

Spatial Modeling in Stan

Mitzi Morris

2024-09-09

Stan Development Team



Overview

Workshop Goal

Learn how to use existing models and develop new ones.

Outline

- About Stan
- Stan Workflow
- ICAR Model for Spatial Smoothing
- BYM2 Model for Spatial and Intrinsic Random Effects
- Extensions to the BYM2 Model

What is Stan?

- A Probabilistic Programming Language
 - A Stan program defines a probability model
- A Collection of Inference Algorithms
 - Exact Bayesian inference using MCMC (NUTS-HMC)
 - Aproximate Bayesian inference using ADVI, Pathfinder
 - Point estimates using optimization, Laplace approximation
- Interfaces and Analysis Tools
 - R: [CmdStanR](#), [Posterior](#), [Bayesplot](#), [LOO](#) (Also: [RStan](#), [RStanARM](#), [BRMS](#))
 - Python: [CmdStanPy](#), [ArviZ](#)
 - Julia: [Stan.jl](#), [ArviZ.jl](#)
 - Command shell: [CmdStan](#)

Why Stan?

- **Goal:** Develop multi-level models for real-world applications
- *Problem:* Need descriptive power, clarity of BUGS
- *Solution:* Compile a domain-specific language

Why Stan?

- **Goal:** Develop multi-level models for real-world applications
- *Problem:* Need descriptive power, clarity of BUGS
- *Solution:* Compile a domain-specific language
- *Problem:* Pure directed graphical language inflexible
- *Solution:* Imperative probabilistic programming language

Why Stan?

- **Goal:** Develop multi-level models for real-world applications
- *Problem:* Need descriptive power, clarity of BUGS
- *Solution:* Compile a domain-specific language
- *Problem:* Pure directed graphical language inflexible
- *Solution:* Imperative probabilistic programming language
- *Problem:* Heterogeneous user base
- *Solution:* Many interfaces (R, Python, Julia, Mathematica, MATLAB), domain-specific examples, case studies, and field guides

Why Stan?

- **Goal:** Develop multi-level models for real-world applications
- *Problem:* Need descriptive power, clarity of BUGS
- *Solution:* Compile a domain-specific language
- *Problem:* Pure directed graphical language inflexible
- *Solution:* Imperative probabilistic programming language
- *Problem:* Heterogeneous user base
- *Solution:* Many interfaces (R, Python, Julia, Mathematica, MATLAB), domain-specific examples, case studies, and field guides
- *Problem:* Restrictive licensing limits use
- *Solution:* Code and doc open source (BSD, CC-BY)

Why Stan?

- **Goal:** Robust, fully Bayesian inference.
- *Problem:* Gibbs and Metropolis too slow (diffusive)
- *Solution:* Hamiltonian Monte Carlo (flow of Hamiltonian dynamics)

Why Stan?

- **Goal:** Robust, fully Bayesian inference.
- *Problem:* Gibbs and Metropolis too slow (diffusive)
- *Solution:* Hamiltonian Monte Carlo (flow of Hamiltonian dynamics)
- *Problem:* Need to tune parameters for HMC
- *Solution:* Tune step size and estimate mass matrix during warmup; determine number of steps on-the-fly (NUTS)

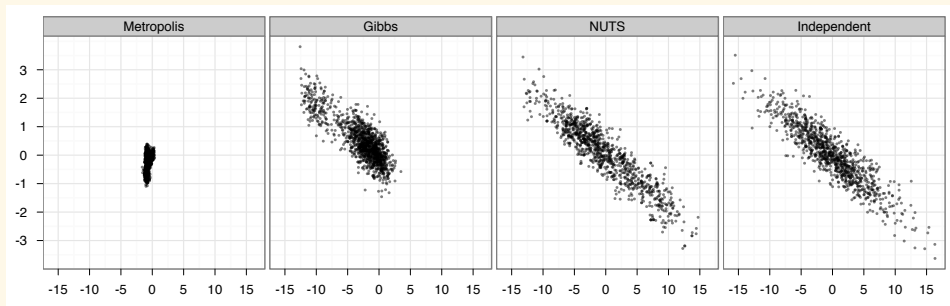
Why Stan?

