

Multi-Spacecraft Concept and Autonomy Tool (MuSCAT) Version 2

Dr. Saptarshi Bandyopadhyay,
Jet Propulsion Laboratory, California Institute of Technology
Email: `saptarshi.bandyopadhyay@jpl.nasa.gov`

March 13, 2025

Contents

1	Introduction	3
1.1	Coding Conventions	5
1.2	Coding Tips for Multi-Mission Usage	5
1.3	Acknowledgment	6
2	Environment Classes	7
2.1	Storage	7
2.2	True_Gravity_Gradient	17
2.3	True_Ground_Station	20
2.4	True_GS_Radio_Antenna	28
2.5	True_Solar_System	34
2.6	True_SRP	43
2.7	True_Stars	49
2.8	True_Target_SPICE	52
2.9	True_Time	64
3	SC Physics Based Simulation Layer Classes	71
3.1	True_SC_ADC	71
3.2	True_SC_Body	80
3.3	True_SC_Data_Handling	91
3.4	True_SC_Navigation	102
3.5	True_SC_Power	112
4	SC Sensors and Actuators Classes	123
4.1	True_SC_Battery	123
4.2	True_SC_Camera	128
4.3	True_SC_Communication_Link	138
4.4	True_SC_Chemical_Thruster	146
4.5	True_SC_Fuel_Tank	160
4.6	True_SC_Generic_Sensor	165
4.7	True_SC_IMU	170
4.8	True_SC_Micro_Thruster	176
4.9	True_SC_Onboard_Computer	183
4.10	True_SC_Onboard_Clock	188
4.11	True_SC_Onboard_Memory	193
4.12	True_SC_Radio_Antenna	197
4.13	True_SC_Reaction_Wheel	204
4.14	True_SC_Science_Processor	213

4.15	True_SC_Science_Radar	219
4.16	True_SC_Solar_Panel	230
4.17	True_SC_Star_Tracker	238
4.18	True_SC_Sun_Sensor	245
5	SC System-Level and Functional-Level Autonomy Software Layer Classes	252
5.1	Software_SC_Communication	252
5.2	Software_SC_Control_Attitude	262
5.3	Software_SC_Control_Orbit	298
5.4	Software_SC_Data_Handling	320
5.5	Software_SC_Estimate_Attitude	325
5.6	Software_SC_Estimate_Orbit	332
5.7	Software_SC_Executive	341
5.8	Software_SC_Power	351
6	Main File	356
6.1	main_v3	356
7	Mission Classes	366
7.1	Mission_DART	366
	Bibliography	388

Chapter 1

Introduction

Multi-Spacecraft Concept and Autonomy Tool (MuSCAT) open-source simulation software offers an integrated platform for conducting low-fidelity simulations of single/multiple cruising/orbiting spacecraft mission concepts and test of autonomy algorithms. MuSCAT encompasses various spacecraft subsystems such as (i) Navigation, (ii) Attitude Determination and Control, (iii) Power Management, (iv) Data Handling, (v) Communication (including direct-to-Earth and inter-spacecraft), and (vi) Scientific Instruments. It provides mission designers with a means to quantitatively verify if the mission concept meets level-1 science functional requirements.

[1] provides a detailed description of the software architecture and use cases of MuSCAT.

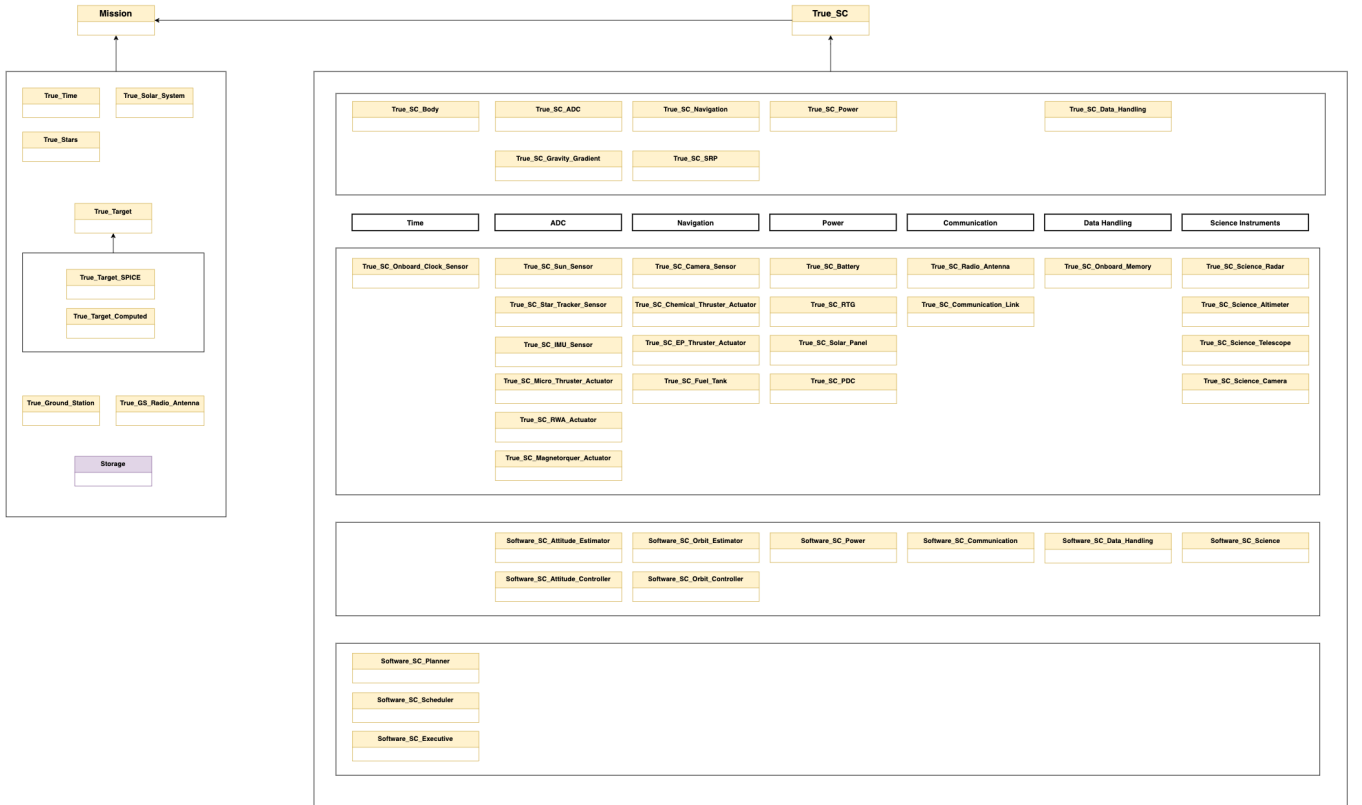


Figure 1.1: Diagram showing all the classes that are inside MuSCAT.

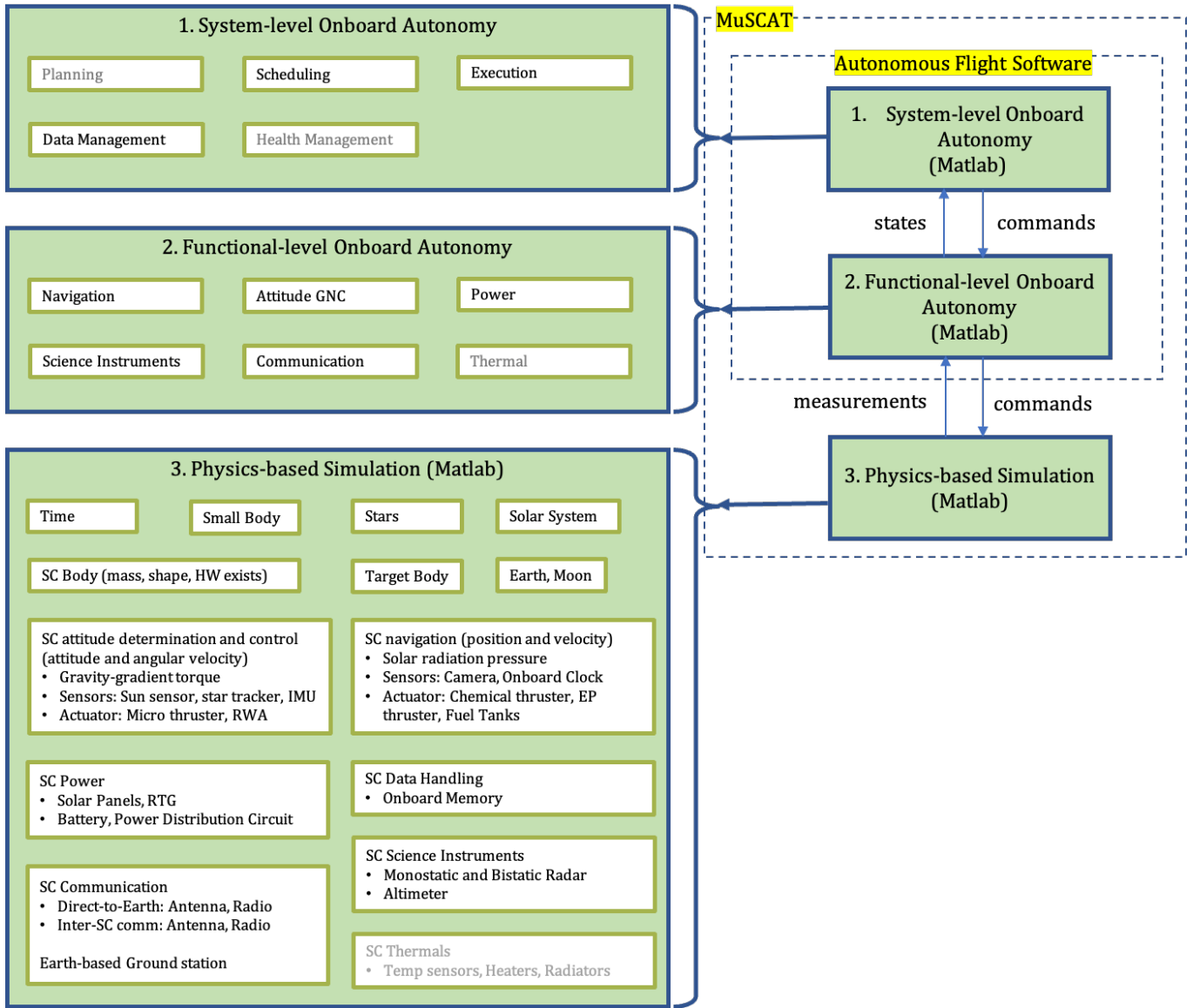


Figure 1.2: Block diagram that shows the different components inside the MuSCAT simulator. The blocks in grey are currently under development.

1.1 Coding Conventions

- Items marked with `#` are not initialized, and computed internally
- In naming functions,
`update` is used when the function is called often (every time step),
`compute` is used when it is called rarely (or only once).
- All functions are prefixed with `func_`, and colored green.
- All classes within the physics-based simulation represent real objects like Solar System or small body, and are prefixed with `True_`, and those that belong to the spacecraft are prefixed with `True_SC_`.
- All classes within the autonomous flight software that only exist within the spacecraft's onboard computer are prefixed with `Software_SC_`.

1.2 Coding Tips for Multi-Mission Usage

- Within each class, use `data` to track all mission-specific variables.
- If a component uses different power in different spacecraft modes, use `data.instantaneous_power_consumed_per_SC_mode` to track that.

1.3 Acknowledgment

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. ©2025 California Institute of Technology. Government sponsorship acknowledged.

The authors also thank the following people for their invaluable time and technical expertise in constructing this code base:

- Thibault M Wartel from Institut supérieur de l’aéronautique et de l’espace (ISAE-SUPAERO), France
- Cesc Casanovas Gasso from International Space University, France
- Ian Aenishanslin from Institut polytechnique des sciences avancées (IPSA), France
- Carmine Buonagura from Politecnico di Milano, Italy
- Chris Agia from Stanford, USA
- Kazu Echigo from University of Washington, USA
- Minduli C Wijayatunga from University of Auckland, New Zealand

Chapter 2

Environment Classes

2.1 Storage

Table of Contents

Class: Storage	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Other Useful Variables	2
Methods	2
[] Methods: Constructor	3
[] Methods: Update Storage Flag	5
[] Methods: Update Storage Flag Attitude	6
[] Methods: Update Real-Time Plot	6
[] Methods: Visualize Simulation Data	7

Class: Storage

Helps store all the data from the simulation

```
classdef Storage < handle
```

Properties

properties

[] Properties: Initialized Variables

```
time_step_storage % [sec] : Storage time step
time_step_storage_attitude % [sec] : Storage time step for attitude
dynamics loop (Optional)

flag_visualize_SC_attitude_orbit_during_sim % [Boolean] : 1 = Shows
the attitude and position during simulation (Optional)
wait_time_visualize_SC_attitude_orbit_during_sim % [sec] (Optional)
flag_visualize_past_SC_orbit_during_sim % [Boolean] : 1 = Shows the
entire orbit from the start of sim (Optional)
```

[] Properties: Variables Computed Internally

```
time_prev_storage % [sec] : Previous time when variables were stored #
num_storage_steps % [integer] : Number of storage variables #
flag_store_this_time_step % [Boolean] : 1 = Store, else dont store #
k_storage % [integer] : Storage counter variable #

time_prev_storage_attitude % [sec] : Previous time when attitude
variables were stored #
num_storage_steps_attitude % [integer] : Number of storage variables
for Attitude Dynamics Loop #
```

```

        flag_store_this_time_step_attitude % [Boolean] : 1 = Store, else dont
store, for Attitude Dynamics Loop #
        k_storage_attitude % [integer] : Storage counter variable, for
Attitude Dynamics Loop #

        prev_time_visualize_SC_attitude_orbit_during_sim % [sec]

        % Real-time plotting variables
        flag_realtime_plotting % [Boolean] : 1 = Enable real-time performance
plotting
        realtime_plot_handle % Handle to the real-time plot figure
        realtime_plot_last_update % [sec] : Time of last real-time plot update
        realtime_plot_update_interval % [sec] : Minimum time between real-time
plot updates
        realtime_plot_subhandles % Cell array of subplot handles for real-time
plotting

        % Initialize real-time visualization settings
        last_viz_update_time % Time of last visualization update
        viz_update_interval % Update visualization every 50 simulation
seconds (reduced from 100)

```

[] Properties: Other Useful Variables

```

        numerical_accuracy_factor % [float <= 1, but limit -> 1] : Used to
take care of issues arising due to numerical accuracy of integer computations
#

        plot_parameters % Parameters used for plotting
        % - color_array
        % - marker_array
        % - standard_font_size
        % - standard_font_type
        % - title_font_size
        % - flag_save_plots % [Boolean] 1: Save them (takes little time), 0:
Doesnt save them
        % - flag_save_video % [Boolean] 1: Save them (takes a lot more time),
0: Doesnt save them
        % - quiver_auto_scale_factor % [float] : scale factor used for quiver3

        output_folder % [Boolean] Folder to store all outputs

        flag_stop_sim % flag to stop simulation

        last_mode % Cell array to store the last mode of each spacecraft

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```
function obj = Storage(init_data, mission)

    if init_data.time_step_storage == 0
        % Use 0 to use the mission.true_time.time_step value
        obj.time_step_storage = mission.true_time.time_step; % [sec]

    else
        obj.time_step_storage = init_data.time_step_storage; % [sec]
    end

    obj.flag_store_this_time_step = 1;
    obj.time_prev_storage = mission.true_time.time; % [sec]
    obj.num_storage_steps = ceil( (mission.true_time.t_final -
mission.true_time.t_initial)/obj.time_step_storage ) + 1;
    obj.k_storage = 1;

    obj.last_viz_update_time = -inf; % Time of last visualization
update
    obj.viz_update_interval = 1000; % Update visualization every 50
simulation seconds (reduced from 100)

    obj.flag_stop_sim = 0;

    if isfield(init_data, 'time_step_storage_attitude')
        % time_step_storage_attitude has been specified

        if init_data.time_step_storage_attitude == 0
            % Use 0 to use the mission.true_time.time_step_attitude
value
            obj.time_step_storage_attitude =
mission.true_time.time_step_attitude; % [sec]

        else
            obj.time_step_storage_attitude =
init_data.time_step_storage_attitude; % [sec]
        end

    else
        obj.time_step_storage_attitude = obj.time_step_storage; %
[sec]
    end

    obj.flag_store_this_time_step_attitude = 1;
    obj.time_prev_storage_attitude = mission.true_time.time; % [sec]
    obj.num_storage_steps_attitude = ceil( (mission.true_time.t_final
- mission.true_time.t_initial)/obj.time_step_storage_attitude ) + 1;
    obj.k_storage_attitude = 1;
```

```

obj.numerical_accuracy_factor = 0.99;

% Set plot_parameters
obj.plot_parameters = [];
obj.plot_parameters.color_array = ['b' 'r' 'g' 'c' 'y' 'm' 'k'];
% (Additional colors using rgb.m function from https://
www.mathworks.com/matlabcentral/fileexchange/24497-rgb-triple-of-color-name-
version-2)

obj.plot_parameters.marker_array =
['o' 's' 'd' '^' 'v' '>' '<' 'p' 'h' '+'];
obj.plot_parameters.standard_font_size = 20;
obj.plot_parameters.standard_font_type = 'Times New Roman';
obj.plot_parameters.title_font_size = 40;

if isfield(init_data, 'flag_save_plots')
    obj.plot_parameters.flag_save_plots =
init_data.flag_save_plots;
else
    obj.plot_parameters.flag_save_plots = 1; % [Boolean] 1: Save
them (takes little time), 0: Doesnt save them
end

if isfield(init_data, 'flag_save_video')
    obj.plot_parameters.flag_save_video =
init_data.flag_save_video;
else
    obj.plot_parameters.flag_save_video = 0; % [Boolean] 1: Save
them (takes a lot more time), 0: Doesnt save them
end

if
isfield(init_data, 'flag_visualize_SC_attitude_orbit_during_sim')
    obj.flag_visualize_SC_attitude_orbit_during_sim =
init_data.flag_visualize_SC_attitude_orbit_during_sim;
else
    obj.flag_visualize_SC_attitude_orbit_during_sim = 1;
end

if
isfield(init_data, 'wait_time_visualize_SC_attitude_orbit_during_sim')
    obj.wait_time_visualize_SC_attitude_orbit_during_sim =
init_data.wait_time_visualize_SC_attitude_orbit_during_sim; % [sec]
else
    obj.wait_time_visualize_SC_attitude_orbit_during_sim = 0; %
[sec]
end
obj.prev_time_visualize_SC_attitude_orbit_during_sim = -inf;

if isfield(init_data, 'flag_visualize_past_SC_orbit_during_sim')
    obj.flag_visualize_past_SC_orbit_during_sim =
init_data.flag_visualize_past_SC_orbit_during_sim;
else

```

```

        obj.flag_visualize_past_SC_orbit_during_sim = 1;
    end

    % Initialize real-time plotting variables
    if isfield(init_data, 'flag_realtime_plotting')
        obj.flag_realtime_plotting = init_data.flag_realtime_plotting;
    else
        obj.flag_realtime_plotting = 1;
    end

    if isfield(init_data, 'quiver_auto_scale_factor')
        obj.plot_parameters.quiver_auto_scale_factor =
init_data.quiver_auto_scale_factor;
    else
        obj.plot_parameters.quiver_auto_scale_factor = 0.1;
    end

    % Output Folder
    obj.output_folder = ['../Output/', mission.name, '_',
char(datetime("now", "Format", "yyyy-MM-dd-HH'h'mm'm'ss's'")), ' (SimTime =
', char(string(mission.true_time.t_final/86400)), ' days)'];
    %obj.output_folder = ['../Output/', mission.name, '_',
char(datetime("now", "Format", "yyyy-MM-dd-HH'h'mm'm'ss's'")), ' (SimTime =
',char(string(mission.true_time.t_final/86400)), ' days)'];
    mkdir(obj.output_folder)

    % Store video of func_visualize_SC_attitude_orbit_during_sim
    if (obj.plot_parameters.flag_save_video == 1) &&
(obj.flag_visualize_SC_attitude_orbit_during_sim == 1)
        obj.plot_parameters.video_filename = [obj.output_folder,
mission.name, '_Attitude_Orbit.mp4'];
        obj.plot_parameters.myVideo =
VideoWriter(obj.plot_parameters.video_filename, 'MPEG-4');
        obj.plot_parameters.myVideo.FrameRate = 30; % Default 30
        obj.plot_parameters.myVideo.Quality = 100; % Default 75
        open(obj.plot_parameters.myVideo);
    end

end
end

```

[] Methods: Update Storage Flag

Set the flag_store_this_time_step after sufficient time

```

function obj = func_update_storage_flag(obj, mission)

    % Reset flags
    obj.flag_store_this_time_step = 0;

    if (mission.true_time.time - obj.time_prev_storage) >=
(obj.time_step_storage * obj.numerical_accuracy_factor)
        obj.flag_store_this_time_step = 1;
    end

```

```

end

if mission.true_time.k == mission.true_time.num_time_steps
    obj.flag_store_this_time_step = 1;
end

if obj.flag_store_this_time_step == 1
    obj.time_prev_storage = mission.true_time.time; % [sec]
    obj.k_storage = obj.k_storage + 1;
end

end

```

[] Methods: Update Storage Flag Attitude

Set the flag_store_this_time_step_attitude after sufficient time

```

function obj = func_update_storage_flag_attitude(obj, mission)

    % Reset flags
    obj.flag_store_this_time_step_attitude = 0;

    if (mission.true_time.time_attitude -
        obj.time_prev_storage_attitude) >= (obj.time_step_storage_attitude *
        obj.numerical_accuracy_factor)
        obj.flag_store_this_time_step_attitude = 1;
    end

    if (mission.true_time.k == mission.true_time.num_time_steps) &&
        (mission.true_time.k_attitude == mission.true_time.num_time_steps_attitude)
        obj.flag_store_this_time_step_attitude = 1;
    end

    if obj.flag_store_this_time_step_attitude == 1
        obj.time_prev_storage_attitude =
mission.true_time.time_attitude; % [sec]
        obj.k_storage_attitude = obj.k_storage_attitude + 1;
    end

end

```

[] Methods: Update Real-Time Plot

Update the real-time performance plot

```

function obj = func_update_realtime_plot(obj, mission)
    if obj.flag_realtime_plotting
        % Initialize last_mode property if it doesn't exist
        if ~isfield(obj, 'last_mode')
            obj.last_mode = {};
            for i_SC = 1:mission.num_SC
                obj.last_mode{i_SC} =
mission.true_SC{i_SC}.software_SC_executive.this_sc_mode;
            end
        end
    end
end

