
    plot(1:2,-F_all(3:4,end,l), 'k-', 'MarkerFaceColor', 'w', 'MarkerSize', 4)
end
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,:),:)), 'ko-', 'MarkerFaceColor', 'k')
plot(-mean(F_all(3:4,end,:),:)), 'ko-', 'MarkerFaceColor', 'k')
title('F(m_theta) [1], V(Sigma,theta) [1], m_1[|sigma^2|] [1], s_1[|sigma^2|] [1]', '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,:),:)), 'ko-', 'MarkerFaceColor', 'k')

```

```

plot(mean(F_all(3:end,:)), 'b', 'Color', [0.5 0.5], 'MarkerFaceColor', [0.5 0.5])
xlabel('t', 'FontSize', 24, 'FontName', 'Calibri Light')
title('f_m_theta', 'FontSize', 14, 'FontName', 'Calibri Light', 'FontWeight', 'Normal', 'FontSize', 22)
legend('u', 'u', 'Location', 'SouthEast')
set(gca, 'FontName', 'Calibri Light', 'FontSize', 24)
xlim([0 5])

end

% -----
% Subfunctions
% -----
function [h_theta_f] = exp_m_f_m_u_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_f : evaluated function

% Copyright (C) Dirk Ostwald
% -----
% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_fm_exp.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.nm)*m_f.dt)+1200;
delay = m_f.Cos(1)+128*theta.R(1,1);
scale = m_f.dur(2)*exp(theta.R(1,2));
u = 32*exp(-t - delay) - 2/(2*scale)^2;

% Integrate system
x = m_f.x; % Initial condition
f = m_f.f; % Function handle spm_fm_exp
dt = m_f.dt; % Integration time bin
[fx,dfdx,D] = [x,u(1),theta,m_f]; % dx(t)/dt and Jacobian df/dx and check for delay operator
D = NaN(size(real(eig(fx))),1);
N = cell(max(1, dt*p^2));
n = spm_length(D);
Q = (spm_exp(dt*D*dfdx/N) - spm_eye(n,n))*spm_inv(dfdx);
v = spm_uv(n); % Initialize state
y = NaN(length(v), length(u)); % Initialize state time-course

% Cycle over time-steps
for i = 1:size(u,1)
    % Update dx = (exp(dt*N) - I)*inv(J)*f(x,u)
    for j = 1:n
        v = v + Q*(v,u(i),theta,m_f);
    end
    y(i,1) = v;
end

% Transpose
f_theta_f = y';

end

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

% State equations for a neural mass model of ERPs
% FOMAT [F,J,Q] = spm_fm_exp(x,u,P,M)
% FOMAT [F,J] = spm_fm_exp(x,u,P,M)
% FOMAT [F] = spm_fm_exp(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(x)/dx(t)
% D - delay operator dx(t)/dt = f(x(t-d))
% = D(d)*f(x(t))

% Prior fixed parameter scaling [Defaults]

% M.p.p.R = [32 16 4]; % extrinsic rates (forward, backward, lateral)
% M.p.p.H = [1 4/5 1/4 1/4]*128; % intrinsic rates (g1, g2, g3, g4)
% M.p.p.D = [2 16]; % propagation delays (intrinsic, extrinsic)
% M.p.p.G = [4 22]; % receptor densities (excitatory, inhibitory)
% M.p.p.T = [8 16]; % synaptic constants (excitatory, inhibitory)
% M.p.p.R = [1 1/2]; % parameter of static nonlinearity

% David O. Friston K2 (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
% SID: spm_fm_exp.m 5369 2013-03-28 20:09:27Z karl s

% Get dimensions and configure state variables
% -----
n = length(P.A[1]); % number of sources
x = spm_uvvec(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 22]; % 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]*R(1);
A[2] = -P.A[2]*R(2);
A[3] = -P.A[3]*R(3);
C = -P.C;

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

% pre-synaptic input a(V)
% -----
S = 1./(1 + exp(-R(1)*(x - R(2)))) - 1./(1 + exp(R(1)*R(2)));

% exogenous input
% -----
U = C*u(1):2;

% State: f(x)
% -----
f(1) = x(1:4); % Voltage change (spiny stellate cells)
f(2) = x(1:5); % Positive voltage change (pyramidal cells) +ve
f(3) = x(1:6); % Negative voltage change (pyramidal cells) -ve

```

