gemlib
Inference & simulation for epidemic modeling

Alin Morariu, *Supervisors: Prof. Chris Jewell, Prof. Paul Fearnhead*

Lancaster University

Abstract

We introduce a Python-based general epidemic modeling language called **gemlib**, a suite of tools for implementing stochastic epidemic models in a Bayesian framework. Most notably, we showcase the inference tools that estimate event times (called data augmentation) and parameter values within the inference algorithm. **gemlib** adds functionality to current inference libraries such as TensorFlow Probability and BlackJax which allows for MCMC kernels to be combined seamlessly to form bespoke samplers needed to fit complex epidemic models.

Motivation and data

- HPAI resurgence from 2020

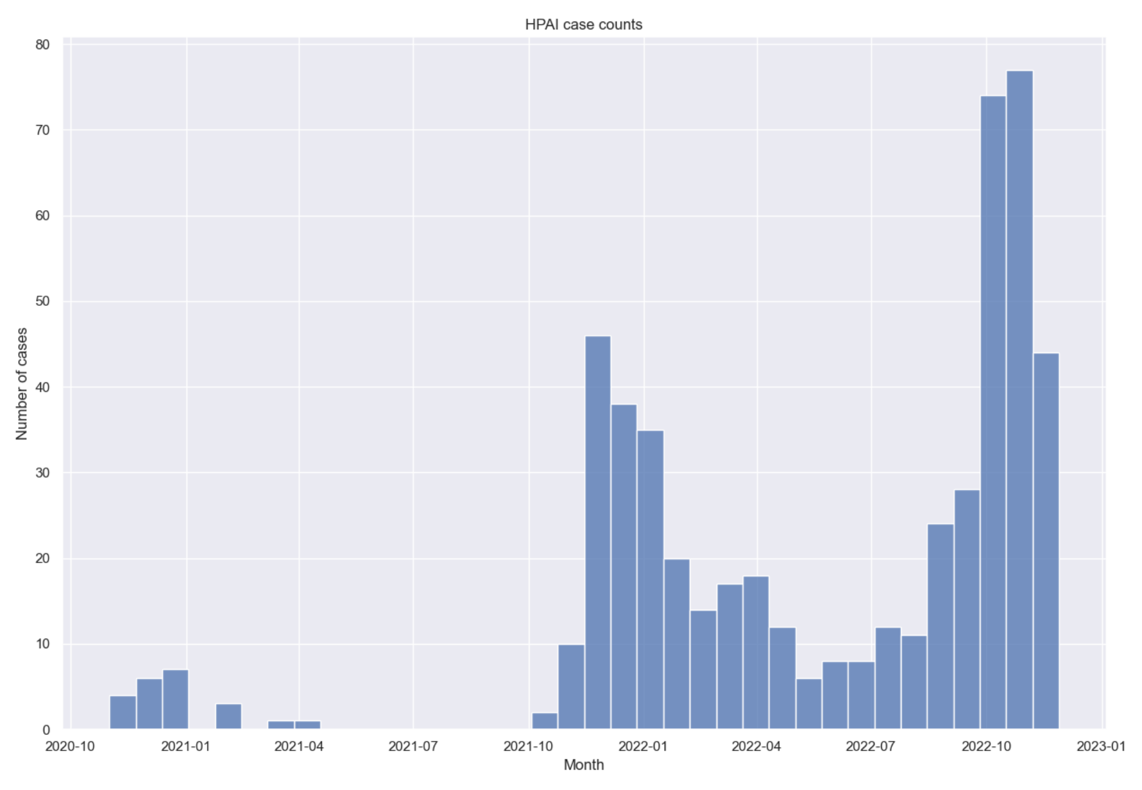

Figure 1. Case count for HPAI over the previous 3 years

- Concern over spread amongst farms (commercial and multi-bird homes)

