Montogue

Tutorial MAT4
Kalman Filters with MATLAB

Lucas Monteiro Nogueira

• Summary •	
Problem 1	**A function used to implement the basic Kalman filter**
Problem 2	**An univariate example of Kalman filter in MATLAB: The combustion chamber problem**
Problem 3	**A multivariate example of Kalman filter in MATLAB: The falling paper wad problem**

MATLAB Files

The MATLAB codes for this tutorial can be found in one of Montogue's Google Drive folders; access it here.

PROBLEMS

▶ Problem 1 – A function used to implement the basic Kalman filter

Following Barreto *et al.* (2021), we begin by developing a MATLAB function that implements the computations associated with both the prediction phase and the correction (or update) phase of a single iteration of the discrete Kalman filter. An iteration of the Kalman filter requires us to first obtain a preliminary estimate of the state vector. If the current state vector is $\mathbf{x}(t)$, the estimate of the state vector predicted by the model will be $\mathbf{x}(t+1)$. Likewise, the predicted uncertainty of the estimate, $\mathbf{P}(t+1)$, is obtained from $\mathbf{P}(t)$. Updated matrices are obtained with the following prediction equations:

$$\mathbf{x}_{\mathbf{M}}(t+1) = \mathbf{F}(t)\mathbf{x}(t) + \mathbf{G}(t)\mathbf{u}(t)$$
$$\mathbf{P}_{\mathbf{M}}(t+1) = \mathbf{F}(t)\mathbf{P}(t)\mathbf{F}(t)^{T} + \mathbf{Q}(t)$$

As shown, in order to implement these equations we also need matrices $\mathbf{F}(t)$, $\mathbf{G}(t)$, and $\mathbf{Q}(t)$. If we are dealing with a time-varying model, we would need to ascertain the correct values of $\mathbf{F}(t)$ and $\mathbf{G}(t)$ for that given time. Although the formulation of the Kalman filter allows for this possibility, it is common to deal with time-invariant systems where the matrices $\mathbf{F}(t)$ and $\mathbf{G}(t)$ in the model are constant. Similarly, in principle we also need the external uncertainty matrix $\mathbf{Q}(t)$, but we often have access to a unique, constant characterization of this uncertainty.

If this system actually receives "control inputs" at every iteration, the current values of those signals would need to be provided in the vector $\mathbf{u}(t)$. Of course, some systems receive no "control inputs," while for others the values in vector $\mathbf{u}(t)$ might be constants.

Once the prediction phase has been carried out and the values of $\mathbf{x_M}(t+1)$ and $\boldsymbol{P_M}(t+1)$ have been determined, we may employ these estimates as prior information for the correction phase. We first perform a simple assignment of variables:

$$\mathbf{x}_{\mathbf{B}} = \mathbf{x}_{\mathbf{M}}(t+1)$$
$$\mathbf{P}_{\mathbf{B}} = \mathbf{P}_{\mathbf{M}}(t+1)$$

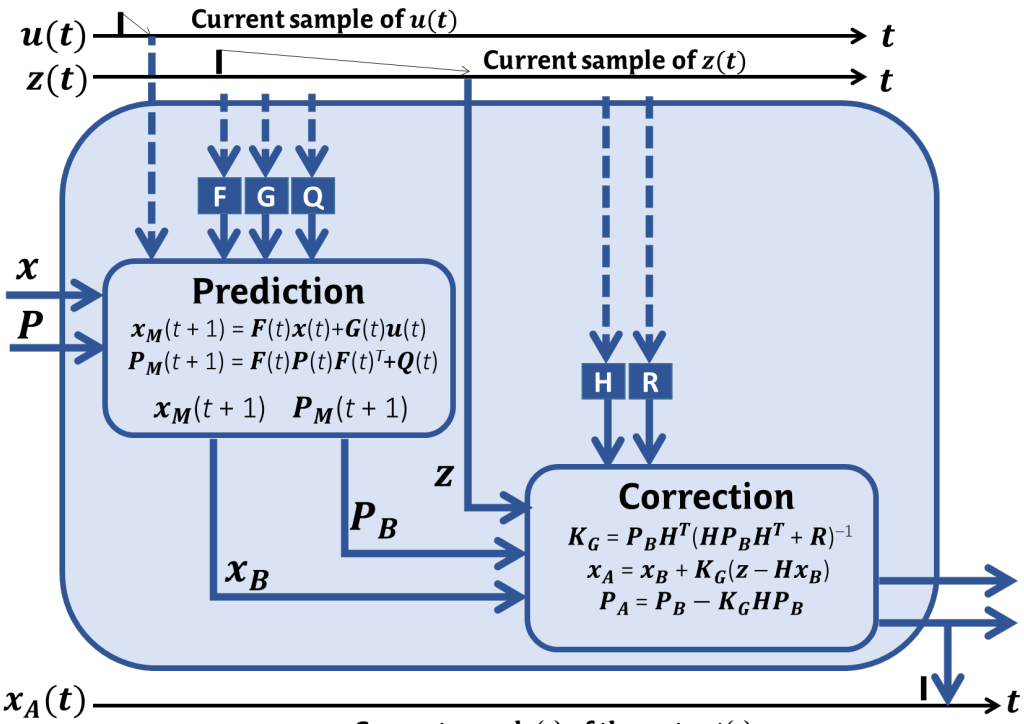
Here, the subindex *B* ("before") is used to indicate that $\mathbf{x_B}$ and $\mathbf{P_B}$ are used not in a time progression from t to $t+1$, but rather as a refinement that will convert these "prior" estimates (before) to enhanced "posterior" estimates (after) in a Bayesian framework. Equipped with $\mathbf{x_B}$ and $\mathbf{P_B}$, we may write the following correction equations:

$$\mathbf{K}_{G} = \mathbf{P}_{\mathbf{B}}\mathbf{H}^{T}\left(\mathbf{H}\mathbf{P}_{\mathbf{B}}\mathbf{H}^{T} + \mathbf{R}\right)^{-1}$$
$$\mathbf{x}_{\mathbf{A}} = \mathbf{x}_{\mathbf{B}} + \mathbf{K}_{\mathbf{G}}\left(\mathbf{z} - \mathbf{H}\mathbf{x}_{\mathbf{B}}\right)$$

$$\mathbf{P}_A = \mathbf{P}_B - \mathbf{K}_G \mathbf{H} \mathbf{P}_B$$

The first equation is used to compute the Kalman gain matrix $\mathbf{K_G}$, which is then used in the next two equations. The equations also call for the substitution of **H**, the matrix that relates measurements to state variables as $\mathbf{z}(t) = \mathbf{H}(t)\mathbf{x}(t)$. Importantly, if $\mathbf{H}(t)$ varies with time, we may need to provide the correct value of $\mathbf{H}(t)$ for that point in time, specifically. However, although the Kalman filter framework allows for such a case, the relationship between measurements and state variables is often constant, so the same matrix **H** can be used for all iterations. A similar reasoning applies to measurement uncertainty term $\mathbf{R}(t)$, which can be taken as a matrix of constant entries if the nature of the uncertainty affecting the measurements does not change with time.

The flow of information in a single Kalman filter iteration is illustrated in the following diagram. The largest box represents the processor implementing the algorithm. The two smaller boxes represent the prediction and correction phases. The vertical lines represent the instantaneous data that may be required for the computation of that particular iteration of the algorithm. The lines for **F**, **G**, **Q**, **H**, and **R** are dashed and lead to internal boxes with these variable names in them, because, if the systems are time invariant and the statistics of the variables involved are constant, these parameters need not be read in every iteration (i.e., they will be fixed values that need only be initialized once).



Current sample(s) of the output(s) x_A

The vertical line labeled **z** is solid because a newly acquired set of measurements must be read in every iteration, as they are necessary to implement the correction phase of the algorithm. The line for **u** is dashed because, as we have explained before, some situations will not have a "control input" applied to the system.