```

```

        end
    end

    % Check for mode changes in any spacecraft
    mode_changed = false;
    for i_SC = 1:mission.num_SC
        current_mode =
mission.true_SC{i_SC}.software_SC_executive.this_sc_mode;
        if ~strcmp(current_mode, obj.last_mode{i_SC})
            mode_changed = true;
            obj.last_mode{i_SC} = current_mode;
        end
    end

    % Update visualization if time interval has elapsed OR mode
has changed
    if (mission.true_time.time - obj.last_viz_update_time >=
obj.viz_update_interval) || mode_changed
        % Update visualization with attitude rotation and store
the time

        func_visualize_SC(obj, mission, true);
        obj.last_viz_update_time = mission.true_time.time;

        % Force display update without blocking execution
drawnow limitrate;
    end
end
end
end

```

[] Methods: Visualize Simulation Data

Visualize all simulation data

```

function obj = func_visualize_simulation_data(obj, mission)
    % First, ensure we're not keeping any unnecessary figures open
close all
    mission.flag_stop_sim = 1;

    % Close all video files
    if (obj.plot_parameters.flag_save_video == 1)
        if isfield(mission.storage.plot_parameters, 'myVideo')
            close(obj.plot_parameters.myVideo);
        end

        for i_SC = 1:1:mission.num_SC
            % Close Camera video files
            for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_camera
                if
isfield(mission.true_SC{i_SC}.true_SC_camera{i_HW}.data, 'myVideo')
close(mission.true_SC{i_SC}.true_SC_camera{i_HW}.data.myVideo);
                end
            end
        end
    end
end

```

```

        end

        % Close Radar video files
        for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_science_radar
            if
isfield(mission.true_SC{i_SC}.true_SC_science_radar{i_HW}.data, 'myVideo')
close(mission.true_SC{i_SC}.true_SC_science_radar{i_HW}.data.myVideo);
            end
        end
    end
end

    % Memory optimization: create a function to run between plots to
cleanup memory
    function cleanup_memory()
        drawnow; % Flush graphics queue and allow MATLAB to reclaim
memory
        % More can be added here in case of need !
    end

    % Process one spacecraft at a time to limit memory usage
    for i_SC = 1:1:mission.num_SC
        % Basic plots for all spacecraft (minimal memory use)
        % Orbit Vizualization (shared plot)

        if i_SC == 1
            % Shared for all spacecraft
            fprintf('Plotting orbit visualization...\n');
            func_plot_orbit_visualization(mission);
            cleanup_memory();
        end

        % Spacecraft-specific plots
        fprintf('Plotting SC%d orbit estimator...\n', i_SC);
        func_plot_orbit_estimator(mission, i_SC);
        cleanup_memory();

        fprintf('Plotting SC%d orbital control performance...\n',
i_SC);
        func_plot_orbital_control_performance(mission, i_SC);
        cleanup_memory();

        fprintf('Plotting SC%d attitude visualization...\n', i_SC);
        func_plot_attitude_visualization(mission, i_SC);
        cleanup_memory();

        fprintf('Plotting SC%d attitude actuator performance...\n',
i_SC);
        func_plot_attitude_actuator_performance(mission, i_SC);
        cleanup_memory();

        fprintf('Plotting SC%d power visualization...\n', i_SC);

```

2.2 True_Gravity_Gradient

```

classdef True_Gravity_Gradient < handle
    % True_SC_Gravity_Gradient
    properties
        disturbance_torque_G2 % [Nm]
        disturbance_force_G2 % [N] Force induced by gravity gradient
        enable_G2 % [boolean]
        main_body % [String] Main body around which we orbit. Has to fit with
mission.true_solar_system.all_SS_body_data
        store
    end

    methods

        function obj = True_Gravity_Gradient(init_data, mission, i_SC)

            % Optional parameters
            if isfield(init_data, 'main_body')
                obj.main_body = init_data.main_body;
            else
                obj.main_body = 'Earth';
            end

            obj.disturbance_torque_G2 = zeros(3,1);
            obj.disturbance_force_G2 = zeros(3,1);
            obj.enable_G2 = init_data.enable_G2;

            % Calculate GG before first iteration of ADL
            obj.func_update_disurbance_torque_G2(mission, i_SC);

            % Initialize storage
            obj.store.disturbance_torque_G2 =
zeros(mission.storage.num_storage_steps, 3);
            obj.store.disturbance_force_G2 =
zeros(mission.storage.num_storage_steps, 3);

        end
    end
end

```

[] Methods: Store

```

function obj = func_update_true_gravity_gradient_store(obj, mission)
    if mission.storage.flag_store_this_time_step == 1
        obj.store.disturbance_torque_G2(mission.storage.k_storage,:) =
obj.disturbance_torque_G2';
        obj.store.disturbance_force_G2(mission.storage.k_storage,:) =
obj.disturbance_force_G2';
    end
end

```

[] Methods: Main Disturbance torque

```

function obj = func_update_disurbance_torque_G2(obj, mission, i_SC)

```

```

    % Reset disturbance torque and force
    obj.disturbance_torque_G2 = zeros(3,1);
    obj.disturbance_force_G2 = zeros(3,1);

    if obj.enable_G2 == 1
        % Gravity direction from SB
        Rc =
1e3*(mission.true_SC{i_SC}.true_SC_navigation.position_relative_target);
        %[m]
        Rc_sc = (mission.true_SC{i_SC}.true_SC_adc.rotation_matrix') *
Rc';
        %[m]

        % Gravity gradient torque [Nm]
        obj.disturbance_torque_G2 = (3*(mission.true_target{1,
1}.mu * 1e9)/(norm(Rc_sc)^5))*cross(Rc_sc,
mission.true_SC{i_SC}.true_SC_body.total_MI *Rc_sc);

        mission.true_SC{i_SC}.true_SC_adc.disturbance_torque
= mission.true_SC{i_SC}.true_SC_adc.disturbance_torque +
obj.disturbance_torque_G2;

        % Gravity gradient force [N]
        % Using the standard gravity gradient force equation
        mu = mission.true_target{1, 1}.mu * 1e9; % Convert to m^3/s^2
        R = norm(Rc_sc);
        unit_Rc = Rc_sc/R;

        % Calculate gravity gradient force
        obj.disturbance_force_G2 = -
mu*mission.true_SC{i_SC}.true_SC_body.total_mass/(R^2) * unit_Rc;
    end

    % Update storage after calculation
    obj.func_update_true_gravity_gradient_store(mission);
end

end
end

```

Published with MATLAB® R2022a

2.3 True_Ground_Station

Table of Contents

Class: True_Ground_Station	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Initialize list_HW_data_transmitted	3
[] Methods: Initialize list_HW_data_received	3
[] Methods: Initialize Store	4
[] Methods: Store	4
[] Methods: Main	5
[] Methods: Update Instantaneous Data transmitted	5
[] Methods: Update Instantaneous Data received	6

Class: True_Ground_Station

Tracks the data sent and recieved by Ground Station

```
classdef True_Ground_Station < handle
```

Properties

properties

[] Properties: Initialized Variables

```
num_GS_radio_antenna % [integer] Number of GS Radio Antenna
```

[] Properties: Variables Computed Internally

```
instantaneous_data_transmitted % [kb] Data transmitted by GS over  
mission.true_time.time_step sec
```

```
total_data_transmitted % [kb] Data transmitted by GS over time
```

```
instantaneous_data_received % [kb] Data received by GS over  
mission.true_time.time_step sec
```

```
total_data_received % [kb] Data received by GS over time
```

```
list_HW_data_transmitted % List of HW that transmitted data
```

```
array_HW_data_transmitted % [kb] Total data transmitted by this HW
```

```
list_HW_data_received % List of HW that received data
```

```
array_HW_data_received % [kb] Total data received by this HW
```

```
warning_counter % [integer] Counter stops the warning after 10  
displays
```

```
data % Other useful data
```

[] Properties: Storage Variables

```
store  
  
end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_Ground_Station(init_data, mission)  
  
    obj.num_GS_radio_antenna = init_data.num_GS_radio_antenna;  
  
    obj.instantaneous_data_transmitted = 0; % [kb]  
    obj.total_data_transmitted = 0; % [kb]  
    obj.instantaneous_data_received = 0; % [kb]  
    obj.total_data_received = 0; % [kb]  
  
    obj.list_HW_data_transmitted = [];  
    obj.array_HW_data_transmitted = [];  
    obj.list_HW_data_received = [];  
    obj.array_HW_data_received = [];  
  
    obj.warning_counter = 0;  
  
    if isfield(init_data, 'data')  
        obj.data = init_data.data;  
    else  
        obj.data = [];  
    end  
  
    % Initialize Variables to store  
    obj.store = [];  
  
    obj.store.instantaneous_data_transmitted =  
zeros(mission.storage.num_storage_steps,  
length(obj.instantaneous_data_transmitted));  
    obj.store.instantaneous_data_received =  
zeros(mission.storage.num_storage_steps,  
length(obj.instantaneous_data_received));
```

```

        obj.store.total_data_transmitted =
zeros(mission.storage.num_storage_steps, length(obj.total_data_transmitted));
        obj.store.total_data_received =
zeros(mission.storage.num_storage_steps, length(obj.total_data_received));

    end

```

[] Methods: Initialize list_HW_data_transmitted

Initialize list_HW_data_transmitted for HW and Classes

```

function obj = func_initialize_list_HW_data_transmitted(obj,
equipment, mission)

    this_name = equipment.name;
    flag_name_exisits = 0;

    for i = 1:length(obj.list_HW_data_transmitted)
        if strcmp( obj.list_HW_data_transmitted{i}, this_name )
            flag_name_exisits = 1;
        end
    end

    if flag_name_exisits == 0
        i = length(obj.list_HW_data_transmitted);
        obj.list_HW_data_transmitted{i+1} = this_name;

        if isprop(equipment, 'instantaneous_data_rate_transmitted')
            this_instantaneous_data_transmitted
= (equipment.instantaneous_data_rate_transmitted *
mission.true_time.time_step); % [kb]
        elseif
isprop(equipment, 'instantaneous_data_transmitted_per_sample')
            this_instantaneous_data_transmitted =
equipment.instantaneous_data_transmitted_per_sample; % [kb]
        else
            error('Data transmitted incorrect!')
        end

        obj.array_HW_data_transmitted(1,i+1) =
this_instantaneous_data_transmitted; % [kb]
    end

end

```

[] Methods: Initialize list_HW_data_received

Initialize list_HW_data_received for HW and Classes

```

function obj = func_initialize_list_HW_data_received(obj, equipment,
mission)

    this_name = equipment.name;

```

```

        flag_name_exisits = 0;

        for i = 1:1:length(obj.list_HW_data_received)
            if strcmp( obj.list_HW_data_received{i}, this_name )
                flag_name_exisits = 1;
            end
        end

        if flag_name_exisits == 0
            i = length(obj.list_HW_data_received);
            obj.list_HW_data_received{i+1} = this_name;

            if isprop(equipment, 'instantaneous_data_rate_received')
                this_instantaneous_data_received =
(equipment.instantaneous_data_rate_received * mission.true_time.time_step); %
[kb]
            elseif
isprop(equipment, 'instantaneous_data_received_per_sample')
                this_instantaneous_data_received =
equipment.instantaneous_data_received_per_sample; % [kb]
            else
                error('Data received incorrect!')
            end

            obj.array_HW_data_received(1,i+1) =
this_instantaneous_data_received; % [kb]
        end

    end
end

```

[] Methods: Initialize Store

Initialize store of array_HW_data_transmitted and array_HW_data_received

```

function obj = func_initialize_store_HW_data_transmitted_received(obj,
mission)

    obj.store.list_HW_data_transmitted = obj.list_HW_data_transmitted;
    obj.store.list_HW_data_received = obj.list_HW_data_received;

    obj.store.array_HW_data_transmitted =
zeros(mission.storage.num_storage_steps,
length(obj.array_HW_data_transmitted));
    obj.store.array_HW_data_received =
zeros(mission.storage.num_storage_steps, length(obj.array_HW_data_received));

    obj = func_update_true_ground_station_store(obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_ground_station_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1

obj.store.instantaneous_data_transmitted(mission.storage.k_storage,:) =
obj.instantaneous_data_transmitted; % [kb]

obj.store.instantaneous_data_received(mission.storage.k_storage,:) =
obj.instantaneous_data_received; % [kb]

        obj.store.total_data_transmitted(mission.storage.k_storage,:) =
= obj.total_data_transmitted; % [kb]
        obj.store.total_data_received(mission.storage.k_storage,:) =
obj.total_data_received; % [kb]

obj.store.array_HW_data_transmitted(mission.storage.k_storage,:) =
obj.array_HW_data_transmitted; % [kb]
        obj.store.array_HW_data_received(mission.storage.k_storage,:) =
= obj.array_HW_data_received; % [kb]
    end

end

```

[] Methods: Main

Main Ground Station code

```

function obj = func_main_true_ground_station(obj, mission, i_SC)

    obj.total_data_transmitted = obj.total_data_transmitted +
obj.instantaneous_data_transmitted; % [kb]
    obj.total_data_received = obj.total_data_received +
obj.instantaneous_data_received; % [kb]

    % Store
    obj = func_update_true_ground_station_store(obj, mission);

    % Reset All Variables
    obj.instantaneous_data_transmitted = 0; % [kb]
    obj.instantaneous_data_received = 0; % [kb]

end

```

[] Methods: Update Instantaneous Data trans- mitted

Updates instantaneous_data_transmitted by all HW and Classes

```

function obj = func_update_instantaneous_data_transmitted(obj,
equipment, mission)

```

```

        if isprop(equipment, 'instantaneous_data_rate_transmitted')
            this_instantaneous_data_transmitted
= (equipment.instantaneous_data_rate_transmitted *
mission.true_time.time_step); % [kb]
        elseif
isprop(equipment, 'instantaneous_data_transmitted_per_sample')
            this_instantaneous_data_transmitted =
equipment.instantaneous_data_transmitted_per_sample; % [kb]
        else
            error('Data transmitted incorrect!')
        end

        obj.instantaneous_data_transmitted =
obj.instantaneous_data_transmitted + this_instantaneous_data_transmitted; %
[kb]

        this_name = equipment.name;
        flag_name_exisits = 0;
        this_idx = 0;

        for i = 1:length(obj.list_HW_data_transmitted)
            if strcmp( obj.list_HW_data_transmitted{i}, this_name )
                flag_name_exisits = 1;
                this_idx = i;
            end
        end

        if flag_name_exisits == 0
            error('HW not found!')
        else
            obj.array_HW_data_transmitted(1,this_idx)
= obj.array_HW_data_transmitted(1,this_idx) +
this_instantaneous_data_transmitted; % [kb]
        end
    end
end

```

[] Methods: Update Instantaneous Data received

Updates instantaneous_data_received by all HW and Classes

```

function obj = func_update_instantaneous_data_received(obj, equipment,
mission)

    if isprop(equipment, 'instantaneous_data_rate_received')
        this_instantaneous_data_received =
(equipment.instantaneous_data_rate_received * mission.true_time.time_step); %
[kb]
    elseif isprop(equipment, 'instantaneous_data_received_per_sample')
        this_instantaneous_data_received =
equipment.instantaneous_data_received_per_sample; % [kb]
    end
end

```

```

        else
            error('Data received incorrect!')
        end

        obj.instantaneous_data_received = obj.instantaneous_data_received
+ this_instantaneous_data_received; % [kb]

        this_name = equipment.name;
        flag_name_exisits = 0;
        this_idx = 0;

        for i = 1:1:length(obj.list_HW_data_received)
            if strcmp( obj.list_HW_data_received{i}, this_name )
                flag_name_exisits = 1;
                this_idx = i;
            end
        end

        if flag_name_exisits == 0
            error('HW not found!')
        else
            obj.array_HW_data_received(1,this_idx) =
obj.array_HW_data_received(1,this_idx) + this_instantaneous_data_received; %
[kb]
        end

    end

end

end

```

Published with MATLAB® R2022a

2.4 True_GS_Radio_Antenna

Table of Contents

Class: True_GS_Radio_Antenna	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	4

Class: True_GS_Radio_Antenna

Tracks the GS's Radio Antennas

```
classdef True_GS_Radio_Antenna < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
antenna_type % [string]
% - 'Dipole'
% - 'High Gain'

mode_true_GS_radio_antenna_selector % [string]
% - TX
% - RX

% Optional (only for Link Margin Calculations)

antenna_gain % [dB] gain of Earth receiver

noise_temperature % [K] temperature noise

beamwidth % [MHz] receiver beamwidth

energy_bit_required % [dB] Minimum energy bit required

line_loss % [dB] Loss due to pointing or others

coding_gain % [dB] Coding gain
```

[] Properties: Variables Computed Internally

```
name % [string] 'GS Radio Antenna i'
```

```

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

instantaneous_data_rate_transmitted % [kbps] : Data rate, in kilo bits
per sec (kbps) due to RX

instantaneous_data_rate_received % [kbps] : Data rate, in kilo bits
per sec (kbps) due to TX

data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_GS_Radio_Antenna(init_data, mission, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['GS Radio Antenna ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.flag_executive = 0;

    obj.antenna_type = init_data.antenna_type;

    obj.mode_true_GS_radio_antenna_selector =
init_data.mode_true_GS_radio_antenna_selector;

    if isfield(init_data, 'antenna_gain')

        obj.antenna_gain = init_data.antenna_gain; % [dB]
        obj.noise_temperature = init_data.noise_temperature; % [K]
        obj.beamwidth = init_data.beamwidth; % [MHz]
    end
end

```

```

        obj.energy_bit_required = init_data.energy_bit_required; %
[dB]
        obj.line_loss = init_data.line_loss; % [dB]
        obj.coding_gain = init_data.coding_gain; % [dB]

    end

    obj.instantaneous_data_rate_transmitted = 0; % [kbps]
    obj.instantaneous_data_rate_received = 0; % [kbps]

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    % Initialize Variables to store: flag_executive mode_TX_RX
    obj.store = [];

    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive)); %
[integer]
    obj.store.mode_TX_RX = zeros(mission.storage.num_storage_steps,
1); % [integer]
    obj.store.instantaneous_data_rate_transmitted =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_transmitted)); % [kbps]
    obj.store.instantaneous_data_rate_received =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_received)); % [kbps]

    % Update Storage
    obj = func_update_true_GS_radio_antenna_store(obj, mission);

    % Update Ground Station Class (Generated and Removed)

    func_initialize_list_HW_data_transmitted(mission.true_ground_station, obj,
mission);

    func_initialize_list_HW_data_received(mission.true_ground_station,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_GS_radio_antenna_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage,:) =
obj.flag_executive; % [integer]
    end

```

```

obj.store.instantaneous_data_rate_transmitted(mission.storage.k_storage,:) =
obj.instantaneous_data_rate_transmitted; % [kbps]

obj.store.instantaneous_data_rate_received(mission.storage.k_storage,:) =
obj.instantaneous_data_rate_received; % [kbps]

        switch obj.mode_true_GS_radio_antenna_selector
            case 'TX'
                obj.store.mode_TX_RX(mission.storage.k_storage,1) = 1;
            case 'RX'
                obj.store.mode_TX_RX(mission.storage.k_storage,1) = 2;
            otherwise
                error('Should not reach here!')
            end
        end
    end
end

```

[] Methods: Main

Update Camera

```

function obj = func_main_true_GS_radio_antenna(obj, mission)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % TX or RX Data

        % Update SC Data Handling Class (Generated and Removed)

        func_update_instantaneous_data_transmitted(mission.true_ground_station, obj,
mission);

        func_update_instantaneous_data_received(mission.true_ground_station, obj,
mission);

    else
        % Do nothing
    end

    % Update Storage
    obj = func_update_true_GS_radio_antenna_store(obj, mission);

    % Reset All Variables
    obj.flag_executive = 0;
    obj.instantaneous_data_rate_transmitted = 0; % [kbps]
    obj.instantaneous_data_rate_received = 0; % [kbps]

end

end

```

end

Published with MATLAB® R2022a

2.5 True_Solar_System

Table of Contents

Class: True_Solar_System	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3
[] Methods: Load Data	4

Class: True_Solar_System

Tracks the position of Sun, Earth, Moon, etc. and other useful planetary bodies

```
classdef True_Solar_System < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
num_SS_body
```

[] Properties: Variables Computed Internally

```
SS_body % Data about selected SS body
% - name
% - radius [km]
% - mu [km^3 sec^-2]
% - mass [kg]
% - position [km] wrt Sun-centered J2000
% - velocity [km/sec] wrt Sun-centered J2000
% - position_array % [km] Position array wrt Sun-centered J2000,
corresponding to time array in mission.true_time.time_position_array
% - rgb_color [string] Used for plotting

solar_constant_AU % [W/m^2]
AU_distance % [km]
light_speed % [m/sec]
gravitational_constant % [km^3 kg^{#1} s^{#2}]

index_Sun % [integer] : Index of Sun
index_Earth % [integer] : Index of Earth
```

[] Properties: Storage Variables

```
store  
  
end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = TrueSolarSystem(init_data, mission)  
  
    obj.solar_constant_AU = 1361; % [W/m^2]  
    obj.AU_distance = 1.49598e8; % [km]  
    obj.light_speed = 299792458; % [m/sec]  
    obj.gravitational_constant = 6.67430e-20; % [km^3 kg^{#1} s^{#2}]  
  
    obj.num_SS_body = length(init_data.SS_body_names);  
    obj.SS_body = [];  
  
    all_SS_body_data = func_load_all_SS_body_data(obj);  
  
    for i = 1:1:obj.num_SS_body  
  
        this_SS_body_name =  
convertStringsToChars(init_data.SS_body_names(i));  
  
        for j = 1:1:length(all_SS_body_data)  
  
            if strcmp(this_SS_body_name, all_SS_body_data{j}.name)  
                obj.SS_body{i} = all_SS_body_data{j};  
            end  
  
        end  
  
        obj.SS_body{i}.position = zeros(1,3); % [km]  
        obj.SS_body{i}.velocity = zeros(1,3); % [km/sec]  
        obj.SS_body{i}.position_array =  
zeros( mission.true_time.num_time_steps_position_array,3); % [km]  
  
        if strcmp(obj.SS_body{i}.name, 'Sun')  
            obj.index_Sun = i;  
        end  
  
        if strcmp(obj.SS_body{i}.name, 'Earth')  
            obj.index_Earth = i;  
        end  
    end  
end
```

```

        end

    end

    % Update Position and Velocity
    cspice_furnsh(' ../../MuSCAT_Supporting_Files/SPICE/de440s.bsp')

    obj = func_main_true_solar_system(obj, mission);

    % Initialize Variables to store position and velocity of each body
    obj.store = [];

    for i = 1:1:obj.num_SS_body

        obj.store.SS_body{i}.name = obj.SS_body{i}.name;
        obj.store.SS_body{i}.position =
zeros(mission.storage.num_storage_steps, length(obj.SS_body{i}.position));
        obj.store.SS_body{i}.velocity =
zeros(mission.storage.num_storage_steps, length(obj.SS_body{i}.velocity));

    end

    obj = func_update_solar_system_store(obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_solar_system_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        for i = 1:1:obj.num_SS_body
            obj.store.SS_body{i}.position(mission.storage.k_storage,:)
= obj.SS_body{i}.position; % [km]
            obj.store.SS_body{i}.velocity(mission.storage.k_storage,:)
= obj.SS_body{i}.velocity; % [km/sec]
        end
    end

end

