Variational Gradient Matching for Dynamical Systems: Dynamic Causal Models

Table of Contents

Authors: **Nico Stephan Gorbach** and **Stefan Bauer**, email: nico.gorbach@gmail.com

Instructional code for the NIPS (2018) paper " **Scalable Variational Inference for Dynamical Systems** " by Nico S. Gorbach, Stefan Bauer and Joachim M. Buhmann. The paper is available at https://papers.nips.cc/paper/7066-scalable-variational-inference-for-dynamical-systems.pdf. Please cite our paper if you use our program for a further publication. Part of the derivation below is described in Wenk et al. (2018).

Example dynamical system used in this code: Lotka-Volterra system with **half** of the time points **unobserved**. The ODE parameters are also unobserved.

Advantages of Variational Gradient Matching

The essential idea of gradient matching (Calderhead et al., 2002) is to match the gradient governed by the ODEs with that inferred from the observations. In contrast to previous approaches gradient matching introduces a prior over states instead of a prior over ODE parameters. The advantages of gradients matching is two-fold:

1. A prior over the functional form of state dynamics as opposed to ODE parameters facilitates a more expert-aware estimation of ODE parameters since experts can provide a better *a priori* description of state dynamics than ODE parameters.

2. Gradient matching yields a global gradient as opposed to a local one which offers significant computational advantages and provides access to a rich source of sophisticated optimization tools.

Clear workspace and close figures

```
clear all; close all;
```

Simulation Settings

```
simulation.odes = 'fwd_mod_driving';
simulation.state_obs_variance = @(mean)(bsxfun(@times,
[0.5^2,0.5^2],...
    ones(size(mean))));
     % observation noise
simulation.ode_param = -0.8 + (0.8-(-0.8)) * rand(1,11);
      % true non-selfinhibitory neuronal couplings (sampled uniformily
 in the interval [-0.8,0.8];
simulation.ode_param(end-4:end) = -1;
     % self-inhibotory neuronal couplings set to -1.
simulation.final_time = 359*3.22;
     % end time for integration
simulation.int_interval = 0.01;
     % integration interval
simulation.time_samp = 0:0.1:simulation.final_time;
     % sample times for observations
simulation.observed_states = {};
     % indices of states that are directly observed (Boolean)
simulation.SNR = 5;
     % Signal-to-noise-ratio
```

User Input

```
candidate_odes = 'fwd_mod_driving';
```

```
param_prior_variance = realmax;
```

Kernel parameters ϕ:

```
kernel.param = [10,0.2];
     % set values of rbf kernel parameters
```

Error variance on state derivatives (i.e. γ):

```
state.derivative_variance = 0.9.*ones(11-3,1);
     % \gamma for gradient matching model


time.est= 0:3.22:359*3.22;
     % estimation times


opt_settings.pseudo_inv_type = 'modified Moore-Penrose';
      % Type of pseudo inverse; options: 'Moore-Penrose' or 'modified
 Moore-Penrose'
opt_settings.coord_ascent_numb_iter = 200;
     % number of coordinate ascent iterations
opt_settings.clamp_obs_state_to_GP_regression = true;
     % The observed state trajectories are clamped to the trajectories
 determined by standard GP regression (Boolean)


damping = 0.1;
       % Since the hemodynamic states are inferred locally w.r.t. the
 hemodynamic ODEs we add a damping in the inference.


state.ext_input = importdata('dcm/external_input.txt');
     % importing external inputs
time.samp = state.ext_input(:,1)';
     % unpack sampling time
```

Import Candidate ODEs

```
symbols = importdata(['dcm/ODEs/' candidate_odes '_symbols.mat']);
     % symbols of parameters and states and in 'ODEs.txt' file
ode = import_odes(symbols,candidate_odes);
ode = write_ODEs_as_symbolic_expression(symbols,ode);

disp('candidate ODEs:'); disp(ode.raw)
```

```
candidate ODEs:
    '-(5.*exp((17.*[v_{1}])./8))./8-(25.*exp(-
[q_{1}]).*exp([f_{1}]).*((3./5).^exp(-[f_{1}])-1))./16'
    '-(5.*exp((17.*[v_{3}])./8))./8-(25.*exp(-
[q_{3}]).*exp([f_{3}]).*((3./5).^exp(-[f_{3}])-1))./16'
    '-(5.*exp((17.*[v_{2}])./8))./8-(25.*exp(-
[q_{2}]).*exp([f_{2}]).*((3./5).^exp(-[f_{2}])-1))./16'
    '(5.*exp(-[v_{1}]).*exp([f_{1}]))./8-
(5.*exp((17.*[v_{1}])./8))./8'
    '(5.*exp(-[v_{3}]).*exp([f_{3}]))./8-
(5.*exp((17.*[v_{3}])./8))./8'
    '(5.*exp(-[v_{2}]).*exp([f_{2}]))./8-
(5.*exp((17.*[v_{2}])./8))./8'
    '[s_{1}].*exp(-[f_{1}])'
    '[s_{3}].*exp(-[f_{3}])'
    '[s_{2}].*exp(-[f_{2}])'
```

```
    '[n_1]-(3.*[s_{1}])./5-(8.*exp([f_{1}])./25+8./25'
    '[n_3]-(3.*[s_{3}])./5-(8.*exp([f_{3}])./25+8./25'
    '[n_2]-(3.*[s_{2}])./5-(8.*exp([f_{2}])./25+8./25'
    '[a_{11}].*[n_1]+[a_{12}].*[n_2]+[c_{11}].*[u_{1}]'
    '[a_{32}].*[n_2]+[a_{33}].*[n_3]+[c_{33}].*[u_{3}]'

'[a_{22}].*[n_2]+[a_{23}].*[n_3]+[n_1].*([a_{21}]+[b_{212}].*[u_{2}]+[b_{213}].*[
```

Mass Action Dynamical Systems

A deterministic dynamical system is represented by a set of K ordinary differential equations (ODEs) with model parameters $\theta \in R^d$ that describe the evolution of K states $\mathbf{x}(t) = [x_1(t), \ldots, x_K(t)]^T$ such that:

$$\dot{\mathbf{x}}(t) = \frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), \theta) \qquad (1)$$

A sequence of observations, $\mathbf{y}(t)$, is usually contaminated by measurement error which we assume to be normally distributed with zero mean and variance for each of the K states, i.e. $\mathbf{E} \sim \mathcal{N}(\mathbf{E}; \mathbf{0}, \mathbf{D})$, with $\mathbf{D}_{ik} = \sigma_k^2 \delta_{ik}$. For X distinct time points the overall system may therefore be summarized as:

$$\mathbf{Y} = \mathbf{X} + \mathbf{E},$$

where

$$\mathbf{X} = [\mathbf{x}(t_1), \ldots, \mathbf{x}(t_N)] = [\mathbf{x}_1, \ldots, \mathbf{x}_K]^T,$$

$$\mathbf{Y} = [\mathbf{y}(t_1), \ldots, \mathbf{y}(t_N)] = [\mathbf{y}_1, \ldots, \mathbf{y}_K]^T,$$

and $\mathbf{x}_k = [x_k(t_1), \ldots, x_k(t_N)]^T$ is the k'th state sequence and $\mathbf{y}_k = [y_k(t_1), \ldots, y_k(t_N)]^T$ are the observations. Given the observations $\mathbf{Y}$ and the description of the dynamical system (1), the aim is to estimate both state variables $\mathbf{X}$ and parameters θ.

We consider only dynamical systems that are locally linear with respect to ODE parameters θ and individual states $\mathbf{n}_u$. Such ODEs include mass-action kinetics and are given by:

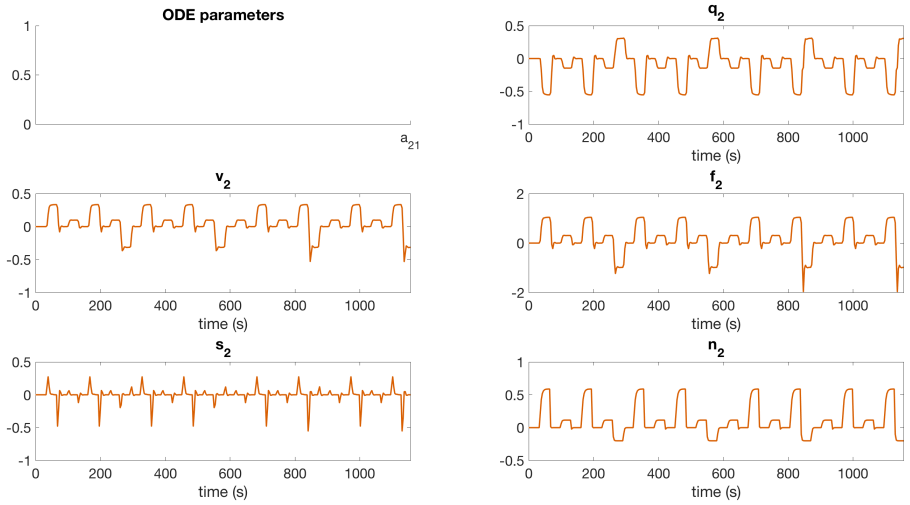
$$f_k(\mathbf{x}(t), \theta) = \sum_{i=1} \theta_{ki} \prod_{j \in \mathcal{M}_{ki}} x_j \qquad (2)$$

with $\mathcal{M}_{ki} \subseteq \{1, \ldots, K\}$ describing the state variables in each factor of the equation (i.e. the functions are linear in parameters and contain arbitrary large products of monomials of the states).

