Group_tssl_lab3

October 9, 2022

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.

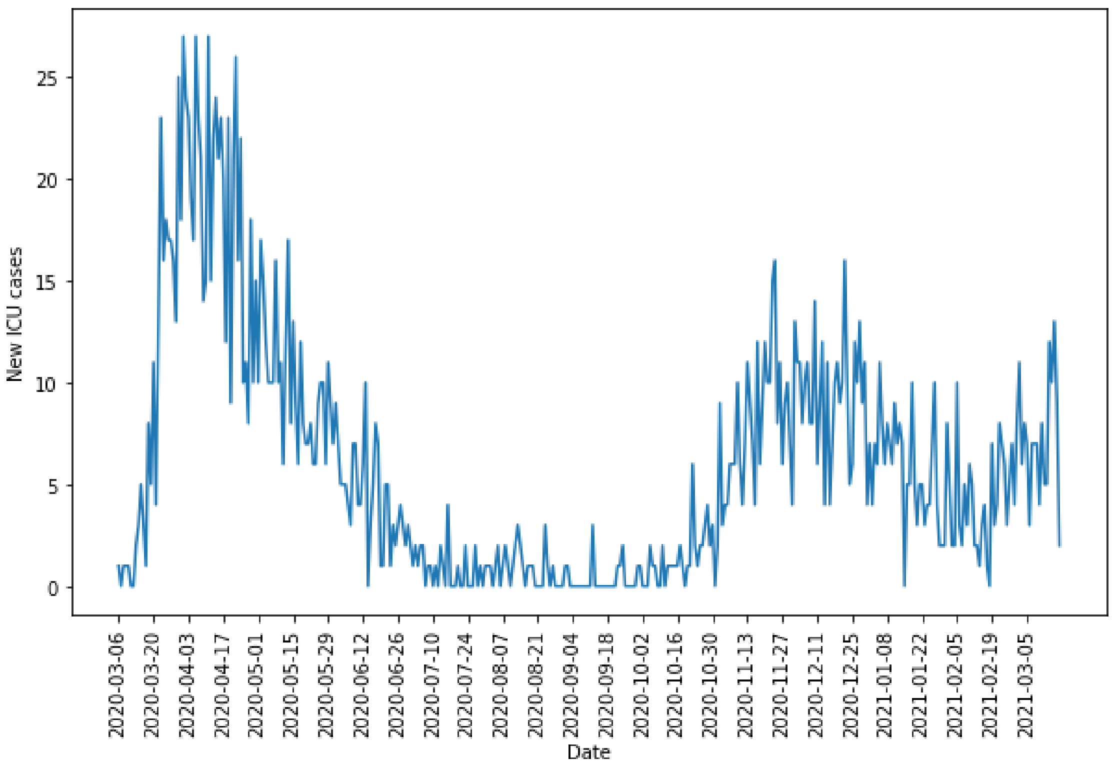
We load a few packages that are useful for solving this lab assignment.

[26]:
```
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 2020 to March 2021. As always, we start by loading and plotting the data.

[27]:
```
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, 14), u_sthlm[::14], rotation = 90)  # Show only one␣
 ↪tick per 2 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: Discrete data; According to the above figure, we can find that this is an asymmetric bimodal distribution, which is obviously inappropriate for a Gaussian likelihood model.

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.

[28]:
```python
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.

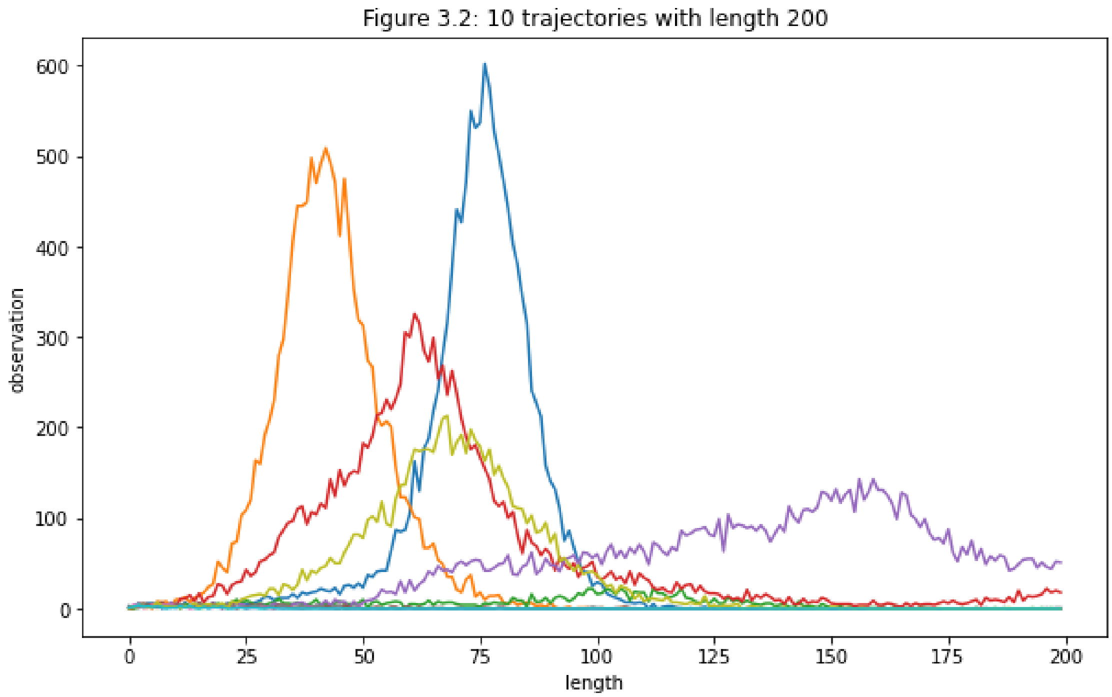
(hint: The SEIR class has a simulate method)

A1: In the real data we have two peaks. But all trajectories shows only one peak value. So data can not been generated using this model.

[29]:
```
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.
```

[30]:
```
np.random.seed(0)
alpha1, y1 = model.simulate(T=200, N=10) # time steps/length = 200; num of␣
 ↪trajectories = 10
i = 0
while(i<10):
    plt.plot(y1[0,i,])
    plt.xlabel("length")
    plt.ylabel("observation")
    plt.title('Figure 3.2: 10 trajectories with length 200')
    i = i+1
```



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)

[32]:
```
from tssltools_lab3 import smc_res

def exp_norm(logwgt):
    """
```

```python
    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/!
    """
    # PP EXERCISE 3 Q1
    log_max = max(logwgt)
    wgt1 = logwgt - log_max
    exp_wgt = np.exp(wgt1)
    wgt2 = exp_wgt/sum(exp_wgt)

    logZ = np.log(sum(np.exp(wgt1)))+log_max
    return wgt2, 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.
    """
    # L8 P20
    ess = (sum(wgt)**2)/sum(wgt**2)
    return ess

