CHAPTER 12

확장 칼만 필터

칼만 필터가 개발된 지는 반세기가 넘어가지만 칼만 필터는 여전히 인기 있는 연구주제로, 아직도 많은 논문이 발표되고 있습니다. 주로 새로운 응용 사례를 다루는 연구가 많지만, 칼만 필터 알고리즘 자체를 개선하고 확장하는 이론 연구도 계속되고 있습니다. 이론적인 연구는 대부분 칼만 필터를 비선형 시스템에 적용하기 위한 시도라고 보면 틀림 없습니다.

그런데 칼만 필터를 비선형 문제에 적용하려는 연구가 여전히 활발한 까닭은 무엇일까요? 그 이유는 우리 주변에 존재하는 시스템 대부분이 비선형이기 때문입니다. 칼만 필터는 선형 시스템을 가정하고 개발된 알고리즘이라 비선형 시스템에는 적용할 수 없습니다. 비선형 시스템에 칼만 필터를 쓸 수 없다면 칼만 필터의 활용 범위는 대단히 제한될 수 밖에 없습니다. 다행히 초기 연구자들은 칼만 필터의 이러한 한계를 잘 알았고, 그 해결책도 찾아냈습니다. 이 문제를 극복하지 못했다면 칼만 필터는 그냥 묻혀 버렸거나 그 가치를 인정받는 데 오랜 시간이 필요했을지도 모릅니다. 칼만 필터를 비선형 시스템까지 확장하는 방법이 개발된 덕분에 칼만 필터는 발표되자마자 급속히 확산되게 됩니다. 이때 개발된 알고리즘이 **확장 칼만 필터**extended Kalman filter (이하 **EKF**)입니다.

세계 최초로 실제 문제에 구현된 칼만 필터도 선형 칼만 필터가 아니라 EKF였습니다. 그만큼 EKF의 역사는 깊습니다. 만약 비선형 시스템에 칼만 필터를 적용해야 한다면, 가장 먼저 EKF를 고려해보기 바랍니다. EKF는 오랜 세월 동안 검증되고 널리 알려져 왔기에 관련 자료가 많고 찾기도 쉽습니다.

그런데 EKF에는 큰 단점이 있습니다. 바로 알고리즘이 발산할 위험이 있다는 점입니다. EKF를 적용할 때는 이 점에 유의해야 합니다. EKF 이후에 개발된 알고리즘은 대부분 이러한 문제를 극

복하기 위한 시도라고 해도 과언이 아닙니다.

12.1 선형화 칼만 필터

EKF를 설명하기 전에 먼저 **선형화 칼만 필터**linearized Kalman filter를 소개하겠습니다. 선형화 칼만 필터는 [그림 4-1]에서 소개한 선형 칼만 필터[1] 알고리즘과 완전히 동일합니다. 즉 선형 칼만 필터 설계와 전혀 다를 게 없습니다. 그런데 왜 굳이 선형화 칼만 필터라는 별개의 이름을 붙였을까요? 선형화 칼만 필터와 선형 칼만 필터는 시스템 모델에서 약간 차이가 있기 때문입니다.

선형 칼만 필터의 시스템 모델은 원래부터 선형인데 반해, 선형화 칼만 필터는 비선형 시스템을 하나의 기준점 주위에서 선형화시켜 얻은 선형 모델 A와 H를 사용합니다. 즉 비선형 모델을 선형화한 다음, 이 선형 모델에 대해 설계한 칼만 필터가 바로 선형화 칼만 필터입니다. 칼만 필터의 설계자 입장에서는 시스템 모델이 원래 선형이었는지 아니면 비선형 모델을 선형화시킨 모델인지는 전혀 중요치 않습니다. 두 경우 모두 칼만 필터를 설계하는 과정이 똑같기 때문입니다. 그래서 선형화 칼만 필터의 설계는 특별히 주의해야 할 사항이 없습니다.

하지만 선형화 칼만 필터를 실제 시스템에 적용할 때는 시스템의 운용 범위에 각별히 유의해야 합니다. 선형화 모델은 기준점 근처에서만 실제 시스템과 비슷한 특성을 보이기 때문입니다. 이 범위를 벗어나면 선형화 모델은 더 이상 믿을 수 없게 됩니다. 선형화의 개념에 대해 생각해 보면 당연한 이야기입니다. 선형화 모델의 이런 한계를 무시하고 선형화 칼만 필터를 함부로 적용하면 제대로 된 성능을 내지 못하게 됩니다. 심하면 전체 시스템이 발산할 수도 있습니다. 선형화 칼만 필터를 선형 칼만 필터와 구분 지어 설명하는 이유도 이 때문입니다.

결국 선형화 칼만 필터의 성공과 실패를 가르는 핵심 요소는 비선형 시스템의 선형화 모델입니다. 좋은 선형화 모델이 좋은 성능으로 이어집니다. 따라서 어떻게 선형화 모델을 구해낼지가 중요한데, 여기서는 이 문제를 따로 다루지 않습니다. 칼만 필터와는 다소 동떨어진 주제이고, 각

1 지금부터는 선형 모델의 칼만 필터를 '선형 칼만 필터'라고 부르겠습니다.

문제마다 적절한 선형화 방법이 다 다르기 때문입니다.

요약하면 선형화 칼만 필터의 경우 그 설계 과정에서는 특별한 점이 없지만, 실제 시스템에 적용할 때는 선형화 모델이 유효한 범위 내에서만 사용되어야 합니다. 그렇지 않으면 칼만 필터의 성능을 보장할 수 없습니다.

12.2 EKF 알고리즘

이 절에서는 EKF의 알고리즘에 대해 알아봅니다. EKF와 [그림 4-1]의 선형 칼만 필터를 비교하여 두 알고리즘이 어떻게 다른지 설명하겠습니다. EKF는 칼만 필터를 비선형 시스템까지 확대 적용하는 대표적인 알고리즘으로, 오랜 세월 동안 이론과 실전에서 충분히 검증된 방법입니다. 따라서 수식 유도에 대해 고민할 필요 없이 그냥 알고리즘대로 구현하면 됩니다.