The vertical line at the bottom of the biggest box represents the fact that, in each iteration, the process will generate a final, improved estimate of the state of the system, $\mathbf{x_A}$; an estimate of $\mathbf{P_A}$ is also available at the end of every iteration.

The horizontal lines coming into the process from the left represent the previous estimates of **x** and **P** obtained in the previous iteration of the algorithm (or provided as initial values). The horizontal lines leaving the biggest rectangle refer to the posterior values of $\mathbf{x_A}$ and $\mathbf{P_A}$, which will be passed on to the next iteration of the filter.

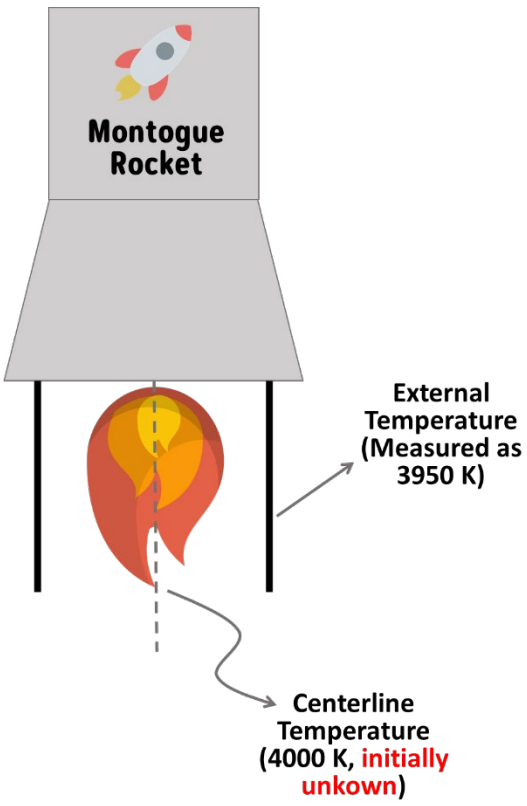
The MATLAB code *onedkf.m* implements the first iteration of the discrete Kalman filter.

```
% Function onedkf - Implements a single iteration of the discrete-time
% Kalman filter algorithm.
% Syntax: [PA, xA, KG] = onedkf(F, G, Q, H, R, P, x, u, z);
% Uses (receives) matrices F, G for the model equations
% Uses (receives) the process noise covariance matrix, Q
% Uses (receives) matrix H for the measurement equation
% Uses (receives) the measurement noise cov. matrix, R
% The following are expected to change in every iteration:
% Receives the state vector, x, and is cov. matrix, P, from the
previous iteration of the algorithm
```

```
% Receives the current vector of inputs, u.
% Receives the current vector of measurements, z.
% Performs the Prediction and Correction phases.
% Returns the POSTERIOR estimation of state vector, xA
% and its covariance matrix, PA.
% It also returns the calculated KG matrix.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [PA, xA, KG] = onedkf(F,G,Q,H,R,P,x,u,z)
%% PREDICTION PHASE, using the model
%The model predicts the new x:
FT = transpose(F);
xM = F*x + G*u;

%The model predicts the new P:
PM = F*P*FT + Q;

%Change of variables to clearly separate the 2 phases:
xB = xM;
PB = PM;

%%CORRECTION (UPDATE) PHASE
%Finding POSTERIOR (A = After) parameters, from PRIOR (B = Before),
%through Bayesian estimation
HT = transpose(H);
%First calculate the Kalman gain (KG) for this iteration
KG = PB*HT*(inv(H*PB*HT+R));

%Calculate POSTERIOR ESTIMATE of the state vector
xA = xB + KG*(z-H*xB);
%Calculate POSTERIOR ESTIMATE of state vector's covar. matrix
PA = PB - KG*H*PB;

end
```

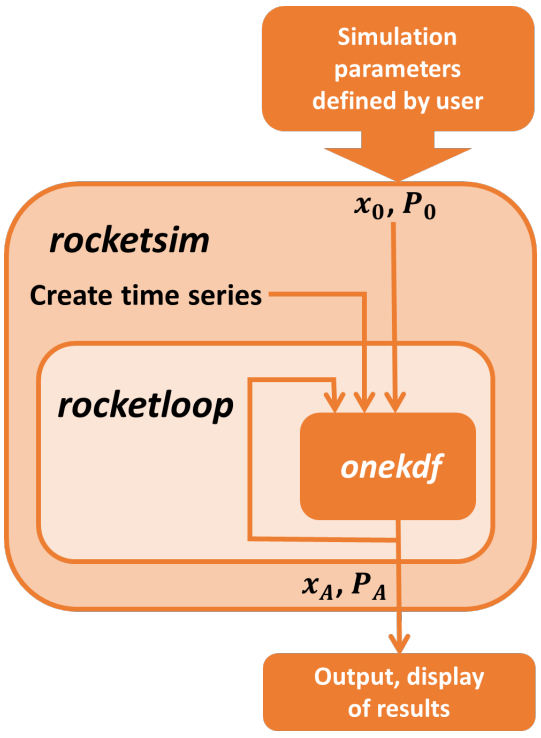
▶ Problem 2 – An univariate example of Kalman filter in MATLAB: The combustion chamber problem

The first problem we tackle is straightforward. A rocket is travelling in outer space towards another planet. For diagnostic purposes, the spacefarers in the rocket want to keep track of the burned gas temperature within the centerline of the combustion chamber; however, such a temperature cannot be measured directly in midflight. Still, the temperature in the outer, external region of the chamber is measured in successive intervals and can be processed to yield reliable estimates of the temperature in the centerline; this is where the Kalman filter comes in.



In order to continue to provide posterior estimates of state variable vector $\mathbf{x_A}$ and the covariance matrix $\mathbf{P_A}$, function *onedkf.m* must be executed recursively. The resulting $\mathbf{x_A}$ and $\mathbf{P_A}$ from one iteration are fed forward as the $\mathbf{x}(t)$ and $\mathbf{P}(t)$ for the prediction phase of the next iteration. For the execution of the very first iteration there is no previous $\mathbf{x_A}$ and $\mathbf{P_A}$ available, therefore we must supply initial values with which to begin the filtering process.

To implement *onedkf* recursively, we call this function within a piece of code (in this case, also a function) that encloses a timing loop. We call this piece of code *rocketloop*. In every iteration of the loop, the Kalman filter algorithm must draw new samples from the measurements (and other variables, such as control inputs, and parameters, if they are not constant). Then, we may embed the loop function within a top-level simulation function in which we will create, in advance, all the time series that will be required, in their entirety, and any other settings associated with the filtering process. We call this top-level function *rocketsim*. The relationship between *rocketloop* and *rocketsim* is illustrated below.



The top-level function *rocketsim* should enable us to:

1. Define the parameters of the model we want to use, including the levels of uncertainty associated with the external noise (**Q**) and the measurements (**R**).
2. Use those parameters to create all the time series that will be required for the complete simulation, in advance. This may include the creation of a time series of "true values" for the state variables, if appropriate.
3. Define the initial values of **x**(*t*) and **P**(*t*) that are needed for the first execution of *onedkf*.
4. Assign the maximum number of iterations to be simulated, *iter*, so that the Kalman filter algorithm will be run from *t* = 1 to *t* = *iter*.
5. Include plotting commands with which to follow the evolution of important variables and their associated uncertainties.

