tssl_lab3

October 6, 2020

1 TSSL Lab 3 - Nonlinear state space models and Sequential Monte Carlo

In this lab we will make use of a non-linear state space model for analyzing the dynamics of SARS-CoV-2, the virus causing covid-19. We will use an epidemiological model referred to as a Susceptible-Exposed-Infectious-Recovered (SEIR) model. It is a stochastic adaptation of the model used by the The Public Health Agency of Sweden for predicting the spread of covid-19 in the Stockholm region early in the pandemic, see Estimates of the peak-day and the number of infected individuals during the covid-19 outbreak in the Stockholm region, Sweden February – April 2020.

The background and details of the SEIR model that we will use are available in the document *TSSL Lab 3 Predicting Covid-19 Description of the SEIR model* on LISAM. Please read through the model description before starting on the lab assignments to get a feeling for what type of model that we will work with.

1.0.1 DISCLAIMER

Even though we will use a type of model that is common in epidemiological studies and analyze real covid-19 data, you should *NOT* read to much into the results of the lab. The model is intentionally simplified to fit the scope of the lab, it is not validated, and it involves several model parameters that are set somewhat arbitrarily. The lab is intended to be an illustration of how we can work with nonlinear state space models and Sequential Monte Carlo methods to solve a problem of practical interest, but the actual predictions made by the final model should be taken with a big grain of salt.

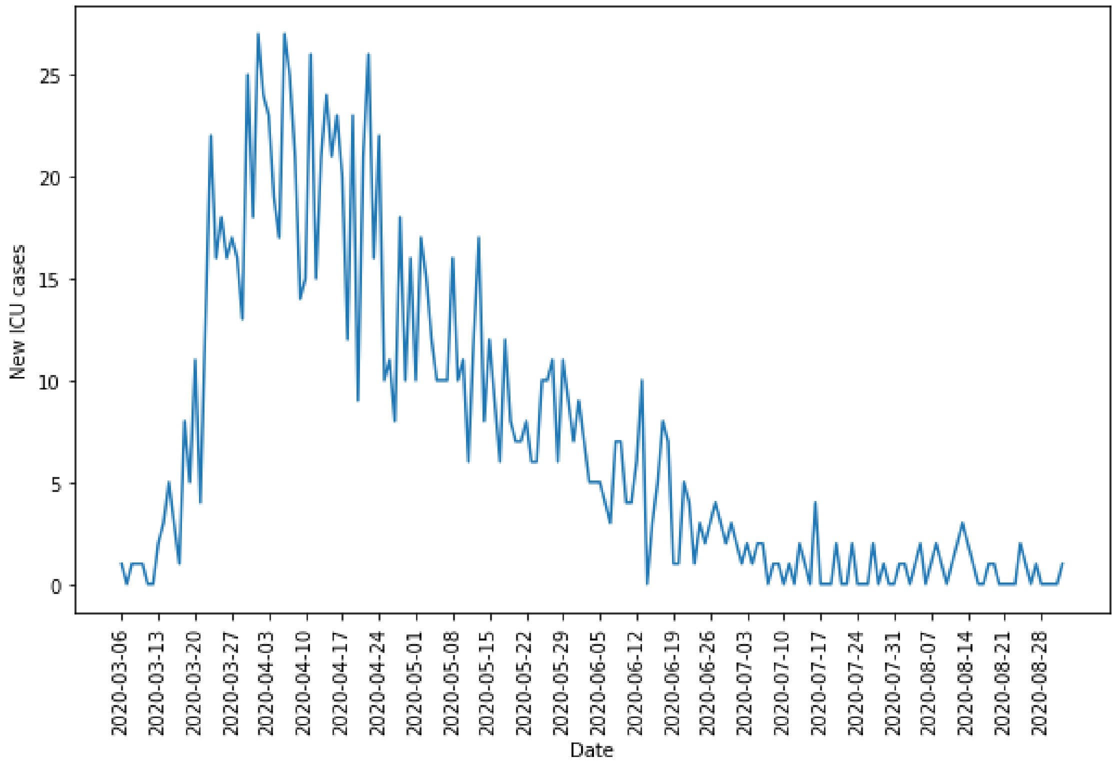
We load a few packages that are useful for solving this lab assignment.

```
[1]: import pandas  # Loading data / handling data frames
     import numpy as np
     import matplotlib.pyplot as plt
     plt.rcParams["figure.figsize"] = (10,6)  # Increase default size of plots
```

1.1 3.1 A first glance at the data

The data that we will use in this lab is a time series consisting of daily covid-19-related intensive care cases in Stockholm from March to August. As always, we start by loading and plotting the data.

[2]:
```
data=pandas.read_csv('SIR_Stockholm.csv',header=0)
y_sthlm = data['ICU'].values
u_sthlm = data['Date'].values
ndata = len(y_sthlm)
plt.plot(u_sthlm,y_sthlm)
plt.xticks(range(0, ndata, 7), u_sthlm[::7], rotation = 90)  # Show only one␣
↪tick per week for clarity
plt.xlabel('Date')
plt.ylabel('New ICU cases')
plt.show()
```



Q0: What type of values can the observations y_t take? Is a Gaussian likelihood model a good choice if we want to respect the properties of the data?

A: Observations y_t can take only positive values (greateer than or equal to zero). Hence, a gaussian likelihood model is not an option as it would be equally likely for negative y_t to be observed under that assumption.

1.2 3.2 Setting up and simulating the SEIR model

In this section we will set up a SEIR model and use this to simulate a synthetic data set. You should keep these simulated trajectories, we will use them in the following sections.

[3]:
```
from tssltools_lab3 import Param, SEIR

"""For Stockholm the population is probably roughly 2.5 million."""
population_size = 2500000

"""" Binomial probabilities (p_se, p_ei, p_ir, and p_ic) and the transmission␣
↪rate (rho)"""
pse = 0         # This controls the rate of spontaneous s->e transitions. It is␣
↪set to zero for this lab.
pei = 1 / 5.1   # Based on FHM report
pir = 1 / 5     # Based on FHM report
pic = 1 / 1000 # Quite arbitrary!
rho = 0.3       # Quite arbitrary!

""" The instantaneous contact rate b[t] is modeled as
  b[t] = exp(z[t])
  z[t] = z[t-1] + epsilon[t], epsilon[t] ~ N(0,sigma_epsilon^2)
"""
sigma_epsilon = .1

""" For setting the initial state of the simulation"""
i0 = 1000   # Mean number of infectious individuals at initial time point
e0 = 5000   # Mean number of exposed...
r0 = 0      # Mean number of recovered
s0 = population_size - i0 - e0 - r0  # Mean number of susceptible
init_mean = np.array([s0, e0, i0, 0.], dtype=np.float64)  # The last 0. is the␣
↪mean of z[0]

"""All the above parameters are stored in params."""
params = Param(pse, pei, pir, pic, rho, sigma_epsilon, init_mean,␣
↪population_size)

""" Create a model instance"""
model = SEIR(params)
```

Q1: Generate 10 different trajectories of length 200 from the model an plot them in one figure. Does the trajectories look reasonable? Could the data have been generated using this model?

