APPM 5510 HW 7

Zane Jakobs

October 18, 2019

1(a)

Plots are not included here because both filters blew up to machine infinity within 7 assimilation cycles, so I suspect there is an error in my implementation (and if this line is still here when I turn this in, that means I ran out of time trying to find it), which is attached to the end of the assignment (and will be on my GitHub). If you want to run this code and don't want to write your own makefile, let me know and I'll send you mine.

2(a)

Solving numerically with the Python code at the end, we have

$$C = \begin{bmatrix} 0.5 & 0.25 & -0.25 \\ 0.25 & 0.5 & 0.25 \\ -0.25 & o.25 & 0.5 \end{bmatrix}$$

2(b)

Again with the Python code, we get something different–since the values are a bit non-integral, here's the terminal output:

```
[[ 0.43250799  0.06829073 -0.36421725]
[ 0.06829073  0.01078275 -0.05750799]
[-0.36421725 -0.05750799  0.30670927]]
```

This is different from the above, because the matrix $\Sigma^{-1}\Sigma^2\Sigma^{-1}$ (where $^{-1}$ indicates taking the pseudoinverse) is not quite the identity (the bottom right element is a zero), so the formula for C is a bit different than in the EnKF case.

2(c)

This time, we get the same result as in part (a), because this time, we put the zero eigenvalue in the third column, which multiplies the zero in $\Sigma^{-1}\Sigma^2\Sigma^{-1}$, and thus does not introduce a new zero into the multiplication chain, which does happen in part (b), since there, the zero eigenvalue multiplies a nonzero part of $\Sigma^{-1}\Sigma^2\Sigma^{-1}$.

