```
clear all
close all
clc


%% Initialize Parameters
mu = 3.986004418e14; % m^3/s^2
GS_1= [-111.536, 35.097, 2.206]; % Geodetic coordinates (lat [deg], long [deg], alt↙
[km])
GS_2= [-70.692, -29.016, 2.380]; % Geodetic coordinates (lat [deg], long [deg], alt↙
[km])
arcsec2rad = (pi / 648000); % Conversion factor from arcsecs to radians
R_GS1= diag([1, 1, 0.01, 0.01].* (arcsec2rad^2));           % Covariance of GS1
R_GS2= diag([0.01, 0.01, 0.0001, 0.0001].* (arcsec2rad^2)); % Covariance of GS2
R_GPS= diag([5000^2, 5000^2, 5000^2, .5^2, .5^2, .5^2]); % Covariance of GPS
txt = '2024-11-24T05:04:30.000';
Epoch = datetime(txt, 'InputFormat', 'yyyy-MM-dd''T''HH:mm:ss.SSS');

%% Read CSV and Dicteate Measuerment Units (m, m/s)
filename = 'measurements.csv';
data = readtable(filename);

% Split data by sensor ID
sensor_ids = unique(string(data.sensor_id));
GPS_Data = data(string(data.sensor_id) == 'gps_measurement', :);
GS1_Data = data(string(data.sensor_id) == 'ground_observer_1', :);
GS2_Data = data(string(data.sensor_id) == 'ground_observer_2', :);

% Convert to matrices
GPS_Obs = table2array(GPS_Data(:, 3:8))*1000;        %Convert from km to meters
t_gps = table2array(GPS_Data(:, 1));
GS1_Obs = deg2rad(table2array(GS1_Data(:, 9:end)));  %Convert from Angles to Radians
t_gs1 = table2array(GS1_Data(:, 1));
GS2_Obs = deg2rad(table2array(GS2_Data(:, 9:end)));  %Convert from Angles to Radians
t_gs2 = table2array(GS2_Data(:, 1));

%% Filters
% Batch Filter for Ground Station 1
[r_GS1, v_GS1, rv_cov_GS1, Ref_t_GS1] = batch_angles_filter(t_gs1, GS1_Obs, R_GS1,↙
t_gps, GPS_Obs, R_GPS, Epoch);

% Batch Filter for Ground Statoin 2
[r_GS2, v_GS2, rv_cov_GS2, Ref_t_GS2] = batch_angles_filter(t_gs2, GS2_Obs, R_GS2,↙
t_gps, GPS_Obs, R_GPS, Epoch);

% Expanded Sequential Filter for GPS Measurements
[r_GPS, v_GPS, rv_cov_GPS, Ref_t_GPS] = sequential_rv_filter(t_gps, GPS_Obs, R_GPS,↙
mu, Epoch);
% [kep_est, rv_cov] = batch_gps_multiarc(t_gps, GPS_Obs, R_GPS, 40*60, Epoch);

% Keplerian Orbital Elements
[a_GPS,e_GPS,inc_GPS,RAAN_GPS,w_GPS,nu_GPS] =ijk2keplerian(r_GPS, v_GPS);
[a_GS1,e_GS1,inc_GS1,RAAN_GS1,w_GS1,nu_GS1] =ijk2keplerian(r_GS1, v_GS1);
[a_GS2,e_GS2,inc_GS2,RAAN_GS2,w_GS2,nu_GS2] =ijk2keplerian(r_GS2, v_GS2);
```

```
%% Fusing of Esimtates (Using Kalman Filter)
% Make sure they are all sorted
ref_times = [Ref_t_GS1, Ref_t_GS2, Ref_t_GPS];
All_r = [r_GS1, r_GS2, r_GPS];
All_v = [v_GS1, v_GS2, v_GPS];
All_P = cat(3, rv_cov_GS1, rv_cov_GS2, rv_cov_GPS);
[~, idx] = sort(ref_times);
All_r = All_r(:, idx);
All_v = All_v(:, idx);
All_P = All_P(:,:, idx);
ref_times = ref_times(idx);

% Extended Sequential filter
[r_fused, v_fused, rv_cov_fused, Ref_t_fused] = sequential_rv_filter(ref_times,↙
[All_r' All_v'], All_P, mu, Epoch);
[a_fused,e_fused,inc_fused,RAAN_fused,w_fused,nu_fused] =ijk2keplerian(r_fused,↙
v_fused);

% Estimated Orbit Calculatiosn
Period = 2*pi*sqrt(a_fused^3/mu);
perigee = a_fused*(1-e_fused);
apogee = a_fused*(1+e_fused);
J=cart2kepJac(r_fused, v_fused);  % Jacobian car2kep
rv_cov_fused_oe = J * rv_cov_fused * J';

%% Plots
% Plot Orbit
[~, ~,r_prop, v_prop,time_prop]= Propagate(Period, r_fused, v_fused, 1);

% Plot Orbit and Change Colors of orbit pased on LOS
GS_Pos = plot_ground_station([GS_1;GS_2],1);
plotOrbitVisibility(a_fused,e_fused,inc_fused,RAAN_fused,w_fused, nu_fused, GS_Pos);

% Plot Uncertainty Elipsoid
cats=cat(3,diag(diag(rv_cov_GS1(1:3, 1:3))), diag(diag(rv_cov_GS2(1:3, 1:3))), diag↙
(diag(rv_cov_GPS(1:3, 1:3))), diag(diag(rv_cov_fused(1:3,1:3))));
plot_measurements_with_uncertainty([r_GS1'/1000;r_GS2'/1000;r_GPS'/1000;↙
r_fused'/1000], cats)

%% Realign fused estiamtion to t=0 position
[r_start_fuse, v_start_fuse, ~,~,~]= Propagate(Period-12000, r_fused, v_fused, 0,↙
1);
[~, ~, r_prop, v_prop, time_prop]= Propagate(Period, r_start_fuse, v_start_fuse, 0,↙
1);

%% Range Calculations for GS1, GS2, General
relative_r_gs1 = r_prop - r_GS1';
relative_r_gs2 = r_prop - r_GS2';
relative_r_gs1_norm = vecnorm(relative_r_gs1, 2, 2)/1000;
relative_r_gs2_norm = vecnorm(relative_r_gs2, 2, 2)/1000;
relative_r_gs1_norm_meas = round(relative_r_gs1_norm((t_gs1(1)+1):(t_gs1(2)-t_gs1↙
(1)):(t_gs1(end)+1)));
relative_r_gs2_norm_meas = round(relative_r_gs2_norm((t_gs2(1)+1):(t_gs2(2)-t_gs2↙
```

```
(1)):(t_gs2(end)+1)));
r_norm = vecnorm(r_prop, 2, 2)/1000;
r_norm_GPS_meas = round(r_norm((t_gps(1)+1):(t_gps(2)-t_gps(1)):(t_gps(end)+1)));

%% Azimuth and Elevation Plots
% Calculate
[el1, az1] = elevazim(r_prop,GS_Pos(1,:));
[el2, az2] = elevazim(r_prop,GS_Pos(2,:));
% Plot
result1 = EL_AZ_Ploter(el1',az1');
result2 = EL_AZ_Ploter(el2',az2');

%% UKF and MC Propagation
% Propagation
[x_mc, P_mc] = monte_carlo_propagation(r_fused, v_fused, rv_cov_fused, 5*3600, 100);
[x_ukf, P_ukf] = ukf_propagation([r_fused; v_fused] , rv_cov_fused, 5*3600);

% Kepler Elemetns
[a_ukf,e_ukf,inc_ukf,RAAN_ukf,w_ukf,nu_ukf] =ijk2keplerian(x_ukf(1:3), x_ukf(4:6));
[a_mc,e_mc,inc_mc,RAAN_mc,w_mc,nu_mc] =ijk2keplerian(x_mc(1:3), x_mc(4:6));

% Plot uncertainty
