Estimating constant and time-varying parameters in dynamical systems via state-parameter augmentation and adaptive PMCMC

Sibo Wang (uid: sibo)

***Abstract*— This project aims to explore system identification. Specifically, the project focuses on following and reconstructing the results obtained by Dureau et al in [1]. The parameters of the SEIR disease model is inferred via a combination of system-state augmentation and adaptive particle Markov chain Monte Carlo (PMCMC).**

I. BACKGROUND

A. System identification overview

System identification (ID) refers to learning the parameters of a dynamical system. A dynamical system is modeled as

$$X_{k+1} = \phi(X_k, \theta) + \xi \tag{1}$$

$$Y_k = h(X_k, \psi) + \eta \tag{2}$$

where $\phi(X_k, \theta)$ is the dynamics model, $h(X_k, \psi)$ is the measurement model, and ξ, η are noises. This project focuses on learning the dynamics model parameters, θ. The inference for measurement model parameters, ψ, follow similar derivations. To learn θ, the posterior distribution to be inferred follows Bayes rule to be,

$$P(\theta \mid \mathcal{Y}_T) = \frac{P(\mathcal{Y}_T \mid \theta)P(\theta)}{P(\mathcal{Y}_T)} \tag{3}$$

where $\mathcal{Y}_T$ is the set of all measurement observations $\mathcal{Y}_T = \{Y_1, Y_2, \ldots, Y_T\}$.

Note that besides formally estimating the above posterior, a common workaround is to model the system parameters as random walks with small process noise. The above system model then becomes

$$X_{k+1} = \phi(X_k, \theta) + \xi \tag{4}$$

$$\theta_{k+1} = \theta_k + \tau \tag{5}$$

$$Y_k = h(X_k, \theta) + \eta \tag{6}$$

where θ can be viewed as an augmented state to X and a standard filter can be directly applied to update both X and θ simultaneously. Note that in practice, the parameter augmentation method often suffer from stability and convergence issues.

Nevertheless, this method is particularly useful when trying to learn time-varying parameters. A time-varying parameter modeled this way would have its posterior trajectories distribution available through filters. Meanwhile, the initial condition θ_0 at $t = 0$ and the random walk variance for τ become the "hyper-parameters" for the time-varying θ and are to be learned by alternative methods. More details tailored to the specific system of interest will be discussed in Proposed Methods.

More formally, since the normalizing constant $P(\mathcal{Y}_T)$ in (3) is unknown, it is natural to used Markov chain Monte Carlo (MCMC) to sample $P(\theta \mid \mathcal{Y}_T)$ by sampling θ from a proposal distribution and evaluating $P(\mathcal{Y}_T \mid \theta)P(\theta)$ to find the acceptance ratio.

However, the likelihood $P(\mathcal{Y}_T \mid \theta)$ is unknown and thus needs to be computed via marginalization,

$$P(\mathcal{Y}_T \mid \theta) = \int P(\mathcal{Y}_T \mid \mathcal{X}_n)P(\mathcal{X}_T \mid \theta)d\mathcal{X}_T \tag{7}$$

where $\mathcal{X}_T$ is the set of all system states $\mathcal{X}_T = \{X_1, X_2, \ldots, X_T\}$. The above integral is high-dimensional as it is over all of the states. Thus, a recursive method is derived and can be evaluated via filtering. For linear systems, Gaussian filters may suffice for estimating the recursive updates. However, for non-linear and high-dimensional models, particle filter is more suitable.

The combined algorithm of outsourcing the computation of $P(\mathcal{Y}_T \mid \theta)$ in MCMC to particle filter is thus called particle Markov chain Monte Carlo (PMCMC) [2]. Note that standard techniques for MCMC, such as adaptive proposal, can still be applied to PMCMC. Again, more details on recursive likelihood and PMCMC in general will be discussed in the Proposed Methods section.

B. SEIR model and time-varying parameter

The system to be inferred is the SEIR disease transmission model outlined in [1]. The model is characterized by the following ODE:

$$\frac{dS_t}{dt} = -\beta_t S_t \frac{I_t}{N} \tag{8}$$

$$\frac{dE_t}{dt} = \beta_t S_t \frac{I_t}{N} - kE_t \tag{9}$$

$$\frac{dI_t}{dt} = kE_t - \gamma I_t \tag{10}$$

$$\frac{dR_t}{dt} = \gamma I_t \tag{11}$$

in which N is the total population, S is the susceptible, E is the exposed, I is the infected, and R is the removed or immunized. β, k, γ are system parameters. In

particular, k, γ can often be approximated accurately via epidemic models [3], while β, the effective contact rate, is greatly influenced by policies, social cycles, climate variations, and other time-variant external factors [1].

Many methods ([4], [5], [6]) have been proposed to estimate the trajectory of β. Overall, Dureau et al has argued that adaptive PMCMC coupled with system-parameter augmentation provides the more holistic posterior [1]. PMCMC allows the simultaneous sampling and estimation of β with other parameters, such as initial populations N_0, I_0, etc.

More generally, PMCMC has been proposed to overcome the weakness of MCMC in time-series analyses [7]. PMCMC has a wide range of application in the biomedical/health science field [2] and in applied science overall. This project seeks to be an rudimentary exploration for system ID, estimating both constant parameters and time-varying parameters.

II. PROBLEM SETUP

A. Dynamics model

The basic dynamics model is presented by (8-11). Then, β, the time-varying parameter, is modeled as a Wiener Gaussian random walk process and augmented to the system states. Meanwhile, it is argued that for large-scale epidemics, random effects in transmission processes are considered to be well-approximated and thus deterministic ([8]). Therefore, the overall dynamics model of the system is given by

$$\frac{dS_t}{dt} = -\beta_t S_t \frac{I_t}{N} \tag{12}$$

$$\frac{dE_t}{dt} = \beta_t S_t \frac{I_t}{N} - kE_t \tag{13}$$

$$\frac{dI_t}{dt} = kE_t - \gamma I_t \tag{14}$$

$$\frac{dR_t}{dt} = \gamma I_t \tag{15}$$

$$\frac{dc_t}{dt} = \sigma dW \tag{16}$$

$$\beta = \exp(c_t) \tag{17}$$

where $c = \log(\beta)$ is the only stochastic process in the model. c is modeled as a Wiener Gaussian random walk with variance σ^2. Note that in order to filter, the above system is discretized via Euler-Maruyama. In the discretization, $c_t = c_{t-1} + \sigma\sqrt{dt}\mathcal{N}(0,1)$, as discussed in Lecture 8 in lecture notes [9].

The goal is to infer the posterior distributions of all the dynamics parameters $\theta = \{S_0, E_0, I_0, R_0, c_0, \sigma, k, \gamma\}$. The posterior trajectory of β_T and potentially other system states are also of interests.

B. Measurement model

In the original paper to be studied [1], the data obtained are noisy observations of weekly new cases of the epidemic. Unfortunately, due to time limitations, the reconstruction of the measurement model has failed. Specifically, the reconstruction required the incorporation of $\int_{week} kE_t dt$ as a state in the dynamics update, which has raised difficulties.

Instead, since the original paper has stated that I_k is also a common source of data [1], simulated weekly I_k with log-normal noise (same as the proposed noise in original paper) are generated and used as reference data. The measurement model then follows to be

$$\log(y_k) = \log(I_k) + \eta, \quad \eta \sim \mathcal{N}(0, \tau^2) \tag{18}$$

where more details specific for the reference system will be discussed next. For now, the measurement model is. Note that since the focus of this project is on inferring system parameters, the change in the state been measured should not effect the overall study.

C. Reference system

A reference system with $t = [0, 20]$ is generated based on the below initial conditions and model parameters. The system is solved using `scipy.integrate.solve_ivp` in Python with method `RK45` and time step $dt = 0.1$.

- $N = 1000, S_0 = 840, E_0 = 5, I_0 = 5, R_1 50$
- $\beta = -\frac{3}{1+e^{-0.5(t-10)}} + 3.5$
- $k = 3, \gamma = 1.5$

Assuming t is measured in weeks, measurement data y_j are generated at integer time steps $t = j, j \in \{1, 2, \ldots, 20\}$ by adding log-normal noise to I according to (18). Figure 1 shows the trajectories of the reference system.

III. PROPOSED METHODS

A. Parameter-state augmentation

As discussed before, the state-parameter augmentation method can be used when trying to learn time-varying parameters. For this particular problem, c_0 and τ become the "hyper-parameters" for $\beta = \exp(c_t)$ and thus need to be learned. These two hyper-parameters are learned alongside other parameters via adaptive PMCMC.

B. PMCMC

PMCMC, firstly introduced in [7], is particularly suitable when inferring parameters in non-linear and/or high-dimensional systems. In general, PMCMC outsources the computation of $P(\mathcal{Y}_T \mid \theta)$ to sequential Monte Carlo (SMC) via particle filter, which will be discussed in the next subsection.