- **Goal:** Robust, fully Bayesian inference.
- *Problem:* Gibbs and Metropolis too slow (diffusive)
- *Solution:* Hamiltonian Monte Carlo (flow of Hamiltonian dynamics)
- *Problem:* Need to tune parameters for HMC
- *Solution:* Tune step size and estimate mass matrix during warmup; determine number of steps on-the-fly (NUTS)
- *Problem:* Need gradients of log posterior for HMC
- *Solution:* Reverse-mode algorithmic differentiation

Why Stan?

- **Goal:** Robust, fully Bayesian inference.
- *Problem:* Gibbs and Metropolis too slow (diffusive)
- *Solution:* Hamiltonian Monte Carlo (flow of Hamiltonian dynamics)
- *Problem:* Need to tune parameters for HMC
- *Solution:* Tune step size and estimate mass matrix during warmup; determine number of steps on-the-fly (NUTS)
- *Problem:* Need gradients of log posterior for HMC
- *Solution:* Reverse-mode algorithmic differentiation
- *Problem:* Need unconstrained parameters for HMC
- *Solution:* Variable transforms w. Jacobian determinants

NUTS-HMC vs. Gibbs and Metropolis



- Plot shows 2 dimensions of highly correlated 250-dim normal
- **1,000,000 draws** from Metropolis and Gibbs (thin to 1000)
- **1000 draws** from NUTS (no thinning); 1000 independent draws
- *The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo*, fig. 7

MCMC Sampling Efficiency

- MCMC samples have autocorrelation
- N_{eff} is the number of independent samples with the same estimation power as the N autocorrelated samples.
- When comparing algorithms, compare N_{eff} **per second**
- Gibbs and Metropolis have
 - High iterations per second but low effective samples per iteration
 - Both are weak when there is high correlation among the parameters in the posterior
- Hamiltonian Monte Carlo & NUTS
 - Fewer iterations per second than Gibbs or Metropolis
 - But *much* higher N_{eff} per second
 - More robust to highly correlated or differently scaled parameters.

Other Supported Inference Algorithms

Faster, more scalable methods

- Variational Inference

Generate samples from a variational approximation to the posterior distribution

- ADVI - use stochastic gradient ascent
- Pathfinder - follow quasi-Newton optimization path

- Optimization: find posterior mode

- MLE - maximum likelihood estimate (of the data given the parameters)
- MAP - maximum a posteriori estimate (of the value of the posterior density on the unconstrained scale)
- Laplace approximation - obtain samples from posterior mode

Stan Toolset

- Programming language interfaces: CmdStanR, CmdStanPy, Stan.jl, (MatLabStan)
 - Core methods for model compilation, doing inference, accessing results
- Tools for visualization and validation: Bayesplot, LOO, ArviZ, Arviz.jl
 - Evaluate the goodness of fit; make plots to communicate findings
- Higher Level Interface for R: BRMS - Bayesian Regression Models in Stan
 - Model specification via extended R formula, data supplied as R dataframe
 - BRMS translates formula to Stan, creates list of inputs from dataframe, runs NUTS-HMC sampler, checks inferences.
- BridgeStan - algorithm development and exploration
 - Evaluate the log posterior density at a point
 - Constrain/unconstrain model parameters

The Stan Language

- A Stan program
 - Declares data and (constrained) parameter variables
 - Defines log posterior (or penalized likelihood)
 - Computes quantities of interest
- Syntax
 - Influenced by BUGS (plus Java/C++ punctuation)
 - Explicit variable types (like Java/C++, not like Python, R)
 - Named program blocks - distinguish between data and parameters (not like BUGS)
 - Control flow: `if` statements - dynamic branch points (more powerful than BUGS)
- Mathematical operations
 - Stan language includes a very large set of probability distributions and mathematical functions from Stan's math library
 - Efficient, vectorized (almost always)

Stan Example: Laplace's Model of Birth Rate by Sex

Laplace's data on live births in Paris from 1745–1770:

<i>sex</i>	<i>live births</i>
female	241 945
male	251 527

- *Question 1* (Estimation): What is the birth rate of boys vs. girls?
- *Question 2* (Event Probability): Is a boy more likely to be born than a girl?

Laplace computed this analytically. Let's use Stan's NUTS-HMC sampler instead.