[~]= Propagate(Period, r_fused, v_fused, 1);
plot_measurements_with_uncertainty([x_ukf(1:3)'/1000;x_mc(1:3)'/1000], cats_prop,↙
[{'UKF Uncertainty'}, {'MC Uncertainty'}])
```

```
function [R, V, rv_cov, refrence_time] = batch_angles(times, measurements, R_obs,↙
GPS_times, GPS_measurements,P_GPS, Epoch)
%-------------------------------------------------------------------------
% Batch least squares orbit estimation from ECI RA, DEC, and their rates
%-------------------------------------------------------------------------
% Inputs:
%   times           [Nx1]   times in seconds of angle observations (assumes
%                           ascending order
%
%   measurements    [Nx4]   [RA, DEC, RA_dot, DEC_dot] in radians and rad/s
%
%   R_obs           [4x4]   covariance of each measurement (constant diagonal)
%
%   GPS_measurments [Nx6]   matris of gps measurments to find the
%                           closes time to set as refrence orbit.  Each measurment↙
has r_x, r_y,
%                           r_z [in meters], v_x, v_y, v_z [m/s]
%
%   GPS_times       [NX1]   array of the times from the epoch time [sec]
%                           that the gps measurments were taken
%
%   P_GPS           [6x6]   GPS measurment covariance matrix first 3x3 in [m^2],
%                           second 3x3 in [(m/s)^2]
%
%   Epoch           datetime    The Epoch of all the measurements

% Outputs:
%   kep_est [6x1] [a, e, i, RAAN, omega, nu]
%
%   rv_cov  [6x6] covariance of coodinate system [m^2 and (m/s)^2]
%
%   refrence_time  The time [s] of the final calcualted refrence trajectory
%
% Assumptions:
%   - RA/DEC are in the ECI frame
%   - Keplerian two-body motion (no perturbations)


% Parameters
mu = 3.986004418e14; % m^3/s^2
N_obs = length(times);

% From closest GPS sighting
[~, idx] = min(abs(GPS_times - times(1)));
last_gps_sighting = Epoch + seconds(GPS_times(idx-1));  %idx-1 becuase our↙
measurments start at t=0

% Initial State and error
X_0_star = GPS_measurements(idx-1, :)';
x_0_bar =  X_0_star - X_0_star;  % We set the refrence trajectory equal to the↙
latest gps measrument.  Ajustable for further use
% X_0 = X_0_star + x_0_bar;

% Convergance Limits
```

```
tol = 1e-8;
max_iter = 20;
norm_tracker=[];

for iter = 1:max_iter

    % A priori estimate
    Lambda = P_GPS^-1;
    N = Lambda * x_0_bar;

    for k = 1:N_obs

        % Propogate X to time of measurmant k
        R_k = inv(R_obs);  % Editable for future use in case of non constant R_obs
        dt = Epoch + seconds(times(k)) - last_gps_sighting;
        Phi_k = Kepler_STM_UV(seconds(dt), X_0_star, mu, 'seconds');  % state
transition matrix for keplerian orbits
        X_pred = Phi_k * X_0_star;
        r = [X_pred(1:3)];
        v = [X_pred(4:6)];


        % Compute predicted RA/DEC and their rates in ECI
        RA_pred = wrapTo2Pi(atan2(r(2), r(1)));
        DEC_pred = asin(r(3)/norm(r));
        RA_dot_pred = (r(1)*v(2) - r(2)*v(1)) / (r(1)^2 + r(2)^2);
        DEC_dot_pred = (v(3) - (r(3)/norm(r))*dot(r,v)/norm(r)) / sqrt(r(1)^2 + r(2)
^2);

        dRA = wrapToPi(measurements(k,1) - RA_pred);
        dDEC = measurements(k,2) - DEC_pred;  % or wrapToPi if needed
        dRAdot = measurements(k,3) - RA_dot_pred;
        dDECdot = measurements(k,4) - DEC_dot_pred;
        y_k = [dRA; dDEC; dRAdot; dDECdot];

        % Analytic measurement Jacobian
        H_tilda_k = compute_H_analytic(r, v);
        H_k = H_tilda_k * Phi_k;

        % Update Observation Reading
        Lambda = Lambda + H_k' * (R_k \ H_k);
        N = N + H_k' * (R_k \ y_k);
    end

    % Update Covariance
    P_0 = inv(Lambda);

    % Batch Least Squares update
    x_0_hat = Lambda \ N;
    X_0_star = X_0_star + x_0_hat;
    x_0_bar = x_0_bar - x_0_hat;

    % Convergance Tracker
    norm_tracker=[norm_tracker; norm(x_0_hat)];
```

```
    if norm(x_0_hat) < tol
        break;
    end
end

R = X_0_star(1:3);
V = X_0_star(4:6);

% Covariance
rv_cov = diag(diag(P_0));

refrence_time = seconds(last_gps_sighting-Epoch);
end
```

```
function H = compute_H_analytic(r, v)
%-------------------------------------------------------------------------
% Computes analytically the H matrix for a batch filter taking in RA, DEc,
% and their rates and coverts them to r and v
%-------------------------------------------------------------------------
% Inputs: r and v [m] and [m/s] of refrence trajectory to find the H matrix for
    rx = r(1); ry = r(2); rz = r(3);
    vx = v(1); vy = v(2); vz = v(3);

    rnorm2 = rx^2 + ry^2 + rz^2;
    rhor2 = rx^2 + ry^2;
    rho_norm = sqrt(rnorm2);
    rhor = sqrt(rhor2);

    % RA
    dRA_dr = [-ry/rhor2, rx/rhor2, 0];
    dRA_dv = [0,0,0];

    % DEC
    dDEC_dr = [-rx*rz/(rhor2*rho_norm), -ry*rz/(rhor2*rho_norm), rhor/rnorm2];
    dDEC_dv = [0,0,0];

    % RA_dot
    dRAdot_dr = [(ry*(rx*vx+ry*vy)/rhor2 - vy)/rhor2, (-rx*(rx*vx+ry*vy)/rhor2 + vx)↙
/rhor2, 0];
    dRAdot_dv = [-ry/rhor2, rx/rhor2, 0];

    % DEC_dot
    dot_rv = dot(r,v);
    dDECdot_dr = [-(vx*rhor2 - rx*(dot_rv*rz/rho_norm^2))/(rho_norm*rhor), ...
