Fitting Ornstein-Uhlenbeck-type Student's t-processes in Stan

With Applications for Population Dynamics Data

Aaron Goodman

Jan 10, 2018

Introduction

This work is part of my ongoing research using Bayesian hierarchical models for population dynamic inference. Here I present a portion of that research that I hope will be of broad interest to the Stan community. In this work, I present three Stan models for inferring state-space models driven by non-Gaussian Ornstein-Uhlenbeck processes and present some applications for population dynamic data.

Beyond modeling population dynamics, these non-Gaussian processes have been used to model stochastic volatility in finance, disease diffusion in epidemiology, and for Bayesian nonparametric function approximations. However, these models are typically fit using maximum likelihood estimation. By running these models in Stan, we are able to perform full Bayesian inference.

Furthermore, this work demonstrates how to implement non-Gaussian Lévy processes and the Matérn 1/2 covariance kernel in Stan. In doing so, we highlight some of the alterative formulations of this kernel and exploit them for more efficient Stan models.

Preliminaries

Gaussian processes are statistical models over a continuous domain in which any finite set of points are jointly distributed multivariate normal. These models are often used in time-series modeling, spatial statistics, and nonparametric function approximation.

One particularly useful Gaussian process is the Ornstein-Uhlenbeck process, which is a mean-reverting process. This process is a generalization of a random walk, in which the walk has a tendency to drift back towards a central location over time, and the rate at which the walk reverts is proportional to the distance from the central location. The Ornstein-Uhlenbeck has applications in many fields, such as physics to model the dynamics of springs, financial economics to model the volatility of asset returns (Vasicek model) and biology to model population dynamics (stochastic Gompertz model).

These Gaussian processes are often not observed directly, but represent some latent state of the system that is observed through some other noisy process. In the case of price volatility, the returns are observed, but the underlying variance is not. In population dynamics, the true population size is typically not measured directly, but observed through surveying a subset of the population or using a technique such as mark-recapture. Thus, we can model the dynamics using a stochastic process governing the underlying latent variable, and the measurement noise associated with observing the variable.

Ornstein-Uhlenbeck Process

The Ornstein-Uhlenbeck process can be seen as the continuous time generalization to an auto-regressive process. In an autoregressive model:

$$x_t = \alpha(\mu - x_{t-1}) + \sigma\epsilon$$

Where $\epsilon \sim \mathcal{N}(0,1)$. Under this formulation μ represents the mean to which the process reverts, *alpha* represents the strength of mean reversion, and σ represents the strength of the deviations from the mean.

The autorgressive model can be generalized to continuous time, where it becomes the Ornstein-Uhlenbeck process, and is characterized by the stochastic differential equation (SDE):

$$dx_t = \lambda(\mu - x_t)dt + \sqrt{2\kappa\lambda}\, dZ_t$$

Which states that the instantanous change of x at time t is governed by a deterministic portion and a stochastic portion. The deterministic portion is proportional to the mean reversion rate, *i.e.*, λ, and the distance of x at time t from the mean, *i.e.* μ. The stochastic portion, $\sqrt{2\kappa\lambda}dZ_t$ represents instantaneous noise, dZ_t scaled by $\sqrt{2\kappa\lambda}$ where κ represents the magnitude of deviations from the mean. In the case of the standard Ornstein-Uhlenbeck, the driving process dZ_t, is taken to be Brownian motion.

This formulation is used by Fuglstad among others (Fuglstad et al. 2017). However the SDE can also be written as $dx_t = \lambda(\mu - x_t)dt + \sigma dZ_t$. The Fuglstad representation is advantageous since $\frac{\sigma^2}{\lambda}$ is identifiable under infill asymptotics while σ and λ individually are not.

The half life, or expected time for a deviation from the mean to revert to half its magnitude is characterized by $\frac{\log 2}{\lambda}$.

When dZt is taken to be standard Brownian motion, and the process has reached its stationary distribution at t_0 then the value of X at time t, X_t, conditioned on X at the start of the observation, X_0 is normally distributed:

$$X_t|X_0 \sim \mathcal{N}\left(\mu - (\mu - X_0)e^{-\lambda t}, \kappa(1 - e^{-2\lambda t})\right)$$

and,

$$\mathrm{Cov}[X_t, X_{t+\Delta t}] = \kappa e^{-\lambda \Delta t}.$$

If we have discrete observations at times $\{t_1, t_2, \ldots, t_n\}$, which need not be spaced evenly, then the probability distribution can be expressed as a multivariate normal.

$$\vec{X} \sim \mathcal{N}(\mu, \Sigma)$$

Where: $\Sigma_{ij} = \kappa e^{-\lambda|t_i - t_j|}$.

This allows for a particularly efficient formulation in which Σ^{-1} is a tridiagonal matrix (Finley et al. 2009), with diagonal elements:

$$\Sigma^{-1}_{ii} = \begin{cases} 1 + \frac{e^{-2\lambda(t_2 - t_1)}}{1 - e^{-2\lambda(t_2 - t_1)}} & \text{for } i = 1 \\ 1 + \frac{e^{-2\lambda(t_n - t_{n-1})}}{1 - e^{-2\lambda(t_n - t_{n-1})}} & \text{for } i = n \\ 1 + \frac{e^{-2\lambda(t_i - t_{i-1})}}{1 - e^{-2\lambda(t_i - t_{i-1})}} + \frac{e^{-2\lambda(t_{i+1} - t_i)}}{1 - e^{-2\lambda(t_{i+1} - t_i)}} & \text{otherwise} \end{cases}$$

super- and sub-diagonal elements:

$$\Sigma^{-1}_{ij} = \begin{cases} \frac{e^{-\lambda|t_i - t_j|}}{1 - e^{-2\lambda|t_i - t_j|}} & \text{for } |i - j| = 1 \\ 0 & \text{for } |i - j| > 0 \end{cases}$$

Measurement process

Often the underlying data is not observed directly, but rather from some measurement process that introduces noise into the estimates. If the the measurement process introduces i.i.d. Gaussian noise with mean 0 and variance η we have.

$$\vec{X} \sim \mathcal{N}(\mu, \Sigma)$$

$$\vec{Y} \sim \mathcal{N}(\vec{X}, \eta\mathcal{I})$$

and the latent variable can be marginilized out to get

$$\vec{X} \sim \mathcal{N}(\mu, \Sigma + \eta\mathcal{I})$$

However, if the measurement process is a counting process, such as

$$Y_i \sim \text{Poisson}(X_i)$$

or if the driving process is something other than the Brownian motion, the latent varibale can not be marginilized out in this way.