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

```python
    # 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(N=N) # Initialize from␣
↪p(alpha_1)
            logW[0, :, 0] = model.log_lik(y[0],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[t],particles[:,:,t])+logW[0,:,t-1]␣
↪# Update weights

        # Normalize the importance weights and compute N_eff
        W[0, :, t], _ = 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)

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

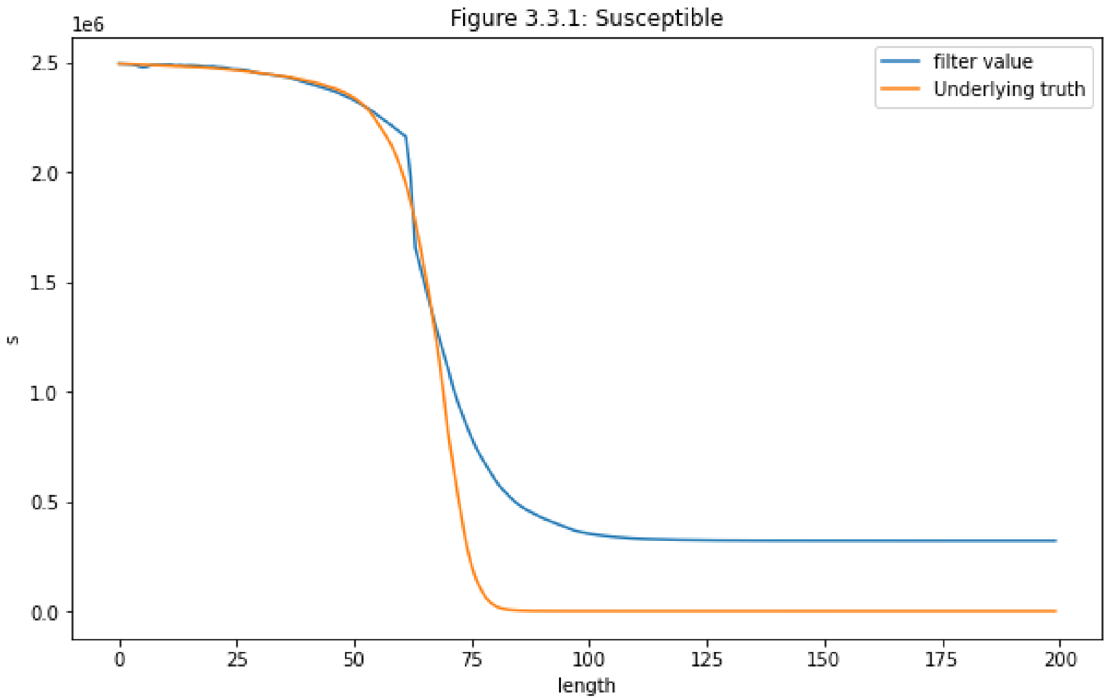
Q3: Choose one of the simulated trajectories and run the SIS algorithm using $N = 100$ particles. Show plots comparing the filter means from the SIS algorithm with the underlying truth of the Infected, Exposed and Recovered.

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

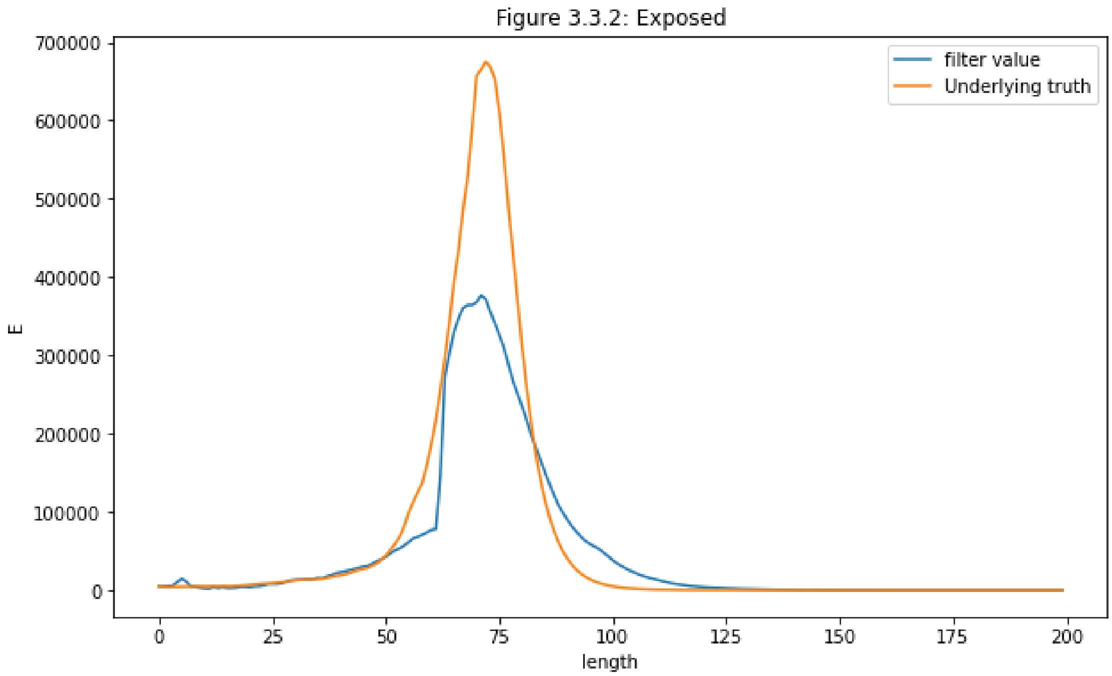
(hint: In the model we use the S, E, I as states, but S will be much larger than the others. To calculate R, note that $S + E + I + R = Population$)

[34]:
```python
# np.random.seed(0)
# i = np.random.randint(0,9,1)[0]
tra = y1[0, 0, ] # here we choose No.1 trajectories
filter_res = sis_filter(model, tra, N = 100)