                  -(vy*rhor2 - ry*(dot_rv*rz/rho_norm^2))/(rho_norm*rhor), ...
                  -(vz*rhor2 - rz*(dot_rv*rz/rho_norm^2) + dot_rv)/(rho_norm*rhor)];
    dDECdot_dv = [-rx*rz/(rho_norm*rhor), -ry*rz/(rho_norm*rhor), rhor/rho_norm];

    H = [dRA_dr, dRA_dv;
         dDEC_dr, dDEC_dv;
         dRAdot_dr, dRAdot_dv;
         dDECdot_dr, dDECdot_dv];
end
```

```
function [Pos, Vel, P_f, refrence_time] = sequential_rv_filter(times, measurements,↙
R_obs, mu, Epoch)
%-------------------------------------------------------------------------
% Extended Kalman Filter for orbit estimation from ECI position and
% velocity
%-------------------------------------------------------------------------
% Inputs:
%   times           [Nx1]   times in seconds of observations (assumes
%                           ascending order
%
%   measurements    [Nx4]   [r, v] in [m] and [v/s]
%
%   R_obs           [4x4]   covariance of each measurement (constant diagonal)
%
%   mu               [1]    Gravitational Parameter [m^3/s^2 ]
%
%   Epoch           datetime    The Epoch of all the measurements

% Outputs:
%   Pos     [3x1]  Estimated final position in ECI [m]
%
%   Vel     [3x1]  Estimated final velcoity in ECI [m]
%
%   P_f     [6x6] covariance of coodinate system [m^2 and (m/s)^2]
%
%   refrence_time  The time [s] of the final calcualted estimated trajectory
%
% Assumptions:
%   - initial state is in the ECI frame
%   - Keplerian two-body motion (no perturbations)

N = length(times);

% Initialization
X_0_star = measurements(1,:)';  % Refrence trajectory is first measurments
X_k_star = X_0_star;
P_k = R_obs(:,:,1);

% Initialize for History
X_k_hist = zeros(6, N);
X_k_hist(:, 1) = X_k_star;
P__k_hist = zeros(6, 6, N);
P__k_hist(:,:,1) = P_k;

% Initialize Times
t_prev = times(1);
Epoch_prev = Epoch;
dt_max = 5*60;  % 15 minutes in seconds largest propogation interval

for k = 2:N

    % Propagation Step
    dt_total = times(k) - t_prev;
```

```
    % Splitting if dt is to large:
    if dt_total<=dt_max
        %Single step propogation
        Phi = Kepler_STM_UV(dt_total, X_k_star, mu, 'seconds');
        X_pred = Phi * X_k_star;
        P_k = Phi* P_k *Phi';
    else
        N_steps = ceil(dt_total / dt_max);
        dt_step = dt_total / N_steps;
        X_temp = X_k_star;
        P_temp = P_k;
        for i = 1:N_steps
            Phi_step = Kepler_STM_UV(dt_step, X_temp, mu, 'seconds');
            X_temp = Phi_step * X_temp;
            P_temp = Phi_step * P_temp * Phi_step';
        end
        X_pred = X_temp;
        P_k = P_temp;
    end

    % Measurements Reading
    Y_k = measurements(k,:)';
    G = eye(length(measurements(1,:)));  % Measruements in r and v already
    y_k = Y_k - G*X_pred;
    H_k_tilda = eye(6);

    % Kalman gain
    if (isscalar(R_obs(1,1,:)))
        R = R_obs;
    else
        R = R_obs(:,:,k);
    end
    S = H_k_tilda * P_k * H_k_tilda' + R;
    K = P_k * H_k_tilda' / S;

    % Update
    x_k_hat = K * y_k;
    X_k_star = X_pred + x_k_hat;
    I = eye(6);
    P_k = (I - K*H_k_tilda) * P_k;
    Epoch_prev = Epoch_prev+seconds(dt_total);
    t_prev = times(k);

    % Save
    X_k_hist(:,k) = X_k_star;
    P__k_hist(:,:,k) = P_k;
end

X_f_star =  X_k_hist(:, end);
Pos = X_f_star(1:3);
Vel = X_f_star(4:6);
P_f = P__k_hist(:,:,end);
refrence_time = t_prev;
end
```

```
function Phi = Kepler_STM_UV(dts, varargin)
%-------------------------------------------------------------------------
% STM_UNIVERSAL  Compute 6x6 state transition matrix using universal variables
%-------------------------------------------------------------------------
% Inputs are Variation but should be:
%   State - 6x1 initial position [m] and velocity [m/s] in cartesian
%   dt    - time of flight (t - t0) [s]
%   mu    - gravitational parameter [m^3/s^2] (optional; default Earth)
%
% Output:
%   Phi  - 6x6 state transition matrix
%
% Uses universal variable formulation (Goodyear)


% standard error
    error(nargchk(3, 9, nargin));%#ok

    % parse input
    narg = nargin;
    time_unit = 'days';
    if (narg >= 8)
        % otherwise
        r1 = [varargin{1:3}];
        v1 = [varargin{4:6}];
        muC = varargin{7};
    end
    if (narg == 9), time_unit = varargin{8}; end
    if (narg <= 4)
        % otherwise
        r1 = varargin{1}(1:3);
        v1 = varargin{1}(4:6);
        muC = varargin{2};
    end
    if (narg == 4), time_unit = varargin{3}; end

    % force everyting to be row-matrices
    r1 = r1(:).';   v1 = v1(:).';

    % initialize output
    zero                        = zeros(numel(dts), 1);
    exitflag                    = zero;
    final_states                = NaN(numel(dts), 6);
    output.Kepler_iterations    = zero;
    output.cont_frac_iterations = zero;
    output.time_error           = zero;

    % progress only one orbit (to command window, debugging puruposes only)
    if numel(r1) ~= 3
        error('progress_orbit:only_one_body_allowed',...
              'I can only progress one orbit at a time. Please use ARRAYFUN().');
    end

    % times are given in days, unless specified otherwise
```

```
if strcmpi(time_unit, 'days')
    dts = dts * 86400;
elseif ~strcmpi(time_unit, 'seconds')
    error('progress_orbit:invalid_timeunit',...
