

[New chat](#)[My stuff](#)[Gems](#)[Chats](#)[write in matlab a kalman filter with m...](#)[Kalman Filter for Bistatic Measureme...](#)[in matlab the kalman filter predict eq...](#)[Cache-Friendly C++ Sets and Maps](#)

write in matlab a kalman filter with measurement of bistatic range, angle of transmission, and range rate; and have state in a geodetic enu velocity coordinate system, and use a constant velocity model.



Implementing a Kalman Filter for a bistatic radar system requires an **Extended Kalman Filter (EKF)** because the relationship between your geodetic states (ENU) and the bistatic measurements is inherently non-linear.

1. State Space and Model

We are using a **Constant Velocity (CV)** model in the East-North-Up (ENU) frame. The state vector is defined as:

$$\mathbf{x} = [p_e, p_n, p_u, v_e, v_n, v_u]^T$$

- **Bistatic Range:** The total path length from Transmitter → Target → Receiver.
- **Angle of Transmission (ϕ):** The azimuth angle from the transmitter to the target.
- **Range Rate:** The time derivative of the bistatic range (doppler shift).

2. MATLAB Implementation

Matlab



```
% Extended Kalman Filter for Bistatic Tracking
clear; clc;

%% 1. Setup Parameters
dt = 0.1; % Time step (s)
T_pos = [0, 0, 0]; % Transmitter position (ENU)
R_pos = [1000, 500, 0]; % Receiver position (ENU)

% Process Noise Covariance (Small for Constant Velocity)
Q = eye(6) * 0.1;

% Measurement Noise Covariance
R = diag([10^2, deg2rad(0.5)^2, 1^2]); % [Range, Angle, RangeRate]

%% 2. Initialize State
x = [500; 500; 100; 10; 5; 0]; % Initial guess [pe, pn, pu, ve, vn, vu]
P = eye(6) * 100; % Initial uncertainty

% Transition Matrix (Constant Velocity)
F = [1 0 0 dt 0 0;
      0 1 0 0 dt 0;
      0 0 1 0 0 dt;
      0 0 0 1 0 0;
      0 0 0 0 1 0;
      0 0 0 0 0 1];

%% 3. Simulation Loop (Example Step)
% Simulated Measurement [BistaticRange; TransmitAngle; RangeRate]
z_meas = [1200; 0.78; 8.5];

% --- PREDICT ---
x_pred = F * x;
P_pred = F * P * F' + Q;

% --- UPDATE (EKF) ---
% Extract predicted positions/velocities
pe = x_pred(1); pn = x_pred(2); pu = x_pred(3);
ve = x_pred(4); vn = x_pred(5); vu = x_pred(6);

% Distances to Transmitter (T) and Receiver (R)
Rt = sqrt((pe-T_pos(1))^2 + (pn-T_pos(2))^2 + (vu-T_pos(3))^2) -
```

```

Rt = sqrt((pe-T_pos(1))^2 + (pn-T_pos(2))^2 + (pu-T_pos(3))^2);
Rr = sqrt((pe-R_pos(1))^2 + (pn-R_pos(2))^2 + (pu-R_pos(3))^2);

% h(x): Measurement function
h_range = Rt + Rr;
h_angle = atan2(pn - T_pos(2), pe - T_pos(1));
h_rate = ((pe-T_pos(1))*ve + (pn-T_pos(2))*vn + (pu-T_pos(3))*vu)/Rt + ...
           ((pe-R_pos(1))*ve + (pn-R_pos(2))*vn + (pu-R_pos(3))*vu)/Rr;

z_hat = [h_range; h_angle; h_rate];

% H: Jacobian Matrix (dh/dx)
% (Simplified placeholders for brevity - you must compute partial derivatives)
H = zeros(3, 6);
% Row 1: d(Range)/d(pos)
H(1,1:3) = [(pe-T_pos(1))/Rt + (pe-R_pos(1))/Rr, ...
              (pn-T_pos(2))/Rt + (pn-R_pos(2))/Rr, ...
              (pu-T_pos(3))/Rt + (pu-R_pos(3))/Rr];
% Row 2: d(Angle)/d(pos)
dist_sq = (pe-T_pos(1))^2 + (pn-T_pos(2))^2;
H(2,1:2) = [-(pn-T_pos(2))/dist_sq, (pe-T_pos(1))/dist_sq];
% Row 3: d(RangeRate) - involves complex chain rule on velocities/positions
% Typically computed via numerical differentiation or symbolic toolbox

% Update Steps
S = H * P_pred * H' + R;
K = P_pred * H' / S;
x = x_pred + K * (z_meas - z_hat);
P = (eye(6) - K * H) * P_pred;

fprintf('Updated Position: E:%.2f, N:%.2f, U:%.2f\n', x(1), x(2), x(3));