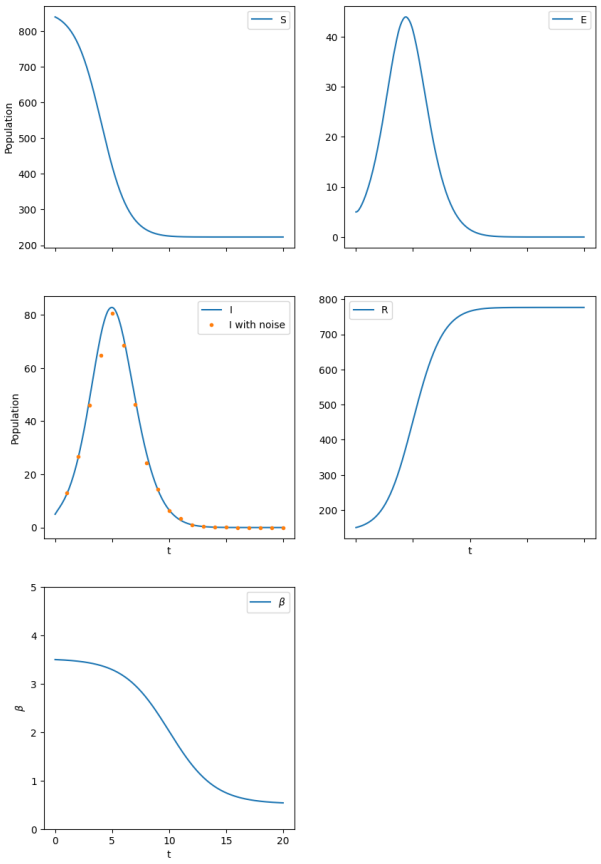


Fig. 1. Trajectories of reference system with measured data

Then, with known $P(\mathcal{Y}_T \mid \theta)$, (3) can be sampled via Metropolis-Hastings Markov chain Monte Carlo (MH-MCMC). To emphasize, the posterior distribution to be sampled from is

$$P(\theta \mid \mathcal{Y}_T) \propto P(\mathcal{Y}_T \mid \theta)P(\theta) \tag{19}$$

which leads to the following MH acceptance probability,

$$a((\mathcal{X}_T, \theta), (\mathcal{X}_T^*, \theta^*)) = \frac{P(\mathcal{Y}_T \mid \theta^*)P(\theta^*)\pi(\theta)}{P(\mathcal{Y}_T \mid \theta)P(\theta)\pi(\theta^*)} \tag{20}$$

where $\theta^* \sim \pi$ are the proposed parameters, $\mathcal{X}_T^*$ is a sample of the system states sampled from the particle filter. $P(\mathcal{Y}_T \mid \theta^*)$ is also obtained from the particle filter. $P(\theta^*)$ is the prior distribution of θ. $\pi(\theta)$ and $\pi(\theta^*)$ are equal and cancel with each other when a Gaussian random walk proposal for θ is used.

Adaptive PMCMC then refers to using adaptive covariance for random walk proposal π [1]. More details for adaptive Metropolis-Hastings can be found in lecture notes [9].

C. Particle filter (SMC)

For a dynamical system with process noise, $P(\mathcal{Y}_T \mid \theta)$ can be approximated using filters. For a particle filter, an empirical distribution of the system states are updated at each time step. The states $x^{(i)}$ are propagated based on the dynamics model (1). With the bootstrap (dynamical) proposal, at $t = k$ when there is data, the particle weights $w_k^{(i)}$ are updated simply by the likelihood,

$$w_k^{(i)} = P(y_k \mid x_k^{(i)}, \theta)w_{k-1}^{(i)} \tag{21}$$

where $P(y_k \mid x_k^{(i)}, \theta)$ is given based on (2). Since the empirical distribution of $P(X_k \mid \theta, \mathcal{Y}_{k-1})$ is maintained, the marginal likelihood for each y_k can be found based on marginalization and Monte Carlo,

$$P(y_k \mid \theta) = \int P(y_k \mid X_k, \theta)P(X_k \mid \theta, \mathcal{Y}_{k-1})dX_k \tag{22}$$

$$P(y_k \mid \theta) \approx \frac{1}{N}\sum_{i=1}^{N} P(y_k \mid x_k^{(i)}, \theta) \tag{23}$$

Then, by utilizing the Markov property of the system, the overall marginal likelihood can be approximated as

$$P(\mathcal{Y}_T \mid \theta) = \prod_{k=1}^{T} P(y_k \mid \theta) \tag{24}$$

$$P(\mathcal{Y}_T \mid \theta) \approx \prod_{k=1}^{T} \left[\frac{1}{N}\sum_{i=1}^{N} P(y_k \mid x_k^{(i)}, \theta)\right] \tag{25}$$

D. Algorithm overview

Algorithm 1 illustrates how particle filter is used within PMCMC.

Algorithm 1 Particle filiter in PMCMC for SEIR

Input: N, proposed $\theta = \{S_0, E_0, I_0, R_0, c_0, \sigma, k, \gamma\}$
$X_0^{(i)} = \{S_0, E_0, I_0, R_0, c_0\}$
$w_0^{(i)} = 1/N$
for $k = 1, 2, \ldots, T$ **do**
  $X_k^{(i)} = \phi(X_{k-1}^{(i)}, \theta = \{\sigma, k, \gamma\})$
  if $t_k \mod 1 = 0$ **then**
    Compute $P(y_k \mid x_k^{(i)}, \theta)$ based on $h(X_k^{(i)})$
    Add $\frac{1}{N}\sum P(y_k \mid x_k^{(i)}, \theta)$ to $P(\mathcal{Y}_T \mid \theta)$
    $w_k^{(i)} = P(y_k \mid x_k^{(i)}, \theta)w_{k-1}^{(i)}$
    Normalize weights
  end if
  Compute effective sample size ESS
  Re-sample if ESS < 0.5
end for
Return $P(\mathcal{Y}_T \mid \theta)$ and a sample from $\mathcal{X}_T^{(1,\ldots,N)}$

Note that k in $\phi(\cdot)$ is the parameter for SEIR, while other k refers to time steps. Also, to overcome general computational instability, all probabilities are computed in their log forms whenever possible. Moreover, because of the filtering of Wiener process, at some $t = k$, if

$\min\Big(P(y_k \mid x_k^{(i)}, \theta)\Big) < -400$, which would cause computationally instability because of the extremely small probability, that run of particle filter is terminated and $\log\Big(P(\mathcal{Y}_T \mid \theta)\Big)$ for that proposed θ is forced to $-1e9$. Finally, Algorithm 2 illustrates the overall adaptive PMCMC.

Algorithm 2 Adaptive PMCMC for SEIR

Input: $P(\theta)$, prior θ_0, prior RW covariance Σ_0, M
Compute $P(\theta) = P(\theta_0)$
$P(\mathcal{Y}_T \mid \theta)$, $\mathcal{X}_{T,m=0}$ from Alg. 1 based on θ_0
$\Sigma = \Sigma_0$
for $m = 1, 2, 3, \ldots, M-1$ **do**
  Propose $\theta^* = \pi(\theta_{m-1}, \Sigma)$
  Compute $P(\theta^*)$
  $P(\mathcal{Y}_T \mid \theta^*)$, $\mathcal{X}_{T,\theta^*}$ from Alg. 1 based on θ^*
  $a = \frac{P(\mathcal{Y}_T|\theta^*)P(\theta^*)}{P(\mathcal{Y}_T|\theta)P(\theta)}$
  $u \sim \mathcal{U}(0,1)$
  if $u < a$ **then**
    $P(\theta) = P(\theta^*)$
    $P(\mathcal{Y}_T \mid \theta) = P(\mathcal{Y}_T \mid \theta^*)$
    $\theta_m = \theta^*$
    $\mathcal{X}_{T,m} = \mathcal{X}_T^*$
  else
    $\theta_m = \theta_{m-1}$
    $\mathcal{X}_{T,m} = \mathcal{X}_{T,m-1}$
  end if
  Recursively adapt Σ based on $\{k_0, s_d, \epsilon\}$
end for

IV. RESULTS

A. Priors and parameter tuning

Firstly, similar to [1] and [2], informative priors are provided for $\{R_0, k, \gamma\}$, while vague ranges, reflecting lack of knowledge, are given to $\{S_0, E_0, I_0, c_0, \sigma\}$. Specifically, $P(\theta)$ is given by

$$S_0 = 1000 - E_0 - I_0 - R_0 \tag{26}$$
$$E_0 = \mathcal{U}(0, 10) \tag{27}$$
$$I_0 = \mathcal{U}(0, 10) \tag{28}$$
$$R_0 = \mathcal{N}(150, 10) \tag{29}$$
$$c_0 = \mathcal{U}(-2, 2) \tag{30}$$
$$\sigma = \mathcal{U}(0, 1) \tag{31}$$
$$k = \mathcal{N}(3, 0.5) \tag{32}$$
$$\gamma = \mathcal{N}(1.5, 0.5) \tag{33}$$

Note that because the system states represent non-negative populations, $\log\Big(P(\theta)\Big) = -1e9$ are assigned whenever any of $\{S_0, E_0, I_0, R_0\}$ are sampled to be negative.

Following similar information restriction assumptions, for θ_0, true reference values are given to $\{R_0, k, \gamma\} = \{150, 3, 1.5\}$, while others are randomly sampled based on their respective distributions in $P(\theta)$. Σ_0 for π is then assigned as

$$\Sigma_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.05 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0.01 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.005 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 & \end{bmatrix} \tag{34}$$

Section 4 of the introduction to PMCMC [7] has shown that the number of particles used in particle filter does not effect the distribution to which PMCMC is converging. Therefore, the particle filter used in PMCMC does not need to be tuned as carefully as one directly applied on itself. A typical number of particles ranges in $[100, 1000]$ ([2]). 1000 particles are used for this project given the high dimensions and stochasity in Wiener random walk for β. The number of samples for PMCMC is 5000, same as what was used in [2].

