Sequential Monte Carlo for Dynamic Linear Models

Computational Statistics Project 2

Leonardo Ruggieri

1 Introduction

Sequential Monte Carlo algorithms [5] have proven to be successful when the object of interest is the filtering distribution of a state-space model. In this project, a particle filtering algorithm for Dynamic Linear Models [4] is implemented in Python. Since the implementation involves a Gaussian Dynamic Linear Model, the results are compared with those obtained by using the closed-form solutions provided by the Kalman filter.

2 The model

A Dynamic Linear Model (DLM) is a Gaussian linear state-space model defined by an observation equation and a state equation:

$$Y_t = F_t\theta_t + v_t \quad v_t \sim N_m(0, V_t) \tag{1}$$

$$\theta_t = G_t\theta_{t-1} + w_t \quad w_t \sim N_p(0, W_t) \tag{2}$$

A normal prior for the p-dimensional state vector at time t is specified as well:

$$\theta_0 \sim N_p(m_0, C_0)$$

We are going to consider a time-invariant DLM, i.e. where $V_t = V$, $W_t = W$, $F_t = F$ and $G_t = G$. In addition[1], $p = 1$, $m = 1$ and $F = 1$ and $V = 1$. In this way, we obtain a simple random walk plus noise model, also called local level model. θ is a Markov state process and Y_t's are, conditional on θ_t, independent. By conditional independence, the following equalities are true:

$$\pi(y_t|\theta_{0:t-1}, y_{1:t-1}) = \pi(y_t|\theta_t) \tag{3}$$

$$\pi(\theta_t|\theta_{0:t-1}, y_{1:t-1}) = \pi(\theta_t|\theta_{t-1}) \tag{4}$$

[1] We can imagine that these values have been estimated by fitting the time-invariant local level model to the data.

3 Sequential Importance Sampling Monte Carlo

We can use a discrete approximation to the target function. In general, let f be the target distribution and suppose we are dealing with an *integration problem*, hence we are looking for $E_f(h(x)) = \int h(x)f(x)$. Introducing the importance density[2] g, we have:

$$E_f(h(x)) = \int h(x)\frac{f(x)}{g(x)}g(x)dx = \int h(x)w^*(x)g(x)dx = E_g(h(x)w^*(x))$$

As shown in the Appendix 1 [4], since the normalized weights sum to one, a sample $(x^{(1)}, ..., x^{(n)}) \sim \text{Categ}(\mathbf{w})$ can be seen as a discrete approximation of the target function f.

When the target function is high dimensional, Sequential Monte Carlo techniques are employed: starting from the initial high-dimensional problem, simpler steps are implemented, as to provide univariate updates. The collection of these updates, appended one after another at each step, forms a draw from the target distribution. A sequential procedure is set up, in order to obtain the online estimates of the filtering, at each time t. A random sample of size N is generated by the particle filtering algorithm.

In filtering problems, the target is $\pi(\theta_{0:t}|y_{1:t})$, which changes at every time t. At each t, a Monte Carlo approximation of the current filtering distribution $\pi(\theta_{0:t}|y_{1:t})$, called $\hat{\pi}_t$, can be obtained as follows:

$$\hat{\pi}_t = \sum_{i=1}^{N} w_t^{(i)} \delta_{\theta_{0:t}^{(i)}} \tag{5}$$

We are also interested in obtaining an approximation of the filtering distribution of θ_t at time t, namely $\pi(\theta_t|y_{0:t})$, which can be obtained as the marginal distribution of $\hat{\pi}_t$. The Monte Carlo estimate of the expectation of the filtering distribution of θ_t at time t will be compared with the Kalman filter.

$$\pi(\theta_t|y_{1:t}) \approx \hat{\pi}_{t,t} = \sum_{i=1}^{N} w_t^{(i)} \delta_{\theta_t^{(i)}} \tag{6}$$

3.1 The importance function

We use an importance function of Markovian form. In particular, it can be factorized as $g_t(\theta_{0:t}|y_{1:t}) = g_{t|t-1}(\theta_t|\theta_{0:t-1}, y_{1:t}) \cdot g_{t-1}(\theta_{0:t-1}|y_{1:t-1})$. This structure allows to combine θ_t, generated from the first factor called transition density, and $\theta_{0:t-1}$, generated at the step before by the second component.