        'Only ''seconds'' and ''days'' are valid timeunits.');
end

% intitialize
nu0  = r1*v1.';
r1m  = sqrt(r1*r1.');
beta = 2*muC/r1m - v1*v1.';

% period effects
DeltaU = zero;
if (beta > 0)
    P = 2*pi*muC*beta^(-3/2);
    n = floor((dts + P/2 - 2*nu0/beta)/P);
    DeltaU = 2*pi*n*beta^(-5/2);
end

% loop through all requested time steps
for i = 1:numel(dts)

    % extract current time step
    dt = dts(i);

    % quick exit for trivial case
    if (dt == 0)
        final_states(i, :) = [r1,v1];
        exitflag(i) = 1;
        output.Kepler_iterations(i)    = 0;
        output.cont_frac_iterations(i) = 0;
        output.time_error(i)           = 0;
        continue;
    end

    % loop until convergence of the time step
    u = 0; t = 0; qisbad = false; iter = 0; cont_frac = 0; deltaT = t-dt;
    while abs(deltaT) > 1 % one second accuracy seems fine

        % increase iterations
        iter = iter + 1;

        % compute q
        % NOTE: [q] may not exceed 1/2. In principle, this will never
        % occur, but the iterative nature of the procedure can bring
        % it above 1/2 for some iterations.
        bu = beta*u*u;
        q  = bu/(1 + bu);

        % escape clause;
        % The value for [q] will almost always stabilize to a value less
        % than 1/2 after a few iterations, but NOT always. In those
        % cases, just use repeated coordinate transformations
```

```
        if (iter > 25) || (q >= 1), qisbad = true; break; end

        % evaluate continued fraction (when q < 1, always converges)
        A =  1;   B = 1;   G = 1;   n = 0;
        k = -9;   d = 15;  l = 3;   Gprev = inf;
        while abs(G-Gprev) > 1e-14
            k = -k;                 l = l + 2;
            d = d + 4*l;            n = n + (1+k)*l;
            A = d/(d - n*A*q);      B = (A-1)*B;
            Gprev = G;              G = G + B;
            cont_frac = cont_frac + 1;
        end % continued fraction evaluation

        % continue kepler loop
        U0w2   = 1 - 2*q;
        U1w2   = 2*(1-q)*u;
        U      = 16/15*U1w2^5*G + DeltaU(i);
        U0     = 2*U0w2^2-1;
        U1     = 2*U0w2*U1w2;
        U2     = 2*U1w2^2;
        U3     = beta*U + U1*U2/3;
        r      = r1m*U0 + nu0*U1 + muC*U2;
        t      = r1m*U1 + nu0*U2 + muC*U3;
        deltaT = t - dt;
        % Newton-Raphson method works most of the time, but is
        % not too stable; the method fails far too often for my
        % liking...
        % u = u - deltaT/4/(1-q)/r;
        % Halley's method is much better in that respect. Working
        % out all substitutions and collecting terms gives the
        % following simplification:
        u = u - deltaT/((1-q)*(4*r + deltaT*beta*u));
    end % time loop

    % do it the slow way if state transition matrix fails for some q
    if qisbad
        % repeated coordinate transformations
        [aa, ee, ii, OO, oo, nu] = ijk2keplerian(r1, v1);

        % Compute mean anomaly
        cosE = (ee + cos(nu)) ./ (1 + ee*cos(nu));
        sinE = sqrt(1 - ee^2) .* sin(nu) ./ (1 + ee*cos(nu));
        E = atan2(sinE, cosE);    % robust quadrant
        E = mod(E, 2*pi);
        M = E - e .* sin(E);

        % Update mean anomly
        M = M + sqrt(muC/abs(aa)^3)*dt;

        % Convert tru anomly
        M = mod(M, 2*pi);
        E = M; % initial guess
        tol = 1e-12;
        diff = 1;
```

```
            while abs(diff) > tol
                diff = E - ee*sin(E) - M;
                E = E - diff ./ (1 - ee*cos(E));
            end
            nu = 2 * atan2( sqrt(1+ee)*sin(E/2), sqrt(1-ee)*cos(E/2) );
            nu = mod(nu, 2*pi);

            [x, y, z, xd, yd, zd] =  keplerian2ijk(aa, ee, ii, OO, oo, M);
            final_states(i, :) = [x, y, z, xd, yd, zd];
            % this means failure
            exitflag(i) = -1;

        % use state transition matrix if all went well
        else
            % Kepler solution
            f = 1 - muC/r1m*U2;      F = -muC*U1/r/r1m;
            g = r1m*U1 + nu0*U2;     G = 1 - muC/r*U2;
            % create new position and velocity matrices
            final_states(i, :) = [r1*f+v1*g, r1*F+v1*G];
            % all went fine
            exitflag(i) = 1;
        end % "q is bad" clause

        Phi =  [eye(3)*f  eye(3)*g; eye(3)*F eye(3)*G];


        % process output
        output.Kepler_iterations(i)    = iter;
        output.cont_frac_iterations(i) = cont_frac;
        output.time_error(i)           = abs(t-dt);

    end % loop through [dt]

    % generate properly formatted output
    narg = nargout;
    if (narg <= 3)
        varargout{1} = final_states;     % and output array
        varargout{2} = exitflag;         % exitflag
        varargout{3} = output;           % output
    elseif (narg > 3)
        varargout{1} = final_states(:, 1);  varargout{4} = final_states(:, 4);
        varargout{2} = final_states(:, 2);  varargout{5} = final_states(:, 5);
        varargout{3} = final_states(:, 3);  varargout{6} = final_states(:, 6);
        varargout{7} = exitflag;         % exitflag
        varargout{8} = output;           % output
    end % process output

end % progress orbit
```

```
function J = cart2kepJac(r,v)
%-------------------------------------------------------------------------
% Computes the numerical jacobian from cartisian coordiantes to orbital
% elemnts
%-------------------------------------------------------------------------
% Inputs are Variation but should be:
%   r     - 3x1 initial position [m] in cartesian
%   v     - 3x1 initial velocity [m/s] in cartesian
% Output:
%   J  - 6x6 Jacobian Matrix
%
    x = [r; v];
    kep0=zeros(1,6);
    [kep0(1), kep0(2), kep0(3), kep0(4), kep0(5), kep0(5), kep0(6)]= ijk2keplerian↙
(r,v);
    J = zeros(6,6);
    for j = 1:6
        for k = 1:6
            eps = 1e-6*x(k);
            dx = zeros(6,1);
            dx(k) = eps;
            kep_plus=zeros(1,6);
            [kep_plus(1), kep_plus(2), kep_plus(3), kep_plus(4), kep_plus(5),↙
kep_plus(6)] = ijk2keplerian(x(1:3)+dx(1:3), x(4:6)+dx(4:6));
            J(j,k) = (kep_plus(j) - kep0(j))/eps;
        end
    end
end
```

```
function [r_f,v_f, r, v, time] = Propagate(tf, r0, v0, Plot, dt)
%-------------------------------------------------------------------------
% Uses STM to propagate an orbit forward
%-------------------------------------------------------------------------
% Inputs are Variation but should be:
%   tf     - final time to prpogate to [s]
%   r0     - 3x1 initial position [m] in cartesian