For the tuning of parameters for adaptive PMCMC, initially $k_0 = 100, s_d = 2.4^2/d = 0.72, \epsilon = 1e-8$ are assigned. Then, k_0 and s_d are hand-tuned in the general aim of lowering integrated autocorrelation (IAC) values and maintaining the acceptance ratio steadily around 0.3. After trials and errors, k_0 is increased to 500 given the high dimensions and s_d has stayed unchanged as 0.72. The diagonal terms in Σ has been observed at selected time steps and are generally above $1e-8$ by at least one magnitude, thus ϵ is also not changed. Burn-in is applied such that the first 20% of samples are burned and 70% of remaining samples are selected evenly.

B. Mixing and marginals

Figure 2 shows the mixing of the parameters. As seen, the mixing is mediocre for all eight parameters. Their sampling patterns over time look similar, with some degree of randomness, but also noticeable correlations among samples and many localized intervals. Although the patterns look far from pure white noise, given the number of dimensions and non-linearity among some of the parameters, the mixing is acceptable.

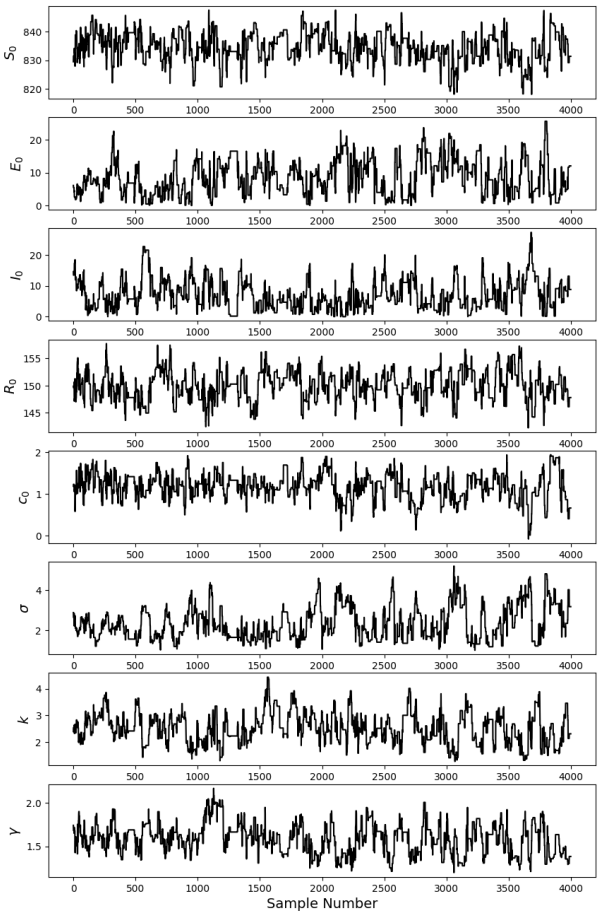


Fig. 2. Mixing of θ samples overtime for PMCMC

Figure 3 shows the 1D and 2D marginals of the sampled parameters. First of all, by looking at the 1D marginals, only $\{S_0, R_0, c_0\}$ appear to be somewhat symmetric and Gaussian. Tee marginals for other parameters are all skewed, indicating the difficulties in sampling. A few 2D marginals, such as S_0 vs. c_0 and S_0 vs. R_0, appear to be Gaussian, while others are more randomly shaped.

C. Acceptance ratio and IAC

Listed below are the accepted samples ratio for adaptive PMCMC at selected intervals. As discussed before, the hyper-parameters for adaptive MCMC are tuned such that the acceptance ratio is around 0.3. In this case, the ratio steadily dropped from 0.3 to 0.25, which is satisfactory.

- sample 500: 0.289
- sample 1000: 0.293
- sample 1500: 0.295
- sample 2000: 0.284
- sample 2500: 0.275
- sample 3000: 0.263
- sample 3500: 0.258
- sample 4000: 0.251
- sample 4500: 0.247

Shown in Figure 4 are the auto-correlations for the samples. For most of the parameters, the correlation drops to zero near when the lag is near 100. The mediocre decrease in R matches with the mixing patterns shown earlier in Figure 2. More specifically, the respective IAC values are $\{47.13, 52.35, 47.96, 37.18, 42.32, 66.07, 46.53, 56.00\}$. Back in project 2b, when estimating simple SIR model, the IAC values are around 10 for identifiable and around 100 for non-identifiable. Given the context, these IAC values, which are around 50, have shown the correlation is in an acceptable range. The true reference values, indicated by orange dot in 2D marginals and blue line in 1D marginals, have all been covered by the sampling distributions (except for c's random walk standard deviation σ, which does not have a true reference value). Moreover, except for R_0 and γ, the distribution of all other parameters have successfully recovered the true reference values very close to their MAP point, indicating the success in MCMC sampling.

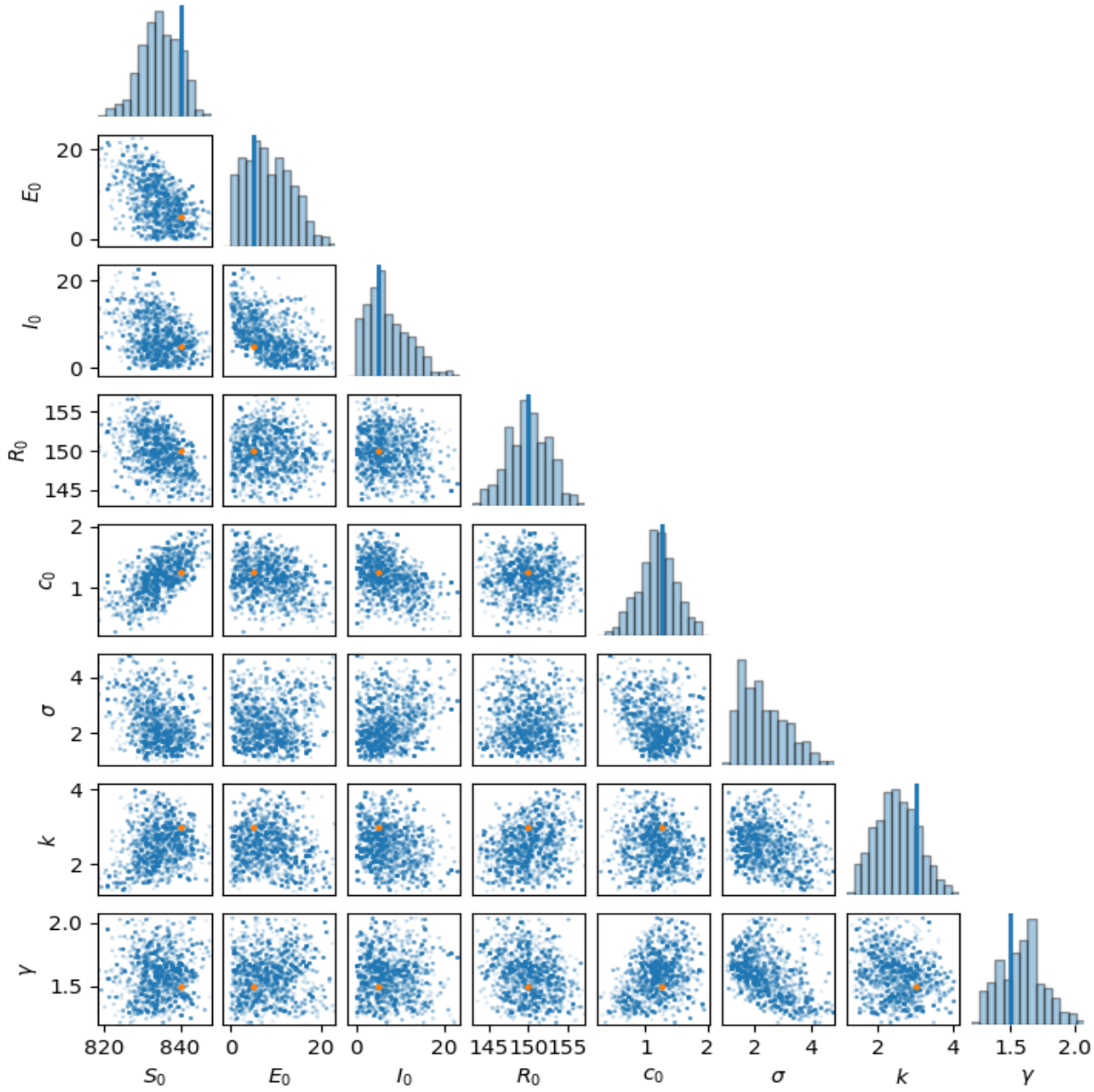


Fig. 3. 1D and 2D marginals of the sampled parameters from PMCMC

D. Trajectory estimation

Figure 5 shows the posterior system states compared to the reference system and data. The posterior states for one of every ten posterior samples are plotted as light grey trajectories. The true reference are plotted as blue lines, while the data are plotted as orange dots. PMCMC has successfully recovered the states $\{S, E, I, R\}$ with small variances. the variance for S and R increased when the state reached equilibrium, where incoming data cold no longer provide more meaningful information. In the distribution for E and I, the effect of data on decreasing the variance is clearly visible. PMCMC has also been able to recover the trajectory of β by the simple Gaussian random walk model, although big variance are clearly observed.