Student's t-processes

The Gaussian Ornstein-Uhlenbeck process is often overly smooth since it does not allow for jumps in the data. A common extension to these models is to use a Lévy process as the driver of the Ornstein-Uhlenbeck dynamics (Barndorff-Nielsen and Shephard 2001). In choosing the Lévy process for the model, it is often convenient to start with a tractable, infinitely divisible, marginal distribution rather than with the Lévy process (Eliazar and Klafter 2005). One particularly useful generalization is the Student's t-process, which has an analytically tractable marginal distribution, and background driving Lévy process described by Heyde and Leonenko (2005) and Grigelionis (2013). The Student's t-distribution also has the convenient mathematical property of being elliptically symmetric, a fact that we exploit later in a reparameterization of the model. In fact, the Gaussian process and Student's t-process are the only elliptically symmetric processes with closed-form solutions (Shah, Wilson, and Ghahramani 2014).

We say that $\vec{y} \sim \text{MVT}_n(\nu, \vec{\mu}, \Sigma)$ when:

$$p(\vec{y}) = \frac{\Gamma(\frac{\nu+n}{2})}{((\nu-2)\pi)^{\frac{n}{2}}\Gamma(\frac{\nu}{2})}|\Sigma|^{-\frac{1}{2}} \times \left(1 + \frac{(\vec{y}-\vec{\mu})'\Sigma^{-1}(\vec{y}-\vec{\mu})}{\nu-2}\right)^{-\frac{\nu+n}{2}}$$

So in the OU-type Student's t-process, we will define the process in terms of the residuals, which becomes convenient for modeling the process in Stan.

$$\epsilon_1 = (X_1 - \mu)\kappa^{-\frac{1}{2}}$$

$$\epsilon_i = (X_i - \mu + (\mu - \epsilon_{i-1})e^{\lambda t_i})\kappa^{-\frac{1}{2}}\sqrt{1 - e^{-2\lambda(t_i - t_{i-1})}}$$

And thus

$$\epsilon_i|\epsilon_{1<j<i} \sim \text{MVT}_1\left(\nu + i, 0, \frac{\nu - 2 + \epsilon_1 + \cdots + \epsilon_{j-1}}{\nu + i}\right)$$

We can also model the process using the MVT specification:

$$\vec{X} \sim MVT_n(\nu, \mu, \Sigma)$$

Modeling

First let us load a package that we will need later: `rpgdd` installed from github using the `install_github` function from the `devtools` package.

```
library(rstan)
library(knitr)
library(stanhl)

library(Matrix)

library(rgpdd)

library(dplyr)
library(reshape2)
library(ggplot2)
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
theme_set(theme_light())
```

Synthetic Data

In actual data, there is often some underlying process driving the dynamics of the quantity of interest which is measured through another process that also introduces noise.

From here on, we will consider the case where the measurement process is a counting process.

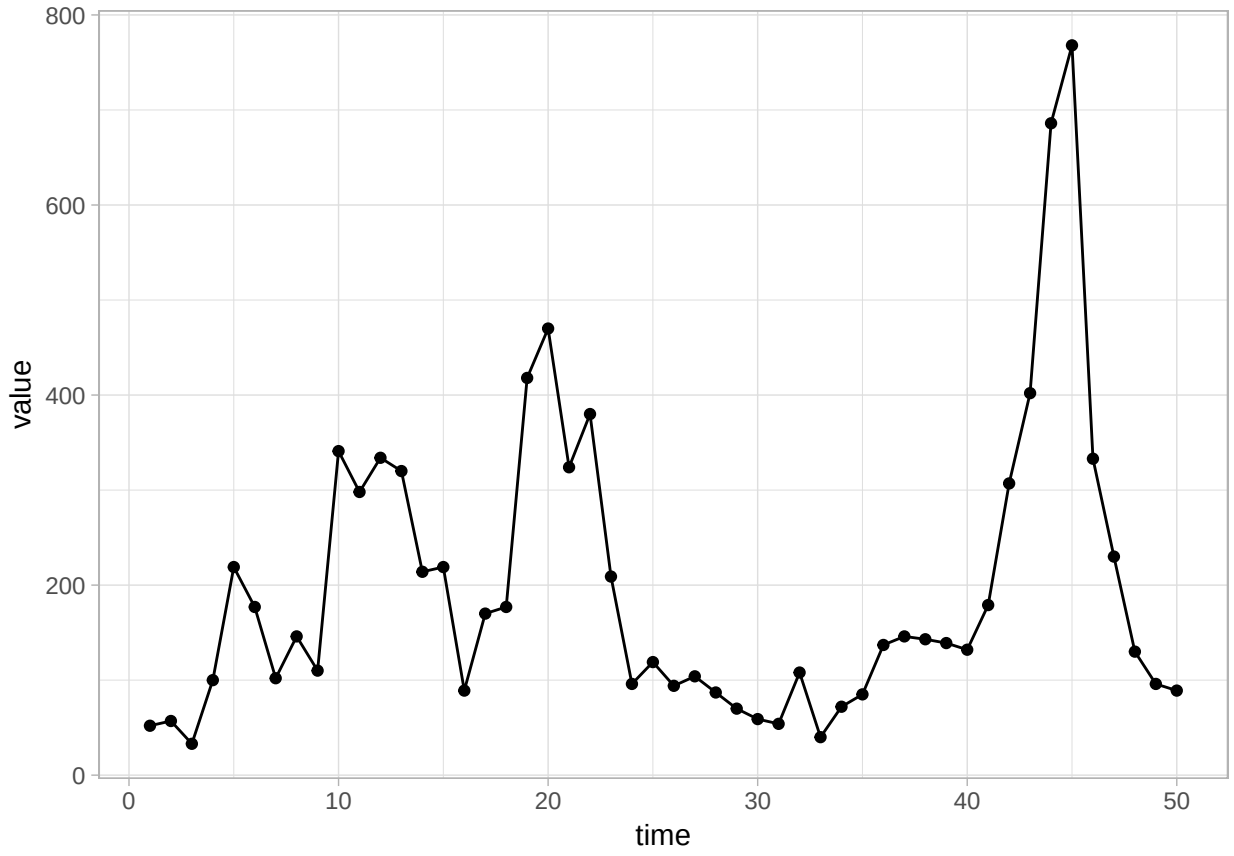
To simulate data from the Student's t-process, we will use the method of S. Yu, Tresp, and Yu (2007), of first simulating unit-random normal variates, scaling them by a random gamma $\left(\frac{\nu}{2}, \frac{\nu-2}{2}\right)$ variate.

```
###
### Generate data from an Student-t Ornstein-Uhlenbeck process
### lambda: Mean reversion parameter
### kappa: Deviation parameter = sigma^2/(2*lambda)
### intervals: Periods at which to generate sample
### t.df: Degrees of freedom for Student t-process, between 0 and infinity.
###       When t.df = Inf, this reduces to the Gaussian OU Process
###
generateStanData <- function(kappa,
                             lambda,
                             mu,
                             intervals,
                             t.df = Inf,
                             seed = 1){
  set.seed(seed)
  N <- length(intervals)
  lv.variates <- rnorm(N)
  dt <- outer(intervals,intervals,function(x,y) abs(x-y))
  x <- kappa * exp(-lambda*dt)
  L <- chol(x)

  scale <- if(is.finite(t.df)) rep(sqrt(rgamma(1,t.df/2,(t.df-2)/2)),each=N) else 1
  out.data <- list()
  out.data$latent_value <- as.vector(t(L) %*% (rnorm(N) * scale)) + mu
```

```
  out.data$value <- rpois(N,exp(out.data$latent_value))
  out.data$time <- intervals
  out.data$replicates <- 1L
  out.data$replicate_samples <- array(length(intervals))
  out.data$NSMPL <- length(intervals)
  out.data
}
```

```
stan.data <- generateStanData(kappa=0.75, lambda=.1, mu=log(100),intervals= 1:50, t.df=7)
plot.data <- as.data.frame(stan.data[c('time','value')])
ggplot(plot.data,
       aes(x=time,y=value))+geom_point()+geom_line()
```