For reproducability, we set the seed of the random number generator to 0 before simulating the trajectories using np.random.seed(0)

Save these 10 generated trajectories for future use.

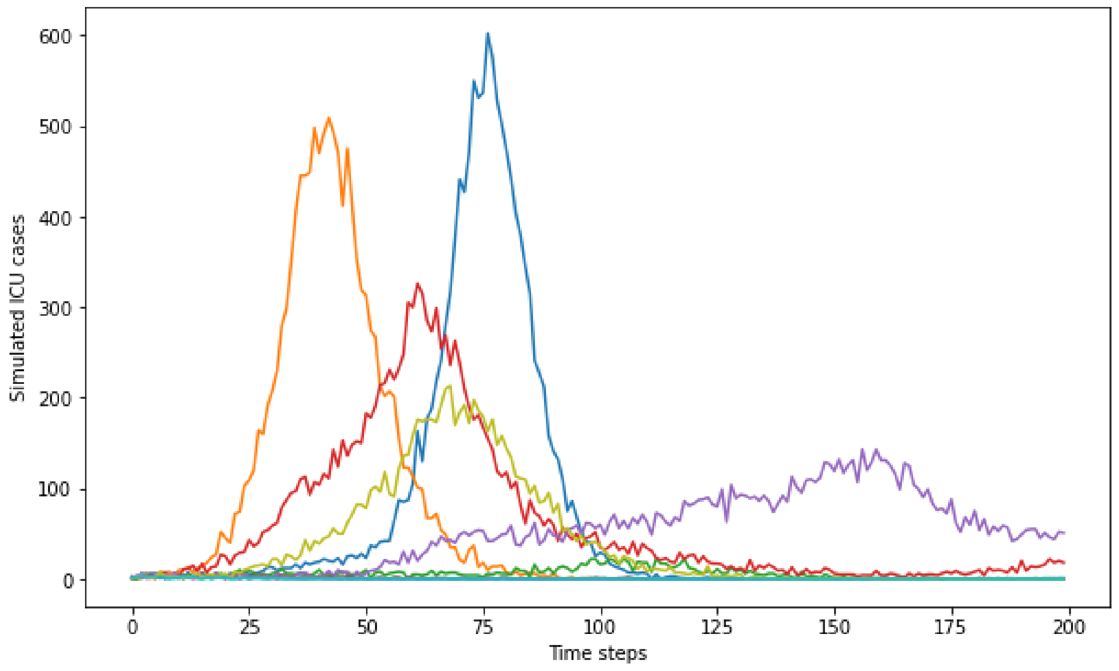
(hint: The SEIR class has a simulate method)

[4]:
```
np.random.seed(0)
help(model.simulate)
```

```
Help on method simulate in module tssltools_lab3:

simulate(T, N=1) method of tssltools_lab3.SEIR instance
    Simulates the SEIR model for a given number of time steps. Multiple
trajectories
    can be simulated simulataneously.

    Parameters
    ----------
    T : int
        Number of time steps to simulate the model for.
    N : int, optional
        Number of independent trajectories to simulate. The default is 1.

    Returns
    -------
    alpha : ndarray
        Array of size (d,N,T) with state trajectories. alpha[:,i,:] is the i:th
trajectory.
    y : ndarray
        Array of size (1,N,T) with observations.
```

[5]:
```
N=10
model_sim = model.simulate(T=200, N=N)
alpha_sim = model_sim[0]
y_sim     = model_sim[1]
```

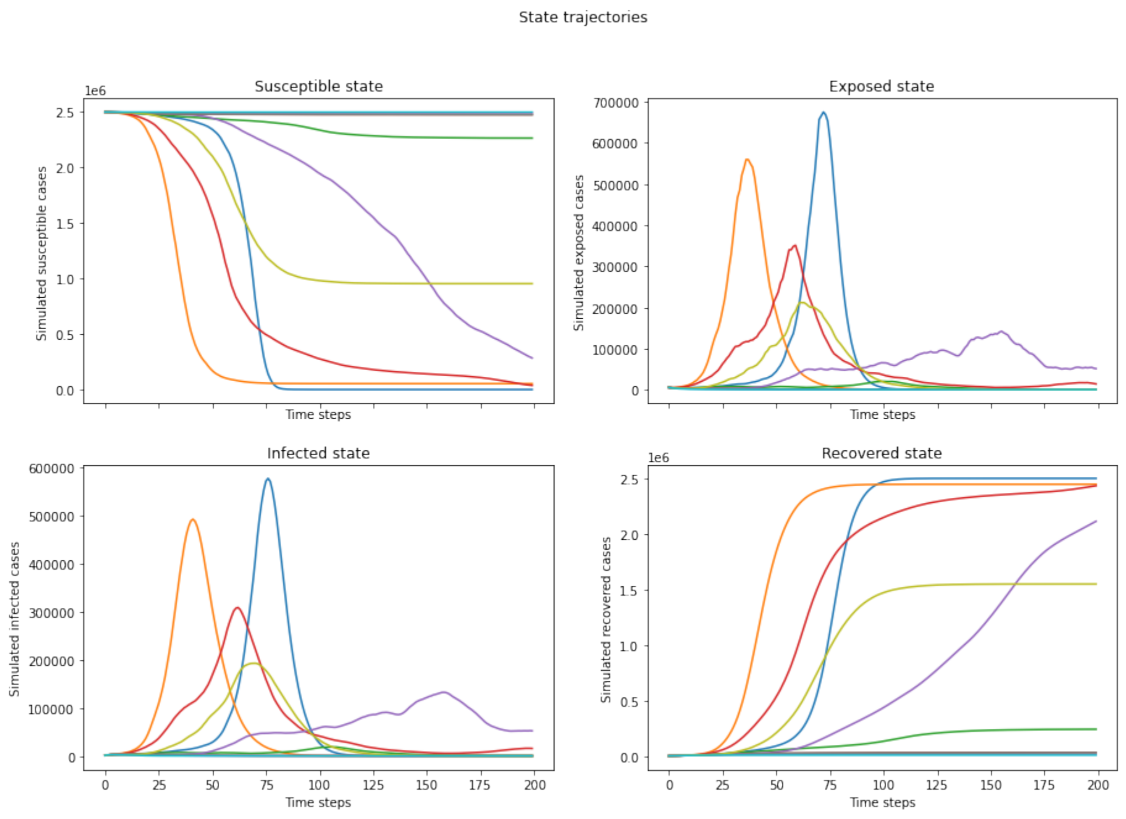
[6]:
```
# Plot trajectories of simulated observations (ICU cases)
for i in range(N):
    plt.plot(y_sim[0, i, :])
    plt.xlabel('Time steps')
    plt.ylabel('Simulated ICU cases')
```