Laplace's Model in Stan

```
transformed data {  
  int male = 251527;  
  int female = 241945;  
}  
parameters {  
  real<lower=0, upper=1> theta;  
}  
model {  
  male ~ binomial(male + female, theta);  
}  
generated quantities {  
  int<lower=0, upper=1> theta_gt_half = (theta > 0.5);  
}
```

Laplace's Answer

```
births_model = cmdstan_model("laplace.stan") # compile model
births_fit = births_model$sample()           # run inference algorithm
as.data.frame(births_fit$summary())          # manage results
```

variable	mean	median	sd	q5	q95
theta	0.51	0.51	0.000725	0.509	0.511
theta_gt_half	1.00	1.00	0.000000	1.000	1.000

- *Question 1* (Estimation): What is the birth rate of boys vs. girls?
 θ is 90% certain to lie in (0.509, 0.511)
- *Question 2* (Event Probability): Is a boy more likely to be born than a girl?
Laplace “morally certain” boys more prevalent

Stan Program File

A Stan program consists of one or more named program blocks, strictly ordered

```
functions {  
  // declare, define functions  
} data {  
  // declare input data  
} transformed data {  
  // transform inputs, define program data  
} parameters {  
  // declare (continuous) parameters  
} transformed parameters {  
  // define derived parameters  
} model {  
  // compute the log joint distribution  
} generated quantities {  
  // define quantities of interest  
}
```

Stan Program Blocks - Execution During Sampling

- data, transformed data blocks - executed once on startup
- parameters -
 - on startup: initialize parameters
 - at every step of inference algorithm: validate constraints
- transformed parameters, model blocks - executed every *step* of the sampler
- generated quantities - executed every *iteration* of the sampler
- After every sampler iteration, program outputs current values of all variables in parameters, transformed parameters, and generated quantities blocks.

The Stan Language

- Variables have strong, static types
 - Strong: only values of that type will be assignable to the variable, (type promotion allowed, e.g., `int` to `real`, `complex`)
 - Static: variable type is constant throughout program
 - Types: `vector`, `row_vector`, `matrix`, `complex`, `array`, and `tuple`
 - Variables can be declared with constraints on `upper` and/or `lower` bounds
- Motivation
 - Static types make programs easier to comprehend, debug, and maintain
 - Programming errors can be detected at compile-time
 - Constrained types catch runtime errors

```
int<lower=0> N, N_time;           // constraint applies to both N, N_time
vector[3]<lower=0> sigma;
array[3] matrix[M, N] some_xs;
array[N] int<lower=0, upper=N_time> time; // use arrays for structured int data
```

The Stan Language

The `model` block

- defines the joint log probability density function given parameters
- this function is the sum of all distribution and log probability increment statements in the model block

Distribution statements

```
y ~ normal(mu, sigma);  
mu ~ normal(0, 10);  
sigma ~ normal(0, 1);
```

Log probability increment statements

```
target += normal_lupdf(y | mu, sigma);  
target += normal_lupdf(mu | 0, 10);  
target += normal_lupdf(sigma | 0, 1);
```

Processing Steps

- Compile model
 - Stan compiler translates Stan file to C++ file
 - C++ file is compiled to executable program, via GNU Make
 - *prerequisites: a C++17 compiler, GNU-Make*
both CmdStanPy and CmdStanR have utilities to install these
- Run inference algorithm
 - Interfaces run the compiled executable (as a subprocess) and manage the per-chain outputs (modified CSV format files)
- Get results via interface methods
 - Parse CSV outputs into in-memory object
 - Access individual parameters and quantities of interest
 - Run summary and diagnostic reports

Install Stan

- If you don't already have Stan running on your machine, 10 minutes to get it working now.
 - *If you already have Stan running on your machine, please help your neighbor.*
- See handout: `h1_install_stan.html`

Notebook: Spatial Data for Python and R

Dataset taken from [Bayesian Hierarchical Spatial Models: Implementing the Besag York Mollié Model in Stan](#).

- Aggregated counts of car vs. child traffic accidents, localized to US Census tract.

