ODE parameter inference with deGradInfer

Benn Macdonald and Frank Dondelinger

2017-11-29

The package `deGradInfer` is an R package for efficient parameter inference in systems of ordinary differential equations (ODEs), using adaptive gradient matching via Gaussian processes. This implementation follows from the work in Calderhead et al. (2009); Dondelinger et al. (2013) and Macdonald (2017), although the software has been extended with several crucial features to make it robust and useful for practical applications. Features include:

- Specifying custom ODE systems as R functions.
- Dealing with missing/unobserved variables.
- Visualization for monitoring MCMC run evolution.
- Specification of tolerable mismatch between data and inferred ODE dynamics.

In the next section, we will briefly sketch the underlying statistical model and inference, before moving on to the practical examples of how to use the package.

Model and Inference

The package implements the adaptive gradient matching approach of Dondelinger et al. (2013), with the option of using parallel tempering to fix the mismatch parameter in place, as described in Macdonald (2017). The advantage of gradient matching as a parameter inference technique is that it does not require us to numerically solve the ODE system at each step, a potentially costly operation for large systems.

Briefly, assuming Gaussian noise on the observed variable $\mathbf{y}_k$, $P(y_k(t)) = \mathcal{N}(y_k(t)|x_k(t), \sigma_k)$, we can model the latent variables $\mathbf{x}_k$ using a Gaussian process prior:

$$P(\mathbf{x}_k|\Psi_k) = \mathcal{N}(\mathbf{x}_k|\mathbf{0}, \mathbf{C}_{\Psi_k}),$$

where $\mathbf{C}_{\Psi_k}$ denotes a positive definite matrix of covariance functions with hyperparameters Ψ_k. We can use the properties of Gaussian processes to obtain a model (in closed form) for the gradient $\dot{\mathbf{x}}_k$, which we combine with the ODE system model via a product of experts:

$$P(\dot{\mathbf{x}}_k|\mathbf{X}, \Theta, \Psi_k, \gamma_k) \propto P(\dot{\mathbf{x}}_k|\mathbf{x}_k, \Psi_k)P(\dot{\mathbf{x}}_k|\mathbf{X}, \Theta, \gamma_k),$$

where $\mathbf{X}$ is a $T \times K$ matrix where column k corresponds to $\mathbf{x}_k$. Note that this approach essentially provides two models for $\dot{\mathbf{x}}_k$; a Gaussian process model which is based on a smoothing of the observed variables, and the ODE model, with parameters Θ. The parameters γ_k allow for some Gaussian noise on the ODE estimates of the gradients, and thus effectively control the allowable mismatch between the ODE and Gaussian process estimates.

Rather than inferring the gradient mismatch parameters, we can instead use parallel tempering to set them. If the ODE model properly (or adequately) describes the process then ideally we would want no (or little) mismatch between the predctions from the ODEs and the interpolant, when sampling from the posterior. However, bringing in this knowledge by forcing there to be no mismatch between the gradients at the beginning of the MCMC run can cause the algorithm to converge to suboptimal states. Instead, by running a number of parallel chains, setting large values of the gradient mismatch parameter for chains closer to the prior and small values for chains closer to the posterior, this prior knowledge can be brought in whilst still allowing the algorithm enough flexibility to explore. See Macdonald (2017) for details.

We integrate out the latent variables $\dot{\mathbf{X}}$ and do inference over the joint model $P(\mathbf{Y}, \mathbf{X}, \mathbf{\Theta}, \mathbf{\Psi}, \sigma)$ (see Dondelinger et al. (2013) for details). Inference is done using population MCMC, which can deal with multimodal posterior landscapes, and will give a posterior distribution for the parameters, in addition to the posterior mean estimates.