We are going to assume that $g_{t|t-1}(\theta_t|\theta_{0:t-1}, y_t) = g_{t|t-1}(\theta_t|\theta_{0:t-1}^{(i)}, y_t)$, which allows to use the observations in the importance transition density. Thanks to the

[2] The support of g includes the support of f.

conditional independence of the model, $g_{t|t-1}(\theta_t|\theta_{0:t-1}, y_t) = g_{t|t-1}(\theta_t|\theta_{t-1}, y_t)$. By Gaussian conjugacy, one gets[3]:

$$\theta_t|\theta_{t-1}, y_t \sim N(\theta_{t-1} + W(y_t - \theta_{t-1}), W - W^T(V+W)^{-1}W)$$

3.2 Strengths of the algorithm

In this project, the algorithm has been implemented to a case in which also a closed-form solution for $\theta_t|y_{1:t}$, provided by the Kalman filter, was available. In these cases, one may usually want to opt for the closed-form solution, being the former an approximation of the target distribution and the latter analytically exact. However, when in presence of a non-linear, non-Gaussian discrete state-space model, for which closed-form solution do not exist and integrations can be puzzling, the algorithm offers an easy way to recover the filtering distribution. Particularly, the sequential nature of the algorithm suits very well with the online filtering problem.

3.3 Implementation issues

One issue of this class of algorithms is the *degeneracy* or deterioration. Indeed, after some iterations, looking at $\hat{\pi}_t$, it is frequent that few sequences $\theta_{0:t}^{(i)}$ concentrate the majority of the total weight, and at the same time, many sequences may experience negligible weights [1]. To overcome this problem, one can sample with replacement the sequences $\theta_{0:t}^{(i)}$ with weights (probabilities) $w_t^{(i)}(\theta_{0:t})$ and reset the weights to $\frac{1}{N}$. This resampling step is implemented if a certain concentration of weights occurs. Kong et al. [2] and Liu [3] suggested to resample if the effective sample size, computed after drawing the particles and updating the weights, falls below a certain threshold, that we call *tolerance* in the Algorithm 1. Hence, a multinomial resampling step is implemented and the weights are reset to $\frac{1}{N}$ every time the effective sample size falls below the tolerance level, set to $\frac{N}{2}$.

4 Python implementation

The algorithm has been implemented using Python3 and the code is also available in the Appendix 2 of this document.

The code allows to generate a process from the specification of m_0, C_0, V and W. Also, the number of particles N can be specified before launching the algorithm. A comparison with the Kalman filter is also provided, along with some graphs of the results obtained. In addition, a graph of the ESS is drawn, in order to visualize when the multinomial resampling step has been automatically implemented. Although several *tolerance* values for the effective sample

[3] As in the previous section, the following is expressed in terms of matrix. However, our implementation is tailored for a one-dimensional state vector.