plt.plot(filter_res.alpha_filt[0,0,:],label ='filter value')
plt.plot(alpha1[0,0,:],label ='Underlying truth')
```

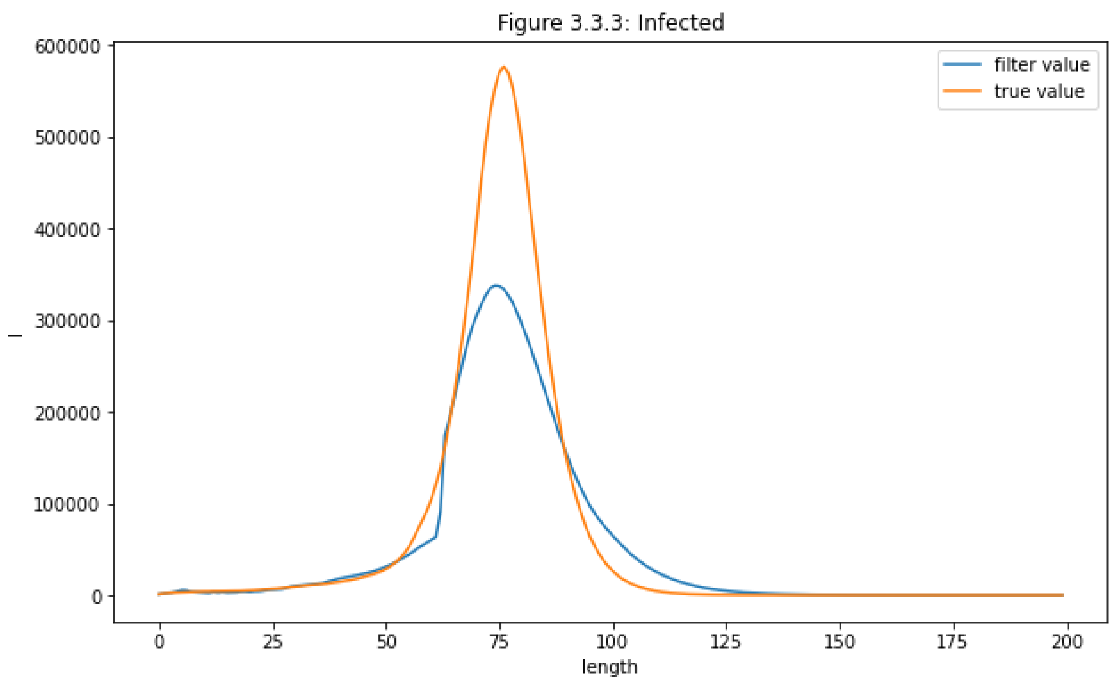
```python
plt.ylabel('s')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.3.1: Susceptible')
plt.show()
```



[35]:
```python
plt.plot(filter_res.alpha_filt[1,0,:],label ='filter value')
plt.plot(alpha1[1,0,:],label ='Underlying truth')
plt.ylabel('E')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.3.2: Exposed')
plt.show()
```

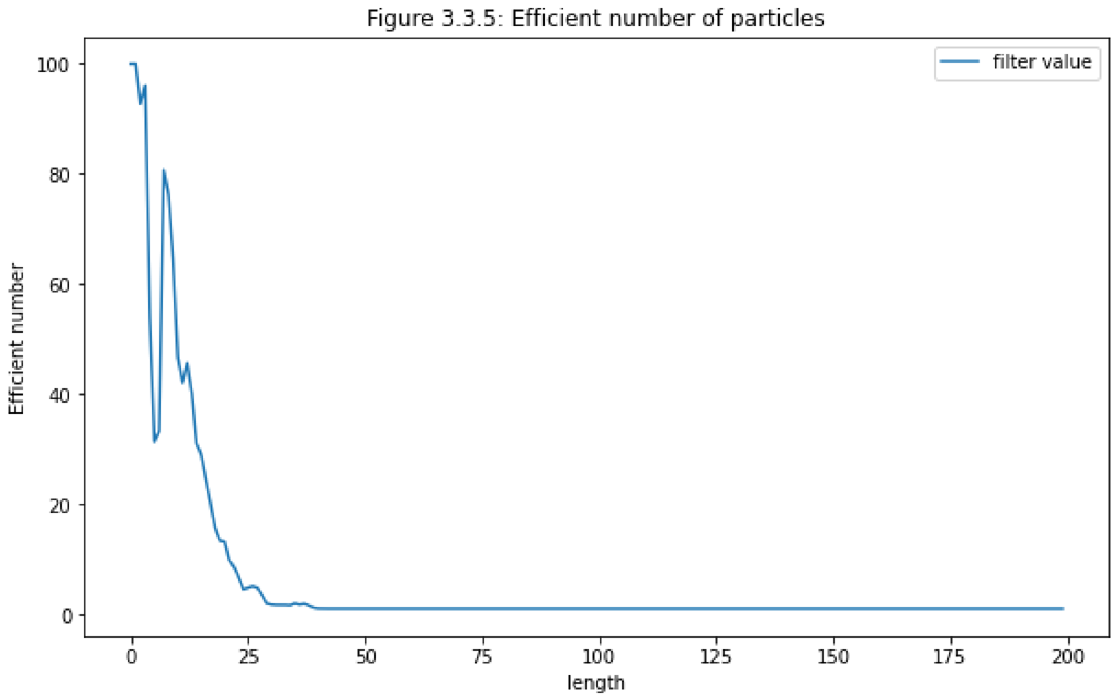


[36]:
```
plt.plot(filter_res.alpha_filt[2,0,:],label ='filter value')
plt.plot(alpha1[2,0,:],label ='true value')
plt.ylabel('I')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.3.3: Infected')
plt.show()
```



[37]:
```
plt.plot(filter_res.alpha_filt[3,0,:],label ='filter value')
plt.plot(alpha1[3,0,:],label ='true value')
plt.ylabel('R')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.3.4: Recovered')
plt.show()
```

[38]:
```
plt.plot(filter_res.N_eff,label ='filter value')
plt.ylabel('Efficient number')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.3.5: Efficient number of particles')
plt.show()
```



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.

[39]:
```
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(N=N)
```

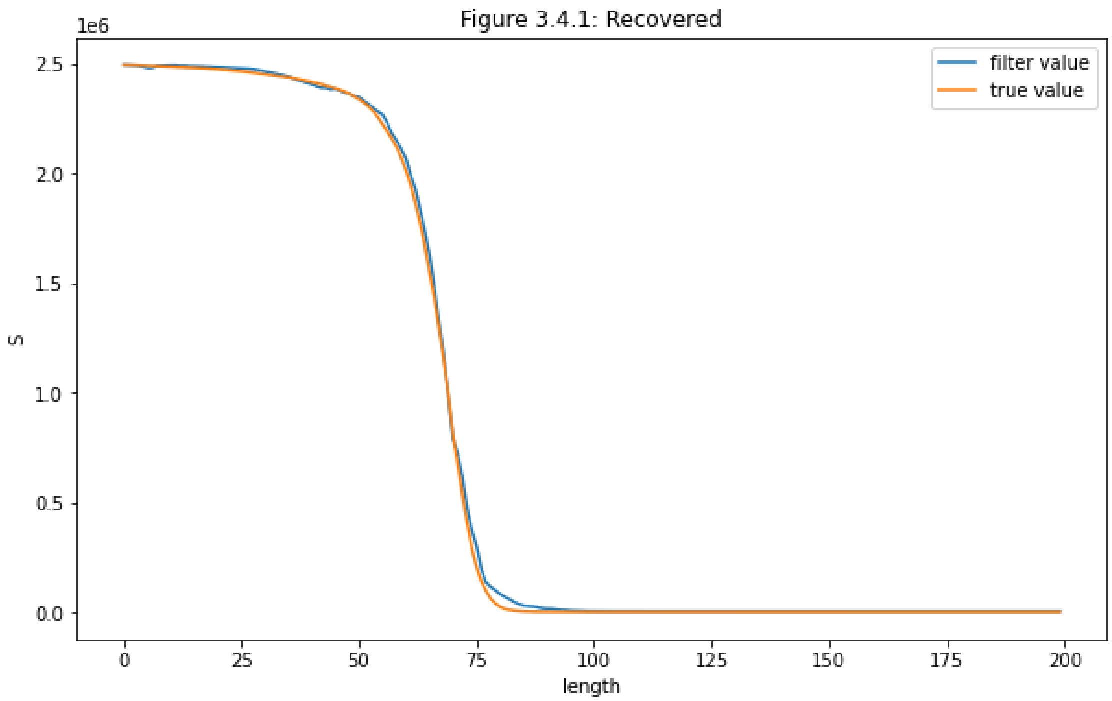
```
        else:  # Resample and propagate according to dynamics
            ind = np.random.choice(N, N, replace=True, p=W[0, :, t-1])
            resampled_particles = particles[:, ind, t-1]
            particles[:, :, t] = model.sample_state(alpha0 =resampled_particles␣
↪,N = N)