The simulated trajectories are reasonably similar to the distribution of observed data. However the range of y_t is quite high in the simulated samples. But that can be tuned by adjusting the parameters of the data generating model. In conclusion the data could have been generated by this model or a similar one with different hyperparameters.

[7]:
```
# Plot the trajectories of simulated states - susceptible, exposed, infected␣
↪cases and transmission rates
fig, ax = plt.subplots(nrows=2, ncols=2, sharex='all', figsize=(15,10))
fig.suptitle('State trajectories')

# Recovered state
recovered = population_size - np.sum(alpha_sim[:model.d-1, :, :], axis=0)

for i in range(N):
    ax[0, 0].plot(alpha_sim[0, i, :])
    ax[0, 0].set(xlabel='Time steps', ylabel='Simulated susceptible cases',␣
↪title='Susceptible state')

    ax[0, 1].plot(alpha_sim[1, i, :])
    ax[0, 1].set(xlabel='Time steps', ylabel='Simulated exposed cases',␣
↪title='Exposed state')

    ax[1, 0].plot(alpha_sim[2, i, :])
    ax[1, 0].set(xlabel='Time steps', ylabel='Simulated infected cases',␣
↪title='Infected state')
```

```
    ax[1, 1].plot(recovered[i, :])
    ax[1, 1].set(xlabel='Time steps', ylabel='Simulated recovered cases',␣
↪title='Recovered state')

    #ax[2, 0].plot(alpha_sim[3, i, :])
    #ax[2, 0].set(xlabel='Time steps', ylabel='Simulated transmission rates',␣
↪title='Transmission rate')

    #ax[2, 1].axis('off')
```



1.3 3.3 Sequential Importance Sampling

Next, we pick out one trajectory that we will use for filtering. We use simulated data to start with, since we then know the true underlying SEIR states and can compare the filter results with the ground truth.

Q2: Implement the **Sequential Importance Sampling** algorithm by filling in the following functions.

The **exp_norm** function should return the normalized weights and the log average of the unnormalized weights. For numerical reasons, when calculating the weights we should "normalize" the log-weights first by removing the maximal value.

Let $\bar{\omega}_t = \max(\log \omega_t^i)$ and take the exponential of $\log \tilde{\omega}_t^i = \log \omega_t^i - \bar{\omega}_t$. Normalizing $\tilde{\omega}_t^i$ will yield the normalized weights!

For the log average of the unnormalized weights, care has to be taken to get the correct output, $\log(1/N \sum_{i=1}^N \tilde{\omega}_t^i) = \log(1/N \sum_{i=1}^N \omega_t^i) - \bar{\omega}_t$. We are going to need this in the future, so best to implement it right away.

(hint: look at the SEIR model class, it contains all necessary functions for propagation and weighting)

[8]:
```python
from tssltools_lab3 import smc_res

def exp_norm(logwgt):
    """
    Exponentiates and normalizes the log-weights.

    Parameters
    ----------
    logwgt : ndarray
        Array of size (N,) with log-weights.

    Returns
    -------
    wgt : ndarray
        Array of size (N,) with normalized weights, wgt[i] = exp(logwgt[i])/
↪sum(exp(logwgt)),
        but computed in a /numerically robust way/!
    logZ : float
        log of the normalizing constant, logZ = log(sum(exp(logwgt))),
        but computed in a /numerically robust way/!
    """
    N = len(logwgt)

    C_t = max(logwgt)
    logwt_tilde = logwgt - C_t

    wgt  = np.exp(logwt_tilde) / np.sum(np.exp(logwt_tilde))
    logZ = np.log(np.sum(np.exp(logwt_tilde))) + C_t

    return wgt, logZ

def ESS(wgt):
    """
    Computes the effective sample size.

    Parameters
    ----------
    wgt : ndarray
        Array of size (N,) with normalized importance weights.

    Returns
    -------
    ess : float
        Effective sample size.
    """

    ess = np.sum(wgt)**2 / np.sum(wgt**2)

    return ess

def sis_filter(model, y, N):
    d = model.d
    n = len(y)

    # Allocate memory
    particles = np.zeros((d, N, n), dtype = float)  # All generated particles
    logW = np.zeros((1, N, n))  # Unnormalized log-weight
    W    = np.zeros((1, N, n))  # Normalized weight
    alpha_filt = np.zeros((d, 1, n))  # Store filter mean
    N_eff = np.zeros(n)  # Efficient number of particles
    logZ  = 0.           # Log-likelihood estimate

    # Filter loop
    for t in range(n):
        # Sample from "bootstrap proposal"
        if t == 0:
            particles[:, :, 0] = model.sample_state(alpha0=None, N=N)       #␣
↪Initialize from p(alpha_1)
            logW[0, :, 0] = model.log_lik(y=y[t], alpha=particles[:, :, 0]) #␣
↪Compute weights
        else:
            particles[:, :, t] = model.sample_state(alpha0=particles[:, :,␣
↪t-1], N=N)      # Propagate according to dynamics
            logW[0, :, t] = model.log_lik(y=y[t], alpha=particles[:, :, t]) +␣
↪logW[0, :, t-1] # Update weights

        # Normalize the importance weights and compute N_eff
        W[0, :, t], logZ = exp_norm(logW[0, :, t])
        N_eff[t] = ESS(W[0, :, t])

        # Compute filter estimates
        alpha_filt[:, 0, t] = np.sum(W[0, :, t]*particles[:, :, t], axis=1)

    loglik = logZ - np.log(N) # Log-likelihood
    return smc_res(alpha_filt, particles, W, logW=logW, N_eff=N_eff)
```

Q3: Choose one of the simulated trajectories and run the SIS algorithm using $N = 100$ particles. Show plots comparing the filter means from the SIS algorithm with the underlying truth of the Infected, Exposed and Recovered.