Per-tract data includes:

- raw counts of accidents, population
 - measures of foot traffic, car traffic
 - socio-economic indicators: median income, neighborhood transiency
- Handouts
 - `h2_spatial_data.html`
 - R version `h2_spatial_data.Rmd`
 - Python version: `h2_spatial_data.ipynb`

Stan Model Building Workflow

When writing a Stan model, as when writing any other computer program,
the fastest way to success is to go slowly.

- Incremental development
 - Write a (simple) model
 - Fit the model to data (either simulated or observed)
 - Check the fit
- Then modify (*stepwise*) and repeat
- Compare successive models

Base Model: Poisson Regression

Data

- N - the number of census tracts
- y - the array of observed outcomes - accidents per tract
- E - the population per tract (“exposure”)
- K - the number of predictors
- xs - the $N \times K$ data matrix of predictors

Regression Parameters

- β_0 - global intercept
- β - a K -length vector of regression co-efficients

Base Model: Poisson Regression

Distribution statement (likelihood)

```
y ~ poisson_log(log(E) + beta0 + X * betas);
```

- `poisson_log` distribution uses the log rate as a parameter
 - don't need to exponentiate, better numerical stability
- `poisson_log_rng` pseudo-random number generator function (PRNG function)
 - **PRNG functions** used to replicate, simulate data.
 - Poisson variate is an **integer**, largest integer value is 2^{30} , argument to `poisson_log_rng` must be less than $30 \log 2$, ≈ 28

Base Stan Program

```
data {  
  int<lower=0> N;  
  array[N] int<lower=0> y; // count outcomes  
  vector<lower=0>[N] E; // exposure  
  int<lower=1> K; // num covariates  
  matrix[N, K] xs; // design matrix  
}  
transformed data {  
  vector[N] log_E = log(E);  
}  
parameters {  
  real beta0; // intercept  
  vector[K] betas; // covariates  
}  
model {  
  y ~ poisson_log(log_E + beta0 + xs * betas); // likelihood  
  beta0 ~ std_normal(); betas ~ std_normal(); // priors  
}
```

Posterior Predictive Checks

- The posterior predictive distribution is the distribution over new observations given previous observations.
- In the absence of new observations, we can simulate new observations, `y_rep` in the `generated quantities` block.

y^{rep} of the original data set y given model parameters θ is defined by

$$p(y^{\text{rep}} \mid y) = \int p(y^{\text{rep}} \mid \theta) \cdot p(\theta \mid y) \mathrm{d}\theta.$$

Base Stan Program - Generated Quantities Block

```
generated quantities {  
  array[N] int y_rep;  
  vector[N] log_lik;  
  { // local block variables not recorded  
    vector[N] eta = log_E + beta0 + xs * betas;  
    if (max(eta) > 26) { // avoid overflow in poisson_log_rng  
      print("max eta too big: ", max(eta));  
      for (n in 1:N) {  
        y_rep[n] = -1; log_lik[n] = -1;  
      }  
    } else {  
      for (n in 1:N) {  
        y_rep[n] = poisson_log_rng(eta[n]);  
        log_lik[n] = poisson_log_lpmf(y[n] | eta[n]);  
      }  
    }  
  }  
}
```


Notebook: Stan Model Building Workflow

- Start with base model
- Best practice one: mean-center, scale predictor data
- Refine model: add a random effects component
- Check fits, compare models
- Handouts
 - `h3_stan_workflow.html`
 - R version `h3_stan_workflow.Rmd`
 - Python version: `h3_stan_workflow.ipynb`

Spatial Smoothing For Areal Data

Study goal: explain *rate* of traffic accidents (count / kid_pop)

- Counts of rare events in small-population regions are noisy
 - solution: pool information from *neighboring* areas
- Besag, 1973, 1974: Conditional Auto-Regressive Model (CAR), and **Intrinsic Conditional Auto-Regressive** (ICAR) model
- CAR model has parameter α for the amount of spatial dependence;
 - CAR model requires computing matrix determinants
 - cubic operation (calculated at every *step* of the sampler)
- ICAR simplification: let $\alpha == 1$ (complete spatial dependence)
 - ICAR model computes pairwise distance between neighbors
 - quadratic operation

Neighbor Relationship

- The binary neighbor relationship (written $i \sim j$ where $i \neq j$) is
 - 1 if regions n_i and n_j are neighbors
 - 0 otherwise
- Many possible definitions of “neighbor”
 - ‘rook’ - areas share a bounding line
 - ‘queen’ - areas share a boundary point
 - *transit patterns* - residents travel (frequently, directly) between areas
- For CAR models, neighbor relationship is:
 - symmetric - if $i \sim j$ then $j \sim i$
 - not reflexive - a region is not its own neighbor ($i \not\sim i$)

From Map to Graph

The neighborhood network can be represented as either a matrix or a graph.