Simulate Trajectory Observations

```
non_diverging_trajectories = false;
while ~non_diverging_trajectories

symbols_simulation = importdata(['dcm/ODEs/'
 simulation.odes '_symbols.mat']);         % symbols of parameters and
 states and in 'ODEs.txt' file
```

```
ode_simulation = import_odes(symbols_simulation,simulation.odes);

simulation.ode_param = -0.8 + (0.8-(-0.8)) *
 rand(1,length(symbols_simulation.param));% true non-selfinhibitory
 neuronal couplings (sampled uniformily in the interval [-0.8,0.8];
simulation.ode_param(end-2:end) = -1;
     % self-inhibitory neuronal couplings set to -1.

state_orig = state;
[state,time,ode_simulation,bold_response] =
 simulate_dynamics_by_numerical_integration(state,time,ode_simulation,simulation,s

if ~any(any(isnan(state.true))) && time.samp(end) > 1000;
 non_diverging_trajectories = 1; end

end


[state,time,obs_to_state_relation] =
 generate_state_observations(state,time,simulation,symbols);

% mean correction
bold_response.obs =
 bsxfun(@minus,bold_response.obs,mean(bold_response.obs,1));


state.sym.mean = sym('n%d%d',[length(time.est),length(ode.system)]);
state.sym.variance = sym('sigma%d%d',
[length(time.est),length(ode.system)]);
ode_param.sym.mean = sym('param%d',[length(symbols.param),1]);
 assume(ode_param.sym.mean,'real');


h_bold =
 setup_plots_for_bold_response_and_ext_input(state,bold_response,time,symbols);
[h_states,h_param,p] = setup_plots_for_states(state,time,symbols);
```



Prior on ODE parameters

Constuct prior on ODE parameters.

```
ode_param =
 prior_on_ODE_param(ode_param,param_prior_variance,symbols.param);%
 prior on ODE parameters
```

Confounding effects

BOLD response observations are given by the signal change equation plus an intercept due to confounding effects:

$$\mathbf{y} = \lambda(\mathbf{q}, \mathbf{v}, \mathbf{u}) + \mathbf{X}\beta + \epsilon$$

```
bold_response = confounding_effects(bold_response);

tic; %start timer
```

Prior on States and State Derivatives

Gradient matching with Gaussian processes assumes a joint Gaussian process prior on states and their derivatives:

$$\begin{pmatrix} \mathbf{X} \\ \dot{\mathbf{X}} \end{pmatrix} \sim p\left(\begin{matrix} \mathbf{X} \\ \dot{\mathbf{X}} \end{matrix} ; \begin{matrix} \mathbf{0} \\ \mathbf{0} \end{matrix}, \begin{matrix} \mathbf{C}_\phi & \mathbf{C}'_\phi \\ {}'\mathbf{C}_\phi & \mathbf{C}''_\phi \end{matrix} \right) \qquad (3)$$

,

$$\mathrm{cov}(x_k(t), x_k(t)) = C_{\phi_k}(t, t')$$

$$\mathrm{cov}(\dot{n}_k(t), x_k(t)) = \frac{\partial C_{\phi_k}(t,t')}{\partial t} =: C'_{\phi_k}(t, t')$$

$$\mathrm{cov}(x_k(t), \dot{n}_k(t)) = \frac{\partial C_{\phi_k}(t,t')}{\partial t'} =: {}'C_{\phi_k}(t, t')$$

$$\mathrm{cov}(\dot{n}_k(t), \dot{n}_k(t)) = \frac{\partial C_{\phi_k}(t,t')}{\partial t \partial t'} =: C''_{\phi_k}(t, t').$$

Matching Gradients

Given the joint distribution over states and their derivatives (3) as well as the ODEs (2), we therefore have two expressions for the state derivatives:

$$\dot{\mathbf{X}} = \mathbf{F} + \epsilon_1, \epsilon_1 \sim \mathcal{X}\left(\epsilon_1; \mathbf{0}, \mathbf{I}\gamma\right)$$

$$\dot{\mathbf{X}} = {}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1}\, \mathbf{X} + \epsilon_2, \epsilon_2 \sim \mathcal{X}\left(\epsilon_2; \mathbf{0}, \mathbf{A}\right)$$

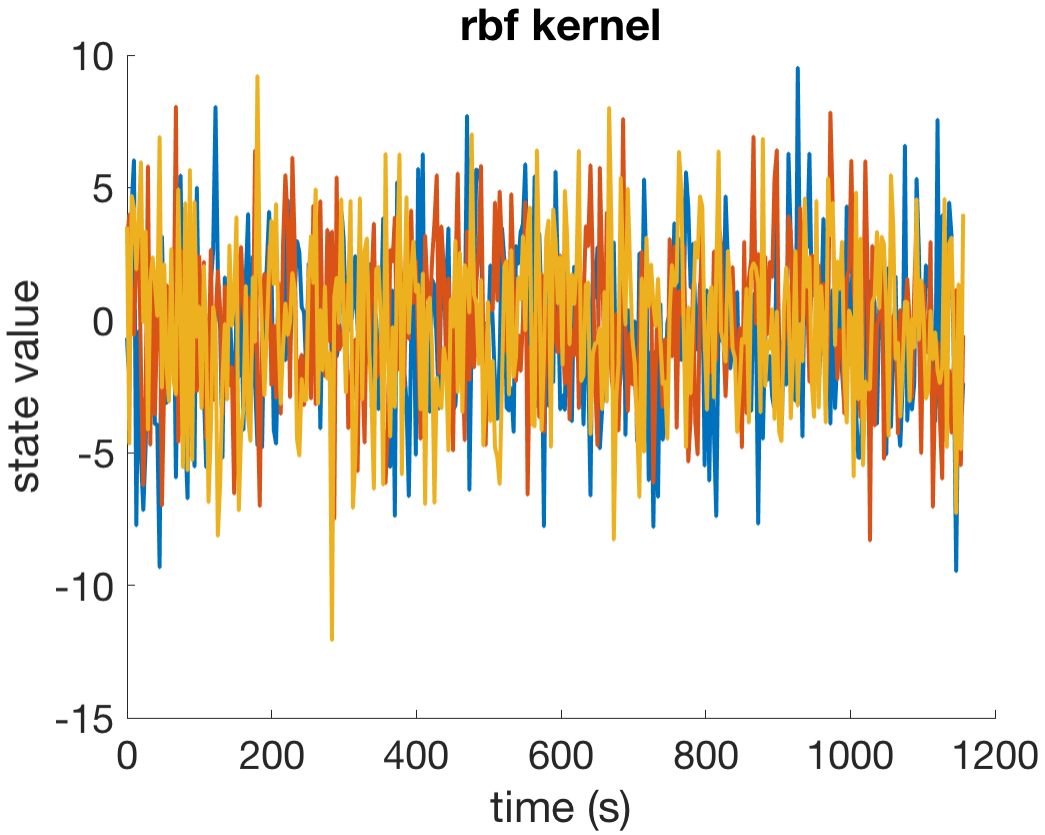
where $\mathbf{F} := \mathbf{f}(\mathbf{X}, \theta)$, $\mathbf{A} := \mathbf{C}''_\phi - {}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{C}'_\phi$ and γ is the error variance in the ODEs. Note that, in a deterministic system, the output of the ODEs $\mathbf{F}$ should equal the state derivatives $\dot{\mathbf{X}}$. However, in the

first equation above we relax this contraint by adding stochasticity to the state derivatives $\dot{\mathbf{X}}$ in order to compensate for a potential model mismatch. The second equation above is obtained by deriving the conditional distribution for $\dot{\mathbf{X}}$ from the joint distribution in equation (3). Equating the two expressions in the equations above we can eliminate the unknown state derivatives $\dot{\mathbf{X}}$:

$$\mathbf{F} = {}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X} + \epsilon_0 \qquad (4)$$

with $\epsilon_0 := \epsilon_2 - \epsilon_1$.

```
[dC_times_invC,inv_C,A_plus_gamma_inv] =
 kernel_function(kernel,state,time.est);
```



State Couplings in ODEs

```
coupling_idx = state_couplings_in_odes(ode,symbols);
```

Rewrite ODEs as Linear Combination in Parameters

We rewrite the ODEs in equation (2) as a linear combination in the parameters:

$$\mathbf{B}_\theta \theta + \mathbf{b}_\theta \stackrel{!}{=} \mathbf{f}(\mathbf{X}, \theta) \qquad (5)$$

where matrices $\mathbf{B}_\theta$ and $\mathbf{b}_\theta$ are defined such that the ODEs $\mathbf{f}(\mathbf{X},\theta)$ are expressed as a linear combination in θ.

```
[ode_param.lin_comb.B,ode_param.lin_comb.b] =
 rewrite_odes_as_linear_combination_in_parameters(ode,symbols);
```

Posterior over ODE Parameters

Inserting (5) into (4) and solving for θ yields:

$$\boldsymbol{\theta} = \mathbf{B}_\theta^+ \left({}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X} - \mathbf{b}_\theta + \boldsymbol{\epsilon}_0 \right),$$

where $\mathbf{B}_\theta^+$ denotes the pseudo-inverse of $\mathbf{B}_\theta$. We can therefore derive the posterior distribution over ODE parameters:

$$p(\boldsymbol{\theta} \mid \mathbf{X}, \boldsymbol{\phi}, \boldsymbol{\gamma}) = \boldsymbol{\mathcal{N}} \left(\boldsymbol{\theta}; \mathbf{B}_\theta^+ \left({}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X} - \mathbf{b}_\theta \right), \mathbf{B}_\theta^+ \left(\mathbf{A} + \mathbf{I}\gamma \right) \mathbf{B}_\theta^{+T} \right) \qquad (6)$$

```
state_enumeration = {'q','v','f','s','n'};
for u = 1:length(state_enumeration)
```

Rewrite Hemodynamic ODEs as Linear Combination in (monotonic functions of) Individual Hemodynamic States

We rewrite the ODE(s) $\mathbf{f}_k(\mathbf{X},\theta)$ as a linear combination in the individual state $\mathbf{x}_u$:

$$\mathbf{B}_{uk}\mathbf{x}_u + \mathbf{b}_{uk} \stackrel{!}{=} \mathbf{f}_k(\mathbf{X},\theta),$$

where matrices $\mathbf{B}_{uk}$ and $\mathbf{b}_{uk}k$ are defined such that the ODE $\mathbf{f}_k(\mathbf{X},\theta)$ is expressed as a linear combination in the individual state $\mathbf{x}_u$.

```
    if strcmp(state_enumeration{u},'q')
```

Rewrite the BOLD signal change equation as a linear combination in a monotonic function of the deoxyhemoglobin content e^q.

$$\mathbf{B}_{q\lambda} e^{\mathbf{q}} + \mathbf{b}_{v\lambda} \stackrel{!}{=} \lambda(q,v),$$

```
       [state.deoxyhemo.B,state.deoxyhemo.b] =
 rewrite_bold_signal_eqn_as_linear_combination_in_deoxyhemo(symbols);

    elseif strcmp(state_enumeration{u},'v')
```

Rewrite the deoxyhemoglobin content ODE as a linear combination in a monotonic function of the blood volume e^v.

$\mathbf{B}_{v\dot{q}}e^{\mathbf{v}} + \mathbf{b}_{v\dot{q}} \stackrel{!}{=} \mathbf{f}_{\dot{q}}(\mathbf{X}, \theta)$.

```
        [state.vol.B,state.vol.b] =
rewrite_deoxyhemo_ODE_as_linear_combination_in_vol(ode,symbols);

    elseif strcmp(state_enumeration{u},'f')
```

Rewrite the blood volume ODE as a linear combination in a monotonic function of the blood flow e^{f}.