%   v0     - 3x1 initial velocity [m/s] in cartesian
%   Plot   - True or False whether to plot the orbit
%   dt     - optional delta to propagate
% Output:
%   J  - 6x6 Jacobian Matrix
%
mu = 3.986004418e14; % m^3/s^2

if nargin < 5
    dt=10;
end

%J2=.001082; %Non spherical coefficient


%% Calculation of Oribt for Earth as a Sphere
counter=0; %used to count each itiration
t=0;  %initial time
r_temp=r0;  %initial position (as a row) [m]
v_temp=v0;  %initial velocity (as a row) [m/s]
time=[];
while t<=tf
    counter=counter+1;
    time(counter)=t;
    r(counter, :)=r_temp;   %next position iteratio
    v(counter, :)=v_temp;   %next velocit iteration

    %Runga Kutta m4th order method for both r and v
    k1=dt*Derivative(r(counter, :), v(counter, :), mu);
    k2=dt*Derivative(r(counter, :)+k1(1, :)/2, v(counter, :)+k1(2, :)/2, mu);
    k3=dt*Derivative(r(counter, :)+k2(1, :)/2, v(counter, :)+k2(2, :)/2, mu);
    k4=dt*Derivative(r(counter, :)+k3(1, :), v(counter, :)+k3(2, :), mu);
    t=t+dt; %increase iteration of time
    r_temp=r(counter, :)+(k1(1,:)+2*k2(1,:)+2*k3(1,:)+k4(1,:))/6; %set next↙
iteration of r to runga kutta approxiamtion
    v_temp=v(counter, :)+(k1(2,:)+2*k2(2,:)+2*k3(2,:)+k4(2,:))/6;  %set next↙
iteration of v to runga kutta approxiamtion
end

r_f=r(end,:)';
v_f=v(end,:)';

if Plot
    figure
    earth_sphere('km');
    axis equal
    hold on
```

```
    % Plot propagated orbit
    plot3(r(:, 1)/1000, r(:, 2)/1000, r(:, 3)/1000, 'b', 'LineWidth', 1.5);
    % Labels and title
    title('Orbit of Satellite Around (Spherical) Earth');
    xlabel('x in ECI [km]');
    ylabel('y in ECI [km]');
    zlabel('z in ECI [km]');
    grid on
    hold off
end
end

function M = Derivative(x, y, mu)
M(1, :)=y;   %x dot, y dot and v dot are already given in the velocity vector
M(2, :)=-mu*(x/(norm(x)^3));  %accoridng to equation r double dot = -u* r hat / r↙
magnitude ^3
end

% function state_der = SatelliteDynODE(t, state, Mu) %, Re, j2, A_D, C_D, m, C_sr,↙
A_sp, Date)
%     r_vec = [state(1) state(2) state(3)]';
%     v_vec = [state(4) state(5) state(6)]';
%     r = norm(r_vec);
%     %[a,e,i,w,OMEGA,Nu] = ijk2keplerian(r_vec, v_vec);
%     %F_d = Drag_Per(Re, A_D, C_D, m, r, v_vec);
%     %F_j = j2_Per(Re, Mu, r, i, Nu, w, OMEGA, j2);
%     %F_s = Solar_Per(Date, C_sr, A_sp, m, r_vec);  % Pass r_vec to Solar_c
%     F_tot=[0, 0, 0]; %F_tot = F_d + F_j + F_s;
%     X_2dot = -(Mu/r^3)*r_vec(1) + F_tot(1);
%     Y_2dot = -(Mu/r^3)*r_vec(2) + F_tot(2);
%     Z_2dot = -(Mu/r^3)*r_vec(3) + F_tot(3);
%     state_der = [v_vec; [X_2dot Y_2dot Z_2dot]'];
% end
```

```
function Positions = plot_ground_station(GS, Plot)
%-------------------------------------------------------------------------
% plot_ground_station: plots a ground station on a spherical Earth
%-------------------------------------------------------------------------
%
% Inputs:
%   GS          - List of Nx3 of geodetic lat [deg], long [deg], alt [km]
%   markerSize  - Size of the dot (optional, default 100)
%   markerColor - Color of the dot (optional, default 'r')

lat_list = GS(:,1);
lon_list = GS(:,2);
alt_list = GS(:,3);

markerSize = 100;  % default size

% Earth's radius (mean) in km
Re = 6378;
if Plot
    hold on
end
% Convert geodetic to ECEF (simple spherical Earth) and Plot GS
N = length(GS(:,1));
legendNames = cell(1, N);

Positions = [];

for k = 1:N
    lat = deg2rad(lat_list(k));
    lon = deg2rad(lon_list(k));
    r = Re + alt_list(k);
    x = r * cos(lat) * cos(lon);
    y = r * cos(lat) * sin(lon);
    z = r * sin(lat);
    if Plot
        legendNames{k} = sprintf('Ground Station %d', k);
        handles(k) = scatter3(x, y, z, markerSize);
    end
    Positions = [Positions; x*1000,y*1000,z*1000];
end

if Plot
    legend(handles, legendNames);
    hold off
end

end
```

```
function plotOrbitVisibility(a_fused,e_fused,inc_fused,RAAN_fused,w_fused, nu_fused,↙
gsECI)
%-------------------------------------------------------------------------
% plotOrbitVisibility
%-------------------------------------------------------------------------
% Inputs:
%   meas = initial keplerian elements [a, e, i, RAAN, w, nu]
%   groundStations = struct array:
%       groundStations(j).lat   = [deg]
%       groundStations(j).lon   = [deg]
%       groundStations(j).alt   = [km]
%   mu   = gravitational parameter
%   t0   = initial time (seconds)
%   tf   = final time (seconds)
%   dt   = propagation step (seconds)
%

% Convert initial orbital elements to ECI
k = linspace(0, 360, 720*2);
r_test = zeros(length(k), 3);
v_test = zeros(length(k), 3);

for counter = 1:length(k)
    [r(counter, :), v(counter, :)] = keplerian2ijk(a_fused, e_fused, inc_fused,↙
RAAN_fused, w_fused, k(counter));
end

% Convert ground stations to ECEF/ECI
numGS = 2;
N=length(k);
% Determine LoS visibility for each point
vis1 = false(N,1);
vis2 = false(N,1);

for i = 1:length(k)
    [el1, ~] = elevazim(r(i,:), gsECI(1,:));
    [el2, ~] = elevazim(r(i,:), gsECI(2,:));

    vis1(i) = el1 > 0;
    vis2(i) = el2 > 0;
end

% Four visibility cases
idx_none = ~vis1 & ~vis2;
idx_gs1  =  vis1 & ~vis2;
idx_gs2  = ~vis1 &  vis2;
idx_both =  vis1 &  vis2;

% Split r into 4 sets
r_none = r(idx_none, :);
r_gs1  = r(idx_gs1,  :);
r_gs2  = r(idx_gs2,  :);
r_both = r(idx_both, :);
```

```
% Assign Colors
% None visible  → black
% GS1 only      → blue
% GS2 only      → red
% Both          → green

% colors = zeros(N,3);
%
% for k = 1:N
%     if vis1(k) && vis2(k)
%         colors(k,:) = [0,1,0];          % green
%     elseif vis1(k)
%         colors(k,:) = [0,0,1];          % blue
%     elseif vis2(k)
%         colors(k,:) = [1,0,0];          % red
%     else
%         colors(k,:) = [0,0,0];          % black
%     end
% end

% Plot the Earth
figure
hold on
earth_sphere('km');
axis equal
hold on

hNone = scatter3(r_none(:,1)/1000, r_none(:,2)/1000, r_none(:,3)/1000, 10, 'k');
hGS1  = scatter3(r_gs1(:,1)/1000,  r_gs1(:,2)/1000,  r_gs1(:,3)/1000, 10, 'b');
hGS2  = scatter3(r_gs2(:,1)/1000,  r_gs2(:,2)/1000,  r_gs2(:,3)/1000, 10, 'r');
hBoth = scatter3(r_both(:,1)/1000, r_both(:,2)/1000, r_both(:,3)/1000, 10, [0.5 0↙
0.5]); % purple

hold on
earth_sphere('km');
axis equal

hold on
hgs1=scatter3(gsECI(1,1)/1000, gsECI(1,2)/1000, gsECI(1,3)/1000, 200, 'b',↙
'filled');  % GS1 color
hgs2=scatter3(gsECI(2,1)/1000, gsECI(2,2)/1000, gsECI(2,3)/1000, 200, 'r',↙
'filled');  % GS2 color



xlabel('x [km]'); ylabel('y [km]'); zlabel('z [km]');
title('Orbit Visibility by Ground Stations');
legend([hNone hGS1 hGS2 hBoth hgs1 hgs2], ...