```

[] Methods: Main

% Function to update position of Sun, Earth, Moon, etc. ... given current time

```

function obj = func_main_true_solar_system(obj, mission)

    for i = 1:1:obj.num_SS_body

```

```

        body_pos_vel_this_time =
cspice_spkezr(obj.SS_body{i}.spice_name,mission.true_time.date,'J2000','NONE','SUN');

        obj.SS_body{i}.position = body_pos_vel_this_time(1:3,1)'; %
[km]
        obj.SS_body{i}.velocity = body_pos_vel_this_time(4:6,1)'; %
[km]

        body_pos_vel_array = (cspice_spkezr(obj.SS_body{i}.name,
mission.true_time.prev_date +
mission.true_time.time_position_array' , 'J2000', 'NONE', 'SUN'))';
        obj.SS_body{i}.position_array = body_pos_vel_array(:,1:3);

    end

    % Store
    obj = func_update_solar_system_store(obj, mission);

end

```

[] Methods: Load Data

Store all useful data about all SS bodies

```

function all_SS_body_data = func_load_all_SS_body_data(obj)

    all_SS_body_data = [];
    k = 0;

    % Sun's Data
    this_data = [];
    this_data.name = 'Sun';
    this_data.spice_name = '10'; % [string] : Body's SPICE Name
    this_data.radius = 6.95700e5; % [km]
    this_data.mu = 1.32712440018e11; % [km^3 sec^-2]
    this_data.mass = 1.9885e30; % [kg]
    this_data.rgb_color = 'Gold'; % [string]
    k = k + 1;
    all_SS_body_data{k} = this_data;

    % Mercury's Data
    this_data = [];
    this_data.name = 'Mercury';
    this_data.spice_name = '199'; % [string] : Body's SPICE Name
    this_data.radius = 2.4397e3; % [km] https://en.wikipedia.org/wiki/
    Mercury_(planet)
    this_data.mu = 2.2032e4; % [km^3 sec^-2] https://en.wikipedia.org/
    wiki/Standard_gravitational_parameter
    this_data.mass = 3.3011e23; % [kg]
    this_data.rgb_color = 'Silver'; % [string]
    k = k + 1;
    all_SS_body_data{k} = this_data;

```

```

    % Venus's Data
    this_data = [];
    this_data.name = 'Venus';
    this_data.spice_name = '299'; % [string] : Body's SPICE Name
    this_data.radius = 6.0518e3; % [km] https://en.wikipedia.org/wiki/
Venus
    this_data.mu = 3.24859e5; % [km^3 sec^-2] https://en.wikipedia.org/wiki/Standard\_gravitational\_parameter
    this_data.mass = 4.8675e24; % [kg]
    this_data.rgb_color = 'Yellow'; % [string]
    k = k + 1;
    all_SS_body_data{k} = this_data;

    % Earth's Data
    this_data = [];
    this_data.name = 'Earth';
    this_data.spice_name = '399'; % [string] : Body's SPICE Name
    this_data.radius = 6.371e3; % [km]
    this_data.mu = 3.986004418e5; % [km^3 sec^-2] https://en.wikipedia.org/wiki/Standard\_gravitational\_parameter
    this_data.mass = 5.9722e24; % [kg]
    this_data.rgb_color = 'Navy'; % [string]
    k = k + 1;
    all_SS_body_data{k} = this_data;

    % Moon's Data
    this_data = [];
    this_data.name = 'Moon';
    this_data.spice_name = '301'; % [string] : Body's SPICE Name
    this_data.radius = 1.7374e3; % [km]
    this_data.mu = 4.9048695e3; % [km^3 sec^-2] https://en.wikipedia.org/wiki/Standard\_gravitational\_parameter
    this_data.mass = 7.342e22; % [kg]
    this_data.rgb_color = 'Silver'; % [string]
    k = k + 1;
    all_SS_body_data{k} = this_data;

    % Mars's Data
    this_data = [];
    this_data.name = 'Mars';
    this_data.spice_name = '4'; % [string] : Body's SPICE Name
    this_data.radius = 3.3895e3; % [km] https://en.wikipedia.org/wiki/
Mars
    this_data.mu = 4.282837e4; % [km^3 sec^-2] https://en.wikipedia.org/wiki/Standard\_gravitational\_parameter
    this_data.mass = 6.4171e23; % [kg]
    this_data.rgb_color = 'DarkRed'; % [string]
    k = k + 1;
    all_SS_body_data{k} = this_data;

    % Jupiter's Data
    this_data = [];
    this_data.name = 'Jupiter';
    this_data.spice_name = '5'; % [string] : Body's SPICE Name

```

```

        this_data.radius = 6.9911e4; % [km] https://en.wikipedia.org/wiki/
Jupiter
        this_data.mu = 1.26686534e8; % [km^3 sec^-2] https://
en.wikipedia.org/wiki/Standard_gravitational_parameter
        this_data.mass = 1.8982e27; % [kg]
        this_data.rgb_color = 'Orange'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

        % Saturn's Data
        this_data = [];
        this_data.name = 'Saturn';
        this_data.spice_name = '699'; % [string] : Body's SPICE Name
        this_data.radius = 5.8232e4; % [km] https://en.wikipedia.org/wiki/
Saturn
        this_data.mu = 3.7931187e7; % [km^3 sec^-2] https://
en.wikipedia.org/wiki/Standard_gravitational_parameter
        this_data.mass = 5.6834e26; % [kg]
        this_data.rgb_color = 'Goldenrod'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

        % Mimas's Data
        this_data = [];
        this_data.name = 'Mimas';
        this_data.spice_name = '601'; % [string] : Body's SPICE Name
        this_data.radius = 1.982e2; % [km] https://en.wikipedia.org/wiki/
Mimas
        this_data.mass = 3.75094e19; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

        % Enceladus's Data
        this_data = [];
        this_data.name = 'Enceladus';
        this_data.spice_name = '602'; % [string] : Body's SPICE Name
        this_data.radius = 2.521e2; % [km] https://en.wikipedia.org/wiki/
Enceladus
        this_data.mass = 1.080318e20; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

        % Tethys's Data
        this_data = [];
        this_data.name = 'Tethys';
        this_data.spice_name = '603'; % [string] : Body's SPICE Name
        this_data.radius = 5.614e2; % [km] https://en.wikipedia.org/wiki/
Tethys_(moon)

```

```

        this_data.mass = 6.1749e20; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

% Dione's Data
this_data = [];
this_data.name = 'Dione';
this_data.spice_name = '604'; % [string] : Body's SPICE Name
this_data.radius = 5.31e2; % [km] https://en.wikipedia.org/wiki/
Dione_(moon)
        this_data.mass = 1.0954868e21; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

% Rhea's Data
this_data = [];
this_data.name = 'Rhea';
this_data.spice_name = '605'; % [string] : Body's SPICE Name
this_data.radius = 7.635e2; % [km] https://en.wikipedia.org/wiki/
Rhea_(moon)
        this_data.mass = 2.3064854e21; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

% Titan's Data
this_data = [];
this_data.name = 'Titan';
this_data.spice_name = '606'; % [string] : Body's SPICE Name
this_data.radius = 2.57473e3; % [km] https://en.wikipedia.org/
wiki/Titan_(moon)
        this_data.mass = 1.3452e23; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

% Hyperion's Data
this_data = [];
this_data.name = 'Hyperion';
this_data.spice_name = '607'; % [string] : Body's SPICE Name
this_data.radius = 1.35e2; % [km] https://en.wikipedia.org/wiki/
Hyperion_(moon)
        this_data.mass = 5.5510e18; % [kg]

```

```

        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

        % Iapetus's Data
        this_data = [];
        this_data.name = 'Iapetus';
        this_data.spice_name = '608'; % [string] : Body's SPICE Name
        this_data.radius = 7.344e2; % [km] https://en.wikipedia.org/wiki/Iapetus\_\(moon\)
        this_data.mass = 1.80565e21; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

        % Phoebe's Data
        this_data = [];
        this_data.name = 'Phoebe';
        this_data.spice_name = '609'; % [string] : Body's SPICE Name
        this_data.radius = 1.065e2; % [km] https://en.wikipedia.org/wiki/Iapetus\_\(moon\)
        this_data.mass = 8.3123e18; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

        % Helene's Data
        this_data = [];
        this_data.name = 'Helene';
        this_data.spice_name = '612'; % [string] : Body's SPICE Name
        this_data.radius = 1.81e2; % [km] https://en.wikipedia.org/wiki/Iapetus\_\(moon\)
        this_data.mass = 7.1e15; % [kg]
        this_data.mu = obj.gravitational_constant * this_data.mass; %
[km^3 sec^-2]
        this_data.rgb_color = 'Gray'; % [string]
        k = k + 1;
        all_SS_body_data{k} = this_data;

    end

end

end

```

Published with MATLAB® R2022a

2.6 True_SRP

Table of Contents

Class: True_SC_SRP	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	1
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3

Class: True_SC_SRP

Tracks the Solar Radiation Pressure effects on spacecraft

```
classdef True_SRP < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
enable_SRP % [boolean] Enable/disable SRP calculations
```

[] Properties: Variables Computed Internally

```
disturbance_torque_SRP % [Nm] Torque induced by solar radiation
pressure
disturbance_force_SRP % [N] Force induced by solar radiation pressure

num_faces % Number of spacecraft body faces
face_data % Data for spacecraft body faces
% - reflectance_factor : # [0, 1] for ith face
% - area [m^2] : Area of face
% - orientation [unit vector] : Normal vector in body frame B
% - location_center_of_pressure [m] : Center of pressure

num_solar_panel_faces % Number of solar panel faces
solar_panels_face_data % Data for solar panel faces
% - reflectance_factor : # [0, 1] for ith face
% - area [m^2] : Area of face
% - orientation [unit vector] : Normal vector in body frame B
% - location_center_of_pressure [m] : Center of pressure
```

[] Properties: Storage Variables

```
store
```

```
end
```

Methods

```
methods
```

[] Methods: Constructor

```
function obj = True_SRP(init_data, mission, i_SC)

% Initialize SRP enable flag
obj.enable_SRP = init_data.enable_SRP;

% Initialize disturbance vectors
obj.disturbance_torque_SRP = zeros(3,1);
obj.disturbance_force_SRP = zeros(3,1);

% Process spacecraft body faces
obj.num_faces = 0;
for i_shape =
1:length(mission.true_SC{i_SC}.true_SC_body.shape_model)
    shape_i =
mission.true_SC{i_SC}.true_SC_body.shape_model{i_shape};
    obj.num_faces = obj.num_faces + size(shape_i.Faces,1);

    for i_face = 1:size(shape_i.Faces,1)
        i_face_cnt = i_face + (i_shape-1)*size(shape_i.Faces,1);
        obj.face_data(i_face_cnt).reflectance_factor =
shape_i.Face_reflectance_factor(i_face);
        obj.face_data(i_face_cnt).area =
shape_i.Face_area(i_face);
        obj.face_data(i_face_cnt).orientation =
shape_i.Face_normal(i_face,:);
        obj.face_data(i_face_cnt).location_center_of_pressure =
shape_i.Face_center(i_face,:);
    end
end

% Process solar panel faces
obj.num_solar_panel_faces = 0;
if isfield(mission.true_SC{i_SC}, 'true_SC_solar_panel')
    for i_SP = 1:length(mission.true_SC{i_SC}.true_SC_solar_panel)
        SP = mission.true_SC{i_SC}.true_SC_solar_panel{i_SP};
        n_face = size(SP.shape_model.Faces,1);
        obj.num_solar_panel_faces = obj.num_solar_panel_faces +
2*n_face; % Both sides

        for j = 1:n_face
            % Solar cell side
            idx = (i_SP-1)*2*n_face + j;
            obj.solar_panels_face_data(idx).reflectance_factor =
SP.shape_model.Face_reflectance_factor_solar_cell_side(j);
```

```

        obj.solar_panels_face_data(idx).area =
SP.shape_model.Face_area;
        obj.solar_panels_face_data(idx).orientation =
SP.shape_model.Face_orientation_solar_cell_side;

obj.solar_panels_face_data(idx).location_center_of_pressure =
SP.shape_model.Face_center(j,:);

        % Opposite side
        idx = (i_SP-1)*2*n_face + j + n_face;
        obj.solar_panels_face_data(idx).reflectance_factor =
SP.shape_model.Face_reflectance_factor_opposite_side(j);
        obj.solar_panels_face_data(idx).area =
SP.shape_model.Face_area;
        obj.solar_panels_face_data(idx).orientation = -
SP.shape_model.Face_orientation_solar_cell_side;

obj.solar_panels_face_data(idx).location_center_of_pressure =
SP.shape_model.Face_center(j,:);
    end
end
end

% Calculate SRP before first iteration of ADL
obj.func_main_true_SRP(mission, i_SC);

% Initialize storage
obj.store.disturbance_torque_SRP =
zeros(mission.storage.num_storage_steps, 3);
obj.store.disturbance_force_SRP =
zeros(mission.storage.num_storage_steps, 3);

end

```

[] Methods: Store

```

function obj = func_update_true_SC_SRP_store(obj, mission)
    if mission.storage.flag_store_this_time_step == 1
        obj.store.disturbance_torque_SRP(mission.storage.k_storage,:)
= obj.disturbance_torque_SRP';
        obj.store.disturbance_force_SRP(mission.storage.k_storage,:) =
obj.disturbance_force_SRP';
    end
end

```

[] Methods: Main

```

function obj = func_main_true_SRP(obj, mission, i_SC)
    % Reset disturbance terms
    obj.disturbance_torque_SRP = zeros(3,1);
    obj.disturbance_force_SRP = zeros(3,1);

    if obj.enable_SRP == 1

```

```

        % Process all faces (spacecraft body + solar panels)
        for i = 1:obj.num_faces + obj.num_solar_panel_faces
            % Get face data
            if i <= obj.num_faces
                face_i = obj.face_data(i).orientation';
                faceCP_sc =
obj.face_data(i).location_center_of_pressure';
                face_area = obj.face_data(i).area;
                face_reflectance_factor =
obj.face_data(i).reflectance_factor;
            else
                face_i = obj.solar_panels_face_data(i-
obj.num_faces).orientation';
                faceCP_sc = obj.solar_panels_face_data(i-
obj.num_faces).location_center_of_pressure';
                face_area = obj.solar_panels_face_data(i-
obj.num_faces).area;
                face_reflectance_factor =
obj.solar_panels_face_data(i-obj.num_faces).reflectance_factor;
            end

            % Calculate sun vector and incidence
            faceCP_sun =
(mission.true_SC{i_SC}.true_SC_navigation.position -
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.position)'
+ ...
            mission.true_SC{i_SC}.true_SC_adc.rotation_matrix *
            faceCP_sc;

            faceCP_sun_normalized = faceCP_sun/norm(faceCP_sun);

            SC_face_normal =
mission.true_SC{i_SC}.true_SC_adc.rotation_matrix * face_i;
            SC_face_normal_normalized = SC_face_normal/
norm(SC_face_normal);

            % Calculate incidence angle
            incidence = real(acosd(dot(SC_face_normal_normalized,
faceCP_sun_normalized)));

            % Calculate force and torque if face is illuminated
            if abs(incidence) < 90
                % Force calculation
                F_SRP_magnitude =
mission.true_solar_system.solar_constant_AU /
mission.true_solar_system.light_speed * ...
                (mission.true_solar_system.AU_distance /
norm(mission.true_SC{i_SC}.true_SC_navigation.position -
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.position))^2
                * ...
                face_area * (1 + face_reflectance_factor) *
cosd(incidence);

```

```

        F_SRP_vector_J2000 = F_SRP_magnitude *
faceCP_sun_normalized;

        % Torque calculation
        force_on_surface_at_cp =
(mission.true_SC{i_SC}.true_SC_adc.rotation_matrix)' * F_SRP_vector_J2000;
        this_face_lever_arm = faceCP_sc -
mission.true_SC{i_SC}.true_SC_body.location_COM';
        this_face_torque = cross(this_face_lever_arm,
force_on_surface_at_cp);

        % Accumulate disturbances
        obj.disturbance_torque_SRP =
obj.disturbance_torque_SRP + this_face_torque;

        obj.disturbance_force_SRP = obj.disturbance_force_SRP
+ F_SRP_vector_J2000;
    end
end
end

        mission.true_SC{i_SC}.true_SC_adc.disturbance_torque
= mission.true_SC{i_SC}.true_SC_adc.disturbance_torque +
obj.disturbance_torque_SRP;

        % Update storage
        obj = func_update_true_SC_SRP_store(obj, mission);
    end
end
end
end

```

Published with MATLAB® R2022a

2.7 True_Stars

Table of Contents

Class: True_Stars	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
Methods	1
[] Methods: Constructor	1

Class: True_Stars

Stars in the sky

```
classdef True_Stars < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
    maximum_magnitude % [float] Maximum magnitude of stars visible to  
    camera (Optional)
```

[] Properties: Variables Computed Internally

```
    num_stars % [integer] Number of stars  
    sao_name % [string] Smithsonian Astrophysical Observatory (SAO) Star  
    Catalog's name of star  
    magnitude_visible % [float] Magnitude of star  
    all_stars_unit_vector % [] Unit vector denoting position of all stars  
  
end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class, Initialize from SAO Catalog

```
function obj = True_Stars(mission)  
  
    YYYY = str2double(mission.true_time.t_initial_date_string(8:11));  
  
    % SAO Star Catalog : http://tdc-www.harvard.edu/catalogs/sao.html
```

```

fid=fopen('saoNAN.txt');
M=textscan(fid, '%f %f %f %f %f %f %f %f', 'headerlines', 1);
fclose(fid);

obj.sao_name=M{1};
obj.magnitude_visible=M{3};
RA=(M{5}+(M{7}*(YYYY-2000)))/15;
DEC=(M{6}+(M{8}*(YYYY-2000)));

obj.num_stars = length(obj.sao_name);

obj.all_stars_unit_vector = zeros(obj.num_stars,3);
for i=1:1:obj.num_stars

    x_hat = [1 0 0]';

    RA_angle = deg2rad(RA(i)*15); % [rad]
    Dec_angle = deg2rad(DEC(i)); % [rad]

    Rot_Z_star_RA = [cos(RA_angle) -sin(RA_angle) 0;
                     sin(RA_angle)  cos(RA_angle) 0;
                     0                0 1];

    Rot_Y_star_Dec = [cos(Dec_angle) 0 sin(Dec_angle);
                     0 1 0;
                     -sin(Dec_angle) 0 cos(Dec_angle)];

    Rot_RA_dec = Rot_Y_star_Dec * Rot_Z_star_RA;

    obj.all_stars_unit_vector(i,:) = (Rot_RA_dec*x_hat)';

end

obj.maximum_magnitude = max(obj.magnitude_visible);

end

end

end

```

Published with MATLAB® R2022a

2.8 True_Target_SPICE

Table of Contents

Class: True_Target_SPICE	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	9
[] Methods: Main	9
[] Methods: Compute Rotation Matrix	10
[] Methods: OLD	11

Class: True_Target_SPICE

Tracks the main target body (uses SPICE for updating position, velocity)

```
classdef True_Target_SPICE < handle
```

Properties

properties

[] Properties: Initialized Variables

```
name % [string] Name of Target
```

[] Properties: Variables Computed Internally

```
rotation_period % [sec] Period of one rotation  
rotation_rate % [rad/sec]
```

```
gravity_filename % [string] Filename of the Target gravity field in a  
particular format
```

```
gravity_field % Target's gravity field, computed from gravity_filename  
gravity_degree_harmonics % [integer] Degree harmonics of the gravity  
field
```

```
shape_model % Shape model of Target  
% - shape_model.Vertices [m] : Position of vertices  
% - shape_model.Faces : Triplet of vertex indices define a face  
shape_model_type % [string] Type of shape model  
radius % [km] Radius of Target
```

```
pole_RA % [deg] Right Ascension (RA) of Target's pole  
pole_Dec % [deg] Declination (DEC) of Target's pole  
prime_meridian % [deg] : Initial Prime Meridian angle of Target  
rotation_matrix_pole_RA_Dec % Rotation matrix of Target's pole due to  
RA, Dec only (dosent' change with time)
```

```

rotation_matrix % Rotation matrix of Target from Body Frame to J2000
(changes with time)

spice_filename % [string] Target's SPICE FileName
spice_name % [string] Target's SPICE Name

mu % [km^3 sec^-2] Target's standard gravitational parameter  $\mu$  = GM
mass % [kg] Mass of Target

position % [km] Current position of Target wrt Sun-centered J2000
velocity % [km/sec] Current velocity of Target wrt Sun-centered J2000
position_array % [km] Position array of Target wrt Sun-centered J2000,
corresponding to time array in mission.true_time.time_position_array

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_Target_SPICE(init_data, mission)

path_body_data = '../..../MuSCAT_Supporting_Files/SB_data/';

obj.name = init_data.target_name;

switch obj.name

case 'Bennu'
    % obj.name = 'Bennu';

    obj.rotation_period = 4.296057*3600; % [sec] From https://
en.wikipedia.org/wiki/101955_Bennu

    % Gravity Model
    obj.gravity_filename = [path_body_data 'Bennu/
bennu_harmonics_jpl5.txt'];
    obj.gravity_field = GravityField(obj.gravity_filename); %
Load gravity field
    obj.gravity_degree_harmonics = 8;

    % Shape Model
    % readObj.m from https://www.mathworks.com/matlabcentral/
fileexchange/18957-readobj

```

```

        obj_shape = readObj([path_body_data 'Bennu/
bennu_g_06290mm_spc_obj_0000n00000_v008.obj']);
        % other options are Bennu_OSIRIS-REx_2018.obj (2.6MB) and
        Bennu_OSIRIS-REx_2019.obj (7.2MB)

        obj.shape_model_type = 'trisurf';
        obj.shape_model = [];
        obj.shape_model.Vertices = (1e3)*obj_shape.v; % [m]
        obj.shape_model.Faces = obj_shape.f.v;

        % SPICE
        obj.spice_filename = [path_body_data 'Bennu/
sb-101955-118.bsp'];
        obj.spice_name = '2101955';