$\mathbf{B}_{f\dot{v}}e^{\mathbf{f}} + \mathbf{b}_{f\dot{v}} \stackrel{!}{=} \mathbf{f}_{\dot{v}}(\mathbf{X}, \theta)$

```
        [state.flow.B,state.flow.b] =
rewrite_vol_ODE_as_linear_combination_in_flow(ode,symbols);

    elseif strcmp(state_enumeration{u},'s')
```

Rewrite the blood flow and vasoginalling ODEs as a linear combination in vasosignalling s.

$\mathbf{B}_{sf}\mathbf{s} + \mathbf{b}_{sf} \stackrel{!}{=} \mathbf{f}_{f}(\mathbf{X}, \theta)$

$\mathbf{B}_{s\dot{s}}\mathbf{s} + \mathbf{b}_{s\dot{s}} \stackrel{!}{=} \mathbf{f}_{\dot{s}}(\mathbf{X}, \theta)$

```
        [state.vaso.B,state.vaso.b] =
rewrite_vaso_and_flow_odes_as_linear_combination_in_vaso(ode,symbols);
```

Rewrite Neuronal ODEs as Linear Combination in Individual Neuronal States

We rewrite the ODE(s) $\mathbf{f}_k(\mathbf{X}, \theta)$ as a linear combination in the individual state $\mathbf{x}_u$:

$\mathbf{B}_{uk}\mathbf{x} + \mathbf{b}_{uk} \stackrel{!}{=} \mathbf{f}_{k}(\mathbf{X}, \theta)$

where matrices $\mathbf{B}_{uk}$ and $\mathbf{b}_{uk}k$ are defined such that the expression $\mathbf{f}_k(\mathbf{X}, \theta)$ is expressed as a linear combination in the individual state $\mathbf{x}_u$.

```
    elseif strcmp(state_enumeration{u},'n')
        [state.neuronal.B,state.neuronal.b] =
rewrite_odes_as_linear_combination_in_ind_neuronal_states(ode,symbols,coupling_id
    end

end
```

Posterior over Individual States

Inserting (7) into (4) and solving for $\mathbf{n}_u$ yields:

$$\mathbf{x}_u = \mathbf{B}_u^+ \left(\epsilon_0 - \mathbf{b}_u\right),$$

where $\mathbf{B}_u^+$ denotes the pseudo-inverse of $\mathbf{B}_u$. We can therefore derive the posterior distribution over an individual state $\mathbf{n}_u$:

$$p(\mathbf{x}_u \mid \mathbf{X}_{-u}, \phi, \gamma) = p\left(\mathbf{x}_u; -\mathbf{B}_u^+ \mathbf{b}_u,\ \mathbf{B}_u^+ \left(\mathbf{A} + \mathbf{I}\gamma\right) \mathbf{B}_u^{+T}\right) \qquad (8),$$

with $\mathbf{X}_{-u}$ denoting the set of all states except state $\mathbf{n}_u$.

Mean-field Variational Inference

To infer the parameters θ, we want to find the maximum a posteriori estimate (MAP):

$$\theta^* := arg\max_\theta \ \ln p(\theta \mid \mathbf{Y}, \phi, \gamma, \sigma)$$

$$= arg\max_\theta \ \ln \int p(\theta, \mathbf{X} \mid \mathbf{Y}, \phi, \gamma, \sigma) d\mathbf{X}$$

$$= arg\max_\theta \ \ln \int p(\theta \mid \mathbf{X}, \phi, \gamma) p(\mathbf{X} \mid \mathbf{Y}, \phi, \sigma) d\mathbf{X} \qquad (9).$$

However, the integral above is intractable due to the strong couplings induced by the nonlinear ODEs $\mathbf{f}$ which appear in the term $p(\theta \mid \mathbf{X}, \phi, \gamma)$.

We use mean-field variational inference to establish variational lower bounds that are analytically tractable by decoupling state variables from the ODE parameters as well as decoupling the state variables from each other. Note that, since the ODEs described by equation (2) are **locally linear**, both conditional distributions $p(\theta \mid \mathbf{X}, \mathbf{Y}, \phi, \gamma, \sigma)$ (equation (6)) and $p(\mathbf{n}_u \mid \theta, \mathbf{X}_{-u}, \mathbf{Y}, \phi, \gamma, \sigma)$ (equation (8)) are analytically tractable and Gaussian distributed as mentioned previously.

The decoupling is induced by designing a variational distribution $Q(\theta, \mathbf{X})$ which is restricted to the family of factorial distributions:

$$\mathcal{Q} := \left\{ Q : Q(\theta, \mathbf{X}) = q(\theta) \prod_u q(\mathbf{n}_u) \right\}.$$

The particular form of $q(\theta)$ and $q(\mathbf{n}_u)$ are designed to be Gaussian distributed which places them in the same family as the true full conditional distributions. To find the optimal factorial distribution we minimize the Kullback-Leibler divergence between the variational and the true posterior distribution:

$$\hat{Q} := arg\min_{Q(\theta,\mathbf{X}) \in \mathcal{Q}} \mathrm{KL}\left[Q(\theta, \mathbf{X}) \ || \ p(\theta, \mathbf{X} \mid \mathbf{Y}, \phi, \gamma, \sigma)\right] \qquad (10),$$

where $\hat{Q}$ is the proxy distribution. The proxy distribution that minimizes the KL-divergence (10) depends on the true full conditionals and is given by:

$$\hat{q}(\theta) \propto \exp\left(\ E_{Q_{-\theta}} \ln p(\theta \mid \mathbf{X}, \mathbf{Y}, \phi, \gamma, \sigma)\ \right) \qquad (11)$$

$$\hat{q}(\mathbf{n}_u) \propto \exp\left(\ E_{Q_{-u}} \ln p(\mathbf{n}_u \mid \theta, \mathbf{X}_{-u}, \mathbf{Y}, \phi, \gamma, \sigma)\ \right) \qquad (12).$$

Denoising BOLD Observations

We denoise the BOLD observation by standard GP regression.

```
bold_response.denoised_obs =
 denoising_BOLD_observations(bold_response,inv_C,symbols,simulation.SNR);
```

Fitting Observations of State Trajectories

We fit the observations of state trajectories by standard GP regression. The data-informed distribution $p(\mathbf{X} \mid \mathbf{Y}, \phi, \sigma)$ in euqation (9) can be determined analytically using Gaussian process regression with the GP prior $p(\mathbf{X} \mid \phi) = \prod_k \mathcal{X}(\mathbf{n}_k; \mathbf{0}, \mathbf{C}_\phi)$.

$$p(\mathbf{X} \mid \mathbf{Y}, \phi, \gamma) = \prod_k \mathcal{X}(\mathbf{n}_k; \mu_k(\mathbf{y}_k), \Sigma_k),$$

where $\mu_k(\mathbf{y}_k) := \sigma_k^{-2} \left(\sigma_k^{-2}\mathbf{I} + \mathbf{C}_{\phi_k}^{-1}\right)^{-1} \mathbf{y}_k$ and $\Sigma_k^{-1} := \sigma_k^{-2}\mathbf{I} + \mathbf{C}_{\phi_k}^{-1}$.

```
[mu,inv_sigma] =
 fitting_state_observations(state,inv_C,obs_to_state_relation,symbols,simulation.S
```

Coordinate Ascent Variational Gradient Matching

We **locally** minimize the KL-divergence in equation (10) by coordinate descent (where each step is analytically tractable) by iterating between determining the proxy for the distribution over ODE parameters $\hat{q}(\theta)$ and the proxies for the distribution over individual states $\hat{q}(\mathbf{n}_u)$.

```
bold_response.obs_old = bold_response.denoised_obs;

state_enumeration = {'q','v','f','s','n'};
state_enumeration(find(ismember(state_enumeration,simulation.observed_states)))
 = [];

ode_param.proxy.mean = zeros(length(symbols.param),1);
state.proxy.mean = mu;

for i=1:opt_settings.coord_ascent_numb_iter
```

Intercept due to Confounding Effects

The intercept is determined by a minimum least squares estimator:

$$\mathbf{X}\hat{\beta} := \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{y} - \mathbf{h}(\mathbf{q}, \mathbf{v}, \mathbf{u}))$$

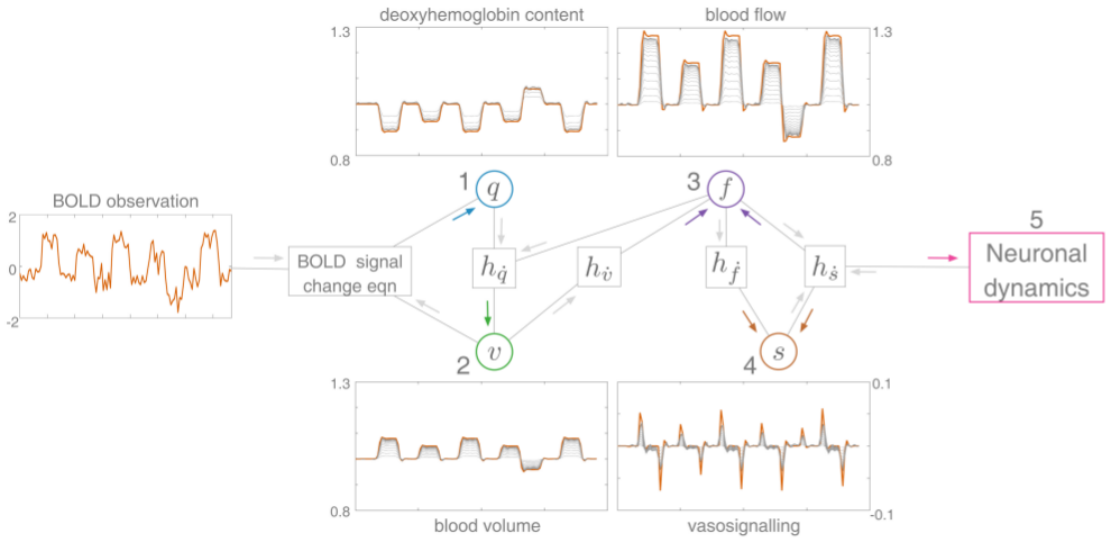
```
    vol_idx = cellfun(@(n) strcmp(n(2),'v'),symbols.state);
    deoxyhemo_idx = cellfun(@(n) strcmp(n(2),'q'),symbols.state);
```

```
    bold_response_signal_change =
 bold_signal_change_eqn(state.proxy.mean(:,vol_idx),state.proxy.mean(:,deoxyhemo_i
    bold_response.confounding_effects.intercept
 = determine_intercept(bold_response.obs_old-
bold_response_signal_change,...

 bold_response.confounding_effects.X0,bold_response.confounding_effects.X0_penrose

    bold_response.confounding_effects.intercept =
 zeros(size(bold_response.obs,1),size(bold_response.obs,2));
    bold_response.denoised_obs = bold_response.obs_old -
 bold_response.confounding_effects.intercept;

    for j = 1:length(state_enumeration)
```

Proxies for Hemodynamic States

Determine the proxies for the states, starting with deoxyhemoglobin followed by blood volume, blood flow and finally vasosignalling.