```

f(:,4) = (h0.*[A1] + A3)*S1(:,9) + G(1,1)*S1(:,9) + U1 - 2*x(1,4) - x(1,1)/T0; % current change (spinly stellate cells) depolarizing
f(:,5) = (h0.*[A1] + A3)*S1(:,9) + G(1,2)*S1(:,9) + U2 - 2*x(1,5) - x(1,2)/T0; % current change (pyramidal cells) depolarizing
f(:,6) = (h1.*G(1,4)*S1(:,7) - 2*x(1,6) - x(1,3)/T1); % voltage change (pyramidal cells) hyperpolarizing
f(:,7) = x(1,8); % voltage change (inhibitory interneurons) hyperpolarizing
f(:,8) = (h0.*[A2] + A3)*S1(:,9) + G(1,3)*S1(:,9) - 2*x(1,8) - x(1,7)/T0; % current change (inhibitory interneurons) depolarizing
f(:,9) = x(1,9) - x(1,6); % voltage (pyramidal cells)

% vectorize
f = spm_vecvec(f);
% avoid infinite recursion of spm_diff
if nargin < 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 = diff(f(x(t)))
% J(d) = q/diddf/dx
%-----
De = D(2); %exp(9, D)/1000;
Di = D(1)/1000;
De = (1 - spye(size(De))); %De;
Di = (1 - spye(size(Di))); %Di;
De = kron(ones(9,9),De);
Di = kron(Di,spye(size(Di)));
D = Di + De;
% Implement: dx(t)/dt = f(x(t-d)) - inv(1 + D.*diddf)*f(x(t))
% Q = f - Q.*f(x(t))
%-----
Q = spm_inv(speye(length(J)) + D.*J);
end
function [g_theta_g] = exp_g(m_g)
% This function evaluates the EEG forward model of the delay differential
% equation model for EEPs based on a structural formulation of the forward
% model and parameter settings
%
% 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_channels,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,nd);
for i = 1:nd
    L_can(:,i) = ft_compute_leadfield(dp(i,:), m_g.sens, m_g.vol);
end
% compute lead-field according to SNM
L_can = L_can./10^11*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,i);
end
% evaluate g(theta_g)
g_theta_g = kron(theta.J,L_dip);
end
function [VFE] = VFE(m_theta,s_theta,ms_sigsgqr,vfearg)
% This function evaluates the variational free energy for EEP-DCM toy
% problem
%
% Inputs
% m_theta : p x 1 variational expectation for theta
% s_theta : p x p variational variance for theta
% ms_sigsgqr : 2 x 1 variational parameters for sigma^2
% vfearg : additional argument structure with fields
% y : n x 1 array - data
% rh : string - function handle
% mu_theta : scalar prior expectation for theta
% sig_theta : scalar prior variance for theta
% mu_sigsgqr : scalar prior scale parameter for sigma^2
% sl_sigsgqr : 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
ms_sigsgqr = ms_sigsgqr(1);
s_theta = ms_sigsgqr(2);
y = vfearg.y;
rh = vfearg.rh;
mu_theta = vfearg.mu_theta;
sig_theta = vfearg.sig_theta;
mu_sigsgqr = vfearg.mu_sigsgqr;
sl_sigsgqr = vfearg.sl_sigsgqr;
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_theta)
h_m_theta = exp_h(m_theta,m_g,spm_unvec(m_theta, theta));
Jh_m_theta = full(spm_cat(spm_diff(@(m_theta) exp_h(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_sigsgqr - (1/2)*exp(-m_sigsgqr + s*m_sigsgqr)*(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(s_theta)) - (1/2)*(y - h_m_theta - mu_theta)'*inv(s_theta)*(y - h_m_theta - mu_theta) + trace(s_theta*(s_theta));
T3 = -(1/2)*log(|sl_sigsgqr|) - (1/2)*log(det(s_theta)) - (1/2)*(mu_theta - mu_theta)'*inv(sl_sigsgqr)*(mu_theta - mu_theta) + trace(sl_sigsgqr*(s_theta));
T4 = -(1/2)*log(det(s_theta)) + (p/2)*log(2*pi*exp(1));
T5 = (1/2) + (1/2)*log(2*pi*exp(1)) + m_sigsgqr;
% evaluate the negative free energy
VFE = -(T1 + T2 + T3 + T4 + T5);
end
function [t_k] = backtrack_VFE_m_theta(x_k,s_theta,ms_sigsgqr,vfearg,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 : p x 1 array of function argument
% s_theta : 2 x 2 array of additional VFE arguments
% ms_sigsgqr : 2 x 1 array of additional VFE arguments
% vfearg : additional input for VFE
% df_x_k : p x 1 array of function gradient at x_k
% c_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_sigsgqr,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_sigsgqr,vfearg);
% check sufficient decrease of the objective function and reset step-size
while f_x_k_p_1 > f_x_k + c*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_sigsgqr,vfearg);
end
function [t_k] = backtrack_VFE_ms_sigsgqr(m_theta,s_theta,x_k,vfearg,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
% 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
%
% Output
% t_k : backtracking/Armijo step-size

```