■ 비선형 시스템 모델

먼저 시스템 모델을 알아보겠습니다. EKF 알고리즘은 다음과 같은 비선형 시스템 모델을 고려합니다.

$$x_{k+1} = f(x_k) + w_k \tag{12.1}$$

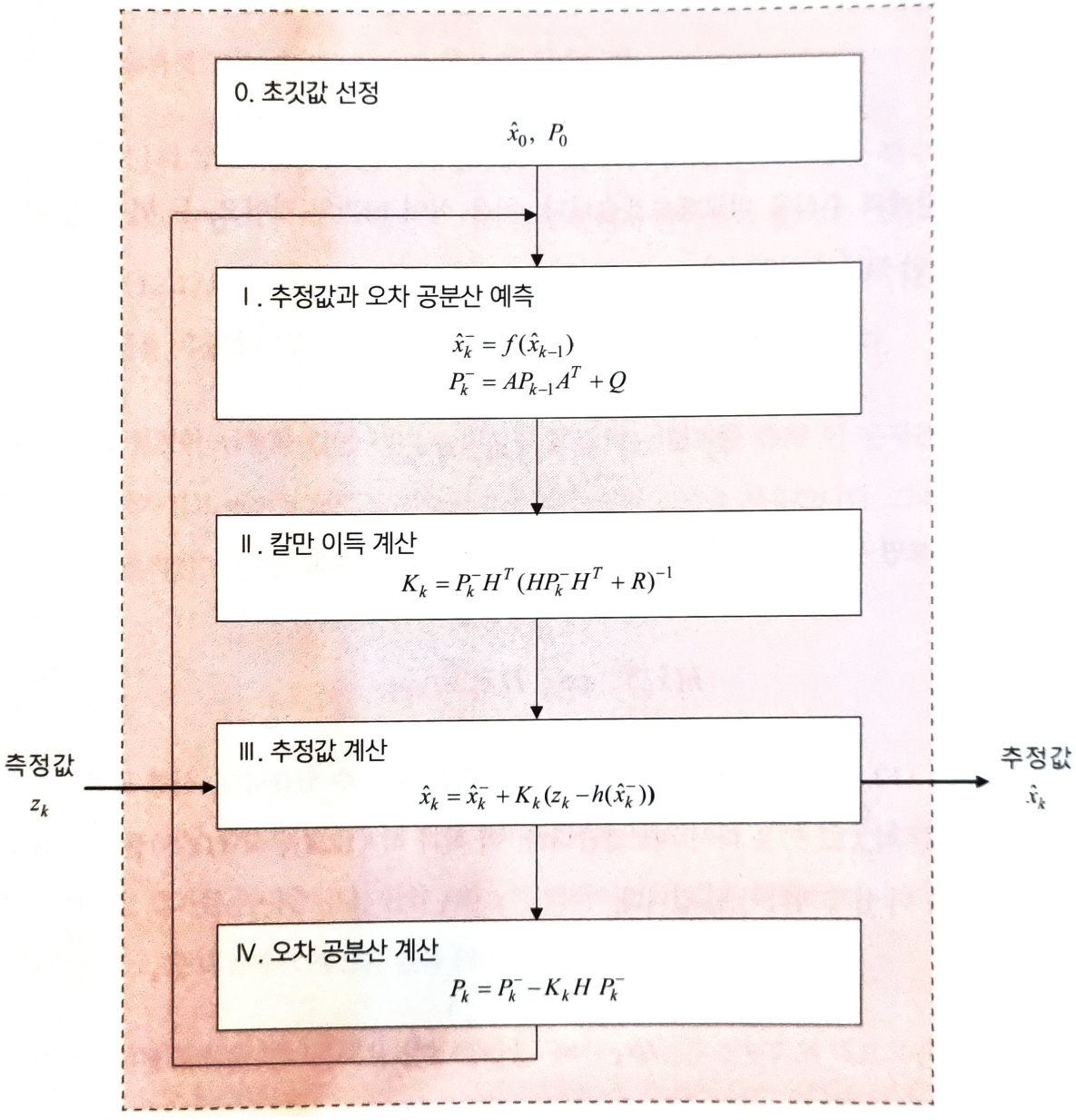
$$z_k = h(x_k) + v_k \tag{12.2}$$

위의 모델이 지금까지 봐온 선형 모델과 다른 점은 상태 변수(x_k)와 계수를 따로 분리해서 쓰지 못한다는 점입니다. 위의 비선형 모델을 식 (7.1)과 (7.2)의 선형 모델과 비교해 보면, 아래와 같이 선형 행렬식이 비선형 함수로 바뀌었음을 알 수 있습니다. 그 밖에 다른 변수의 의미와 형태는 똑같습니다.

$$\begin{aligned} Ax_k &\Rightarrow f(x_k) \\ Hx_k &\Rightarrow h(x_k) \end{aligned} \tag{12.3}$$

■ EKF 알고리즘

그러면 EKF 알고리즘을 구체적으로 알아보겠습니다. [그림 12-1]에 EKF의 알고리즘을 정리했습니다. 알고리즘의 전체 과정은 [그림 4-1]의 선형 칼만 필터 알고리즘과 동일한데, 단계별 수식에서 약간 차이가 있습니다. 이미 익숙한 알고리즘이니 차이점을 찾는 것은 그리 어렵지 않을 것입니다.



[그림 12-1] EKF 알고리즘

차근차근 하나씩 살펴보겠습니다. 먼저 Ⅰ단계의 상태 변수 예측값을 비교해 보겠습니다. 첫 번째가 EKF의 관계식이고 두 번째가 선형 칼만 필터의 식입니다.

$$\hat{x}_{k+1}^{-} = f(\hat{x}_k)$$
$$\hat{x}_{k+1}^{-} = A\hat{x}_k$$

두 식을 비교해보면 우변이 서로 다릅니다.

$$f(\hat{x}_k) \quad \Leftrightarrow \quad A\hat{x}_k \tag{12.4}$$

마찬가지로 Ⅲ단계의 수식을 비교해보겠습니다. 처음 식이 EKF의 것이고, 두 번째가 [그림 4-1]의 선형 칼만 필터 수식입니다.

$$\hat{x}_k = \hat{x}_k^{-} + K_k\left\{z_k - h(\hat{x}_k^{-})\right\}$$
$$\hat{x}_k = \hat{x}_k^{-} + K_k\left\{z_k - H\hat{x}_k^{-}\right\}$$

두 식을 비교해 보면 우변의 마지막 항만 다르고 나머지는 똑같습니다.

$$h(\hat{x}_k^{-}) \quad \Leftrightarrow \quad H\hat{x}_k^{-} \tag{12.5}$$

식 (12.4)와 식 (12.5)를 곰곰이 살펴보면 일정한 규칙을 찾을 수 있습니다. 선형 모델과 비선형 모델을 비교한 식 (12.3)을 다시 써보겠습니다. 이 식과 식 (12.4), 식 (12.5)를 같이 놓고 비교해보시면 좀 더 쉽게 파악이 될 겁니다.

$$Ax_k \quad \Rightarrow \quad f(x_k)$$
$$Hx_k \quad \Rightarrow \quad h(x_k)$$

그렇습니다. EKF는 선형 알고리즘의 모델식 Ax_k와 Hx_k 자리에 비선형 모델의 관계식을 사용합니다. 사실 생각해 보면 지극히 당연한 결과입니다. 시스템 모델이 달라졌는데, 알고리즘에서 사용하는 시스템 모델식도 따라 바뀌는 게 당연하지 않겠습니까?

그 밖에 또 어떤 차이가 있는지 찾아볼까요? 그렇다고 정말로 찾아보지는 마세요. [그림 4-1]과 [그림 12-1]에서는 더 이상 다른 곳이 없습니다. 나머지 수식은 모두 같습니다. 그럼 EKF는 식 (12.4)와 (12.5)처럼 선형 모델을 비선형 모델식으로 바꾸기만 하면 되는 것일까요? 그렇지는 않습니다. 숨겨져 있는 중요한 차이가 하나 더 있습니다.