The plotted synthetic data generated from the stochastic Gompertz model looks stylistically like population abundance data with boom and bust cycles that reverts to the stationary level of 100.

We now implement the Stan model using the conditional expectation formulation. Note that this is parameterized by ϵ_i rather than the latent parameters X_i.

```
state_space <- stan_model("state_space_noncenter.stan")
```

```
conditional.formulation <- sampling(state_space,stan.data,seed=123,open_progress=FALSE)
```

We can check how the model performed in terms of sampler warnings, diagnostics, and parameter recovery.

```
rstan:::throw_sampler_warnings(conditional.formulation)
```

```
## Warning: There were 5 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help.
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

```
## Warning: Examine the pairs() plot to diagnose sampling problems
```

```
sapply(conditional.formulation@sim[[1]],function(x) attr(x,'elapsed_time')) %>%
  kable(digits=1)
```

warmup	22.7	24.1	24.2	24.0
sample	20.0	22.6	21.1	21.4

```
summary(conditional.formulation,pars=c('kappa','mu','lambda','student_df'))[[1]] %>%
  kable(digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
kappa	0.832	0.018	0.412	0.298	0.537	0.752	1.010	1.871	509.884	1.008
mu	4.870	0.014	0.343	4.120	4.689	4.890	5.076	5.501	631.139	1.006
lambda	0.238	0.005	0.102	0.084	0.163	0.224	0.297	0.472	460.068	1.003
student_df	20.052	0.216	13.664	3.442	9.981	16.802	26.846	54.172	4000.000	1.000

We see that we had 5 divergent transitions, and each chain took about 40 seconds to run. Nevertheless, most of the mean values for the parameters were close to the input in the simulated data. However, the degrees of freedom of the Student t-distribution was not recovered well, and is very close to the prior distribution.

To avoid the divergent transitions, we can switch to using the centered parameterization of the model. The key difference is in the centered formulation we parameterize ϵ_i, use the unit t-distribution in the likelihood computation, and transform ϵ_i to X_i in the transformed parameter block. The full code is presented in the appendix.

```
conditional.center <- stan_model("state_space_center.stan")
```

```
conditional.center.formulation <- sampling(conditional.center,
                                           stan.data,seed=123,
                                           open_progress=FALSE)
```

```
rstan:::throw_sampler_warnings(conditional.center.formulation)
sapply(conditional.center.formulation@sim[[1]],function(x) attr(x,'elapsed_time')) %>%
  kable(digits=1)
```

warmup	2.3	2.7	2.6	2.8
sample	2.7	1.5	1.3	1.9

```
summary(conditional.center.formulation,pars=c('kappa','mu','lambda','student_df'))[[1]] %>%
  kable(digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
kappa	0.793	0.006	0.402	0.292	0.514	0.702	0.969	1.780	4000	1.000
mu	4.888	0.005	0.347	4.128	4.705	4.911	5.104	5.517	4000	1.000
lambda	0.239	0.002	0.103	0.085	0.164	0.227	0.298	0.470	4000	1.000
student_df	20.749	0.220	13.907	3.808	10.328	17.521	27.752	57.394	4000	0.999

This model runs in roughly 1/10th the time, has mean values as close to the true values as the previous model, and generates more effective samples.

We can use the multivariate expression of the stochastic process to reparameterize the model using the centered representation. Rather than parameterizing the model in terms of bivariate Student t-distributions, we parameterize it as a single multivariate Student t-distribution. Given the tridiagonal structure of the precision matrix, we can evaluate the log probability in $O(n)$ time.

```
gp.center <- stan_model("gp_center.stan")
```

```
gp.center.formulation <- sampling(gp.center,stan.data,seed=123,open_progress=FALSE)
```

```
rstan:::throw_sampler_warnings(gp.center.formulation)
sapply(gp.center.formulation@sim[[1]],function(x) attr(x,'elapsed_time')) %>%
  kable(digits=1)
```

warmup	1.0	1.4	1.6	1.7
sample	1.7	3.7	1.5	1.3

```
summary(gp.center.formulation,pars=c('kappa','mu','lambda','student_df'))[[1]] %>%
  kable(digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
kappa	0.913	0.007	0.463	0.323	0.586	0.813	1.130	2.114	4000	0.999
mu	4.566	0.005	0.300	3.939	4.377	4.585	4.783	5.094	4000	0.999
lambda	0.201	0.002	0.098	0.058	0.130	0.185	0.254	0.440	4000	1.000
student_df	21.107	0.211	13.368	4.140	11.146	18.308	27.969	54.434	4000	1.000

This offers a further speed up over the conditional probability formulation. This is the same model as the conditional formulation, just computed in a more efficient manner. We can confirm that the models are the same by seeing that the log likelihoods are the same given the same parameter values, up to some floating point issues.

```
set.seed(3456)
x <- rnorm(get_num_upars(gp.center.formulation));
log_prob(gp.center.formulation,x);
```

```
## [1] -840.7981
```

```
log_prob(conditional.center.formulation,x);
```

```
## [1] -840.7882
```

