Variational Gradient Matching for Dynamical Systems: Lotka-Volterra

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.state_obs_variance = @(mean)(bsxfun(@times,
[0.5^2,0.5^2],...
    ones(size(mean))));
     % observation noise
simulation.ode_param = [2,1,4,1];
      % true ODE parameters [2 1 4 1] is used as a benchmark in many
 publications;
simulation.final_time = 2;
     % 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.init_val = [5 3];
     % state values at first time point
simulation.state_obs_idx = [1,1];
     % indices of states that are directly observed (Boolean)
```

User Input

```
kernel.param = [10,0.2];
     % set values of rbf kernel parameters
state.derivative_variance = [6,6];
     % gamma for gradient matching model


time.est = 0:0.1:4;
     % estimation times
coord_ascent_numb_iter = 200;
     % number of coordinate ascent iterations
clamp_obs_state_to_GP_regression = false;
     % The observed state trajectories are clamped to the trajectories
 determined by standard GP regression (Boolean)


symbols.state = {'[prey]','[predator]'};
     % symbols of states in 'ODEs.txt' file
symbols.param = {'[\theta_1]','[\theta_2]','[\theta_3]','[\theta_4]'};
     % symbols of parameters in 'ODEs.txt' file
```

Import ODEs

```
ode = import_odes(symbols);
```

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

```
ODEs:
    '[\theta_1].*[prey] - [\theta_2].*[prey].*[predator]'
    '-[\theta_3].*[predator] + [\theta_4].*[prey].*[predator]'
```

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 N 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 $\mathbf{x}_u$. 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 Data

```
[state,time,ode] =
 generate_ground_truth(time,state,ode,symbols,simulation);
```

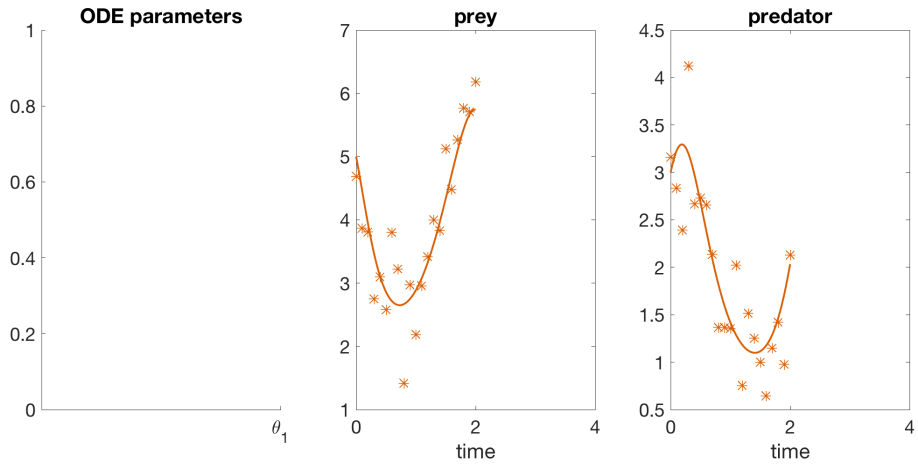
```
[state,time,obs_to_state_relation] =
 generate_state_obs(state,time,simulation);
```

```
state.sym.mean = sym('x%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');
```

Only the state dynamics are (partially) observed.

```
[h,h2] = setup_plots(state,time,simulation,symbols);
```

```
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:

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

,

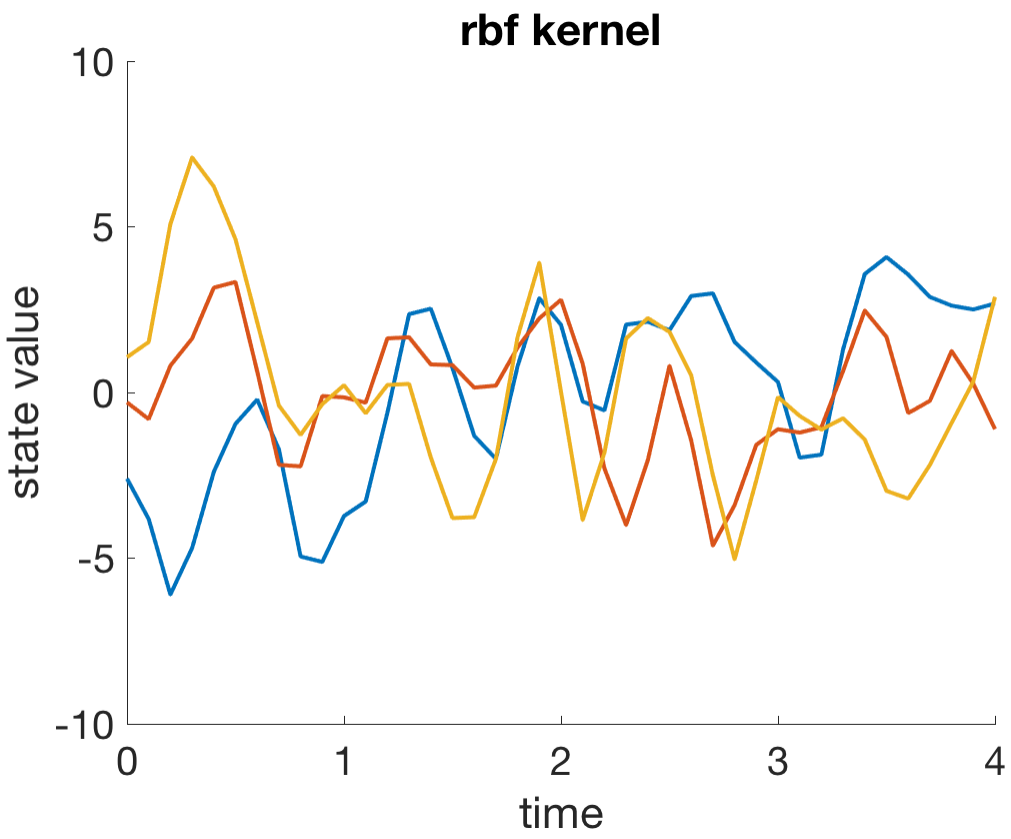
$$\mathrm{cov}(x_k(t), x_k(t)) = C_{\phi_k}(t,t')$$

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

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

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

```
[Lambda,dC_times_invC,inv_Cxx,time.est] =
 kernel_function(kernel,state,time.est);
```



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:

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

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

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

State Couplings in ODEs

```
coupling_idx = find_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.B,ode_param.b,ode_param.r,ode_param.B_times_Lambda_times_B]
 = rewrite_odes_as_linear_combination_in_parameters(ode,symbols);
```

Posterior over ODE Parameters

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

$$\boldsymbol{\theta} = \mathbf{B}_\theta^+ \left({}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X} - \mathbf{b}_\theta + \epsilon_0 \right),$$

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

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

Rewrite ODEs as Linear Combination in Individual States

We rewrite the expression $\mathbf{f}(\mathbf{X}, \theta) - {}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X}$ in equation (4) as a linear combination in the individual state $\mathbf{x}_u$:

$$\mathbf{B}_u \mathbf{x}_u + \mathbf{b}_u \stackrel{!}{=} \mathbf{f}(\mathbf{X}, \theta) - {}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X} \qquad (7).$$

where matrices $\mathbf{B}_u$ and $\mathbf{b}_u$ are defined such that the expression $\mathbf{f}(\mathbf{X}, \theta) - {}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X}$ is rewritten as a linear combination in the individual state $\mathbf{x}_u$.

```
state =
 rewrite_odes_as_linear_combination_in_ind_states(state,ode,symbols,coupling_idx.s
```

Posterior over Individual States

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

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

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

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 $p(\mathbf{x}_u \mid \theta, \mathbf{X}_{-u}, \mathbf{Y}, \phi, \gamma, \sigma)$ (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:

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

The particular form of $q(\theta)$ and $q(\mathbf{x}_u)$ 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:

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

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:

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

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

GP Regression for Observations

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 $\boldsymbol{p(\mathbf{X} \mid \phi) = \prod_k \mathcal{N}(\mathbf{x}_k; \mathbf{0}, \mathbf{C}_\phi)}$:

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

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 $\mathbf{\Sigma}_k^{-1} := \sigma_k^{-2}\mathbf{I} + \mathbf{C}_{\phi_k}^{-1}$.

```
[mu,inv_sigma] =
 GP_regression(state,inv_Cxx,obs_to_state_relation,simulation);
```

Coordinate Ascent Variational Gradient Matching

We 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 $\hat{q}(\mathbf{x}_u)$.

```
state.proxy.mean = mu;
     % Initialize the state estimation by the GP regression posterior
for i = 1:coord_ascent_numb_iter
```