We expect no dynamic evolution of this system with time, therefore the model equation is simply

$$\mathbf{x}(t+1) = \mathbf{x}(t)$$

Function *rocketsim* will receive the following inputs:

- `xTru`: A scalar value used to signify the mean of the "true" values of the centerline temperature for the simulation.
- `x0, P0`: Initial values that will be used in the first execution of *onedkf*. Because there is only one state variable (temperature) in this scenario, *P0* will be a scalar representing the initial model uncertainty for that variable, expressed as its variance.
- `Q,R`: Covariance matrices (in this case, scalar variances) which represent the uncertainty of the model attributed to external inputs (noise) and the uncertainty in the temperature measurements.
- `iter`: The number of total iterations we want to simulate (from *t* = 1 to *t* = *iter*).

Since this is a simulation, it will be useful to have a time series that we may consider to be the "true" temperature *xTru* throughout the analysis time. Equipped with this time series, we may perceive how close or how far our

Kalman filter output is relatively to the "true" state variable (namely, the temperature).

While there may be no external control inputs (i.e., **u***(t)*) applied to the system at hand, we must still be able to account for external noise associated with phenomena that we cannot control but are nonetheless present. For this purpose, we incorporate a nonzero value of **Q** and use it in the Kalman filter computations. Since we are dealing with a univariate problem, **Q** will be a 1×1 matrix whose sole entry is the variance of the external noise. The "true" temperature we aim for is a certain mean value added to a time series of Gaussian random values with zero mean and a standard deviation that is the square root of the only entry in **Q**.

To perform the first iteration of the Kalman filter, we will also require initial values for **x***(t)* and its covariance matrix, **P**(t). For the former, we assume that theoretical calculations for the combustion chamber suggested an unburned gas temperature of 3950 K in the centerline. We will set up our code so that the actual centerline temperature is found to be 4000 K. In turn, **P**(0) is just a scalar variance representing the level of uncertainty we assign to the initial centerline temperature estimate; we arbitrarily take **P**(0) = 0.6.

```
% rocketsim - Top-level function for simulation of univariate Kalman
filter
% SYNTAX: [XAVECT, PAVECT, KGVECT] = rocketsim(xTru, x0, P0, Q, R,
iter)

function [XAVECT, PAVECT, KGVECT] = rocketsim(xTru, x0, P0, Q, R,
iter)

%Resets the random number generator to ver. 5 normal
%and sets seed to 12345.
rng(12345, 'V5normal');

%Creation of vector of true temperature values and implementation of
%noise
%Note that Q = variance of noise
%Creation of vector of TRUE temperature values:
xTru_noNoise = ones(iter,1)*xTru;
extNoise = randn(iter,1)*(sqrt(Q));
xTrue = xTru_noNoise + extNoise;

%Creation of vector of MEASURED values, taking into account that the
%measuring system ADDS noise to each TRUE VALUE measured:
msdError = randn(iter,1)*(sqrt(R));
zVect = xTrue + msdError;

[XAVECT, PAVECT, KGVECT] = rocketloop(xTru, zVect, x0, P0, Q, R,
iter);
colorVect = [0.5, 0.5, 0.5];

figure(1)
plot(xTrue, 'Color', colorVect, 'LineWidth', 1.8);
hold on
plot(XAVECT, 'r', 'LineWidth', 1.8);
plot(zVect, 'k-.', 'LineWidth', 1.8);
hold off
title('xTrue, xA and z');
xlabel('Kalman Filter Iterations');
legend('xTrue', 'xA', 'zVec', 'Location', 'southeast');
grid on

figure(2)
plot(PAVECT, 'k', 'LineWidth', 1.8);
title('PA');
grid
xlabel('Kalman Filter Iterations');
figure(3)
plot(KGVECT, 'k', 'LineWidth', 1.8);
title('KG Values')
grid
xlabel('Kalman Filter Iterations');

end
```

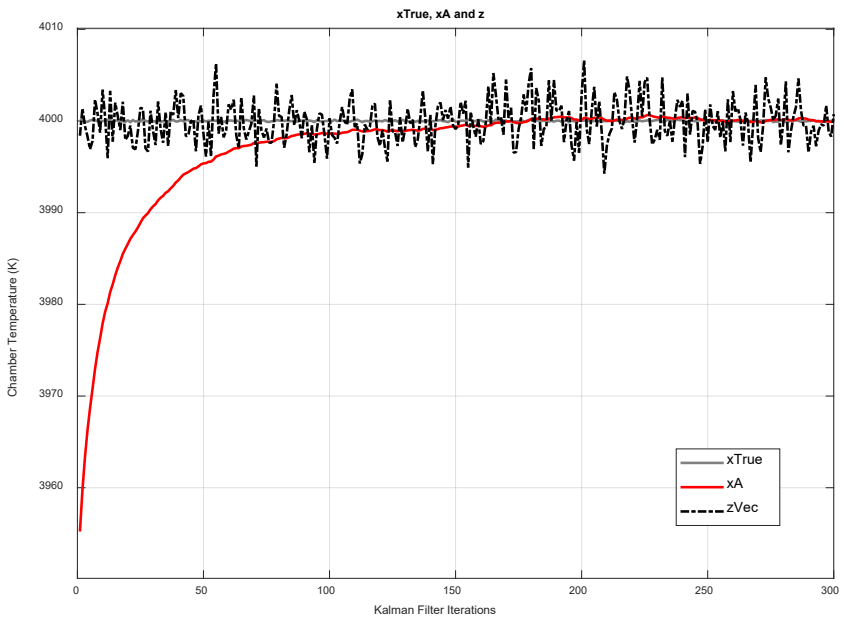
Next, function *rocketloop* receives the initial values of *x0* and *P0* and loads them into the variables *x* and *P*, which in turn will be passed to *onedkf*. It also passes along parameters *Q* and *R*, which will remain constant. It receives the

time series *xTru* and *zVect*. The vector *zVect* will be accessed by *onekdf* in every iteration. In addition, *rocketloop* receives the value of *iter* to establish the number of iterations.