E. Overall summary

Overall, adaptive PMCMC has successfully estimated all the constant system parameters, as well as the trajectory of β, which is time varying. The trajectory for the system states has also been well recovered. All these estimates are only based on informative priors on three

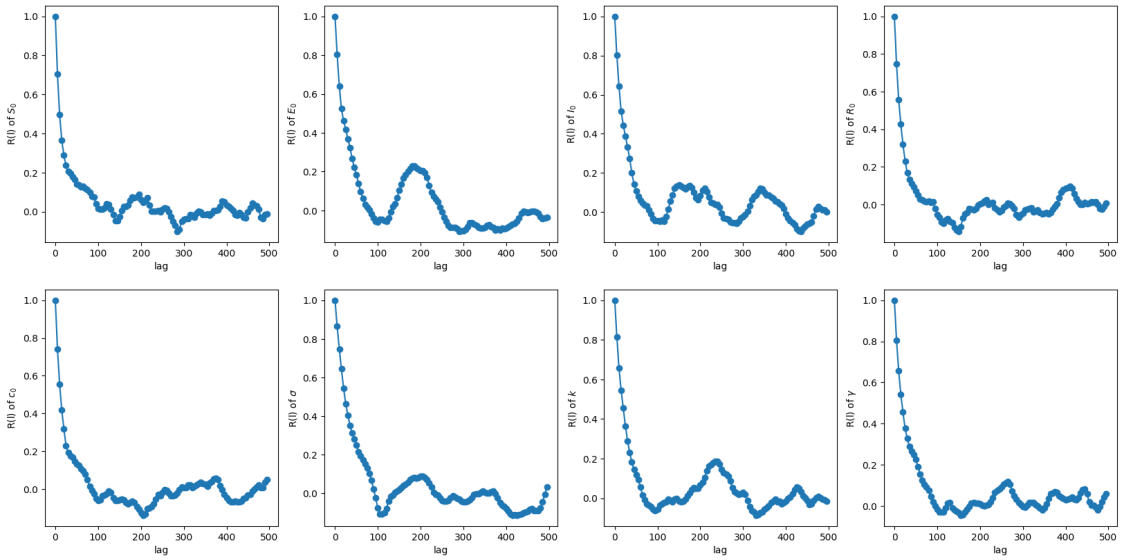


Fig. 4. Autocorrelations of the sampled parameters from PMCMC

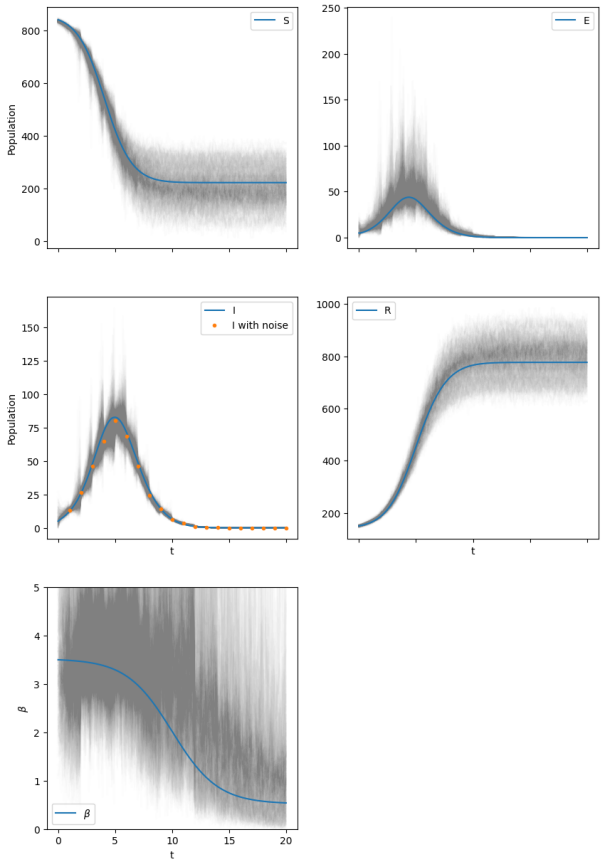


Fig. 5. Distribution of posterior system state trajectories form PMCMC

parameters, namely $\{R_0, k, \gamma\}$, while only vague ranges are provided for other parameters.

In this project, assuming no prior knowledge on the trend of β, a simple Brownian random walk is used to model its change over time, which poses increasing stochastic behavior to the system. For other systems in practice, prior knowledge for the behavior of time varying parameters could be incorporated into the modelling of β. For instance, if the parameter is expected to converge, then an Ornstein Uhlenbeck process can be chosen [1].

In the future, the effect of increasing sample size for PMCMC on decreasing the posterior variances could be further analyzed. Furthermore, the effect of incorporating delayed rejection in MCMC and the use of proposal function other than the dynamics model in particle filtering could also be analyzed. Also, for particle filter, its re-sampling ratio, MSE, and marginals of empirical distributions could also be assessed. However, as discussed before, the fine-tuning of particle filter typically does not have significant effect on the convergence of the overall results of PMCMC.

Overall, this project has successfully constructed a functional PMCMC model and has demonstrated its performance on the 8D non-linear parameter inference problem.

ACKNOWLEDGMENT

I would like to thank Professor Alex Gorodetsky for offering this course. The lecture materials are very enjoyable and the projects are fun yet challenging. Professor Gorodetsky has been very patient and timely in answering and explaining all my questions both in-

person and on Piazza. I would also like to thank the GSIs, Liliang Wang and Sanjan Muchandimath, for their great support through out the course.

APPENDIX

Please refer to the appended pages for the code scripts.

REFERENCES

[1] J. Dureau, K. Kalogeropoulos, and M. Baguelin, "Capturing the time-varying drivers of an epidemic using stochastic dynamical systems," *Biostatistics*, vol. 14, no. 3, pp. 541–555, 01 2013. [Online]. Available: https://doi.org/10.1093/biostatistics/kxs052

[2] A. Endo, E. van Leeuwen, and M. Baguelin, "Introduction to particle markov-chain monte carlo for disease dynamics modellers," *Epidemics*, vol. 29, p. 100363, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1755436519300301

[3] M. Baguelin, A. J. V. Hoek, S. Flasche, P. J. White, and W. J. Edmunds, "Vaccination against pandemic influenza a/h1n1v in england: a real-time economic evaluation," *Vaccine*, vol. 28, p. 2370–84, 2010. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/20096762/

[4] E. L. Ionides, C. Bretó, and A. A. King, "Inference for nonlinear dynamical systems," *Proceedings of the National Academy of Sciences*, vol. 103, no. 49, pp. 18 438–18 443, 2006. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.0603181103

[5] S. Cauchemez and N. M. Ferguson, "Likelihood-based estimation of continuous-time epidemic models from time-series data: application to measles transmission in london," *Journal of The Royal Society Interface*, vol. 5, no. 25, pp. 885–897, Jan. 2008. [Online]. Available: https://doi.org/10.1098/rsif.2007.1292

[6] S. Cauchemez, A.-J. Valleron, P.-Y. Boëlle, A. Flahault, and N. M. Ferguson, "Estimating the impact of school closure on influenza transmission from sentinel data," *Nature*, vol. 452, no. 7188, pp. 750–754, Apr. 2008. [Online]. Available: https://doi.org/10.1038/nature06732

[7] C. Andrieu, A. Doucet, and R. Holenstein, "Particle markov chain monte carlo methods," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 72, no. 3, pp. 269–342, 2010. [Online]. Available: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9868.2009.00736.x

[8] T. G. Kurtz, "3. markov processes and their generators," in *Approximation of Population Processes*. Society for Industrial and Applied Mathematics, Jan. 1981, pp. 15–21.