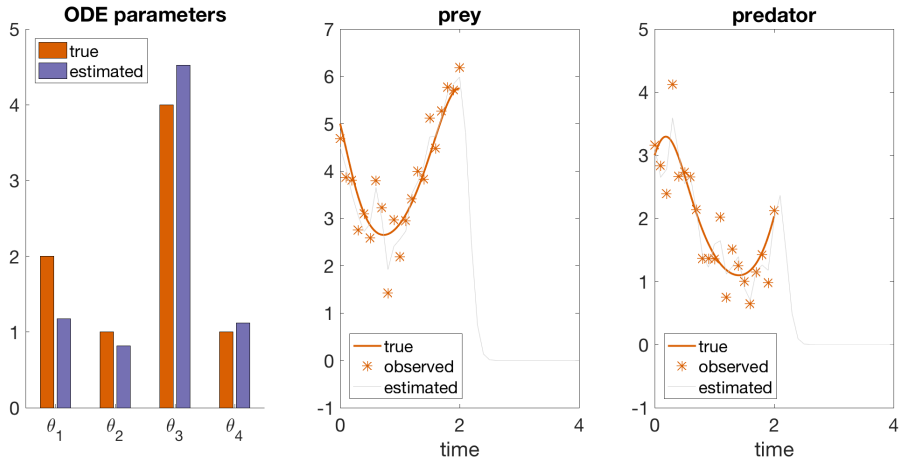
Expanding the proxy distribution in equation (11) for θ yields:

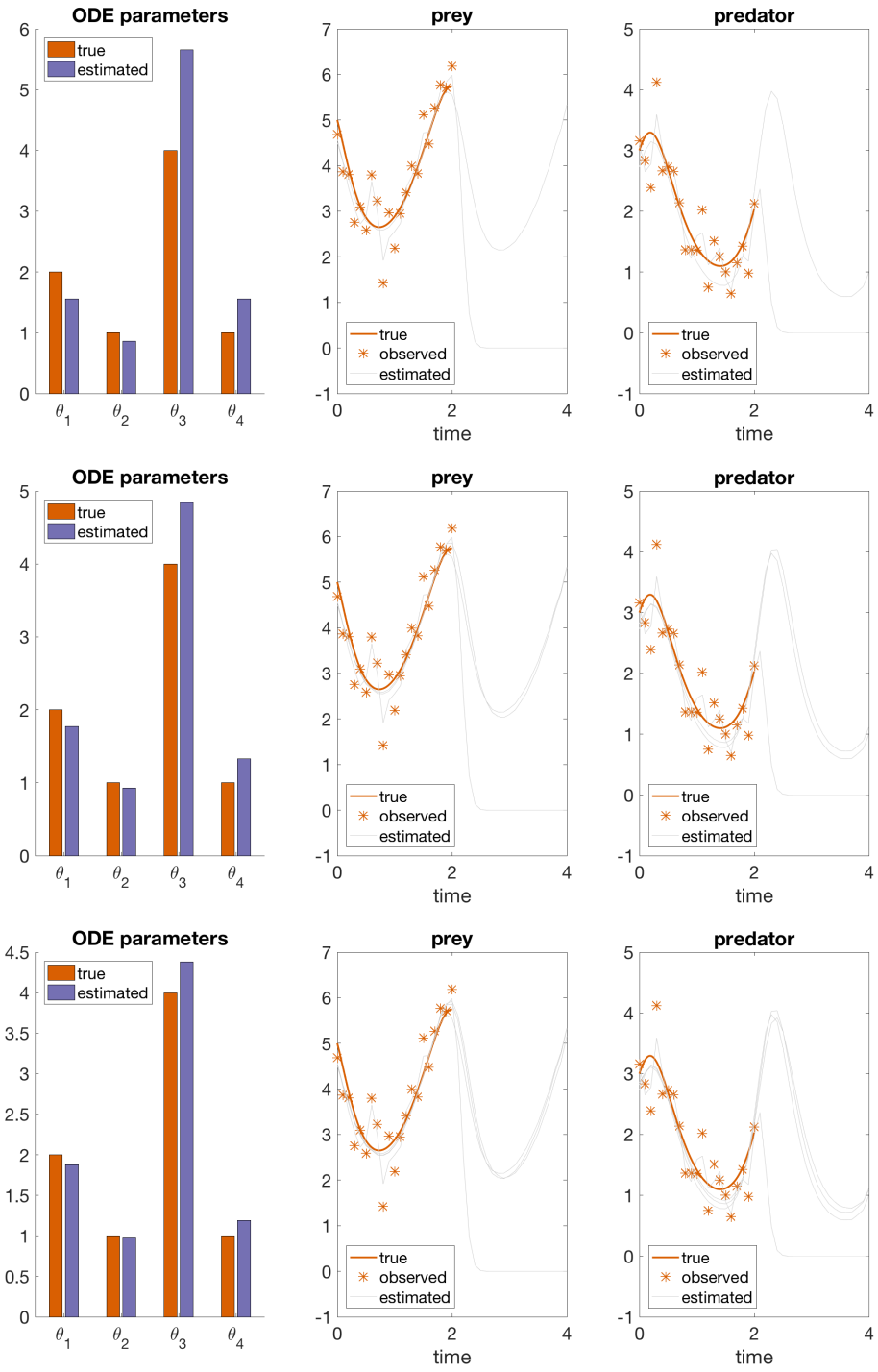
$$\hat{q}(\theta) \overset{(a)}{\propto} \exp\left(E_{Q_{-\theta}} \ln p(\theta \mid \mathbf{X}, \mathbf{Y}, \phi, \gamma, \sigma) \right)$$