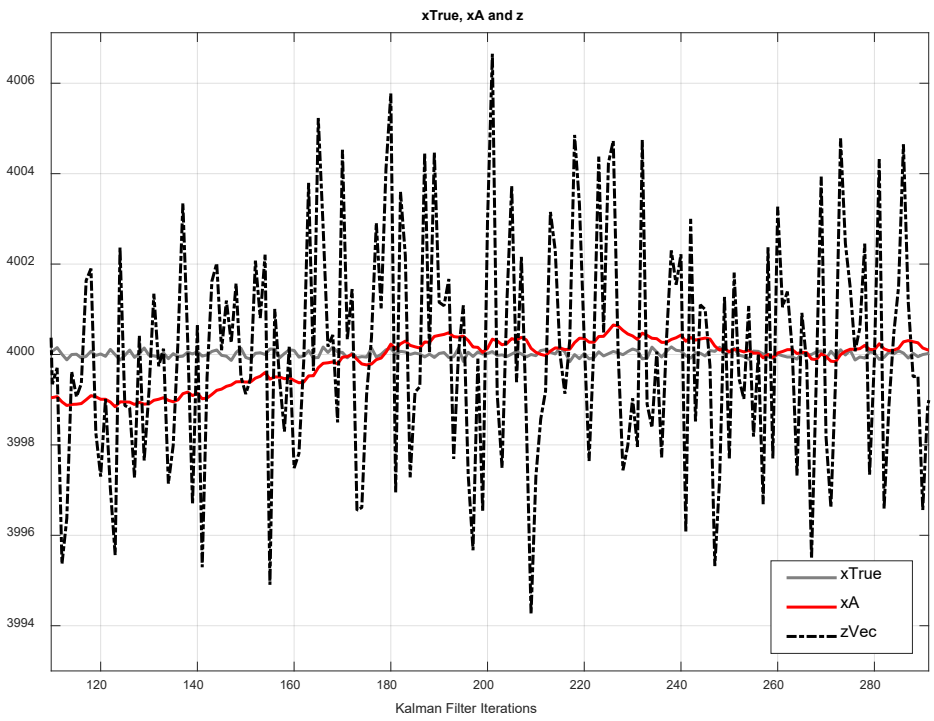
Function *rocketloop* establishes the parameters of the scenario (model and measurement equations), assigning values to **F** and **H**, which remain constant throughout the iterations. Note that **G** is assigned a value of 0 because there are no intentional control inputs. The function also stores the state variable estimate in a variable named *xA*, its corresponding uncertainty (variance) in variable *PA*, and the Kalman gain in a variable *KG*. By continuously updating these three variables, we can ultimately plot the evolution of the filtering process over the course of, say, 300 iterations.

```
% rocketloop - Timing loop for simulation of univariate Kalman
filter for the estimation of an inaccessible temperature in an
exhaust plume
% SYNTAX: [XAVECT, PAVECT, KGVECT] = rocketloop(xTru, zVect, x0, P0,
Q, R, iter)
function [XAVECT, PAVECT, KGVECT] = rocketloop(xTru, zVect, x0, P0,
Q, R, iter)

F = 1;
G = 0;
H = 1;

x = x0;
P = P0;

%Input time series
u = zeros(iter, 1);
z = zVect;

%Set up vectors to store outputs (all iterations)
PAVECT = zeros(iter,1);
XAVECT = zeros(iter,1);
KGVECT = zeros(iter,1);

for t = 1:iter %START OF TIMING LOOP
    [PA, xA, KG] = onedkf0(F,G,Q,H,R,P,x,u(t),z(t));
    PAVECT(t) = PA;
    XAVECT(t) = xA;
    KGVECT(t) = KG;

    %Pass results as inputs to the next iteration:
    P = PA;
    x = xA;
end % END OF TIMING LOOP
end

%Setting up the Kalman filter simulation
clc
clear all
close all

xTru = 4000; %Assuming that the actual centerline temperature is
4000 K
x0 = 3950; %Hydrodynamics suggests that centerline temp. equals 3950
K
P0 = 0.6; %Assigning an initial variance of 0.6 for the results from
the model
Q = 0.005; %Assigning a variance of 0.005 to the external noise
R = 5; %Assigning a variance of 5 to the temp. measurements
iter = 300; %No. of iterations

%Simulating
[XAVECT, PAVECT, KGVECT] = rocketsim(xTru,x0,P0,Q,R,iter);
```

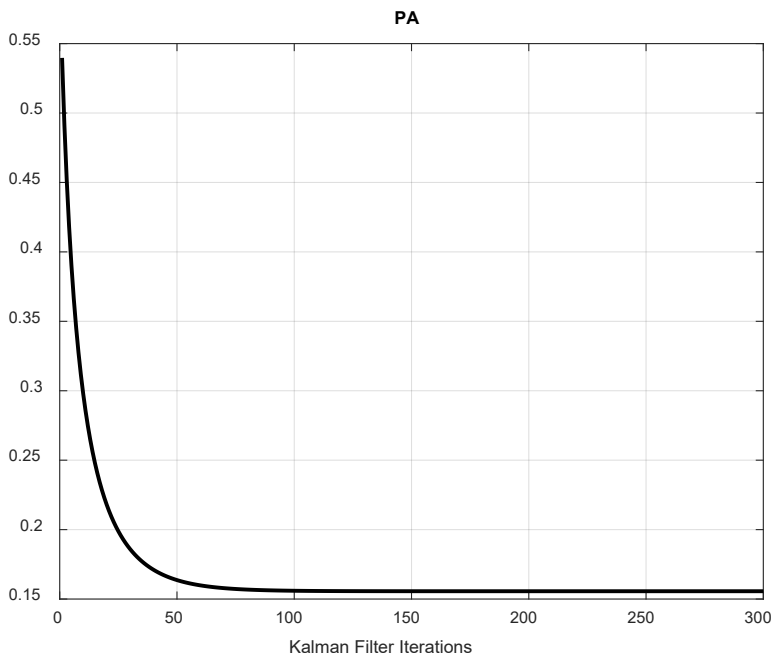
The main plot we require is shown below. The gray trace is the "true" temperature in the combustion chamber; the dot-dashed trace is the time series of measurements $\mathbf{z}(t)$; lastly, the red trace is the posterior estimate of the state variable as computed by the filter algorithm. A few things are immediately apparent. Firstly, note that the measurements are quite noisy, which encourages the modeler to resort to a state estimation algorithm such as the Kalman filter. The filtered state variable starts off at several kelvin away from the true values but, by iteration 150 or so, closely matches the actual data – much better, in fact, than the highly noisy measurements represented by the dot-dashed line.



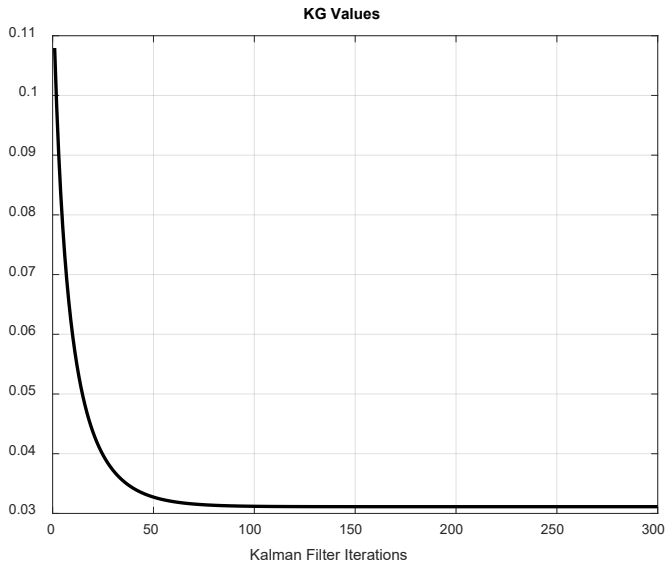
A zoomed-in capture of the plot centered at iteration No. 200 is shown below.



The foregoing code also outputs the evolution of the variance P_A, which in the present case is just a scalar. As can be seen, the variance decreases rapidly within the first 50 iterations, and eventually stabilizes at ≈0.156.



Finally, the foregoing code also yields the evolution of the Kalman gain (again, a scalar in this case), which decreases rapidly before stabilizing at ≈0.0311.



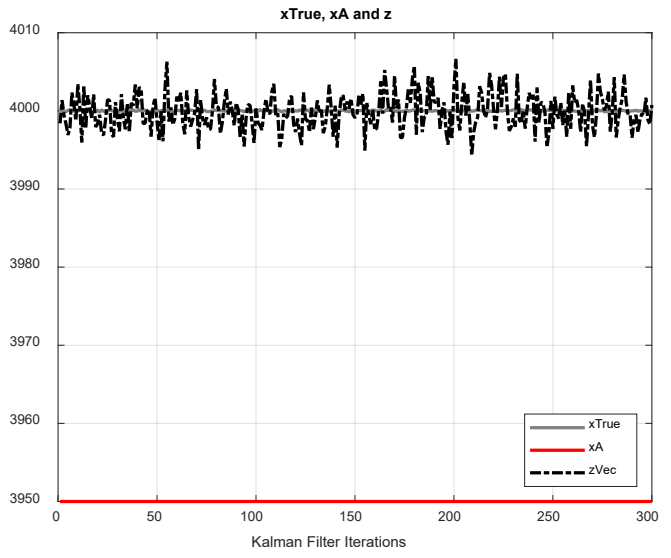
Following Barreto *et al.* (2021), we may ask what would happen if we were to void the correction phase of the Kalman filter process and ignored the information provided by the measurements. To see the effect of such a modification, refer to the final two equations of the filter scheme:

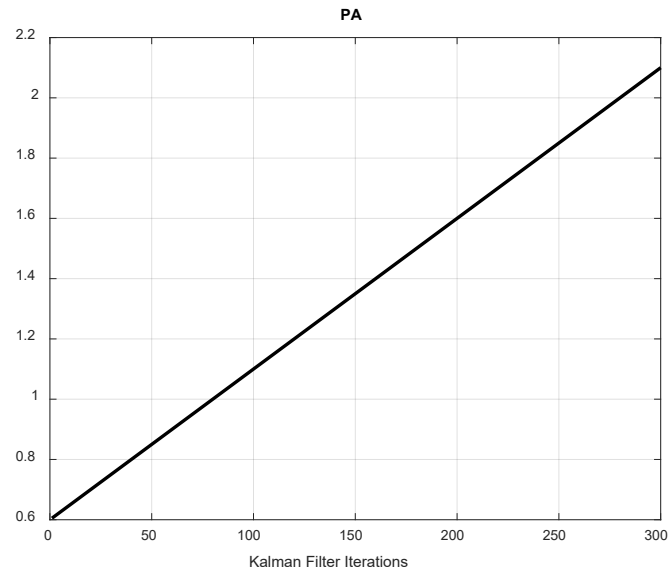
$$\mathbf{x}_\mathbf{A} = \mathbf{x}_\mathbf{B} + \mathbf{K}_\mathbf{G}\left(\mathbf{z} - \mathbf{H}\,\mathbf{x}_\mathbf{B}\right)$$
$$\mathbf{P}_\mathbf{A} = \mathbf{P}_\mathbf{B} + \mathbf{K}_\mathbf{G}\mathbf{H}\,\mathbf{P}_\mathbf{B}$$

It is easy to see that, if we restate the contents of the matrix $\mathbf{K_G}$ with all zeros, the second terms in the right-hand side of the two equations vanish and we end up with the elementary relationships $\mathbf{x_A} = \mathbf{x_B}$ and $\mathbf{P_A} = \mathbf{P_B}$; this implies that the Kalman filter is forced to rely only on the model, ignoring information from the measurements altogether. To analyze the effect of such a simplification, we may add the following line of code to the *onedkf* function, just below the line *KG = PB*HT*(inv(H*PB*T+R))*:

```
KG = zeros(size(KG));
```

and save the modified function as *onedkf0.m*, which is available in our Google Drive folder. Then, we head to *rocketloop.m* and replace the call to *onedkf* in the *for* loop with a call for *onedkf0*. Run the pertaining code and the program will output two interesting graphs, as shown in continuation:

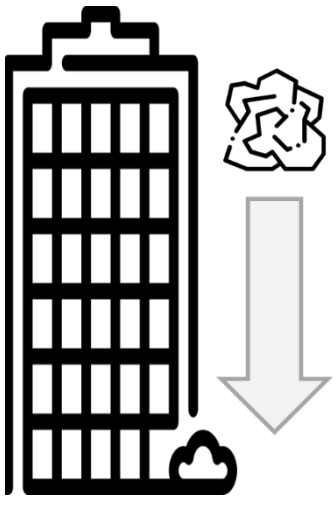




The first graph shows that, in the absence of interactions between measured data and the algorithm, the model invariably outputs the same temperature throughout the entire simulation. The second graph, in turn, shows that the uncertainty P_A in this anomalous scenario actually *increases* steadily with each iteration, in direct contrast to what we'd expect in a "normal" Kalman filter operation.

▶ Problem 3 – A multivariate example of Kalman filter in MATLAB: The falling paper wad problem

Following Barreto *et al.* (2021), the second practical example of Kalman filtering we consider is the falling paper wad. Simply put, in this problem we wish to describe the height *y* above ground level of a paper wad released from the top of a building. The state variables in this problem are the height y_k and the falling velocity y'_k at iteration *k*. These are described by the equations



$$\begin{cases} y_k = y_{k-1} + (\Delta T)\, y'_{k-1} - \dfrac{(\Delta T)^2 g}{2} \\ y'_k = y'_{k-1} - g\Delta T \end{cases}$$

where ΔT is the time interval encompassed by an iteration and $g \approx 9.81$ m/s^2. The Kalman-filter equation that describes the system is

$$\mathbf{x}(t+1) = \mathbf{F}(t)\,\mathbf{x}(t) + \mathbf{G}(t)\,\mathbf{u}(t)$$

where

$$\mathbf{x}(t) = \begin{bmatrix} y_k \\ y'_k \end{bmatrix}$$

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} -\dfrac{1}{2}(\Delta T)^2 \\ -\Delta T \end{bmatrix}$$

Substituting $\mathbf{u}(t) = g$, we can restate the two equations as

$$\mathbf{x}(t+1) = \mathbf{F}(t)\,\mathbf{x}(t) + \mathbf{G}(t)\,\mathbf{u}(t)$$

$$\begin{bmatrix} y_{k+1} \\ y'_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_k \\ y'_k \end{bmatrix} + \begin{bmatrix} -\left(\dfrac{1}{2}\right)(\Delta T)^2 \\ -\Delta T \end{bmatrix} g$$

Crucially, the fall of the object may not be driven exclusively by the acceleration of gravity. In the case of, say, a big paper wad, downward motion is resisted by air flow around and within the recesses of the wad, which has a highly irregular shape. This can be accounted for by combining the freefall gravity *g* with a mean opposing acceleration component g_{back}, and then incorporating it into the problem as an "actual" acceleration $g_{actual,k} = g - g_{back}$. This modified acceleration is associated with a standard deviation $\sigma_{\text{actual},k}$, which we denote in code as *gsd*.

The initial values of the state variables are the initial speed, which we assume to be 0, and the initial height from which we drop the paper wad, which we shall take as 100 m. In the Kalman filter framework, these values are associated with some uncertainty, which we incorporate in an initial covariance matrix $\mathbf{P_0}$, as shown in the upcoming code.

Further, we assume that the falling trajectory of the paper wad is tracked by a position-measuring instrument, which Barreto *et al.* (2021) refer to as a 'laser rangefinder'. The rangefinder is quite noisy, offering only mediocre-accuracy height data with which we shall feed the Kalman filter algorithm in the hopes of tracking the paper wad with greater precision. The measurements are assigned to a variable *z*(*t*), and the relationship between the (scalar) measurement *z*(*t*) and the state variable vector reads

$$z(t) = \mathbf{H}\mathbf{x}(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ y' \end{bmatrix} \quad \text{(I)}$$

The uncertainty in the measurement, usually represented by the covariance matrix **R**, will be just a scalar expressed as the variance of the height measurements afforded by the rangefinder. In a real situation, we'd expect that this parameter be supplied by the manufacturer of the instrument.

$$\mathbf{R} = \begin{bmatrix} \sigma_z^2 \end{bmatrix}$$

Analogously to the previous example, we will model the present system using three codes: *onedkf*, *papersim*, and *paperloop*. Function *paperloop* receives the matrices **Q** and **R**, which are fixed, as well as the sampling interval ΔT and the total number of iterations *iter*. As usual, the loop must be fed initial values for the state vector, $\mathbf{x_0}$, and for the state covariance matrix, $\mathbf{P_0}$.

Once all the arguments are received, *paperloop* creates **F** and **G** (which will not change) and computes matrix **H**. Within the timing loop, *paperloop* calls *onekdf* for each iteration of the Kalman filter, writing each pair of iteration results $\mathbf{x_A}$ and $\mathbf{P_A}$ on the variables *x* and *P*, which will be used in the next call to *onekdf*. Further, before the end of the timing loop, there are commands that save not only $\mathbf{x_A}$ to return all the state vectors calculated to *papersim*, but also the two values of the Kalman gain matrix, $\mathbf{K_G}$, and the value in the first row and first column of the matrix $\mathbf{P_A}$ calculated in each iteration. That position in $\mathbf{P_A}$ is occupied by the variance associated with the first state variable, which in the case at hand is the estimated height of the paper wad – namely, the main state variable of interest.