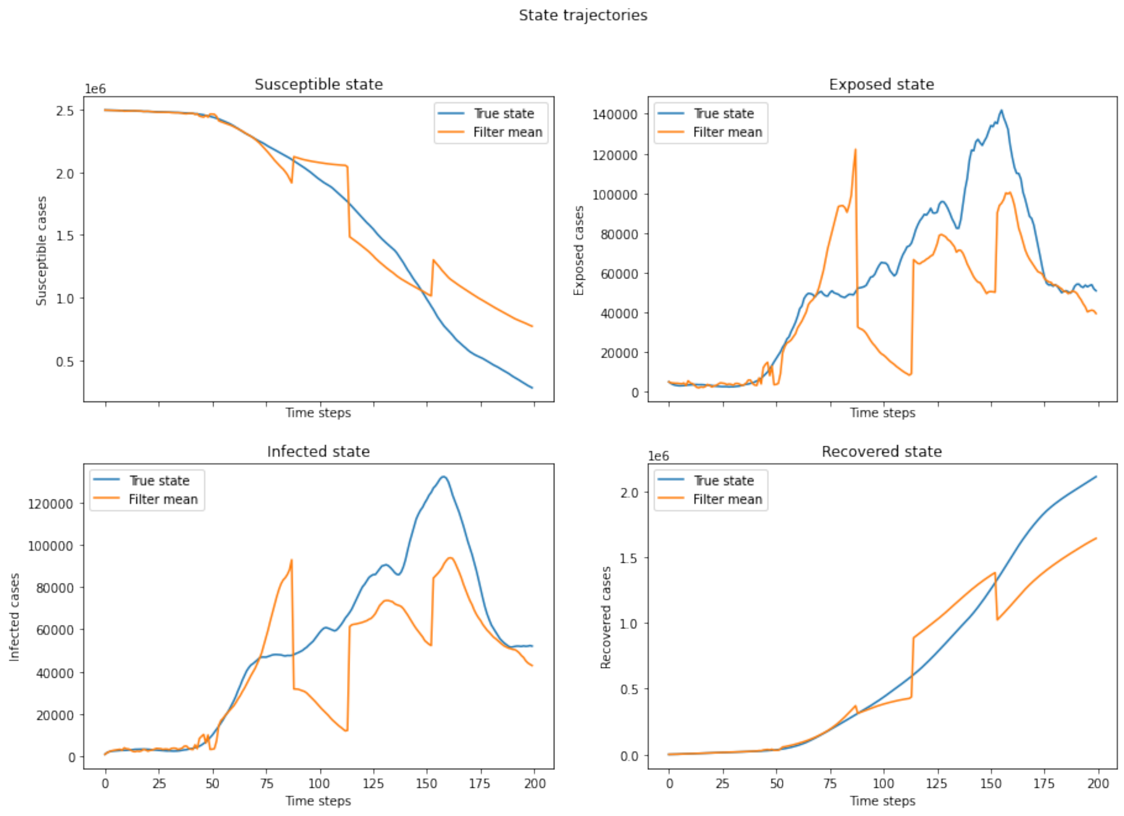
Also show a plot of how the ESS behaves over the run.

(hint: In the model we use the S, E, I as states, but S will be much larger than the others. To calculate R, note that S + E + I + R = Population)

[9]:
```
# Choose a simulated trajectory and run SIS using 100 particles
selected_trajectory = 4
sis = sis_filter(model, y=y_sim[0, selected_trajectory, :], N=100)
```

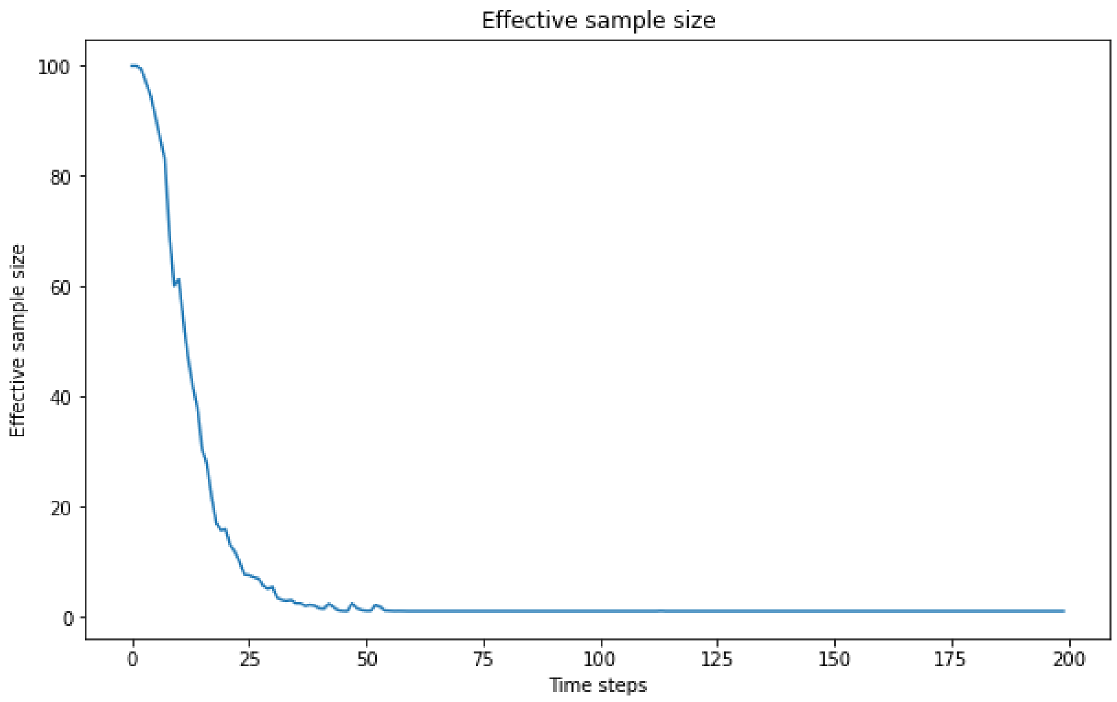
[10]:
```
# Plot the trajectories of filter means from SIS algorithm and the true␣
↪(simulated) states
fig, ax = plt.subplots(nrows=2, ncols=2, sharex='all', figsize=(15,10))
fig.suptitle('State trajectories')

# Recovered state
recovered = population_size - np.sum(alpha_sim[:model.d-1, selected_trajectory,␣
↪:], axis=0)
recovered_filter = population_size - np.sum(sis.alpha_filt[:model.d-1, 0, :],␣
↪axis=0)

ax[0, 0].plot(alpha_sim[0, selected_trajectory, :], label='True state')
ax[0, 0].plot(sis.alpha_filt[0, 0, :], label='Filter mean')
ax[0, 0].set(xlabel='Time steps', ylabel='Susceptible cases',␣
↪title='Susceptible state')
ax[0, 0].legend()

ax[0, 1].plot(alpha_sim[1, selected_trajectory, :], label='True state')
ax[0, 1].plot(sis.alpha_filt[1, 0, :], label='Filter mean')
ax[0, 1].set(xlabel='Time steps', ylabel='Exposed cases', title='Exposed state')
ax[0, 1].legend()

ax[1, 0].plot(alpha_sim[2, selected_trajectory, :], label='True state')
ax[1, 0].plot(sis.alpha_filt[2, 0, :], label='Filter mean')
ax[1, 0].set(xlabel='Time steps', ylabel='Infected cases', title='Infected␣
↪state')
ax[1, 0].legend()

ax[1, 1].plot(recovered, label='True state')
ax[1, 1].plot(recovered_filter, label='Filter mean')
ax[1, 1].set(xlabel='Time steps', ylabel='Recovered cases', title='Recovered␣
↪state')
ax[1, 1].legend()
```

```
[10]: <matplotlib.legend.Legend at 0x2a0be4a6b88>
```



```
[11]: # Plot of effective sample size
      plt.plot(sis.N_eff)
      plt.title('Effective sample size')
      plt.xlabel('Time steps')
      plt.ylabel('Effective sample size')
```

```
[11]: Text(0, 0.5, 'Effective sample size')
```