        # Compute weights
        logW[0, :, t] = model.log_lik(y[t],alpha = particles[:, :, t])
        W[0, :, t], logZ_now = exp_norm(logW[0, :, t])
        # l(y1:n) = sum(ct)+log(capital_omiga_t_tilde)-log(N)

        # logZ_now = ct+log()
        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)
```

comments about updating logZ
From the pp exercise 3
$l(y_{1:n}) = \sum_{t=1}^{n}(C_t + log(\tilde{\Omega}_t) - log(N))$
$logZ_{now} = C_t + log(\tilde{\Omega}_t)$
So updating logZ need to substract logN

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.

[40]:
```
filter_res = bpf(model,tra,100)

plt.plot(filter_res.alpha_filt[0,0,:],label ='filter value')
plt.plot(alpha1[0,0,:],label ='true value')
plt.ylabel('S')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.4.1: Recovered')
plt.show()

plt.plot(filter_res.alpha_filt[1,0,:],label ='filter value')
plt.plot(alpha1[1,0,:],label ='true value')
plt.ylabel('E')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.4.2: Exposed')
```

```
plt.show()

plt.plot(filter_res.alpha_filt[2,0,:],label ='filter value')
plt.plot(alpha1[2,0,:],label ='true value')
plt.ylabel('I')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.4.3: Infected')
plt.show()

plt.plot(filter_res.alpha_filt[3,0,:],label ='filter value')
plt.plot(alpha1[3,0,:],label ='true value')
plt.ylabel('R')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.3.4: Recovered')
plt.show()