For completeness, we can also formulate the model where we use the noncentered parameterization in terms of μ and κ, but use the Gram matrix induced induced by the afformentioned kernel and parameterized by λ. Code for this model is in the appendix.

```
gp <- stan_model("gp_noncenter.stan")
```

```
prec.formulation <- sampling(gp,stan.data,seed=123,open_progress=FALSE)
```

```
rstan:::throw_sampler_warnings(prec.formulation)
sapply(prec.formulation@sim[[1]],function(x) attr(x,'elapsed_time')) %>%
  kable(digits=1)
```

warmup	9.2	6.0	9.2	7.6
sample	5.5	7.9	5.9	6.0

```
summary(prec.formulation,pars=c('kappa','mu','lambda','student_df'))[[1]] %>%
  kable(digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
kappa	1.016	0.072	0.511	0.394	0.665	0.898	1.214	2.436	50.236	1.077
mu	4.555	0.038	0.283	3.924	4.376	4.584	4.763	5.007	55.389	1.117
lambda	0.195	0.009	0.099	0.054	0.121	0.178	0.251	0.434	121.114	1.052
student_df	21.582	0.609	14.358	3.963	11.170	18.200	28.481	57.798	555.600	1.007

This model is less sample efficient than any of the previous models, but unlike the noncentered precision matrix parameterization, using the noncentered conditional formulation does not lead to divergent transitions.

Population Dynamic Models

We can apply the Ornstein-Uhlenbeck model to population dynamics to predict population sizes in the future and infer the carrying capacity of an ecosystem.. The stochastic Gompertz model of density-limited population growth is equivalent to an OU process in log space (Dennis and Ponciano 2014). To test the model, we will fit it to the population dynamics of the North American badger from the Global Poplution Dynamics Database ("Global Population Dynamics Database (Version 2)" 2010).

```
###
### Create a dataset for our stan models from the gpdd
### dataest.
### id: sample id in gppdd
### n: Take the first n data points from the dataset, and hold out the remainder.
###
buildStanDataGPDD <- function(id,n=Inf){
```

```
  x <- dplyr::filter(gpdd_data,MainID == id) %>% head(n)
  time <- x$SeriesStep
  value <- x$PopulationUntransformed
  list(time = time,
       value = value,
       replicates = 1L,
       NSMPL = length(time),
       replicate_samples = array(length(time)))
}
```

```
taxa.id <- 70
train <- 30
pop.data <- buildStanDataGPDD(taxa.id,train)
pop.data.all <- buildStanDataGPDD(taxa.id)
pop.params <- sampling(gp.center,pop.data,seed=123,open_progress=FALSE)
```

```
summary(pop.params,pars=c('kappa','mu','lambda','student_df'))[[1]] %>% kable(digits=3)
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
kappa	2.430	0.015	0.951	0.833	1.765	2.350	2.983	4.542	4000	1.000
mu	2.458	0.009	0.597	1.248	2.080	2.474	2.843	3.608	4000	1.000
lambda	0.698	0.004	0.273	0.289	0.516	0.657	0.838	1.345	4000	1.001
student_df	17.514	0.201	12.689	3.279	8.063	14.257	23.152	50.677	4000	0.999

Thus we expect the carrying capacity of the population to be around $e^{\mu} \approx 12$ individuals, and the population to have a recovery half time of around $\log(2)/\lambda \approx 1$ year, consistent with the yearly reproductive cycle of the badger population.

```
###
###
generateStanDataConditional <- function(kappa,
                                lambda,
                                mu,
                                x0,
                                intervals,
                                t.df = Inf,
                                seed = 1){
  set.seed(seed)
  N <- length(intervals)
  lv.variates <- rnorm(N)
  dt <- outer(intervals,intervals,function(x,y) abs(x-y))
  min.t <- outer(intervals,intervals,function(x,y) pmin(x,y))
  mu_ <- mu - (mu - x0) * exp(-lambda*intervals)
  x <- kappa * exp(-lambda*dt) * (1-exp(-2*lambda*min.t))
  L <- chol(x)
  scale <- if(is.finite(t.df)) rep(sqrt(rgamma(1,t.df/2,(t.df-2)/2)),each=N) else 1
  out.data <- list()
  out.data$latent_value <- as.vector(t(L) %*% (rnorm(N) * scale)) + mu_
  out.data$value <- suppressWarnings(rpois(N,exp(out.data$latent_value)))
  out.data$time <- intervals
  out.data$replicates <- 1L
```

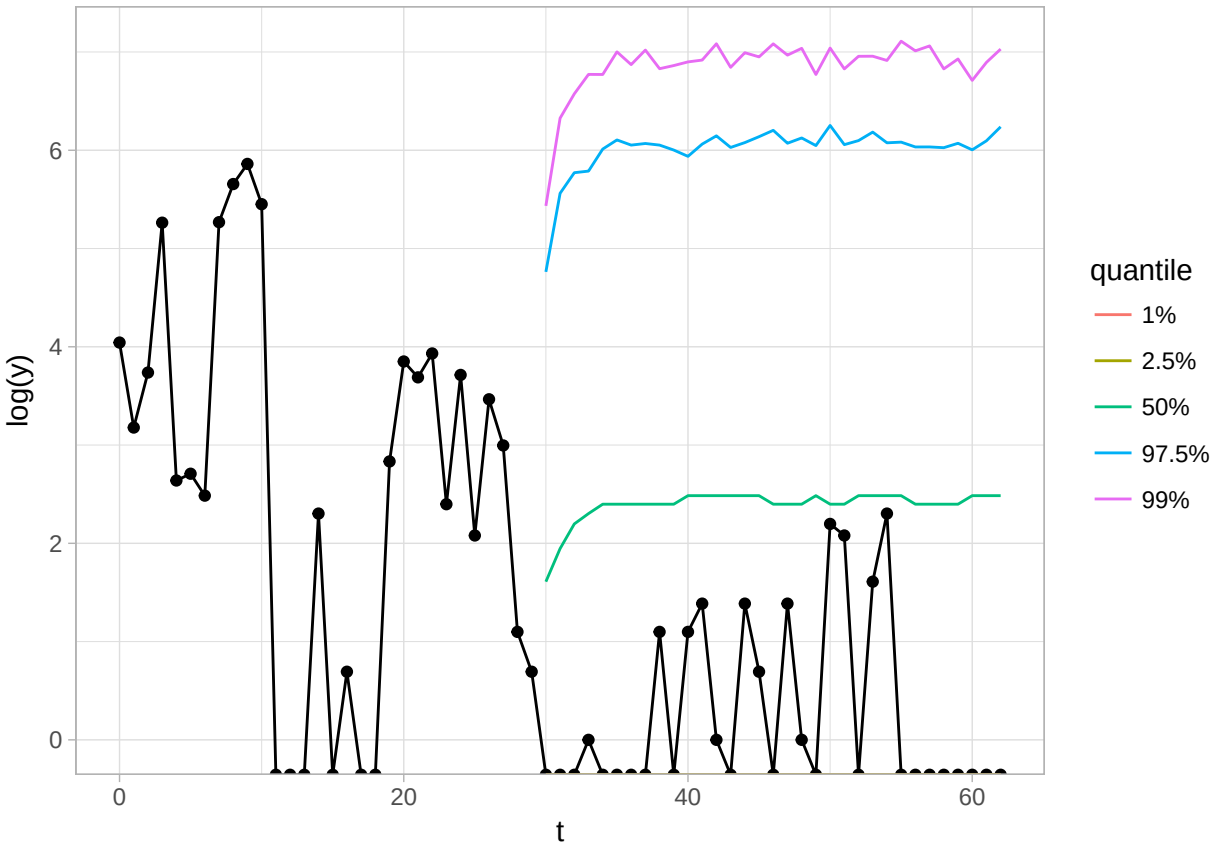
```
  out.data$replicate_samples <- array(length(intervals))
  out.data$NSMPL <- length(intervals)
  out.data
  }
```

```
train.name <- paste0('latent_value[',train,']')
pars <- extract(pop.params,c("lambda","mu","kappa",train.name,"student_df"))
simulated <- matrix(ncol=length(pars$lambda),
                    nrow=pop.data.all$NSMPL-train)
for(i in 1:length(pars$lambda)){
  simulated[,i] <- generateStanDataConditional(pars$kappa[i],
                               pars$lambda[i],
                               pars$mu[i],
                               pars[[train.name]][i],
                               tail(pop.data.all$time,-train) - pop.data.all$time[train],
                               t.df = pars$student_df[i],
                               seed = sample.int(.Machine$integer.max, 1))$value
}

df <- data.frame(t = pop.data.all$t,
                 y = pop.data.all$value)
df.2 <- cbind(t= tail(pop.data.all$time,-train), as.data.frame(t(apply(simulated,1,function(x) quantile
```