```

% df_xk      : 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_xk = VFE(m_theta, s_theta, x_k, vfeary);
%
% initialise step-size
t_k = 1;
%
% evaluation of f(x_k+1) given the initial step size
f_xk_p_1 = VFE(m_theta, s_theta, x_k + t_k*p_k, vfeary);
%
% check sufficient increase of the objective function and reset step-size
while f_xk_p_1 > f_xk + c*t_k*(df_xk'*p_k) || isnan(f_xk_p_1) || x_k(2) + t_k*p_k(2) < 0;
    t_k = t_k*rho;
    f_xk_p_1 = VFE(m_theta, s_theta, x_k+t_k*p_k, vfeary);
end
end

```

PDDE_9.m – ERP-DCM experimental application (Figure 13)

```

function pdde_9_m
% 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 neurophysiological model.
% Copyright (C) Dirk Ostwald
% -----
clear all

% -----
% ECG Data Loading
% -----
spm_dir = 'D:\Neuro\ERP\Projects\VBEM\Delay_Differential_SPM\Revision\Matlab\JMG12';
fname = fullfile(spm_dir, 'Grand_Mean_all_all.mat'); % source directory
twidth = 1; % peristimulus time indices of a 0-100 ms time window
sr = 1; % data scale factor to avoid numerical overflow
addpath(genpath(spm_dir)); % invoke SPM
D = spm_spm_load(fname); % input function parameters rho_1 and rho_2
y = D.*spm_vec(D); % peristimulus time
par = D.times(1000); % time bin length
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.m = spm_m_erp; % neural evolution function
m_f.mse = length(y); % number of time bins
m_f.dt = dt; % time bin length (sec)
m_f.pst = par; % 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.sme = 10; % system input onset (ms)
m_f.dur = 10; % duration (width) of the input function
m_f.x = spars(m_f.n,m_f.m); % initial condition
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.L = 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(od, '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.nme = numel(m_g.elab); % number of electrodes
m_g.X0 = zeros(m_f.n*m_g.n,m_g.n*m_g.n); % confound design matrix
m_g.lpos = [42 54; 31 -22; 58 18]; % dipole coordinate specifications
m_g.L = [116.64 20.87; -3.65 -24.43; 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

% -----
% VBEM initialization
% -----
% invoke Fieldtrip
addpath(genpath(fullfile(spm_dir, 'Fieldtrip')));
max_i = 12; % maximal number of fixed-form VB Newton algorithm
max_k = 8; % maximal number of Newton iterations for m_theta optimization
delta = sqrt(eps); % Hessian modification constant
c = 1e-4; % backtracking constant c
rho = 0.9; % backtracking constant rho
vareps = 1e-6; % gradient norm convergence criterion
vardelta = 1e-3; % parameter norm change convergence criterion

% model-independent prior formulation
% -----
m_theta = zeros(p,1); % prior expectation parameter setting theta
m_theta_sigqr = [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,nm); % theta variational covariances
m_sigqr_m = NaN(2,max_k,max_i-1); % sigma^2 variational parameters
F_m = NaN(nm,max_k,max_i-1); % variational free energy evolutions