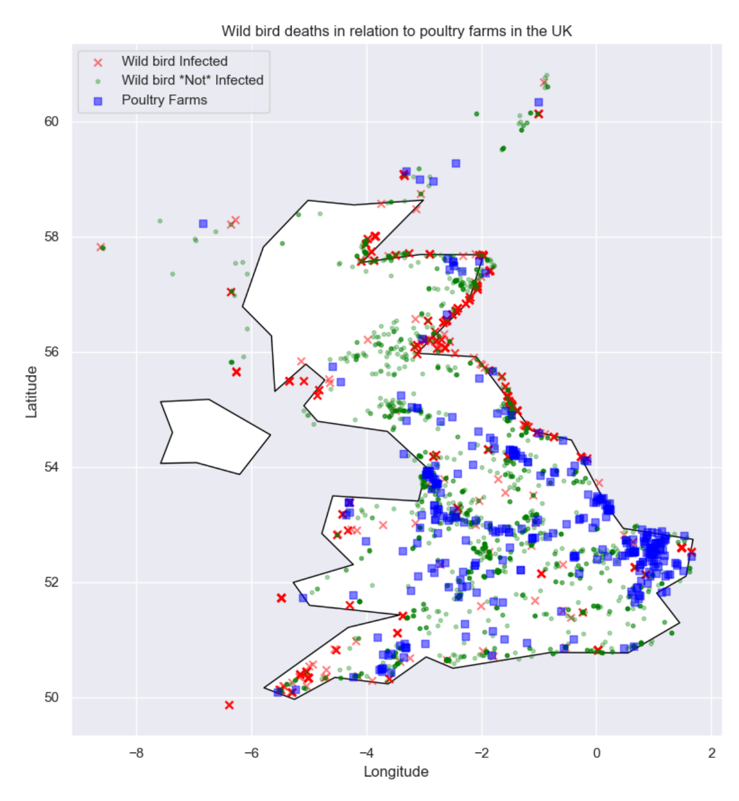

Figure 2. Map of HPAI cases across the UK overlaid with HPAI-positive wild bird deaths

Key Question

Is the driver of infection farm-to-farm or wild bird-to-farm interactions?

HPAI Stochastic epidemic model

Graphical representation

$$\begin{tikzcd} \boxed{S} \arrow[r, "h"] & \boxed{I} \arrow[r, "\kappa h"] & \boxed{N} \arrow[r, "\gamma"] & \boxed{R} \end{tikzcd}$$

Figure 3. Susceptible-Infected-Notified-Removed compartmental model, lines represent the transition events, letters above the line indicate the transmission rates

Key model features

- Continuous time, individual level model (each farm is an individual)
- Infectious pressure comes from 3 sources
  1. Spatial component
  2. Individuals' characteristics (eg. number of birds)
  3. Exogenous component (external infectious pressure from wild bird interactions)

Mathematical details

- Pairwise infectious pressure function

$$h_{i,j}(t) = \exp\left\{\omega_j^T \beta\right\} \exp\left\{-\frac{\|x_i - x_j\|^2}{\phi^2}\right\} \tag{1}$$

- Individual instantaneous infectious pressure

$$h_j(t) = \exp\left\{\rho_x(t)\right\} + \sum_{i \in \mathcal{I}_t} h_{i,j} + \nu \sum_{i \in \mathcal{N}_t} h_{i,j} \tag{2}$$

- Wild bird interaction

$$\operatorname{logit}(\rho_x(t)) = \operatorname{GP}(\rho_x, \Sigma) \tag{3}$$

$$\frac{v_x}{m_x} \sim \operatorname{Binom}(v_x, \rho_x) \tag{4}$$

- Farms stay infected for some amount of time (detection is not immediate) - assign a gamma distribution

$$\mathcal{N}_i - \mathcal{I}_i \sim \operatorname{Gamma}(3, \psi) \tag{5}$$

- Removal happens 3 days after a farm is notified

Likelihood function

Model parameters are $\theta = (\alpha, \beta, \phi, \psi, \mathcal{I})$ - the Gaussian process parameters, regression parameters, spatial decay, and infection times

$$L(\theta|\mathcal{D}) \propto \prod_{i \neq \min \mathcal{I}} \left(\sum_{j=1}^{N} h_{i,j}(t)\right) \cdot \exp\left\{-\sum_{j=1}^{N} \int_t h_j(u)du\right\} \cdot f_\Gamma(\mathcal{N}_j - \mathcal{I}_j) \tag{6}$$