We can plot the population dynamics of the (log) of the badger population counts. We are fitting the data on the first 30 years of data, and then simulating data from the parameters for the remaining 33 years of observations to generate a posterior predictive distribution.

```
ggplot(df,aes(x=t,y=log(y)))+geom_point()  +geom_line()+ theme_light() +
  geom_line(aes(color=quantile),data=df.2)
```



We see that the observed badger counts fall within the the 95% posterior range for the remaning time. Even though the badger population appears to go extinct at year 56, this is within the range predicted by our model.

Conclusion

In this work I have presented several methods to of estimating Ornstein-Uhlenbeck-type Student's t-processes using Stan, and shown a potential application in modeling population dynamics. I have presented three different ways to parameterize the model, one of which has two Stan implementations. These efficient implementations of OU-type Student's t-processes could have uses beyond biology, such as in Bayesian nonparametric function approximations and in financial time-series modeling.

Acknowledgements

I would like to thank Elhanan Borenstein and the Borenstein lab for helpful discussions and insights into microbial population dynamics modeling. I would also like to thank Bob Carpenter, Michael Betancourt and the members of the Stan Discourse forum for all the advice they have given me on this project.

Appendix

Conditional formulation, noncentered parameterization

```
invisible(gsub("@","\\@",stanhl(state_space@model_code[1])))
```

```
//Copyright 2017 Aaron Goodman <aaronjg@stanford.edu>. Licensed under the GPLv3 or later.
data{
  int <lower=0> NSMPL;
  int <lower=0> replicates;
  int <lower=0> replicate_samples[replicates];
  int <lower=0> value[NSMPL];
  vector [NSMPL] time;
}
transformed data{
  vector[NSMPL] value_vec = to_vector(value);
  vector[NSMPL] time_vec = to_vector(time);
  int sample_start[replicates];
  sample_start[1] = 1;
  if(replicates > 1)
    for(i in 2:replicates)
      sample_start[i] = sample_start[i-1] + replicate_samples[i-1];
}

parameters{
  real lambda_log;
  real kappa_log;
  real mu;
  real <lower=2> student_df;
  vector[NSMPL] latent_value_raw;
}

transformed parameters{
  real sigma_log = 0.5*(kappa_log + lambda_log + log2());
  real sigma = exp(sigma_log);
  real lambda = exp(lambda_log);
  real kappa = exp(kappa_log);
  real kappa_inv = exp(-kappa_log);
  real kappa_sqrt = exp(0.5*kappa_log);
  vector[NSMPL] latent_value;

  // For each raw latent value, transform it using the
  // conditional expression for the OU process.
  for(i in 1:replicates){
    int n = replicate_samples[i];
    int offset = sample_start[i];
    vector [n-1] time_diff = segment(time_vec, offset + 1, n - 1) -
      segment(time_vec,offset,n-1);
    real cum_squares = 0;
    real last_t = -1;
    real lv;
    for(k in 0:n-1){
```

```
      real lv_raw = latent_value_raw[offset + k];
      if(k == 0){
        //For the first latent value use the stationary distribution.
        lv = mu + lv_raw * kappa_sqrt;
      }else{
        real t = time_diff[k];
        real exp_neg_lambda_t = exp(-t*lambda);
        real sd_scale = kappa_sqrt .* sqrt(1-square(exp_neg_lambda_t));
        lv = mu - (mu - lv) .* exp_neg_lambda_t + lv_raw .* sd_scale;
        last_t = t;
      }
      latent_value[offset+k] = lv;
    }
  }
}
model {
  target += lambda_log;
  target += kappa_log;
  student_df ~ gamma(2,.1);

  // Increment the log probability according to the conditional expression for
  // the unit multivariate t-distribution.
  for(i in 1:replicates){
    int n = replicate_samples[i];
    int offset = sample_start[i];
    vector [n] sq_lv = square(segment(latent_value_raw,offset,n));
    vector [n] cum_squares = cumulative_sum(append_row(0,sq_lv[1:n-1]));
    for(k in 1:NSMPL){
      target +=  (lgamma((student_df + k) * 0.5) - lgamma((student_df+ k - 1 )* 0.5));
      target += -0.5 * (student_df + k) * log1p(sq_lv[k] / (student_df + cum_squares[k] - 2));
      target += -0.5 * log(student_df + cum_squares[k] - 2);
    }
  }
  // Add the log probability for the observations given the latent values
  value ~ poisson_log(latent_value);

  // Prior probabilities.
  lambda ~ gamma(2,2);
  kappa ~ gamma(2,2);
  mu ~ normal(0,5);
}

generated quantities{
  real carrying_capacity = mu + kappa;
}
```