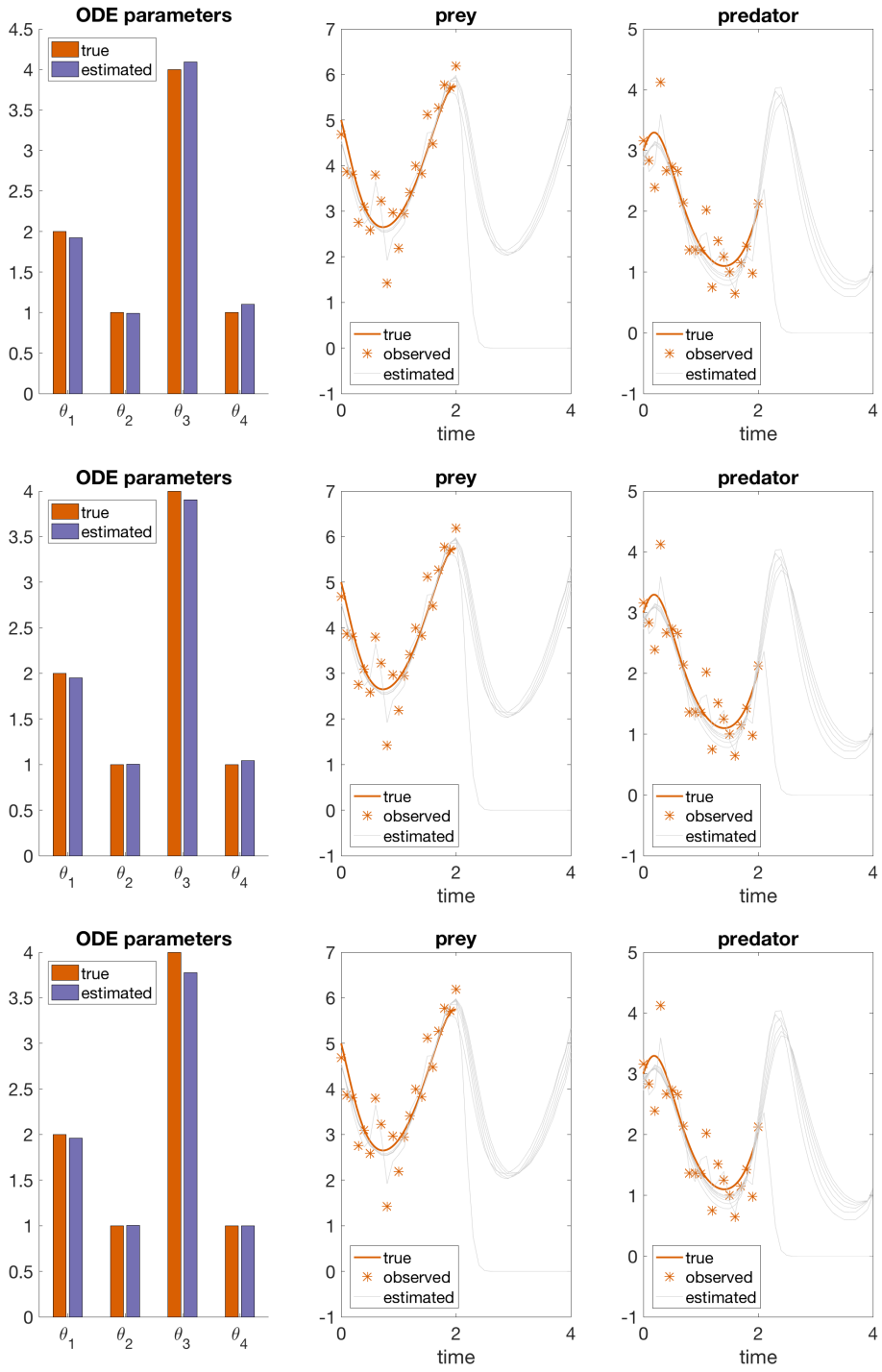
$$\overset{(b)}{\propto} \exp\left(E_{Q_{-\theta}} \ln \mathcal{N}\left(\theta; \mathbf{B}_\theta^+ \left('\mathbf{C}_\phi \mathbf{C}_\phi^{-1}\mathbf{X} - \mathbf{b}_\theta\right), \mathbf{B}_\theta^+ (\mathbf{A} + \mathbf{I}\gamma) \mathbf{B}_\theta^{+T}\right) \right),$$