Function *papersim* receives the parameters that a user may wish to change from one simulation to another, such as the assumed parameters for the "diminished downward acceleration," whose mean value is set by subtracting g_{back} from 9.81 m/s^2 and its standard deviation, stored in the MATLAB variable *gsd*. Also included in *papersim* are the matrix **R**, the initial state-variable vector $\mathbf{x_0}$, the initial covariance matrix $\mathbf{P_0}$, the sampling interval ΔT, the "true initial height of the fall" *y0tr*, and the total number of iterations *iter*.

Function *papersim* creates the external error covariance, $\mathbf{Q}(t)$, which will be fixed, as the covariance matrix for **u**(t), the 2×1 vector that appears as the rightmost term in the filter equation (I) above. To express all values of **Q** as functions of the variance of g_{actual}, σ^2_{actual}, we will employ the following property of the covariance function:

$$COV\left(ax_1, bx_2\right) = ab \times COV\left(x_1, x_2\right)$$

Thus, for vector **u**, which is expressed as

$$\mathbf{u}_k = \begin{bmatrix} -\left(\dfrac{\Delta T^2}{2}\right) g_{\text{actual,k}} \\ -\left(\Delta T\right) g_{\text{actual,k}} \end{bmatrix}$$

the corresponding covariance matrix **Q** becomes

$$\mathbf{Q} = \begin{bmatrix} \left(\dfrac{\Delta T^4}{4}\right)\sigma^2_{\text{actual,g}} & \left(\dfrac{\Delta T^3}{2}\right)\sigma^2_{\text{actual,g}} \\ \left(\dfrac{\Delta T^3}{2}\right)\sigma^2_{\text{actual,g}} & \left(\Delta T^2\right)\sigma^2_{\text{actual,g}} \end{bmatrix}$$

Thus, *papersim* can create a suitable matrix **Q**, on the basis of the sampling interval ΔT and the variance of the fluctuations $\sigma^2_{\text{actual,g}}$.

Upon execution, *papersim* first creates all the necessary time series that will be accessed by *onedkf* and then, after the invocation of *paperloop*, performs some basic visualization of the results.

The computation of a vector containing *iter* samples of the "true height" of the falling paper wad is achieved by iterating over the basic freefall model, where the values of the "diminished downward acceleration" are fetched from the previously created sequence of actual values $g_{\text{actual},k}$. This sequence is generated by adding noise samples from a normal distribution with 0 mean and

standard deviation *gsdva* to the mean value, which is $g - g_{back}$. The values used as the **u** vector will be created from the values of $g_{actual,k}$ using the expressions contained in the 2×1 column vector provided above. Then, assigning as the true initial height *y0tr* and as initial speed 0, the "true" values of height and speed are calculated for *iter* iterations. All the "true values" of height are collectively called *ytr*.

The values in the vector of heights that the rangefinder would report, *z*, are calculated by adding to the "true height" vector samples from a normal distribution with 0 mean and standard deviation $\sqrt{\sigma_z^2}$.

After *paperloop* is called, the matrices returned, namely *XAVECT*, *PAVECT*, and *KGVECT*, can be used for some basic display of the results, similarly to what we did in the previous problem. In a first graph, *papersim* displays the "true heights" available in *ytr*, superimposed with the time series of heights estimated (as the first state variable) by the Kalman filter. In a second graph, *papersim* plots the evolution of the element in the first row and first column of *PA*, which is the variance of the height estimates that the Kalman filter produces as the first element of the posterior state vector, *xA*. This graph is important because we would like to see that the uncertainty of the height estimates produced by the Kalman filter is reduced as the algorithm carries out the calculations. A third graph shows the evolution of the first element of the *KG* matrix for further analysis.

```
% papersim - Function to simulate the fall of a paper wad taking
into account variable air resistance and implementing Kalman filter
to obtain height estimates.
% SYNTAX: [XAVECT, PAVECT, KGVECT] =
papersim(gback,gsd,y0tr,x0,P0,R,DT,iter);

function [XAVECT, PAVECT, KGVECT] =
papersim(gBack,gsd,y0tr,x0,P0,R,DT,iter)
rng(12345, 'v5normal'); %Resets the random number generator

%Calculate true heights with variable air friction
g = 9.81;

gsd2 = gsd^2; %Variance of the fluctuations in actualg
DT2 = DT^2;
DT3 = DT^3;
DT4 = DT^4;

%Creating matrix Q
Q = [(gsd2*DT4/4), (gsd2*DT3/2); (gsd2*DT3/2), (gsd2*DT2)];

noiseg = randn(1,iter)*gsd; %Creating the fluctuations for actualg

actualg = (ones(1,iter)*(g-gBack)) + noiseg;

ytr = zeros(1,iter);

%Create the "true" time series of heights ytr
F = [1, DT; 0, 1];
G = eye(2);
H = [1, 0];

%Create u(t) in advance
u11coeff = DT2/(-2);
u21coeff = (-1)*DT;
u = zeros(2,iter);
for t = 1:iter
    u(:,t) = [(u11coeff*actualg(t)); (u21coeff*actualg(t))];
end

%Create "TRUE" height series
y = [y0tr; 0];
for t = 1:iter
    yNext = F*y + G*u(:,t);
    ytr(1,t) = yNext(1,1); %Preserve in vector ytr only the first
value in yNext, which is the height
    y = yNext; %Feed back the result in the model for next iteration
end

%Create a z time series with the laser height measurements,
including measurement noise
mnoise = randn(1, iter)*(-sqrt(R));
```

```
z = ytr + mnoise;

%Run the timing loop
[XAVECT, PAVECT, KGVECT] = paperloop(z,u,x0,P0,Q,R,DT,iter);
%Plot some results
heightFromKF = XAVECT(1,:);
gray6 = [0.6, 0.6, 0.6];
figure;
plot(z,'Color',gray6);
hold on
plot(heightFromKF, 'r', 'LineWidth', 1.8);
plot(ytr, 'y', 'LineWidth', 1.5);
hold off; grid
title('True height, KF-estimated height and rangefinder values');
ylabel('meters')
xlabel('Kalman Filter iterations')
legend('z', 'HeightFromKF', 'ytr', 'Location', 'southwest');

%Studying the evolution of the variance of the y's
%estimate(xA(1,1))
figure;
plot(PAVECT, 'k', 'LineWidth', 1.5); grid;
title('Variance of KF-estimated height');
ylabel('Square Meters');
xlabel('Kalman Filter Iterations');

%Now plotting the evolution of the 1st element in KG
figure;
plot(KGVECT(1,:),'k','LineWidth',1.5); grid;
title('Evolution of the first element of KG (KG1) in this example');
xlabel('Kalman Filter iterations');

end

% paperloop - Timing loop for simulation of Kalman filter applied to
% falling motion of a paper wad
% SYNTAX: [XAVECT, PAVECT, KGVECT] =
paperloop(zVect,u,y0,P0,Q,R,DT,iter)
function [XAVECT, PAVECT, KGVECT] = paperloop(zVect, u, y0, P0, Q,
R, DT, iter)

F = [1, DT; 0, 1];
G = eye(2);
H = [1, 0];

x = y0;
P = P0;
%Measurement time series
z = zVect;

%Set up vectors to store selected elements of xA and PA from all
iterations
PAVECT = zeros(1,iter); %We will only store the variance of yk from
PA
XAVECT = zeros(2,iter); %We will store both yk and y'k
KGVECT = zeros(2,iter); %We will store both values in KG, which will
be a 2x1 vector

for t = 1:iter %START OF TIMING LOOP
    [PA, xA, KG] = onedkf(F,G,Q,H,R,P,x,u(:,t),z(t));

    PAVECT(t) = PA(1,1); %We are only storing the variance of the
first state variable, which is yk, located in cell (1,1) of PA
    XAVECT(:,t) = xA; %We will store the estimates of both state
variables contained in xA
    KGVECT(:,t) = KG;

    %Pass results as inputs to the NEXT iteration
    P = PA;
    x = xA;
end %END OF TIMING LOOP
end
```