       {'No LOS','LOS GS1 Only','LOS GS2 Only','LOS Both', 'GS1 Station', 'GS2↙
Station'}, ...
       'Location','bestoutside');
grid on;
hold off;
```

```
end
```

```
function plot_measurements_with_uncertainty(r_meas, P_meas, lab)
%-------------------------------------------------------------------------
% Adds measurements and their 3D uncertainty ellipsoids to an existing figure.
%-------------------------------------------------------------------------
%
% Inputs:
%   r_meas : Nx3 matrix of measurement positions [x, y, z]
%   P_meas : 3x3xN covariance matrices for each measurement
%   color  : optional, color of the ellipsoids and measurement markers
%
% Example:
%   plot_measurements_with_uncertainty(r_meas, P_meas, 'r');

if nargin < 3
    lab = [];
end

color=['m'; 'y'; 'g'; 'c'];

N = size(r_meas,1);

% figure
hold on;
htemp= [];

newHandles=[];
newLabels=[];

for i = 1:N
    % Plot measurement point
    scatter3(r_meas(i,1), r_meas(i,2), r_meas(i,3), 50, color(i), 'filled');

    % Eigen decomposition to get axes of the ellipsoid
    [V, D] = eig(P_meas(:,:,i));

    % Take square root of eigenvalues
    D_sqrt = diag(sqrt(diag(D)));  % 3x3 diagonal matrix of std deviations

    % Generate a unit sphere
    [x, y, z] = ellipsoid(0, 0, 0, 1, 1, 1, 20);

    % Flatten sphere points to 3xN
    sphere_pts = [x(:)'; y(:)'; z(:)'];

    % Rotate and scale sphere to match covariance
    ellipsoid_pts = V * D_sqrt * sphere_pts;  % 3xN

    % Translate to measurement position
    x_e = reshape(ellipsoid_pts(1,:) + r_meas(i,1), size(x));
    y_e = reshape(ellipsoid_pts(2,:) + r_meas(i,2), size(y));
    z_e = reshape(ellipsoid_pts(3,:) + r_meas(i,3), size(z));

    % Plot transparent ellipsoid
    htemp(i) = surf(x_e, y_e, z_e, 'FaceColor', color(i), 'FaceAlpha', 0.2,↙
```

```
'EdgeColor', 'none');

    if nargin < 3
        newHandles = [newHandles; htemp(i)];
        newLabels = [newLabels, lab(i)];
    end
end
lgd  = legend;
existingHandles = lgd.PlotChildren;   % list of graphics handles already in legend
existingLabels  = lgd.String;
if ~(nargin <3)
    newHandles = [existingHandles(1:6); htemp(1); htemp(2); htemp(3); htemp(4)];
    newLabels  = [existingLabels(1:6), {'GS1 Uncertainty'}, {'GS2 Uncertainty'},↙
{'GPS Uncertainty'}, {'Fused Uncertainty'}];
end
legend(newHandles, newLabels);
axis equal;
xlabel('X [km]'); ylabel('Y [km]'); zlabel('Z [km]');
grid on;
hold off;

end
```

```
function [elevation, azimuth] = elevazim(satLoc,obsLoc)
%-------------------------------------------------------------------------
% Calculates the elevation and azimuth from a reference position specified in ECEF coordinates (e.g. antenna
% location) to another position specified in ECEF coordinates (e.g. satellite location)
%-------------------------------------------------------------------------
%
% input:'satLoc'  matrix of rows that contain ECEF coordiantes of satellites [m]
%                 could be differentiated by different satelits or
%                 different times for the same satellite
%       'obsLoc' vector which contains the ECEF coordinates
%                (meters) of a reference position (Ground station) which elevation and azimuth
%                will be calculated from
%
% output: the elevation and azimuth look angles [degrees] to the satellite
% from the ground station
%


degrad = pi/180.0;
% define satellite locations in ECEF coordinates
    satX = satLoc(:,1); % meters
    satY = satLoc(:,2); % meters
    satZ = satLoc(:,3); % meters
% define observation location in ECEF coordinates
    obsX = obsLoc(1);       % meters
    obsY = obsLoc(2);       % meters
    obsZ = obsLoc(3);       % meters
% compute unit vector from observation station to satellite position
    r = sqrt((satX - obsX).^2 + (satY - obsY).^2 + ...
        (satZ - obsZ).^2);
    dx = (satX - obsX) ./ r;
    dy = (satY - obsY) ./ r;
    dz = (satZ - obsZ) ./ r;
% compute the observation latitude and longitude
    obsLoc = latlong(obsLoc);
    latOBS = obsLoc(1) * degrad;        % radians
    longOBS = obsLoc(2) * degrad;       % radians
% compute the rotated unit vectors in VEN from observation station to satellite position
    north = dz .* cos(latOBS) - sin(latOBS) .* ...