Algorithm 1: Particle Filter Algorithm

```
Initialize $(\theta_0^{(1)}, ..., \theta_0^{(n)})$ from the prior. Set $w_0^{(i)} = 1/n \ \forall i = 1, ..., n$.
for $t$ = 1,...,$T$ do
    for $i$ = 1,...$n$ do
        Draw $\theta_t^{(i)}$ from $g_{t|t-1}(\theta_t|\theta_{0:t-1}^{(i)}, y_{1:t})$.
        Set $\tilde{w}_t^{(i)} = w_{t-1}^{(i)} \frac{\pi(\theta_t^{(i)}, y_t|\theta_{t-1}^{(i)})}{g_{t|t-1}(\theta_t^{(i)}|\theta_{0:t-1}^{(i)}, y_{1:t})}$
        Normalize $w^{(i)} = \frac{\tilde{w}^{(i)}}{\sum_{i=1}^n \tilde{w}^{(i)}}$
        Compute $ESS = \left( \sum_{i=1}^n (w_t^{(i)})^2 \right)^{-1}$
    end
    if $ESS$ < tolerance then
        for $i$=1,...,$n$ do
            Draw $\theta_{0:t}^{(i)}$ from a Multinomial$(w_t^{(1)}, ..., w_t^{(N)})$
            Set $w_t^{(i)} = \frac{1}{N}$
        end
    end
    Output $\hat{\pi}_{t,t} = \sum_{i=1}^N w_t^{(i)} \delta_{\theta_t^{(i)}}$
end
```

size have been tested, the best results have been achieved with the criterion of the Algorithm 1.

As one can notice in Figure 1, the results of the algorithm are very closed to the exact Kalman filtering, thus qualitatively showing the good performance of the algorithm.

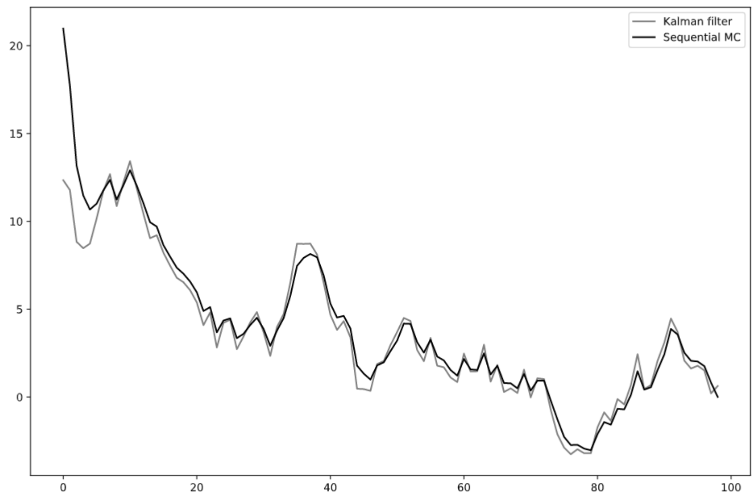

Fig. 1: Filtering estimates – Kalman filter and Particle filter

Figure 2 shows the ESS, where one can clearly see when the multinomial resampling step took place.

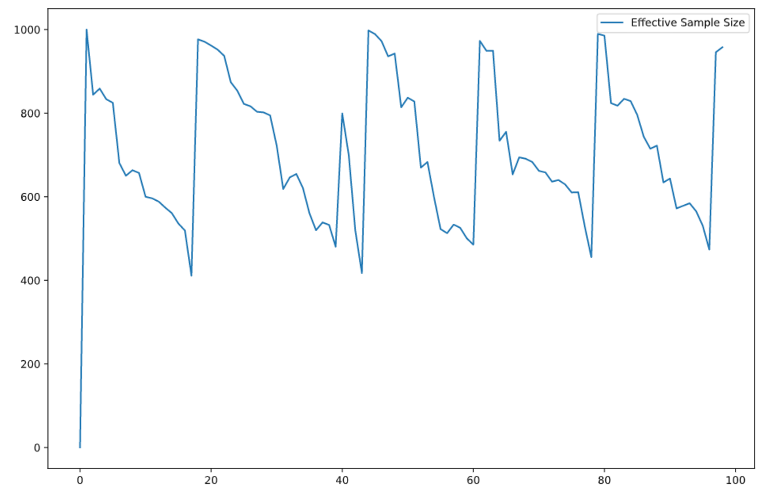

Fig. 2: Effective sample size and the effect of multinomial resampling

References

1. Johansen, A.M., Evers, L., Whiteley, N.: Monte carlo methods. Lecture notes **200** (2010)
2. Kong, A., Liu, J.S., Wong, W.H.: Sequential imputations and bayesian missing data problems. Journal of the American statistical association **89**(425), 278–288 (1994)
3. Liu, J.S.: Metropolized independent sampling with comparisons to rejection sampling and importance sampling. Statistics and computing **6**(2), 113–119 (1996)
4. Petris, G., Petrone, S., Campagnoli, P.: Dynamic Linear Models with R. useR!, Springer-Verlag, New York (2009)
5. Smith, A.: Sequential Monte Carlo methods in practice. Springer Science & Business Media (2013)