[9] A. Gorodetsky, *AE740 Parameter Inference and State Estimation*, 2019.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize, integrate
from mcmc_utils_and_plot import scatter_matrix
```

Estimating time-varying system parameter using particle MCMC

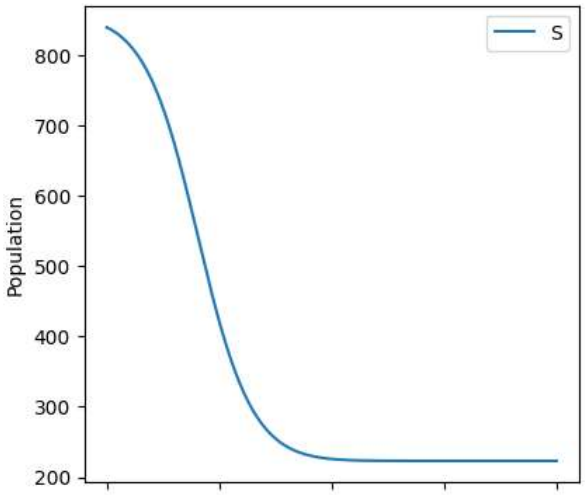
Sibo Wang December, 2022

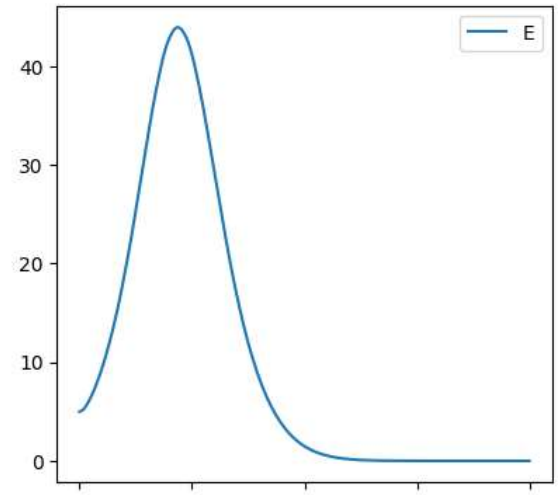
Reference system

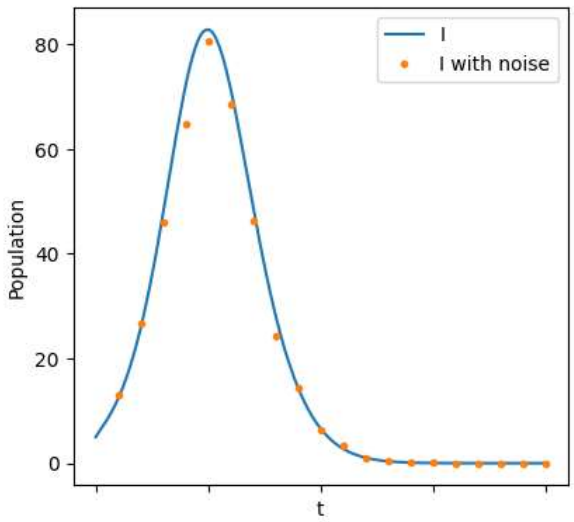
$\theta = \{S_0, E_0, I_0, R_0, c_0, \sigma, k, \gamma\} = \{840, 5, 5, 150, 3.5, \sigma, 3, 1.5\}$

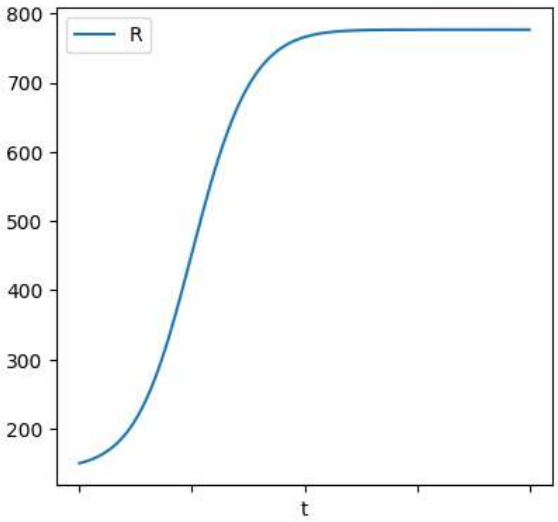
In [ ]:
```
class ReferenceSimulation():
    def __init__(self, seed = None, y0 = [840, 5, 5, 150, 3.5, 3, 1.5]):
        # S E I R
        self.y0 = y0[0:5]
        self.tspan = [0,20]
        self.t = np.linspace(0, 20, 201)

        self.seed = seed
        self.meas_std = 0.1

        self.N = 1000
        self.k = y0[5]
        self.gam = y0[6]

        self.system_states = None
        self.noise = None
        self.data = None

    def ivp_func(self, t, y):
        St = y[0]
        Et = y[1]
        It = y[2]
        Rt = y[3]
        bt = y[4]
        dS = - bt * St * It / self.N
        dE = bt * St * It / self.N - Et * self.k
        dI = Et * self.k - It * self.gam
        dR = It * self.gam
        db = -(1.5 * np.exp(-0.5*(t-10))) / (np.power(1 + np.exp(-0.5*(t-10)), 2))
        # db = 0

        return [dS, dE, dI, dR, db]

    def solve_system(self):
        sol = integrate.solve_ivp(self.ivp_func, self.tspan, self.y0, method="RK45"
        self.system_states = sol.y
        return sol.t, sol.y

    def generate_data(self):
        self.solve_system()
```

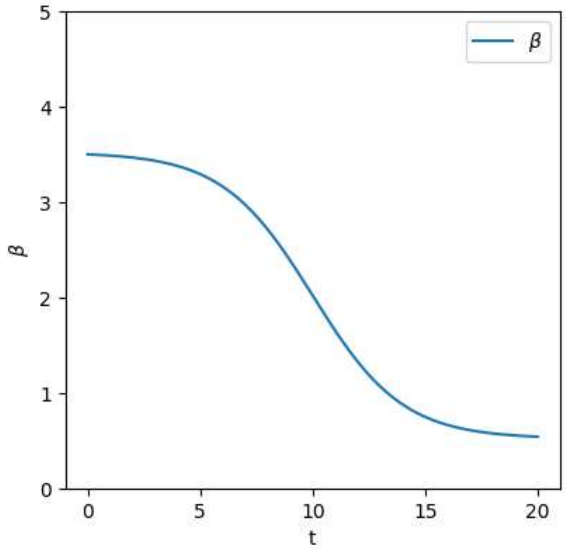
```
        rng = np.random.default_rng(seed=self.seed)
        self.noise = self.meas_std * rng.normal(size = 20)

        self.data = np.exp(np.log(self.system_states[2, 0::10][1:]) + self.noise)
        # self.data = self.system_states[2, 0::10][1:] + self.noise

        return self.t, self.data

    def plot_system_state(self, state_samples=None):
        fig, ax = plt.subplots(3,2,figsize=(10,15), sharex='all')
        if (state_samples is not None):
            for state_sample in state_samples:
                self.plot_system_state_separate(t, state_sample, ax)

        ax[0,0].plot(self.t, self.system_states[0, :], label="S")
        ax[0,1].plot(self.t, self.system_states[1, :], label="E")
        ax[1,0].plot(self.t, self.system_states[2, :], label="I")
        ax[1,0].plot(self.t[0::10][1:], self.data, '.', label="I with noise")
        ax[1,1].plot(self.t, self.system_states[3, :], label="R")
        ax[2,0].plot(self.t, self.system_states[4, :], label=r"$\beta$")
        ax[0,0].legend()
        ax[0,1].legend()
        ax[1,0].legend()
        ax[1,1].legend()
        ax[2,0].legend()
        ax[2,0].set_ylim(0,5)
        ax[1, 0].set_xlabel("t")
        ax[1, 1].set_xlabel("t")
        ax[2, 0].set_xlabel("t")
        ax[0, 0].set_ylabel("Population")
        ax[1, 0].set_ylabel("Population")
        ax[2, 0].set_ylabel(r"$\beta$")
        ax[2, 1].axis('off')

    def plot_system_state_separate(self, t, system_data, ax):
        ax[0,0].plot(t, system_data[0, :], color='grey', alpha=0.01)
        ax[0,1].plot(t, system_data[1, :], color='grey', alpha=0.01)
        ax[1,0].plot(t, system_data[2, :], color='grey', alpha=0.01)
        ax[1,1].plot(t, system_data[3, :], color='grey', alpha=0.01)
        ax[2,0].plot(t, np.exp(system_data[4, :]), color='grey', alpha=0.01)


reference = ReferenceSimulation(seed=2018)
t, data = reference.generate_data()
reference.plot_system_state()
```











PMCMC setup

Adaptive MCMC

In [ ]:
```
class PMCMC:
    '''
    Partilce MCMC
    Use particle filter to sample X_{1:T} based on new params and compute lopdf_pro
```

```
    Accept or reject new params based on Metrpolis-Hastings
    '''
    def __init__(self, mcmc_nsamples, nparticles, rwalk_cov, k0 = 50, s_d = 2.4 * 2
        self.mcmc_nsamples = mcmc_nsamples
        self.nparticles = nparticles

        # === More setups for adaptive MCMC ===
        self.rwalk_cov = rwalk_cov
        self.cov_chol = np.linalg.cholesky(rwalk_cov)   # Gaussian random walk cova
        self.k0 = k0
        self.s_d = s_d
        self.eps = eps


    def sample_mcmc(self, params_pdf_func, prior_params,
                          particle_filter_sampler,
                          data, meas_noise,
                          T, dt):
        # === Assumptions ===
        # 1. Data is 1-dimensional and are avalaible iff t = 1, 2, 3, ...
        # 2. The total time T and step size dt are tuned such that t = 1, 2, 3, ...

        # === Initial setup ===

        t_vec = np.linspace(0, T, num=int(T/dt)+1)
        nsteps = t_vec.shape[0]

        def states_pdf_func(p_params):
            return particle_filter_sampler(p_params, data, meas_noise, t_vec, self.

        # === n = 0 (intial sample of params) ===
        # Calculate the pdf of prior params, and run particle filter to sample cond
        params_logpdf = params_pdf_func(prior_params)
        states_logpdf, system_state_init_sample = states_pdf_func(prior_params)

        self.dim_params = prior_params.shape[0]                 # dimension of para
        self.dim_states = system_state_init_sample.shape[0]      # dimension of sta

        self.params_samples = np.zeros((self.mcmc_nsamples, self.dim_params))
        self.system_state_samples = np.zeros((self.mcmc_nsamples, self.dim_states,

        self.params_samples[0, :] = prior_params
        self.system_state_samples[0, :, :] = system_state_init_sample

        x_mean_prior = prior_params
        count_accept = 1
        count_samples = 1
        for n in range(1, self.mcmc_nsamples):
            proposed_params = self.params_samples[n-1, :] + np.dot(self.cov_chol, r
            # Impose S + E + I + R = N
            proposed_params[0] = reference.N - proposed_params[1] - proposed_params
            proposed_params_logpdf = params_pdf_func(proposed_params)

            proposed_state_logpdf, proposed_system_state = states_pdf_func(proposed

            # Determine acceptance probability
            # For the normal random walk proposal, q(x|y) = q(y|x). Thus a = min{f
            a = np.exp(np.min((proposed_state_logpdf + proposed_params_logpdf - par