Correct format for a user specified prior

The package supports priors defined by the user. The user should write their function to return a vector of log densities, for a given parameter set. For example, a Uniform prior (lowerbound=0, upperbound=10) on parameter 1, a Gamma prior (shape=4, scale=0.5) on parameters 2 and 3 and a Chi-squared prior (degrees of freedom=2) on parameter 4 would take the form

```
testLogPrior <- function(params)
{
    return(c(dunif(params[1],0,10,log=TRUE),
        dgamma(params[2:3],shape=4,scale=0.5,log=TRUE),
        dchisq(params[4],2,log=TRUE)))
}
```

Specifying ODE systems as R functions

We will use the example of a simple Lotka-Volterra prey-predator system. This consists of two variables representing a predator species and a prey species. The system is governed by four parameters and takes the following form:

$$\frac{dx_1}{dt} = x_1(a - bx_2) \qquad \text{Prey} \frac{dx_2}{dt} = -x_2(cx_2 - dx_1) \qquad \text{Predators}$$

where x_1 represents the number of prey and x_2 represents the number of predators.

In order to use `deGradInfer`, we need to represent this system as an R function. Our implementation is inspired by the package `deSolve` for numerically solving ODE systems represented as R functions. In particular, we require the function to be of the form `func <- function(t,X,params,...)` where `t` is a T-dimensional vector of time points, `X` is a $T \times K$ matrix containing the values for the K variables in the system, `params` is a p-dimensional vector of parameters, and `...` denotes optional parameters that may be passed to the function. The output should be a $T \times K$ matrix of the gradients $\frac{dx_k}{dt}$ for all variables k. For the Lotka-Volterra system, we can define an R function as:

```
LV_func <- function(t, X, params) {
    dxdt = cbind(
      X[,1]*(params[1] - params[2]*X[,2]), # Prey
    - X[,2]*(params[3] - params[4]*X[,1])  # Predators
    )
    return(dxdt)
}
```

Note that if one wishes to use their function with `deSolve`, it needs to be compatible with scalar `t`, and additionally needs to return a list (this is a requirement by `deSolve` to allow for returning global values that are needed at each time point; we do not currently return any such global values). This is best achieved by writing a wrapper for the function:

```
library('deSolve')
```

```r
# Wrapper function
deSolve_LV_func = function(t,y,params) {
  list(LV_func(t,matrix(y,1,length(y)),params))
}

# Generate some data
test.times = seq(0,2,0.1)
test.data = ode(c(5,3), test.times, deSolve_LV_func, c(2,1,4,1))
test.data = test.data[,2:3] +
  rnorm(dim(test.data)[1]*2,0,0.1) # add some observational noise
```

Parameter inference in the Lotka-Volterra system

The main interface to the package is provided via the `agm` function. The user needs to provide the data for the observed time points, the R function representing the ODE system, the number of parameters in the system, and the expected standard deviation of the observational noise. Note that we do not currently allow the noise to be inferred for the gradient matching approach, as this tends to overestimate the true noise in the data.