Conditional formulation, centered parameterization

```
invisible(gsub("@","\\@",stanhl(conditional.center@model_code[1])))
```

```
//Copyright 2017 Aaron Goodman <aaronjg@stanford.edu>. Licensed under the GPLv3 or later.
data{
```

```
  int <lower=0> NSMPL;
  int <lower=0> replicates;
  int <lower=0> replicate_samples[replicates];
  int <lower=0> value[NSMPL];
  vector [NSMPL] time;
}
transformed data{
  vector[NSMPL] value_vec = to_vector(value);
  vector[NSMPL] time_vec = to_vector(time);
  int sample_start[replicates];
  vector [NSMPL - replicates] time_diffs;

  sample_start[1] = 1;
  if(replicates > 1)
    for(i in 2:replicates)
      sample_start[i] = sample_start[i-1] + replicate_samples[i-1];

  for(i in 1:replicates){
    int smpl = replicate_samples[i];
    int start = sample_start[i];
    time_diffs[(start - (i-1)):(start - (i - 1) + smpl-2)]  = segment(time_vec, start + 1, smpl - 1) -
      segment(time_vec, start, smpl - 1);
  }
}

parameters{
  real lambda_log;
  real kappa_log;
  real mu;
  real <lower=2> student_df;
  vector[NSMPL] latent_value;
}

transformed parameters{
  real sigma_log = 0.5*(kappa_log + lambda_log + log2());
  real sigma = exp(sigma_log);
  real lambda = exp(lambda_log);
  real kappa = exp(kappa_log);
  real kappa_inv = exp(-kappa_log);
  real kappa_sqrt = exp(0.5*kappa_log);
  vector[NSMPL] latent_value_raw;

  for(i in 1:replicates){
    int n = replicate_samples[i];
    int offset = sample_start[i];
    vector [n-1] time_diff = segment(time_diffs, offset - (i-1), n - 1);
    real cum_squares = 0;
    real last_t = -1;
    real lv_raw;
    for(k in 0:n-1){
      real lv = latent_value[offset + k];
      if(k == 0){
        lv_raw = (lv - mu)/kappa_sqrt;
      }else{
```

```
        real t = time_diff[k];
        real exp_neg_lambda_t = exp(-t*lambda);
        real sd_scale = kappa_sqrt .* sqrt(1-square(exp_neg_lambda_t));
        lv_raw = (lv - mu - (latent_value[offset + k - 1] - mu) * exp_neg_lambda_t)/sd_scale;
      }
      latent_value_raw[offset+k] = lv_raw;
    }
  }
}
model {
  target += lambda_log;
  target += kappa_log;
  student_df ~ gamma(2,.1);
  for(i in 1:replicates){
    int n = replicate_samples[i];
    int offset = sample_start[i];
    vector [n] sq_lv = square(segment(latent_value_raw, offset, n));
    vector [n] cum_squares = cumulative_sum(append_row(0, sq_lv[1:n-1]));
    for(k in 1:NSMPL){
      target +=  (lgamma((student_df + k) * 0.5) - lgamma((student_df+ k - 1 )* 0.5));
      target += -0.5 * (student_df + k) * log1p(sq_lv[k] / (student_df + cum_squares[k] - 2));
      target += -0.5 * log(student_df + cum_squares[k] - 2);
    }
  }

  //jacobian of the transformation
  target += -NSMPL * (0.5 * kappa_log);
  target += -0.5 * log1m_exp(-2*lambda*time_diffs);

  value ~ poisson_log(latent_value);
  lambda ~ gamma(2,2);
  kappa ~ gamma(2,2);
  mu ~ normal(0,5);
}

generated quantities{
  real carrying_capacity = mu + kappa;
}
```

Precision formulation, noncentered parameterization

```
invisible(gsub("@","\\@",stanhl(gp@model_code[1])))
```

```
//Copyright 2017 Aaron Goodman <aaronjg@stanford.edu>. Licensed under the GPLv3 or later.
functions{

  // helper functions for calculating the precion matrix
  real exp_logit(real x){
    return x / (1 - x);
  }
  real exp_half_logit(real x){
    return  x / (1 - square(x));
```

```
  }
  real on_diagonal_diff(real sq_exp_lambda_t){
    return(1+exp_logit(sq_exp_lambda_t));
  }
  real on_diagonal_middle_diff(real sq_exp_lambda_t1, real sq_exp_lambda_t2){
    return(1+exp_logit(square(sq_exp_lambda_t1)) + exp_logit(square( sq_exp_lambda_t2)));
  }
  real off_diagonal_diff(real exp_lambda_t){
    return(- exp_half_logit(exp_lambda_t));
  }

  // calculate the determinant of the a symmetric, tridiagonal matrix with
  // diagonal elements in m[1,1:n] and off diagonals in m[2,1:n-1]
  real tridiag_det(real [,] m){
    real f0 = 1;
    real f1 = m[1,1];
    real ftemp;
    for(i in 2:dims(m)[2]){
      ftemp = f1;
      f1 = m[1,i] .* f1 - square(m[2,i-1]) .* f0;
      f0 = ftemp;
    }
    return f1;
  }

  // calculate the precision matrix of an OU covariance kernel
  // with mean reversion parameter lambda and time intervals week_num
  real [,] ou_precision(real []week_num,real lambda){
    int n = size(week_num);
    vector [n] weeks = to_vector(week_num);
    real  ou_p [2,n];
    real min_diff = min(weeks[2:] - weeks[:n-1]);
    ou_p[1,1] = on_diagonal_diff(exp((week_num[1] - week_num[2])*lambda));
    ou_p[1,n] = on_diagonal_diff(exp((week_num[n-1] - week_num[n])*lambda));
    for(k in 1:(n-1)){
      real difference = week_num[k+1] - week_num[k];
      real exp_lambda_t = exp(lambda *(-difference));
      real off_diag = off_diagonal_diff(exp_lambda_t);
      ou_p[2,k] = off_diag;
      if(k != 1){
        real back_difference = week_num[k] - week_num[k-1];
        real exp_lambda_t0 = exp(lambda *(-back_difference));
        real on_diag = on_diagonal_middle_diff(exp_lambda_t0,exp_lambda_t);
        ou_p[1,k] = on_diag;
      }
    }
    return(ou_p);
  }
}
data{
  int <lower=0> NSMPL;
  int <lower=0> replicates;
  int <lower=0> replicate_samples[replicates];
  int <lower=0> value[NSMPL];
```

```
  vector [NSMPL] time;
}

transformed data{
  vector[NSMPL] value_vec = to_vector(value);
  vector[NSMPL] time_vec = to_vector(time);
  real mean_temporal_difference = (time[NSMPL] - time[1]) / (rows(time) - 1);
  int sample_start[replicates];
  sample_start[1] = 1;
  if(replicates > 1)
    for(i in 2:replicates)
      sample_start[i] = sample_start[i-1] + replicate_samples[i-1];
}

parameters{
  real lambda_log;
  real kappa_log;
  real mu;
  real <lower=2> student_df;
  vector[NSMPL] latent_value_raw;
}

transformed parameters{
  real sigma_log = 0.5*(kappa_log + lambda_log + log2());
  real sigma = exp(sigma_log);
  real lambda = exp(lambda_log);
  real kappa = exp(kappa_log);
  real kappa_inv = exp(-kappa_log);
  real kappa_sqrt = exp(0.5*kappa_log);
  vector[NSMPL] latent_value = mu + latent_value_raw*kappa_sqrt;
}
model {
  target += lambda_log;
  target += kappa_log;
  {
    int sample_offset = 1;
    int taxa_offset = 1;
    for(i in 1:replicates){
      int n = replicate_samples[i];
      int offset = sample_start[i];
      real next_lv;
      real prev_lv;
      vector[n] tot;
      real time_sample [n] = to_array_1d(segment(time,offset,n));
      real ou_p [2,n] = ou_precision(time_sample,lambda);
      for(k in 0:n){
        real normalized;
        real lv = next_lv;
        if(k > 0){
          normalized = lv * ou_p[1,k];
        }
        if(k > 1){
          normalized = normalized + prev_lv * ou_p[2,k-1];
        }
```

```
        if(k < n){
          prev_lv = lv;
          next_lv = latent_value_raw[offset+k];
          if(k > 0){
            normalized = normalized + next_lv * ou_p[2,k];
          }
        }
        if(k > 0){
          tot[k] = normalized .* lv;
        }
      }
      target += -0.5*(student_df + n) * log1p(sum(tot) /
                                              (student_df - 2));
      target +=  -n * 0.5 * log(student_df - 2) +
                  lgamma(0.5*(student_df + n)) -
                  lgamma(0.5*student_df);
      target += 0.5 * log(tridiag_det(ou_p));
    }
  }

  student_df ~ gamma(2,.1);
  lambda ~ gamma(2,2);
  kappa ~ gamma(2,2);
  mu ~ normal(0,5);

  value ~ poisson_log(latent_value);
}
generated quantities{
  real carrying_capacity = exp(mu + kappa);
}
```