[그림 12-1]의 알고리즘을 보면 미리 결정되어 있어야 하지만 아직 모르는 변수가 있습니다. 시스템 행렬 A와 H입니다. 이 두 행렬은 시스템 모델에서 받아와야 합니다. 그런데 EKF의 시스템 모델인 식 (12.1)과 (12.2)에는 이 두 행렬이 보이지 않습니다. 따라서 식 (12.1)과 (12.2)의 비선형 모델을 이용해 어떻게든 선형 모델의 행렬 A와 H를 유도해야 합니다.

앞 절에서 설명한 선형화 칼만 필터는 비선형 모델의 선형화를 통해 이 문제를 해결했습니다. EKF도 마찬가지로 비선형 모델을 선형화시켜 얻은 선형 모델을 사용합니다. 그리고 선형화 방법도 아주 고전적인 기법을 그대로 이용합니다.

$$A \equiv \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_k}, \quad H \equiv \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_k^-} \tag{12.6}$$

여기서 $\frac{\partial}{\partial x}$는 해당 함수를 변수 x에 대해 편미분한다는 뜻입니다. 따라서 $\frac{\partial f}{\partial x}$는 함수 f를 x에 대해 편미분한 결과를 의미합니다. 만약 f와 x가 벡터이면 이 값은 행렬이 됩니다[2]. 그리고 $\left.\frac{\partial f}{\partial x}\right|_{\hat{x}_k}$는 행렬 $\frac{\partial f}{\partial x}$에 $\hat{x}_k$를 대입해서 계산한 값을 의미합니다.

이렇게 비선형 모델을 편미분해서 선형 행렬을 구하는 방법은 선형화의 대표적인 기법으로, EKF

2 이렇게 계산하는 행렬을 야코비안(Jacobian)이라고 합니다.

만의 고유한 특성은 아닙니다. EKF가 선형화 과정을 거친다는 점은 선형화 칼만 필터와 같지만, EKF는 직전 추정값 $\hat{x}_k$를 선형화의 기준으로 삼는다는 점에서 선형화 칼만 필터와 차이가 납니다. EKF를 적용할 때는 이 차이를 굳이 알지 못해도 상관 없지만, 알아두면 유용하게 쓰일 때가 있으니 간단히 설명하겠습니다. 특히 선형화 칼만 필터를 사용해야 할지 아니면 EKF를 써야 할지를 판단할 때 이 내용이 도움이 됩니다.

EKF는 미리 결정된 선형화 기준을 사용하지 않고, 직전 추정값($\hat{x}_k$)을 기준으로 삼아 매번 새로 선형 모델을 구합니다. 시스템의 실제 상태와 가장 가까운 값을 직전 추정값으로 보고, 이 값을 기준으로 선형 모델을 계산하는 것입니다. 따라서 이 방식은 선형화의 기준점을 사전에 설정하기 어려운 시스템에 적합합니다. 반면 선형화의 기준을 이미 알고 있다면 굳이 이렇게 할 필요는 없습니다. 선형화 칼만 필터를 사용하면 됩니다. 일정 궤도를 도는 위성이나 궤적이 미리 정해져 있는 위성 발사체 등이 이런 경우에 해당됩니다.

설명이 다소 어려워졌습니다. 그냥 간단하게 EKF는 직전 추정값을 기준으로 선형 모델을 계산한다는 정도로만 정리해 두겠습니다. 자세한 내용은 EKF의 알고리즘을 배우고 실제로 설계해보면 차차 이해할 수 있을 것입니다. 어쨌든 핵심은 EKF에서는 행렬 A와 H를 식 (12.6)과 같이 계산한다는 점입니다.

지금까지의 설명을 정리해 보겠습니다. EKF 알고리즘의 전체 과정은 선형 칼만 필터 알고리즘과 같습니다. 각 단계별 계산식은 약간 다른데, 크게 두 가지 차이점이 있습니다. 먼저, EKF는 선형 모델식 Ax_k와 Hx_k 자리에 비선형 시스템 모델식 $f(x_k)$와 $h(x_k)$를 사용합니다. 두 번째로 EKF는 비선형 모델의 야코비안으로 행렬 A와 H를 구한다는 점입니다. 이때 야코비안 행렬은 직전 추정값($\hat{x}_k$)을 기준으로 계산합니다. 그 밖에 다른 계산식은 선형 칼만 필터 알고리즘과 동일합니다.

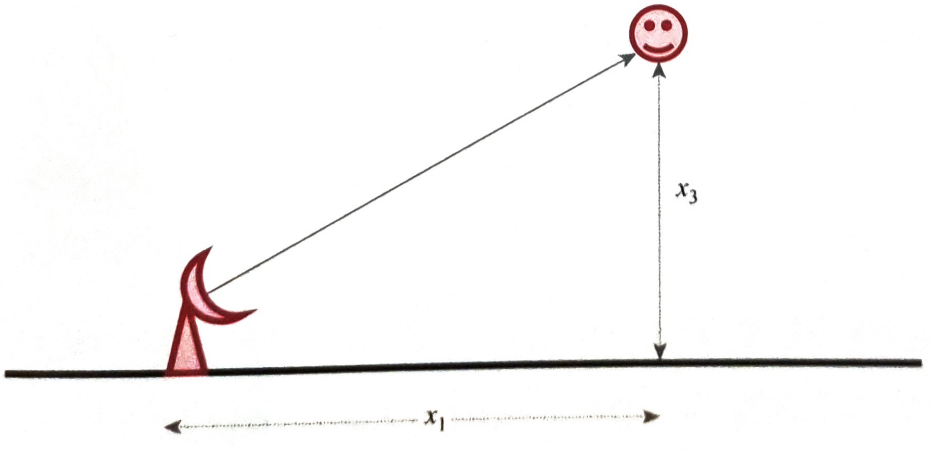
어떻습니까? 생각보다 간단하지 않나요? 선형 칼만 필터 알고리즘을 이해했다면 그리 어렵지 않았을 겁니다. 만약 아직도 잘 정리되지 않는다면 두 알고리즘의 차이점에 유의해서 [그림 12-1]을 찬찬히 다시 한번 살펴보시기 바랍니다. 절대 복잡하거나 어려운 알고리즘이 아닙니다.

그럼 간단한 예제를 통해 EKF의 실제 구현에 대해 알아보겠습니다. 소개할 예제는 두 개입니다. 첫 번째 예제는 레이다로 물체까지의 거리를 측정해서 이동 속도와 고도를 추정해내는 문제입니다. 측정 모델이 비선형인 시스템에 EKF를 설계하는 방법을 소개합니다. 두 번째로는 앞 장에서 소개한 자이로와 가속도계를 융합해서 기울기 자세를 추정하는 문제를 EKF로 해결해 보겠습니다. 이 예제는 측정 모델은 선형인데 시스템 모델은 비선형인 시스템입니다.