The information flow in the hemodynamic system is shown in its factor graph below:

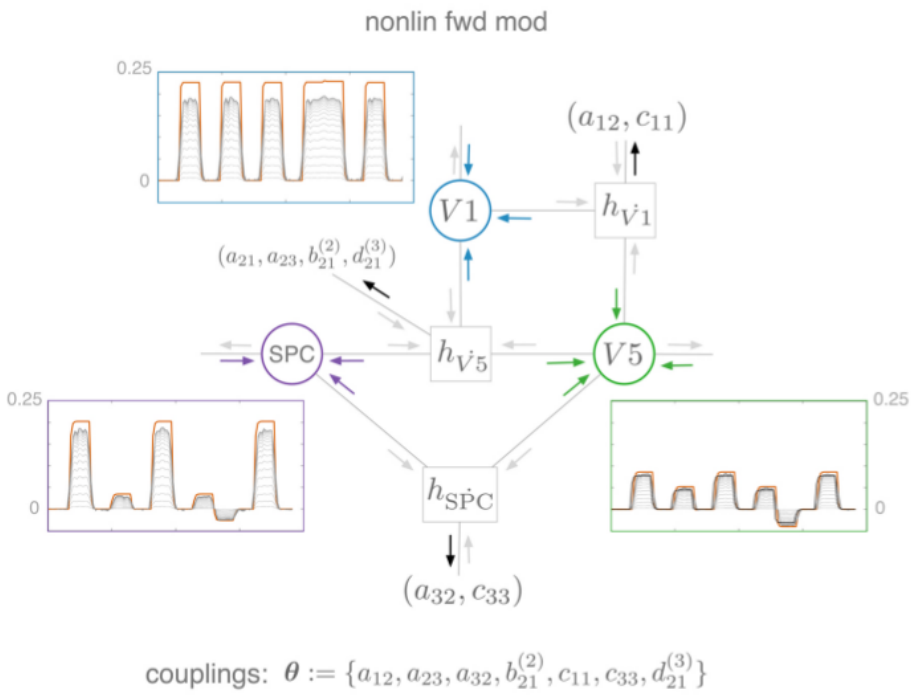


The model inversion in the hemodynmic factor graph above occurs locally w.r.t. individual states. Given the expression for the BOLD signal change equation, we invert the BOLD signal change equation analytically to determine the deoxyhemoglobin content q (1). The newly inferred deoxyhemoglobin content q influences the expression for the factor associated with the change in deoxyhemoglobin content $h_{\dot{q}}$, which we subsequently invert analytically to infer the blood volume v (2). Thereafter, we infer the blood flow f (3) by inverting the factors associated with the change in blood volume $h_{\dot{v}}$ as well as vasosignalling $h_{\dot{s}}$, followed by inferring vasosignalling s (4) by inverting the factors associated with blood flow induction $h_{\dot{f}}$ and vasosignalling $h_{\dot{s}}$. Finally, the neuronal dynamics (5) are learned, in part, by inverting the factor associated with vasosignalling $h_{\dot{s}}$. The typical trajectories of each of the states are shown (red) together with their iterative approximation (grey lines) obtained by graphical DCM.

```
        if strcmp(state_enumeration{j},'q')


            state_idx = cellfun(@(n) strcmp(n(2),'q'),symbols.state);
            state_tmp =
proxy_for_deoxyhemoglobin_content(state.deoxyhemo,state.proxy.mean,...

bold_response.denoised_obs,symbols,A_plus_gamma_inv,opt_settings);
            state.proxy.mean(:,state_idx) = (1-damping) *
state.proxy.mean(:,state_idx) + damping * state_tmp;

        elseif strcmp(state_enumeration{j},'v')


            state_idx = cellfun(@(n) strcmp(n(2),'v'),symbols.state);
            state_tmp =
proxy_for_blood_volume(state.vol,dC_times_invC,state.proxy.mean,...

ode_param.proxy.mean,symbols,A_plus_gamma_inv,opt_settings);
            state.proxy.mean(:,state_idx) = (1-damping) *
state.proxy.mean(:,state_idx) + damping * state_tmp;

        elseif strcmp(state_enumeration{j},'f')


            state_idx = cellfun(@(n) strcmp(n(2),'f'),symbols.state);
            state_tmp =
proxy_for_blood_flow(state.flow,dC_times_invC,state.proxy.mean,...

ode_param.proxy.mean,symbols,A_plus_gamma_inv,opt_settings);
            state.proxy.mean(:,state_idx) = (1-damping) *
state.proxy.mean(:,state_idx) + damping * state_tmp;

        elseif strcmp(state_enumeration{j},'s')


            state_idx = cellfun(@(n) strcmp(n(2),'s'),symbols.state);
            state.proxy.mean(:,state_idx) =
proxy_for_vasosignalling(state.vaso,dC_times_invC,...

state.proxy.mean,ode_param.proxy.mean,symbols,A_plus_gamma_inv,opt_settings);

        elseif strcmp(state_enumeration{j},'n')
```

Proxies for Neuronal States

Determine the proxies for the neuronal states. An example of the information flow in the neuronal part of the nonlinear forward modulating (nonlin_fwd_mod) is shown in its factor graph below:



In the neuronal factor graph (for the nonlinear forwad modulation) above each individual state appears linear in every factor in the neuronal model. We can therefore analytically invert every factor to determine the neuronal state. The typical trajectories of each of the states are shown (red) together with their iterative approximation (grey lines) obtained by variational gradient matching.

```
            state_idx = cellfun(@(n) strcmp(n(2),'n'),symbols.state);
            state.proxy.mean(:,state_idx) =
proxy_for_neuronal_populations(state.neuronal,state.proxy.mean,ode_param.proxy.me

dC_times_invC,coupling_idx.states,symbols,A_plus_gamma_inv,opt_settings);

        end
        state.proxy.mean(:,1:15) =
bsxfun(@minus,state.proxy.mean(:,1:15),state.proxy.mean(1,1:15));

    end

    if i==1 || ~mod(i,20);
plot_results_for_states(h_states,h_param,state,time,simulation,ode_param.proxy.me
```

Proxy for neuronal couplings (ODE parameters)

```
    if i>200 || i==coord_ascent_numb_iter
        [ode_param.proxy.mean,ode_param.proxy.inv_cov] =
proxy_for_ode_parameters(state.proxy.mean,dC_times_invC,ode_param.lin_comb,symbol
    end

end
```

Numerical integration with parameters estimated by variational gradient matching

See whether we actually fit the BOLD response well. Curves are shown in black.

```
[state,bold_response] =
 simulate_trajectory_with_vgm_param_est(ode_param,state,state_orig,bold_response,s
```

Final result

```
plot_results_for_bold_response(h_bold,bold_response,time);
plot_results_for_states(h_states,h_param,state,time,simulation,ode_param.proxy.mea
```

Time Taken

```
disp(['time taken: ' num2str(toc) ' seconds'])
```

References

- **Gorbach, X.S.** , **Bauer, S.** and Buhmann, J.M., Scalable Variational Inference for Dynamical Systems. 2017a. Neural Information Processing Systems (NIPS). https://papers.nips.cc/paper/7066-scalable-variational-inference-for-dynamical-systems.pdf, arxiv: https://arxiv.org/abs/1705.07079.
- **Bauer, S.** , **Gorbach, X.S.** and Buhmann, J.M., Efficient and Flexible Inference for Stochastic Differential Equations. 2017b. Neural Information Processing Systems (NIPS). https://papers.nips.cc/paper/7274-efficient-and-flexible-inference-for-stochastic-systems.pdf
- Wenk, P., Gotovos, A., Bauer, S., Gorbach, X.S., Krause, A. and Buhmann, J.M., Fast Gaussian Process Based Gradient Matching for Parameters Identification in Systems of Nonlinear ODEs. 2018. In submission to Conference on Uncertainty in Artificial Intelligence (UAI).
- Calderhead, B., Girolami, M. and Lawrence. X.D., 2002. Accelerating Bayesian inference over nonlinear differential equation models. *In Advances in Neural Information Processing Systems (NIPS)* . 22.

The authors in bold font have contributed equally to their respective papers.

Subroutines

Gradient matching with Gaussian processes assumes a joint Gaussian process prior on states and their derivatives:

$$\begin{pmatrix} \mathbf{X} \\ \dot{\mathbf{X}} \end{pmatrix} \sim p\left(\begin{pmatrix} \mathbf{X} \\ \dot{\mathbf{X}} \end{pmatrix}; \begin{matrix} \mathbf{0} \\ \mathbf{0} \end{matrix}, \begin{matrix} \mathbf{C}_\phi & \mathbf{C}'_\phi \\ {}'\mathbf{C}_\phi & \mathbf{C}''_\phi \end{matrix} \right),$$

$$\mathrm{cov}(x_k(t), x_k(t)) = C_{\phi_k}(t,t')$$

$$\mathrm{cov}(\dot{n}_k(t), x_k(t)) = \frac{\partial C_{\phi_k}(t,t')}{\partial t} =: C'_{\phi_k}(t,t')$$

$$\mathrm{cov}(x_k(t), \dot{n}_k(t)) = \frac{\partial C_{\phi_k}(t,t')}{\partial t'} =: {}'C_{\phi_k}(t,t')$$