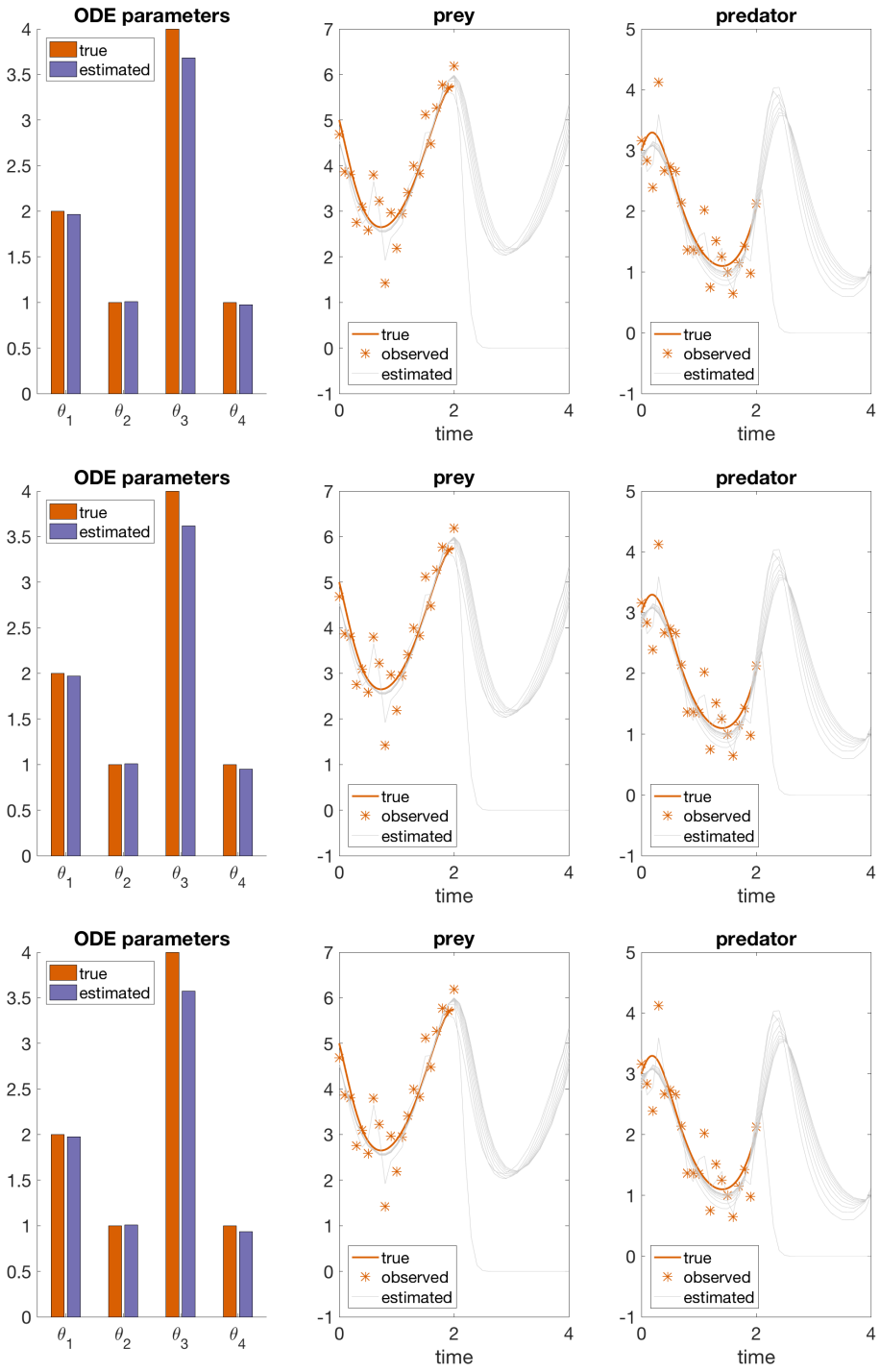
which can be normalized analytically due to its exponential quadratic form. In (a) we recall that the ODE parameters depend only indirectly on the observations $\mathbf{Y}$ through the states $\mathbf{X}$ and in (b) we substitute $p(\theta \mid \mathbf{X}, \phi, \gamma)$ by its density given in equation (6).