plt.plot(filter_res.N_eff,label ='filter value')
plt.ylabel('EFF')
plt.xlabel('length')
plt.legend()
plt.title('Figure 3.4.5: Efficient number of particles')
plt.show()
```



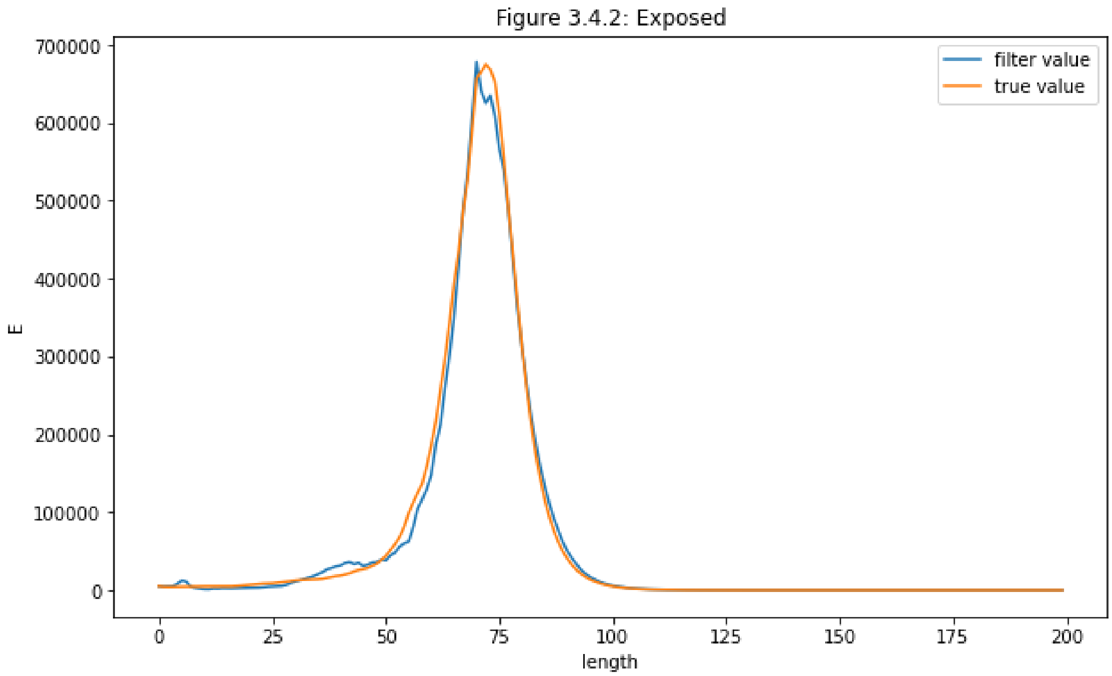


Figure 3.4.2: Exposed
filter value
true value
E
length
0
100000
200000
300000
400000
500000
600000
700000
0
25
50
75
100
125
150
175
200

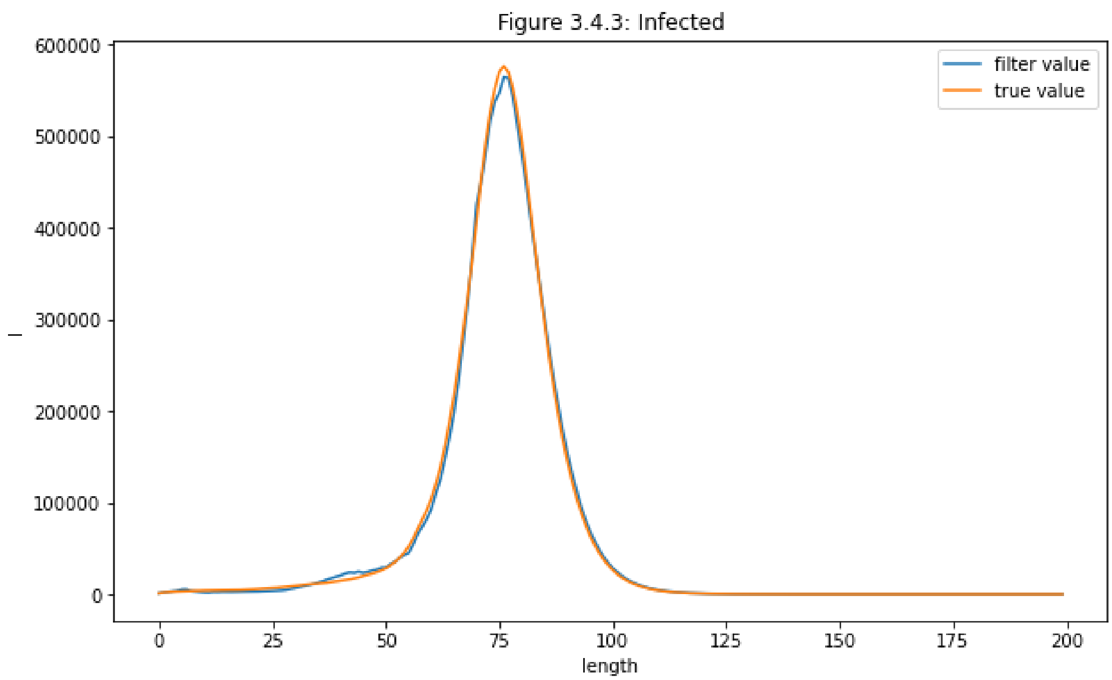
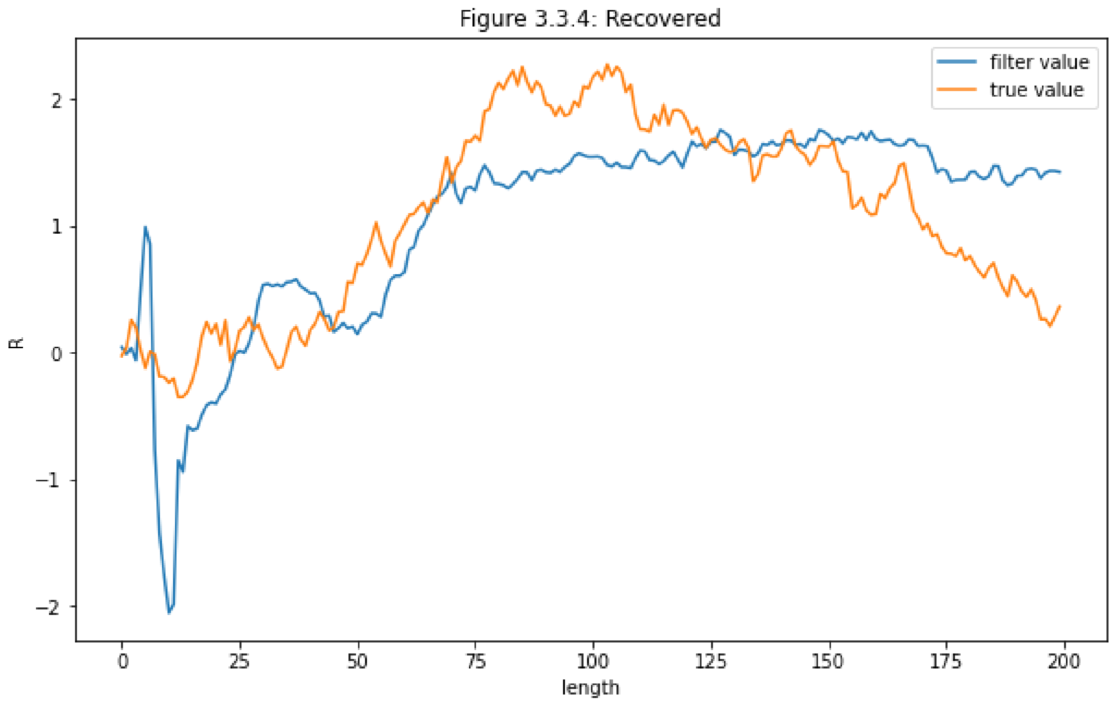
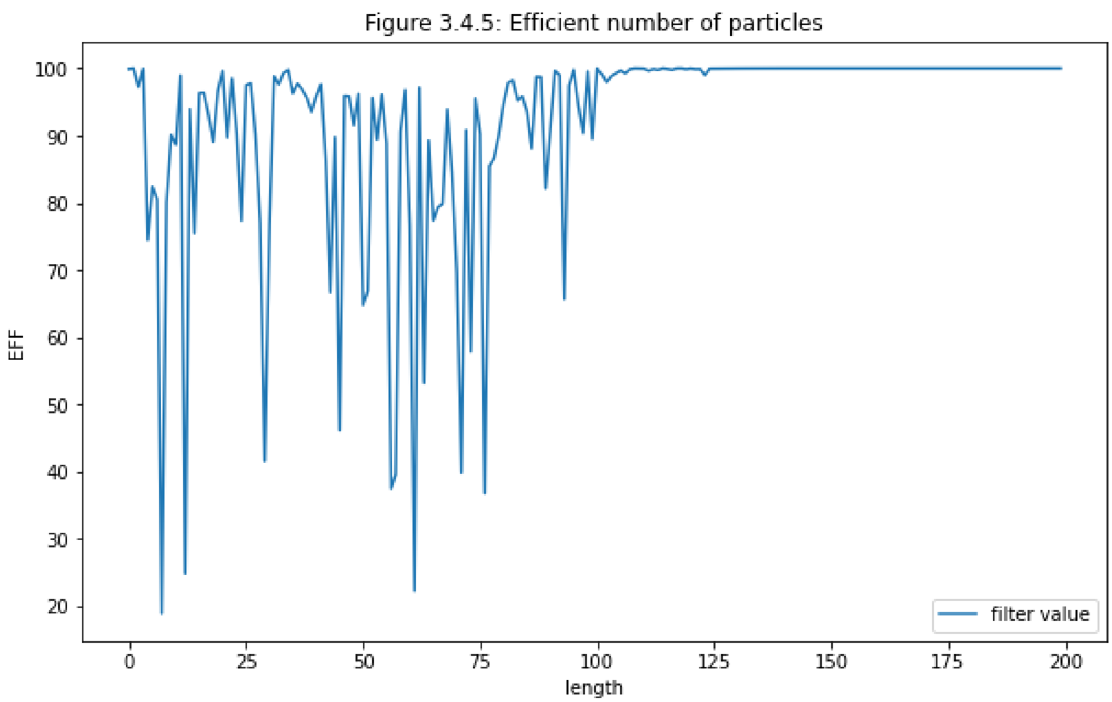


Figure 3.4.3: Infected
filter value
true value