알고리즘의 큰 틀은 선형 칼만 필터와 다르지 않기 때문에 구현 코드가 앞의 예제와 비슷합니다. 선형 칼만 필터와 EKF의 차이점에 유념해서 두 코드를 비교해보면 금방 이해되실 것입니다. 두 예제를 모두 이해하면 시스템 모델과 측정 모델이 모두 비선형인 시스템에 EKF를 설계하는 것도 그리 어렵지 않을 겁니다. 그럼 시작하겠습니다.

12.3 예제 1 : 레이다 추적

이 절에서는 EKF의 첫 번째 예제로 물체까지의 직선 거리를 측정해서 위치와 고도를 추정하는 문제를 소개합니다. 앞서 말한 것처럼 이 예제의 목적은 측정 모델이 비선형인 시스템에 대해 EKF를 설계하는 것입니다. 문제를 단순하게 하기 위해 물체는 2차원 평면 상에서 일정한 고도와 속도를 유지하면서 움직인다고 가정하겠습니다. 물론 물체의 이동 속도와 고도는 모르는 값입니다. 다음 그림에서 x_1은 레이다에서 물체까지의 수평 거리를, x_3은 물체의 고도를 의미합니다.



[그림 12-2] 레이다와 물체

■ 시스템 모델

먼저 시스템의 상태 변수를 다음과 같이 정의합니다.

$$x = \begin{Bmatrix} \text{수평 거리} \\ \text{이동 속도} \\ \text{고도} \end{Bmatrix} \equiv \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}$$

물체의 이동 속도와 고도는 일정하므로 시스템 모델은 다음과 같이 표현됩니다.

$$\begin{aligned} \begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{Bmatrix} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} + \begin{Bmatrix} 0 \\ w_1 \\ w_2 \end{Bmatrix} \\ &\equiv Ax + w \end{aligned} \tag{12.7}$$

이 시스템 모델이 의미하는 바는 간단합니다. 먼저 첫 번째 식은 수평 거리와 이동 속도의 관계를 반영한 것입니다. 잘 알다시피 수평 거리의 변화율(미분)이 이동 속도입니다.

$$\dot{x}_1 = x_2$$

두 번째와 세 번째 식은 이동 속도와 고도가 일정하다는 가정을 수학적으로 표현한 식입니다. 이상적인 시스템이라면 변화율(미분값)이 0이 되어야 하지만, 실제로는 여러 오차 요인이 있어 약간씩 값이 변합니다. 이러한 영향을 수학적으로 모델링한 것이 시스템 잡음 w_1과 w_2입니다.

$$\begin{aligned} \dot{x}_2 &= w_1 \\ \dot{x}_3 &= w_2 \end{aligned}$$

이제 측정 모델을 살펴보겠습니다. 레이다가 측정하는 값은 이동 물체까지의 직선 거리입니다. [그림 12-2]에서 직선 거리는 다음과 같이 정의됩니다.

$$
\begin{aligned}
r &= \sqrt{x_1^2 + x_3^2} + v \\
&\equiv h(x) + v
\end{aligned}
\tag{12.8}
$$

식 (12.8)에서 r은 물체까지의 직선 거리를 뜻하고, v는 레이다의 측정 잡음을 의미합니다.

이렇게 해서 식 (12.7)과 (12.8)의 시스템 모델을 모두 구했습니다. 두 식을 보면 식 (12.7)은 선형인데, 식 (12.8)의 측정 모델은 비선형입니다. 따라서 이 예제에 선형 칼만 필터를 바로 적용하지는 못합니다.

■ EKF 함수

식 (12.7)의 시스템 모델은 선형이므로, EKF 알고리즘에서 상태 변수를 예측할 때 선형 칼만 필터의 수식을 사용합니다.

$$
\hat{x}_{k+1}^{-} = f(\hat{x}_k) \quad \Rightarrow \quad \hat{x}_{k+1}^{-} = A\hat{x}_k
$$

수식으로 써 놓으니 뭔가 복잡해 보이지만 사실 위의 식은 아주 당연한 이야기를 어렵게 설명한 것에 불과합니다. 식 (12.1)과 (12.7)에서 함수 $f(x)$는 다음과 같이 정의되므로, 위의 관계는 당연합니다.

$$
f(x) \equiv Ax \tag{12.9}
$$

마찬가지로 [그림 12-1]의 알고리즘에서 함수 $f(x)$의 야코비안 대신 A를 바로 사용합니다. 식 (12.9)를 보면 함수 $f(x)$의 야코비안이 바로 A가 되니 당연한 이야기입니다.

[그림 12-1]에 보인 EKF는 식 (12.1)과 (12.2)과 같은 이산 시스템에 대한 알고리즘입니다. 그런데 식 (12.7)의 시스템 모델은 이산 시스템이 아닙니다. 따라서 구현할 때는 시스템 모델을

이산 시스템으로 바꿔줘야 합니다. 식 (12.7)을 이산화하면 시스템 행렬 A는 다음과 같이 주어집니다.

$$A = I + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times dt$$

위의 이산 시스템 모델은 식 (12.7)을 오일러 적분법으로 dt시간 동안 적분한 결과입니다. 필요하면 더 정밀한 수치 적분법을 통해 실시간으로 계산하는 방법도 있습니다.

반면 측정 모델은 비선형 모델이므로, 행렬 H는 다음과 같이 식 (12.8)의 야코비안을 계산해야 합니다.

$$\begin{aligned} H &= \begin{bmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} & \frac{\partial h}{\partial x_3} \end{bmatrix} \\ &= \begin{bmatrix} \frac{x_1}{\sqrt{x_1^2 + x_3^2}} & 0 & \frac{x_3}{\sqrt{x_1^2 + x_3^2}} \end{bmatrix} \end{aligned} \tag{12.10}$$

다음에 EKF 함수 RadarEKF의 구현 결과를 나타냈습니다.

RadarEKF(z, dt) : EKF 함수

- **매개변수** : z(측정 거리), dt(샘플링 간격)
- **반환값** : 수평 위치, 이동 속도, 고도

이 함수는 측정 거리와 샘플링 간격을 인자로 받고, 수평 위치와 이동 속도, 고도를 반환합니다. 코드의 구조는 앞서 구현한 칼만 필터 함수와 거의 동일합니다. 초기화 코드가 첫 부분에 나오고,

그 다음에 [그림 12-1]의 알고리즘 순서에 따라 추정값을 계산하는 EKF 알고리즘을 구현했습니다.

코드를 보면 [그림 12-1]의 알고리즘을 시작하기 전에 행렬 H를 먼저 계산합니다. Hjacob함수는 식 (12.10)에 따라 측정 모델의 야코비안 H를 계산하는 서브 함수로, 상태 변수를 인자로 받아 H를 반환합니다(26행).