% -----
% Cycle over probabilistic models
% -----
for m = 1:nm
% model-dependent prior covariance matrix
switch m
% loose prior on e_1, tight prior on e_2
case 1
sig_theta = [1e6 0; 0 1e-6]; % prior covariance parameter setting theta
% loose prior on e_1 and e_2
case 2
sig_theta = 1e6*eye(p); % prior covariance parameter setting theta
end

% -----
% VBEM iteration 0
% -----
% initialization of the variational parameters to prior parameters
theta = m_theta;
s_theta = s_theta_m;
m_sigqr = m_sigqr_m;
vfeary = y;
vfeary_m_f = m_f.f;
vfeary_m_g = m_g;
vfeary_m_theta = m_theta;
vfeary_sig_theta = sig_theta;
vfeary_m_sigqr = m_sigqr(1);
vfeary_m_sigqr = m_sigqr(2);
vfeary_s_theta = s_theta;

% parameter iteration arrays
m_theta_i = NaN(p,max_i-1);
F_m_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);
m_sigqr_k = NaN(2,max_k,max_i-1);
F_m_sigqr_k = NaN(max_k,max_i-1);

% initialization and initial evaluation of the negative free energy
F = NaN(1,max_i-1);
v(f) = -VFE(m_theta, s_theta, m_sigqr, vfeary);