```

3. Key Considerations

- **The Jacobian (H):** The third row of the Jacobian (Range Rate) is mathematically dense as it involves the derivative of the dot product of the velocity and unit position vectors. If you have the **Symbolic Math Toolbox**, use `jacobian()` to generate this row precisely.
- **Coordinate Singularity:** `atan2` is used for the angle of transmission to handle all quadrants, but be careful with "wrap-around" (the difference between 359° and 1° should be 2° , not 358°).
- **Bistatic Geometry:** Remember that bistatic range defines an **ellipse** where the transmitter and receiver are the foci. The angle of transmission is what "pins" the target's location on that ellipse.

Would you like me to provide the full symbolic derivation for the Range Rate Jacobian row to paste into your script?

↶ ↷ ↴ :

Use a unscented kalman filter instead



Switching to an **Unscented Kalman Filter (UKF)** is a smart move for bistatic systems. Because the geometry involves ellipses and trigonometric mappings, the linear approximations used in an EKF (the Jacobians) often lead to divergence if the initial uncertainty is high.

The UKF handles this by using **Sigma Points** to propagate the mean and covariance through the nonlinear functions directly.

MATLAB Implementation: UKF for Bistatic Tracking

This script uses a Constant Velocity model in ENU coordinates. It defines the state transition and measurement functions separately to handle the Sigma Point transformations.

Matlab



```
% Unscented Kalman Filter (UKF) for Bistatic Radar
clear; clc;
```

```

%% 1. Parameters & Setup
dt = 0.5; % Sampling time
T_pos = [0; 0; 0]; % Transmitter (E, N, U)
R_pos = [2000; 1000; 0]; % Receiver (E, N, U)

% UKF Scaling Parameters (Alpha, Beta, Kappa)
n = 6; % State dimension
alpha = 1e-3; % Default tune
ki = 0; % Secondary scaling
beta = 2; % Optimal for Gaussian distributions
lambda = alpha^2 * (n + ki) - n;

% Weights for Mean and Covariance
wm = [lambda/(n+lambda), repmat(1/(2*(n+lambda)), 1, 2*n)];
wc = wm;
wc(1) = wc(1) + (1 - alpha^2 + beta);

%% 2. Initialization
% State: [pe, pn, pu, ve, vn, vu]
x = [1000; 800; 200; 15; -5; 0];
P = eye(6) * 50; % Initial State Covariance
Q = eye(6) * 0.05; % Process Noise
R = diag([15^2, deg2rad(0.3)^2, 0.5^2]); % Measurement Noise [Range, Angle, Rate]

% Simulated Measurement (Normally this comes from your sensors)
z_meas = [2500; 0.65; -10.2];

%% 3. UKF Prediction Step
% Generate Sigma Points
A = chol(P + Q)'; % Cholesky decomposition for spread
sigmas = [x, x + sqrt(n+lambda)*A, x - sqrt(n+lambda)*A];

% Propagate Sigma Points through State Transition (Constant Velocity)
F = [1 0 0 dt 0 0; 0 1 0 0 dt 0; 0 0 1 0 0 dt; 0 0 0 1 0 0; 0 0 0 0 1 0; 0 0 0 0
sigmas_pred = F * sigmas;

% Recombine for Predicted Mean and Covariance
x_pred = sum(wm .* sigmas_pred, 2);
P_pred = Q;
for i = 1:2*n+1
    diff = sigmas_pred(:,i) - x_pred;
    P_pred = P_pred + wc(i) * (diff * diff');
end

%% 4. UKF Update Step
% Transform Sigma Points into Measurement Space h(x)
Z_sig = zeros(3, 2*n+1);
for i = 1:2*n+1
    sp = sigmas_pred(:,i);
    % Geometry distances
    Rt = norm(sp(1:3) - T_pos);
    Rr = norm(sp(1:3) - R_pos);

    % h1: Bistatic Range
    h_rho = Rt + Rr;
    % h2: Angle of Transmission (Azimuth)
    h_theta = atan2(sp(2)-T_pos(2), sp(1)-T_pos(1));
    % h3: Range Rate (Doppler)
    v_vec = sp(4:6);
    u_t = (sp(1:3) - T_pos) / Rt; % Unit vector t -> target
    u_r = (sp(1:3) - R_pos) / Rr; % Unit vector R -> Target
    h_rdot = dot(u_t, v_vec) + dot(u_r, v_vec);

    Z_sig(:,i) = [h_rho; h_theta; h_rdot];
end

% Predicted Measurement Mean
z_hat = sum(wm .* Z_sig, 2);

% Measurement Covariance (S) and Cross-Covariance (Pxz)
S = R;
Pxz = zeros(n, 3);
for i = 1:2*n+1
    z_diff = Z_sig(:,i) - z_hat;
    x_diff = sigmas_pred(:,i) - x_pred;
    S = S + wc(i) * (z_diff * z_diff');
    Pxz = Pxz + wc(i) * (x_diff * z_diff');
end