We observe from the plot of state trajectories that the estimated filtered distributions of the states using the SIS algorithm are good for a few initial time steps but becomes quite erratic as the time steps progresses into the future. This can be explained as soon as we look at the effective sample size (ESS) plot. The ESS drops to 1 somewhere around the 30th time step i.e. from t=30, only a single sample (particle) contributes to the estimated value of the filter distribution. This is due to the weight degeneracy problem which occurs in SIS due to repeated multiplication of weights. Hence the obtained filter estimates of the states are not reliable.

1.4 3.4 Sequential Importance Sampling with Resampling

Pick the same simulated trajectory as for the previous section.

Q4: Implement the **Sequential Importance Sampling with Resampling** or **Bootstrap Particle Filter** by completing the code below.

[12]:
```
def bpf(model, y, numParticles):
    d = model.d
    n = len(y)
    N = numParticles

    # Allocate memory
    particles = np.zeros((d, N, n), dtype = float)  # All generated particles
    logW = np.zeros((1, N, n))  # Unnormalized log-weight
    W = np.zeros((1, N, n))     # Normalized weight
    alpha_filt = np.zeros((d, 1, n))  # Store filter mean
    N_eff = np.zeros(n)  # Efficient number of particles
    logZ = 0.            # Log-likelihood estimate

    # Filter loop
    for t in range(n):
        # Sample from "bootstrap proposal"
        if t == 0:  # Initialize from prior
            particles[:, :, 0] = model.sample_state(alpha0=None, N=N)

        else:  # Resample and propagate according to dynamics
            ind = np.random.choice(N, N, replace=True, p=W[0, :, t-1])
            resampled_particles = particles[:, ind, t-1]
            particles[:, :, t] = model.sample_state(alpha0=resampled_particles,␣
 ↪N=N)

        # Compute weights
        logW[0, :, t] = model.log_lik(y=y[t], alpha=particles[:, :, t])
        W[0, :, t], logZ_now = exp_norm(logW[0, :, t])
        logZ +=  logZ_now - np.log(N) # Update log-likelihood estimate
        N_eff[t] = ESS(W[0, :, t])

        # Compute filter estimates
        alpha_filt[:, 0, t] =  np.sum(W[0, :, t]*particles[:, :, t], axis=1)

    return smc_res(alpha_filt, particles, W, N_eff = N_eff, logZ = logZ)
```

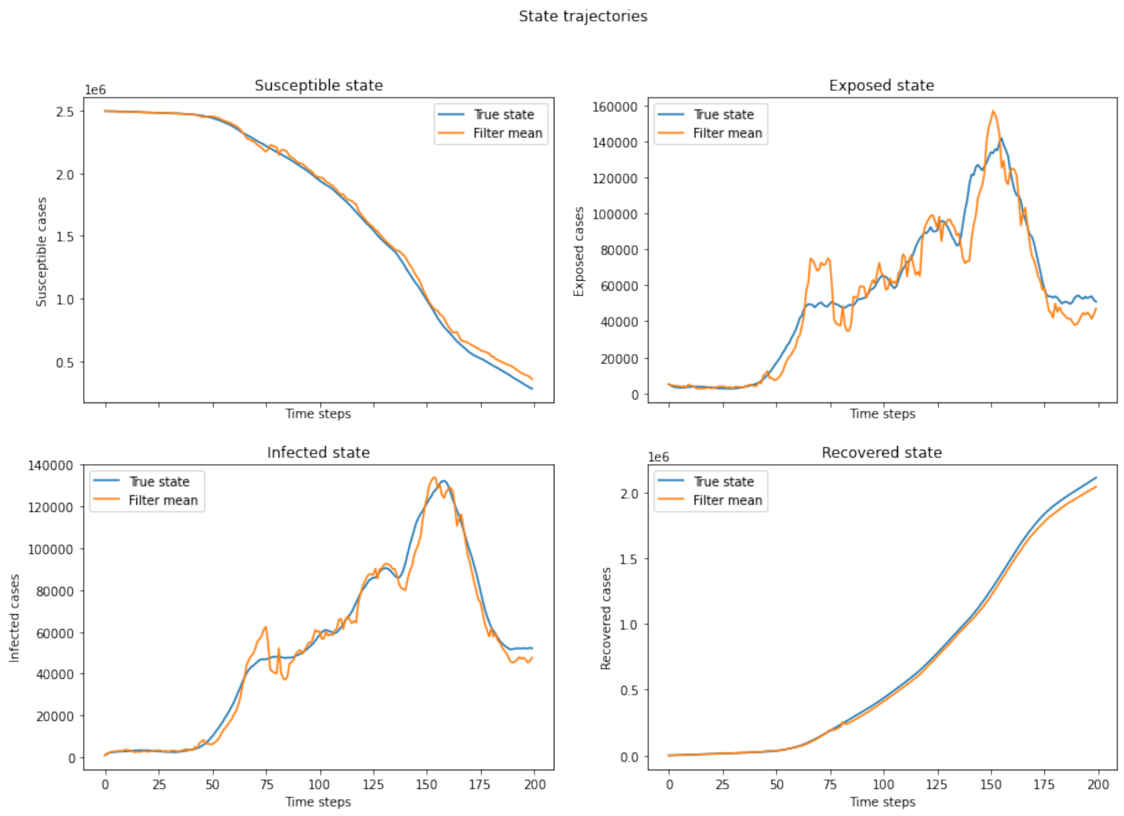
Q5: Use the same simulated trajectory as above and run the BPF algorithm using $N = 100$ particles. Show plots comparing the filter means from the Bootstrap Particle Filter algorithm with the underlying truth of the Infected, Exposed and Recovered. Also show a plot of how the ESS behaves over the run. Compare this with the results from the SIS algorithm.