Precision formulation, centered parameterization

```
invisible(gsub("@","\\@",stanhl(gp.center@model_code[1])))
```

```
//Copyright 2017 Aaron Goodman <aaronjg@stanford.edu>. Licensed under the GPLv3 or later.
functions{
  real exp_logit(real x){
    return x / (1 - x);
  }
  real exp_half_logit(real x){
    return  x / (1 - square(x));
  }
  real on_diagonal_diff(real sq_exp_lambda_t){
    return(1+exp_logit(sq_exp_lambda_t));
  }
  real on_diagonal_middle_diff(real sq_exp_lambda_t1, real sq_exp_lambda_t2){
    return(1+exp_logit(square(sq_exp_lambda_t1)) + exp_logit(square( sq_exp_lambda_t2)));
  }
  real off_diagonal_diff(real exp_lambda_t){
    return(- exp_half_logit(exp_lambda_t));
  }
```

```
  real tridiag_det(real [,] m){
    real f0 = 1;
    real f1 = m[1,1];
    real ftemp;
    for(i in 2:dims(m)[2]){
      ftemp = f1;
      f1 = m[1,i] .* f1 - square(m[2,i-1]) .* f0;
      f0 = ftemp;
    }
    return f1;
  }
  real [,] ou_precision(real []week_num,real lambda){
    int n = size(week_num);
    vector [n] weeks = to_vector(week_num);
    real  ou_p [2,n];
    real min_diff = min(weeks[2:] - weeks[:n-1]);
    ou_p[1,1] = on_diagonal_diff(exp((week_num[1] - week_num[2])*lambda));
    ou_p[1,n] = on_diagonal_diff(exp((week_num[n-1] - week_num[n])*lambda));
    for(k in 1:(n-1)){
      real difference = week_num[k+1] - week_num[k];
      real exp_lambda_t = exp(lambda *(-difference));
      real off_diag = off_diagonal_diff(exp_lambda_t);
      ou_p[2,k] = off_diag;
      if(k != 1){
        real back_difference = week_num[k] - week_num[k-1];
        real exp_lambda_t0 = exp(lambda *(-back_difference));
        real on_diag = on_diagonal_middle_diff(exp_lambda_t0,exp_lambda_t);
        ou_p[1,k] = on_diag;
      }
    }
    return(ou_p);
  }
}
data{
  int <lower=0> NSMPL;
  int <lower=0> replicates;
  int <lower=0> replicate_samples[replicates];
  int <lower=0> value[NSMPL];
  vector [NSMPL] time;
}
transformed data{
  vector[NSMPL] value_vec = to_vector(value);
  vector[NSMPL] time_vec = to_vector(time);
  real mean_temporal_difference = (time[NSMPL] - time[1]) / (rows(time) - 1);
  int sample_start[replicates];
  sample_start[1] = 1;
  if(replicates > 1)
    for(i in 2:replicates)
      sample_start[i] = sample_start[i-1] + replicate_samples[i-1];
}
parameters{
  // Parameterize on untransformed parameters and handle the
  // Jacobian transformation manually, since we will need
  // the log anyway
```

```
  real lambda_log;

  // kappa = sigma^2/(2*lambda) and model is more stable
  // under this parameterization
  real kappa_log;
  real mu;

  real <lower=2> student_df;
  vector[NSMPL] latent_value;
}

transformed parameters{
  real sigma_log = 0.5*(kappa_log + lambda_log + log2());
  real sigma = exp(sigma_log);
  real lambda = exp(lambda_log);
  real kappa = exp(kappa_log);
  real kappa_inv = exp(-kappa_log);
  real kappa_sqrt = exp(0.5*kappa_log);
  real kappa_sqrt_inv = exp(-0.5*kappa_log);
  vector[NSMPL] latent_value_raw = (latent_value - mu)*kappa_sqrt_inv;
}

model {
  // Add Jacobian transformations for lambda and kappa
  target += lambda_log;
  target += kappa_log;

  {
    int sample_offset = 1;
    int taxa_offset = 1;
    for(i in 1:replicates){
      int n = replicate_samples[i];
      int offset = sample_start[i];
      real next_lv;
      real prev_lv;
      vector[n] tot;
      real time_sample [n] = to_array_1d(segment(time, offset, n));
      real ou_p [2,n] = ou_precision(time_sample, lambda);
      for(k in 0:n){
        real normalized;
        real lv = next_lv;
        if(k > 0){
          normalized = lv * ou_p[1,k];
        }
        if(k > 1){
          normalized = normalized + prev_lv * ou_p[2,k-1];
        }
        if(k < n){
          prev_lv = lv;
          next_lv = latent_value_raw[offset+k];
          if(k > 0){
            normalized = normalized + next_lv * ou_p[2,k];
          }
        }
```

```
        if(k > 0){
          tot[k] = normalized .* lv;
        }
      }
      target += -0.5*(student_df + n) * log1p(sum(tot) /
                                              (student_df - 2));
      target += lgamma(0.5*(student_df + n)) - lgamma(0.5*student_df) - 0.5*n * log(student_df -2);
      target += 0.5 * log(tridiag_det(ou_p));
      target += -0.5 * n * kappa_log;
    }
  }

  student_df ~ gamma(2,.1);
  lambda ~ gamma(2,2);
  kappa ~ gamma(2,2);
  mu ~ normal(0,5);

  value ~ poisson_log(latent_value);
}
generated quantities{
  real carrying_capacity = exp(mu + kappa);
}
```