        % Pole Data
        obj.pole_RA = 85.65; % [deg]
        obj.pole_Dec = -60.17; % [deg]
        obj.prime_meridian = 0; % [deg] (at t_init)

    case 'Apophis'
        % obj.name = 'Apophis';

        obj.rotation_period = (360/3.155588020452885e+02)*86400; %
[sec] From SPICE

        % Gravity Model
        obj.gravity_filename = [path_body_data 'Apophis/
Apophis_CMoffset.txt'];
        obj.gravity_field = GravityField(obj.gravity_filename); %
Load gravity field
        obj.gravity_degree_harmonics = 8;

        % Shape Model
        obj_shape = readObj([path_body_data 'Apophis/
ApophisModel1.obj']);
        obj.shape_model = [];
        obj.shape_model_type = 'trisurf';
        obj.shape_model.Vertices = obj.gravity_field.R * 1e3 *
obj_shape.v; % [m]
        obj.shape_model.Faces = obj_shape.f.v;

        % SPICE
        obj.spice_filename = [path_body_data 'Apophis/
apophis.bsp'];
        obj.spice_name = '2099942';

        % Pole Data
        obj.pole_RA = 250; % [deg]
        obj.pole_Dec = -75; % [deg]
        obj.prime_meridian = 0; % [deg] (at t_init)

```

```

        case 'Toutatis'
            % obj.name = 'Toutatis';

            % Rotation Rate
            obj.rotation_period = 176*3600; % [sec] https://en.wikipedia.org/wiki/4179\_Toutatis
            obj.rotation_rate = 2*pi/obj.rotation_period; % [rad /sec]
            warning('Toutatis is tumbling!') % http://abyss.uoregon.edu/~js/ast121/lectures/toutatis.html

            % Gravity Model
            obj.gravity_filename = 'Toutatis_CMoffset.txt';
            obj.gravity_field = GravityField(obj.gravity_filename); %
Load gravity field
            obj.gravity_degree_harmonics = 8;

            obj_shape =
readObj([path_body_data 'Toutatis/4179toutatis.tab.obj']);
            obj.shape_model = [];
            obj.shape_model_type = 'trisurf';
            obj.shape_model.Vertices = obj.gravity_field.R * 1e3 *
obj_shape.v; % [m]
            obj.shape_model.Faces = obj_shape.f.v;

            obj.spice_filename =
[path_body_data 'Toutatis/2004179.bsp'];
            obj.spice_name = '2004179';

            % Pole Data
            obj.pole_RA = 0; % [deg]
            obj.pole_Dec = 90*pi/180; % [deg]
            obj.prime_meridian = 0; % [deg] (at t_init)

            obj.mass = 1.9e13;
            obj.ode_options = odeset('RelTol',1e-14,'AbsTol',1e-14);

        case 'Itokawa'
            % obj.name = 'Itokawa';

            % Rotation Rate
            obj.rotation_period = 12.132*3600; % [sec] https://en.wikipedia.org/wiki/25143\_Itokawa
            obj.rotation_rate = 2*pi/obj.rotation_period; % [rad /
sec]

            % Gravity Model
            obj.gravity_filename = 'Itokawa_CMoffset.txt';
            obj.gravity_field = GravityField(obj.gravity_filename); %
Load gravity field

```

```

obj.gravity_degree_harmonics = 8;

% Shape Model
% Shape model from https://sbn.psi.edu/pds/shape-models/
% readObj.m from https://www.mathworks.com/matlabcentral/
fileexchange/18957-readobj
obj_shape = readObj([path_body_data 'Itokawa/
Itokawa_ver64q.tab.obj']);
obj.shape_model = [];
obj.shape_model_type = 'trisurf';
obj.shape_model.Vertices = obj.gravity_field.R * 1e3 *
obj_shape.v; % [m]
obj.shape_model.Faces = obj_shape.f.v;

obj.spice_filename =
[path_body_data 'Itokawa/2025143.bsp'];
obj.spice_name = '2025143';

% Pole Data
obj.pole_RA = 90.53*pi/180; % [rad]
obj.pole_Dec = -66.30*pi/180; % [rad]
obj.prime_meridian = 0; % [deg] (at t_init)

obj.mass = 3.51e10;
obj.ode_options = odeset('RelTol',1e-14,'AbsTol',1e-14);

case '1996HW1'
% obj.name = '1996HW1';

% Rotation Rate
obj.rotation_period = 8.762*3600; % [sec] https://
echo.jpl.nasa.gov/asteroids/1996HW1/1996hw1.html
obj.rotation_rate = 2*pi/obj.rotation_period; % [rad /sec]

% Gravity Model
obj.gravity_filename = '1996HW1_CMoffset.txt';
obj.gravity_field = GravityField(obj.gravity_filename); %
Load gravity field
obj.gravity_degree_harmonics = 8;

% Shape Model
% Shape model from https://sbn.psi.edu/pds/shape-models/
% readObj.m from https://www.mathworks.com/matlabcentral/
fileexchange/18957-readobj
obj_shape =
readObj([path_body_data '1996HW1/1996HW1_a8567.tab.obj']);
obj.shape_model = [];
obj.shape_model_type = 'trisurf';
obj.shape_model.Vertices = obj.gravity_field.R * 1e3 *
obj_shape.v; % [m]
obj.shape_model.Faces = obj_shape.f.v;

```

```

        obj.spice_filename =
[path_body_data 'Itokawa/2008567.bsp'];
        obj.spice_name = '2008567';

        % Pole Data
        obj.pole_RA = 281*pi/180; % [rad]
        obj.pole_Dec = -30*pi/180; % [rad]
        obj.prime_meridian = 0; % [deg] (at t_init)

        obj.mass = 3.51e10; % I CANT FIND IT
        obj.ode_options = odeset('RelTol',1e-14,'AbsTol',1e-14);

        case 'Earth'
            % obj.name = 'Earth';

            obj.rotation_period = 23.9345 * 3600; % [sec] From
https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html

            % Gravity Model (https://www2.csr.utexas.edu/grace/
gravity/ggm02/)
            obj.gravity_filename = [path_body_data 'Earth/
ggm02c.txt'];
            obj.gravity_field = GravityField(obj.gravity_filename); %
Load gravity field
            if isfield(mission_init_data,'gravity_degree_harmonics')
                obj.gravity_degree_harmonics =
mission_init_data.gravity_degree_harmonics;
            else
                obj.gravity_degree_harmonics = 8;
            end

            % Shape Model
            obj.shape_model = [];
            obj.shape_model_type = 'sphere';
            obj.shape_model.img = [path_body_data 'Earth/
earth_surface.jpg'];

            % SPICE
            obj.spice_name = '399';
            obj.spice_filename = [path_body_data 'Earth/
earth_200101_990825_predict.bpc'];

            % Pole Data
            % From https://nssdc.gsfc.nasa.gov/planetary/factsheet/
earthfact.html

            ref_date_time = cal2sec('01-JAN-2000 00:12:00');
            T = (mission_true_time.t_initial_date - ref_date_time) /
(365.525*86400*100); % Julian centuries from reference date
            obj.pole_RA = 0.00 - 0.641 * T; % [deg]
            obj.pole_Dec = 90.00 - 0.557 * T; % [deg]

            cspice_furnsh([path_body_data 'Earth/naif0012.tls']) %
Leapseconds kernel file

```

```

        JD.UTC =
cspice_et2utc(mission_true_time.t_initial_date, 'J', 6); % Format 'JD
2446533.18834276'

        JD.UTC = strsplit(JD.UTC, ' ');
        JD.UTC = str2double(JD.UTC{2});
        JD_UT1 = JD.UTC; % Approximation, UTC is design to follow
UT1 within +/- 0.9s
        % From NASA TP 20220014814, Sec 4.3.2 Sidereal Motion
        obj.prime_meridian =
rad2deg(wrapTo2Pi(2*pi*(0.7790572732640 + 1.00273781191135448 * (JD_UT1 -
2451545.0)))); % [deg] (at t_init)

        case 'IBD_Asteroid'
            % obj.name = 'IBD_Asteroid';

            obj.rotation_period = (360/3.15588020452885e+02)*86400; %
[sec] From SPICE

            % Gravity Model
            obj.gravity_filename = [path_body_data 'IBDAst/
Apophis_CMoffset.txt'];
            obj.gravity_field = GravityField(obj.gravity_filename); %
Load gravity field
            obj.gravity_degree_harmonics = 8;

            % Shape Model
            obj_shape = readObj([path_body_data 'IBDAst/Bennu-
Radar.obj']);
            obj.shape_model = [];
            obj.shape_model_type = 'trisurf';
            obj.shape_model.Vertices = obj.gravity_field.R * 1e3 *
obj_shape.v; % [m]
            obj.shape_model.Faces = obj_shape.f.v;

            % SPICE
            obj.spice_filename = [path_body_data 'IBDAst/
apophis.bsp'];
            obj.spice_name = '2099942';

            % Pole Data
            obj.pole_RA = 250; % [deg]
            obj.pole_Dec = -75; % [deg]
            obj.prime_meridian = 0; % [deg] (at t_init)

            obj.mass = 6.1e10;
            obj.ode_options = odeset('RelTol',1e-14,'AbsTol',1e-14);

        case 'Enceladus'
            % obj.name = 'Enceladus';

            obj.rotation_period = 1.370218*86400; % [sec] https://
en.wikipedia.org/wiki/Enceladus

```

```

        % Gravity Model
        obj.gravity_filename = [path_body_data 'Bennu/
bennu_harmonics_jpl5.txt'];
        obj.gravity_field = GravityField(obj.gravity_filename); %
        Load gravity field
        obj.gravity_degree_harmonics = 8;

        % Shape Model
        obj.shape_model = [];
        obj.shape_model_type = 'sphere';
        obj.shape_model_img = [path_body_data 'Enceladus/
Enceladus_surface_color.jpg'];

        % SPICE
        obj.spice_filename = '../..../MuSCAT_Supporting_Files/
SC_data/Nightingale/insar_6stride_26d_v7_scpse.bsp';
        obj.spice_name = '602';

        % Pole Data
        obj.pole_RA = 40.7; % [deg]
        obj.pole_Dec = 83.5; % [deg]
        obj.prime_meridian = 0; % [deg] (at t_init)

        % Use SPICE TPC file instead
        cspice_furnsh('../..../MuSCAT_Supporting_Files/SB_data/
Enceladus/enceladus_ssd_230702_v1.tpc')
        % cspice_furnsh('../..../MuSCAT_Supporting_Files/SB_data/
Enceladus/pck_sat441.tpc')
        % cspice_furnsh('../..../MuSCAT_Supporting_Files/SB_data/
Enceladus/pck_sat441_enceladus_frame_edit_only_for_recreation.tpc')

        obj.mass = 1.080318e20; % [kg]
        obj.radius = 252; % [km]

    otherwise
        error('Invalid Target type')
    end

    cspice_furnsh('../..../MuSCAT_Supporting_Files/SPICE/de440s.bsp');
    cspice_furnsh(obj.spice_filename);

    obj.rotation_rate = 360/obj.rotation_period; % [deg /sec]

    % Rotation Matrix to Target's Body-fixed Inertial Frame.
    % See details here: https://naif.jpl.nasa.gov/pub/naif/
    toolkit\_docs/MATLAB/req/pck.html#Orientation%20Models%20used%20by%20PCK
    %20Software
    Rot_Z_pole_RA = [
        cosd(90 + obj.pole_RA) -sind(90 + obj.pole_RA) 0;
        sind(90 + obj.pole_RA) cosd(90 + obj.pole_RA) 0;
        0 0 1];
    Rot_X_pole_Dec = [
        1 0 0;

```

```

        0 cosd(90 - obj.pole_Dec) -sind(90 - obj.pole_Dec);
        0 sind(90 - obj.pole_Dec)  cosd(90 - obj.pole_Dec)];

obj.rotation_matrix_pole_RA_Dec = Rot_Z_pole_RA * Rot_X_pole_Dec;

% Other Parameters
obj.mu = obj.gravity_field.GM; % [km^3 sec^-2]
obj.radius = obj.gravity_field.R ; % [km]

% Initialize position, velocity, and rotation matrix

obj.position = zeros(1,3); % [km]
obj.velocity = zeros(1,3); % [km/sec]
obj.position_array =
zeros( mission.true_time.num_time_steps_position_array,3); % [km]

obj = func_main_true_target(obj, mission);

% Initialize Variables to store position and velocity of target
obj.store = [];

obj.store.name = obj.name;
obj.store.position = zeros(mission.storage.num_storage_steps,
length(obj.position));
obj.store.velocity = zeros(mission.storage.num_storage_steps,
length(obj.velocity));

obj = func_update_target_store(obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_target_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.position(mission.storage.k_storage,:) =
obj.position; % [sec]
        obj.store.velocity(mission.storage.k_storage,:) =
obj.velocity; % [sec]
    end

end

```

[] Methods: Main

Update target's position, velocity, rotation matrix for current time

```

function obj = func_main_true_target(obj,mission)

```

```

        target_pos_vel_this_time =
cspice_spkezr(obj.spice_name,mission.true_time.date,'J2000','NONE','SUN');
        obj.position = target_pos_vel_this_time(1:3)'; % [km]
        obj.velocity = target_pos_vel_this_time(4:6)'; % [km/sec]

        target_pos_vel_array = (cspice_spkezr(obj.spice_name,
mission.true_time.prev_date +
mission.true_time.time_position_array' , 'J2000','NONE','SUN'))';
        obj.position_array = target_pos_vel_array(:,1:3);

        % Small bodies
        obj.rotation_matrix = func_compute_target_rotation_matrix(obj,
mission);

        % Planets
        % obj.rotation_matrix = cspice_pxform('J2000', ['IAU_',obj.name],
mission_true_time.date);

        % Store
        obj = func_update_target_store(obj, mission);
end

```

[] Methods: Compute Rotation Matrix

Update target's rotation matrix for current time

```

function rot = func_compute_target_rotation_matrix(obj, mission)

switch obj.name

    case 'Enceladus'
        fromFrame = 'IAU_ENCELADUS'; % Body-fixed frame of
Enceladus
        toFrame = 'J2000';           % Inertial frame (e.g.,
J2000)

        rot = cspice_pxform(fromFrame, toFrame,
mission.true_time.date);

    otherwise
        theta_PM = obj.prime_meridian + obj.rotation_rate *
(mission.true_time.time - mission.true_time.t_initial);
        Rot_Z_PM = [
            cosd(theta_PM) sind(theta_PM) 0;
            -sind(theta_PM) cosd(theta_PM) 0;
            0 0 1];
        rot = Rot_Z_PM*obj.rotation_matrix_pole_RA_Dec; % From
J2000 to Body frame

        rot = rot'; % From Body frame to J2000
    end
end

```

[] Methods: OLD

```
%          function [rv, rot] = func_get_position_velocity_rot(obj,
true_time, tspan)
%          % Input:
%          %   true_time: True time object
%          %   tspan: Time span [s] (N x 1)
%          % Output:
%          %   rv: Position and velocity of SB in inertial frame
[km, km/sec] (N x 6)
%          %   rot: Rotation matrix of from inertial frame to the
body-fixed frame (N x 3 x 3)
%
%          rv = cspice_spkezr(obj.spice_name,
true_time.t_initial_date + tspan, 'J2000', 'NONE', 'SUN')';
%          if nargout > 1
%              rot = zeros(length(tspan),3,3);
%              for i = 1:length(tspan)
%                  rot(i, :, :) =
func_compute_target_rotation_matrix(obj, true_time, tspan(i));
%              end
%          end
%          end
end

end
```

Published with MATLAB® R2022a

2.9 True_Time

Table of Contents

Class: True_Time	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Initialize Store	4
[] Methods: Store	4
[] Methods: Store Attitude	4
[] Methods: Main	5
[] Methods: Main Attitude	6
[] Methods: Set Time	6

Class: True_Time

Keeps track of the all time variables in the simulation

```
classdef True_Time < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
t_initial % [sec] : Start time
t_final % [sec] : Final time
time_step % [sec] : Simulation time step
t_initial_date_string % [string] : Start date of simulation. Format =
[DD-MMM(words)-YYYY HH-MM-SS].
```

```
time_step_attitude % [sec] : Time step for attitude dynamics
(Optional)
```

```
print_progress_steps % [integer] : Number of steps to skip between
printing progress (Optional)
```

```
time_step_position_array % [sec] : Time step for position dynamics
array (Optional)
```

[] Properties: Variables Computed Internally

```
t_initial_date % [sec from J2000] : Start date of simulation.
```

```

        num_time_steps % [integer] : Number of simulation time steps      #
        time % [sec] : Current true time                                  #
        date % [sec from J2000] : Current true date                      #

        num_time_steps_attitude % [integer] : Number of attitude dynamics loop
time steps within one simulation time step #
        time_attitude % [sec] : Current true time within attitude dynamics
loop #

        k % [integer] : Time loop variable                               #
        k_attitude % [integer] : Attitude Dynamics Time loop variable    #

        num_time_steps_position_array % [integer]
        time_position_array % [sec]

        prev_time % [sec] : Previous true time
#
        prev_date % [sec from J2000] Previous true date

        data % to store other values

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_Time(init_data)

    obj.t_initial = init_data.t_initial;
    obj.time = obj.t_initial;

    obj.t_final = init_data.t_final;
    obj.time_step = init_data.time_step;

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    obj.num_time_steps = ceil((obj.t_final - obj.t_initial)/
obj.time_step);

```

```

obj.t_initial_date_string = init_data.t_initial_date_string;
obj.t_initial_date = cal2sec(obj.t_initial_date_string);
obj.date = obj.t_initial_date + obj.time; % Seconds from '01-
JAN-2000 00:00:00'

if isfield(init_data, 'time_step_attitude')
    % time_step_attitude_dynamics has been specified
    obj.time_step_attitude = init_data.time_step_attitude;
else
    obj.time_step_attitude = obj.time_step;
end

obj.num_time_steps_attitude = (obj.time_step/
obj.time_step_attitude);

if ~isinteger(int32(obj.num_time_steps_attitude))
    error('time_step must be divisble by
time_step_attitude_dynamics!');
end

obj.time_attitude = obj.time;

obj.k = 0;
obj.k_attitude = 0;

if isfield(init_data, 'print_progress_steps')
    % print_progress_steps has been specified
    obj.print_progress_steps = init_data.print_progress_steps;
else
    obj.print_progress_steps = ceil((obj.num_time_steps)/1000);
end

% Create Time Array for Position Dynamics

if isfield(init_data, 'time_step_position_array')
    obj.time_step_position_array =
init_data.time_step_position_array;
else
    obj.time_step_position_array = min([10, obj.time_step,
obj.time_step_attitude]); % [sec]
end

obj.num_time_steps_position_array = (obj.time_step/
obj.time_step_position_array) + 1;

if ~isinteger(int32(obj.num_time_steps_position_array))
    error('time_step must be divisble by
time_step_position_array!');
end

obj.time_position_array = [0: obj.time_step_position_array :
obj.time_step];

```

```
obj.prev_time = obj.time; % [sec]
obj.prev_date = obj.date; % [sec from J2000]
```

```
end
```

[] Methods: Initialize Store

Initialize the store variable

```
function obj = func_initialize_time_store(obj, mission)

    % Variables to store: time, date, time_attitude
    obj.store.time = zeros(mission.storage.num_storage_steps,
length(obj.time));
    obj.store.date = zeros(mission.storage.num_storage_steps,
length(obj.date));
    obj.store.time_attitude =
zeros(mission.storage.num_storage_steps_attitude, length(obj.time_attitude));

    % Store first set of variables
    obj = func_update_time_store(obj, mission);
    obj = func_update_time_store_attitude(obj, mission);

end
```

[] Methods: Store

Update the store variable

```
function obj = func_update_time_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.time(mission.storage.k_storage,:) = obj.time; %
[sec]
        obj.store.date(mission.storage.k_storage,:) = obj.date; %
[sec]
    end

end
```

[] Methods: Store Attitude

Update the attitude store variable

```
function obj = func_update_time_store_attitude(obj, mission)

    if mission.storage.flag_store_this_time_step_attitude == 1
        obj.store.time_attitude(mission.storage.k_storage_attitude,:)
= obj.time_attitude; % [sec]
    end

end
```

end

[] Methods: Main

Function to update current time and date within main loop

```
function obj = func_update_true_time_date(obj, k)

    obj.k = k;

    obj.prev_time = obj.time; % [sec]
    obj.prev_date = obj.date; % [sec from J2000]

    obj.time = obj.time + obj.time_step;
    obj.date = obj.t_initial_date + obj.time;

    % Print progress
    if mod(obj.k, obj.print_progress_steps) == 0

        % Expected time left
        time_elap = seconds(toc);
        time_per_loop = time_elap / obj.k;
        time_left = time_per_loop * (obj.num_time_steps - obj.k);

        time_sim_elapsed = seconds(obj.time - obj.t_initial);
        time_sim_total = seconds(obj.t_final - obj.t_initial);

        perc = round(time_sim_elapsed / time_sim_total * 100, 1);

        format = 'dd:hh:mm:ss';
        time_elap.Format = format;
        time_left.Format = format;
        time_sim_elapsed.Format = format;
        time_sim_total.Format = format;

        % Base progress message
        progress_msg = ['- Simulation: ', char(time_sim_elapsed), ' /', char(time_sim_total), ' (', num2str(perc), '%), ' ...
            ' Elapsed: ', char(time_elap), ', Left: ',
            char(time_left), ', Total: ', char(time_elap + time_left)];

        % Add memory info every 100 iterations
        if mod(obj.k, obj.print_progress_steps * 100) == 0
            memoryInfo = evalc('dispmemory()');
            progress_msg = [progress_msg, ' | Current Memory Usage
By Matlab: ', strtrim(memoryInfo)];
        end

        disp(progress_msg)
    end
end
```

[] Methods: Main Attitude

Function to update current time within attitude dynamics loop

```
function obj = func_update_true_time_attitude(obj, k_attitude)

    obj.k_attitude = k_attitude;
    obj.time_attitude = obj.time_attitude + obj.time_step_attitude;

end
```

[] Methods: Set Time

Set time to a specific value

```
function obj = func_set_time(obj, time)

    obj.time = time;
    obj.date = obj.t_initial_date + obj.time;

end

end

end
```

Published with MATLAB® R2022a

Chapter 3

SC Physics Based Simulation Layer Classes

3.1 True_SC_ADC

Table of Contents

Class: True_SC_ADC	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3
[] Methods: Update Rotation Matrix	4
[] Methods: Rigid Attitude Dynamics	4
[] Methods	5

Class: True_SC_ADC

Tracks the attitude and angular velocity of the SC

```
classdef True_SC_ADC < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
    attitude % [quaternion] : Orientation of inertial frame I with  
respect to the body frame B  
  
    angular_velocity % [rad/sec] : Angular velocity of inertial frame I  
with respect to the body frame B  
  
    mode_true_SC_attitude_dynamics_selector % [string] Different attitude  
dynamics modes  
    % - 'Rigid' : Use Rigid Body Dynamics
```

[] Properties: Variables Computed Internally