[13]:
```
# Choose a simulated trajectory and run BPF using 100 particles
sisr = bpf(model, y=y_sim[0, selected_trajectory, :], numParticles=100)
```

[14]:
```
# Plot the trajectories of filter means from SISR algorithm and the true␣
 ↪(simulated) states
fig, ax = plt.subplots(nrows=2, ncols=2, sharex='all', figsize=(15,10))
fig.suptitle('State trajectories')

# Recovered state
recovered = population_size - np.sum(alpha_sim[:model.d-1, selected_trajectory,␣
 ↪:], axis=0)
recovered_filter = population_size - np.sum(sisr.alpha_filt[:model.d-1, 0, :],␣
 ↪axis=0)

ax[0, 0].plot(alpha_sim[0, selected_trajectory, :], label='True state')
ax[0, 0].plot(sisr.alpha_filt[0, 0, :], label='Filter mean')
```

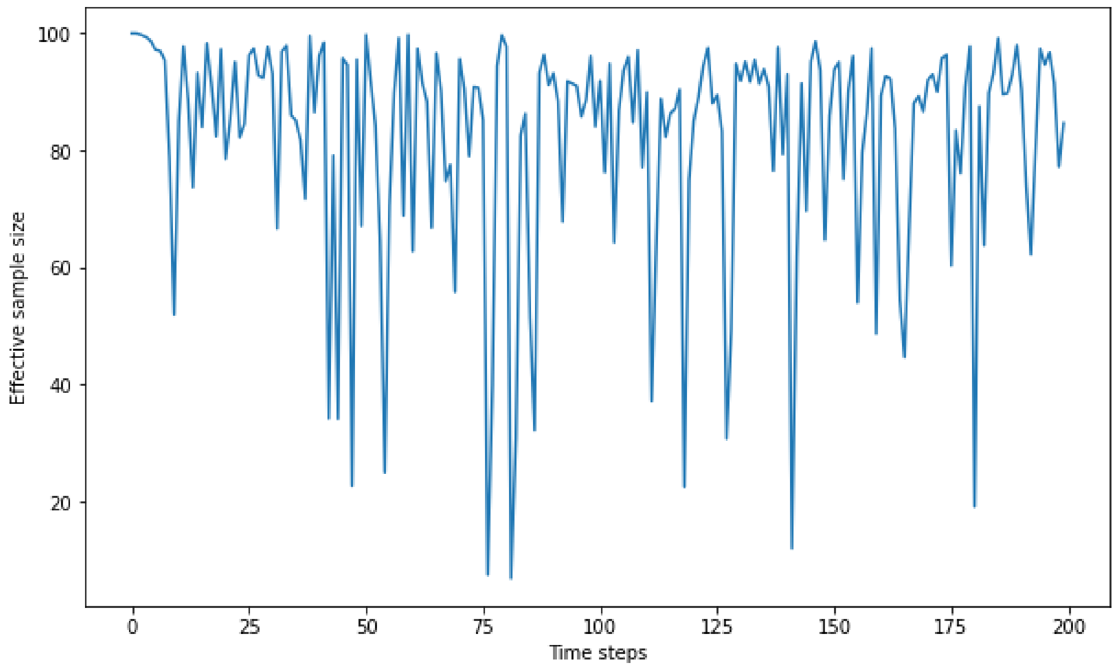
```
ax[0, 0].set(xlabel='Time steps', ylabel='Susceptible cases',␣
 ↪title='Susceptible state')
ax[0, 0].legend()

ax[0, 1].plot(alpha_sim[1, selected_trajectory, :], label='True state')
ax[0, 1].plot(sisr.alpha_filt[1, 0, :], label='Filter mean')
ax[0, 1].set(xlabel='Time steps', ylabel='Exposed cases', title='Exposed state')
ax[0, 1].legend()

ax[1, 0].plot(alpha_sim[2, selected_trajectory, :], label='True state')
ax[1, 0].plot(sisr.alpha_filt[2, 0, :], label='Filter mean')
ax[1, 0].set(xlabel='Time steps', ylabel='Infected cases', title='Infected␣
 ↪state')
ax[1, 0].legend()

ax[1, 1].plot(recovered, label='True state')
ax[1, 1].plot(recovered_filter, label='Filter mean')
ax[1, 1].set(xlabel='Time steps', ylabel='Recovered cases', title='Recovered␣
 ↪state')
ax[1, 1].legend()
```

```
[14]: <matplotlib.legend.Legend at 0x2a0be773ec8>
```



```
[15]: # Plot of effective sample size
      plt.plot(sisr.N_eff)
      plt.xlabel('Time steps')
      plt.ylabel('Effective sample size')
```

```
[15]: Text(0, 0.5, 'Effective sample size')
```



We observe from the state plot that the bootstrap filter estimates of the states are much more accurate than the SIS particle filter estimates. And understandably, the number of samples (particles) used to estimate the filter distribution is consistently high with only a few steps where the ESS drops below 50. Hence, the bootstrap estimates are more reliable.

1.5 3.5 Estimating the data likelihood and learning a model parameter

In this section we consider the real data and learning the model using this data. For simplicity we will only look at the problem of estimating the ρ parameter and assume that others are fixed.