Original Computing Environment

```
devtools::session_info("rstan")
```

```
## Session info -------------------------------------------------------------

##  setting  value
##  version  R version 3.4.3 (2017-11-30)
##  system   x86_64, linux-gnu
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  tz       Canada/Pacific
##  date     2018-01-19

## Packages -----------------------------------------------------------------

##  package      * version   date       source
##  BH             1.62.0-1  2016-11-19 CRAN (R 3.4.1)
##  colorspace     1.3-2     2016-12-14 CRAN (R 3.4.1)
##  dichromat      2.0-0     2013-01-24 CRAN (R 3.4.1)
##  digest         0.6.12    2017-01-27 CRAN (R 3.4.1)
##  ggplot2      * 2.2.1     2016-12-30 CRAN (R 3.4.1)
##  graphics     * 3.4.3     2017-12-01 local
##  grDevices    * 3.4.3     2017-12-01 local
##  grid           3.4.3     2017-12-01 local
##  gridExtra      2.2.1     2016-02-29 CRAN (R 3.4.1)
##  gtable         0.2.0     2016-02-26 CRAN (R 3.4.1)
```

```
##  inline         0.3.14    2015-04-13 CRAN (R 3.4.1)
##  labeling       0.3       2014-08-23 CRAN (R 3.4.1)
##  lattice        0.20-35   2017-03-25 CRAN (R 3.3.3)
##  lazyeval       0.2.1     2017-10-29 CRAN (R 3.4.2)
##  magrittr       1.5       2014-11-22 CRAN (R 3.4.1)
##  MASS           7.3-47    2017-04-21 CRAN (R 3.4.0)
##  Matrix       * 1.2-11    2017-08-16 CRAN (R 3.4.1)
##  methods      * 3.4.3     2017-12-01 local
##  munsell        0.4.3     2016-02-13 CRAN (R 3.4.1)
##  plyr           1.8.4     2016-06-08 CRAN (R 3.4.1)
##  RColorBrewer   1.1-2     2014-12-07 CRAN (R 3.4.1)
##  Rcpp           0.12.12   2017-07-15 CRAN (R 3.4.1)
##  RcppEigen      0.3.3.3.0 2017-05-01 CRAN (R 3.4.1)
##  reshape2     * 1.4.2     2016-10-22 CRAN (R 3.4.1)
##  rlang          0.1.1     2017-05-18 CRAN (R 3.4.1)
##  rstan        * 2.16.2    2017-07-03 CRAN (R 3.4.1)
##  scales         0.4.1     2016-11-09 CRAN (R 3.4.1)
##  StanHeaders  * 2.16.0-1  2017-07-03 CRAN (R 3.4.1)
##  stats        * 3.4.3     2017-12-01 local
##  stats4         3.4.3     2017-12-01 local
##  stringi        1.1.5     2017-04-07 CRAN (R 3.4.1)
##  stringr        1.2.0     2017-02-18 CRAN (R 3.4.1)
##  tibble         1.3.3     2017-05-28 CRAN (R 3.4.1)
##  tools          3.4.3     2017-12-01 local
##  utils        * 3.4.3     2017-12-01 local
```

Licenses

Bibliography

Barndorff-Nielsen, Ole E., and Neil Shephard. 2001. "Non-Gaussian Ornstein-Uhlenbeck-based models and some of their uses in financial economics." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63 (2): 167–241. doi:10.1111/1467-9868.00282.

Dennis, Brian, and Josémiguel Ponciano. 2014. "Density-dependent state-space model for population-abundance data with unequal time intervals." *Ecology* 95 (8): 2069–76. doi:10.1890/13-1486.1.

Eliazar, Iddo, and Joseph Klafter. 2005. "Lévy, Ornstein-Uhlenbeck, and subordination: Spectral vs. Jump description." *Journal of Statistical Physics* 119 (1-2): 165–96. doi:10.1007/s10955-004-2710-9.

Finley, Andrew O., Sudipto Banerjee, Patrik Waldmann, and Tore Ericsson. 2009. "Hierarchical Spatial Modeling of Additive and Dominance Genetic Variance for Large Spatial Trial Datasets." *Biometrics* 65 (2): 441–51. doi:10.1111/j.1541-0420.2008.01115.x.

Fuglstad, Geir-Arne, Daniel Simpson, Finn Lindgren, and Håvard Rue. 2017. "Constructing Priors that Penalize the Complexity of Gaussian Random Fields." *ArXiv E-Prints*. arxiv:1503.00256.

"Global Population Dynamics Database (Version 2)." 2010. http://www.sw.ic.ac.uk/cpb/cpb/gpdd.html.

Grigelionis, Bronius. 2013. *Student's t-Distribution and Related Stochastic Processes*. SpringerBriefs in Statistics. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-31146-8.

Heyde, Chris C., and N. N. Leonenko. 2005. "Student processes." *Advances in Applied Probability* 37 (2): 342–65. doi:10.1239/aap/1118858629.

Shah, Amar, Andrew Gordon Wilson, and Zoubin Ghahramani. 2014. "Student-t Processes as Alternatives to Gaussian Processes." *ArXiv E-Prints*. arxiv:1402.4306.

Yu, Shipeng, Volker Tresp, and Kai Yu. 2007. "Robust Multi-Task Learning with t-Processes." *International Conference on Machine Learning*, 1103–10. doi:10.1145/1273496.1273635.