As a sample application of the codes listed above, we consider a paper wad being released from the top of a building of height equal to 120 meters. The modeler is not equipped with this value and hence places the initial height at a value of, say, 100 m. The paper wad begins its falling trajectory with a velocity equal to zero. Thus, we may type

```
x0 = [100; 0];
y0tr = 120;
```

(The second entry in column vector $\mathbf{x_0}$ is the initial velocity, which is zero). Next, we set up the initial covariance matrix, noting that the uncertainty associated with speed, which we know to be initially zero, is much less than that associated with the height of the building. Arbitrary values are chosen; feel free to choose different values if you wish.

```
P0 = [12, 0; 0, 0.03]; %The variance for speed is lower; we are
confident that 0 is the initial speed
```

We then use the following values to estimate the mean level and variability of the acceleration time series (also arbitrary):

```
gBack = 0.095; %Mean value will be g - gBack = 9.81 - 0.095 = 9.715
gsd = 0.0095; %Variability will be represented by std. deviation of
0.0095
```

As the first state variable represents height in meters and the rangefinder measures height in meters:

```
R = 1;
```

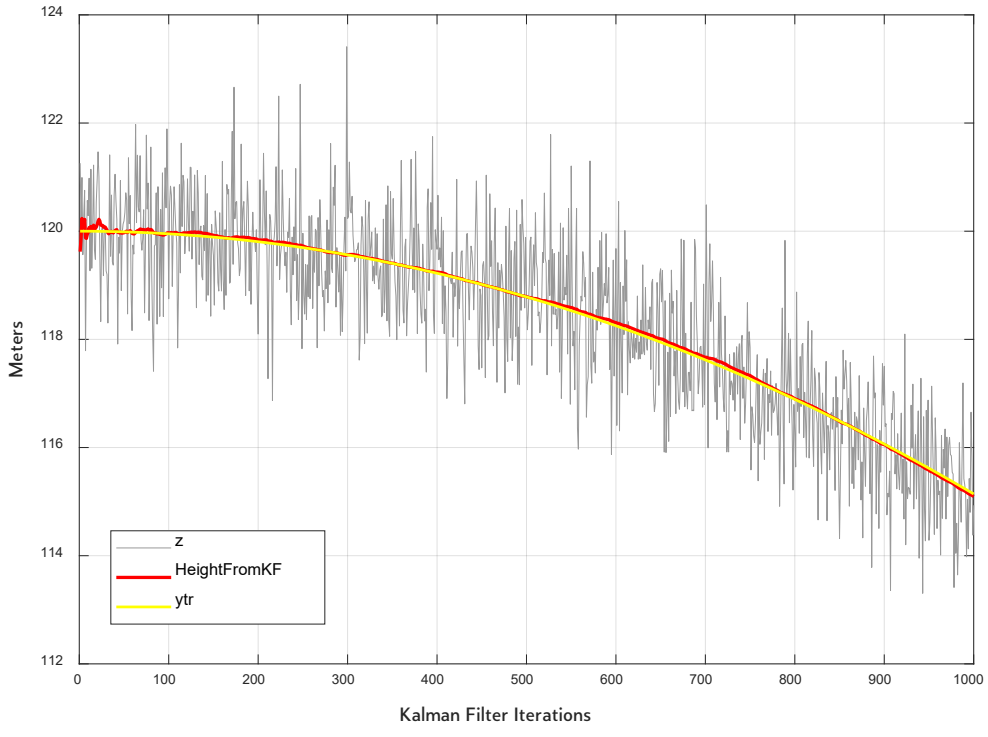
Following Barreto *et al.* (2021), we shall employ a sampling interval of 1 millisecond and set up 1000 iterations, totaling 1 second of simulation time:

```
DT = 0.001;
iter = 1000;
```

The final step is to call function *papersim*:

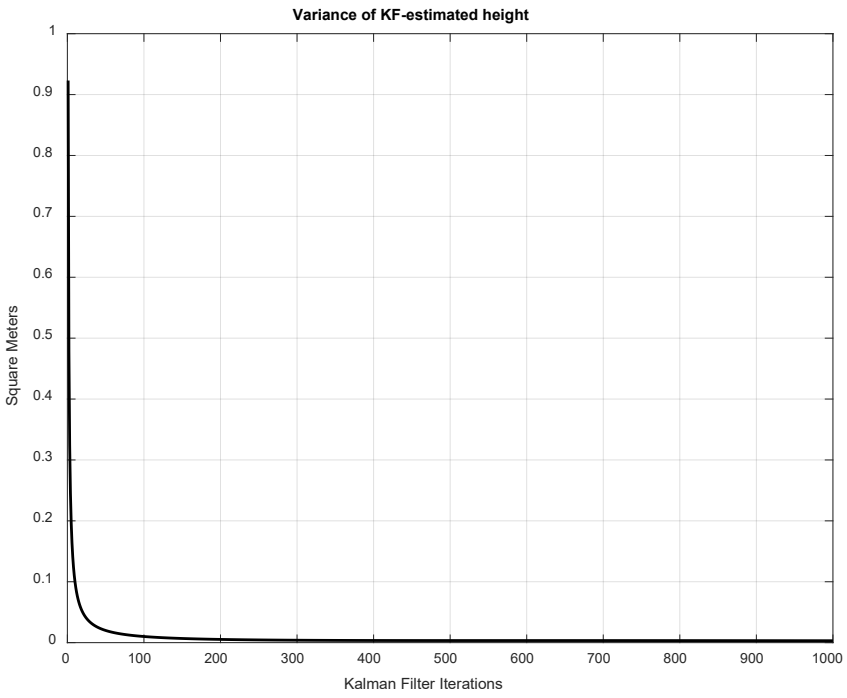
```
[XAVECT, PAVECT, KGVECT] = papersim(gBack, gsd, y0tr, x0, P0, R, DT,
iter);
```

The main output graph is the height versus No. of iterations plot shown next.



The gray trace is the series of rangefinder measurements; the yellow trace is the evolution of "true" height; the red trace is the Kalman-filtered height. As can be seen, while the initial estimate for the height was mistaken by several meters, the height value provided by the Kalman filter almost instantaneously adjusts itself to track the plot of the true heights with excellent accuracy. Also visible is the fact that whereas the trace displaying the rangefinder measurements is quite noisy, the estimate obtained with the Kalman filter is free from violent oscillations.

The *papersim* call also outputs the variance associated with the Kalman filter estimate of height, as shown below. Clearly, while the variance associated with the KF-estimated height is initially high, it drops precipitously within about 100 iterations and remains close to zero for the remainder of the simulation. Similar findings apply to the first element of the Kalman gain $\mathbf{K_G}$, which starts at a high value and then stabilizes within ≈200 iterations.