```
    rotation_matrix % : Rotation matrix that converts vector in body frame  
B to vector in inertial frame I  
    dot_angular_velocity % [rad/sec^2] : Time derivative of  
angular_velocity (needed by RWA)  
    % RWA - Reaction Wheels | MT momentum Thrusters?  
    control_torque % [Nm] : Control torque about Center of Mass of SC,  
usually generated by MT thrusters and RWA  
    disturbance_torque % [Nm] : Disturbance torque about Center of Mass of  
SC  
    total_torque % [Nm] - Sum of the two above
```

```

        total_wheel_momentum % [kg#m^2/s] : Total momentum of all reaction
wheels

        ode_options % : Options for Matlab's ODE function
odeset( 'RelTol',1e-14,'AbsTol',1e-14)

        plot_handle % : plot handle for attitude visualization

```

[] Properties: Storage Variables

```

        store

    end

```

Methods

```

    methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_SC_ADC(init_data, mission)

    obj.attitude = func_quaternion_properize(init_data.attitude); %
[quaternion]
    obj.angular_velocity = init_data.angular_velocity; % [rad/sec]

    % Compute Rotation Matrix
    obj = func_update_true_SC_ADC_rotation_matrix(obj);

    obj.dot_angular_velocity = zeros(3,1);
    obj.control_torque = zeros(3,1);
    obj.disturbance_torque = zeros(3,1);
    obj.total_wheel_momentum = zeros(3,1); % Initialize as 3x1 vector
    obj.total_torque = zeros(3,1);

    obj.ode_options = odeset('RelTol',1e-14,'AbsTol',1e-14);

    obj.mode_true_SC_attitude_dynamics_selector =
init_data.mode_true_SC_attitude_dynamics_selector;

    % Initialize Variables to store: attitude and angular velocity of
SC
    obj.store = [];

    obj.store.attitude =
zeros(mission.storage.num_storage_steps_attitude, length(obj.attitude));
    obj.store.angular_velocity =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.angular_velocity));

```

```

        obj.store.control_torque =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.control_torque));
        obj.store.disturbance_torque =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.disturbance_torque));
        obj.store.total_torque =
zeros(mission.storage.num_storage_steps_attitude, length(obj.total_torque));

        obj = func_update_true_SC_adc_store(obj, mission);
    end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_adc_store(obj, mission)
    if mission.storage.flag_store_this_time_step_attitude == 1
        obj.store.attitude(mission.storage.k_storage_attitude,:) =
obj.attitude; % [sec]

        obj.store.angular_velocity(mission.storage.k_storage_attitude,:) =
obj.angular_velocity; % [sec]

        obj.store.control_torque(mission.storage.k_storage_attitude,:) =
obj.control_torque; % [sec]

        obj.store.disturbance_torque(mission.storage.k_storage_attitude,:) =
obj.disturbance_torque; % [sec]
        obj.store.total_torque(mission.storage.k_storage_attitude,:) =
obj.total_torque; % [sec]
    end
end

```

[] Methods: Main

Update SC's Attitude and Angular Velocity

```

function obj = func_main_true_SC_attitude(obj, mission, i_SC)

    switch obj.mode_true_SC_attitude_dynamics_selector

        case 'Rigid'

            % Update disturbance torque is done
            % in the SRP and GG classes

            % Update total wheel momentum
            obj = obj.func_update_total_wheel_momentum(mission, i_SC);

            % Update attitude and angular velocity

```

```

        obj = obj.func_true_SC_attitude_dynamics_rigid(mission,
i_SC);

        otherwise
            error('Havent written yet!')

        end

        % Sum torques for storage
        obj.total_torque = obj.control_torque + obj.disturbance_torque;

        % Compute Rotation Matrix
        obj = func_update_true_SC_ADC_rotation_matrix(obj);

        % Store
        obj = func_update_true_SC_adc_store(obj, mission);

        % Reset variables
        obj.control_torque = zeros(3,1);      % [Nm]

        % Reset every main time step
        if (mission.true_time.k_attitude ==
mission.true_time.num_time_steps_attitude)
            obj.disturbance_torque = zeros(3,1); % Reset torque
        end
    end
end

```

[] Methods: Update Rotation Matrix

Update SC's Rotation Matrix

```

function obj = func_update_true_SC_ADC_rotation_matrix(obj)

    % Compute Rotation Matrix
    SC_True_e_current = obj.attitude(1:3)/norm(obj.attitude(1:3));
    SC_True_Phi_current = 2*acos(obj.attitude(4)); % [rad]
    obj.rotation_matrix =
func_create_rotation_matrix(SC_True_e_current, SC_True_Phi_current);

end

```

[] Methods: Rigid Attitude Dynamics

Update SC's Attitude and Angular Velocity

```

function obj = func_true_SC_attitude_dynamics_rigid(obj, mission,
i_SC)

    SC_Quaternion_Omega_current = [obj.attitude';
obj.angular_velocity'];

    this_time_array = [0 mission.true_time.time_step_attitude];

```

```

        [T,X]=ode45(@(t,X) func_ode_attitude_dynamics(t, X, mission,
i_SC), this_time_array, SC_Quaternion_Omega_current, obj.ode_options);

        new_SC_Quaternion_Omega_current = X(end,:);

        obj.attitude =
func_quaternion_properize(new_SC_Quaternion_Omega_current(1:4)); %
[quaternion]
        obj.angular_velocity = new_SC_Quaternion_Omega_current(5:7); %
[rad/sec]

        %retrieve cache
        X_dot = func_ode_attitude_dynamics(this_time_array(end),
new_SC_Quaternion_Omega_current', mission, i_SC);
        obj.dot_angular_velocity = X_dot(5:7)';

end

```

[] Methods

```

function obj = func_update_total_wheel_momentum(obj, mission, i_SC)
    % Initialize total wheel momentum as a 3x1 vector
    obj.total_wheel_momentum = zeros(3, 1);

    % Loop through all reaction wheels and add their individual
momentum vectors
    for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
        % Each wheel's total_momentum is already a vector in the body
frame
        if
isfield(mission.true_SC{i_SC}.true_SC_reaction_wheel{i}, 'total_momentum')
&& ...

~isempty(mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.total_momentum)
            obj.total_wheel_momentum = obj.total_wheel_momentum +
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.total_momentum;
        end
    end
end

function obj = func_visualize_attitude(obj,storage_data,
true_SC_body, true_SC_solar_panel,
software_SC_control_attitude,true_SC_micro_thruster_actuator,
true_SC_chemical_thruster_actuator,mission_true_time)

    mArrow3([0 0 0]',[1 0 0]', 'facealpha',
0.1, 'color', 'r', 'stemWidth', 0.01);
    hold on
    mArrow3([0 0 0]',[0 1 0]', 'facealpha',
0.1, 'color', 'g', 'stemWidth', 0.01);

```

```

        mArrow3([0 0 0]',[0 0 1]', 'facealpha',
0.1, 'color', 'b', 'stemWidth', 0.01);

        mArrow3([0 0 0]',obj.rotation_matrix*[0.5 0 0]', 'facealpha',
1, 'color', 'r', 'stemWidth', 0.005);
        mArrow3([0 0 0]',obj.rotation_matrix*[0 0.5 0]', 'facealpha',
1, 'color', 'g', 'stemWidth', 0.005);
        mArrow3([0 0 0]',obj.rotation_matrix*[0 0 0.5]', 'facealpha',
1, 'color', 'b', 'stemWidth', 0.005);

% SC body shape
SC_Shape_Model = [];
SC_Shape_Model.Vertices = (obj.rotation_matrix *
true_SC_body.shape_model.Vertices)';
SC_Shape_Model.Faces = true_SC_body.shape_model.Faces;
patch(SC_Shape_Model, 'FaceColor',0.7*[1 1
1], 'EdgeColor', 'none')

% SP shape
for i=1:true_SC_solar_panel.num_solar_panels
    SP_Shape_Model = [];
    SP_Shape_Model.Vertices = (obj.rotation_matrix *
true_SC_solar_panel.solar_panel_data(i).shape_model.Vertices)';
    SP_Shape_Model.Faces =
true_SC_solar_panel.solar_panel_data(i).shape_model.Faces;
    patch(SP_Shape_Model, 'FaceColor','blue', 'EdgeColor', 'none')
end

% Microthruster
if true_SC_body.flag_hardware_exists.adc_micro_thruster == 1
    for i=1:true_SC_micro_thruster_actuator.num_micro_thruster
        loc = obj.rotation_matrix *
true_SC_micro_thruster_actuator.MT_data(i).location;
        dir = obj.rotation_matrix *
true_SC_micro_thruster_actuator.MT_data(i).orientation;
        scale =
true_SC_micro_thruster_actuator.MT_data(i).commanded_thrust*0.1/
true_SC_micro_thruster_actuator.MT_data(i).maximum_thrust;
        quiver3(loc(1),loc(2),loc(3) ,
dir(1),dir(2),dir(3),"LineWidth",3,"DisplayName",[ 'MT
',num2str(i)],"AutoScaleFactor",scale)
    end
end

% Chemical Thruster
if true_SC_body.flag_hardware_exists.navigation_chemical_thruster
== 1
    for
i=1:true_SC_chemical_thruster_actuator.num_chemical_thruster

```

```

        loc = obj.rotation_matrix *
true_SC_chemical_thruster_actuator.chemical_thruster_data(i).location;
        dir = obj.rotation_matrix *
true_SC_chemical_thruster_actuator.chemical_thruster_data(i).orientation;
        scale =
true_SC_chemical_thruster_actuator.chemical_thruster_data(i).commanded_thrust*0.15/
true_SC_micro_thruster_actuator.MT_data(i).maximum_thrust;
        quiver3(loc(1),loc(2),loc(3) ,
dir(1),dir(2),dir(3),"LineWidth",10,"DisplayName",[ 'CT
',num2str(i)],"AutoScaleFactor",scale)
        end
    end

view([-25,30])
axis equal
xlim([-1 1])
ylim([-1 1])
zlim([-1 1])
if software_SC_control_attitude.actuator_to_use == 1
    info_actuator = "Attitude Actuator in use : MT";
elseif software_SC_control_attitude.actuator_to_use == 2
    info_actuator = "Attitude Actuator in use : RWA";
elseif software_SC_control_attitude.actuator_to_use == 3
    info_actuator = "Attitude Actuator in use : both (DESAT)";
else
    % nothing
end
if software_SC_control_attitude.desired_SC_attitude_mode == 1
    info_pointing = "Pointing for : SB";
elseif software_SC_control_attitude.desired_SC_attitude_mode == 2
    info_pointing = "Pointing for : SUN";
elseif software_SC_control_attitude.desired_SC_attitude_mode == 3
    info_pointing = "Pointing for: DELTA V";
elseif software_SC_control_attitude.desired_SC_attitude_mode == 4
    info_pointing = "Pointing for: DTE";
elseif software_SC_control_attitude.desired_SC_attitude_mode == 5
    info_pointing = "Pointing for: INTERSAT comm";
else
    % nothing
end
if
sum([true_SC_chemical_thruster_actuator.chemical_thruster_data.command_actuation])>0
    info_thrust = "Thruster firing !";
else
    info_thrust = "Thruster OFF";
end
title ([[ 'Mission Simulation, Time
= ',num2str(round(mission_true_time.time)), '
sec'],info_actuator,info_pointing,info_thrust])

light
camlight('headlight')

```

```
        grid on

        xlabel('X_{SUN}')
        ylabel('Y_{SUN}')
        zlabel('Z_{SUN}')

set(gca, 'fontsize', storage_data.plot_parameters.standard_font_size, 'FontName', 'Times
New Roman')
        hold off

        drawnow limitrate

    end

end

end
```

Published with MATLAB® R2022a

3.2 True_SC_Body

Table of Contents

Class: True_SC_Body	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	3
[] Properties: Storage Variables	3
Methods	3
[] Methods: Constructor	3
[] Methods: Store	6
[] Methods: Main	7
[] Methods: Update Mass COM MI	7
[] Methods: Update Mass Class	9

Class: True_SC_Body

Tracks the SC Body

```
classdef True_SC_Body < handle
```

Properties

properties

[] Properties: Initialized Variables

```
name % [string] = 'SC j' for jth SC

mass % : Mass of the SC
% - dry : Dry mass of the SC, that doesn't change position/attitude
% - - mass [kg] : Actual mass
% - - location [m] : Location of this mass in body frame B
% - - MI_over_m [m^2] : Computed once during initialization, shouldn't
change with time (NOTE: NOT MULTIPLIED BY MASS)

% - supplement : Positive/negative mass that is added/removed from the
SC. (e.g. sample collection or projectile) (Optional)

% - # propellant : Propellant mass of the SC that does change value
(Optional)
% Take propellant mass from True_SC_Fuel_Tank

% - # solar_panel : Solar panel mass of the SC, that doesnot change
value (Optional)
% Take solar panel mass from True_SC_Solar_Panel

total_mass % [kg] : Total mass of the SC
```

```

mode_COM_selector % Select which COM to use
% 'given' : Give apriori
% 'update' : Computed by the code

location_COM % [m] : Compute CM from above data

shape_model % : Cell of SC shape models
% - Vertices [m] : Position of vertices in body frame B
% - Faces : Triplet of vertex indices define a face
% - Face_reflectance_factor in [0, 1] : Used for ith face (used for
SRP)
% - r_CM [m] : CM of this shape
% - I_through_r_CM [kg m^2] : Intertia matrix of this shape, about its
CM
% - volume [m^2] : Volume of this shape
% - Face_center [m] : Center of each Face
% - Face_normal [unit vector] : Normal out vector of each face (used
for SRP)
% - Face_area [m^2] : Area of each face
% - type [string] : Type of shape is used for MI and volume
calculations

total_volume % [m^3] : Total volume of the SC

mode_MI_selector % Select which MI to use
% 'given' : Give apriori
% 'update' : Computed by the code

total_MI % [kg m^2] : Total MI of the SC

num_hardware_exists % Data structure that denotes if a hardware exists
on a SC or not (1 >= HW exists, 0 = It doesn't exist)
% Initialized to zero using init_num_hardware_exists.m file
% - num_onboard_clock [integer]
% - num_sun_sensor [integer]
% - num_star_tracker [integer]
% - num_imu [integer]
% - num_micro_thruster [integer]
% - num_reaction_wheel [integer]
% - num_magnetorquer [integer]
% - num_camera [integer]
% - num_chemical_thruster [integer]
% - num_ep_thruster [integer]
% - num_solar_panel [integer]
% - num_RTG [integer]
% - num_battery [integer]
% - num_PDC [integer]
% - num_radio_antenna [integer]
% - num_dte_communication [integer]
% - num_intersat_communication [integer]
% - num_onboard_memory [integer]
% - num_science_radar [integer]
% - num_science_altimeter [integer]
% - num_science_telescope [integer]

```

```
% - num_science_camera [integer]
```

[] Properties: Variables Computed Internally

```
flag_update_SC_body_total_mass_COM_MI % [Boolean] Flag sets if these  
variables should be computed again
```

[] Properties: Storage Variables

```
store
```

```
end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Body(init_data, mission)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['SC ', num2str(init_data.i_SC)];
    end

    % mass
    obj.mass = [];
    obj.mass.dry{1}.mass = 0; % [kg]
    obj.mass.dry{1}.location = [0 0 0]; % [m]
    obj.mass.dry{1}.MI_over_m = zeros(3,3); % [m^2]

    if isfield(init_data.mass, 'supplement')
        obj.mass.supplement = init_data.mass.supplement;
    else
        obj.mass.supplement{1}.mass = 0;
        obj.mass.supplement{1}.location = [0 0 0]; % [m]
        obj.mass.supplement{1}.MI_over_m = zeros(3,3); % [m^2]
    end

    obj.mass.propellant{1}.mass = 0; % [kg]
    obj.mass.propellant{1}.location = [0 0 0]; % [m]
    obj.mass.propellant{1}.MI_over_m = zeros(3,3); % [m^2]

    obj.mass.solar_panel{1}.mass = 0; % [kg]
    obj.mass.solar_panel{1}.location = [0 0 0]; % [m]
    obj.mass.solar_panel{1}.MI_over_m = zeros(3,3); % [m^2]
```

```

obj.mode_COM_selector = init_data.mode_COM_selector;

if strcmp(obj.mode_COM_selector, 'given')
    obj.total_mass = init_data.total_mass; % [kg]
    obj.location_COM = init_data.location_COM; % [m]
    obj.total_volume = init_data.total_volume; % [m^3]
else

    obj.total_mass = 0; % [kg]
    obj.location_COM = [0 0 0]; % [m]
    obj.total_volume = 0; % [m^3]
end

% shape model
obj.shape_model = init_data.shape_model;

for i_shape = 1:length(obj.shape_model)
    % warning('Shape model should be a cell of structs with fields
(r_CM, I_over_m, volume, etc)');
    shape = obj.shape_model{i_shape};

    % Center of mass
    r_CM = mean(shape.Vertices, 1); % [m]

    switch obj.shape_model{i_shape}.type

        case 'cuboid'

            % Inertia matrix (assume cuboid)
            L = max(shape.Vertices(:,1)) -
min(shape.Vertices(:,1)); % [m]
            W = max(shape.Vertices(:,2)) -
min(shape.Vertices(:,2)); % [m]
            H = max(shape.Vertices(:,3)) -
min(shape.Vertices(:,3)); % [m]
            I_through_r_CM = diag([1/12*(W^2+H^2), 1/12*(L^2+H^2),
1/12*(L^2+W^2)]); % [m^2]

            % Volume
            volume = L*W*H; % [m^3]

        otherwise
            error('Havent written yet!')

    end

    % Update
    obj.shape_model{i_shape}.r_CM = r_CM;
    obj.shape_model{i_shape}.I_through_r_CM = I_through_r_CM;
    obj.shape_model{i_shape}.volume = volume;

    % obj.total_volume = obj.total_volume +
obj.shape_model{i_shape}.volume; % [m^3]

```

```

end

% compute face orientation + normal vector + area
for i_shape = 1:length(obj.shape_model)

    shape = obj.shape_model{i_shape};
    Face_center = zeros(size(shape.Faces));
    Face_normal = zeros(size(shape.Faces));
    Face_area = zeros(size(shape.Faces,1),1);

    SC_centroid = mean(shape.Vertices);

    for i=1:size(shape.Faces,1)
        % face center : [(V1x+V2x+V3x)/3 ; (V1y+V2y+V3y)/3 : (V1z
+V2z+V3z)/3 ]
        Face_center(i,:) = [

            (shape.Vertices(shape.Faces(i,1),1)+shape.Vertices(shape.Faces(i,2),1)+shape.Vertices(shape.Faces(i,3),1))/3 ;
            (shape.Vertices(shape.Faces(i,1),2)+shape.Vertices(shape.Faces(i,2),2)+shape.Vertices(shape.Faces(i,3),2))/3 ;
            (shape.Vertices(shape.Faces(i,1),3)+shape.Vertices(shape.Faces(i,2),3)+shape.Vertices(shape.Faces(i,3),3))/3 ]
        % normal vector : cross(V1-V3,V2-V3)
        normal_vector_unsigned = cross(...
            shape.Vertices(shape.Faces(i,1),:)-
shape.Vertices(shape.Faces(i,3),:), ...
            shape.Vertices(shape.Faces(i,2),:)-
shape.Vertices(shape.Faces(i,3),:));
        normal_vector_unsigned = normal_vector_unsigned/
norm(normal_vector_unsigned);
        % correct to get normal pointing outward
        out_vector = (Face_center(i,:)-SC_centroid)/
norm(Face_center(i,:)-SC_centroid); % vector_pointing_out : form SC centroid
to face centroid
        if acos(dot(out_vector,normal_vector_unsigned)) > pi/2
            Face_normal(i,:) = -normal_vector_unsigned;
        else
            Face_normal(i,:) = normal_vector_unsigned;
        end
        % Face area
        vertex_index = shape.Faces(i,:); % index of vertices for
this face
        a = norm(shape.Vertices(vertex_index(1),:) -
shape.Vertices(vertex_index(2),:));
        b = norm(shape.Vertices(vertex_index(2),:) -
shape.Vertices(vertex_index(3),:));
        c = norm(shape.Vertices(vertex_index(3),:) -
shape.Vertices(vertex_index(1),:));
        s = (a+b+c)/2; %
semi perimeter
        Face_area(i) = sqrt(s*(s-a)*(s-b)*(s-c)); % Heron
formula
    end
end

```

```

        obj.shape_model{i_shape}.Face_center = Face_center;
        obj.shape_model{i_shape}.Face_normal = Face_normal;
        obj.shape_model{i_shape}.Face_area = Face_area;
    end

    % Add all HW on SC
    obj.num_hardware_exists = init_data.num_hardware_exists;

    % Moment of Intertia
    obj.mode_MI_selector = init_data.mode_MI_selector;

    if strcmp(obj.mode_MI_selector, 'given')
        obj.total_MI = init_data.total_MI; % [kg m^2]
    end

    % Compute dry mass
    if ~strcmp(obj.mode_MI_selector, 'given')
        for i_shape = 1:length(obj.shape_model)
            obj.mass_dry{i_shape}.mass =
obj.shape_model{i_shape}.mass; % [kg]
            obj.mass_dry{i_shape}.location =
obj.shape_model{i_shape}.r_CM; % [m]
            obj.mass_dry{i_shape}.MI_over_m =
obj.shape_model{i_shape}.I_through_r_CM; % [m^2]
        end

        % Compute total mass and COM and MI
        obj = func_update_SC_body_total_mass_COM_MI(obj);
    end

    % Reset Flag
    obj.flag_update_SC_body_total_mass_COM_MI = 0;

    % Initialize Variables to store: total_mass location_COM total_MI
    obj.store = [];

    obj.store.total_mass = zeros(mission.storage.num_storage_steps,
length(obj.total_mass));
    obj.store.location_COM = zeros(mission.storage.num_storage_steps,
length(obj.location_COM));
    obj.store.total_MI = zeros(mission.storage.num_storage_steps, 9);

    obj = func_update_true_SC_body_store(obj, mission);

end

```

[] Methods: Store

Update the store variable

```
function obj = func_update_true_SC_body_store(obj, mission)
```

```

        if mission.storage.flag_store_this_time_step == 1
            obj.store.total_mass(mission.storage.k_storage,:) =
obj.total_mass; % [kg]
            obj.store.location_COM(mission.storage.k_storage,:) =
obj.location_COM; % [m]
            obj.store.total_MI(mission.storage.k_storage,:) =
reshape(obj.total_MI,1,9); % [kg m^2]
        end
    end
end

```

[] Methods: Main

Main function

```

function obj = func_main_true_SC_body(obj, mission, i_SC)

    % First update fuel and solar panel masses if needed
    if obj.flag_update_SC_body_total_mass_COM_MI == 1
        % Update mass values from fuel tanks and solar panels first
        obj = func_update_SC_body_mass(obj, mission, i_SC);