I
length
0
100000
200000
300000
400000
500000
600000
0
25
50
75
100
125
150
175
200





A5: We can see ESS doesn't decrease to zero when time step increases wieh resampling method.

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.

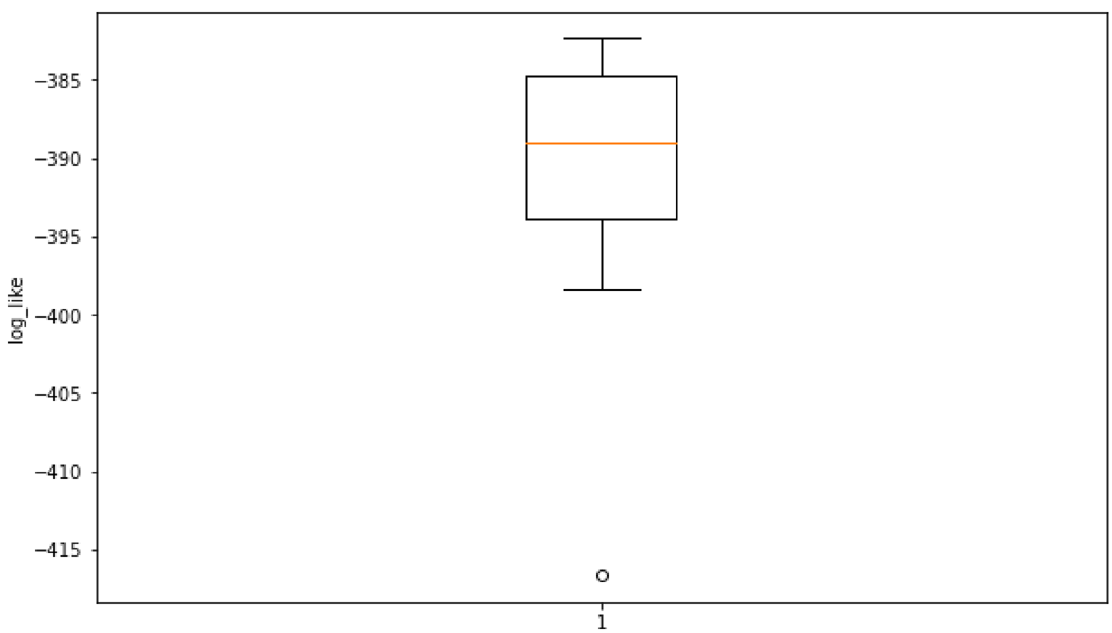
You are more than welcome to also study the other parameters.

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.

[43]:
```
loglike = np.zeros(20)
for i in range(20):
    filter_res = bpf(model,tra,200)
    loglike[i] = filter_res.logZ