그런데 EKF는 선형 모델을 구할 때 현재의 추정값을 기준으로 선형 모델을 구합니다. 따라서 이 함수도 현재의 추정값을 인자로 받아야 합니다. RadarEKF 함수를 최초로 실행할 때 이 조건이 문제가 됩니다. 제일 처음에는 계산한 추정값이 없어 선형 모델을 계산하지 못하기 때문입니다. 선형 모델이 없으면 알고리즘을 수행해 추정값을 얻을 수 없습니다. 결국 EKF 알고리즘을 시작하지 못하는 문제가 생깁니다. 이러한 모순을 해결하려면, 최초로 실행할 때에는 상태 변수의 초기 추정값($\hat{x}_0$)으로 선형 모델을 구할 수 밖에 없습니다(19행).

그런데 임의로 예측한 초기 추정값이 실제 상태와 많이 다르면 문제가 됩니다. 초깃값 오차가 크면 이 값으로 구한 선형 모델과 실제 시스템의 차이가 많이 나게 됩니다. 그런데 시스템 모델의 오차는 EKF의 추정값 오차를 키우고, 다시 이 추정값으로 계산한 시스템 모델의 오차는 더 커집니다. 이런 식으로 반복되다 보면 결국 알고리즘이 발산하게 됩니다. 이처럼 EKF는 초깃값에 민감하기 때문에, 시스템을 충분히 연구해서 초깃값을 잘 선정하는 것이 매우 중요합니다. 이 문제에 대한 일반적인 해법은 없습니다. 대상 시스템을 잘 분석해서 최대한 정확한 초깃값을 찾는 수 밖에 없습니다.

한편 서브 함수로 정의된 hx 함수는 식 (12.8)의 측정 모델식을 구현한 함수입니다. 이 함수는 상태 변수를 인자로 받아 물체까지의 직선 거리를 반환합니다(43행).

M-File RadarEKF.m

```
function [pos, vel, alt] = RadarEKF(z, dt)
%
%
persistent A Q R
persistent x P
persistent firstRun


if isempty(firstRun)
  A = eye(3) + dt*[ 0 1 0;
                    0 0 0;
                    0 0 0 ];
  Q = [ 0  0      0;
        0  0.001  0;
        0  0      0.001 ];

  R = 10;

  x = [0 90 1100]';                    % 임의로 예측한 초기 추정값
  P = 10*eye(3);

  firstRun = 1;
end


H = Hjacob(x);                         % 시스템 행렬의 야코비안 H 계산

xp = A*x;
Pp = A*P*A' + Q;

K = Pp*H'*inv(H*Pp*H' + R);

x = xp + K*(z - hx(xp));
P = Pp - K*H*Pp;


pos = x(1);
vel = x(2);
```

```
alt = x(3);


%-----------------------------
function zp = hx(xhat)                    % 직선 거리 반환
%
%
x1 = xhat(1);
x3 = xhat(3);

zp = sqrt(x1^2 + x3^2);


%-----------------------------
function H = Hjacob(xp)
%
%
H = zeros(1, 3);

x1 = xp(1);
x3 = xp(3);

H(1) = x1 / sqrt(x1^2 + x3^2);
H(2) = 0;
H(3) = x3 / sqrt(x1^2 + x3^2);
```

■ 테스트 프로그램

그럼 앞에서 구현한 EKF 함수가 제대로 작동하는지 확인해보겠습니다. TestRadarEKF.m은 테스트 프로그램의 코드입니다. 지금까지 익히 봐온 테스트 프로그램과 같은 구조로 되어 있습니다.

테스트 프로그램은 레이다로 직선 거리를 측정한 다음, 이 값을 앞에서 구현한 EKF 함수에 넘겨 위치와 속도, 고도의 추정값을 얻어 옵니다. 그리고 이 값을 저장했다가 그래프로 그립니다.

M-File TestRadarEKF.m

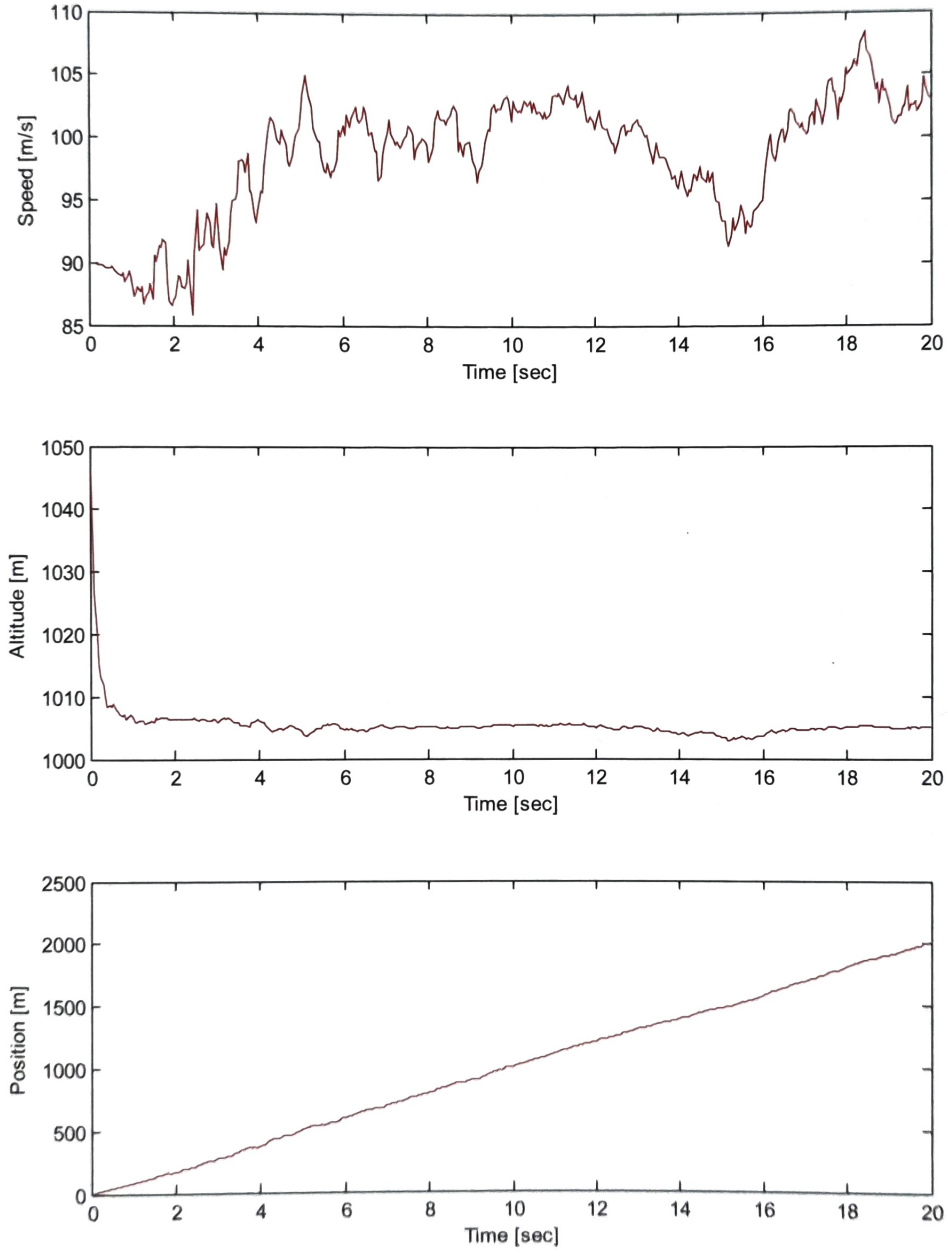
```
clear all

dt = 0.05;
t  = 0:dt:20;

Nsamples = length(t);

Xsaved = zeros(Nsamples, 3);
Zsaved = zeros(Nsamples, 1);

for k=1:Nsamples
  r = GetRadar(dt);

  [pos, vel, alt] = RadarEKF(r, dt);

  Xsaved(k, :) = [pos vel alt];
  Zsaved(k)    = r;
end


PosSaved = Xsaved(:, 1);
VelSaved = Xsaved(:, 2);
AltSaved = Xsaved(:, 3);

t = 0:dt:Nsamples*dt-dt;

figure
plot(t, PosSaved)

figure
plot(t, VelSaved)

figure
plot(t, AltSaved)
```