% -----
% VBEM iterations 1,2,3,...
% -----
for i = 2:max_i
% user update
% -----
fprintf('Fixed-Form VB Newton Iteration %d.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) exp_h(m_f,m_g, spm_unvec(m_theta), m_theta)), m_theta));
s_theta = spm_inv(exp(-m_sigqr(1)/(1/2)*m_sigqr(2))*Jh_m_theta'*Jh_m_theta + (spm_inv(s_theta)));
% save iterand and objective function
s_theta_i(i-1:i-1) = s_theta;
F_m_theta_i(i-1) = -VFE(m_theta, s_theta, m_sigqr, vfeary);
% 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(i-1:i-1) = s_theta;
F_m_theta_j(i-1:i-1) = -VFE(m_theta, s_theta, m_sigqr, vfeary);
% numerical gradient and Hessian of the negative free energy
dF_m_theta = full(spm_cat(spm_diff@(m_theta) VFE(m_theta, s_theta, m_sigqr, vfeary), m_theta, 1));
d2F_m_theta = full(spm_cat(spm_diff@(m_theta) VFE(m_theta, s_theta, m_sigqr, vfeary), 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 = -d2F_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
    dF_m_theta = dF_m_theta + max(0,delta - min(eig(dF_m_theta))))*eye(p);
    % re-evaluate Newton search direction
    p_j = -dF_m_theta/dF_m_theta;
end
% backtracking evaluation of the step length
t_j = backtrack_VFE_m_theta(m_theta, s_theta, ms_sigqr, 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(sgm_cat(sgm_diff(@(m_theta) VFE(m_theta, s_theta, ms_sigqr, vfearg);m_theta,1)));
dF_m_theta = -full(sgm_cat(sgm_diff(@(m_theta) VFE(m_theta, s_theta, ms_sigqr, vfearg);m_theta,1,1)));
% record updates
m_theta_j(:,j,1-1) = m_theta;
F_m_theta_j(j,1-1) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
if norm(dF_m_theta) < vareps || norm(m_theta_j(:,j,1-1) - m_theta_j(:,j-1,1-1)) < 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_sigqr_k(1,1-1) = ms_sigqr;
F_ms_sigqr_k(1,1-1) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
% evaluation of the current negative free energy gradient and Hessian wrt ms_sigqr
dF_ms_sigqr = -full(sgm_cat(sgm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg);ms_sigqr,1)));
dF_ms_sigqr = -full(sgm_cat(sgm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg);ms_sigqr,1,1)));
% Newton iterations 1,2,3,...
if norm(dF_ms_sigqr) > vareps
    for k = 2:imax_k
        % Newton search direction
        p_k = -dF_ms_sigqr/dF_ms_sigqr;
        % Hessian modification if p_j is not a descent direction
        if ~(p_k)'*dF_ms_sigqr < 0
            % diagonal Hessian modification based on spectral decomposition
            dF_ms_sigqr = dF_ms_sigqr + max(0,delta - min(eig(dF_ms_sigqr))))*eye(2);
        % re-evaluate Newton search direction
        p_k = -dF_ms_sigqr/dF_ms_sigqr;
        end
        % backtracking evaluation of the step length
        t_k = backtrack_VFE_ms_sigqr(m_theta, s_theta, ms_sigqr, vfearg, dF_ms_sigqr, p_k,c,rho);
        % perform Newton update
        ms_sigqr = ms_sigqr + t_k*p_k;
        % update remaining entries in free energy arrays
        dF_ms_sigqr = -full(sgm_cat(sgm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg);ms_sigqr,1)));
        dF_ms_sigqr = -full(sgm_cat(sgm_diff(@(ms_sigqr) VFE(m_theta,s_theta,ms_sigqr,vfearg);ms_sigqr,1,1)));
        % record updates
        ms_sigqr_k(k,1-1) = ms_sigqr;
        F_ms_sigqr_k(k,1-1) = VFE(m_theta,s_theta,ms_sigqr,vfearg);
        if norm(dF_ms_sigqr) < vareps || norm(ms_sigqr_k(:,k,1-1) - ms_sigqr_k(:,k-1,1-1)) < vardelta
            break
        end
    end
end
% inform user
fprintf(' finished.\n')
% free energy update
% -----
F(i) = VFE(m_theta,s_theta,ms_sigqr,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_sigqr_m(:,m) = ms_sigqr; % sigma^2 variational parameters
F_m(:) = F; % variational free energy evaluations
end
% -----
% Visualization
% -----
% Data
% -----
fig = figure;
set(fig,'Color',[1 1 1]);
subplot(2,3,1)
y_eeq = sgm_unvec(y, NaN(m,g,ne,m_f,nc))/sf;
imagesc(m_f,pat,1: numel(m,g,elab),y_eeq,[-1 1])
xlim([m_f,pat(1) m_f,pat(end)])
cb = colorbar;
title('M', 'FontName','Calibri Light', 'FontSize', 24, 'FontWeight','Normal')
ylabel('b', 'FontName', '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:m
    h_theta_map = sgm_h(m_f,m_d, m_theta_m(1,m)); % data expectation
    y_map = sgm_unvec(h_theta_map, NaN(m,g,ne,m_f,nc))/sf;
    subplot(2,3,m+1)
    imagesc(m_f,pat,1: numel(m,g,elab),y_map,[-1 1])
    xlim([m_f,pat(1) m_f,pat(end)])
    cb = colorbar;
    title(['M', num2str(m)] ' Y MAP', 'FontName', 'Calibri Light', 'FontSize', 24, 'FontWeight','Normal')
    ylabel('b', 'FontName', '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(1,1),[c_1(i) c_2(j)],s_theta_m(:,1:1)) * mvnpdf(m_theta_m(1,2),[c_1(i) c_2(j)],s_theta_m(:,2:2));
    end
end
imagesc(c_2,c_1,q_c)
set(gca, 'dir','Normal')
text(m_theta_m(2,1)+.01,m_theta_m(1,1),q(c) M_1, 'Color',[1 1 1], 'FontSize', 22, 'FontName', 'Calibri Light');
text(m_theta_m(2,2)+.01,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')
ylim([c_2_min c_2_max])
xlabel('c_1', 'FontName', 'Calibri Light', 'FontSize', 24)
ylabel('c_2', 'FontName', 'Calibri Light', 'FontSize', 24, 'Rotation', 0)
% sigma^2 estimates
% -----
subplot(2,3,5)
hold on
sigqr_x = linspace(1e4,5e5);
old = [b; c];
for m = 1:m
    plot(sigqr_x, logmpdf(sigqr_x, ms_sigqr_m(1,m), sqrt(ms_sigqr_m(2,m))), col(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', 20)
end

```