MCMC scheme

Inference is difficult due to missing data

- Need to estimate model parameters **and** missing event times (eg. $\mathcal{I}$)
- Sampling algorithm must alternate between fixing event times in order to estimate transmission function parameters and fixing transmission function parameters to estimate event times
- Suggests a Metropolis-within-Gibbs sampler

Ideal inference scheme:

1. (β, ϕ, ψ) - Random Walk Metropolis-Hastings with the conditional target log-likelihood (cTLP) $\log L(\beta, \phi, \psi|\alpha, \mathcal{I}, \mathcal{D})$ (denote this kernel K_1)
2. $(\mathcal{I})$ - Data augmentation w/ cTLP $\log L(\mathcal{I}|\alpha, \beta, \phi, \psi, \mathcal{D})$ (denote this kernel K_2)
3. (α) - Hamiltonian Monte Carlo w/ cTLP $\log L(\alpha|\beta, \phi, \psi, \mathcal{I}, \mathcal{D})$ (denote this kernel K_3)

Problem: Probabilistic programming languages require users to write their own algorithm from scratch in this scenario - there is no way to combine MCMC kernels by chaining them to do this automatically. We *need* a way to compose kernels (similar to the **pipe** operator in R).

MCMC kernel in gemlib

Use a functional programming approach to implement MCMC kernels (can wrap existing kernels or write your own special kernel).

- Functions encode a series of instructions
- Functions operate on **types** instead of specific arguments therefore functions can be chained (so long as the output **type** of the first function matches the input **type** of the second)
- MCMC kernels are *state monads*. They change an overall state type and output the same chain type and some other information (eg. accept/reject

$$K :: \texttt{ChainState} \rightarrow (\texttt{ChainState}, \texttt{Info}) \tag{7}$$

Chained MCMC kernels

Chaining kernels involves the management of the global state θ and targeting subsets of it with specific kernels (Steps 1, 2, 3 in Section 1)