Filter Implementation

```
public:

constexpr EnKF() {};

//filter with one observation
/* virtual void filter(const Obs_t& y);*/

virtual Obs_t observe(const State_t& x);

virtual State_t forecast(State_t& x, ForecastArgs&... args);

virtual ~EnKF() {};
};

//scalar EnKF with linear observations
template<class... ForecastArgs>
class VectorEnKF : public EnKF<Vec, Vec, Mat, ForecastArgs...>
{

private:

Mat  m_H;

Vec m_state;

Vec m_obs = m_H * m_state;

std::function<Vec(Vec&, ForecastArgs&...)> m_forecast;

Vec m_ensemble_mean;

Mat m_ensemble_covariance;

Mat m_obs_covariance;

int m_ensemble_size;

int m_dimension;

Mat m_ensemble = Mat::Zero(m_dimension, m_ensemble_size);
```

```
bool m_initialized_ensemble = false;

Mat m_background_covariance = Mat::Identity(m_dimension, m_dimension);

Mat m_ensemble_transform = gen_covariance_transform(m_ensemble_covariance);
public:

VectorEnKF(const std::function<Vec(Vec&, ForecastArgs&...)>& n_forecast,
const Mat& n_H,
const Vec& n_initial_state,
const Mat& n_observe_err_covariance,
const Vec& n_initial_ensemble_mean,
const Mat&   n_initial_ensemble_covariance,
const int    n_ensemble_size
) : EnKF<Vec, Vec, Mat, ForecastArgs...>(),
m_H(n_H),
m_state(n_initial_state),
m_forecast(n_forecast),
m_ensemble_mean(n_initial_ensemble_mean),
m_ensemble_covariance(n_initial_ensemble_covariance),
m_obs_covariance(n_observe_err_covariance),
m_ensemble_size(n_ensemble_size),
m_dimension(static_cast<int>(n_initial_state.size()))

{};


void set_state(const Vec& n_state) noexcept
{
m_state = n_state;
}

void set_ensemble_size(const int n_ensemble_size) noexcept
{
assert(n_ensemble_size > 0);
m_ensemble_size = n_ensemble_size;
}
```

```
Vec state()
{
return m_state;
}

int ensemble_size()
{
return m_ensemble_size;
}

Vec ensemble_mean()
{
return m_ensemble_mean;
}

Mat ensemble_covariance()
{
return m_ensemble_covariance;
}

Vec observe(const Vec& x){
return m_H * x;
}

Vec forecast(Vec& x, ForecastArgs&... args){
return m_forecast(x, args...);
}


private:

Mat gen_covariance_transform(const Mat& target_covariance) const
{
return target_covariance.ldlt().matrixL();
}

Mat gen_obs_perturbations()
{
auto rcovtransform = gen_covariance_transform(m_obs_covariance);
```

```
std::random_device rd{};
std::mt19937 gen{rd()};
std::normal_distribution<> dis{0.0, 1.0};

Mat A(m_H.rows(), m_ensemble_size);
Vec epsilon(m_H.rows());

for(auto i = 0; i < m_ensemble_size; ++i){
for(auto e = 0; e < m_H.rows(); ++e){
epsilon[e] = dis(gen);
}
A.col(i) = epsilon;
}

return rcovtransform * A;
}


Mat gen_ensemble_perturbations(bool gen_cov_transform=true)
{
std::random_device rd{};
std::mt19937 gen{rd()};
std::normal_distribution<> dis{0.0, 1.0};

Mat A(m_dimension,m_ensemble_size);
Vec epsilon(m_dimension);

if(gen_cov_transform){
m_ensemble_transform = gen_covariance_transform(m_ensemble_covariance);
}


for(auto i = 0; i < m_ensemble_size; ++i){
for(auto e = 0; e < m_dimension; ++e){
epsilon[e] = dis(gen);
}
A.col(i) = epsilon + m_ensemble_mean;
}

return  m_ensemble_transform * A;// * A;
```

```
}

/*std::pair<Vec,Mat> sample_mean_covariance(const std::vector<Vec>& sample)
{
Vec mu(sample[0].size());

for(auto i = 0; i < m_ensemble_size; ++i){
for(auto i = 0; i < sample[0].size(); ++i){
mu[i] += s[i];
}
}

mu /= sample.size();

Mat cov = Mat::Zero(sample.size(), sample.size());

for(const auto& s : sample){
auto delta = s - mu;
cov.noalias() += delta * delta.transpose();
}

cov /= (sample.size() - 1);

return std::make_pair(mu, cov);
}*/

//A is the result of a call to form_A
Mat kalman_gain_A(const Mat& A)
{
auto V = m_H * A;

auto vvr = V * V.transpose();// + m_obs_covariance;
return A * V.transpose() * vvr.inverse();
}

//copy ensemble here
Mat form_A(Mat ensemble)
{
for(auto i = 0; i < m_ensemble_size; ++i){
ensemble.col(i) -= m_ensemble_mean;
```

```
}

return ensemble / std::sqrt(m_ensemble_size - 1);
}

Mat kalman_gain_ensemble(const Mat& ensemble)
{
auto A = form_A(ensemble);
return kalman_gain_A(A);
}

//B matrix given an ensemble
Mat ensemble_prior_covariance(const Mat& A)
{
return A * A.transpose();
}


Vec ensemble_member_update(const Mat& K, const Vec& perturbed_y, const Vec& xi)
{
auto err = perturbed_y - m_H * xi;

return xi + K * err;
}

Mat ensemble_update(const Mat& K, const Mat& perturbed_yvals, const Mat& ensemble_xvals
{
Mat updated_xvals(ensemble_xvals.rows(), ensemble_xvals.cols());
for(auto i = 0; i < m_ensemble_size; ++i){
updated_xvals.col(i) = ensemble_member_update(K, perturbed_yvals.col(i), ensemble_xvals
}
return updated_xvals;
}

Mat ETKF_ensemble_update_mat(Mat& A)
{
auto V = m_H * A;
auto Id = Mat::Identity(m_ensemble_size);
auto emat = Id + V.transpose() * m_obs_covariance.inverse() * V;
```

```
Eigen::SelfAdjointEigenSolver<Mat> esolver(emat);

assert(esolver.info() == Eigen::Success);//, "ETKF eigendecomposition failed.");

Mat Gamma = esolver.eigenvalues().asDiagonal();/* I + Gamma in the notes*/
auto Q = esolver.eigenvectors();

for(auto i = 0; i < Gamma.cols(); ++i){
Gamma(i,i) = std::sqrt(Gamma(i,i));
}

return Q * Gamma * Q.transpose();
}

Mat ETKF_posterior_ensemble(Mat& A)
{
// A is prior ensemble
auto X = ETKF_ensemble_update_mat(A);
auto one = Vec::Ones(m_ensemble_size);
auto aplus = A * one / m_ensemble_size;
return A - aplus * one.transpose();
}


Mat ensemble_posterior_covariance(const Mat& B, const Mat& K)
{
return B - K * m_H * B;
}



public:

void ETKF_filter(Vec& obs, ForecastArgs&... args)
{
for(auto i = 0; i < m_ensemble_size; ++i){
m_ensemble.col(i) = m_state;
}

m_ensemble += gen_ensemble_perturbations();
```

```
m_ensemble = ETKF_posterior_ensemble(m_ensemble);

m_state = m_ensemble.rowwise().mean();

m_ensemble_mean = m_state;

m_ensemble_covariance = m_ensemble * m_ensemble.transpose();
}


void filter(Vec& obs, ForecastArgs&... args)
{
// generate ensemble
if(!m_initialized_ensemble){
for(auto i = 0; i < m_ensemble_size; ++i){
m_ensemble.col(i) = m_state;
}
//m_initialized_ensemble = true;
}
m_ensemble += gen_ensemble_perturbations();
/* forecast ensemble */
for(auto i = 0; i < m_ensemble_size; ++i){
Vec xstate = m_ensemble.col(i);
m_ensemble.col(i) = m_forecast(xstate, args...);
}

m_ensemble_mean = m_ensemble.rowwise().mean();

auto A = form_A(m_ensemble);
m_background_covariance = A * A.transpose();

Mat obs_perturbed = gen_obs_perturbations();

for(auto i = 0; i < m_ensemble_size; ++i){
obs_perturbed.col(i) += obs;
}
auto K = kalman_gain_A(A);
A = ensemble_update(K, obs_perturbed / std::sqrt(m_ensemble_size-1), A);
```

```
m_ensemble_covariance = A * A.transpose();

/* get ensemble back */
A *= std::sqrt(m_ensemble_size - 1);
//add prior mean back and re-scale
m_ensemble_mean = std::sqrt(m_ensemble_size - 1) * (A.rowwise().mean() + m_ensemble_mea
m_state = m_ensemble_mean;
}




};


}//namespace fastmath
#endif
```