        % Then compute total mass and COM and MI
        obj = func_update_SC_body_total_mass_COM_MI(obj);
    end

    % Update Store
    obj = func_update_true_SC_body_store(obj, mission);

end

```

[] Methods: Update Mass COM MI

Update total mass and COM

```

function obj = func_update_SC_body_total_mass_COM_MI(obj)

    % Reset Flag
    obj.flag_update_SC_body_total_mass_COM_MI = 0;

    % Update Total Mass and COM
    if strcmp(obj.mode_COM_selector, 'update')

        obj.total_mass = 0; % [kg]
        obj.location_COM = [0 0 0]; % [m]

        for i_mass_class = 1:1:4

            this_mass_class = [];

            switch i_mass_class

                case 1

```

```

        this_mass_class = obj.mass.dry;
    case 2
        this_mass_class = obj.mass.supplement;
    case 3
        this_mass_class = obj.mass.propellant;
    case 4
        this_mass_class = obj.mass.solar_panel;
    otherwise
        error('this_mass_class does not exist!')
    end

    for i=1:length(this_mass_class)
        obj.location_COM = ((obj.total_mass
* obj.location_COM) + (this_mass_class{i}.mass *
this_mass_class{i}.location)); % [kg m]
        obj.total_mass = obj.total_mass +
this_mass_class{i}.mass; % [kg]
        obj.location_COM = (obj.location_COM /
obj.total_mass); % [m]
    end

end

end

% Update MI
if strcmp(obj.mode_MI_selector, 'update')

    obj.total_MI = zeros(3,3); % [kg m^2]

    for i_mass_class = 1:4

        this_mass_class = [];

        switch i_mass_class

            case 1
                this_mass_class = obj.mass.dry;
            case 2
                this_mass_class = obj.mass.supplement;
            case 3
                this_mass_class = obj.mass.propellant;
            case 4
                this_mass_class = obj.mass.solar_panel;
            otherwise
                error('this_mass_class does not exist!')
            end

            for i=1:length(this_mass_class)

                % Displacement vector from the shape CM to the SC CM
                r_CM_i = obj.location_COM -
this_mass_class{i}.location;

```

```

        % Parallel Axis Theorem: I_cm_sc = I_cm_shape + m *
        (r' * r * I_3 - r * r')
        I_i = (this_mass_class{i}.mass *
this_mass_class{i}.MI_over_m) + (this_mass_class{i}.mass * ( (r_CM_i *
r_CM_i' * eye(3)) - (r_CM_i' * r_CM_i)));

        % Add inertia matrix to the total inertia matrix
        obj.total_MI = obj.total_MI + I_i;

    end

end

end

end

```

[] Methods: Update Mass Class

Update the mass class with solar panel and propellant mass

```

function obj = func_update_SC_body_mass(obj, mission, i_SC)

    if isfield(mission.true_SC{i_SC}, 'true_SC_solar_panel')
        for i_SP = 1:1:obj.num_hardware_exists.num_solar_panel
            obj.mass.solar_panel{i_SP}.mass =
mission.true_SC{i_SC}.true_SC_solar_panel{i_SP}.mass; % [kg]
            obj.mass.solar_panel{i_SP}.location =
mission.true_SC{i_SC}.true_SC_solar_panel{i_SP}.shape_model.r_CM; % [m]
            obj.mass.solar_panel{i_SP}.MI_over_m =
mission.true_SC{i_SC}.true_SC_solar_panel{i_SP}.shape_model.I_through_r_CM; %
[kg m^2]
        end
    end

    if isfield(mission.true_SC{i_SC}, 'true_SC_fuel_tank')
        for i_FT = 1:1:obj.num_hardware_exists.num_fuel_tank
            % Update propellant mass from fuel tank
            obj.mass.propellant{i_FT}.mass =
mission.true_SC{i_SC}.true_SC_fuel_tank{i_FT}.instantaneous_fuel_mass; % [kg]
            obj.mass.propellant{i_FT}.location =
mission.true_SC{i_SC}.true_SC_fuel_tank{i_FT}.location; % [m]

            % Calculate approximate moment of inertia for a simple
            cuboid fuel mass
            % This is a simplified approach - for greater accuracy, a
            detailed shape model would be better
            if
~isempty(mission.true_SC{i_SC}.true_SC_fuel_tank{i_FT}.shape_model)
                % If shape model exists, use it
                if
isfield(mission.true_SC{i_SC}.true_SC_fuel_tank{i_FT}.shape_model, 'I_through_r_CM')

```

```

                                obj.mass.propellant{i_FT}.MI_over_m =
mission.true_SC{i_SC}.true_SC_fuel_tank{i_FT}.shape_model.I_through_r_CM;
                                else
                                    % Approximate as cuboid if dimensions available
                                obj.mass.propellant{i_FT}.MI_over_m = zeros(3,3);
                                end
                                else
                                    % Fallback to simple approximation - treat as point
mass with small inertia
                                obj.mass.propellant{i_FT}.MI_over_m = 1e-3 * eye(3); %
[m^2]
                                end

                                % Set flag to update total mass, COM, and MI
                                obj.flag_update_SC_body_total_mass_COM_MI = 1;
                                end
                            end

                        end

                    end

                end
            end
end

```

Published with MATLAB® R2022a

3.3 True_SC_Data_Handling

Table of Contents

Class: True_SC_Data_Handling	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Initialize list_HW_data_generated	3
[] Methods: Initialize list_HW_data_removed	3
[] Methods: Initialize Store	4
[] Methods: Store	5
[] Methods: Main	5
[] Methods: Main Function for Generic	6
[] Methods: Update Instantaneous Data Generated	8
[] Methods: Update Instantaneous Data Removed	9

Class: True_SC_Data_Handling

Tracks the Data Generated onboard the spacecraft

```
classdef True_SC_Data_Handling < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
mode_true_sc_data_handling_selector % [string] Select which Mode to  
run
```

[] Properties: Variables Computed Internally

```
instantaneous_data_change % [kb] Data generated - Data removed over  
mission.true_time.time_step sec
```

```
instantaneous_data_generated % [kb] Data generated by HW and Classes  
over mission.true_time.time_step sec
```

```
instantaneous_data_removed % [kb] Data removed by Communication over  
mission.true_time.time_step sec
```

```
list_HW_data_generated % List of HW and Classes that generates data
```

```
array_HW_data_generated % [kb] Total data generated by this HW and  
Class
```

```
list_HW_data_removed % List of HW and Classes that removes data

array_HW_data_removed % [kb] Total data removed by this HW and Class

warning_counter % [integer] Counter stops the warning after 10
displays

data % Other useful data
```

[] Properties: Storage Variables

```
store

end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Data_Handling(init_data, mission)

    obj.mode_true_SC_data_handling_selector =
init_data.mode_true_SC_data_handling_selector;

    obj.instantaneous_data_change = 0; % [kb]
    obj.instantaneous_data_generated = 0; % [kb]
    obj.instantaneous_data_removed = 0; % [kb]

    obj.list_HW_data_generated = [];
    obj.array_HW_data_generated = [];
    obj.list_HW_data_removed = [];
    obj.array_HW_data_removed = [];

    obj.warning_counter = 0;

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end
    obj.data.store_instantaneous_data_change =
obj.instantaneous_data_change; % [kb]

    % Initialize Variables to store
    obj.store = [];
```

```

        obj.store.instantaneous_data_change =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_change));
        obj.store.instantaneous_data_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_generated));
        obj.store.instantaneous_data_removed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_removed));

end

```

[] Methods: Initialize list_HW_data_generated

Initialize list_HW_data_generated for HW and Classes

```

function obj = func_initialize_list_HW_data_generated(obj, equipment,
mission)

    this_name = equipment.name;
    flag_name_exisits = 0;

    for i = 1:length(obj.list_HW_data_generated)
        if strcmp( obj.list_HW_data_generated{i}, this_name )
            flag_name_exisits = 1;
        end
    end

    if flag_name_exisits == 0
        i = length(obj.list_HW_data_generated);
        obj.list_HW_data_generated{i+1} = this_name;

        if isprop(equipment, 'instantaneous_data_rate_generated')
            this_instantaneous_data_generated
= (equipment.instantaneous_data_rate_generated *
mission.true_time.time_step); % [kb]
        elseif
isprop(equipment, 'instantaneous_data_generated_per_sample')
            this_instantaneous_data_generated =
equipment.instantaneous_data_generated_per_sample; % [kb]
        else
            error('Data generated incorrect!')
        end

        obj.array_HW_data_generated(1,i+1) =
this_instantaneous_data_generated; % [kb]
    end

end

```

[] Methods: Initialize list_HW_data_removed

Initialize list_HW_data_removed for HW and Classes

```

function obj = func_initialize_list_HW_data_removed(obj, equipment,
mission)

    this_name = equipment.name;
    flag_name_exisits = 0;

    for i = 1:length(obj.list_HW_data_removed)
        if strcmp( obj.list_HW_data_removed{i}, this_name )
            flag_name_exisits = 1;
        end
    end

    if flag_name_exisits == 0
        i = length(obj.list_HW_data_removed);
        obj.list_HW_data_removed{i+1} = this_name;

        if isprop(equipment, 'instantaneous_data_rate_removed')
            this_instantaneous_data_removed =
(equipment.instantaneous_data_rate_removed * mission.true_time.time_step); %
[kb]
        elseif
isprop(equipment, 'instantaneous_data_removed_per_sample')
            this_instantaneous_data_removed =
equipment.instantaneous_data_removed_per_sample; % [kb]
        else
            error('Data removed incorrect!')
        end

        obj.array_HW_data_removed(1,i+1) =
this_instantaneous_data_removed; % [kb]
    end

end

```

[] Methods: Initialize Store

Initialize store of array_HW_data_generated and array_HW_data_removed

```

function obj = func_initialize_store_HW_data_generated_removed(obj,
mission)

    obj.store.list_HW_data_generated = obj.list_HW_data_generated;
    obj.store.list_HW_data_removed = obj.list_HW_data_removed;

    obj.store.array_HW_data_generated =
zeros(mission.storage.num_storage_steps,
length(obj.array_HW_data_generated));
    obj.store.array_HW_data_removed =
zeros(mission.storage.num_storage_steps, length(obj.array_HW_data_removed));

    obj = func_update_true_SC_data_store(obj, mission);

end

```

[] Methods: Store

Update the store variable

```
function obj = func_update_true_SC_data_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1

obj.store.instantaneous_data_change(mission.storage.k_storage,:) =
obj.data.store_instantaneous_data_change; % [kb]

obj.store.instantaneous_data_generated(mission.storage.k_storage,:) =
obj.instantaneous_data_generated; % [kb]

obj.store.instantaneous_data_removed(mission.storage.k_storage,:) =
obj.instantaneous_data_removed; % [kb]

        obj.store.array_HW_data_generated(mission.storage.k_storage,:)
= obj.array_HW_data_generated; % [kb]

        if isempty(obj.array_HW_data_removed)
            % Do nothing!
        else

obj.store.array_HW_data_removed(mission.storage.k_storage,:) =
obj.array_HW_data_removed; % [kb]
            end
        end

    end

end
```

[] Methods: Main

Main data handling code

```
function obj = func_main_true_SC_data_handling(obj, mission, i_SC)

    switch obj.mode_true_SC_data_handling_selector

        case 'Generic'
            obj = func_true_SC_data_handling_Generic(obj, mission,
i_SC);

        case 'Nightingale'
            obj = func_true_SC_data_handling_Nightingale(obj, mission,
i_SC);

        otherwise
            error('Data Handling mode not defined!')
        end

    end

    % Store
```

```

obj = func_update_true_SC_data_store(obj, mission);

% Reset All Variables
obj.instantaneous_data_change = 0; % [kb]
obj.instantaneous_data_generated = 0; % [kb]
obj.instantaneous_data_removed = 0; % [kb]

end

```

[] Methods: Main Function for Generic

Generic data handling code

```

function obj = func_true_SC_data_handling_Generic(obj, mission, i_SC)

    obj.instantaneous_data_change = (obj.instantaneous_data_generated
- obj.instantaneous_data_removed); % [kb]
    obj.data.store_instantaneous_data_change =
obj.instantaneous_data_change; % [kb]

    if obj.instantaneous_data_change > 0
        % Add Data to Memory

        for i_memory =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_memory

            if
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
>= mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.maximum_capacity
                % Skip this memory

            elseif
(mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.maximum_capacity -
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity)
<= obj.instantaneous_data_change
                % Fill this memory as much as possible
                obj.instantaneous_data_change =
obj.instantaneous_data_change -
(mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.maximum_capacity -
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity); %
[kb]

mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
= mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.maximum_capacity; %
[kb]

            else
                % Put entirely in this memory

mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
=
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
+ obj.instantaneous_data_change; % [kb]

```

```

        obj.instantaneous_data_change = 0; % [kb]

    end

end

if obj.instantaneous_data_change > 0

    if obj.warning_counter < 10
        warning('All Memories are Full!')
        obj.warning_counter = obj.warning_counter + 1;
    end

mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
=
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
+ obj.instantaneous_data_change; % [kb]
        obj.instantaneous_data_change = 0; % [kb]
    else
        obj.warning_counter = 0;
    end

else
    % Remove Data from Memory and obj.instantaneous_data_change <
0

    for i_memory =
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_memory:-1:1

        if
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
<= 0

            % Skip this memory

        elseif
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
< abs(obj.instantaneous_data_change)

            % Delete all this memory
            obj.instantaneous_data_change =
obj.instantaneous_data_change +
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity;

mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
= 0; % [kb]

        else
            % Remove some of this memory

mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
=
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
+ obj.instantaneous_data_change; % [kb]

```

```

        obj.instantaneous_data_change = 0; % [kb]

    end

end

if obj.instantaneous_data_change < 0

    if obj.warning_counter < 10
        warning('All Memories are Empty!')
        obj.warning_counter = obj.warning_counter + 1;
    end

mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
=
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity
+ obj.instantaneous_data_change; % [kb]
        obj.instantaneous_data_change = 0; % [kb]
    else
        obj.warning_counter = 0;
    end

end

end

end

```

[] Methods: Update Instantaneous Data Generated

Updates instantaneous_data_generated by all HW and Classes

```

function obj = func_update_instantaneous_data_generated(obj,
equipment, mission, varargin)

    if isprop(equipment, 'instantaneous_data_rate_generated')
        this_instantaneous_data_generated
= (equipment.instantaneous_data_rate_generated *
mission.true_time.time_step); % [kb]
    elseif
isprop(equipment, 'instantaneous_data_generated_per_sample')
        this_instantaneous_data_generated =
equipment.instantaneous_data_generated_per_sample; % [kb]
    else
        error('Data generated incorrect!')
    end

    if ~isempty(varargin)
        chosen_memory = varargin{1};
        i_SC = varargin{2};
    end

```

```

mission.true_SC{i_SC}.true_SC_onboard_memory{chosen_memory}.instantaneous_capacity
=
mission.true_SC{i_SC}.true_SC_onboard_memory{chosen_memory}.instantaneous_capacity
+ this_instantaneous_data_generated; % [kb]
    else
        obj.instantaneous_data_generated =
obj.instantaneous_data_generated + this_instantaneous_data_generated; % [kb]
    end

    this_name = equipment.name;
    flag_name_exisits = 0;
    this_idx = 0;

    for i = 1:1:length(obj.list_HW_data_generated)
        if strcmp( obj.list_HW_data_generated{i}, this_name )
            flag_name_exisits = 1;
            this_idx = i;
        end
    end

    if flag_name_exisits == 0
        error('HW not found!')
    else
        obj.array_HW_data_generated(1,this_idx) =
obj.array_HW_data_generated(1,this_idx) +
this_instantaneous_data_generated; % [kb]
    end

end

```

[] Methods: Update Instantaneous Data Re-moved

Updates instantaneous_data_removed by all HW and Classes

```

function obj = func_update_instantaneous_data_removed(obj, equipment,
mission, varargin)

    if isprop(equipment, 'instantaneous_data_rate_removed')
        this_instantaneous_data_removed =
(equipment.instantaneous_data_rate_removed * mission.true_time.time_step); %
[kb]
    elseif isprop(equipment, 'instantaneous_data_removed_per_sample')
        this_instantaneous_data_removed =
equipment.instantaneous_data_removed_per_sample; % [kb]
    else
        error('Data removed incorrect!')
    end

    if ~isempty(varargin)
        chosen_memory = varargin{1};
    end

```

```

        i_SC = varargin{2};

mission.true_SC{i_SC}.true_SC_onboard_memory{chosen_memory}.instantaneous_capacity
=
mission.true_SC{i_SC}.true_SC_onboard_memory{chosen_memory}.instantaneous_capacity
- this_instantaneous_data_removed; % [kb]
    else
        obj.instantaneous_data_removed =
obj.instantaneous_data_removed + this_instantaneous_data_removed; % [kb]
    end

    this_name = equipment.name;
    flag_name_exisits = 0;
    this_idx = 0;

    for i = 1:1:length(obj.list_HW_data_removed)
        if strcmp( obj.list_HW_data_removed{i}, this_name )
            flag_name_exisits = 1;
            this_idx = i;
        end
    end

    if flag_name_exisits == 0
        error('HW not found!')
    else
        obj.array_HW_data_removed(1,this_idx) =
obj.array_HW_data_removed(1,this_idx) + this_instantaneous_data_removed; %
[kb]
    end

end

end

end

```

Published with MATLAB® R2022a

3.4 True_SC_Navigation

Table of Contents

Class: True_SC_Navigation	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	4
[] Methods: Main	5
[] Methods: Update Position Velocity SPICE	5
[] Methods: Update Position Velocity Absolute Dynamics	6
Methods: Update Visible Sun Earth	7
Methods: Check SC Crashed	9

Class: True_SC_Navigation

Tracks the position and velocity of the SC

```
classdef True_SC_Navigation < handle
```

Properties

properties

[] Properties: Initialized Variables

```
position % [km] : Current position of SC in inertial frame I
velocity % [km/sec] : Current velocity of SC in inertial frame I
position_relative_target % [km] : Current position of SC relative to
SB-center J2000 inertial frame
velocity_relative_target % [km/sec] : Current velocity of SC relative
to SB-center J2000 inertial frame
name_relative_target % [string] : Name of the target, relative to
which position and velocity are specified

spice_filename % [string] : SC's SPICE FileName
spice_name % [string] : SC's SPICE Name

mode_true_sc_navigation_dynamics_selector % [string] Different
navigation dynamics modes
% - 'SPICE' : Use pre-computed SPICE trajectory
% - 'Absolute Dynamics'
% - 'Relative Dynamics'
```

[] Properties: Variables Computed Internally

```
index_relative_target % [integer] : Index of the target, relative to
which position and velocity are specified
```

```

flag_visible_Sun % [Boolean] Check if Sun is visible
flag_visible_Earth % [Boolean] Check if Earth is visible

flag_SC_crashed % [Boolean] Check if SC has crashed into anything!
position_array % [km] Position array wrt Sun-centered J2000,
corresponding to time array in mission.true_time.time_position_array

control_force % [N] : Control force vector generated by thrusters
(e.g. MT, CT) that passes through Center of Mass of SC
disturbance_force % [N] : Disturbance force vector generated by SRP
that passes through Center of Mass of SC

ode_options % : Options for Matlab's ODE function
odeset( 'RelTol',1e-14,'AbsTol',1e-14)

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_SC_Navigation(init_data, mission)

    if strcmp(mission.frame,'Absolute') ||
isfield(init_data, 'position')
        obj.position = init_data.position; % [km]
        obj.velocity = init_data.velocity; % [km/sec]

        if isfield(init_data, 'name_relative_target')
            obj.name_relative_target = init_data.name_relative_target;
        else
            obj.name_relative_target = mission.true_target{1}.name;
        end

        for i_target = 1:1:mission.num_target
            if strcmp(mission.true_target{i_target}.name,
obj.name_relative_target)
                obj.index_relative_target = i_target;
                position_target =
mission.true_target{i_target}.position;
                velocity_target =
mission.true_target{i_target}.velocity;
            end
        end
    end
end

```

```

        end
    end

    obj.position_relative_target = obj.position -
position_target; % [km]
    obj.velocity_relative_target = obj.velocity -
velocity_target; % [km/sec]

    elseif strcmp(mission.frame,'Relative') ||
isfield(init_data, 'position_relative_target')
        obj.position_relative_target =
sc_body_init_data.position_relative_target; % [km]
        obj.velocity_relative_target =
sc_body_init_data.velocity_relative_target; % [km/sec]
        obj.name_relative_target = init_data.name_relative_target; %
[string]

        for i_target = 1:1:mission.num_target
            if strcmp(mission.true_target{i_target}.name,
obj.name_relative_target)
                obj.index_relative_target = i_target;
                position_target =
mission.true_target{i_target}.position;
                velocity_target =
mission.true_target{i_target}.velocity;
            end
        end

        obj.position = position_target +
obj.position_relative_target; % [km]
        obj.velocity = velocity_target +
obj.velocity_relative_target; % [km/sec]

    else
        error('Navigation initialization incorrect!')
    end

    obj.control_force = [0 0 0]; % [N]
    obj.disturbance_force = [0 0 0]; % [N]
    obj.flag_SC_crashed = 0;

    % Select Dynamics Mode
    obj.mode_true_SC_navigation_dynamics_selector =
init_data.mode_true_SC_navigation_dynamics_selector; % [string]

    if strcmp( obj.mode_true_SC_navigation_dynamics_selector, 'SPICE'
)
        % Use SPICE trajectory
        obj.spice_filename = init_data.spice_filename;
        obj.spice_name = init_data.spice_name;
        cspice_furnsh(obj.spice_filename)
    end

```

```

obj.ode_options = odeset('RelTol',1e-14,'AbsTol',1e-14);

% Update flag visible
obj = func_update_visible_Sun_Earth(obj, mission);