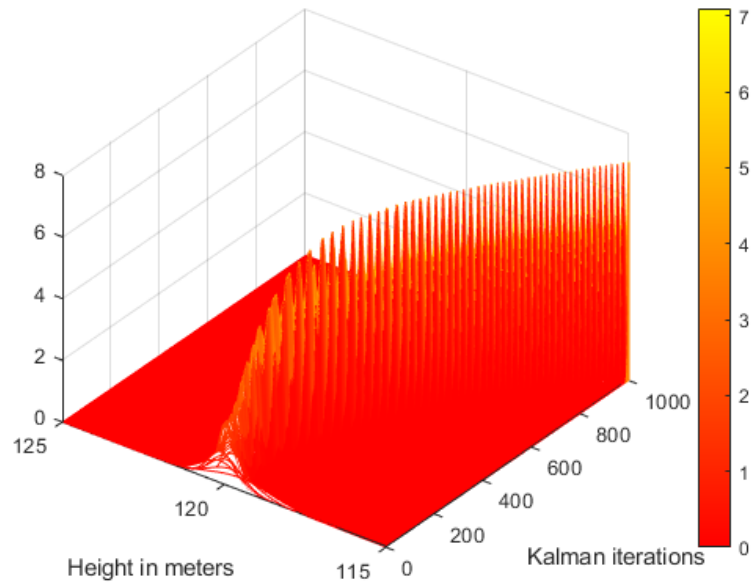


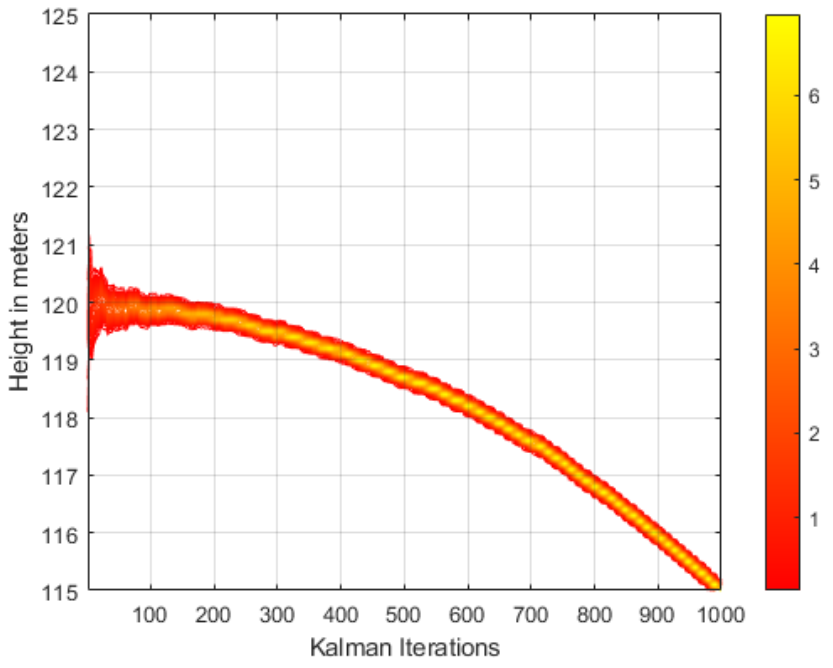
To further emphasize the significant reduction of the uncertainty in the posterior estimate of height afforded by successive iterations of the Kalman filter, it is instructive to create a display of the probability density functions representing the height of the falling paper wad. We know the characteristics (mean value and variance) of these distributions because we have access to the evolution of the first element of $\mathbf{x_A}$ and the evolution of the element in row 1 and column 1 of $\mathbf{P_A}$. The following code extracts these values from matrices *XAVECT* and *PAVECT* and uses them to create the corresponding Gaussian distributions through the function *calcgauss*, which appears in Chapter 7 of Barreto *et al.* (2021) and can be found in the Appendix at the end of this tutorial; it is also available in our Google Drive folder. The Gaussian profiles are arranged in a matrix named *WATFALL*, which is then displayed as a MATLAB waterfall plot with the eponymous command.

```
esth = XAVECT(1,:);
sdh = sqrt(PAVECT(1,:));
szeh = length(esth);
n = linspace(0, (szeh-1), szeh);

%Plot the height estimates waterfall plot:
hMin = 115;
hMax = 125;
hNumGauss = ((hMax-hMin)*10) + 1;
hStep = (hMax-hMin)/(hNumGauss-1);

WATFALL = zeros(hNumGauss, szeh);
for t = 1:szeh
    [valsx, resgauss] = calcgauss(hMin, hNumGauss, hMax, esth(t),
sdh(t));
    WATFALL(:,t) = resgauss;
end

%Create a mesh grid for waterfall contour plots
[TIME,HEIGHT] = meshgrid(1:1:szeh, hMin:hStep:hMax);
```

```
%Waterfall plot (following MATLAB instructions for "column-oriented
%data analysis")
figure
waterfall(TIME', HEIGHT', WATFALL');
colormap('autumn'); colorbar;
xlabel('Kalman iterations');
ylabel('Height in meters');

figure
contour3(TIME', HEIGHT', WATFALL', 50);
view(2)
colormap('autumn'); colorbar;
xlabel('Kalman Iterations');
ylabel('Height in meters');
```

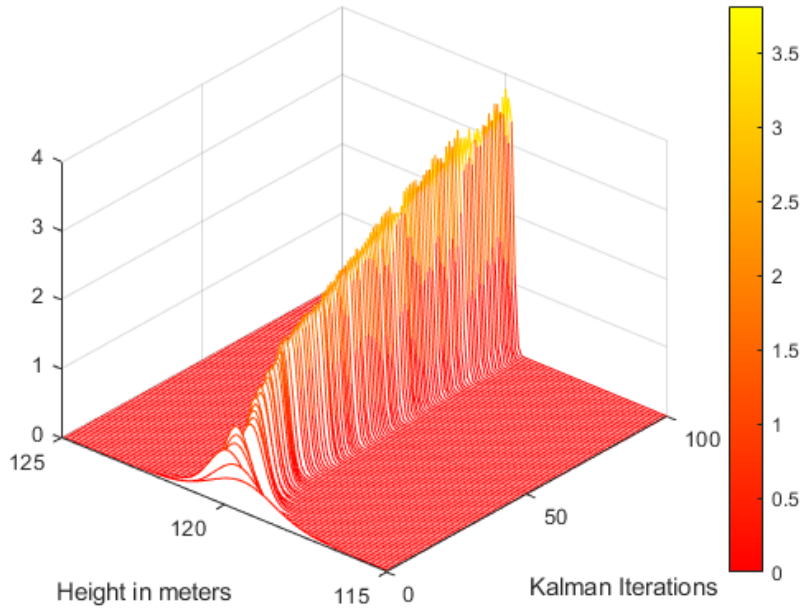
The following figure demonstrates that while the distribution associated with the posterior estimate of height starts up in a rather broad Gaussian curve, implying considerable uncertainty in results, it quickly narrows down over the course of subsequent iterations, implying much-diminished uncertainty. This is made clearer in the second output, which is an upper view of the waterfall plot.





We can see the evolution of the Gaussian curves more clearly by creating a waterfall for the initial 100 iterations only:

```
TIME100 = TIME(:, 1:100);
HEIGHT100 = HEIGHT(:, 1:100);
WATFALL100 = WATFALL(:,1:100);

figure;
waterfall(TIME100', HEIGHT100', WATFALL100');
colormap('autumn'); colorbar;
xlabel('Kalman Iterations');
ylabel('Height in meters');
```



APPENDIX – The *calcgauss* function (Barreto *et al.*, 2021)

This file is also available in our Google Drive folder.

```
%calcgauss.m
function [valsx, resgauss] = calcgauss(startx, numofx, endx, mu,
sigm)
gapx = (endx - startx)/(numofx - 1);
valsx = zeros(numofx, 1);
for i = 1:numofx
    valsx(i,1) = startx + (i*gapx);
end

coef = 1/(sqrt(2*pi*sigm^2));
dnm = 2*sigm^2;

resgauss = coef .* exp(((-1)*(valsx-mu).^2)./dnm);
end
```

REFERENCE

- BARRETO, A., ADJOUADI, M., ORTEGA, F.R. and O-LARNNITHIPONG, N. (2021). *Intuitive Understanding of Kalman Filtering with MATLAB*. Boca Raton: CRC Press.