```

```
% Kalman Gain and Final Update
K = Pxz / S;
x = x_pred + K * (z_meas - z_hat);
P = P_pred - K * S * K';

fprintf('Estimated Velocity (E,N,U): %.2f, %.2f, %.2f\n', x(4), x(5), x(6));
```

Why this is better for Bistatic Range

The "Bistatic Range" is physically an ellipse. In an EKF, you approximate that ellipse with a straight line (tangent) at the current estimate. If your estimate is off by even a few degrees, that line points in the wrong direction.

The UKF, however, picks points **around** the ellipse. When these points are passed through the measurement function, the filter "sees" the curvature of the ellipse, allowing it to capture the true mean and covariance of the target's position much more accurately.

Key Implementation Note: In the measurement update for `h_theta`, if your target crosses the $\pm\pi$ boundary, you must add code to "unwrap" the angles in your Sigma Points. Otherwise, the mean calculation will result in an estimate that is 180° in the wrong direction.

Would you like me to add the **angle-unwrapping logic** to handle targets crossing the North/South axis?

i want the state to be in geodetic (latitude longitude) and enu velocity

- ◆ To transition from a flat-earth ENU model to a **Geodetic State UKF**, we need to account for the Earth's curvature. Your state vector becomes:

$$\mathbf{x} = [\phi, \lambda, h, v_e, v_n, v_u]^T$$

(where ϕ is latitude, λ is longitude, and h is ellipsoidal height).

The primary challenge here is that **velocity** changes the **lat/lon** based on the Earth's radii of curvature (Meridian M and Prime Vertical N).

