```julia
module SpaceFlightDynamics

using LinearAlgebra

# Struct to hold orbital elements
# a    : Semi-major axis (km)
# e    : Eccentricity
# i_deg: Inclination (deg)
# Ω_deg: Right Ascension of the Ascending Node (deg)
# ω_deg: Argument of perigee (deg)
# ν_deg: True anomaly (deg)
struct OrbitalElements
        a::Float64
        e::Float64
        i_deg::Float64
        Ω_deg::Float64
        ω_deg::Float64
        ν_deg::Float64
end

# Struct to hold state vectors: position (r) in km and velocity (v) in km/s
struct StateVectors
        r::Vector{Float64}
        v::Vector{Float64}
end

"""
        oe_to_sv(oe::OrbitalElements; mu=398600.4418)

        Converts orbital elements given in `oe` to the corresponding position (r) and velocity (v) state vectors.

        - oe is the vector of orbital elements
        - `mu` is the gravitational parameter (default is for Earth, in km³/s²).

        Returns an instance of `StateVectors`.
"""
function oe_to_sv(oe::OrbitalElements; mu::Float64 = 398600.4418)
        # unpack orbital elements
        a     = oe.a
        e     = oe.e
        i_deg = oe.i_deg
        Ω_deg = oe.Ω_deg
        ω_deg = oe.ω_deg
        ν_deg = oe.ν_deg

        # convert OEs from degrees to radians
        i = deg2rad(i_deg)
        Ω = deg2rad(Ω_deg)
        ω = deg2rad(ω_deg)
        ν = deg2rad(ν_deg)

        # compute radius
        r_mag = a * (1 - e^2) / (1 + e*cos(ν))

        # perifocal position
        r_pf = [
                r_mag*cos(ν);
                r_mag*sin(ν);
                0.0
        ]

        # semi-latus rectum
        p = a * (1 - e^2)

        # perifocal velocity
        v_pf = [
                -sqrt(mu/p)*sin(ν);
                sqrt(mu/p)*(e + cos(ν));
                0.0
        ]

        # rotate from perifocal frame to geocentric equatorial frame
        R = [
                cos(Ω)*cos(ω) - sin(Ω)*sin(ω)*cos(i)  -cos(Ω)*sin(ω) - sin(Ω)*cos(ω)*cos(i)   sin(Ω)*sin(i);
                sin(Ω)*cos(ω) + cos(Ω)*sin(ω)*cos(i)  -sin(Ω)*sin(ω) + cos(Ω)*cos(ω)*cos(i)  -cos(Ω)*sin(i);
                sin(ω)*sin(i)                          cos(ω)*sin(i)                          cos(i)
        ]
```

```julia
    # rotate to inertial frame
    r = R * r_pf
    v = R * v_pf

    return StateVectors(r, v)
end

"""
    solve_kepler(M::Float64, e::Float64; tol=1e-6, max_iter=1000)

    Solves Kepler’s equation for the eccentric anomaly E:
    E - e*sin(E) = M
    using the Newton–Raphson method.
    - M : Mean anomaly (radians)
    - e : Eccentricity
    - tol : Convergence tolerance
    - max_iter : Maximum number of iterations

    Returns the eccentric anomaly E (in radians).
"""
function solve_kepler(M::Float64, e::Float64; tol::Float64 = 1e-6, max_iter::Int = 1000)
    # normalize M to [-pi, pi]
    M = mod(M, 2*pi)
    if M > pi
        M -= 2*pi
    end

    # initial guess for E
    E = M
    for iter in 1:max_iter
        f = E - e*sin(E) - M
        fp = 1 - e*cos(E)
        E_new = E - f/fp
        if abs(E_new - E) < tol
            return E_new
        end
        E = E_new
    end
    error("Kepler's equation did not converge after $max_iter iterations")
end


"""
    update_orbital_elements(oe::OrbitalElements, dt::Float64; mu=398600.4418)

    Updates the orbital elements after a time increment dt (in seconds). It assumes
    the initial elements are given at time t = 0. The function updates only the true anomaly,
    as the other elements remain constant for a Keplerian orbit.
    - dt : Time elapsed in seconds

    Returns a new `OrbitalElements` instance with the updated true anomaly.
"""
function update_orbital_elements(oe::OrbitalElements, dt::Float64; mu::Float64 = 398600.4418)
    e = oe.e
    ν0 = deg2rad(oe.ν_deg)

    # initial eccentric anomaly E₀ from the true anomaly
    # tan(ν/2) = sqrt((1+e)/(1-e)) * tan(E/2)
    E0 = 2 * atan( sqrt((1 - e)/(1 + e)) * tan(ν0/2) )

    # mean anomaly at epoch
    M0 = E0 - e*sin(E0)

    # compute the mean motion n (rad/s)
    n = sqrt(mu / oe.a^3)

    # new mean anomaly after time dt
    M = M0 + n*dt

    # new eccentric anomaly E
    E = solve_kepler(M, e)

    # new true anomaly from E
    # ν = 2 * atan( sqrt((1+e)/(1-e)) * tan(E/2) )
    ν = 2 * atan( sqrt((1 + e)/(1 - e)) * tan(E/2) )
    ν = mod(ν, 2*pi)  # ensure ν is in the range [0, 2π)
```

```julia
        # new OrbitalElements with updated ν
        return OrbitalElements(oe.a, oe.e, oe.i_deg, oe.Ω_deg, oe.ω_deg, rad2deg(ν))
end

"""
        kepler_predict(oe::OrbitalElements, dt::Float64; mu=398600.4418)

        Given the orbital elements at t = 0, predicts the state vectors (position and velocity)
        after a time increment dt (in seconds). It updates the true anomaly by solving Kepler's equation.

        Returns a tuple (sv_new, oe_new) where:
        - sv_new: StateVectors struct containing the predicted position and velocity vectors.
        - oe_new: The updated OrbitalElements with the new true anomaly.
"""
function kepler_predict(oe::OrbitalElements, dt::Float64; mu::Float64 = 398600.4418)
        oe_new = update_orbital_elements(oe, dt, mu=mu)
        sv_new = oe_to_sv(oe_new, mu=mu)
        return sv_new, oe_new
end

export OrbitalElements, StateVectors, oe_to_sv, kepler_predict

end
```

```
Main.var"##WeaveSandBox#231".SpaceFlightDynamics
```

Published from sfd.jl using Weave.jl v0.10.12 on 2025-04-09.