- $N \times N$ matrix
 - Entries (i, j) and (j, i) are 1 when regions n_i and n_j are neighbors, 0 otherwise
- Undirected graph: regions are vertices, pairs of neighbors are edges
 - Encoded as a set of *edges* - 2 column matrix, each row is a pair of neighbors (n_i, n_j)
- If the number of neighbors per area is relatively small compared to the number of areas, the adjacency matrix will be sparse
- Computing with an edgeset is more efficient
 - data structure requires less storage
 - adjacent nodes in-memory adjacent

Intrinsic Conditional Auto-Regressive (ICAR) Model

- Conditional specification: multivariate normal random vector ϕ where each ϕ_i is conditional on the values of its neighbors
- Joint specification rewrites to *Pairwise Difference*:

$$p(\phi) \propto \exp \left\{ -\frac{1}{2} \sum_{i \sim j} (\phi_i - \phi_j)^2 \right\}$$

- centered at 0, assuming common variance for all elements of ϕ .
- Each $(\phi_i - \phi_j)^2$ contributes a penalty term based on the distance between the values of neighboring regions
- ϕ is non-identifiable, constant added to ϕ washes out of $\phi_i - \phi_j$
 - sum-to-zero constraint centers ϕ

Stan ICAR Model

$$p(\phi) \propto \exp \left\{ -\frac{1}{2} \sum_{i \sim j} (\phi_i - \phi_j)^2 \right\}$$

Use Stan's vectorized operations to compute log probability density

```
target += -0.5 * dot_self(phi[node1] - phi[node2]);
```

Encode neighbor information as graph edgeset, i.e. pairs of indices for neighbors i, j :

```
int<lower = 0> N; // number of areal regions  
int<lower = 0> N_edges; // number of neighbor pairs  
array[2, N_edges] int<lower = 1, upper = N> neighbors; // columnwise adjacent
```

Stan ICAR Model Implementation

```
functions {  
  real standard_icar_lpdf(vector phi, array[ , ] int adjacency, real epsilon) {  
    return -0.5 * dot_self(phi[adjacency[1]] - phi[adjacency[2]])  
      + normal_lupdf(sum(phi) | 0, epsilon * rows(phi));  
  }  
}  
  
data {  
  int<lower=0> N;  
  int<lower=0> N_edges;  
  array[2, N_edges] int<lower=1, upper=N> neighbors;  
}  
  
parameters {  
  vector[N] phi;  
}  
  
model {  
  phi ~ standard_icar(neighbors, 0.001);  
}
```

Notebook: ICAR model

- Add a spatial random effects component to the base model.
- Check fits, compare ICAR, ordinary random effects models
- Handouts
 - `h4_icar.html`
 - R version `h4_icar.Rmd`
 - Python version: `h4_icar.ipynb`

Besag York Mollié (1991) BYM Model

- Account for both spatial and heterogenous variation
- Lognormal Poisson regression $\eta_i = \mu + x\beta + \phi + \theta$
- μ is the fixed intercept.
- x is the design matrix, β is vector of regression coefficients.
- ϕ is an ICAR spatial component
- θ is an vector of ordinary random-effects components.
- MCMC samplers require strong hyperpriors on ϕ and θ .
 - Bernardinelli et. al. (1995): hyperpriors depend on average number of neighbors, i.e., hyperpriors depend on data.