레이다의 측정값은 GetRadar 함수로 얻어옵니다. 이 함수의 구현 코드에는 일정한 고도에서 정속으로 이동하는 물체의 동역학 방정식이 구현되어 있습니다. 여기서는 고도 1km에서 100m/s의 속도로 움직인다고 가정했습니다. 이 두 값은 평균값으로, 실제로는 잡음 때문에 약간씩 변한다고 가정했습니다. 그리고 레이다가 측정하는 직선 거리도 측정 오차가 있다고 가정했습니다. 오차 범위는 측정 거리의 5% 내외로 주었습니다. 이처럼 가상으로 측정 잡음을 만드는 코드는 실제 시스템에서는 필요 없습니다. 잘 이해가 안되면, GetRadar 함수를 그냥 레이다의 직선 거리를 측정하는 함수라고 정리하고 넘어가도 문제 없습니다.

M-File GetRadar.m

```
function r = GetRadar(dt)
%
%
persistent posp


if isempty(posp)
  posp = 0;
end


vel = 100  +  5*randn;
alt = 1000 + 10*randn;

pos = posp + vel*dt;

v = 0 + pos*0.05*randn;

r = sqrt(pos^2 + alt^2) + v;

posp = pos;
```

실행 결과를 살펴보겠습니다. 먼저 속도 추정값부터 보겠습니다. 실제 속도는 평균이 100m/s인데, 추정값의 궤적을 보면 잘 추정한 것을 볼 수 있습니다. 처음에는 초깃값 오차 때문에 추정 오차가 크지만, 시간이 지나면 속도 추정값이 명목 속도 근방으로 수렴합니다. 여기서는 초깃값의 영향을 보기 위해 일부러 오차를 주었지만, EKF에서는 초깃값을 최대한 정확하게 선정해야 합니다. 이 예제에서는 10% 정도 오차에도 발산하지 않고 잘 작동했습니다.

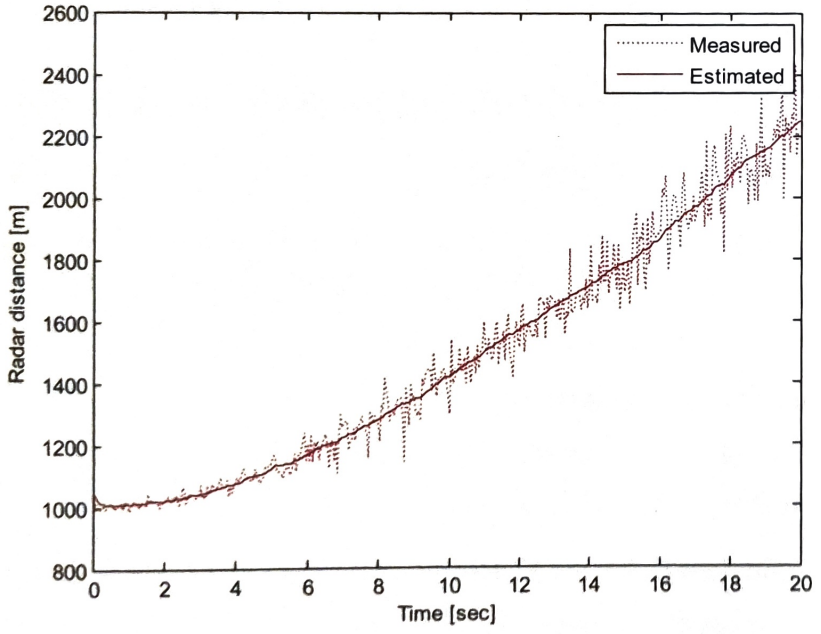


[그림 12-3] EKF 실행 결과

그리고 고도를 보면 실제 고도의 평균은 1000m인데 잘 추정해냈습니다. 속도와 마찬가지로 초기 예측값에 10% 정도의 오차를 주었는데도(1100m) 발산하지 않고 수렴했습니다. 수평 거리의 경우 평균 100m/s 속도로 20초 동안 이동하면 2000m를 이동하는데, 추정 거리도 거의 비슷하게 나왔습니다. EKF가 잘 동작하고 있음을 알 수 있습니다.

[그림 12-4]는 레이다가 측정한 직선 거리와 추정 거리를 비교한 그림입니다. 점선은 레이다의 측정값이고, 실선은 EKF의 추정값입니다. 추정 거리는 EKF로 직접 추정한 값이 아닙니다. 추정 고도와 추정 수평 거리로 계산한 결과입니다.

그림을 보면 측정값의 잡음이 거의 제거되었고, 물체의 이동에 따른 측정 거리 변화도 잘 반영되었음을 확인할 수 있습니다. 지금까지 여러 예제에서 확인했던 칼만 필터의 성능이 제대로 발휘된 결과입니다.



[그림 12-4] 측정 거리와 추정 거리 비교

12.4 예제 2 : 기울기 자세 측정하기

이번에는 시스템 모델이 비선형인 문제를 다뤄보겠습니다. 상태 변수를 오일러 각도에서 쿼터니언으로 변환해야만 했던 11장의 자이로와 가속도계를 이용한 기울기 자세 측정 예제를 다시 한 번 살펴보겠습니다. 어떤 시스템인지는 이미 알고 있을 테니 바로 본론으로 들어가겠습니다.

■ 시스템 모델

이 절에서 설계할 칼만 필터의 역할은 자이로 자세각의 누적 오차를 가속도계로 보정하는 것입니다. 시스템 모델의 상태 변수는 다음과 같이 오일러 각도로 지정합니다. 11장에서는 쿼터니언이 상태 변수였습니다.

$$x = \begin{Bmatrix} \phi \\ \theta \\ \psi \end{Bmatrix}$$

수평면의 기울기 자세를 표현하는 데는 롤각(ϕ)과 피치각(θ)만 있으면 되지만, 수식을 설명하는 데에는 요각(ψ)도 필요해서 요각도 상태 변수에 포함했습니다.