$\mathrm{cov}(\dot{n}_k(t), \dot{n}_k(t)) = \frac{\partial C_{\phi_k}(t,t')}{\partial t \partial t'} =: C''_{\phi_k}(t,t')$.

```
function [dC_times_invC,inv_C,A_plus_gamma] =
 kernel_function(kernel,state,time_est)

kernel.param_sym = sym('rbf_param%d',[1,2]);
 assume(kernel.param_sym,'real');
kernel.time1 = sym('time1'); assume(kernel.time1,'real'); kernel.time2
 = sym('time2'); assume(kernel.time2,'real');
kernel.func = kernel.param_sym(1).*exp(-(kernel.time1-
kernel.time2).^2./(kernel.param_sym(2).^2));                    %
 RBF kernel
kernel.name = 'rbf';

% kernel derivatives
for i = 1:length(kernel)
    kernel.func_d = diff(kernel.func,kernel.time1);
    kernel.func_dd = diff(kernel.func_d,kernel.time2);
    GP.fun = matlabFunction(kernel.func,'Vars',
{kernel.time1,kernel.time2,kernel.param_sym});
    GP.fun_d = matlabFunction(kernel.func_d,'Vars',
{kernel.time1,kernel.time2,kernel.param_sym});
    GP.fun_dd = matlabFunction(kernel.func_dd,'Vars',
{kernel.time1,kernel.time2,kernel.param_sym});
end

% populate GP covariance matrix
for t=1:length(time_est)
    C(t,:)=GP.fun(time_est(t),time_est,kernel.param);
    dC(t,:)=GP.fun_d(time_est(t),time_est,kernel.param);
    Cd(t,:)=GP.fun_d(time_est,time_est(t),kernel.param);
    ddC(t,:)=GP.fun_dd(time_est(t),time_est,kernel.param);
end

% GP covariance scaling
[~,D] = eig(C); perturb = abs(max(diag(D))-min(diag(D))) / 10000;
if any(diag(D)<1e-6); C(logical(eye(size(C,1)))) =
 C(logical(eye(size(C,1)))) + perturb.*rand(size(C,1),1); end
[~,D] = eig(C);
if any(diag(D)<0); error('C has negative eigenvalues!'); elseif
 any(diag(D)<1e-6); warning('C is badly scaled'); end
inv_C = inv_chol(chol(C,'lower'));

dC_times_invC = dC * inv_C;

% plot GP prior samples
figure(3);
hold on;
 plot(time_est,mvnrnd(zeros(1,length(time_est)),C(:,:,1),3),'LineWidth',2);
h1 = gca; h1.FontSize = 20; h1.XLabel.String = 'time (s)';
 h1.YLabel.String = 'state value';
h1.Title.String = [kernel.name ' kernel'];
```

```
% determine A_plus_gamma:
A = ddC - dC_times_invC * Cd;
A_plus_gamma = A + state.derivative_variance(1) .* eye(size(A));
A_plus_gamma = 0.5.*(A_plus_gamma+A_plus_gamma');      % ensure that A
 plus gamma is symmetric
%A_plus_gamma_inv = inv_chol(chol(inv_Lambda,'lower'));

end
```

We denoise the BOLD observation by standard GP regression.

$$p(\mathbf{X} \mid \mathbf{Y}, \phi, \gamma) = \prod_k \mathcal{X}(\mathbf{n}_k; \mu_k(\mathbf{y}_k), \Sigma_k),$$

where $\mu_k(\mathbf{y}_k) := \sigma_k^{-2}\left(\sigma_k^{-2}\mathbf{I} + \mathbf{C}_{\phi_k}^{-1}\right)^{-1}\mathbf{y}_k$ and $\Sigma_k^{-1} := \sigma_k^{-2}\mathbf{I} + \mathbf{C}_{\phi_k}^{-1}$.

```
function [mu,inv_sigma] =
 denoising_BOLD_observations(bold_response,inv_Cxx,symbols,SNR)

inv_Cxx_cell = num2cell(inv_Cxx(:,:,ones(1,sum(cellfun(@(n)
 strcmp(n(2),'n'),symbols.state)))),[1,2]);
inv_Cxx_blkdiag = blkdiag(inv_Cxx_cell{:});

b = repmat(var(bold_response.obs)./SNR,size(bold_response.obs,1),1);
dim = size(inv_Cxx_blkdiag,1);
D = spdiags(reshape(b.^(-1),[],1),0,dim,dim) * speye(dim); %
 covariance matrix of error term (big E)
inv_sigma = D + inv_Cxx_blkdiag;

mu = inv_sigma \ D * reshape(bold_response.obs,[],1);
mu = reshape(mu,[],size(bold_response.obs,2));

end
```

We fit the observations of state trajectories by standard GP regression.

```
function [mu_u,inv_sigma_u] =
 fitting_state_observations(state,inv_C,obs_to_state_relation,symbols,SNR)

state_obs_variance = 1e0*repmat(var(state.obs) ./
 SNR,size(state.obs,1),1);

numb_states = size(state.sym.mean,2);
numb_time_points = size(state.sym.mean,1);

inv_Cxx_tmp = num2cell(inv_C(:,:,ones(1,numb_states)),[1,2]);
inv_Cxx_blkdiag = sparse(blkdiag(inv_Cxx_tmp{:}));

dim = size(state_obs_variance,1)*size(state_obs_variance,2);
D = spdiags(reshape(state_obs_variance.^(-1),[],1),0,dim,dim) *
 speye(dim); % covariance matrix of error term (big E)
A_times_D_times_A = obs_to_state_relation' * D *
 obs_to_state_relation;
```

```
inv_sigma = A_times_D_times_A + inv_Cxx_blkdiag;

mu = inv_sigma \ obs_to_state_relation' * D * reshape(state.obs,[],1);

mu_u = zeros(numb_time_points,numb_states);
for u = 1:numb_states
    idx = (u-1)*numb_time_points+1:(u-1)*numb_time_points
+numb_time_points;
    mu_u(:,u) = mu(idx);
end

inv_sigma_u = zeros(numb_time_points,numb_time_points,numb_states);
for i = 1:numb_states
    idx = [(i-1)*numb_time_points+1:(i-1)*numb_time_points
+numb_time_points];
    inv_sigma_u(:,:,i) = inv_sigma(idx,idx);
end

% external_input
ext_input_idx = cellfun(@(n) strcmp(n(2),'u'),symbols.state);
mu_u(:,ext_input_idx) =
 state.ext_input(state.ext_input_to_bold_response_mapping_idx,2:end);

end


function coupling_idx = state_couplings_in_odes(ode,symbols)

state_sym = sym('state%d',[1,length(ode.system)]);
 assume(state_sym,'real');
for k = 1:length(ode.system)
    tmp_idx = ismember(state_sym,symvar(ode.system_sym(k)));
 tmp_idx(:,k) = 1;
    ode_couplings_states(k,tmp_idx) = 1;
end

for u = find(cellfun(@(x) ~strcmp(x(2),'u'),symbols.state))
    coupling_idx.states{u} = find(ode_couplings_states(:,u));
end

end
```

$$\mathbf{B}_\theta \theta + \mathbf{b}_\theta \stackrel{!}{=} \mathbf{f}(\mathbf{X}, \theta),$$

where matrices $\mathbf{B}_\theta$ and $\mathbf{b}_\theta$ are defined such that the ODEs $\mathbf{f}(\mathbf{X}, \theta)$ are expressed as a linear combination in θ.

```
function [B,b] =
 rewrite_odes_as_linear_combination_in_parameters(ode,symbols)

param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');
```

```
state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');
state0_sym = sym('state0'); assume(state0_sym,'real');
state_const_sym = sym('state_const'); assume(state_const_sym,'real');

% Rewrite ODEs as linear combinations in parameters
[B_sym,b_sym] = equationsToMatrix(ode.system_sym,param_sym);
b_sym = -b_sym; % See the documentation of the function
 "equationsToMatrix"

% Product of ODE factors (product of Gaussians)
for k = 1:length(ode.system)
    B_sym(k,B_sym(k,:)=='0') = state0_sym;
    for i = 1:length(B_sym(k,:))
        sym_var = symvar(B_sym(k,i));
        if isempty(sym_var)
            B_sym(k,i) = B_sym(k,i) + state0_sym;
        end
    end
    B{k} = matlabFunction(B_sym(k,:),'Vars',
{state_sym,state0_sym,state_const_sym});
    b{k} = matlabFunction(b_sym(k,:),'Vars',
{state_sym,state0_sym,state_const_sym});
end

end
```

$$\mathbf{B}_{q\lambda} e^{\mathbf{q}} + \mathbf{b}_{v\lambda} \stackrel{!}{=} \lambda(q,v)$$

```
function [B,b] =
 rewrite_bold_signal_eqn_as_linear_combination_in_deoxyhemo(symbols)

% define symbolic variables
param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');
state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');
v = sym('v'); assume(v,'real');
q = sym('q'); assume(q,'real');
exp_q = sym('exp_q'); assume(exp_q,'real');

% bold signal change equation
bold_signal_change = bold_signal_change_eqn(v,q);
[B_sym,b_sym] =
 equationsToMatrix(subs(bold_signal_change,exp(q),exp_q),exp_q);
b_sym = -b_sym; % See the documentation of the function
 "equationsToMatrix"

B = matlabFunction(B_sym,'Vars',{v,q});
b = matlabFunction(b_sym,'Vars',{v,q});

end
```

$$\mathbf{B}_{v\dot{q}}e^{\mathrm{v}} + \mathbf{b}_{v\dot{q}} \stackrel{!}{=} \mathbf{f}_{\dot{q}}(\mathbf{X},\theta) - {}'\mathbf{C}_{\phi_{\dot{q}}}\mathbf{C}_{\phi_{\dot{q}}}^{-1}\mathbf{X}.$$

```
function [B,b] =
 rewrite_deoxyhemo_ODE_as_linear_combination_in_vol(ode,symbols)

% define symbolic variables
param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');
state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');
exp_v = sym('exp_v'); assume(exp_v,'real');

state_idx = find(cellfun(@(n) strcmp(n(2),'v'),symbols.state));

% deoxyhemoglobin ODE
ode_idx = find(cellfun(@(n) strcmp(n(2),'q'),symbols.state));
j = 0;
for u = state_idx
    j = j+1;
    [B_sym,b_sym] = equationsToMatrix(subs(ode.system{ode_idx(j)}
(state_sym,param_sym),exp((17*state_sym(u)/8)),exp_v),exp_v);
    b_sym = -b_sym; % See the documentation of the function
 "equationsToMatrix"

    B{u} = matlabFunction(B_sym,'Vars',{state_sym,param_sym});
    b{u} = matlabFunction(b_sym,'Vars',{state_sym,param_sym});
end

end
```

$$\mathbf{B}_{f\dot{v}}e^{\mathrm{f}} + \mathbf{b}_{f\dot{v}} \stackrel{!}{=} \mathbf{f}_{\dot{v}}(\mathbf{X},\theta) - {}'\mathbf{C}_{\phi_{\dot{v}}}\mathbf{C}_{\phi_{\dot{v}}}^{-1}\mathbf{X}$$

```
function [B,b] =
 rewrite_vol_ODE_as_linear_combination_in_flow(ode,symbols)

% define symbolic variables
param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');
state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');
exp_f = sym('exp_f'); assume(exp_f,'real');

state_idx = find(cellfun(@(n) strcmp(n(2),'f'),symbols.state));

% blood volume ODE
ode_idx = find(cellfun(@(n) strcmp(n(2),'v'),symbols.state));

j = 0;
for u = state_idx
    j = j+1;
    [B_sym,b_sym] = equationsToMatrix(subs(ode.system{ode_idx(j)}
(state_sym,param_sym),exp(state_sym(u)),exp_f),exp_f);
```

```
    b_sym = -b_sym; % See the documentation of the function
 "equationsToMatrix"

    B{u} = matlabFunction(B_sym,'Vars',{state_sym,param_sym});
    b{u} = matlabFunction(b_sym,'Vars',{state_sym,param_sym});
end

end
```