BYM2 model: Riebler et al, 2016

- Model combination of spatial and non-spatial components $\phi + \theta$ as

$$\left((\sqrt{\rho/s}) \phi^* + (\sqrt{1-\rho}) \theta^* \right) \sigma$$

where:

- $\sigma \geq 0$ is the overall standard deviation.
 - $\rho \in [0, 1]$ - proportion of spatial variance.
 - ϕ^* is the ICAR component.
 - $\theta^* \sim N(0, 1)$ is the vector of ordinary random effects
 - s is a scaling factor s.t. $\text{Var}(\phi_i) \approx 1$; s is data.
- Follows a series of improvements, notably Leroux 2000
 - single scale parameter plus mixing parameter

BYM2 model scaling factor

Besag 1974: joint specification of ϕ is multivariate normal, zero-centered

$$\phi \sim N(0, Q^{-1}).$$

- The variance of ϕ is specified as the precision matrix Q ,
 - inverse of the covariance matrix Σ , i.e. $\Sigma = Q^{-1}$
- The variance of θ is a scalar.
- To clearly interpret σ (variance of combined components), for each i ,
 $\text{Var}(\phi_i) \approx \text{Var}(\theta_i) \approx 1$.
- Scale the proportion of variance ρ contributed by ϕ

Computing ICAR scaling factor

Goal: for each i , $\text{Var}(\phi_i) \approx 1$

“the geometric mean of the average marginal variance of the areal units is 1”

1. Construct Q , the precision matrix for to the neighborhood network

convert the edgelist to an adjacency matrix

2. Add a small amount of noise to the diagonal elements of Q (positive definite)

3. Compute Q_{inv} , the covariance matrix - **computationally expensive**

4. Compute the geometric mean of the diagonal elements of Q_{inv}

$$\tau = \exp(\text{mean}(\log(x)))$$

The scaling factor comes in as data - must be computed in either Python or R

BYM2 Stan Implementation, Data Block

```
data {  
  int<lower=0> N;  
  array[N] int<lower=0> y; // count outcomes  
  vector<lower=0>[N] E; // exposure  
  int<lower=1> K; // num covariates  
  matrix[N, K] xs; // design matrix  
  
  // spatial structure  
  int<lower = 0> N_edges; // number of neighbor pairs  
  array[2, N_edges] int<lower = 1, upper = N> neighbors; // columnwise adjacent  
  
  real tau; // scaling factor  
}
```

BYM2 Stan Implementation, Parameters, Model Blocks

```
parameters {  
  real beta0; // intercept  
  vector[K] betas; // covariates  
  real<lower=0, upper=1> rho; // proportion of spatial variance  
  vector[N] phi; // spatial random effects  
  vector[N] theta; // heterogeneous random effects  
  real<lower = 0> sigma; // scale of combined effects  
}  
  
transformed parameters {  
  vector[N] gamma = (sqrt(1 - rho) * theta + sqrt(rho / tau) * phi); // BYM2  
}  
  
model {  
  y ~ poisson_log(log_E + beta0 + xs_centered * betas + gamma * sigma);  
  rho ~ beta(0.5, 0.5);  
  phi ~ standard_icar(neighbors, 0.001);  
  // std_normal() prior on all other params
```

Notebook: BYM2 Model

- Implement the BYM2 model.
- Check fits, compare to ICAR, ordinary random effects models
- Handouts
 - `h5_bym2.html`
 - R version `h5_bym2.Rmd` (or `h5_bym2.qmd`)
 - Python version: `h5_bym2.ipynb`

BYM2 for Disconnected Regions and Islands

Spatial and Spatio-temporal Epidemiology 26 (2018) 25–34



ELSEVIER

Contents lists available at [ScienceDirect](#)

Spatial and Spatio-temporal Epidemiology

journal homepage: www.elsevier.com/locate/sste



Original Research

A note on intrinsic conditional autoregressive models for disconnected graphs

Anna Freni-Sterrantino^{a,*}, Massimo Ventrucchi^b, Håvard Rue^c



Available on arXiv

- Scale each connected component of size larger than one as in BYM2 model
- Scale islands with standard Normal

Computing ICAR scaling factor for each component

- Compute cardinality of each component
- Scaling factor for singletons is 1
- Scaling factor for multi-node component is computed as before
 - Get subset of regions for that component
 - Compute `scaling_factor`
- Scale each region according to scaling factor for the component they belong to.