Filter Calling Code

```
#include "enkf.hpp"
#include <Eigen/Core>
#include <vector>
#include <fstream>

using Vec = Eigen::VectorXd;
using Mat = Eigen::MatrixXd;

Vec Henon_Map(Vec& x, double& a, double& b){
auto xn = x[0];
auto yn = x[1];

x[0] = 1 - a * xn * xn + yn;
x[1] = b * xn;

return x;
}

double obs_err(){
```

```
std::random_device rd{};
std::mt19937 gen{rd()};
std::normal_distribution<> dis{0.0, std::sqrt(0.02)};

return dis(gen);
}

int main(int argc, char** argv){
using namespace fastmath;

std::function<Vec(Vec&, double&, double&)> fmap = Henon_Map;
int ensemble_size = 31;
auto initial_state = Vec::Zero(2);

Mat H(1,2);

H << 1.0, 0.0;

Vec true_state(2);
true_state << 0.0, 0.0;

Vec ensemble_mean = Vec::Zero(ensemble_size);

Mat ensemble_cov = 0.01 * Mat::Identity(2,2);

Mat obs_err_cov = 0.02 * Mat::Identity(1,1);

int n_assim_cycles = 100;

double a = 1.4, b = 0.3;
Mat state_mat(2, n_assim_cycles);
for(auto i = 0; i < n_assim_cycles; ++i){
true_state = Henon_Map(true_state, a, b);
state_mat.col(i) = true_state;
}

std::vector<double> filter_err, etkf_err;

auto enkf_runner = VectorEnKF<double, double>(fmap, H, initial_state,
obs_err_cov, initial_state,
```

```
ensemble_cov, ensemble_size);

auto etkf_runner = VectorEnKF<double, double>(fmap, H, initial_state,
obs_err_cov, initial_state,
ensemble_cov, ensemble_size);

Vec y(1);
double yval;
Vec analysis_mean;
for(auto i = 0; i < n_assim_cycles; ++i){
true_state = state_mat.col(i);
yval = true_state[0] + obs_err();
y[0] = yval;
enkf_runner.filter(y, a, b);
etkf_runner.filter(y, a, b);
analysis_mean = enkf_runner.state();
Vec etkf_mean = etkf_runner.state();
Vec err = true_state - analysis_mean;
filter_err.push_back(err.norm());
err = true_state - etkf_mean;
etkf_err.push_back(err.norm());
}

std::ofstream fwriter("hw7.csv"), twriter("hw7etkf.csv");
for(const auto& e : filter_err){
fwriter << e << '\n';
}

for(const auto& e : etkf_err){
twriter << e << '\n';
}

return 0;
}
```