그럼 시스템 모델을 먼저 보겠습니다. 앞 장의 식 (11.1)에서 이미 소개한 바 있는 식입니다.

$$\begin{aligned}
\begin{Bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{Bmatrix} &= \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{bmatrix} \begin{Bmatrix} p \\ q \\ r \end{Bmatrix} + w \\
&= \begin{bmatrix} p + q\sin\phi\tan\theta + r\cos\phi\tan\theta \\ q\cos\phi - r\sin\phi \\ q\sin\phi\sec\theta + r\cos\phi\sec\theta \end{bmatrix} + w \\
&\equiv f(x) + w
\end{aligned} \tag{12.11}$$

엄밀히 말하면 위의 식에서 함수 f는 각속도 $(p,\ q,\ r)$의 함수인데, 편의상 표시하지 않았습니

다. 식 (12.11)은 삼각함수가 있어서 이미 비선형인데, 삼각함수끼리도 서로 곱해져 있는 등 상당히 심한 비선형 시스템입니다. 따라서 이 시스템에는 선형 칼만 필터를 적용할 수 없습니다.

이제 측정 모델식을 유도해보겠습니다. 예제의 시스템에서 보정용 측정값은 가속도계로부터 얻습니다. 그런데 가속도로는 요각을 계산해낼 수 없습니다. 따라서 측정 모델은 다음과 같이 쓸 수 있습니다.

$$
\begin{aligned}
z &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{Bmatrix} \phi \\ \theta \\ \psi \end{Bmatrix} + v \\
&= Hx + v
\end{aligned}
\tag{12.12}
$$

이 식은 선형입니다. 따라서 [그림 12.1]의 알고리즘에서 야코비안 대신 행렬 H를 직접 사용하면 됩니다. 식 (12.11)과 (12.12)가 최종 시스템 모델식입니다. 앞 절의 예제와는 반대로 시스템 모델이 비선형인 시스템입니다.

■ EKF 함수

EKF를 구현하려면 먼저 식 (12.11)의 야코비안을 알아야 합니다. 식 (12.11)의 야코비안은 다음과 같이 정의됩니다.

$$
A = \begin{bmatrix} \dfrac{\partial f_1}{\partial \phi} & \dfrac{\partial f_1}{\partial \theta} & \dfrac{\partial f_1}{\partial \psi} \\ \dfrac{\partial f_2}{\partial \phi} & \dfrac{\partial f_2}{\partial \theta} & \dfrac{\partial f_2}{\partial \psi} \\ \dfrac{\partial f_3}{\partial \phi} & \dfrac{\partial f_3}{\partial \theta} & \dfrac{\partial f_3}{\partial \psi} \end{bmatrix}
\tag{12.13}
$$

이 정의에 식 (12.11)을 대입해서 미분하면 행렬 A의 수식이 나옵니다. 여기서는 미분한 수식을 나열하지는 않겠습니다. 자세한 수식은 구현 코드를 참고하기 바랍니다. 그런데 식 (12.13)

은 이산 시스템의 야코비안이 아닙니다. 따라서 구현할 때는 이 행렬을 이산 시스템으로 변환해야 합니다. 이 또한 구현 함수에서 자세히 설명하겠습니다.

이 예제에서는 미분식을 구하는 것이 그리 어렵지 않지만, 복잡한 시스템인 경우에는 해석적인 미분식을 유도하는 게 쉽지 않습니다. 이때는 수치해석 기법을 동원해서 야코비안을 계산해내는 수밖에 없는데, 그 과정에서 오차가 발생할 여지가 많습니다.

다음은 EKF를 구현한 EulerEKF 함수입니다.

EulerEKF(z, rates, dt) : EKF 함수

- **매개변수** : z(가속도계의 자세각), rates(자이로 각속도), dt(샘플링 간격)
- **반환값** : 추정 오일러각

이 함수의 코드는 가속도계의 자세각과 자이로 각속도를 인자로 받아, 추정 자세각을 반환합니다. 프로그램 구조는 앞서 구현한 코드와 똑같습니다. 알고리즘 구현도 대부분 이미 소개했던 코드입니다.

코드에서 Ajacob 함수는 식 (12.13)의 야코비안을 계산합니다(27행). 이 함수는 상태 변수와 자이로 각속도를 인자로 받아 행렬 A를 반환합니다. 내부 구현을 보면 식 (12.11)의 시스템 행렬을 미분한 수식을 확인할 수 있습니다. 마지막 줄에 있는 코드는 식 (12.13)을 이산 시스템으로 바꾸기 위한 것입니다. 앞 절의 예제에서도 같은 코드를 소개한 적이 있습니다. 그리고 서브함수 fx는 식 (12.11)을 이용해 상태 변수의 예측값을 계산하는 함수입니다(29행).

여기서 주의해야 할 점은 함수 fx가 식 (12.11)에서 우변의 $f(x)$를 계산하는 함수가 아니라는 점입니다. 이 함수가 반환하는 값은 함수 $f(x)$의 적분값입니다. 이 함수의 구현을 보면 59행에 다음과 같은 코드가 있습니다.

```
xp = xhat + xdot*dt;
```

여기서 xdot는 $f(x)$의 계산 결과를 담고 있는 변수이고, 변수 xhat은 현재의 추정값을 의미합니다. 즉 위의 코드는 함수 $f(x)$를 오일러 적분하는 코드입니다. 이렇게 적분해서 예측값을 계산하는 이유는 식 (12.11)을 이산 시스템으로 전환하는 게 복잡하기 때문입니다.

M-File EulerEKF.m

```
function [phi, theta, psi] = EulerEKF(z, rates, dt)
%
%
persistent H Q R
persistent x P
persistent firstRun


if isempty(firstRun)
  H = [ 1 0 0;
        0 1 0 ];

  Q = [ 0.0001 0      0;
        0      0.0001 0;
        0      0      0.1 ];

  R = [  10  0;
         0  10 ];

  x = [0 0 0]';
  P = 10*eye(3);

  firstRun = 1;
end


A = Ajacob(x, rates, dt);                    % 야코비안 계산
```

```
28
29  xp = fx(x, rates, dt);                    % 상태 변수 예측값 계산 함수 호출
30  Pp = A*P*A' + Q;
31
32  K = Pp*H'*inv(H*Pp*H' + R);
33
34  x = xp + K*(z - H*xp);
35  P = Pp - K*H*Pp;
36
37
38  phi   = x(1);
39  theta = x(2);
40  psi   = x(3);
41
42
43  %-------------------------------
44  function xp = fx(xhat, rates, dt)
45  %
46  %
47  phi   = xhat(1);
48  theta = xhat(2);
49
50  p = rates(1);
51  q = rates(2);
52  r = rates(3);
53
54  xdot = zeros(3, 1);
55  xdot(1) = p + q*sin(phi)*tan(theta) + r*cos(phi)*tan(theta);
56  xdot(2) =     q*cos(phi)            - r*sin(phi);
57  xdot(3) =     q*sin(phi)*sec(theta) + r*cos(phi)*sec(theta);
58
59  xp = xhat + xdot*dt;                      % 함수 f(x)를 오일러 적분
60
61
62  %-------------------------------
63  function A = Ajacob(xhat, rates, dt)
64  %
65  %
66  A = zeros(3, 3);
67
```

```
68
69  phi   = xhat(1);
70  theta = xhat(2);
71
72  p = rates(1);
73  q = rates(2);
74  r = rates(3);
75
76  A(1,1) = q*cos(phi)*tan(theta)   - r*sin(phi)*tan(theta);
77  A(1,2) = q*sin(phi)*sec(theta)^2 + r*cos(phi)*sec(theta)^2;
78  A(1,3) = 0;
79
80  A(2,1) = -q*sin(phi) - r*cos(phi);
81  A(2,2) = 0;
82  A(2,3) = 0;
83
84  A(3,1) = q*cos(phi)*sec(theta)              - r*sin(phi)*sec(theta);
85  A(3,2) = q*sin(phi)*sec(theta)*tan(theta) + r*cos(phi)*sec(theta)*tan(theta);
86  A(3,3) = 0;
87
88  A = eye(3) + A*dt;                          % 이산 시스템으로 변환
```