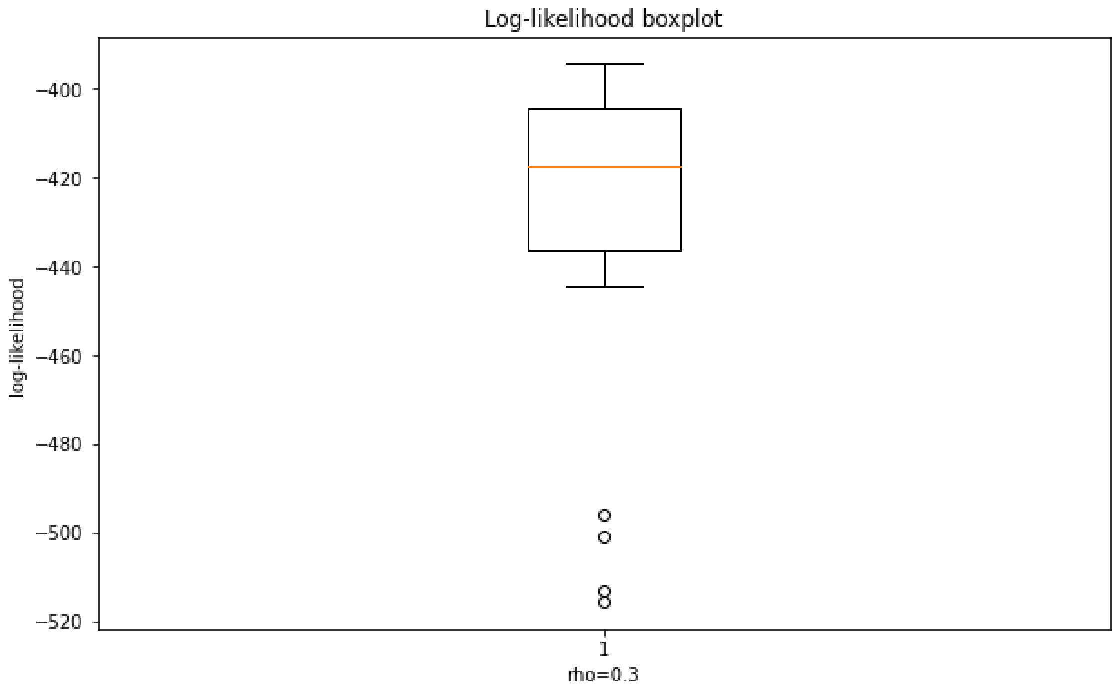
You are more than welcome to also study the other parameters.

Before we begin to tweak the parameters we run the particle filter using the current parameter values to get a benchmark on the log-likelihood.

Q6: Run the bootstrap particle filter using $N = 200$ particles on the real dataset and calculate the log-likelihood. Rerun the algorithm 20 times and show a box-plot of the log-likelihood.

```
[16]: # Run bootstrap particle filter multiple times on the real dataset
      log_lik = np.zeros(20)
      for i in range(20):
          sisr = bpf(model, y=y_sthlm, numParticles=200)
          log_lik[i] = sisr.logZ

      # Box-plot of log-likelihood estimates for actual data in different runs of SISR
      plt.boxplot(log_lik)
      plt.title('Log-likelihood boxplot')
      plt.xlabel('rho='+str(rho))
      plt.ylabel('log-likelihood')
```

```
[16]: Text(0, 0.5, 'log-likelihood')
```



We can see that we obtain a range of log-likelihood values for different runs of the bootstrap particle filter on the same data (the real data). This variance is due to the stochastic nature of the SISR algorithm. There is randomness involved in the resampling and the propagation steps which adds variance to the log-likelihood estimates as well as the filter estimates. The median log-likelihood estimates calculated under the current parameter values is around -415 which can be used as a benchmark in parameter estimation.

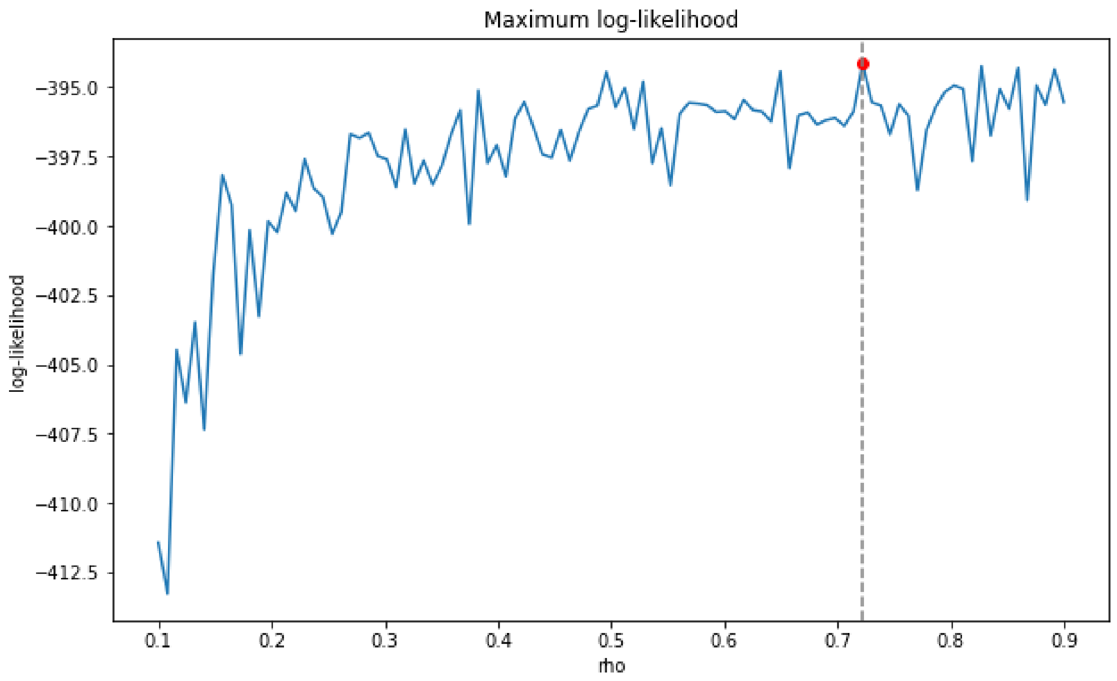
Q7: Make a grid of the ρ parameter in the interval $[0.1, 0.9]$. Use the bootstrap particle filter to calculate the log-likelihood for each value. Run the bootstrap particle filter using $N = 1000$ multiple times (20) per value and use the average as your estimate of the log-likelihood. Plot the log-likelihood function and mark the maximal value.