plt.boxplot(loglike)
plt.ylabel('log_like')
print(np.mean(loglike))
```

```
-390.68636316443
```

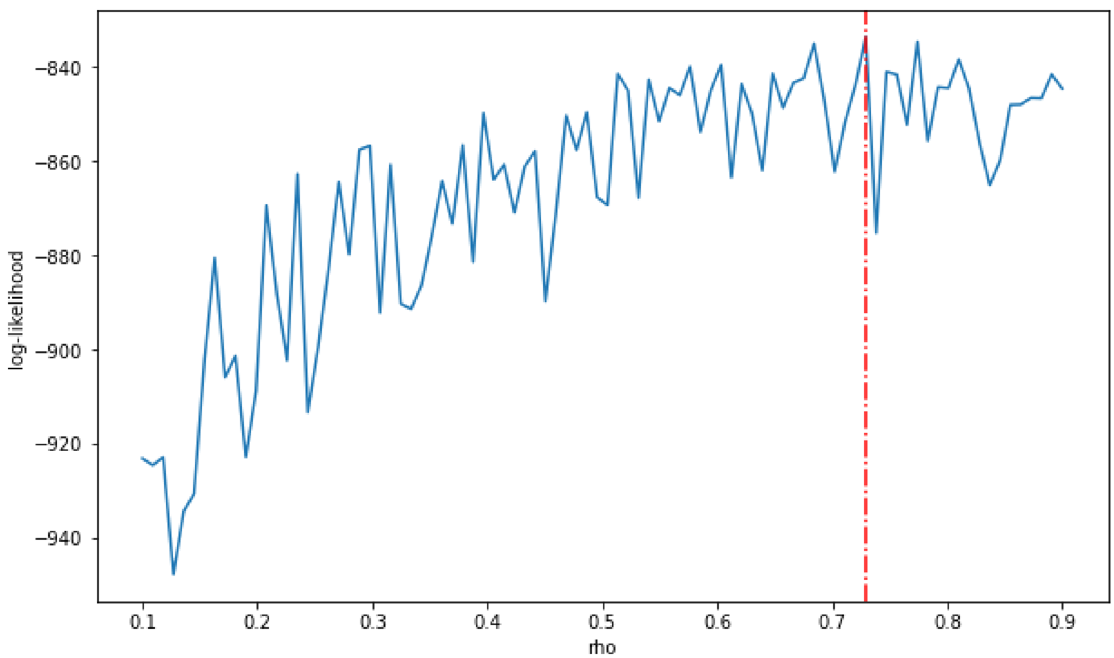


Q7: Make a grid of the ρ parameter in the interval $[0.1, 0.9]$. Use the bootstrap particle filter to calculate the log-likelihood for each value. Run the bootstrap particle filter using $N = 200$ multiple times (at least 20) per value and use the average as your estimate of the log-likelihood. Plot the log-likelihood function and mark the maximal value.

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

[44]:
```python
grid_list = np.linspace(0.1,0.9,num = 90)
res = np.zeros(90)
for i in range(90):
    params = Param(pse, pei, pir, pic, grid_list[i], sigma_epsilon,init_mean,␣
 ↪population_size)
    model = SEIR(params)
    multiple_run = [bpf(model, y_sthlm, numParticles = 200).logZ for j in␣
 ↪range(20)]
    res[i] = np.mean(multiple_run)