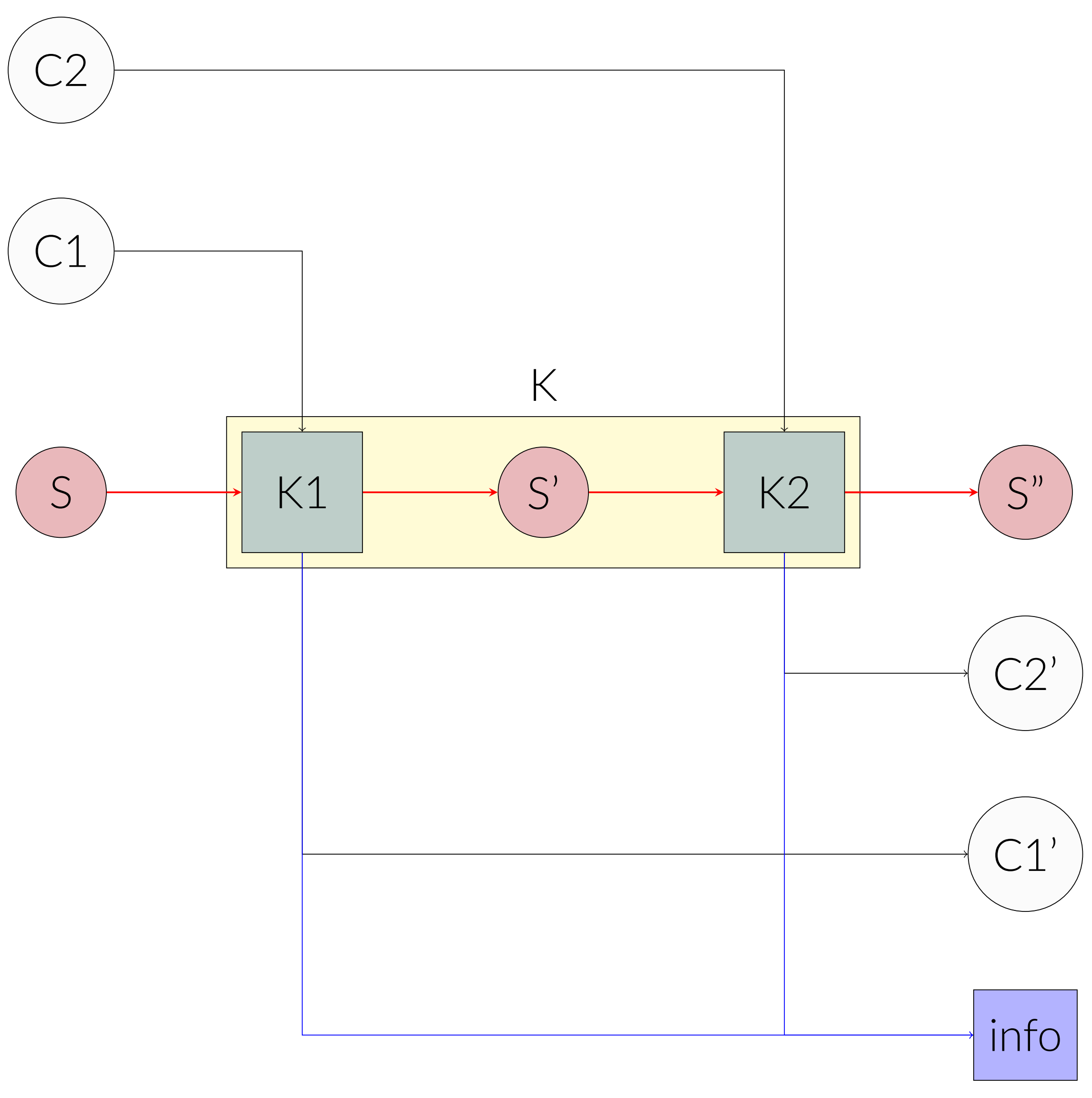

Figure 4. Computational flow of chaining 2 MCMC kernels together. S represents the state, K's are the kernels, C's represent the kernel configurations/parameters (eg. proposal scale for an RWMH)

Python code

```
import gemlib.distributions as gld
# access to discrete time and ODE too
CTSM = ContinuousTimeStateTransitionModel
incidence_matrix = np.array(
    [ #  SE  EI  IR
        [-1,  0,  0],  # S
        [ 1, -1,  0],  # E
        [ 0,  1, -1],  # I
        [ 0,  0,  1],  # R
    ],
    dtype=DTYPE)
def create_transition_rate_fn(parameters):
    def transition_rate_fn(t, state):
        ...
        return se_rate, ei_rate, ir_rate
    return transition_rate_fn
seir_model = CTSM(
    create_transition_rate_fn(),
    incidence_matrix,
    initial_state,
    num_steps = 200)
#-------------------------------------------#
param_mh_kernel = MwgStep(
    sampling_algorithm = adaptive_rwmh(),
    target_names=["beta",
                  "alpha"])
se_move_kernel = MwgStep(
    sampling_algorithm=move_events(
        incidence_matrix=incidence_matrix,
        transition_index=0,
        num_units=1,
        delta_max=5,
        count_max=1,),
    target_names=["seir"],
    kernel_kwargs_fn=lambda _:
        {"initial_conditions": initial_state}
        )
kernel = param_mh_kernel >> multi_scan(10,
                                se_move_kernel)
#-------------------------------------------#
@tf.function(jit_compile=True)
mcmc(sampling_algorithm=kernel,
    target_density_fn=seir_model.log_prob,
    initial_position=initial_position,
    num_samples = 10,000,
    seed = [0,0])
```