UKF with Geodetic Position and ENU Velocity

```
Matlab
```

```
% UKF: Geodetic Position (Lat, Lon, H) and ENU Velocity
clear; clc;

%% 1. Constants & Setup
a = 6378137.0; % WGS84 Semi-major axis (m)
f = 1/298.257223563; % Flattening
e2 = 2*f - f^2; % eccentricity squared

dt = 1.0; % Time step
T_geo = [0.5934, -1.239, 10]; % Transmitter [Lat, Lon, H] (rad)
R_geo = [0.5935, -1.238, 10]; % Receiver [Lat, Lon, H] (rad)

% State: [lat(rad), lon(rad), h(m), ve(m/s), vn(m/s), vu(m/s)]
x = [0.59345; -1.2385; 500; 20; 10; 0];
P = diag([1e-8, 1e-8, 10, 5, 5, 2]);
Q = diag([1e-12, 1e-12, 0.1, 0.5, 0.5, 0.1]);
R = diag([20^2, deg2rad(0.5)^2, 1^2]);

% UKF Weights
n = 6; alpha = 1e-3; ki = 0; beta = 2;
lambda = alpha^2 * (n + ki) - n;
wm = [lambda/(n+lambda), repmat(1/(2*(n+lambda)), 1, 2*n)];
wc = wm; wc(1) = wc(1) + (1 - alpha^2 + beta);
```

```

%% 2. Prediction Step (Geodetic Kinematics)
A = chol(P + Q)';
sigmas = [x, x + sqrt(n+lambda)*A, x - sqrt(n+lambda)*A];

for i = 1:(2*n+1)
    lat = sigmas(1,i); h = sigmas(3,i);
    ve = sigmas(4,i); vn = sigmas(5,i); vu = sigmas(6,i);

    % Radii of Curvature
    M = a*(1-e2) / (1 - e2*sin(lat)^2)^1.5;
    N = a / sqrt(1 - e2*sin(lat)^2);

    % Derivatives: dLat = vn/(M+h), dLon = ve/((N+h)*cos(Lat))
    sigmas(1,i) = lat + (vn / (M + h)) * dt;
    sigmas(2,i) = sigmas(2,i) + (ve / ((N + h) * cos(lat))) * dt;
    sigmas(3,i) = h + vu * dt;
    % Velocities (Constant Velocity model in ENU)
    % Note: In high-precision long-range, you'd add Coriolis/Transport rates here
end

x_pred = sum(wm .* sigmas, 2);
P_pred = Q;
for i = 1:2*n+1
    d = sigmas(:,i) - x_pred;
    P_pred = P_pred + wc(i) * (d * d');
end

%% 3. Measurement Step (Bistatic in Geodetic)
Z_sig = zeros(3, 2*n+1);
for i = 1:(2*n+1)
    % Convert Sigma Point (LLA) to ECEF to calculate Bistatic Range
    pos_ecef = lla2ecef_local(sigmas(1:3,i), a, e2);
    T_ecef = lla2ecef_local(T_geo, a, e2);
    R_ecef = lla2ecef_local(R_geo, a, e2);

    Rt_vec = pos_ecef - T_ecef;
    Rr_vec = pos_ecef - R_ecef;
    Rt = norm(Rt_vec);
    Rr = norm(Rr_vec);

    % Bistatic Range
    Z_sig(1,i) = Rt + Rr;

    % Angle of Transmission (Azimuth in T's ENU frame)
    % Requires converting target pos to Transmitter-centered ENU
    enu_target = ecef2enu_local(pos_ecef, T_geo, a, e2);
    Z_sig(2,i) = atan2(enu_target(1), enu_target(2));

    % Range Rate (Doppler)
    % Convert ENU velocity to ECEF for vector dot product
    v_enu = sigmas(4:6,i);
    v_ecef = enu2ecef_vel(v_enu, sigmas(1:2,i));
    Z_sig(3,i) = dot(Rt_vec/Rt, v_ecef) + dot(Rr_vec/Rr, v_ecef);
end

% Standard UKF Update (Mean/Covariance/Gain)
z_hat = sum(wm .* Z_sig, 2);
S = R; Pxz = zeros(n, 3);
for i = 1:2*n+1
    dz = Z_sig(:,i) - z_hat;
    dx = sigmas(:,i) - x_pred;
    S = S + wc(i)*(dz*dz');
    Pxz = Pxz + wc(i)*(dx*dx');
end
K = Pxz / S;
x = x_pred + K * ([2550; 0.66; -10] - z_hat); % Example observation
P = P_pred - K * S * K';

%% Local Helper Functions
function ecef = lla2ecef_local(lla, a, e2)
    lat = lla(1); lon = lla(2); h = lla(3);
    N = a / sqrt(1 - e2*sin(lat)^2);
    ecef = [(N+h)*cos(lat)*cos(lon); (N+h)*cos(lat)*sin(lon); (N*(1-e2)+h)*sin(lat)];
end

function enu = ecef2enu_local(ecef, ref_llla, a, e2)
    ref_ecef = lla2ecef_local(ref_llla, a, e2);
    rel = ecef - ref_ecef;
    lat = ref_llla(1); lon = ref_llla(2);

```

```

R = [-sin(lon) cos(lon) 0; -sin(lat)*cos(lon) -sin(lat)*sin(lon) cos(lat); ...
      enu = R * rel;
end

function v_ecef = enu2ecef_vel(v_enu, lla)
    lat = lla(1); lon = lla(2);
    R = [-sin(lon) -sin(lat)*cos(lon) cos(lat)*cos(lon); ...
          cos(lon) -sin(lat)*sin(lon) cos(lat)*sin(lon); ...
          0           cos(lat)           sin(lat)];
    v_ecef = R * v_enu;
end

```

Critical Components for Geodetic Tracking

- Curvature Dependence:** In the prediction step, your latitude change depends on M (Meridian radius) and longitude depends on N (Prime Vertical radius). At higher latitudes, the same East velocity produces a much larger change in longitude than at the equator.
- Angle of Transmission:** This measurement is typically relative to the Transmitter's local North. Therefore, you must project the target's position into a **local ENU frame centered at the Transmitter** to calculate the correct azimuth.
- Velocity Projection:** Range rate is the projection of the velocity vector onto the line-of-sight. Since your velocity is in ENU but the range is naturally calculated in ECEF (to avoid earth-curvature errors over long distances), converting the ENU velocity to ECEF for the dot product is the most robust method.

Would you like me to show how to add **Coriolis and Centripetal accelerations** to the velocity state for high-speed (supersonic) target tracking?

↶ ↷ ⌂ ☰ :

Connect ESP32 to Windows Laptop

Cache-Friendly C++ Map Implementa...

⚙️ Settings & help

Ask Gemini 3

⊕ Tools

Fast ▾



Gemini is AI and can make mistakes.