        (dx .* cos(longOBS) + dy .* sin(longOBS));
    east = dy .* cos(longOBS) - dx .* sin(longOBS);
    vertical = cos(latOBS) .* (dx .* cos(longOBS) + ...
        dy .* sin(longOBS)) + dz .* sin(latOBS);
% compute elevation
    elevation = (pi / 2 - acos(vertical)) ./ degrad;    % degrees
% compute azimuth; check for negative angles
    azimuth = atan2(east,north);                % radians
    idx = find(azimuth < 0);
    azimuth(idx) = azimuth(idx) + 2 * pi;
    azimuth = azimuth ./ degrad;                % degrees
```

```
    return;



function ecoord = latlong(location)
% get ECEF location to be converted to latitude-longitude-altitude
degrad = pi/180.0;
% coordinates
    ECEFx = location(:,1);
    ECEFy = location(:,2);
    ECEFz = location(:,3);
% compute the longitude which is an exact calculation
    long = atan2(ECEFy , ECEFx);    % radians
% compute the latitude using iteration
    p = sqrt(ECEFx.^2 + ECEFy.^2);
    % compute approximate latitude
    AA = 6378137.00000;                 % meters
    BB = 6356752.31425;                 % meters
    esquare=(AA^2 - BB^2) / AA^2;
    lat0 = atan((ECEFz ./ p) ./ (1 - esquare));
    stop = 0;
    while (stop == 0)
        N0 = AA^2 ./ (sqrt(AA^2 * (cos(lat0)).^2 + ...
             BB^2 .* (sin(lat0)).^2));
        altitude = (p ./ cos(lat0)) - N0;       % meters
        % calculate improved latitude
        term = (1 - esquare * (N0 ./ (N0 + altitude))).^(-1);
        lat = atan(ECEFz ./ p .* term);         % radians
        % check if result is close enough,
        if (abs(lat - lat0) <  1.0e-12)
            stop = 1;
        end
        lat0 = lat;
    end
% convert the latitude and longitude to degrees
    latitude = lat ./ degrad;       % degrees
    longitude = long ./ degrad;     % degrees
% return location in latitude-longitude-altitude coordinates
    ecoord = [ latitude longitude altitude ];
    return;
```

```
function result = EL_AZ_Ploter(el_matrix,az_matrix)
%-------------------------------------------------------------------------
% Generates a sky plot for satellites whose relative elevation and azimuth
% values are given as matrices.
%-------------------------------------------------------------------------
%
% input:    'svid'          a 1 x num_sv or num_sv x 1 vector of SV prn numbers
%
%           'el_matrix'     a N x 1 array which contains a series of elevation
measurements [deg]
%
%           'az_matrix'     a N x 1 array which contains a series of azimuth
measurements [deg]
%
%
% output: 'result' indicates if the data have been plotted successfully

colormap(lines);
cmap=colormap;
color_dark_gray=[1,1,1]*0.5;
color_light_gray=[1,1,1]*0.95;
s_vec=linspace(0,2*pi,101);

% Figure construction
figure;

% Create auxiliary axes and marking
plot([-90,90],[0,0],'Color',color_dark_gray);
hold on; axis equal;
plot([0,0],[-90,90],'Color',color_dark_gray);
plot(30*sin(s_vec),30*cos(s_vec),'Color',color_dark_gray);
plot(60*sin(s_vec),60*cos(s_vec),'Color',color_dark_gray);
plot(90*sin(s_vec),90*cos(s_vec),'Color',color_dark_gray);
text(-3,95,'N','Color',color_dark_gray);
text(3,63,'30','Color',color_dark_gray);
text(3,33,'60','Color',color_dark_gray);
text(3,3,'90','Color',color_dark_gray);
text(-3,-95,'S','Color',color_dark_gray);


% Check visibility changes in time for current SV
sign_vec=[0;sign(el_matrix(:))];
diff_sign_vec=diff(sign_vec);
diff_sign_vec_ind=[find(diff_sign_vec~=0);size(el_matrix,2)];

% draw visible/hidden trajectory segments
for index2=1:length(diff_sign_vec_ind)-1

    % Calc values for current segment
    sign_type=diff_sign_vec(1)*(-1)^(index2-1);
    el_values=el_matrix(diff_sign_vec_ind(index2):diff_sign_vec_ind(index2+1));
    az_values=az_matrix(diff_sign_vec_ind(index2):diff_sign_vec_ind(index2+1));
    x_values=(90-abs(el_values)).*(sin(az_values*pi/180));
    y_values=(90-abs(el_values)).*(cos(az_values*pi/180));
```

```
    if sign_type==1
        % Case of a visible segment
        h_line=plot(x_values,y_values,'Linewidth',2);
    else
        % Case of an hidden segment
        h_line=plot(x_values,y_values,'LineStyle','--');
    end

    % Additional marking for trajectory start/end points
    set(h_line,'Color',cmap(1,:));
    if index2==1
        h_init=plot(x_values(1),y_values(1),'o','Color',cmap(1,:),'MarkerSize',8);
        %text(x_values(1)+5,y_values(1),mat2str(svid(index1)),'Color',cmap↙
(index1,:),'BackgroundColor',color_light_gray);
        if sign_type==1
            set(h_init,'LineWidth',2);
        end
    end
    if index2==(length(diff_sign_vec_ind)-1)
        h_end=plot(x_values(end),y_values(end),'x','Color',cmap(1,:),'MarkerSize',↙
8);
        if sign_type==1
            set(h_end,'LineWidth',2);
        end
    end

end
set(gca,'Xlim',[-100,100],'Ylim',[-100,100],'Xtick',[],'Ytick',[]);
result=1;
end
```

```
function [x_ukf, P_ukf] = ukf_propagation(x0, P0, tprop)
%--------------------------------------------------------------------------
% UKF_PROPAGATION Propagate orbit uncertainty using Unscented Transform
%--------------------------------------------------------------------------
%
% Inputs:
%   x0    : 6x1 initial state [r0; v0]
%   P0    : 6x6 initial covariance
%   tprop : propagation time [s]
%   mu    : gravitational parameter [m^3/s^2]
%
% Outputs:
%   x_ukf : propagated mean state
%   P_ukf : propagated covariance

    n = 6;              % state dimension
    alpha = 1e-3;
    beta = 2;
    kappa = 0;

    lambda = alpha^2*(n + kappa) - n;

    % Sigma point weights
    Wm = [lambda/(n+lambda); 0.5/(n+lambda)*ones(2*n,1)];
    Wc = Wm;
    Wc(1) = Wm(1) + (1 - alpha^2 + beta);

    % Sigma points
    % P0 = (P0 + P0')/2;  % force symmetry
    % [eigvec, eigval] = eig(P0);
    % eigval(eigval < 0) = 0;  % set negative eigenvalues to zero
    % P_psd = eigvec * eigval * eigvec';
    %
    P0 = (P0 + P0') / 2;
    % Eigen-decomposition
    [V, D] = eig(P0);
    D(D < 0) = 0;           % zero negative eigenvalues
    P_clean = V * D * V';
    % Optional small diagonal for strict PD
    P_clean = P_clean + 1e-12 * eye(size(P_clean));