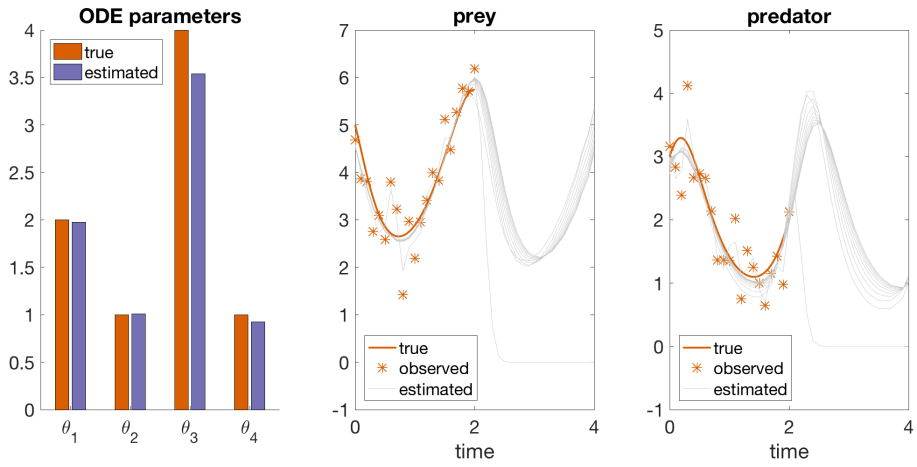
```
   [param_proxy_mean,param_proxy_inv_cov] =
proxy_for_ode_parameters(state.proxy.mean,Lambda,dC_times_invC,ode_param,symbols)
   if i==1 || ~mod(i,20);
plot_results(h,h2,state,time,simulation,param_proxy_mean,'not_final'); end
```




ODE parameters
true
estimated
θ_1 θ_2 θ_3 θ_4
prey
predator
true
observed
estimated
time


ODE parameters
true
estimated
θ_1 θ_2 θ_3 θ_4
prey
predator
true
observed
estimated
time


ODE parameters
true
estimated
θ_1 θ_2 θ_3 θ_4
prey
predator
true
observed
estimated
time



Expanding the proxy distribution in equation (12) over the individual state $\mathbf{x}_u$:

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

$$\overset{(b)}{\propto} \exp\left(E_{Q_{-u}} \ln \mathcal{N}\left(\mathbf{x}_u; -\mathbf{B}_u^{+}\mathbf{b}_u,\ \mathbf{B}_u^{+}\ (\mathbf{A} + \mathbf{I}\gamma)\ \mathbf{B}_u^{+T}\right) + E_{Q_{-u}} \ln \mathcal{N}\left(\mathbf{x}_u; \mu_u(\mathbf{Y}), \Sigma_u\right) \right),$$

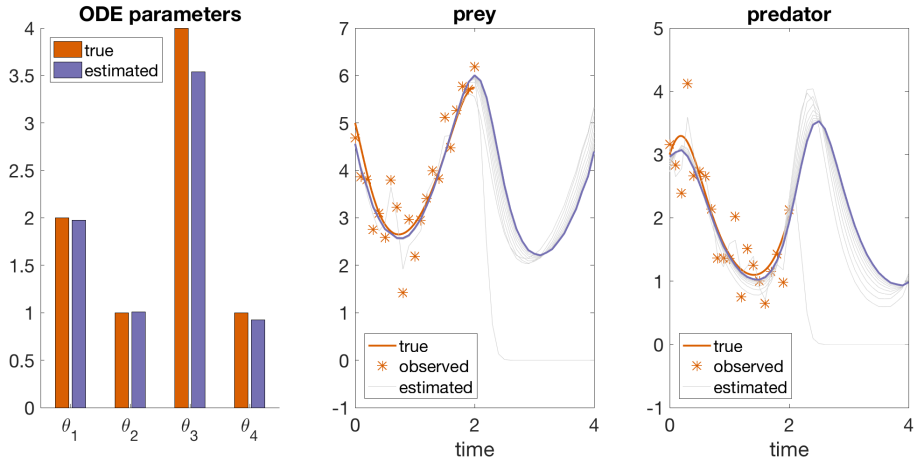
which, once more, can be normalized analytically due to its exponential quadratic form. In (a) we decompose the full conditional into an ODE-informed distribution and a data-informed distribution and in (b) we substitute the ODE-informed distribution $p(\mathbf{x}_u \mid \theta, \mathbf{X}_{-u}, \phi, \gamma)$ with its density given by equation (8).

```
    state.proxy.mean =
proxy_for_ind_states(state.lin_comb,state.proxy.mean,param_proxy_mean',...

 dC_times_invC,coupling_idx.states,symbols,mu,inv_sigma,state.obs_idx,clamp_obs_st

end


plot_results(h,h2,state,time,simulation,param_proxy_mean,'final');
```



Time Taken

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

```
time taken: 21.5814 seconds
```

References

- **Gorbach, N.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, N.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, N.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. N.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:

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

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

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

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

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

```
function [Lambda,dC_times_invC,inv_Cxx,time_est] =
 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_Cxx = inv_chol(chol(C,'lower'));

dC_times_invC = dC * inv_Cxx;

% 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';
 h1.YLabel.String = 'state value';
h1.Title.String = [kernel.name ' kernel'];

% determine \Lambda:
A = ddC - dC_times_invC * Cd;
inv_Lambda = A + state.derivative_variance(1) .* eye(size(A));
inv_Lambda = 0.5.*(inv_Lambda+inv_Lambda');
Lambda = inv_chol(chol(inv_Lambda,'lower'));

end
```