(hint: use np.logspace to create a grid of parameter values)

[17]:
```
# Grid of parameter rho
rho_grid = np.linspace(start=0.1, stop=0.9, num=100)
log_lik  = np.zeros((len(rho_grid), 20))

for i in range(len(rho_grid)):
    # Set the rho parameter value in the model
    model.set_param(rho=rho_grid[i])

    for j in range(20):
        # Run the bootsrtap particle filter with 1000 particles for each value␣
 ↪of rho
        sisr = bpf(model, y=y_sthlm, numParticles=1000)
        log_lik[i, j] = sisr.logZ

# Average of the partile filter estimates of the log-likelihood
loglike_estimate = np.mean(log_lik, axis=1)
```

[18]:
```
# Maximum log-likelihood and optimal rho
max_llik = max(loglike_estimate)
max_rho  = rho_grid[loglike_estimate == max_llik]

# Average log-likelihood estimates for different rho
plt.plot(rho_grid, loglike_estimate)
plt.plot(max_rho, max_llik, marker='o', color='red')
plt.axvline(x=max_rho, color='gray', ls='--')
plt.title('Maximum log-likelihood')
plt.xlabel('rho')
plt.ylabel('log-likelihood')
```

[18]:
```
Text(0, 0.5, 'log-likelihood')
```



We get a much better log-likelihood value for ρ=0.72 compared to the benchmark. We can choose the optimal value for parameter ρ to be approximately 0.72 where rho is the probability of exposure for each encounter with an infectious individual. (i.e. high risk of exposure from encounters)

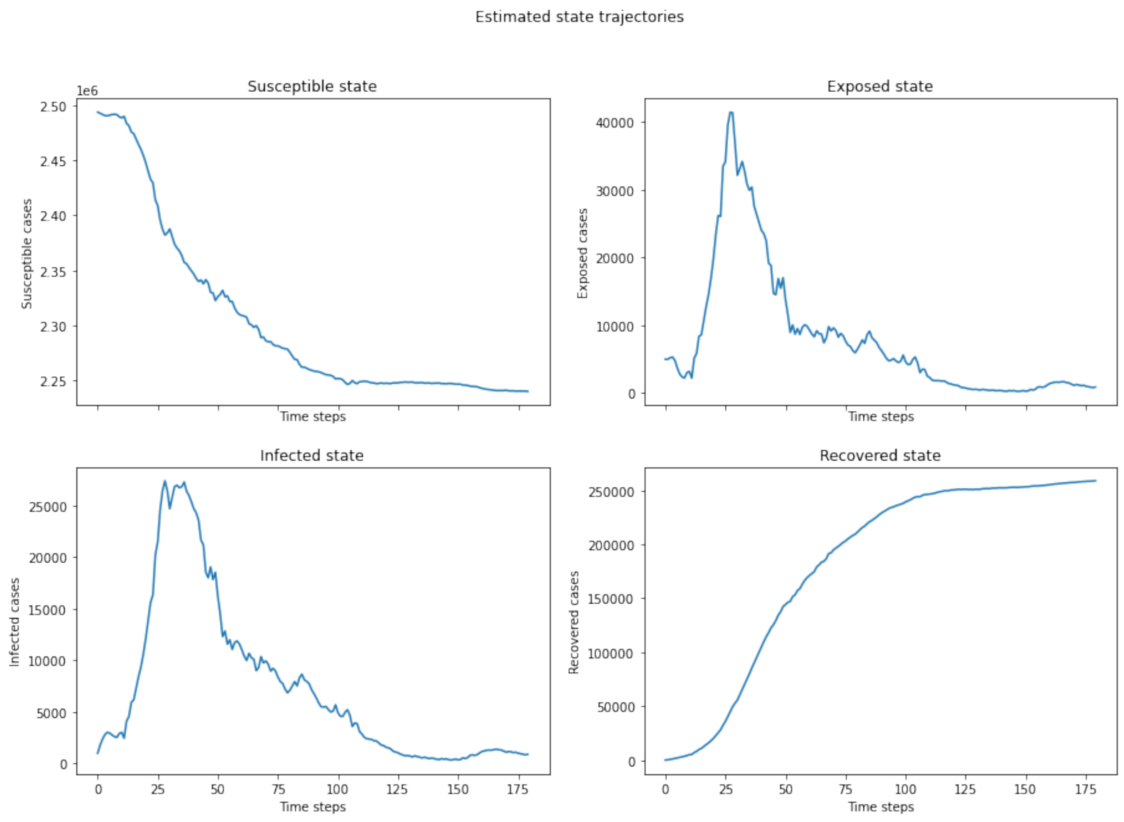
Q8: Run the bootstrap particle filter on the full dataset with the optimal ρ value. Present a plot of the estimated Infected, Exposed and Recovered states.

[19]:
```
# Set the optimal rho parameter value in the SEIR model
rho_opt = max_rho
model.set_param(rho=rho_opt)

# Run the boostrap particle filter on full dataset with optimal rho parameter␣
↪value
sisr = bpf(model, y=y_sthlm, numParticles=200)
```

[20]:
```
# Plot the trajectories of filter means from SISR algorithm and the true␣
↪(simulated) states
fig, ax = plt.subplots(nrows=2, ncols=2, sharex='all', figsize=(15,10))
fig.suptitle('Estimated state trajectories')

# Recovered state
recovered = population_size - np.sum(sisr.alpha_filt[:model.d-1, 0, :], axis=0)

ax[0, 0].plot(sisr.alpha_filt[0, 0, :])
```

```
ax[0, 0].set(xlabel='Time steps', ylabel='Susceptible cases',␣
 ↪title='Susceptible state')

ax[0, 1].plot(sisr.alpha_filt[1, 0, :])
ax[0, 1].set(xlabel='Time steps', ylabel='Exposed cases', title='Exposed state')

ax[1, 0].plot(sisr.alpha_filt[2, 0, :])
ax[1, 0].set(xlabel='Time steps', ylabel='Infected cases', title='Infected␣
 ↪state')

ax[1, 1].plot(recovered)
ax[1, 1].set(xlabel='Time steps', ylabel='Recovered cases', title='Recovered␣
 ↪state')
```

```
[20]: [Text(0.5, 0, 'Time steps'),
       Text(0, 0.5, 'Recovered cases'),
       Text(0.5, 1.0, 'Recovered state')]
```


Estimated state trajectories
Susceptible state
1e6
Susceptible cases
Time steps
Exposed state
Exposed cases
Time steps
Infected state
Infected cases
Time steps
Recovered state
Recovered cases
Time steps