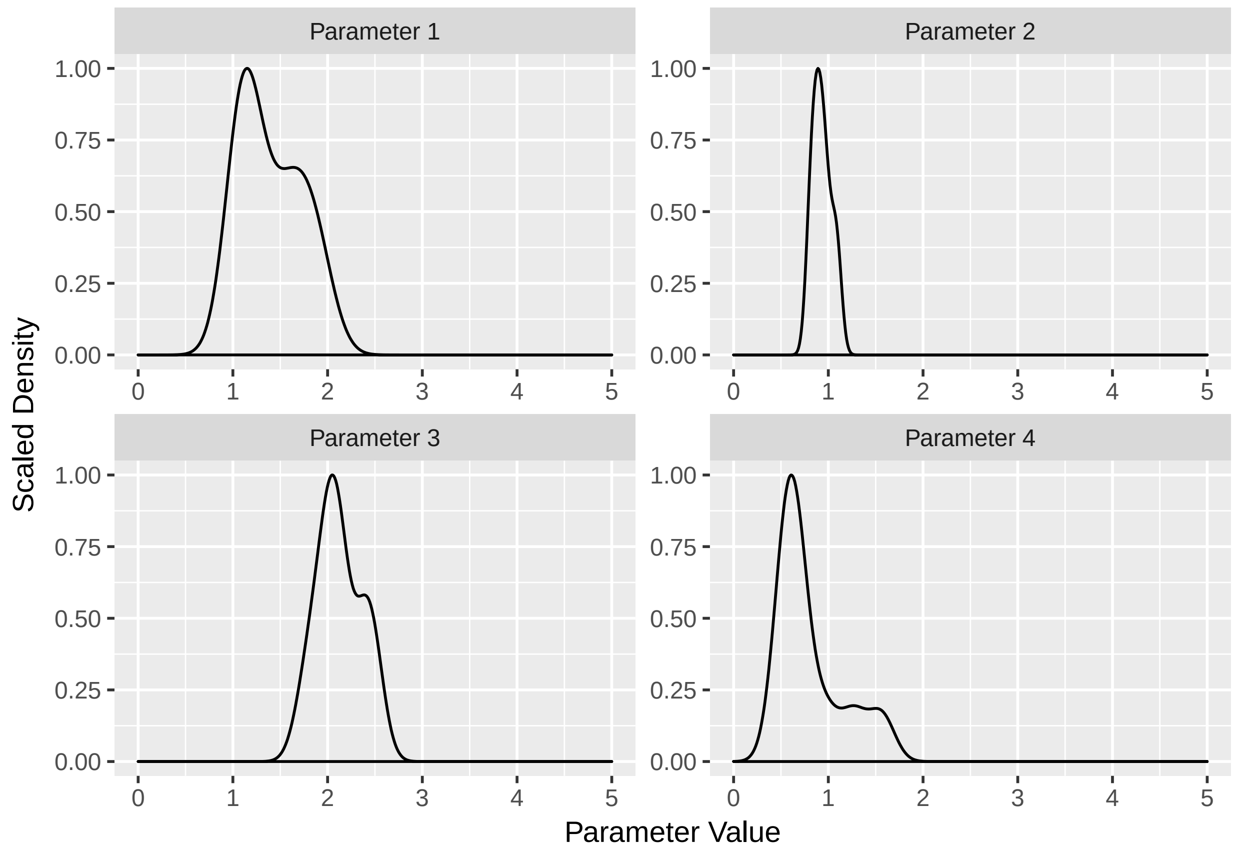
Here we only run the model for 1000 iterations, which is not sufficient to reach convergence, but is good enough for demonstration purposes:

```r
library('deGradInfer')

agm.result = agm(test.data,test.times,LV_func,4,noise.sd=0.1,
                 maxIterations=1000,chainNum=5)

print(agm.result$posterior.mean)
```

```
## [1] 1.4289245 0.9136871 2.1665483 0.7215048
```

