```
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
```