    S = chol((n+lambda)*P_clean, 'lower');
    Xi = [x0, x0+S, x0-S];    % 6 x (2n+1)

    % Propagate sigma points
    Xi_prop = zeros(size(Xi));
    for i = 1:(2*n+1)
        [r_temp,v_temp, ~, ~, ~] = Propagate(tprop, Xi(1:3,i), Xi(4:6,i), 0);
        Xi_prop(:,i) = [r_temp; v_temp];
    end

    % Reconstruct mean
```

```
    x_ukf = Xi_prop * Wm;

    % Reconstruct covariance
    P_ukf = zeros(n);
    for i = 1:(2*n+1)
        dx = Xi_prop(:,i) - x_ukf;
        P_ukf = P_ukf + Wc(i) * (dx*dx');
    end
end
```

```
function [x_mc, P_mc] = monte_carlo_propagation(r0, v0, P0, tprop, N)
%-------------------------------------------------------------------------
% MONTE_CARLO_PROPAGATION Propagate orbit uncertainty using Monte Carlo
%-------------------------------------------------------------------------
%
% Inputs:
%   x0    : 6x1 initial state [r0; v0]  [m, m/s]
%   P0    : 6x6 initial covariance  [m^2, (m/s)^2]
%   tprop : propagation time [s]
%   mu    : gravitational parameter [m^3/s^2]
%   N     : number of Monte Carlo particles
%
% Outputs:
%   x_mc  : propagated mean state
%   P_mc  : propagated covariance

    % Generate Monte Carlo particles
    P0 = (P0 + P0')/2;  % force symmetry
    [eigvec, eigval] = eig(P0);
    eigval(eigval < 0) = 0;  % set negative eigenvalues to zero
    P_psd = eigvec * eigval * eigvec';
    X0 = mvnrnd([r0;v0], P_psd, N)';   % 6 x N

    % Preallocate
    Xprop = zeros(size(X0));

    % Propagate each particle
    for i = 1:N
        [r_mc(:,i) v_mc(:,i), ~, ~, ~] = Propagate(tprop, X0(1:3, i), X0(4:6, i),↙
0);
    end

    % Compute mean and covariance
    x_mc = mean([r_mc; v_mc], 2);
    P_mc = cov([r_mc; v_mc]');
end
```

```
function [kep_est_final, rv_cov_final] = batch_gps_multiarc(times, measurements,↙
R_obs, arc_length, Epoch)
%-------------------------------------------------------------------------
% Multi-arc batch LS where the last observation of each arc becomes the first↙
observation of the next arc
%-------------------------------------------------------------------------
%
% Inputs:
%   times        [Nx1] time stamps
%   measurements [Nx6] r,v measurements
%   R_obs        [6x6] measurement covariance
%   arc_length   arc duration in seconds
%
% Outputs:
%   kep_est_final  final arc's Kepler estimate
%   rv_cov_final   final arc's covariance
%-------------------------------------------------------------------------

N = length(times);
arc_solutions = struct([]);

% ------------------------
% Build arc index ranges
% ------------------------
arc_starts = [];
arc_ends   = [];

i1 = 1; % start index

while true
    t_start = times(i1);
    t_end   = t_start + arc_length;

    % find max index within arc length
    i2 = find(times <= t_end, 1, 'last');

    if isempty(i2) || i2 <= i1
        break;
    end

    arc_starts(end+1) = i1;
    arc_ends(end+1)   = i2;

    % Next arc starts at the LAST index of current arc
    i1 = i2;

    if i1 >= N
        break;
    end
end

n_arcs = length(arc_starts);

% ------------------------
```

```
% Process each arc
% ------------------------
for j = 1:n_arcs
    idx1 = arc_starts(j);
    idx2 = arc_ends(j);

    t_arc   = times(idx1:idx2);
    measArc = measurements(idx1:idx2,:);

    % Perform batch LS on the arc
    [kep_est, rv_cov] = batch_gps(t_arc, measArc, R_obs, Epoch, measurements(1,:));

    arc_solutions(j).kep   = kep_est;
    arc_solutions(j).P     = rv_cov;
    arc_solutions(j).range = [idx1 idx2];

    % Propagate final state to next arc's first timestamp
    if j < n_arcs
        % Convert to Cartesian
        [r0, v0] = keplerian2ijk(kep_est(1), kep_est(2), kep_est(3), kep_est(4),↙
kep_est(5), kep_est(6));

        % Propagate to next arc start time
        dt = times(arc_starts(j+1)) - t_arc(1);

        [r_next, v_next] = propagateOrbit([Epoch,Epoch+seconds(dt)], r0, v0);

        % Overwrite the first measurement of next arc's start
        measurements(arc_starts(j+1),1:3) = r_next(:, end)';
        measurements(arc_starts(j+1),4:6) = v_next(:, end)';
    end
end

% Final outputs = last arc
kep_est_final = arc_solutions(end).kep;
rv_cov_final  = arc_solutions(end).P;

end
```

```
function [a, e, i, RAAN, omega, M] = cart2kep(r,v,mu)
%--------------------------------------------------------------------------
% Converts ECI state orbit to Orbital elements
%--------------------------------------------------------------------------
% Inputs: r and v [m] and [m/s] of orbit
%         mu  [m^3/s^2]
%
% Outputs: a [m] - semi major axis
%          e  - eccentricity
%          i [deg] inclination
%          RAAAN [deg] Right angle of ascending node
%          omega [deg]  argument of perigee
%          M [deg]  mean anomaly
%
%

    h = cross(r,v);
    h_mag = norm(h);
    n = cross([0;0;1],h);
    n_mag = norm(n);
    e_vec = cross(v,h)/mu - r/norm(r);
    e = norm(e_vec);
    a = 1/(2/norm(r) - norm(v)^2/mu);
    i = acos(h(3)/h_mag);
    RAAN = atan2(h(1), -h(2));
    omega = atan2(dot(cross(n,e_vec),h)/h_mag, dot(n,e_vec));
    nu = atan2(dot(cross(e_vec,r),h)/h_mag, dot(e_vec,r));

    if e < 1 - 1e-12         % elliptic
        % Compute eccentric anomaly E robustly from nu
        denom = 1 + e * cos(nu);
        cosE = (e + cos(nu)) ./ denom;
        sinE = sqrt(max(0, 1 - e^2)) .* sin(nu) ./ denom;
        E = atan2(sinE, cosE);
        E = mod(E, 2*pi);
        M = E - e .* sin(E);
        M = mod(M, 2*pi);    % normalized
    elseif e > 1 + 1e-12    % hyperbolic
        % hyperbolic anomaly F (sometimes H)
        denom = 1 + e * cos(nu);
        if abs(denom) < eps
            warning('Denominator near zero when converting nu to hyperbolic anomaly;↙
result may be unstable.');
        end
        sinhF = sqrt(e^2 - 1) .* sin(nu) ./ denom;
        F = asinh(sinhF);
        M = e .* sinh(F) - F;   % hyperbolic "mean anomaly" (signed)
        % do not normalize M to [0,2pi) for hyperbola
    else
        error('Parabolic orbit (e ~ 1): mean anomaly is not defined; use Barker''s↙
equation for time-of-flight.');
    end

    kep = [a; e; i; RAAN; omega; M];
```

```
end
```