```

% ----- Subfunctions -----
%
function [h_theta_fg] = exp_n(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] = exp_f(m_f, vartheta);
g_theta_g = exp_g(m_g);
% evaluate concatenated function h
h_theta_fg = spm_vec(g_theta_g * f_theta_f');
end

function [f_theta_f, u] = exp_f(m_f, vartheta)
%
% This function integrates the neural evolution function of the delay
% differential equation model for ERPs. It is based on spm_exp_exp.m et
% the SPM2 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
u = ((1/m_f.m) * m_f.dt) * 1000;
% delay
delay = m_f.delay * theta.R(1,1);
% scale
scale = m_f.scale * exp(theta.R(1,2));
u = 32 * exp(-t/delay) .* (2 / (2 * scale ^ 2));
% integrate system
%
% x : state vector
% dt : integration time bin
% [fx, dfdx, D] : [f(x,u), theta, m_f] : f(x,u) and Jacobian df/dx and check for delay operator
% D : matrix of partial derivatives of f with respect to x
% H : cell(max(1, dt)*2)
% n : number of states
% G : [spm_exp(dt)*D + dfdx/N] - spm_eye(n,n) * spm_inv(dfdx)
% v : spm_vec(x)
% y : spm_vec(x)
% y : hdd(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,1) = v;
end
% transpose
f_theta_f = y';
end

function [f, J, G] = spm_exp(x, u, P, M)
%
% state equations for a neural mass model of erps
% FORNAT [f, J, G] = spm_exp(x, u, P, M)
% FORNAT [f, J] = spm_exp(x, u, P, M)
% FORNAT [f] = spm_exp(x, u, P, M)
%
% x : state vector
% u : input vector
%
% x(1) - voltage (spiny stellate cells) +ve
% 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(d) * f(x(t))
%
% Prior fixed parameter scaling [Defaults]
%
% M.pP.R = [32 16 4]; % intrinsic rates (forward, backward, lateral)
% M.pP.R = [1 4/5 1/4 1/4] * 128; % intrinsic rates (g1, g2 g3, g4)
% M.pP.D = [2 16]; % propagation delays (intrinsic, extrinsic)
% M.pP.G = [4 32]; % receptor densities (excitatory, inhibitory)
% M.pP.C = [8 16]; % synaptic constants (excitatory, inhibitory)
% M.pP.R = [1 1/2]; % parameter of static nonlinearity
%
% David O. Priston EJ (2003) A neural mass model for MEG/EEG: coupling and
% neural dynamics. NeuroImage 20: 153-175
%
% Copyright (C) 2005 Wellcome Trust Centre for Neuroimaging
%
% Karl Friston
% SID: spm_exp.m 5369 2013-03-28 20:09:272 karl s
%
% get dimensions and configure state variables
%
% D = length(P.A[1]); % number of sources
% x = spm_unvec(x, M.X); % neuronal states
%
% [default] fixed parameters
%
% R = [1 1/2 1/8] * 32; % intrinsic 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)
% C = [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.R);
% G = ones(n,1) * G;
%
% no exponential transforms to foster parameter identifiability
%
% A[1] = P.A[1] * R[1];
% A[2] = P.A[2] * R[2];
% A[3] = P.A[3] * R[3];
% C = P.C;
%
% intrinsic connectivity and parameters
%
% Te = T(1) / 1000 * exp(P.T(1,1)); % excitatory time constants
% Ti = T(2) / 1000 * exp(P.T(1,2)); % inhibitory time constants
% He = H(1) * exp(P.G(1,1)); % excitatory receptor density
% Hi = H(2) * exp(P.G(1,2)); % inhibitory receptor density
%
% pre-synaptic inputs: s(t)
%
% s = 1 ./ (1 + exp(-R(1) * (x - R(2)))) - 1 ./ (1 + exp(R(1) * R(2)));
%
% exogenous input
%
% U = C * u(i) * 2;
%
% State: f(x)
%
% f(1,1) = x(1,4); % voltage change (spiny stellate cells)
% f(1,2) = x(1,5); % positive voltage change (pyramidal cells) +ve
% f(1,3) = x(1,6); % negative voltage change (pyramidal cells) -ve
% f(1,4) = (M.p.A[1] + A[3]) * S(1,9) + G(1,1) * S(1,9) + U - 2 * x(1,4) - x(1,1) / Te; % current change (spiny stellate cells) depolarizing
% f(1,5) = (M.p.A[2] + A[3]) * S(1,9) + G(1,2) * S(1,2) - 2 * x(1,5) - x(1,2) / Te; % current change (pyramidal cells) depolarizing
% f(1,6) = (M.p.C(1) * S(1,7) - 2 * x(1,6) - x(1,3) / Ti; % current change (pyramidal cells) hyperpolarizing
% f(1,7) = x(1,8); % voltage change (inhibitory interneurons)
% f(1,8) = (M.p.A[2] + A[3]) * S(1,9) + G(1,3) * S(1,9) - 2 * x(1,8) - x(1,7) / Te; % voltage change (inhibitory interneurons) depolarizing
% f(1,9) = x(1,5) - x(1,6); % voltage (pyramidal cells)
%
% vectorize
f = spm_vec(f);

```