$$\mathbf{B}_{s\dot{s}}\mathbf{s} + \mathbf{b}_{s\dot{s}} \stackrel{!}{=} \mathbf{f}_{\dot{s}}(\mathbf{X}, \theta) - {}'\mathbf{C}_{\phi_{\dot{s}}}\mathbf{C}_{\phi_{\dot{s}}}^{-1}\mathbf{X}$$

$$\mathbf{B}_{s\dot{f}}\mathbf{s} + \mathbf{b}_{s\dot{f}} \stackrel{!}{=} \mathbf{f}_{\dot{f}}(\mathbf{X}, \theta) - {}'\mathbf{C}_{\phi_{\dot{f}}}\mathbf{C}_{\phi_{\dot{f}}}^{-1}\mathbf{X}$$

```
function [B,b] =
 rewrite_vaso_and_flow_odes_as_linear_combination_in_vaso(ode,symbols)

% define symbolic variables
param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');
state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');

state_idx = find(cellfun(@(n) strcmp(n(2),'s'),symbols.state));

% vasosignaling ODE
ode_idx = find(cellfun(@(n) strcmp(n(2),'s'),symbols.state));
j = 0;
for u = state_idx
    j = j+1;
    [B_sym,b_sym] = equationsToMatrix(ode.system{ode_idx(j)}
(state_sym,param_sym),state_sym(u));
    b_sym = -b_sym; % See the documentation of the function
 "equationsToMatrix"

    B{u}.vaso = matlabFunction(B_sym,'Vars',{state_sym,param_sym});
    b{u}.vaso = matlabFunction(b_sym,'Vars',{state_sym,param_sym});
end

% blood flow ODE
ode_idx = find(cellfun(@(n) strcmp(n(2),'f'),symbols.state));
j = 0;
for u = state_idx
    j = j+1;
    [B_sym,b_sym] = equationsToMatrix(ode.system{ode_idx(j)}
(state_sym,param_sym),state_sym(u));
    b_sym = -b_sym; % See the documentation of the function
 "equationsToMatrix"

    B{u}.flow = matlabFunction(B_sym,'Vars',{state_sym,param_sym});
    b{u}.flow = matlabFunction(b_sym,'Vars',{state_sym,param_sym});
end
```

```
end
```

$$\mathbf{B}_{uk}\mathbf{x} + \mathbf{b}_{uk} \stackrel{!}{=} \mathbf{f}_k(\mathbf{X},\theta) - {}'\mathbf{C}_{\phi_k}\mathbf{C}_{\phi_k}^{-1}\mathbf{X}$$

```
function [B,b]=
 rewrite_odes_as_linear_combination_in_ind_neuronal_states(ode,symbols,coupling_id

state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');
param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');

state_idx = find(cellfun(@(n) strcmp(n(2),'n'),symbols.state));

for u = state_idx
    for k = coupling_idx{u}'
        [B_sym,b_sym] = equationsToMatrix(ode.system{k}
(state_sym,param_sym'),state_sym(:,u));
        b_sym = -b_sym; % See the documentation of the function
 "equationsToMatrix"

        B{u,k} = matlabFunction(B_sym,'Vars',{state_sym,param_sym});
        b{u,k} = matlabFunction(b_sym,'Vars',{state_sym,param_sym});
    end
end

end
```

$$\hat{q}(\theta) \propto \exp\left(E_{Q_{-\theta}} \ln \mathcal{X}\left(\theta; \mathbf{B}_\theta^+ \left({}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1}\mathbf{X} - \mathbf{b}_\theta\right),\ \mathbf{B}_\theta^+ \left(\mathbf{A} + \mathbf{I}\gamma\right) \mathbf{B}_\theta^{+T}\right)\right),$$

```
function [param_proxy_mean,param_proxy_inv_cov] =
 proxy_for_ode_parameters(state_proxy_mean,dC_times_invC,lin_comb,symbols,ode_para

state0 = zeros(size(dC_times_invC,1),1);
param_proxy_inv_cov = zeros(length(symbols.param));
local_mean_sum = zeros(length(symbols.param),1);
for k = 1: 1:sum(cellfun(@(n) ~strcmp(n(2),'u'),symbols.state))
    B = lin_comb.B{k}
(state_proxy_mean,state0,ones(size(state_proxy_mean,1),1));
    b = lin_comb.b{k}
(state_proxy_mean,state0,ones(size(state_proxy_mean,1),1));

    local_mean = B' * A_plus_gamma_inv * (dC_times_invC *
 state_proxy_mean(:,k) - b);
    local_inv_cov = B' * A_plus_gamma_inv * B;
    % The next two code lines (instead of the top two) works also:
    % local_mean = B' * (dC_times_invC * state_proxy_mean(:,k) - b);
    % local_inv_cov = B' * B;¨

    local_mean_sum = local_mean_sum + local_mean;
    param_proxy_inv_cov = param_proxy_inv_cov + local_inv_cov;
```

```
end

if isfield(ode_param,'prior')
    local_mean_sum = local_mean_sum +
 ode_param.prior.inv_cov*ode_param.prior.mean;
    param_proxy_inv_cov = param_proxy_inv_cov +
 ode_param.prior.inv_cov;
end

% Check consistency of covariance matrix
[~,D] = eig(param_proxy_inv_cov);
if any(diag(D)<0)
    warning('ode_param.proxy.inv_cov has negative eigenvalues!');
elseif any(diag(D)<1e-3)
    warning('ode_param.proxy.inv_cov is badly scaled')
    disp('perturbing diagonal of ode_param.proxy.inv_cov')
    perturb = abs(max(diag(D))-min(diag(D))) / 10000;
    param.proxy.inv_cov(logical(eye(size(param_proxy_inv_cov,1)))) =
 param_proxy_inv_cov(logical(eye(size(param_proxy_inv_cov,1)))) ...
        + perturb.*rand(size(param_proxy_inv_cov,1),1);
end
```