Python Code (Problem 2)

```
import numpy as np
```

```
def gamma_pseudoinverse(gmat, n):
g = np.zeros_like(gmat)
for i in range(n):
g[i,i] = 1/ np.sqrt(1 + g[i,i])

return g

if __name__ ==’__main__’:
x = 2 ** 0.5 * np.array([[1.0,1.0,0.0],[-1.0,0.0, 1.0],[0.0,-1.0,-1.0]])

H = np.eye(3)
R = np.eye(3)

xdelta = x - x.mean(axis=0, keepdims=True)

B = 0.5 * xdelta.T @ xdelta

#H is identity
K = B @ np.linalg.inv(B + R)

#again, H is identity
C = (H - K) @ B

print("Posterior covariance: ", C)

A = xdelta.T / (2 ** 0.5)

V = H @ A

F, sgm, W = np.linalg.svd(A.T @ A)
sigma = np.diag(sgm)
sigmaInv = np.linalg.pinv(sigma)

sigmaPseudoId = sigmaInv @ sigma @ sigma @ sigmaInv

#V^T V is Hermitian, and R is identity so V^T R V = V^T V
Gamma, Q = np.linalg.eigh(V.T @ V)

Gamma[0] = 0.0
```

```
#2b

GammaInv = np.diag(1.0/np.sqrt(Gamma + 1))

C = A @ Q @ GammaInv @ sigmaPseudoId @ GammaInv @ Q.T @ A.T

print("EAKF C: ", C)

#2c, same as above, just flip the columns in Gamma and Q

A = xdelta.T / (2 ** 0.5)

V = H @ A

F, sgm, W = np.linalg.svd(A.T @ A)
sigma = np.diag(sgm)
sigmaInv = np.linalg.pinv(sigma)

sigmaPseudoId = sigmaInv @ sigma @ sigma @ sigmaInv
Gamma, Q = np.linalg.eigh(V.T @ V)

Qp = np.fliplr(Q)

Gamma[0] = Gamma[2]
Gamma[2] = 0.0

GammaInv = np.diag(1.0/np.sqrt(Gamma + 1))

print(sigmaPseudoId)

C = A @ Qp @ GammaInv @ sigmaPseudoId @ GammaInv @ Qp.T @ A.T

print("EAKF C: ", C)
```