            # Accept or reject
```

```
            u = np.random.rand()
            if u < a:
                self.system_state_samples[n, :, :] = proposed_system_state
                self.params_samples[n, :] = proposed_params
                states_logpdf = proposed_state_logpdf
                params_logpdf = proposed_params_logpdf
                count_accept += 1
            else:
                self.system_state_samples[n, :, :] = self.system_state_samples[n-1,
                self.params_samples[n, :] = self.params_samples[n-1, :]
            count_samples += 1

            # Adapt covariance (Sigma)
            k = (count_samples - 1)
            cov_adapt, x_mean_prior = self.adapt_covariance(self.rwalk_cov, x_mean_
            if  k >= self.k0:
                self.rwalk_cov = cov_adapt
                self.cov_chol = np.linalg.cholesky(self.rwalk_cov)

            # Print progress at intervals
            if n % int(self.mcmc_nsamples / 10) == 0:
                print(f"On sample {n}: Accepted samples ratio = {count_accept / cou
            if n == self.mcmc_nsamples - 1:
                self.accept_ratio = count_accept / count_samples

    def adapt_covariance(self, S, x_mean_prior, x, k, s_d, eps):
        x_mean = 1.0 / (k + 1.0) * x + k / (k + 1.0) * x_mean_prior

        x_mean_prior_matrix = np.outer(x_mean_prior, x_mean_prior)
        x_mean_matrix = np.outer(x_mean, x_mean)
        x_matrix = np.outer(x, x)

        S = (k - 1.0) / k * S + s_d / k * (eps * np.eye(self.dim_params) + k * x_me
        return S, x_mean
```

Parameter distribution $P(\theta)$

In [ ]:

```
def seir_params_pdf_func(params):
    # params: {S_0, E_0, I_0, R_0, c_0, \sigma, k, \gamma}
    # S_0:      N - E0 - I0 - R0
    # E_0:      U(0, 10)
    # I_0:      U(0, 10)
    # R_0:      N(150, 10)
    # c_0:      U(-2, 2)
    # sigma:    U(0, 1)
    # k:        N(3, 0.5)
    # gamma:    N(1.5, 0.5)
    if (np.any(params[0:4]<0)):
        return -1e9
    pdf_R = -0.5 * (params[3] - 150) ** 2 / 10
    pdf_k = -0.5 * (params[6] - reference.k) ** 2 / 0.5
    pdf_gamma = -0.5 * (params[7] - reference.gam) ** 2 / 0.5
    return pdf_R + pdf_k + pdf_gamma
```

Measurement model $P(Y_k \mid X_k)$

In [ ]:

```
def seir_measurement_pdf(state, data, meas_noise):
    log_sys_data = np.log(state[:, 2])
```

```
    log_data = np.log(data)
    residual = log_sys_data - log_data
    # residual = state[:, 2] - data
    # print(data)
    return -0.5 * np.power(residual, 2) / meas_noise
```

Particle filter for $P(Y \mid \theta)$

In [ ]:

```
# log likelihood of a given set of params based on particle filter
def seir_particle_filter_sampler(proposed_params, all_data, meas_noise, t_vec, npar
    # params: {S_0, E_0, I_0, R_0, c_0, \sigma, k, \gamma}
    nsteps = t_vec.shape[0]
    dt = t_vec[1] - 0
    dim = 5

    samples = np.zeros((nparticles, dim, nsteps))
    weights = np.zeros((nparticles, nsteps))
    eff = np.zeros((nsteps))
    chosen_path = np.zeros((dim, nsteps))
    rr = np.arange(nparticles)          # For low-variance resampling

    samples[:, :, 0] = np.tile(proposed_params[0:5], (nparticles, 1))
    weights[:, 0] = 1.0 / nparticles    # all weights are equal because of indepen
    eff[0] = nparticles
    chosen_path[:, 0] = samples[0, :, 0] # choose any, they are the same

    data_ii = 0
    loglike_cum = 0
    for ii, t in enumerate(t_vec):
        if ii == 0:
            continue

        # Propose based on system dynamics
        samples[:, :, ii] = seir_forward_euler(samples[:, :, ii-1], t, dt, propose

        if (t % 1 == 0):
            # Update weights
            loglike = seir_measurement_pdf(samples[:, :, ii], all_data[data_ii], me
            # print("t = ", t, "loglike = ", np.max(loglike), np.min(loglike))

            if (np.min(loglike) < -400 or np.any(np.isnan(loglike))):
                loglike_cum = -1e9
                # print("\tparams: ", proposed_params, "\tt = ", t, "\tmin log like
                break

            loglike_cum += np.log(np.mean(np.exp(loglike)))
            log_current_weights = loglike + np.log(weights[:, ii-1])

            weights[:, ii] = np.exp(log_current_weights) / np.sum(np.exp(log_curre

            data_ii += 1
        else:
            weights[:, ii] = weights[:, ii-1]

        chosen_path[:, ii] = samples[np.random.choice(rr, p=weights[:, ii]), :, ii

        eff[ii] = 1.0 / np.sum(weights[:, ii]**2)
        if eff[ii] < 0.5 * nparticles:
```

```
            # Low variance resampling
            samp_inds = np.random.choice(rr, nparticles, p=weights[:, ii])
            samples[:, :, ii] = samples[samp_inds, :, ii]
            weights[:, ii] = np.ones((nparticles))/nparticles

    return loglike_cum, chosen_path


def seir_forward_euler(y, t, dt, proposed_params):
    # y: (nparticles, dim)
    N = reference.N
    sigma = proposed_params[0]
    k = proposed_params[1]
    gamma = proposed_params[2]

    St = y[:,0]
    Et = y[:,1]
    It = y[:,2]
    Rt = y[:,3]
    xt = y[:,4]
    bt = np.exp(xt)

    dS = - bt * St * It / N
    dE = bt * St * It / N - Et * k
    dI = Et * k - It * gamma
    dR = It * gamma
    dx = sigma * np.sqrt(dt) * np.random.randn(y.shape[0])

    St = St + dS * dt
    Et = Et + dE * dt
    It = It + dI * dt
    Rt = Rt + dR * dt
    xt = xt + dx * dt

    return np.array([St, Et, It, Rt, xt]).T
```

Result

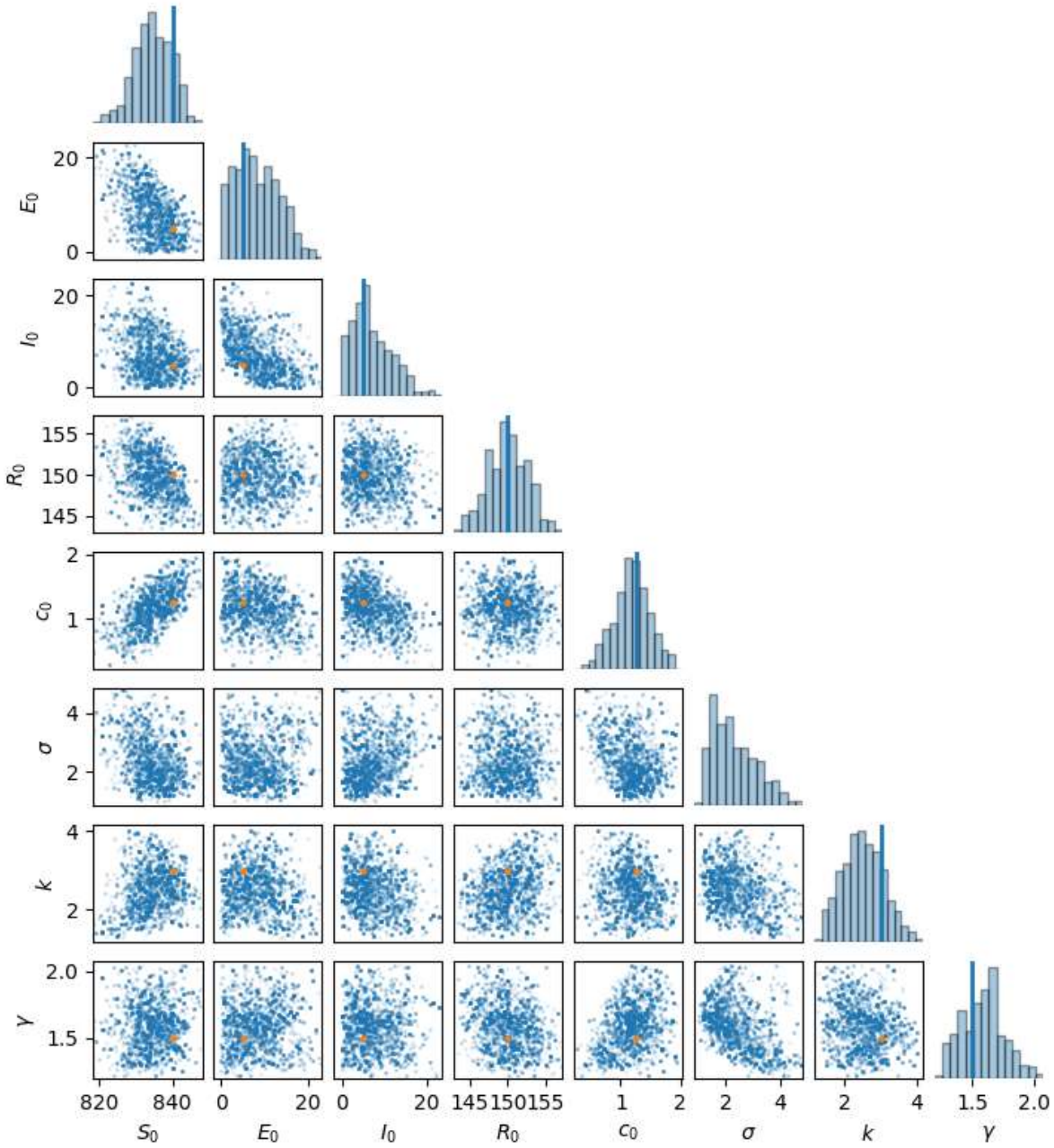
In [ ]:

```
# Reference: $\theta = \{S_0, E_0, I_0, R_0, c_0, \sigma, k, \gamma\} = \{840, 5, 5
# S_0:      N - E0 - I0 - R0
# E_0:      U(0, 10)
# I_0:      U(0, 10)
# R_0:      N(150, 10)
# c_0:      U(-2, 2)
# sigma:    U(0, 1)
# k:        N(3, 0.5)
# gamma:    N(1.5, 0.5)
# prior_params = np.array([840, 5, 5, 150, np.random.uniform(-2,2), np.random.unifo
prior_params = np.array([840, np.random.uniform(0,10), np.random.uniform(0,10), 150
prior_params[0] = reference.N - prior_params[1] - prior_params[2] - prior_params[3]

# S_0, E_0, I_0, R_0, c_0, \sigma, k, \gamma
rwalk_cov = np.diag([1, 0.1, 0.1, 1, 0.05, 0.01, 0.005, 0.01])


pmcmc = PMCMC(mcmc_nsamples = 5000, nparticles = 1000, rwalk_cov = rwalk_cov, k0 =
pmcmc.sample_mcmc(params_pdf_func = seir_params_pdf_func, prior_params = prior_para
                            particle_filter_sampler = seir_particle_filter_sampler,
                            data = reference.data, meas_noise = reference.meas_std*
                            T=20, dt=0.1)
```

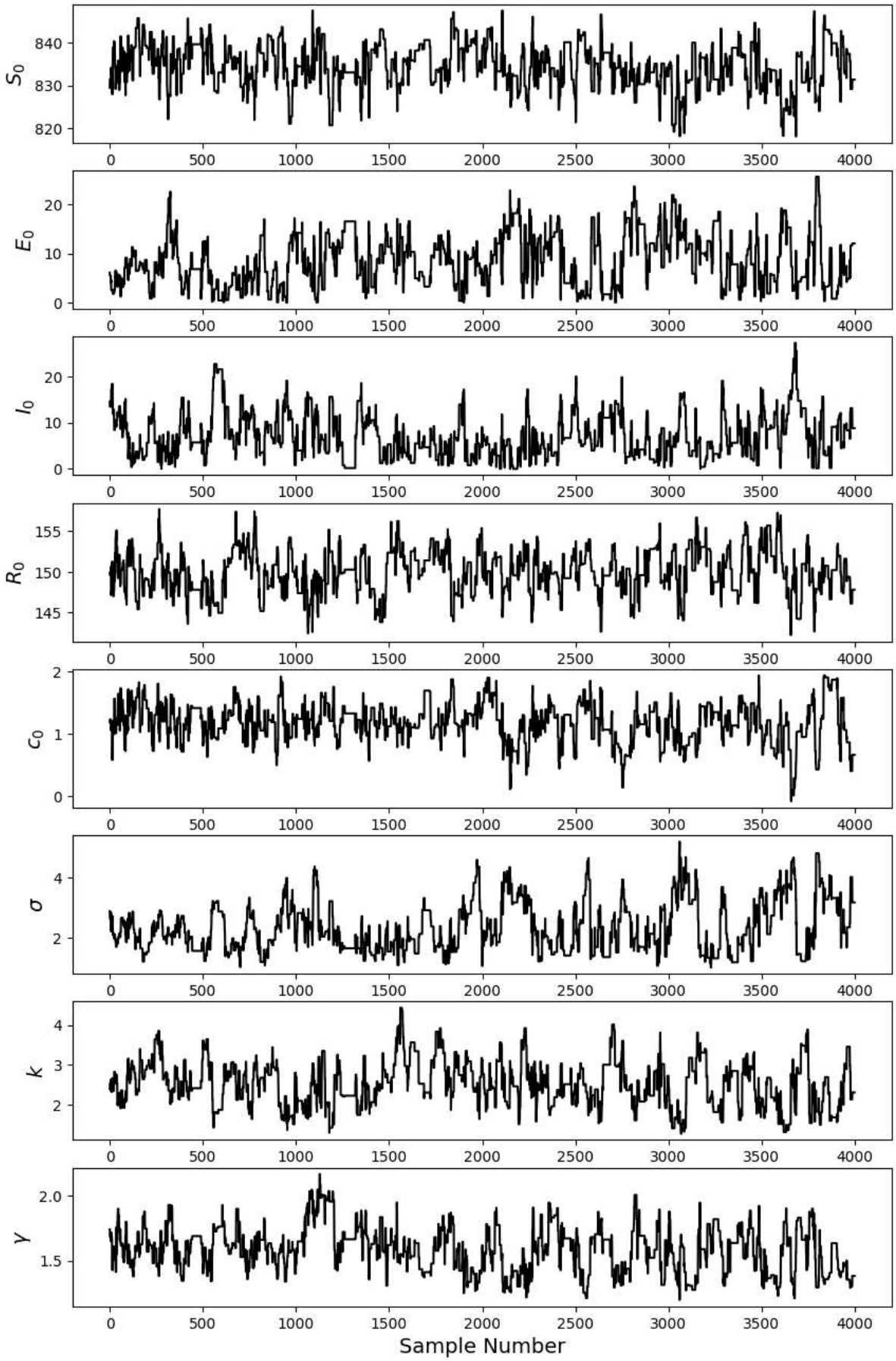
```
On sample 500: Accepted samples ratio = 0.2894211576846307, states log like: -22.4
8838210681411
On sample 1000: Accepted samples ratio = 0.29270729270729273, states log like: -2
6.518386717789244
C:\Users\Sybokia\AppData\Local\Temp\ipykernel_21752\4087969586.py:2: RuntimeWarnin
g: invalid value encountered in log
  log_sys_data = np.log(state[:, 2])
On sample 1500: Accepted samples ratio = 0.29447035309793473, states log like: -2
1.861584755978473
On sample 2000: Accepted samples ratio = 0.28385807096451776, states log like: -2
3.644146550758997
On sample 2500: Accepted samples ratio = 0.2746901239504198, states log like: -21.
203966222582217
On sample 3000: Accepted samples ratio = 0.263245584805065, states log like: -24.2
25937770185126
On sample 3500: Accepted samples ratio = 0.2582119394458726, states log like: -23.
349210722401622
On sample 4000: Accepted samples ratio = 0.2516870782304424, states log like: -22.
237948597749607
On sample 4500: Accepted samples ratio = 0.24705620973117084, states log like: -2
1.7231104679241
```

In [ ]:
```
def sub_sample_data(samples, system_state_samples, frac_burn=0.2, frac_use=0.7):
    nsamples = samples.shape[0]
    inds = np.arange(nsamples, dtype=int)
    start = int(frac_burn * nsamples)
    inds = inds[start:]
    nsamples = nsamples - start
    step = int(nsamples / (nsamples * frac_use))
    inds2 = np.arange(0, nsamples, step)
    inds = inds[inds2]
    subsamples = samples[inds, :]
    system_state_samples = system_state_samples[inds, :, :]
    return subsamples, system_state_samples

sub_samples, sub_state_samples = sub_sample_data(pmcmc.params_samples, pmcmc.syster
```

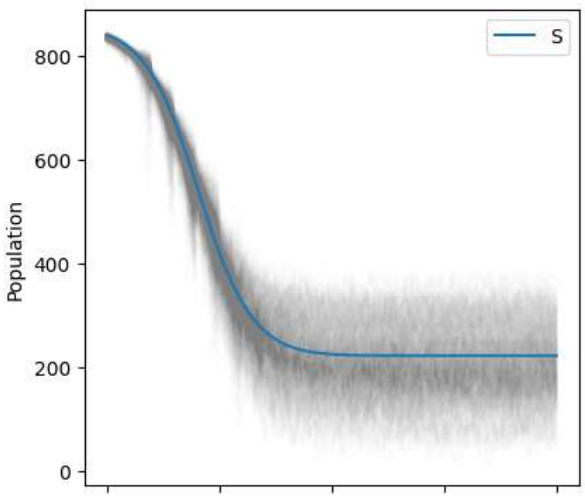
In [ ]:
```
true_param = np.array([840, 5, 5, 150, np.log(3.5), np.random.uniform(0, 1), 3, 1.5

def plot_identifiable_posterior(samples):
   fig, axs, gs = scatter_matrix([samples], #list of chains
                                 labels=[r"$S_0$", r"$E_0$", r"$I_0$", r"$R_0$",
                                hist_plot=False, # if false then only data
                                gamma=0.1,
                                specials={"vals":true_param}
                                )
   fig.set_size_inches(8, 8)