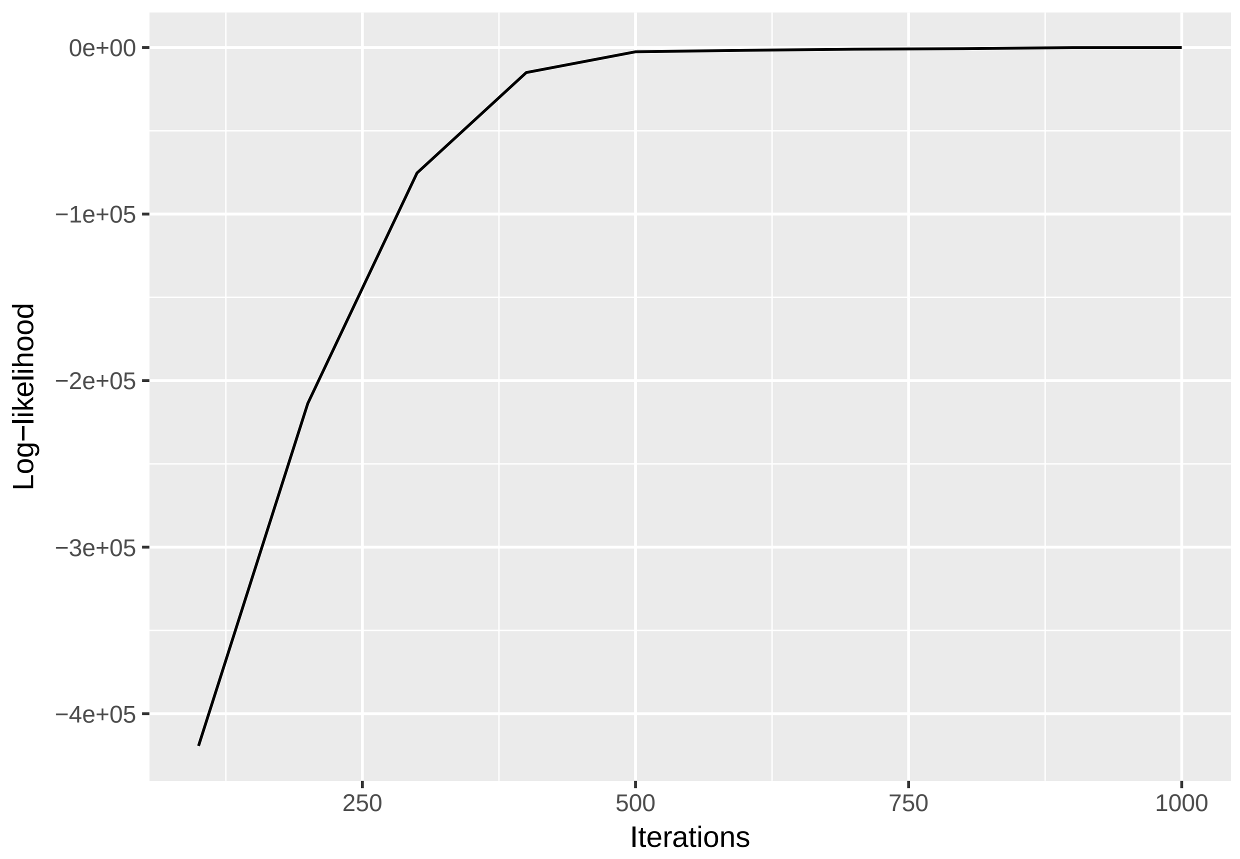
We can plot the posterior distribution of the inferred parameter values using the samples from the posterior.

```r
library('ggplot2')

plotting.frame = data.frame(Param.Values=c(agm.result$posterior.samples),
                            Param.Num=rep(1:4, each=dim(agm.result$posterior.samples)[1]))

ggplot(plotting.frame, aes(x=Param.Values, y=..scaled..)) +
  facet_wrap(~paste('Parameter', Param.Num), scales = 'free') +
  geom_density() +
  labs(y='Scaled Density', x='Parameter Value') +
  xlim(c(0,5))
```



We can also inspect the evolution of the log likelihood:

```r
qplot(x=seq(100,1000,100), y=c(agm.result$ll), geom = 'line') +
  labs(x='Iterations', y='Log-likelihood')
```



Tempering the mismatch parameter

The mismatch between the Gaussian process model and the ODE model of the system is captured by the parameter γ. Macdonald (2017) introduced a tempering scheme to replace naive inference of this parameter. This scheme can be used in our model by setting the `temperMismatchParameter` option.

```
agm.result = agm(test.data,test.times,LV_func,4,noise.sd=0.1,temperMismatchParameter=TRUE,
    originalSignalOnlyPositive=TRUE,
    logPrior="Gamma", defaultTemperingScheme="LB10")
```