Appendix 1

We can approximate the quantity of interest as:

$$E_f(x(x)) \approx \frac{1}{n}\sum_{i=1}^{n} h(x^{(i)})w^*(x^{(i)})$$

If the target function is known up to a constant, $C{\cdot}f(x)$, calling $\tilde{w}^{(i)} = C \cdot w^*(x^{(i)})$ and taking $h(x) \equiv C$, we can seek for an approximate value of C, which can be written as $E_f(h(x)) \approx \frac{1}{n}\sum_{i=1}^{n} C \cdot \frac{\tilde{w^{(i)}}}{C} = E_f(C) = C$. Plugging it into the desired quantity, we end up having:

$$E_f(h(x)) \approx \frac{\frac{1}{n}\sum_{i=1}^{n} h(x^{(i)}\tilde{w}^{(i)})}{\sum_{i=1}^{n} \tilde{w}^{(i)}} = \frac{1}{n}\sum_{i=1}^{n} h(x^{(i)})w^{(i)} \tag{7}$$

where $w^{(i)} = \frac{\tilde{w}^{(i)}}{\sum_{i=1}^{n} \tilde{w}^{(i)}}$ are the normalized weights.

Appendix 2

```python
# -*- coding: utf-8 -*-
'''
The following code implements a Sequential Monte Carlo for a "local level" Dynamic Linear Model
The algorithm is from Petris et al. - Dynamic Linear Models with R
'''

import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import random

'''
The Dynamic Linear Model is specified by the observation and state equation as follows:
y[t] = theta[t] + v, where v is distributed as a Normal(0,V)
theta[t] = theta[t-1] + w, where w is distributed as a Normal(0,W)

In addition, the prior on theta is a distributed as a Normal(m0,c0)

In the following implementation, the parameters of the model are considered known.
We then generate a process of dimension t with the specified parameters.
'''
m0, C0, V, W = 5, 3, 2, 1
t = 100
theta = np.zeros(t)
theta[0] = stats.norm.rvs(loc = m0, scale = C0)
y = np.zeros(t)
for t in range(1, t):
    theta[t] = stats.norm.rvs(loc = theta[t-1], scale = W)
    mt = theta[t]
    y[t] = stats.norm.rvs(loc = mt, scale = V, size = 1)


fig, ax = plt.subplots(figsize=(16,9)) # Plotting the generated process - latent state theta and observation y
ax.plot(y[1:])
ax.plot(theta[1:])