% Initialize Variables to store: position and velocity of SC
obj.store = [];

obj.store.position = zeros(mission.storage.num_storage_steps,
length(obj.position));
obj.store.velocity = zeros(mission.storage.num_storage_steps,
length(obj.velocity));
obj.store.position_relative_target =
zeros(mission.storage.num_storage_steps,
length(obj.position_relative_target));
obj.store.velocity_relative_target =
zeros(mission.storage.num_storage_steps,
length(obj.velocity_relative_target));
obj.store.flag_visible_Sun =
zeros(mission.storage.num_storage_steps, length(obj.flag_visible_Sun));
obj.store.flag_visible_Earth =
zeros(mission.storage.num_storage_steps, length(obj.flag_visible_Earth));

obj = func_update_true_SC_navigation_store(obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_navigation_store(obj, mission)

if mission.storage.flag_store_this_time_step == 1
    obj.store.position(mission.storage.k_storage,:) =
obj.position; % [km]
    obj.store.velocity(mission.storage.k_storage,:) =
obj.velocity; % [km/sec]

obj.store.position_relative_target(mission.storage.k_storage,:) =
obj.position_relative_target; % [km]

obj.store.velocity_relative_target(mission.storage.k_storage,:) =
obj.velocity_relative_target; % [km/sec]
    obj.store.flag_visible_Sun(mission.storage.k_storage,:) =
obj.flag_visible_Sun; % [Boolean]
    obj.store.flag_visible_Earth(mission.storage.k_storage,:) =
obj.flag_visible_Earth; % [Boolean]
end

end

```

[] Methods: Main

Select method to update position and velocity

```
function obj = func_main_true_SC_navigation(obj, mission, i_SC)

    switch obj.mode_true_SC_navigation_dynamics_selector

        case 'SPICE'
            % Use SPICE trajectory
            obj =
func_update_SC_navigation_position_velocity_SPICE(obj, mission);

        case 'Absolute Dynamics'
            % Use Absolute Dynamics
            obj =
func_update_SC_navigation_position_velocity_Absolute_Dynamics(obj, mission,
i_SC);

        otherwise
            error('Navigation mode not defined!')
    end

    % Update flag visible
    obj = func_update_visible_Sun_Earth(obj, mission);

    % Update SC crash
    obj = func_update_flag_SC_crashed(obj, mission, i_SC);

    % Store
    obj = func_update_true_SC_navigation_store(obj, mission);

    obj.control_force = [0 0 0]; % [N]
    obj.disturbance_force = [0 0 0]; % [N]

end
```

[] Methods: Update Position Velocity SPICE

Update position and velocity using SPICE

```
function obj = func_update_SC_navigation_position_velocity_SPICE(obj,
mission)

    SC_pos_vel =
cspice_spekr(obj.spice_name,mission.true_time.date,'J2000','NONE','SUN');

    obj.position = SC_pos_vel(1:3)'; % [km]
    obj.velocity = SC_pos_vel(4:6)'; % [km/sec]

    position_target =
mission.true_target{obj.index_relative_target}.position;
```

```

        velocity_target =
mission.true_target{obj.index_relative_target}.velocity;

        obj.position_relative_target = obj.position - position_target; %
[km]
        obj.velocity_relative_target = obj.velocity - velocity_target; %
[km/sec]

        SC_pos_vel_array = (cspice_spkezr(obj.spice_name,
mission.true_time.prev_date +
mission.true_time.time_position_array' , 'J2000', 'NONE', 'SUN'))';
        obj.position_array = SC_pos_vel_array(:,1:3);

end

```

[] Methods: Update Position Velocity Absolute Dynamics

Update position and velocity using Absolute dynamics

```

function obj =
func_update_SC_navigation_position_velocity_Absolute_Dynamics(obj, mission,
i_SC)

    this_time_array = mission.true_time.time_position_array;

    obj = obj.func_update_disturbance_force(mission, i_SC);

    SC_pos_vel_current = [obj.position'; obj.velocity']; % [km, km/
sec] in (6,1) format
    [T,X]=ode113(@(t,X) func_ode_orbit_inertial_absolute_dynamics(t,
X, mission, i_SC), this_time_array, SC_pos_vel_current, obj.ode_options);

    new_SC_pos_vel_current = X(end,:);

    obj.position = new_SC_pos_vel_current(1:3); % [km]
    obj.velocity = new_SC_pos_vel_current(4:6); % [km/sec]

    position_target =
mission.true_target{obj.index_relative_target}.position;
    velocity_target =
mission.true_target{obj.index_relative_target}.velocity;

    obj.position_relative_target = obj.position - position_target; %
[km]
    obj.velocity_relative_target = obj.velocity - velocity_target; %
[km/sec]

    obj.position_array = X(:,1:3); % [km]

end

```

```

function obj = func_update_disturbance_force(obj,mission, i_SC)
    % Sum up all disturbance forces
    obj.disturbance_force =
mission.true_SC{i_SC}.true_gravity_gradient.disturbance_force_G2' +
mission.true_SC{i_SC}.true_SRP.disturbance_force_SRP';
end

```

Methods: Update Visible Sun Earth

Update flag_visible_Sun and flag_visible_Earth

```

function obj = func_update_visible_Sun_Earth(obj, mission)

    % Use lins math: https://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html

    for i_vis = 1:1:2

        flag_visible = 1;

        if i_vis == 1
            % Sun
            x1 =
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.position; %
[km]

            elseif i_vis == 2
                % Earth
                x1 =
mission.true_solar_system.SS_body{mission.true_solar_system.index_Earth}.position; %
[km]

            else
                error('Shouldnt reach here')
            end

            % SC Position
            x2 = obj.position; % [km]

            % Check all Targets
            for i_target = 1:1:mission.num_target

                x0 = mission.true_target{i_target}.position; % [km]

                d = norm(cross(x0 - x1, x0 - x2))/norm(x2 - x1);
                if d >= mission.true_target{i_target}.radius % [km]
                    % No chance of eclipse
                else
                    % Check t
                    t = - dot(x1 - x0, x2 - x1) / (norm(x2 - x1))^2;
                    if (t >= 0) && (t <= 1)
                        flag_visible = 0;
                    else
                        % Body is outside the line of sight

```

```

        end
    end
end

% Check all SS Bodies
for i_SS_body = 1:1:mission.true_solar_system.num_SS_body

    if (i_vis == 1) && (i_SS_body ==
mission.true_solar_system.index_Sun)
        % Skip this!

    elseif (i_vis == 2) && (i_SS_body ==
mission.true_solar_system.index_Earth)
        % Skip this!

    else

        x0 =
mission.true_solar_system.SS_body{i_SS_body}.position; % [km]

        d = norm(cross(x0 - x1, x0 - x2))/norm(x2 - x1);
        if d >=
mission.true_solar_system.SS_body{i_SS_body}.radius % [km]
            % No chance of eclipse
        else
            % Check t
            t = - dot(x1 - x0, x2 - x1) / (norm(x2 - x1))^2;
            if (t >= 0) && (t <= 1)
                flag_visible = 0;
            else
                % Body is outside the line of sight
            end
        end
    end

end

end

if i_vis == 1
    % Sun
    obj.flag_visible_Sun = flag_visible;
elseif i_vis == 2
    % Earth
    obj.flag_visible_Earth = flag_visible;
else
    error('Shouldnt reach here')
end

end

end
end

```

Methods: Check SC Crashed

Update flag_SC_crashed

```
function obj = func_update_flag_SC_crashed(obj, mission, i_SC)

    for t = 1:1:length(mission.true_time.time_position_array)

        % SC position
        x1 = obj.position_array(t,:);

        % Check all Targets
        for i_target = 1:1:mission.num_target

            x0 = mission.true_target{i_target}.position_array(t,:); %
[ km]

            if norm(x0 - x1) <= mission.true_target{i_target}.radius %
[ km]

                obj.flag_SC_crashed = 1;

disp('-----')
                disp(['SC ',num2str(i_SC),' crashed into
',mission.true_target{i_target}.name])

disp('-----')
                mission.storage.flag_stop_sim = 1;
                break;
            end
        end
    end
end
end
end
end
end
end
end
```

Published with MATLAB® R2022a

3.5 True_SC_Power

Table of Contents

Class: True_SC_Power	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Initialize list_HW_energy_consumed	3
[] Methods: Initialize list_HW_energy_generated	4
[] Methods: Initialize Store	4
[] Methods: Store	4
[] Methods: Main	5
[] Methods: Update Instantaneous Power Consumed	8
[] Methods: Update Instantaneous Power Consumed Attitude	8
[] Methods: Update Instantaneous Power Generated	9

Class: True_SC_Power

Track the power status onboard the spacecraft

```
classdef True_SC_Power < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
power_loss_rate % [float] Fraction of the total power loss: Power loss  
[W] = (1 + power_loss_rate) * total_instantaneous_power_consumed
```

[] Properties: Variables Computed Internally

```
name % [string] 'Power Subsystem'  
  
instantaneous_total_power_consumed % [W] Total power consumed by all  
sensors and actuators over mission.true_time.time_step sec  
  
instantaneous_total_power_generated % [W] Total power generated by  
solar panels or RTG over mission.true_time.time_step sec  
  
instantaneous_energy % [W hr] Converted into Energy for storage in  
battery  
  
instantaneous_energy_unused % [W hr] Excess Energy, that is usually  
going to heat the SC  
  
list_HW_energy_consumed % List of HW that consumes power
```

```

list_HW_energy_generated % List of HW that generates power

array_HW_energy_consumed % [W hr] Total energy consumed by this HW

array_HW_energy_generated % [W hr] Total energy generated by this HW

warning_counter % [integer] Counter stops the warning after 10
displays

power_emergency % [boolean] Flag to indicate a critical power deficit
when batteries are empty

power_deficit % [W hr] Track power deficit when batteries are empty

data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_SC_Power(init_data, mission)

    obj.instantaneous_total_power_consumed = 0; % [W]
    obj.instantaneous_total_power_generated = 0; % [W]
    obj.instantaneous_energy = 0; % [W hr]
    obj.instantaneous_energy_unused = 0; % [W hr]

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = 'Power Subsystem';
    end

    obj.power_loss_rate = init_data.power_loss_rate; % [float]

    obj.list_HW_energy_consumed = [];
    obj.list_HW_energy_generated = [];
    obj.array_HW_energy_consumed = [];
    obj.array_HW_energy_generated = [];
    obj.warning_counter = 0;
    obj.power_emergency = false;
    obj.power_deficit = 0;

```

```

        if isfield(init_data, 'data')
            obj.data = init_data.data;
        else
            obj.data = [];
        end
        obj.data.store_instantaneous_energy = obj.instantaneous_energy; %
[W hr]

% Initialize Variables to store: power and energy
obj.store = [];

        obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_total_power_consumed));
        obj.store.instantaneous_power_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_total_power_generated));
        obj.store.instantaneous_energy =
zeros(mission.storage.num_storage_steps, length(obj.instantaneous_energy));
        obj.store.instantaneous_energy_unused =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_energy_unused));

    end

```

[] Methods: Initialize list_HW_energy_consumed

Initialize list_HW_energy_consumed for all HW

```

function obj = func_initialize_list_HW_energy_consumed(obj, equipment,
mission)

    this_name = equipment.name;
    flag_name_exists = 0;

    for i = 1:length(obj.list_HW_energy_consumed)
        if strcmp( obj.list_HW_energy_consumed{i}, this_name )
            flag_name_exists = 1;
        end
    end

    if flag_name_exists == 0
        i = length(obj.list_HW_energy_consumed);
        obj.list_HW_energy_consumed{i+1} = this_name;
        obj.array_HW_energy_consumed(1,i
+1) = equipment.instantaneous_power_consumed *
(mission.true_time.time_step/3600); % [W hr]
    end

end

```

[] Methods: Initialize list_HW_energy_generated

Initialize list_HW_energy_generated for Solar Panels or RTG

```
function obj = func_initialize_list_HW_energy_generated(obj,
equipment, mission)

    this_name = equipment.name;
    flag_name_exists = 0;

    for i = 1:length(obj.list_HW_energy_generated)
        if strcmp( obj.list_HW_energy_generated{i}, this_name )
            flag_name_exists = 1;
        end
    end

    if flag_name_exists == 0
        i = length(obj.list_HW_energy_generated);
        obj.list_HW_energy_generated{i+1} = this_name;
        obj.array_HW_energy_generated(1,i
+1) = equipment.instantaneous_power_generated *
(mission.true_time.time_step/3600); % [W hr]
    end

end
```

[] Methods: Initialize Store

Initialize store of array_HW_energy_consumed and array_HW_energy_generated

```
function obj = func_initialize_store_HW_power_consumed_generated(obj,
mission)

    obj.store.list_HW_energy_consumed = obj.list_HW_energy_consumed;
    obj.store.list_HW_energy_generated = obj.list_HW_energy_generated;

    obj.store.array_HW_energy_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.array_HW_energy_consumed));
    obj.store.array_HW_energy_generated =
zeros(mission.storage.num_storage_steps,
length(obj.array_HW_energy_generated));

    obj = func_update_true_SC_power_store(obj, mission);

end
```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_power_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1

obj.store.instantaneous_power_consumed(mission.storage.k_storage,:) =
obj.instantaneous_total_power_consumed; % [W]

obj.store.instantaneous_power_generated(mission.storage.k_storage,:) =
obj.instantaneous_total_power_generated; % [W]
        obj.store.instantaneous_energy(mission.storage.k_storage,:) =
obj.data.store_instantaneous_energy; % [W hr]

obj.store.instantaneous_energy_unused(mission.storage.k_storage,:) =
obj.instantaneous_energy_unused; % [W hr]

obj.store.array_HW_energy_consumed(mission.storage.k_storage,:) =
obj.array_HW_energy_consumed; % [W hr]

obj.store.array_HW_energy_generated(mission.storage.k_storage,:) =
obj.array_HW_energy_generated; % [W hr]
    end

end

```

[] Methods: Main

Main power code

```

function obj = func_main_true_SC_power(obj, mission, i_SC)

    obj.instantaneous_energy =
(obj.instantaneous_total_power_generated -
obj.instantaneous_total_power_consumed) *
(mission.true_time.time_step/3600); % [W hr]
    obj.data.store_instantaneous_energy = obj.instantaneous_energy; %
[W hr]

    if obj.instantaneous_energy > 0
        % Excess Energy use to Recharge Battery
        obj.power_emergency = false; % Clear emergency flag when we
have excess energy

        for i_batt =
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_battery:-1:1

            if (mission.true_SC{i_SC}.true_SC_battery{i_batt}.health
== 1) && (obj.instantaneous_energy > 0)

                if
mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity >=
mission.true_SC{i_SC}.true_SC_battery{i_batt}.maximum_capacity

```

```

        % Do nothing!

        elseif
(mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity
+ (mission.true_SC{i_SC}.true_SC_battery{i_batt}.charging_efficiency *
obj.instantaneous_energy)) <=
mission.true_SC{i_SC}.true_SC_battery{i_batt}.maximum_capacity
        % Charge this Battery

mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity
= mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity
+ (mission.true_SC{i_SC}.true_SC_battery{i_batt}.charging_efficiency *
obj.instantaneous_energy); % [W hr]
        obj.instantaneous_energy = 0; % [W hr]

        else
        % Fill this Battery
        obj.instantaneous_energy =
obj.instantaneous_energy -
(mission.true_SC{i_SC}.true_SC_battery{i_batt}.maximum_capacity -
mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity)/
mission.true_SC{i_SC}.true_SC_battery{i_batt}.charging_efficiency; % [W hr]

mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity =
mission.true_SC{i_SC}.true_SC_battery{i_batt}.maximum_capacity; % [W hr]

        end

        end

        end

        if obj.instantaneous_energy > 0
            obj.instantaneous_energy_unused =
obj.instantaneous_energy; % [W hr]
        end

        else
            % Discharge Battery (obj.instantaneous_energy < 0)

            for i_batt =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_battery

                if (mission.true_SC{i_SC}.true_SC_battery{i_batt}.health
== 1) && (obj.instantaneous_energy < 0)

                    if
mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity <= 0
                        % Do nothing!

                    elseif
(mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity *
mission.true_SC{i_SC}.true_SC_battery{i_batt}.discharging_efficiency) >=
abs(obj.instantaneous_energy)

```

```

                                % Lots of extra Energy

mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity =
mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity -
(abs(obj.instantaneous_energy) /
mission.true_SC{i_SC}.true_SC_battery{i_batt}.discharging_efficiency); % [W
hr]

                                obj.instantaneous_energy = 0;

                                else
                                    % Empty this Battery!
                                    obj.instantaneous_energy =
obj.instantaneous_energy +
(mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity *
mission.true_SC{i_SC}.true_SC_battery{i_batt}.discharging_efficiency); % [W
hr]

mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity = 0; %
[W hr]

                                end

                                end

                                end

                                if obj.instantaneous_energy < 0
                                    % All batteries are empty but we still need power
                                    % Set power emergency flag and calculate the deficit
                                    obj.power_emergency = true;
                                    obj.power_deficit = abs(obj.instantaneous_energy); % Track
the deficit

                                    % Limit the number of warnings displayed
                                    if obj.warning_counter < 10
                                        warning('All Batteries are Empty! Power deficit: %0.2f
W-hr', obj.power_deficit);
                                        obj.warning_counter = obj.warning_counter + 1;
                                    end

                                    % Reset instantaneous_energy to 0 to prevent negative
values
                                    obj.instantaneous_energy = 0;
                                else
                                    obj.power_emergency = false;
                                    obj.warning_counter = 0;
                                    obj.power_deficit = 0;
                                end

                                end

                                % Store
                                obj = func_update_true_SC_power_store(obj, mission);

                                % Reset All Variables

```

```

obj.instantaneous_total_power_consumed = 0; % [W]
obj.instantaneous_total_power_generated = 0; % [W]
obj.instantaneous_energy = 0; % [W hr]
obj.instantaneous_energy_unused = 0; % [W hr]

end

```

[] Methods: Update Instantaneous Power Consumed

Updates instantaneous_power_consumed by all HW

```

function obj = func_update_instantaneous_power_consumed(obj,
equipment, mission)

    obj.instantaneous_total_power_consumed =
obj.instantaneous_total_power_consumed +
equipment.instantaneous_power_consumed * (1 + obj.power_loss_rate); % [W]

    this_name = equipment.name;
    flag_name_exists = 0;
    this_idx = 0;

    for i = 1:length(obj.list_HW_energy_consumed)
        if strcmp( obj.list_HW_energy_consumed{i}, this_name )
            flag_name_exists = 1;
            this_idx = i;
        end
    end

    if flag_name_exists == 0
        error('HW not found!')
    else
        obj.array_HW_energy_consumed(1,this_idx)
= obj.array_HW_energy_consumed(1,this_idx) +
(equipment.instantaneous_power_consumed*(mission.true_time.time_step/3600)); %
[W hr]
    end
end

```

[] Methods: Update Instantaneous Power Consumed Attitude

Updates instantaneous_power_consumed by all HW, within ADL

```

function obj = func_update_instantaneous_power_consumed_attitude(obj,
equipment, mission)
    obj.instantaneous_total_power_consumed =
obj.instantaneous_total_power_consumed +

```

```

equipment.instantaneous_power_consumed*(mission.true_time.time_step_attitude/
mission.true_time.time_step); % [W]

    this_name = equipment.name;
    flag_name_exists = 0;
    this_idx = 0;

    for i = 1:1:length(obj.list_HW_energy_consumed)
        if strcmp( obj.list_HW_energy_consumed{i}, this_name )
            flag_name_exists = 1;
            this_idx = i;
        end
    end

    if flag_name_exists == 0
        error('HW not found!')
    else
        obj.array_HW_energy_consumed(1,this_idx)
= obj.array_HW_energy_consumed(1,this_idx) +
(equipment.instantaneous_power_consumed*(mission.true_time.time_step_attitude/3600)); %
[W hr]
    end

end

```

[] Methods: Update Instantaneous Power Generated

Updates instantaneous_power_generated by Solar Panels or RTG

```

function obj = func_update_instantaneous_power_generated(obj,
equipment, mission)
    obj.instantaneous_total_power_generated =
obj.instantaneous_total_power_generated +
equipment.instantaneous_power_generated; % [W]

    this_name = equipment.name;
    flag_name_exists = 0;
    this_idx = 0;

    for i = 1:1:length(obj.list_HW_energy_generated)
        if strcmp( obj.list_HW_energy_generated{i}, this_name )
            flag_name_exists = 1;
            this_idx = i;
        end
    end

    if flag_name_exists == 0
        error('HW not found!')
    else
        obj.array_HW_energy_generated(1,this_idx)
= obj.array_HW_energy_generated(1,this_idx) +

```

```
(equipment.instantaneous_power_generated*(mission.true_time.time_step/3600)); %  
[w hr]  
    end  
  
    end  
  
end  
  
end
```

Published with MATLAB® R2022a

Chapter 4

SC Sensors and Actuators Classes

4.1 True_SC_Battery

Table of Contents

Class: True_SC_Battery	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3

Class: True_SC_Battery

Tracks the Battery state of charge

```
classdef True_SC_Battery < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate generated
during current time step, in kilo bits (kb) per sec

maximum_capacity % [Watts * hr] : Maximum energy storage capacity of
the battery
```

[] Properties: Variables Computed Internally

```
name % [string] 'Batt i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

instantaneous_capacity % [Watts * hr] : Instantaneous capacity of
battery

state_of_charge % [percentage] : SoC is defined by = 100×
instantaneous_capacity / maximum_capacity
```

```
charging_efficiency % float # [0, 1]

discharging_efficiency % float # [0, 1]
```

[] Properties: Storage Variables

```
store

end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Battery(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Battery ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated; % [kbps]

    obj.maximum_capacity = init_data.maximum_capacity; % [W hr]
    obj.instantaneous_capacity = obj.maximum_capacity; % [W hr]
    obj.state_of_charge = 100*obj.instantaneous_capacity/
obj.maximum_capacity; % [percentage]

    obj.charging_efficiency = init_data.charging_efficiency;
    obj.discharging_efficiency = init_data.discharging_efficiency; %
[float <= 1]

    % Initialize Variables to store: instantaneous_capacity
state_of_charge
    obj.store = [];

    obj.store.instantaneous_capacity =
zeros(mission.storage.num_storage_steps, length(obj.instantaneous_capacity));
    obj.store.state_of_charge =
zeros(mission.storage.num_storage_steps, length(obj.state_of_charge));
```

```

        obj = func_update_true_SC_battery_store(obj, mission);

        % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_battery_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.instantaneous_capacity(mission.storage.k_storage,:)
= obj.instantaneous_capacity; % [W hr]
        obj.store.state_of_charge(mission.storage.k_storage,:) =
obj.state_of_charge; % [percentage]
    end

end

```

[] Methods: Main

Update Battery SoC

```

function obj = func_main_true_SC_battery(obj, mission, i_SC)

    obj.state_of_charge = 100*obj.instantaneous_capacity/
obj.maximum_capacity; % [percentage]

    obj = func_update_true_SC_battery_store(obj, mission);

    % Update Power Consumed

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

end

```

end

Published with MATLAB® R2022a

4.2 True_SC_Camera

Table of Contents

Class: True_SC_Camera	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	2
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	4
[] Methods: Main	5
[] Methods: Simple Camera	6
[] Methods: Check Target Visible	8