plot_identifiable_posterior(sub_samples)
```

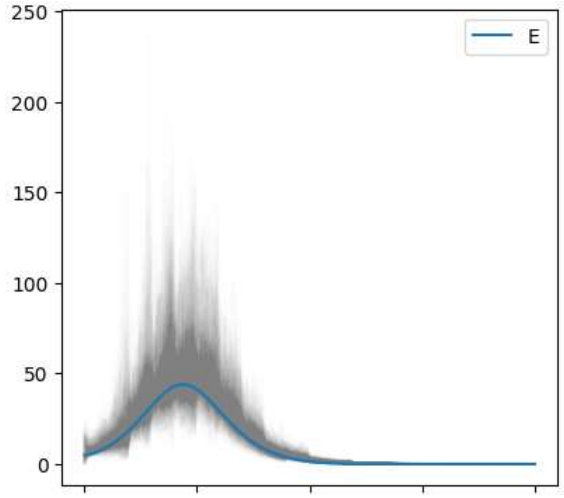


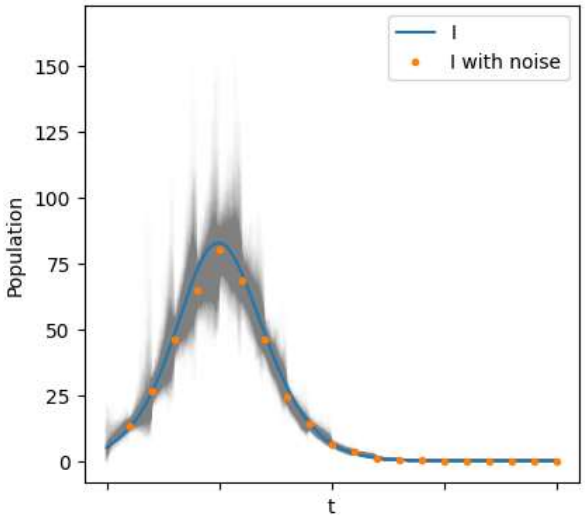
In [ ]:
```
def plot_mixing(samples, labels=None):
    num_params = samples.shape[1]
    fig, axs = plt.subplots(num_params, 1, figsize=(10,2*num_params))
    for ii in range(num_params):
        axs[ii].plot(samples[:, ii], '-k')
        if labels == None:
            axs[ii].set_ylabel(f'$x_{ii+1}$', fontsize=14)
        else:
            axs[ii].set_ylabel(labels[ii], fontsize=14)
    axs[ii].set_xlabel('Sample Number', fontsize=14)

plot_mixing(sub_samples, labels=[r"$S_0$", r"$E_0$", r"$I_0$", r"$R_0$", r"$c_0$",
```

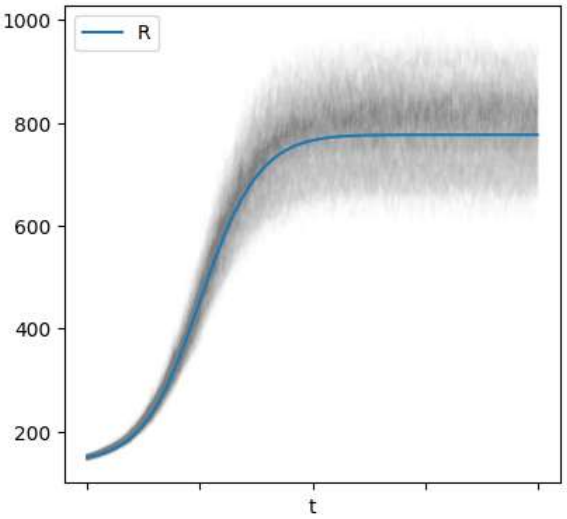


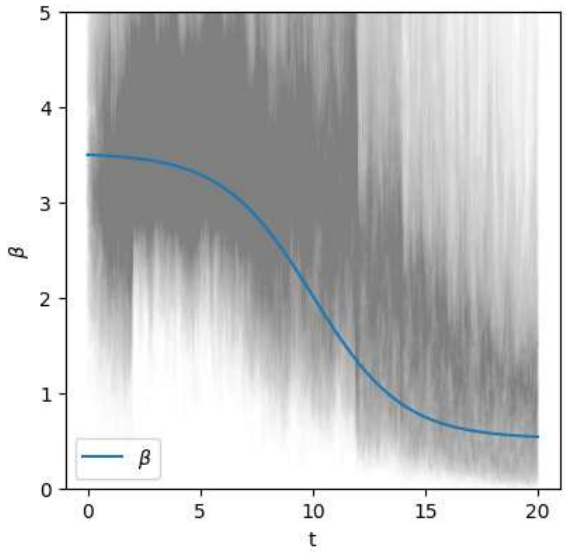
In [ ]:
```
reference.plot_system_state(sub_state_samples[::10, :, :])
```







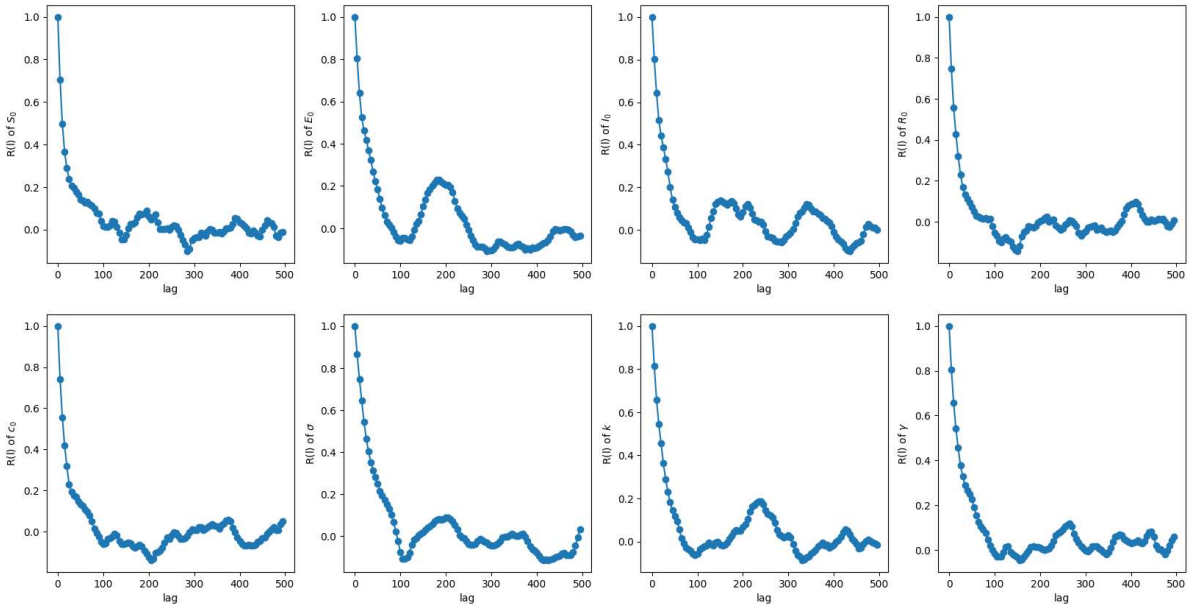




In [ ]:
```
def autocorrelation(samples, maxlag=100, step=1):
    """Compute the correlation of a set of samples

    Inputs
    ------
    samples: (N, d)
    maxlag: maximum distance to compute the correlation for
    step: step between distances from 0 to maxlag for which to compute teh correlat
    """
```

```
    # Get the shapes
    ndim = samples.shape[1]
    nsamples = samples.shape[0]

    # Compute the mean
    mean = np.mean(samples, axis=0)

    # Compute the denominator, which is variance
    denominator = np.zeros((ndim))
    for ii in range(nsamples):
        denominator = denominator + (samples[ii, :] - mean)**2

    lags = np.arange(0, maxlag, step)
    autos = np.zeros((len(lags), ndim))
    for zz, lag in enumerate(lags):
        autos[zz, :] = np.zeros((ndim))
        # compute the covariance between all samples *lag apart*
        for ii in range(nsamples - lag):
            autos[zz,:] = autos[zz, :] + (samples[ii,:]-mean)*(samples[ii + lag,:]
        autos[zz, :] = autos[zz, :]/denominator
    return lags, autos

def plot_autocorrelation(lags, autolag, labels=None):
    num_params = autolag.shape[1]
    fig, axs = plt.subplots(2, int(num_params/2), figsize=(5*num_params/2, 5*2))
    for ii in range(num_params):
        axs[int(ii/4), ii%4].plot(lags, autolag[:, ii],'-o')
        axs[int(ii/4), ii%4].set_xlabel('lag')
        if labels == None:
            axs[int(ii/4), ii%4].set_ylabel(f'autocorrelation dimension {ii+1}')
        else:
            axs[int(ii/4), ii%4].set_ylabel(f"R(l) of {labels[ii]}")

    plt.show()
```

In [ ]:
```
maxlag=500
step=5
lags, autolag = autocorrelation(sub_samples, maxlag=maxlag,step=step)

plot_autocorrelation(lags, autolag, labels=[r"$S_0$", r"$E_0$", r"$I_0$", r"$R_0$",
```



In [ ]:
```
lags_id_all, autolag_id_all = autocorrelation(sub_samples, maxlag=500, step=1)
```

In [ ]:
```
def find_iac(autolag):
    num_dim = autolag.shape[1]

    stop_index = np.argmax(autolag<0, 0)
    stop_index[stop_index==0] = autolag.shape[0]
    print(f"Lag that reached negative: {stop_index}")

    iac = np.empty(num_dim)
    for ii in range(num_dim):
        iac[ii] = 1.0 + 2 * np.sum(autolag[0:stop_index[ii], ii])
    return iac

iac_identifiable = find_iac(autolag_id_all)
print(iac_identifiable)
```

```
Lag that reached negative: [132  81  83  93  84  93  69  96]
[47.13014059 52.35152491 47.95914723 37.17927542 42.31886872 66.07149732
 46.5288139  56.99550596]
```