```

% avoid infinite recursion of spm_diff
if nargin < 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 - spsye(n,n)).*De;
Di = (1 + spsye(9,9)).*Di;
De = kron(ones(9,9),De);
Di = kron(Di,spsye(n,n));
D = Di + De;

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

end

function [g_theta_g] = exp_g(m_g)

% This function evaluates the EED forward model of the delay differential
% equation model for ESPs based on a structural formulation of the forward
% model and parameter setting
%
% Inputs
%   m_g : EED forward model structure and fixed parameters
% Outputs
%   g_theta_g : evaluated EED forward model
%
% Copyright (C) Dirk Ostwald
%-----
% constant parameters
theta_pos = m_g.sigqr;
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_champs,1) ; % number of electrodes
dp = size(theta_pos) ; % 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_sigqr,vfearg)

% This function evaluates the variational free energy for ESP-DCM toy
% problem
%
% Inputs
%   m_theta : p x l variational expectation for theta
%   s_theta : p x p variational parameters for theta
%   ms_sigqr : 2 x l variational parameters for sigma^2
%   vfearg : additional argument structure with fields
%   .y : n x l array - data
%   .h : string - function handle
%   .mu_theta : scalar prior expectation for theta
%   .sig_theta : scalar prior variance for theta
%   .ms_sigqr : scalar prior scale parameter for sigma^2
%   .sl_sigqr : 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_sigqr = ms_sigqr(1);
s_sigqr = ms_sigqr(2);
y = vfearg.y;
m_f = vfearg.m_f;
m_d = vfearg.m_d;
mu_theta = vfearg.mu_theta;
sig_theta = vfearg.sig_theta;
ms_sigqr = vfearg.ms_sigqr;
sl_sigqr = vfearg.sl_sigqr;
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 h'(m_theta)
h_m_theta = exp_h(m_f,m_d,spm_uvvec(m_theta, theta));
Jh_m_theta = full(spm_cat(spm_diff(@(m_theta) exp_h(m_f,m_d,spm_uvvec(m_theta, theta))),m_theta,2));

% evaluation of the variational free energy value
T1 = -(n/2)*log(2*pi) - (n/2)*m_sigqr - (1/2)*exp(-m_sigqr + s*m_sigqr)*(y - h_m_theta)*(y - h_m_theta) + trace((Jh_m_theta'*Jh_m_theta)*s_theta);
T2 = -(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*sl_sigqr) - m_sigqr - (1/2)*((sl_sigqr)/(s_sigqr + (s_sigqr - mu_sigqr)^2));
T4 = (1/2)*log(det(s_theta)) + (p/2)*log(2*pi*exp(1));
T5 = (1/2)*log(2*pi*m_sigqr) + m_sigqr;

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

end

function [t_k] = backtrack_vFE_m_theta(x_k,s_theta,ms_sigqr,vfearg,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 : p x l array of function argument
%   s_theta : p x p array of additional vFE arguments
%   ms_sigqr : 2 x l array of additional vFE arguments
%   vfearg : additional input for vFE
%   df_x_k : [p, l x l] 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_sigqr,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_sigqr,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_sigqr,vfearg);
end

end

function [t_k] = backtrack_vFE_ms_sigqr(m_theta,s_theta,x_k,vfearg,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
%   m_theta : p x l array of additional vFE arguments
%   s_theta : p x p array of additional vFE arguments
%   x_k : 2 x l array of function arguments
%   vfearg : additional input for vFE
%   df_x_k : 2 x l 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) given the initial step size
f_x_k_p_1 = VFE(m_theta, a_theta, x_k + t_k*p_k, vfeary);

% 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*vba;
    f_x_k_p_1 = VFE(m_theta, a_theta, x_k+t_k*p_k, vfeary);
end
end
```