```

```python
N = 1000 # N is the number of "particles", i.e. the dimension of the sample generated.
tol = N/2 # Tolerance level for the Effective Sample Size.

sd_importance = np.sqrt(W - W**2/(V + W)) # Definition of the importance distribution standard deviation
sd_theta_y = np.sqrt(V + W)


'''
In the following, the algorithm is implemented.
Firstly, the arrays used in the algorithm are initialized.
'''

w_t = np.zeros(shape = (t + 1, N))
thetas_sim = np.zeros(shape = (t + 1, N))
pi_hat_sample = np.zeros(shape = (t+1,N))
ESS = np.zeros(t+1)
theta_res = np.zeros(shape = (t+1, N)) # auxiliary array used for the resampling step
thetas_est = np.zeros(t) # Monte Carlo approximations of filtering mean of theta_t|y_1:t
filt_est = np.zeros(shape = (t + 1, N)) # approximate sample from theta_t|y_1:t at each t


thetas_sim[1] = stats.norm.rvs(loc = m0, scale = C0) # initialization from the prior

w_t[1] = np.repeat(1/N,N) # initialization with equal weights

filt_est[1] = np.random.choice(thetas_sim[1], N, p=w_t[1])


for i in range(2,t+1):

    # Drawing theta_i's from the importance distribution
    y_theta = (y[i-1] - thetas_sim[i-1])
    var_sum = W + V
    mean_importance = thetas_sim[i-1] + W * y_theta/var_sum
    thetas_sim[i] = stats.norm.rvs(loc = mean_importance, scale = sd_importance**2)

    # Updating the weights w_t
    pi_g = w_t[i-1] * stats.norm.pdf(y[i-1], loc = thetas_sim[i-1], scale = sd_theta_y**2)
    w_t[i] = pi_g / np.sum(pi_g)

    # Evaluating ESS
    ESS[i] = (np.sum(w_t[i]**2))**(-1)

    # Multinomial resampling
    if ESS[i] < tol:
        index = np.random.choice(range(N), N , p= w_t[i])

        for c in range(N):
            theta_res[:,c] = thetas_sim[:,index[c]]
        thetas_sim = theta_res

        w_t[i] = np.repeat(1/N, N)

    # Drawing a sample from the approximate filtering distribution in t:
    filt_est[i] = np.random.choice(thetas_sim[i], N, p=w_t[i])

    # Monte Carlo approximations of filtering mean at t
    thetas_est[i-1] = np.dot(thetas_sim[i],w_t[i]) / np.sum(w_t[i])


# Graph of ESS, which indicates the points at which the multinomial resampling has been implemented:
fig, ax = plt.subplots(figsize=(12,8))
ax.plot(ESS[1:], label = "Effective Sample Size")
ax.legend();


'''
In the following code, some plots are drawn in order to qualitatively assess the performance of the algorithm compared to the (exact) Kalman filter.
The first plot shows the observation y and the filtering estimates of the algorithm.
Then, some draws from the approximate filtering distribution at various t are plotted.
After computing the Kalman filter estimates, the second plot shows a comparison between Kalman filter and the algorithm implemented.
'''

# Observation and filtering
```

```python
fig, ax = plt.subplots(figsize=(12,8))
ax.plot(y[1:], label = "Observations")
ax.plot(thetas_est[1:], label = "Estimated thetas")
ax.legend();


# Graph of approximate filtering distributions
fig, ax = plt.subplots(figsize=(12,8), nrows = 3, ncols = 2)
c = [10,40,90]
for i,j in enumerate([2,30,80]):
    k = 0
    ax[i][k].hist(filt_est[j], alpha=0.5, bins=100, density=True,
    ↪  stacked=True, label = f"Filtering at t={j}")
    ax[i][k].legend();
    k += 1
    ax[i][k].hist(filt_est[c[i]], alpha=0.5, bins=100,
    ↪  density=True, stacked=True,label = f"Filtering at
    ↪  t={c[i]}")
    ax[i][k].legend();


# Closed-form solutions for Kalman filter
r = np.zeros(t)
q = np.zeros(t)
m = np.zeros(t)
f = np.zeros(t)
c = np.zeros(t)
a = np.zeros(t)
m[0] = m0
c[0] = C0
r[0] = c[0] + W
for t in range(1,t):
    a[t] = m[t-1]
    r[t] = c[t-1] + W

    f[t] = a[t]
    q[t] = r[t] + V

    m[t] = a[t] + r[t]*(y[t]-f[t])/q[t]
    c[t] = r[t] - (r[t]**2) / q[t]
theta_kalman = m

# Comparison between Kalman filter and Sequential MC
fig, ax = plt.subplots(figsize=(12,8))
ax.plot(theta_kalman[1:], label = "Kalman filter")
ax.plot(thetas_est[1:], label = "Sequential MC")
ax.legend();
```