Users can specify their own values for the tempering ladder by ensuring `temperMismatchParameter=TRUE`, `defaultTemperingScheme=FALSE` and then passing their matrix of values to `mismatchParameterValues`. The number of rows of the matrix should be equal to the number of parallel chains and the number of columns should be equal to the number of variables in the system. A typical ladder should have the largest value in the first row and the smallest value in the last row.

```
agm.result = agm(test.data,test.times,LV_func,4,noise.sd=0.1,temperMismatchParameter=TRUE,
    originalSignalOnlyPositive=TRUE,logPrior="Gamma",
    mismatchParameterValues=matrix(seq(0.01,1,length.out=20),nrow=20,ncol=2))
```

Parameter inference by explicitly solving the ODE system

For comparison purposes, we also implemented a version of the MCMC algorithm that explicitly solves the ODE system at each step (with `deSolve`), rather than using gradient matching. This can be invoked by using the `explicit` flag.

```
agm.result = agm(test.data,test.times,LV_func,numberOfParameters=6,explicit=TRUE)
```

Note that the number of parameters has increased; this is because the initial states of the ODE at the first time point also need to be inferred when using this method. So in the Lotka-Volterra case, the number of parameters increases by 2 because there are 2 variables in the ODE system.

A slightly bigger example

The package can in principle deal with ODE systems of any size, although in practice the achievable size will depend on memory and running time considerations. Here we show how to apply it to an ODE system with five equations and six parameters, derived from Vyshemirsky and Girolami (2007).

```
# Function encoding the Vyshemirsky and Girolami ODE model 1.
VG_func <- function(t, X, params) {
  MM <- ((params[5]*X[,5])/(params[6]+X[,5])) # Michaelis-Menten term


  dxdt <- cbind( - params[1]*X[,1] - params[2]*X[,1]*X[,3] +
                   params[3]*X[,4], # S
                 params[1]*X[,1], # dS
                 - params[2]*X[,1]*X[,3] + params[3]*X[,4] +
                   MM, # R
                 params[2]*X[,1]*X[,3] - params[3]*X[,4] -
                   params[4]*X[,4], # RS
                 params[4]*X[,4] - MM # Rpp
           )

  return(dxdt)
}

# Generate data
timeTest = c(0,1,2,4,5,7,10,15,20,30,40,50,60,80,100)
dataTest = ode(c(1,0,1,0,0), timeTest,
               function(t,y,params) list(VG_func(t,matrix(y,1,length(y)),params)), c(0.07,0.6,0.05,0.3,
dataTest = dataTest[,2:6] + rnorm(dim(dataTest)[1]*5,0,0.01)


# Run adaptive gradient matching
agm.result = agm(data=dataTest,time=timeTest,ode.system=VG_func, numberOfParameters=6)
```

Inference with unobserved variables

If not all variables are observed, we can infer the unobserved variables by using a latent Gaussian process. Note that the `data` matrix still needs to have a column for each variable, but unobserved variables can be set to `NA`, and the observed variables need to be specified in `observedVariables`. Partially observed variables are not currently supported.

```
# Remove the observations of the first variable
dataTest[,1] = NA
```

```r
# Run adaptive gradient matching
agm.result = agm(data=dataTest,time=timeTest,ode.system=VG_func, observedVariables=2:5, numberOfParamete
```

References

Calderhead, B., Girolami, M., Lawrence, N.D.: Accelerating Bayesian inference over nonlinear differential equations with Gaussian processes. In: Advances in neural information processing systems. pp. 217–224 (2009)

Dondelinger, F., Husmeier, D., Rogers, S., Filippone, M.: ODE parameter inference using adaptive gradient matching with Gaussian processes. In: Artificial intelligence and statistics. pp. 216–228 (2013)

Macdonald, B.: Statistical inference for ordinary differential equations using gradient matching, (2017)

Vyshemirsky, V., Girolami, M.A.: Bayesian ranking of biochemical system models. Bioinformatics. 24, 833–839 (2007)