We approximate $\left(\mathbf{B}(\mathbf{A}+\mathbf{I}\gamma)\mathbf{B}^T\right)^{-1}$

```
param_proxy_mean = pinv(param_proxy_inv_cov) * local_mean_sum;

end


function [deoxyhemo_proxy_mean,deoxyhemo_proxy_inv_cov] =
 proxy_for_deoxyhemoglobin_content(deoxyhemo,state,...
    bold_response_obs,symbols,A_plus_gamma_inv,opt_settings)

state_idx = find(cellfun(@(x) strcmp(x(2),'q'),symbols.state));
state_partner_idx = find(cellfun(@(x)
 strcmp(x(2),'v'),symbols.state));

% Initialize
j = 0;
global_scaling = zeros(length(symbols.param));
global_mean = zeros(length(symbols.param),1);

% Iteratate through ODEs
for u = state_idx

    % unpack matrices B and b
    j = j+1;
    B = diag(deoxyhemo.B(state(:,state_partner_idx(j))));
    b = deoxyhemo.b(state(:,state_partner_idx(j)));

    if strcmp(opt_settings.pseudo_inv_type,'Moore-Penrose')
        local_scaling = B' * B;
        local_mean =  B' * (bold_response_obs(:,u) - b);
        local_inv_cov = B' * A_plus_gamma_inv * B;
```

```
    elseif strcmp(opt_settings.pseudo_inv_type,'modified Moore-
Penrose')
        local_scaling = B' * A_plus_gamma_inv * B;
        local_mean =  B' * A_plus_gamma_inv * (bold_response_obs(:,u)
 - b);
        local_inv_cov = scaling;
    end

    % global
    global_mean = global_mean + local_mean;
    global_scaling = global_scaling + local_scaling;

    deoxyhemo_proxy_mean(:,u) = log(global_scaling \ local_mean_sum);
    % Check if the deoxyhemoglobin content is positive
    if any(~isreal(deoxyhemo_proxy_mean(:,u)))
        disp('warning: deoxyhemoglobin is not positive')
        deoxyhemo_proxy_mean(:,u) = real(deoxyhemo_proxy_mean(:,u));
    end

end

end
```

```
Undefined function or variable 'scaling'.

Error in dynamic_causal_models2>proxy_for_deoxyhemoglobin_content
 (line 883)
        local_inv_cov = scaling;

Error in dynamic_causal_models2 (line 400)
            state_tmp =
 proxy_for_deoxyhemoglobin_content(state.deoxyhemo,state.proxy.mean,...
```

```
function vol_proxy_mean =
 proxy_for_blood_volume(vol,dC_times_invC,state,ode_param,symbols,...
    A_plus_gamma_inv,opt_settings)

state_idx = find(cellfun(@(x) strcmp(x(2),'v'),symbols.state));
state_partner_idx = find(cellfun(@(x)
 strcmp(x(2),'q'),symbols.state));

% Initialize
j = 0;
global_scaling = zeros(length(symbols.param));
global_mean = zeros(length(symbols.param),1);

% Iteratate through ODEs
for u = state_idx

    % unpack matrices B and b
    j = j+1;
    B = diag(vol.B{u}(state,ode_param'));
    b = vol.b{u}(state,ode_param);
    if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end
```

```
    if strcmp(opt_settings.pseudo_inv_type,'Moore-Penrose')
        local_scaling = B' * B;
        local_mean = B' * (dC_times_invC *
 state_proxy_mean(:,state_partner_idx(j)) - b);
        local_inv_cov = B' * A_plus_gamma_inv * B;
    elseif strcmp(opt_settings.pseudo_inv_type,'modified Moore-
Penrose')
        local_scaling = B' * A_plus_gamma_inv * B;
        local_mean = B' * A_plus_gamma_inv * (dC_times_invC *
 state_proxy_mean(:,state_partner_idx(j)) - b);
        local_inv_cov = scaling;
    end

    % global
    global_mean = global_mean + local_mean;
    global_scaling = global_scaling + local_scaling;

    vol_proxy_mean(:,u) = log(global_scaling \ local_mean_sum);
    % Check if the deoxyhemoglobin content is positive
    if any(~isreal(vol_proxy_mean(:,u)))
        disp('warning: deoxyhemoglobin is not positive')
        vol_proxy_mean(:,u) = real(vol_proxy_mean(:,u));
    end

end

end


function [flow_proxy_mean,flow_proxy_inv_cov] =
 proxy_for_blood_flow(flow,dC_times_invC,state,ode_param,symbols,A_plus_gamma_inv)

state_idx = find(cellfun(@(x) strcmp(x(2),'f'),symbols.state));
state_partner_idx = find(cellfun(@(x)
 strcmp(x(2),'s'),symbols.state));

j = 0;
for u = state_idx

    % unpack matrices B and b
    B = diag(flow.B{u}(state,ode_param'));
    b = flow.b{u}(state,ode_param);
    if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end

    j = j + 1;
    local_mean_sum = B' * A_plus_gamma_inv * ( dC_times_invC *
 state(:,state_partner_idx(j)) - b );
    flow_proxy_inv_cov = B' * A_plus_gamma_inv * B;
    % The next two code lines (instead of the top two) works also:
    % local_mean_sum = B' * ( dC_times_invC *
 state(:,state_partner_idx(j)) - b );
    % flow_proxy_inv_cov = B' * B;
```

```
        flow_proxy_mean(:,j) = real(log(flow_proxy_inv_cov \
 local_mean_sum));

        if any(~isreal(flow_proxy_mean))
            disp('warning: flow is not positive')
            flow_proxy_mean(:,j) = real(flow_proxy_mean(:,j));
        end
    end

    end


    function [vaso_proxy_mean,vaso_proxy_inv_cov] =
     proxy_for_vasosignalling(vaso,dC_times_invC,state,ode_param,symbols,A_plus_gamma_

    state_idx = find(cellfun(@(x) strcmp(x(2),'s'),symbols.state));
    state_partner_idx = find(cellfun(@(x)
     strcmp(x(2),'f'),symbols.state));

    j = 0;
    for u = state_idx

        % unpack matrices B and b for vasosignalling ODE
        B = diag(vaso.B{u}.vaso(state,ode_param'));
        b = vaso.b{u}.vaso(state,ode_param);
        if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end
        B = B -dC_times_invC;

        local_mean_vaso_ode = -B' * A_plus_gamma_inv * b;
        vaso_proxy_inv_cov_vaso_ode = B' * A_plus_gamma_inv * B;
        % The next two code lines (instead of the top two) works also:
        % local_mean_vaso_ode = -B' * b;
        % vaso_proxy_inv_cov_vaso_ode = B' * B;

        % unpack matrices B and b for blood flow ODE
        B = diag(vaso.B{u}.flow(state,ode_param'));
        b = vaso.b{u}.flow(state,ode_param);
        if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end
        if size(b,1) == 1; b = b.*zeros(size(dC_times_invC,1),1); end

        j = j+1;
        local_mean_flow_ode = B' * A_plus_gamma_inv * (dC_times_invC *
     state(:,state_partner_idx(j)) - b);
        vaso_proxy_inv_cov_flow_ode = B' * A_plus_gamma_inv * B;
        % The next two code lines (instead of the top two) works also:
        % local_mean_flow_ode = B' * (dC_times_invC *
     state(:,state_partner_idx(j)) - b);
        % vaso_proxy_inv_cov_flow_ode = B' * B;

        % combined
        vaso_proxy_inv_cov = vaso_proxy_inv_cov_vaso_ode +
     vaso_proxy_inv_cov_flow_ode;
        vaso_proxy_mean(:,j) = (vaso_proxy_inv_cov_vaso_ode +
     vaso_proxy_inv_cov_flow_ode) \ ...
```

```
        (local_mean_vaso_ode + local_mean_flow_ode);

end

end


function [neuronal_proxy_mean,neuronal_proxy_inv_cov] =
 proxy_for_neuronal_populations(neuronal,state,...
    ode_param,dC_times_invC,coupling_idx,symbols,A_plus_gamma_inv)

state_idx = find(cellfun(@(x) strcmp(x(2),'n'),symbols.state));
j = 0;
for u = state_idx

    j = j+1;
    local_mean_sum = zeros(size(dC_times_invC,1),1);
    neuronal_proxy_inv_cov(:,:,j) = zeros(size(dC_times_invC(:,:,1)));

    for k = coupling_idx{u}'

        % unpack matrices B and b
        B = diag(neuronal.B{u,k}(state,ode_param));
        b = neuronal.b{u,k}(state,ode_param);

        if k~=u
            if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end

            local_mean_sum = local_mean_sum + B' * A_plus_gamma_inv *
 (dC_times_invC * state(:,k) - b);
            neuronal_proxy_inv_cov(:,:,j) =
 neuronal_proxy_inv_cov(:,:,j) + B' * A_plus_gamma_inv * B;
            % The next two code lines (instead of the top two) works
 also:
            % local_mean_sum = local_mean_sum + B' * (dC_times_invC *
 state(:,k) - b);
            % neuronal_proxy_inv_cov(:,:,j) =
 neuronal_proxy_inv_cov(:,:,j) + B' * B;
        else
            if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end
            B = B - dC_times_invC;

            local_mean_sum = local_mean_sum - B' * A_plus_gamma_inv *
 b;
            neuronal_proxy_inv_cov(:,:,j) =
 neuronal_proxy_inv_cov(:,:,j) + B' * A_plus_gamma_inv * B;
            % The next two code lines (instead of the top two) works
 also:
            % local_mean_sum = local_mean_sum - B' * b;
            % neuronal_proxy_inv_cov(:,:,j) =
 neuronal_proxy_inv_cov(:,:,j) + B' * B;
        end
    end
```

```
    neuronal_proxy_mean(:,j) = neuronal_proxy_inv_cov(:,:,j) \
 local_mean_sum;
end

end
```

The prior variance on all non-selfinhibitory neuronal couplings is infinity.

```
function ode_param =
 prior_on_ODE_param(ode_param,param_prior,param_symbols)

numb_states = 3;
ode_param.prior.mean = zeros(length(param_symbols),1);
ode_param.prior.mean(end-numb_states+1:end) = -1;
tmp = param_prior*ones(1,length(param_symbols));
tmp(end-numb_states+1:end) = 1e-9;
ode_param.prior.inv_cov = diag(tmp.^(-1));

end


function bold_response = confounding_effects(bold_response)

bold_response.confounding_effects.X0 = importdata('dcm/
confounding_effects_X0.txt');
bold_response.confounding_effects.beta = importdata('dcm/
confounding_effects_beta.txt');

bold_response.confounding_effects.X0_penrose_inv =
 (bold_response.confounding_effects.X0' * ...
    bold_response.confounding_effects.X0)^(-1) *
 bold_response.confounding_effects.X0';

bold_response.confounding_effects.intercept =
 ones(size(bold_response.obs));

end


function ode = import_odes(symbols,candidate_odes)

path_ode = ['./dcm/ODEs/' candidate_odes '.txt'];
     % path to candidtae system of ODEs

ode.raw = importdata(path_ode);
ode.refined = ode.raw;

for k = 1:length(ode.refined)
for u = 1:length(symbols.state); ode.refined{k} =
 strrep(ode.refined{k},[symbols.state{u}],['state(:,'
 num2str(u) ')']); end
for j = 1:length(symbols.param); ode.refined{k} =
 strrep(ode.refined{k},symbols.param{j},['param('
 num2str(j) ')']); end
```

```
end
for k = 1:length(ode.refined); ode.system{k} =
 str2func(['@(state,param)(' ode.refined{k} ')']); end

end


function [state,time,ode,bold_response] =
 simulate_dynamics_by_numerical_integration(state,time,ode,simulation,symbols)

param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');
state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');
for i = 1:length(ode.system)
    ode.system_sym(i) = ode.system{i}(state_sym,param_sym);
end

idx0 = cellfun(@(n) ~strcmp(n(2),'u'),symbols.state);
learn_method.state(idx0) = {'Laplace mean-field'};
learn_method.state(~idx0) = {'external input'};

state.obs_idx = zeros(1,sum(idx0));
state.init_val = zeros(1,sum(idx0));
%
init_val = 0.01*ones(1,sum(idx0));

%
dt = state.ext_input(end,1) - state.ext_input(end-1,1);
ode_system_mat = matlabFunction(ode.system_sym','Vars',
{state_sym(~strcmp(learn_method.state,'external input'))',...
        param_sym',state_sym(strcmp(learn_method.state,'external
 input'))'});

ode_param_true = simulation.ode_param';

% warning ('off','all');
[ToutX,OutX_solver] = ode113(@(t,n)
 ode_function(t,n,ode_system_mat,ode_param_true,state.ext_input(:,2:end),state.ext
    state.ext_input(:,1), init_val);
% warning ('on','all');

[~,idx] = min(pdist2(ToutX,state.ext_input(:,1)),[],1);
ToutX = ToutX(idx); OutX_solver = OutX_solver(idx,:);

% pack
[~,state.ext_input_to_bold_response_mapping_idx] =
 min(pdist2(state.ext_input(:,1),time.est'),[],1);
state.true =
 OutX_solver(state.ext_input_to_bold_response_mapping_idx,:);
state.true(1:5,:) = 0;

time.true = ToutX';
time.samp = time.true(state.ext_input_to_bold_response_mapping_idx);
```

```
% true bold responses
bold_response.true = bold_signal_change_eqn(state.true(:,cellfun(@(n)
 strcmp(n(2),'v'),symbols.state)),state.true(:,cellfun(@(n)
 strcmp(n(2),'q'),symbols.state)));
% mean correction
% bold_response.confounding_effects.intercept =
 mean(bold_response.true,1);
% bold_response.true =
 bsxfun(@minus,bold_response.true,mean(bold_response.true,1));
% % bold_response.confounding_effects.X0 =
 ones(size(bold_response.true));

% observed bold responses
bold_response.obs = bold_response.true +
 bsxfun(@times,sqrt(var(bold_response.true) ./
 simulation.SNR),randn(size(bold_response.true)));
bold_response.confounding_effects.intercept =
 mean(bold_response.obs,1);
bold_response.variance = (repmat(max(bold_response.obs,
[],1),size(bold_response.obs,1),1)./simulation.SNR).^2;

% pack
state.obs = state.true(:,find(state.obs_idx));

% align externel input with observations
shift_num = 1;
e = state.ext_input;
e(shift_num+1:end,2:end) = state.ext_input(1:end-shift_num,2:end);
e(1:shift_num,2:end) = zeros(shift_num,size(state.ext_input,2)-1);
state.ext_input = e;

end


function [state,bold_response] =
 simulate_trajectory_with_vgm_param_est(ode_param,state,state_orig,bold_response,s

bold_response.prediction.num_int_with_gm_param_est = [];

state_orig.init_val = state.proxy.mean(1,cellfun(@(x)
 ~strcmp(x(2),'u'),symbols.state));

simulation.ode_param = ode_param.proxy.mean';

state_sim =
 simulate_dynamics_by_numerical_integration(state_orig,time,ode,simulation,symbols

state.num_int_with_gm_param_est = state_sim.true;

%
bold_response_signal_change =
 bold_signal_change_eqn(state.num_int_with_gm_param_est(:,cellfun(@(n)
 strcmp(n(2),'v'),symbols.state)),...
```