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

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 $\boldsymbol{\Sigma}_k^{-1} := \sigma_k^{-2}\mathbf{I} + \mathbf{C}_{\phi_k}^{-1}$.

```
function [mu_u,inv_sigma_u,state] =
 GP_regression(state,inv_Cxx,obs_to_state_relation,simulation)

state_obs_variance = simulation.state_obs_variance(state.obs);

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

inv_Cxx_tmp = num2cell(inv_Cxx(:,:,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

end


function coupling_idx = find_couplings_in_odes(ode,symbols)

% state couplings
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 = 1:length(symbols.state)
    coupling_idx_tmp = find(ode_couplings_states(:,u));
    coupling_idx.states{u} = coupling_idx_tmp;
```

```
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,r,B_times_Lambda_times_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);

% 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

B_times_Lambda_times_B = @(B,Lambda)(B' * B);
r = @(B,Lambda,dC_times_invC,state,b)(B' * (dC_times_invC * state +
 b));

end
```

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

where matrices $\mathbf{B}_u$ and $\mathbf{b}_u$ are defined such that the expression $\mathbf{f}(\mathbf{X}, \theta) - {}'\mathbf{C}_\phi \mathbf{C}_\phi^{-1} \mathbf{X}$ is rewritten as a linear combination in the individual state $\mathbf{x}_u$.

```
function state =
 rewrite_odes_as_linear_combination_in_ind_states(state,ode,symbols,coupling_idx)
```

```
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');

for u = 1:length(symbols.state)
    for k = coupling_idx{u}'
        [B,b] = equationsToMatrix(ode.system{k}
(state_sym,param_sym'),state_sym(:,u));
        state.lin_comb{u,k}.B = matlabFunction(B,'Vars',
{state_sym,param_sym});
        state.lin_comb{u,k}.b = matlabFunction(b,'Vars',
{state_sym,param_sym});
    end
end

end
```

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

```
function [param_proxy_mean,param_inv_cov] =
 proxy_for_ode_parameters(state_proxy_mean,Lambda,dC_times_invC,ode_param,symbols)

B_global = []; b_global = [];
state0 = zeros(size(dC_times_invC,1),1);
param_inv_cov = zeros(length(symbols.param));
local_mean_sum = zeros(length(symbols.param),1);
for k = 1:length(symbols.state)
    B = ode_param.B{k}
(state_proxy_mean,state0,ones(size(state_proxy_mean,1),1));
    local_inv_cov = ode_param.B_times_Lambda_times_B(B,Lambda);
    b = ode_param.b{k}
(state_proxy_mean,state0,ones(size(state_proxy_mean,1),1));
    local_mean =
 ode_param.r(B,Lambda,dC_times_invC,state_proxy_mean(:,k),b);
    param_inv_cov = param_inv_cov + local_inv_cov;
    local_mean_sum = local_mean_sum + local_mean;

    B_global = [B_global;B];
    b_tmp = b; if length(b_tmp)==1;
 b_tmp=zeros(size(dC_times_invC,1),1);end
    b_global = [b_global;b_tmp];
end

[~,D] = eig(param_inv_cov);
if any(diag(D)<0)
    warning('param_inv_cov has negative eigenvalues!');
elseif any(diag(D)<1e-3)
    warning('param_inv_cov is badly scaled')
    disp('perturbing diagonal of param_inv_cov')
    perturb = abs(max(diag(D))-min(diag(D))) / 10000;
```

```
    param_inv_cov(logical(eye(size(param_inv_cov,1)))) =
 param_inv_cov(logical(eye(size(param_inv_cov,1)))) ...
        + perturb.*rand(size(param_inv_cov,1),1);
end
param_proxy_mean = pinv(param_inv_cov) * local_mean_sum;

end
```