plt.plot(grid_list,res)
plt.axvline(grid_list[np.argmax(res)],linestyle='-.',color = 'red')
plt.xlabel('rho')
plt.ylabel('log-likelihood')
```

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

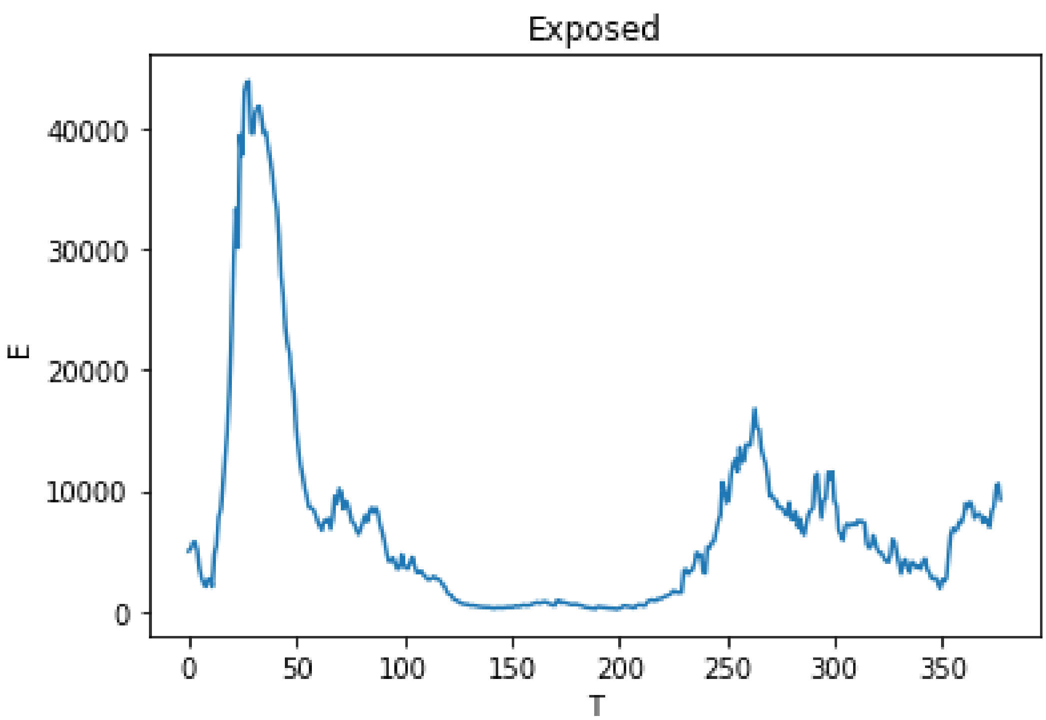


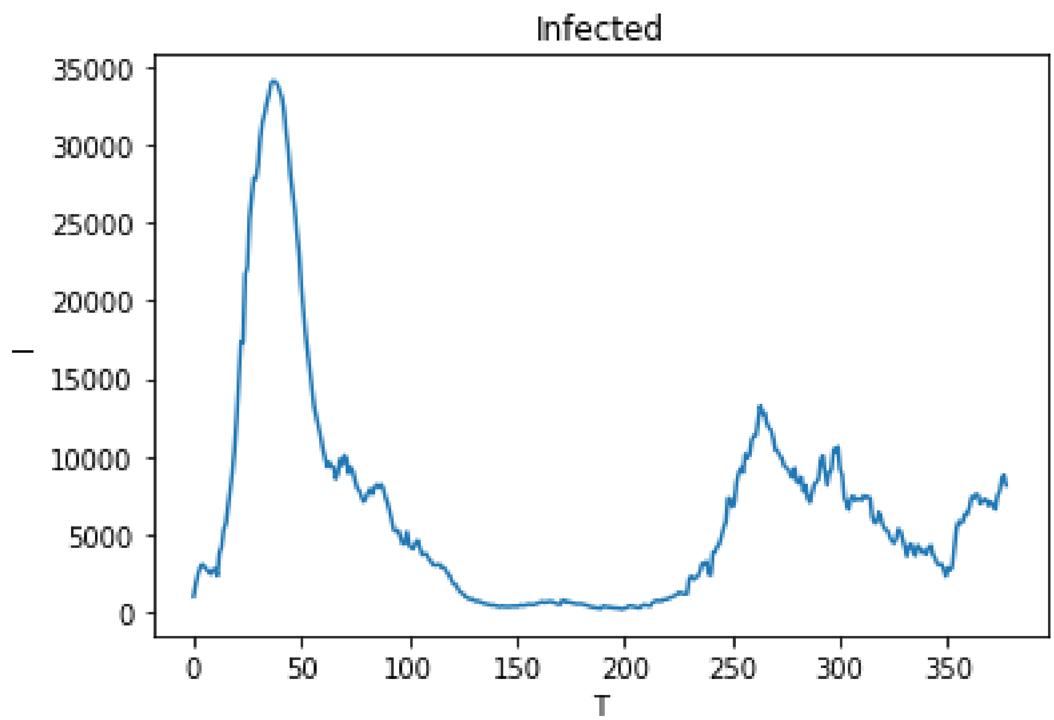
Q8: Run the bootstrap particle filter on the full dataset with the optimal ρ value. Present a plot of the estimated Infected, Exposed and Recovered states.

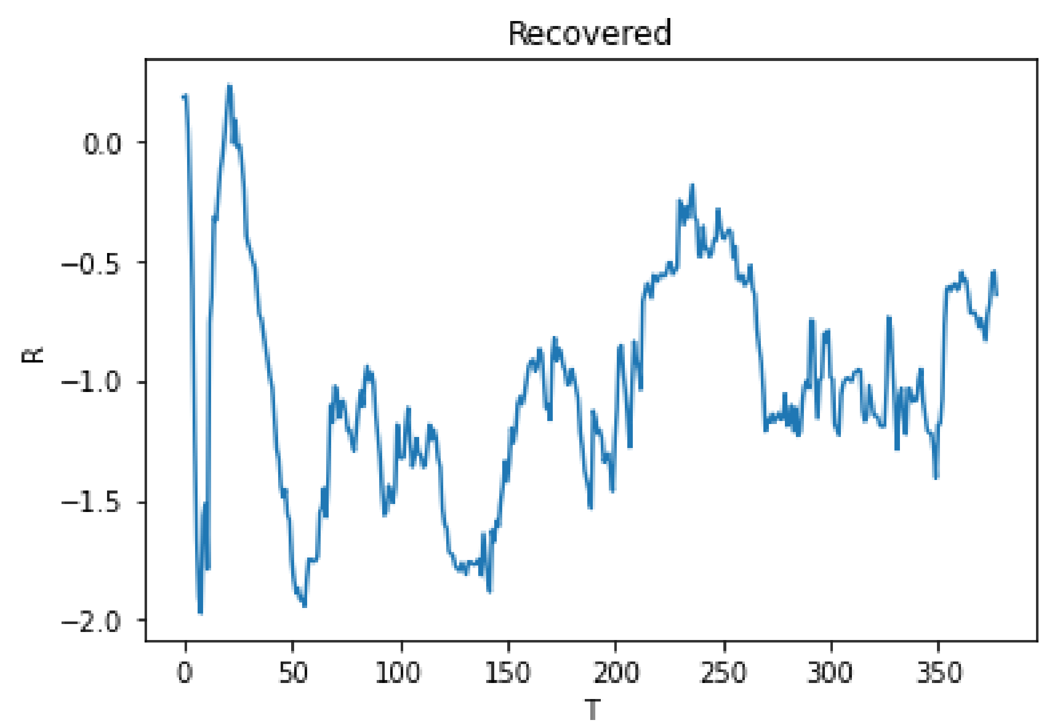
[ ]:
```python
optimal_rho = grid_list[np.argmax(res)]

params = Param(pse, pei, pir, pic, optimal_rho,sigma_epsilon, init_mean,␣
 ↪population_size)
model = SEIR(params)
```

```
res = bpf(model, y_sthlm, numParticles = 200)
```

[ ]:
```
plt.plot(res.alpha_filt[1,0,:])
plt.title('Exposed')
plt.xlabel('T')
plt.ylabel('E')
plt.show()

plt.plot(res.alpha_filt[2,0,:])
plt.title('Infected')
plt.xlabel('T')
plt.ylabel('I')
plt.show()

plt.plot(res.alpha_filt[3,0,:])
plt.title('Recovered')
plt.xlabel('T')
plt.ylabel('R')
plt.show()
```




Infected
35000
30000
25000
20000
15000
10000
5000
0
I
0
50
100
150
200
250
300
350
T


Recovered
0.0
−0.5
−1.0
−1.5
−2.0
R
0
50
100
150
200
250
300
350
T