BYM2 Islands, Compute ICAR Function

```
/**
 * Compute ICAR, use soft-sum-to-zero constraint for identifiability
 * ...
 */
real standard_icar_lpdf(vector phi,
                        array[ , ] int adjacency,
                        array[] int singletons,
                        real epsilon) {
  if (size(adjacency) != 2)
    reject("require 2rows for adjacency array;",
          " found rows = ", size(adjacency));
  return -0.5 * dot_self(phi[adjacency[1]] - phi[adjacency[2]])
    + normal_lpdf(phi[singletons] | 0, 1)
    + normal_lpdf(sum(phi) | 0, epsilon * rows(phi));
}
```

BYM2 Islands Data Block

```
int<lower=0> N;  
array[N] int<lower=0> y; // count outcomes  
vector<lower=0>[N] E; // exposure  
int<lower=1> K; // num covariates  
matrix[N, K] xs; // design matrix  
// spatial structure  
int<lower = 0> N_edges; // number of neighbor pairs  
array[2, N_edges] int<lower = 1, upper = N> neighbors; // columnwise adjacent  
vector<lower = 0>[N] taus; // per-node scaling factor  
int<lower=0, upper=N> N_singletons;  
array[N_singletons] int<lower=0, upper=N> singletons;
```

BYM2 Transformed Parameters, Model Blocks

```
transformed parameters {  
  vector[N] gamma = (sqrt(1 - rho) * theta + sqrt(rho / taus) .* phi);  // BYM2  
}  
  
model {  
  y ~ poisson_log(log_E + beta0 + xs_centered * betas + gamma * sigma);  
  rho ~ beta(0.5, 0.5);  
  phi ~ standard_icar(neighbors, singletons, 0.001);  
  beta0 ~ std_normal();  
  betas ~ std_normal();  
  theta ~ std_normal();  
  sigma ~ std_normal();  
}
```

Notebook: BYM2_Islands Model

- Implement the BYM2 model for all of NYC.
- Handouts
 - `h6_bym2_islands.html`
 - R version `h6_bym2_islands.Rmd` (or `h6_bym2_islands.qmd`)
 - Python version: `h6_bym2_islands.ipynb`

BYM2 Model for Spatio-Temporal Smoothing

PLOS COMPUTATIONAL BIOLOGY

 OPEN ACCESS  PEER-REVIEWED

RESEARCH ARTICLE

Bayesian spatial modelling of localised SARS-CoV-2 transmission through mobility networks across England

Thomas Ward , Mitzi Morris, Andrew Gelman, Bob Carpenter, William Ferguson, Christopher Overton, Martyn Fyles

Version 2  Published: November 13, 2023 • <https://doi.org/10.1371/journal.pcbi.1011580>

- Extend the BYM2 model to include a daily-level random effect, i.e. three components:
 - Spatial effect θ_i
 - Ordinary per-region random effect ϕ_i
 - Ordinary per-region, per_day random effect $\phi_{i,t}$
 - Proportion of variance ρ is a simplex

Stan References and Tutorials

- Stan User's Guide
- Getting Started with Bayesian Statistics
- Hoffman and Gelman, 2014: The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo
- Course notes: *Software for MCMC* from Odd Kolbjørnsen, Spring 2023, Oslo Uni
www.uio.no/studier/emner/matnat/math/STK4051/v23/timeplan/lecture_12_softwareformcmc.pdf

BYM2 Model References and Tutorials

- Riebler et al., 2016: [An intuitive Bayesian spatial model for disease mapping that accounts for scaling](#)
- Freni-Sterrantino et al., 2018: [A note on intrinsic conditional autoregressive models for disconnected graphs](#)
- Morris et al., 2019: [Bayesian Hierarchical Spatial Models: Implementing the Besag York Mollié Model in Stan](#)
- Ward et. al., 2023 [Bayesian spatial modelling of localised SARS-CoV-2 transmission through mobility networks across England](#)
- Stan Case Study: [Spatial Models in Stan: Intrinsic Auto-Regressive Models for Areal Data](#)

Other Epi Models in Stan

- Grinsztajn et. al. [Bayesian workflow for disease transmission modeling in Stan](#)

HMMs in Stan

- Stan Case Study: [HMM Example](#)
- [Hidden Markov Models](#)
- [Stan User's Latent Discrete Parameters Chapter](#)

Many Thanks!

Questions and Discussion