$$\hat{q}(\mathbf{x}_u) \propto \exp\left(E_{Q_{-u}} \ln \mathcal{N}\left(\mathbf{x}_u; -\mathbf{B}_u^{+}\mathbf{b}_u,\ \mathbf{B}_u^{+}\left(\mathbf{A}+\mathbf{I}\gamma\right)\mathbf{B}_u^{+T}\right) + E_{Q_{-u}} \ln \mathcal{N}\left(\mathbf{x}_u; \mu_u(\mathbf{Y}), \Sigma_u\right)\right)$$

```
function [state_mean,state_inv_cov] =
 proxy_for_ind_states(lin_comb,state_mean,...

 ode_param,dC_times_invC,coupling_idx,symbols,mu,inv_sigma,state_obs_idx,...
    clamp_obs_state_to_GP_regression)

if clamp_obs_state_to_GP_regression
    state_enumeration = find(~state_obs_idx);
else
    state_enumeration = 1:length(symbols.state);
end

for u = state_enumeration

    state_inv_cov(:,:,u) = zeros(size(dC_times_invC));
    local_mean_sum = zeros(size(dC_times_invC,1),1);
    for k = coupling_idx{u}'
        if k~=u
            B = diag(lin_comb{u,k}.B(state_mean,ode_param));
            if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end

            state_inv_cov(:,:,u) = state_inv_cov(:,:,u) + B' * B;
            local_mean_sum = local_mean_sum + B' * (dC_times_invC *
 state_mean(:,k) ...
                + lin_comb{u,k}.b(state_mean,ode_param));
        else
            B = diag(lin_comb{u,k}.B(state_mean,ode_param));
            if size(B,1) == 1; B = B.*eye(size(dC_times_invC,1)); end
            B = B - dC_times_invC;

            state_inv_cov(:,:,u) = state_inv_cov(:,:,u) + B' * B;

            l = lin_comb{u,k}.b(state_mean,ode_param); if
 length(l)==1; l = zeros(length(local_mean_sum),1); end
            local_mean_sum = local_mean_sum + B' * l;
        end
    end

    state_mean(:,u) = (state_inv_cov(:,:,u) + inv_sigma(:,:,u)) \
 (local_mean_sum + (inv_sigma(:,:,u) * mu(:,u)));
end

end
```

```
function ode = import_odes(symbols)

path_ode = './Lotka_Volterra_ODEs.txt';
                    % path to 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] =
 generate_ground_truth(time,state,ode,symbols,simulation)

time.true=0:simulation.int_interval:simulation.final_time;
     % true times

Tindex=length(time.true);
     % index time
TTT=length(simulation.time_samp);
     % number of sampled points
itrue=round(simulation.time_samp./simulation.int_interval
+ones(1,TTT));    % Index of sample time in the true time

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

ode_system_mat = matlabFunction(ode.system_sym','Vars',
{state_sym',param_sym'});
[~,OutX_solver]=ode45(@(t,x) ode_system_mat(x,simulation.ode_param'),
 time.true, simulation.init_val);
state.true_all=OutX_solver;
state.true=state.true_all(itrue,:);

state.obs_idx = simulation.state_obs_idx;

end
```

```
function [state,time,obs_to_state_relation] =
 generate_state_obs(state,time,simulation)

% State observations
state_obs_variance = simulation.state_obs_variance(state.true);
state.obs = state.true + sqrt(state_obs_variance) .*
 randn(size(state.true));

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

time.obs_time_to_state_time_relation =
 zeros(length(simulation.time_samp),length(time.est));
 time.obs_time_to_state_time_relation(time.ind) = 1;
state_mat = eye(size(state.true,2));
obs_to_state_relation =
 sparse(kron(state_mat,time.obs_time_to_state_time_relation));
time.samp = simulation.time_samp;

end


function [h,h2] = setup_plots(state,time,simulation,symbols)

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

figure(1); set(1, 'Position', [0, 200, 1200, 500]);

h2 = subplot(1,3,1); h2.FontSize = 20; h2.Title.String = 'ODE
 parameters';
set(gca,'XTick',[1:length(symbols.param)]);
 set(gca,'XTickLabel',symbols.param);
hold on; drawnow

for u = 1:2
    h{u} = subplot(1,3,u+1); cla;
 plot(time.true,state.true_all(:,u),'LineWidth',2,'Color',
[217,95,2]./255);
    hold on; plot(simulation.time_samp,state.obs(:,u),'*','Color',
[217,95,2]./255,'MarkerSize',10);
    h{u}.FontSize = 20; h{u}.Title.String = symbols.state{u}(2:end-1);
 h{u}.XLim = [min(time.est),max(time.est)];
    h{u}.XLabel.String = 'time'; hold on;
end
```

```
end


function
 plot_results(h,h2,state,time,simulation,param_proxy_mean,plot_type)

for u = 1:2
    if strcmp(plot_type,'final')
        hold on; plot(h{u},time.est,state.proxy.mean(:,u),'Color',
[117,112,179]./255,'LineWidth',2);
    else
        hold on;
 plot(h{u},time.est,state.proxy.mean(:,u),'LineWidth',0.1,'Color',
[0.8,0.8,0.8]);
    end
    legend(h{u},
{'true','observed','estimated'},'Location','southwest');
end
cla(h2); b = bar(h2,[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;
h2.XLim = [0.5,length(param_proxy_mean)+0.5]; h2.YLimMode = 'auto';
legend(h2,{'true','estimated'},'Location','northwest');
drawnow

end
```

Published with MATLAB® R2017a