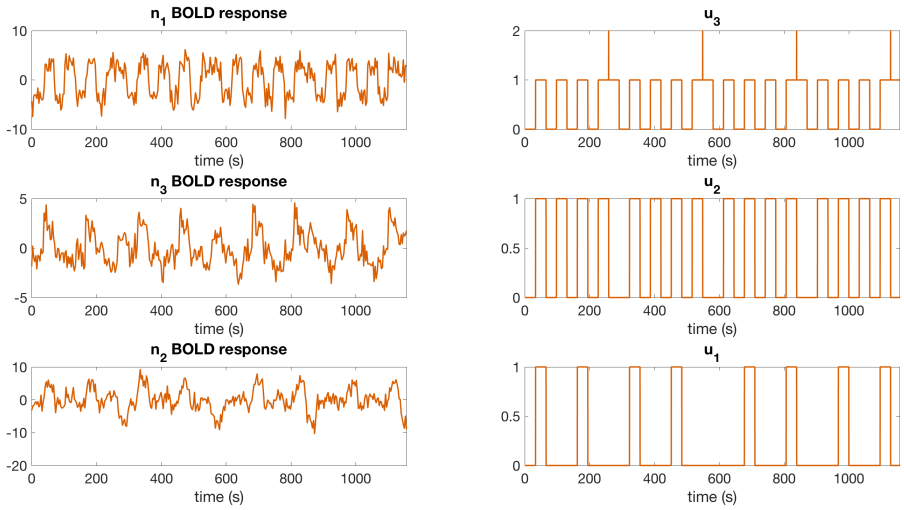
```
    state.num_int_with_gm_param_est(:,cellfun(@(n)
 strcmp(n(2),'q'),symbols.state)));
bold_response.confounding_effects.intercept
 = determine_intercept(bold_response.obs_old-
bold_response_signal_change,...

 bold_response.confounding_effects.X0,bold_response.confounding_effects.X0_penrose
bold_response.prediction.num_int_with_gm_param_est =
 bold_response_signal_change +
 bold_response.confounding_effects.intercept;

%
state.num_int_with_gm_param_est(1,:) = [];
state.num_int_with_gm_param_est(end+1,:) =
 zeros(1,size(state.num_int_with_gm_param_est,2));
bold_response.prediction.num_int_with_gm_param_est(1,:) = [];
bold_response.prediction.num_int_with_gm_param_est(end+1,:) =
 zeros(1,size(bold_response.prediction.num_int_with_gm_param_est,2));

end


function [state,time,obs_to_state_relation] =
 generate_state_observations(state,time,simulation,symbols)

% State observations
tmp = cellfun(@(x)
 {strcmp(x(2),simulation.observed_states)},symbols.state);
state.obs_idx = cellfun(@(x) any(x),tmp);
state.obs_idx(cellfun(@(x) strcmp(x(2),'u'),symbols.state)) = [];
state.obs = state.true(:,state.obs_idx) +
 sqrt(var(state.true(:,state.obs_idx)) ./ simulation.SNR) .*
 randn(size(state.true(:,state.obs_idx)));

% Relationship between states and observations
if length(simulation.time_samp) < length(time.est)
    time.idx = munkres(pdist2(time.samp',time.est'));
    time.ind =
 sub2ind([length(time.samp),length(time.est)],1:length(time.samp),time.idx);
else
    time.idx = munkres(pdist2(time.est',time.samp'));
    time.ind =
 sub2ind([length(time.est),length(time.samp)],1:length(time.est),time.idx);
end

time.obs_time_to_state_time_relation =
 zeros(length(time.samp),length(time.est));
 time.obs_time_to_state_time_relation(time.ind) = 1;
state_mat = eye(size(state.true,2));
 state_mat(~logical(state.obs_idx),:) = [];
obs_to_state_relation =
 sparse(kron(state_mat,time.obs_time_to_state_time_relation));

end
```

```
function state_derivatives =
 ode_function(time,states,ode_system_mat,ode_param,ext_input,time_lst)

[~,idx] = min(pdist2(time,time_lst));
u = ext_input(idx,:);

state_derivatives = ode_system_mat(states,ode_param,u');

end
```

```
Warning: Failure at t=3.546137e+01.  Unable to meet integration
 tolerances
without reducing the step size below the smallest value allowed
 (1.136868e-13)
at time t.
Warning: Failure at t=3.710065e+01.  Unable to meet integration
 tolerances
without reducing the step size below the smallest value allowed
 (1.136868e-13)
at time t.
Warning: Failure at t=3.658686e+01.  Unable to meet integration
 tolerances
without reducing the step size below the smallest value allowed
 (1.136868e-13)
at time t.
Warning: Failure at t=3.607762e+01.  Unable to meet integration
 tolerances
without reducing the step size below the smallest value allowed
 (1.136868e-13)
at time t.
```

```
function intercept =
 determine_intercept(bold_response_diff,X0,X0_penrose_inv)

intercept = X0 * X0_penrose_inv * bold_response_diff;

end
```

```
function [ode,state_sym,param_sym] =
 write_ODEs_as_symbolic_expression(symbols,ode)

param_sym = sym('param%d',[1,length(symbols.param)]);
 assume(param_sym,'real');
state_sym = sym('state%d',[1,length(symbols.state)]);
 assume(state_sym,'real');
for k = 1:length(ode.system)
    ode.system_sym(k) = ode.system{k}(state_sym,param_sym);
end

end
```




```
function [h_states,h_param,p] =
 setup_plots_for_states(state,time,symbols)

for i = 1:length(symbols.param); symbols.param{i} = symbols.param{i}
(2:end-1); end

figure(1); set(1, 'Position', [0, 200, 1600, 800]);

h_param = subplot(3,2,1); h_param.FontSize = 20; h_param.Title.String
 = 'ODE parameters';
set(gca,'XTick',[1:length(symbols.param)]);
 set(gca,'XTickLabel',symbols.param);
hold on;

i = 0;
for u = [3,6,9,12,15]
    i = i+1;
    h_states{u} = subplot(3,2,i+1); cla; p.true =
 plot(time.samp,state.true(:,u),'LineWidth',2,'Color',
[217,95,2]./255);
    try; hold on; p.obs = plot(time.samp,state.obs(:,u),'*','Color',
[217,95,2]./255,'MarkerSize',1);end
    h_states{u}.FontSize = 20; h_states{u}.Title.String
 = symbols.state{u}(2:end-1); h_states{u}.XLim =
 [min(time.est),max(time.est)];
    h_states{u}.XLabel.String = 'time (s)'; hold on;
end
drawnow

end


function [h_bold,h_ext_input] =
 setup_plots_for_bold_response_and_ext_input(state,bold_response,time,symbols)
```

```
for i = 1:length(symbols.param); symbols.param{i} = symbols.param{i}
(2:end-1); end

figure(2); set(2, 'Position', [0, 200, 1600, 800]);

plot_titles_idx = find(cellfun(@(x) strcmp(x(2),'n'),symbols.state));
plot_idx = [1:2:3*2];
for u = 1:3
    h_bold{u} = subplot(3,2,plot_idx(u)); cla;

 plot(h_bold{u},time.samp,bold_response.obs(:,u),'LineWidth',2,'Color',
[217,95,2]./255);
    h_bold{u}.FontSize = 20; h_bold{u}.Title.String =
 [symbols.state{plot_titles_idx(u)}(2:end-1) ' BOLD response'];
    h_bold{u}.XLim = [min(time.est),max(time.est)];
    h_bold{u}.XLabel.String = 'time (s)'; hold on;
end

plot_titles_idx = flipdim(find(cellfun(@(x)
 strcmp(x(2),'u'),symbols.state)),2);
plot_idx = [2:2:3*2];
for i = 1:sum(cellfun(@(x) strcmp(x(2),'u'),symbols.state))
    h_ext_input{i} = subplot(3,2,plot_idx(i));
    plot(h_ext_input{i},time.true,state.ext_input(:,i
+1),'LineWidth',2,'Color',[217,95,2]./255); hold on;
    h_ext_input{i}.FontSize = 20; h_ext_input{i}.Title.String =
 symbols.state{plot_titles_idx(i)}(2:end-1);;
    h_ext_input{i}.XLim = [min(time.est),max(time.est)];
    h_ext_input{i}.XLabel.String = 'time (s)'; hold on;
end
drawnow

end


function
 plot_results_for_states(h_states,h_param,state,time,simulation,param_proxy_mean,s

for u = [3,6,9,12,15]
    hold on; p.vgm =
 plot(h_states{u},time.samp,state.proxy.mean(:,u),'LineWidth',0.1,'Color',
[0.4,0.4,0.4]);
    try; p.num_int =
 plot(h_states{u},time.samp(1,:),state.num_int_with_gm_param_est(:,u),'Color',
[0,0,0],'LineWidth',1); end

    if any(cellfun(@(x) ~strcmp(x,symbols.state{u}
(2)),simulation.observed_states))
        legend(h_states{u},
{'true','observed','estimated'},'Location','northwest','FontSize',10);
    else
        try
```

```
            legend(h_states{u},[p.true,p.vgm,p.num_int],
{'true','estimated','numerical int. with est.
 param.'},'Location','southwest','FontSize',10);
        catch
            legend(h_states{u},[p.true,p.vgm],
{'true','estimated'},'Location','southwest','FontSize',10);
        end
    end
end

cla(h_param);
if strcmp(simulation.odes,candidate_odes)
    b = bar(h_param,1:length(param_proxy_mean),
[simulation.ode_param',param_proxy_mean]);
    b(1).FaceColor = [217,95,2]./255; b(2).FaceColor =
 [117,112,179]./255;
    legend(h_param,
{'true','estimated'},'Location','northeast','FontSize',12);
else
    b = bar(h_param,1:length(param_proxy_mean),param_proxy_mean);
    b.FaceColor = [117,112,179]./255;
    legend(h_param,
{'estimated'},'Location','northeast','FontSize',12);
end
h_param.XLim = [0.5,length(param_proxy_mean)+0.5]; h_param.YLimMode
 = 'auto';
drawnow

end


function plot_results_for_bold_response(h_bold,bold_response,time)

for u = 1:3

 plot(h_bold{u},time.est,bold_response.prediction.num_int_with_gm_param_est(:,u),'
[0,0,0]); hold on;
    legend(h_bold{u},{'observed BOLD response','numerical int. with
 est. param.'},'Location','southwest','FontSize',10);
end
drawnow

end
```

Published with MATLAB® R2017a