■ 테스트 프로그램

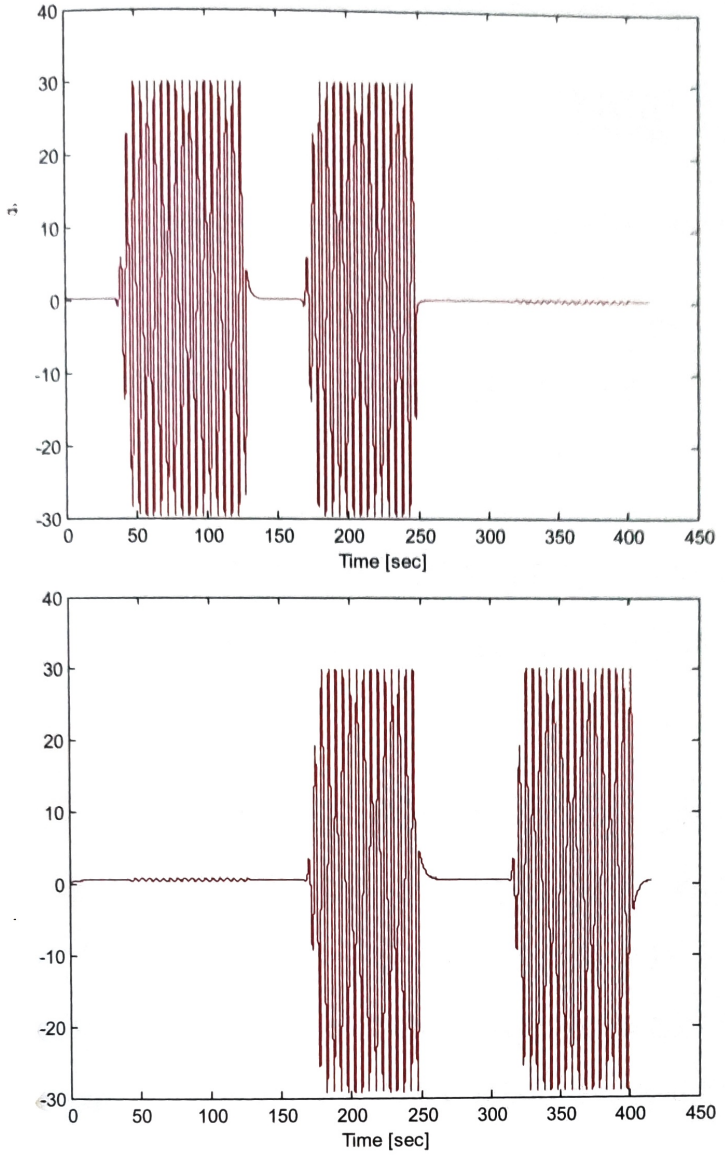
개발한 함수를 검증해 보겠습니다. 자이로와 가속도 측정값으로는 11장의 예제와 같은 데이터를 사용해서 결과를 비교해 보겠습니다. 테스트 프로그램의 코드는 다음과 같습니다. 프로그램에서 자이로와 가속도계의 측정값을 읽어오는 GetGyro 함수와 GetAccel 함수는 11장과 동일한 함수입니다.

M-File TestEulerEKF.m

```
1  clear all
2
3  Nsamples = 41500;
4  EulerSaved = zeros(Nsamples, 3);
5
6  dt = 0.01;
```

```
7
8   for k=1:Nsamples
9     [p, q, r] = GetGyro();
10    [ax, ay, az] = GetAccel();
11    [phi_a, theta_a] = EulerAccel(ax, ay, az);
12
13    [phi, theta, psi] = EulerEKF([phi_a theta_a]', [p q r], dt);
14
15    EulerSaved(k, :) = [ phi theta psi ];
16  end
17
18
19  PhiSaved   = EulerSaved(:, 1) * 180/pi;
20  ThetaSaved = EulerSaved(:, 2) * 180/pi;
21  PsiSaved   = EulerSaved(:, 3) * 180/pi;
22
23  t = 0:dt:Nsamples*dt-dt;
24
25  figure
26  plot(t, PhiSaved)
27
28  figure
30  plot(t, ThetaSaved)
31
32  figure
33  plot(t, PsiSaved)
```

테스트 프로그램은 달라진 게 별로 없습니다. 실행 결과를 바로 살펴보겠습니다. [그림 12-5]는 롤각과 피치각의 추정 결과를 그린 그래프입니다. 11장과 비슷한 결과로 보아, 기울기 자세가 성공적으로 추정된 것을 알 수 있습니다. 비록 결과는 비슷하지만 범용성 측면에서는 EKF가 더 낫습니다. 11장에서는 상태 변수를 쿼터니언으로 바꿨더니 운 좋게 선형 모델로 전환되어 칼만 필터를 설계할 수 있었습니다. 대개의 경우에는 이렇게 선형 모델로 전환하는 게 쉽지 않습니다. 반면 EKF는 어지간한 비선형 시스템에서도 잘 작동합니다. 물론 발산할 염려는 항상 있지만, 대개의 경우에는 쓸만한 성능을 냅니다.



[그림 12-5] EKF로 추정한 자세각의 궤적

12장 요약

지금까지 EKF에 대해 설명했습니다. 칼만 필터의 전체 절차는 변함 없이, 내부 계산식만 약간 바뀌었습니다. 우선 선형 모델식을 비선형 모델식으로 대체하고, 시스템 행렬 A와 H 대신 비선형 모델의 야코비안을 사용했습니다. 이때 야코비안 값은 직전 추정값에 대해 계산했습니다. EKF는 선형 모델을 미리 선정할 수 없는 비선형 시스템에도 칼만 필터를 적용하는 길을 연 탁월한 알고리즘입니다. 더구나 기존 칼만 필터 알고리즘을 자연스럽게 확장해서 이해하기 쉽고, 구현도 간단하다는 장점을 가지고 있습니다.