Class: True_SC_Camera

SC's Navigation Camera

```
classdef True_SC_Camera < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_generated_per_pixel % [kb] : Data generated per
pixel in camera image

instantaneous_data_compression % [float <= 1] : Camera data
compression (Optional)

mode_true_SC_camera_selector % [string]
% - Simple

measurement_wait_time % [sec] How often can measurements be taken?

location % [m] : Location of sensor, in body frame B

orientation % [unit vector] : Normal vector from location

orientation_up % [unit vector] : Up Normal vector from location in
camera frame

resolution % [x y] : Resolution of the camera in pixels (e.g. [1024
1024])

field_of_view % [deg] : Field of view (FOV) of the camera in deg
```

```
flag_show_camera_plot % [Boolean] : 1 = Shows the camera plot

flag_show_stars % [Boolean] : 1 = Show stars upto
mission.true_stars.maximum_magnitude. This takes a lot of time for wide FOV
cameras. (Optional)
```

[] Properties: Variables Computed Internally

```
name % [string] 'Camera i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

measurement_vector % [Image]

measurement_time % [sec] SC time when this measurement was taken

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

plot_handle % Matlab plot handle for the image

flag_target_visible % [Boolean] : 1 = Target is visible to this camera

instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

data % Other useful data
```

[] Properties: Storage Variables

```
store

end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Camera(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
```

```

        obj.name = ['Camera ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_generated_per_pixel =
init_data.instantaneous_data_generated_per_pixel; % [kb]
    obj.instantaneous_data_generated_per_sample = 0; % [kb] Data is
only generated when an image is captured

    obj.mode_true_SC_camera_selector =
init_data.mode_true_SC_camera_selector; % [string]
    obj.measurement_wait_time = init_data.measurement_wait_time; %
[sec]

    obj.flag_executive = 0;

    obj.measurement_time = 0; % [sec]

    obj.location = init_data.location; % [m]
    obj.orientation = init_data.orientation; % [unit vector]
    obj.orientation_up = init_data.orientation_up; % [unit vector]

    obj.resolution = init_data.resolution; % [x y] pixel
    obj.field_of_view = init_data.field_of_view; % [deg]

    obj.flag_show_camera_plot = init_data.flag_show_camera_plot;

    if isfield(init_data, 'flag_show_stars')
        obj.flag_show_stars = init_data.flag_show_stars;
    else
        obj.flag_show_stars = 0;
    end

    obj.flag_target_visible = 0;

    if isfield(init_data, 'instantaneous_data_compression')
        obj.instantaneous_data_compression =
init_data.instantaneous_data_compression;
    else
        obj.instantaneous_data_compression = 1;
    end

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end
    obj.data.flag_take_picture = 0;

    % Initialize Variables to store: flag_target_visible

```

```

        obj.store = [];
        obj.store.flag_target_visible =
zeros(mission.storage.num_storage_steps, length(obj.flag_target_visible));
        obj.store.flag_take_picture =
zeros(mission.storage.num_storage_steps, length(obj.data.flag_take_picture));
        obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
        obj.store.instantaneous_data_generated_per_sample =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_generated_per_sample)); % [kbps]
        obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_power_consumed)); % [W]

        % Update Storage
        obj = func_update_true_SC_camera_store(obj, mission);

        % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

        % Store video of func_visualize_SC_attitude_orbit_during_sim
        if (mission.storage.plot_parameters.flag_save_video == 1) &&
(obj.flag_show_camera_plot == 1)
            obj.data.video_filename = [mission.storage.output_folder,
mission.name, '_SC', num2str(i_SC), '_Camera', num2str(i_HW), '.mp4'];
            obj.data.myVideo =
VideoWriter(obj.data.video_filename, 'MPEG-4');
            obj.data.myVideo.FrameRate = 30; % Default 30
            obj.data.myVideo.Quality = 100; % Default 75
            open(obj.data.myVideo);
        end
    end
end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_camera_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage_attitude,:)
= obj.flag_executive; % [Boolean]

        if obj.flag_executive == 1

```

```

        obj.store.flag_target_visible(mission.storage.k_storage,:)
= obj.flag_target_visible; % [Boolean]
        obj.store.flag_take_picture(mission.storage.k_storage,:) =
obj.data.flag_take_picture; % [Boolean]

obj.store.instantaneous_data_generated_per_sample(mission.storage.k_storage,:)
= obj.instantaneous_data_generated_per_sample; % [kbps]

obj.store.instantaneous_power_consumed(mission.storage.k_storage,:) =
obj.instantaneous_power_consumed; % [W]
    end
end
end
end

```

[] Methods: Main

Update Camera

```

function obj = func_main_true_SC_camera(obj, mission, i_SC, i_HW)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % Take measurement

        if (mission.true_time.time - obj.measurement_time) >=
obj.measurement_wait_time

            % Sufficient time has elapsed for a new measurement
            obj.measurement_time = mission.true_time.time; % [sec]
            obj.data.flag_take_picture = 1;

            switch obj.mode_true_SC_camera_selector

                case 'Simple'
                    obj = func_true_SC_camera_Simple(obj, mission,
i_SC, i_HW);

                otherwise
                    error('Camera mode not defined!')
            end

            % Update Data Generated

            func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

        else
            % Data not generated in this time step
            obj.instantaneous_data_generated_per_sample = 0; % [kb]
            Data is only generated when an image is captured
            obj.data.flag_take_picture = 0;

```

```

        end

        % Update Power Consumed

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

else
    % Do nothing
    obj.data.flag_take_picture = 0;
end

% Update Storage
obj = func_update_true_SC_camera_store(obj, mission);

% Reset Variables
obj.flag_executive = 0;

end

```

[] Methods: Simple Camera

Simple Camera mode

```

function obj = func_true_SC_camera_Simple(obj, mission, i_SC, i_HW)

    obj.instantaneous_data_generated_per_sample =
obj.instantaneous_data_compression *
obj.instantaneous_data_generated_per_pixel * obj.resolution(1) *
obj.resolution(2); % [kb]

    if obj.flag_show_camera_plot == 1
        % Show Camera Plot

        obj.plot_handle = figure( (5*i_SC) + i_HW);
        clf
        set(obj.plot_handle, 'Color', rgb('Black'));
        set(obj.plot_handle, 'Units', 'pixels', 'Position', [1 1
obj.resolution])
        %
        set(obj.plot_handle, 'PaperPositionMode', 'auto');
        set(obj.plot_handle, 'Resize', 'off');
        hold on

        % Plot Target Body
        for i_target = 1:1:mission.num_target

            func_plot_target_shape(i_target, mission);

        end

        axis vis3d off
        axis equal
    end

```

```

material([0 1 0])

% field of view
camva(obj.field_of_view);

camproj('perspective');
axis image
axis off;

% Light coming from sun
%       camlight('headlight')
h = light;
h.Position =
(mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.position
-
mission.true_target{mission.true_SC{i_SC}.true_SC_navigation.index_relative_target}.posit
h.Style = 'local';

% Move SC away from Target

campos(mission.true_SC{i_SC}.true_SC_navigation.position_relative_target);

target_distance =
norm(mission.true_SC{i_SC}.true_SC_navigation.position_relative_target);

% Manage view from current attitude
x_true_hat =
(mission.true_SC{i_SC}.true_SC_adc.rotation_matrix * obj.orientation)';

y_true_hat =
(mission.true_SC{i_SC}.true_SC_adc.rotation_matrix * obj.orientation_up)';

camtarget(target_distance * x_true_hat)
camup([y_true_hat(1), y_true_hat(2), y_true_hat(3)]);

% Plot Stars
if obj.flag_show_stars == 1

    dot_product_angle_array =
acosd(mission.true_stars.all_stars_unit_vector * x_true_hat');
    flag_in_FOV = logical(dot_product_angle_array <=
(obj.field_of_view*2));

    flag_magnitude_limit =
logical(mission.true_stars.magnitude_visible <=
mission.true_stars.maximum_magnitude);

    flag_magnitude_limit_FOV = (flag_in_FOV &
flag_magnitude_limit);

```

```

        idx_array = find(flag_magnitude_limit_FOV);
        hold on

        for s = 1:1:length(idx_array)
            this_star = target_distance *
mission.true_stars.all_stars_unit_vector(idx_array(s),:);

plot3(this_star(1),this_star(2),this_star(3),'*w','MarkerSize',(1 +
mission.true_stars.maximum_magnitude -
mission.true_stars.magnitude_visible(idx_array(s))),'MarkerFaceColor','w','MarkerEdgeColor','w')
        end
    end

    drawnow limitrate

    if (mission.storage.plot_parameters.flag_save_video == 1)
        open(obj.data.myVideo);
        writeVideo(obj.data.myVideo, getframe(obj.plot_handle));
    end

end

obj = func_true_SC_camera_target_visible(obj, mission, i_SC);

% Update Storage
obj = func_update_true_SC_camera_store(obj, mission);

end

```

[] Methods: Check Target Visible

Check if Target is visible in Camera image

```

function obj = func_true_SC_camera_target_visible(obj, mission, i_SC)

    Target_relative_pos_hat =
func_normalize_vec(mission.true_target{mission.true_SC{i_SC}.true_SC_navigation.index_rel
- mission.true_SC{i_SC}.true_SC_navigation.position});

    x_true_hat = (mission.true_SC{i_SC}.true_SC_adc.rotation_matrix *
obj.orientation')';

    Camera_angle = acosd(dot(Target_relative_pos_hat, x_true_hat)); %
[deg]

    if Camera_angle <= (obj.field_of_view)
        obj.flag_target_visible = 1;
    else
        obj.flag_target_visible = 0;
    end

end

end

```

end

Published with MATLAB® R2022a

4.3 True_SC_Communication_Link

Table of Contents

Class: True_SC_Communication_Link	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	4
[] Methods: Main	4

Class: True_SC_Communication_Link

Tracks the Links between Radio Antennas

```
classdef True_SC_Communication_Link < handle
```

Properties

properties

[] Properties: Initialized Variables

```
TX_spacecraft % [integer] Use 0 for Ground Station
TX_spacecraft_Radio_HW % [integer]

RX_spacecraft % [integer] Use 0 for Ground Station
RX_spacecraft_Radio_HW % [integer]

% TODO - Decide if this is should stay here or go to software comm
wait_time_comm_dte % [sec] : Wait time without transmitting to earth.
This is a constant.
last_communication_time % [sec] : Last time that data has been sent.
This is updated in SC_Executive

flag_compute_data_rate % [Boolean]

given_data_rate % [kbps]

instantaneous_power_consumed % [Watts] : Communication Link control
overhead power consumption
```

[] Properties: Variables Computed Internally

```
name % [string] 'Comm Link i'

TX_name % [string]
```

```
RX_name % [string]

this_data_rate % [kbps] Actually used in simulation

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

flag_TX_RX_visible % [Boolean]
```

[] Properties: Storage Variables

```
store

end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Communication_Link(init_data, mission, i_SC,
i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Comm Link ', num2str(i_HW)];
    end

    obj.flag_executive = 0;
    obj.flag_TX_RX_visible = 0;

    obj.TX_spacecraft = init_data.TX_spacecraft;
    obj.TX_spacecraft_Radio_HW = init_data.TX_spacecraft_Radio_HW;

    obj.RX_spacecraft = init_data.RX_spacecraft;
    obj.RX_spacecraft_Radio_HW = init_data.RX_spacecraft_Radio_HW;

    obj.flag_compute_data_rate = init_data.flag_compute_data_rate;
    obj.given_data_rate = init_data.given_data_rate; % [kbps]

    % Initialize the power consumption property
    if isfield(init_data, 'instantaneous_power_consumed')
        obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed;
    else
        obj.instantaneous_power_consumed = 0.5; % Default low power
for link control overhead
```

```

end

if obj.flag_compute_data_rate == 1
    obj.this_data_rate = 0; % [kbps]
else
    obj.this_data_rate = obj.given_data_rate; % [kbps]
end

% Handle TX name
if obj.TX_spacecraft == 0
    obj.TX_name =
mission.true_GS_radio_antenna{obj.TX_spacecraft_Radio_HW}.name;
else
    tx_sc = mission.true_SC{obj.TX_spacecraft};
    if iscell(tx_sc.true_SC_radio_antenna)
        tx_antenna =
tx_sc.true_SC_radio_antenna{obj.TX_spacecraft_Radio_HW};
    else
        if obj.TX_spacecraft_Radio_HW ~= 1
            error('TX_spacecraft_Radio_HW index is %d, but
true_SC_radio_antenna is not a cell array.', obj.TX_spacecraft_Radio_HW);
        end
        tx_antenna = tx_sc.true_SC_radio_antenna;
    end
    obj.TX_name = ['SC ', num2str(obj.TX_spacecraft), ' ',
tx_antenna.name];
end

% Handle RX name
if obj.RX_spacecraft == 0
    obj.RX_name =
mission.true_GS_radio_antenna{obj.RX_spacecraft_Radio_HW}.name;
else
    rx_sc = mission.true_SC{obj.RX_spacecraft};
    if iscell(rx_sc.true_SC_radio_antenna)
        rx_antenna =
rx_sc.true_SC_radio_antenna{obj.RX_spacecraft_Radio_HW};
    else
        if obj.RX_spacecraft_Radio_HW ~= 1
            error('RX_spacecraft_Radio_HW index is %d, but
true_SC_radio_antenna is not a cell array.', obj.RX_spacecraft_Radio_HW);
        end
        rx_antenna = rx_sc.true_SC_radio_antenna;
    end
    obj.RX_name = ['SC ', num2str(obj.RX_spacecraft), ' ',
rx_antenna.name];
end

% Initialize Variables to store
obj.store = [];

obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive)); %
[integer]

```

```

        obj.store.flag_TX_RX_visible =
zeros(mission.storage.num_storage_steps, length(obj.flag_TX_RX_visible)); %
[integer]
        obj.store.this_data_rate =
zeros(mission.storage.num_storage_steps, length(obj.this_data_rate)); %
[kbps]

        % Update Storage
        obj = func_update_true_SC_communication_link_store(obj, mission);

        % Register with power system if this is a spacecraft component
(not a ground station)
        if i_SC > 0

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);
        end
    end
end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_communication_link_store(obj,
mission)

    if mission.storage.flag_store_this_time_step == 1

        obj.store.flag_executive(mission.storage.k_storage,:) =
obj.flag_executive; % [integer]
        obj.store.flag_TX_RX_visible(mission.storage.k_storage,:) =
obj.flag_TX_RX_visible; % [integer]
        obj.store.this_data_rate(mission.storage.k_storage,:) =
obj.this_data_rate; % [kbps]

    end

end

```

[] Methods: Main

Update all variables

```

function obj = func_main_true_SC_communication_link(obj, mission,
i_SC)

    % Check Link Visibility
    if obj.TX_spacecraft == 0
        obj.flag_TX_RX_visible =
mission.true_SC{obj.RX_spacecraft}.true_SC_navigation.flag_visible_Earth;

    elseif obj.RX_spacecraft == 0

```

```

        obj.flag_TX_RX_visible =
mission.true_SC{obj.TX_spacecraft}.true_SC_navigation.flag_visible_Earth;

    else
        error('Need to write this code where both are SC!')
    end

    % Compute Data Rate (if visible)
    if obj.flag_TX_RX_visible == 1
        % LOS to Earth available
        if obj.flag_compute_data_rate == 1
            obj.this_data_rate = 0; % [kbps]
            error('Write code to compute Link Margin!')
        else
            obj.this_data_rate = obj.given_data_rate; % [kbps]
        end
    else
        % In eclipse
        obj.this_data_rate = 0; % [kbps]
    end

    % Perform Data Transfer
    if (obj.flag_executive == 1) % && (obj.flag_TX_RX_visible == 1)

        if (obj.TX_spacecraft > 0) && (obj.flag_TX_RX_visible
== 1) && ((obj.this_data_rate * mission.true_time.time_step) >=
mission.true_SC{obj.TX_spacecraft}.software_SC_data_handling.total_data_storage)
            obj.this_data_rate =
mission.true_SC{obj.TX_spacecraft}.software_SC_data_handling.total_data_storage/
mission.true_time.time_step; % [kbps]
        end

        % Start TX and transmit data
        if obj.TX_spacecraft == 0
            % This is GS
            if
mission.true_GS_radio_antenna{obj.TX_spacecraft_Radio_HW}.flag_executive ~= 0
                error('[GS] TX link is already on!')
            else

mission.true_GS_radio_antenna{obj.TX_spacecraft_Radio_HW}.flag_executive = 1;

mission.true_GS_radio_antenna{obj.TX_spacecraft_Radio_HW}.mode_true_GS_radio_antenna_sele
= 'TX';

mission.true_GS_radio_antenna{obj.TX_spacecraft_Radio_HW}.instantaneous_data_rate_transmi
= obj.this_data_rate; % [kbps]
        end

    else
        % This is SC
        tx_sc = mission.true_SC{obj.TX_spacecraft};
        if iscell(tx_sc.true_SC_radio_antenna)

```

```

        tx_antenna =
tx_sc.true_SC_radio_antenna{obj.TX_spacecraft_Radio_HW};
    else
        if obj.TX_spacecraft_Radio_HW ~= 1
            error('TX_spacecraft_Radio_HW index is %d, but
true_SC_radio_antenna is not a cell array.', obj.TX_spacecraft_Radio_HW);
        end
        tx_antenna = tx_sc.true_SC_radio_antenna;
    end

    if tx_antenna.flag_executive ~= 0
        error('[SC] TX link is already on!')
    else
        tx_antenna.flag_executive = 1;
        tx_antenna.mode_true_SC_radio_antenna_selector = 'TX';
        tx_antenna.instantaneous_data_rate_removed =
obj.this_data_rate; % [kbps]
    end

end

% Start RX and receive data
if obj.RX_spacecraft == 0
    % This is GS
    if
mission.true_GS_radio_antenna{obj.RX_spacecraft_Radio_HW}.flag_executive ~= 0
        error('[GS] RX link is already on!')
    else

mission.true_GS_radio_antenna{obj.RX_spacecraft_Radio_HW}.flag_executive = 1;

mission.true_GS_radio_antenna{obj.RX_spacecraft_Radio_HW}.mode_true_GS_radio_antenna_selector = 'RX';

mission.true_GS_radio_antenna{obj.RX_spacecraft_Radio_HW}.instantaneous_data_rate_receive
= obj.this_data_rate; % [kbps]
    end

    else
        % This is SC
        rx_sc = mission.true_SC{obj.RX_spacecraft};
        if iscell(rx_sc.true_SC_radio_antenna)
            rx_antenna =
rx_sc.true_SC_radio_antenna{obj.RX_spacecraft_Radio_HW};
        else
            if obj.RX_spacecraft_Radio_HW ~= 1
                error('RX_spacecraft_Radio_HW index is %d, but
true_SC_radio_antenna is not a cell array.', obj.RX_spacecraft_Radio_HW);
            end
            rx_antenna = rx_sc.true_SC_radio_antenna;
        end

        if rx_antenna.flag_executive ~= 0
            error('[SC] RX link is already on!')

```

```

        else
            rx_antenna.flag_executive = 1;
            rx_antenna.mode_true_SC_radio_antenna_selector = 'RX';
            rx_antenna.instantaneous_data_rate_generated =
obj.this_data_rate; % [kbps]
        end
    end

    % Update power consumption for this link's control overhead
    if i_SC > 0

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);
    end
end

    % Update Storage
    obj = func_update_true_SC_communication_link_store(obj, mission);

    % Reset All Variables
    obj.flag_executive = 0;

end

end

end

```

Published with MATLAB® R2022a

4.4 True_SC_Chemical_Thruster

Table of Contents

Class: True_SC_Chemical_Thruster	1
Properties	1
[] Properties: Initialized Variables	1
Methods	3
Constructor	3
Update Storage	5
Start Warm-Up	6
Check Warm-Up Status	6
Validate Properties	6
Main Thruster Logic	7

Class: True_SC_Chemical_Thruster

Individual SC's chemical thruster

```
classdef True_SC_Chemical_Thruster < handle

    % True_SC_Chemical_Thruster class represents a single chemical-thruster
    % with various properties and methods for simulation and analysis.
```

Properties

properties

[] Properties: Initialized Variables

```
    instantaneous_power_consumption    %[Watts] : Instantaneous power
consumed by unit (if it is switched on)
    instantaneous_power_consumed       %[Watts] : Alias for power tracking
system
    thruster_warm_up_power_consumed    %[Watts] : Power consumed during
warm-up time
    command_actuation_power_consumed   %[Watts] : Power consumed during
actuation
    name                               % Name of the thruster
    health                             % Health of the thruster (0 : Not
working - 1 : Working)
    temperature                        %[Celsius] Temperature of the
thruster

    instantaneous_data_volume
    instantaneous_data_generated       %[kb] : Data generated during the
current time step, in kilo bits

    control_force_CT                   %[N] : Control force vector generated
by the unit that passes through Center of Mass of SC
    control_torque_CT                  %[Nm] : Control torque about Center
of Mass of SC, generated by the unit
```

```

mode_true_chemical_thruster_selector% Mode (Truth/Simple)
flag_executive % [Bool] do we need to fire this time step (set by sw
orbit control)
flag_warming_up % [Bool] is the thruster in warm-up mode

thruster_warm_up_time          %[sec] : How much warm-up time is
needed?
accumulated_warm_up_time      %[sec] : How long has the thruster
been warming up
command_time                  %[sec] : Time when latest Command was
actuated
command_wait_time             %[sec] : Seconds between commands
warm_up_start_time            %[sec] : Time when warm-up started
ready_state_start_time        %[sec] : Time when thruster entered
ready state
pending_fire                  %[Boolean] : Flag to indicate a
pending fire command between cycles
pending_thrust                %[N] : Remembered thrust value for a
pending fire command

command_actuation             %[Boolean] : 1 = Command actuation
during this time step (after actuator has warmed up)
command_executed              %[Boolean] : 1 = Command executed
during this time step

thruster_state                %[String] : Current state of the
thruster: 'idle', 'warming_up', 'ready', 'firing'

orientation                   %[Unit vector] : Normal vector of
thrust in body frame B
location                      %[m] : Location of actuator, in body
frame B
force_inertial                %[N] : Vector force in inertial frame
applied by this thruster

isp                           %[s] : Specific impulse of the
thruster

maximum_thrust                %[N]
minimum_thrust                %[N] : This need not be 0
commanded_thrust              %[N] : Comes from the SC's orbit
control software
thruster_noise                %[N] : Added to every dimension of
commanded_thrust
true_commanded_thrust         %[N] : Actual thrust with noise
applied, unknown to SC software

% New properties for fuel consumption
fuel_consumed_per_firing      %[kg] : Amount of fuel consumed in
current firing
total_fuel_consumed           %[kg] : Total fuel consumed by this
thruster

```

```
store                                % Structure to store historical data

end
```

Methods

```
methods
```

Constructor

```
function obj = True_SC_Chemical_Thruster(init_data, mission, i_SC,
i_MT)
    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Chemical Thruster ', num2str(i_MT)];
    end
    obj.health = true;
    obj.temperature = 20;
    obj.instantaneous_power_consumption =
init_data.instantaneous_power_consumption; % Watts
    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumption; % Watts

    % Initialize specific power consumption settings
    if isfield(init_data, 'thruster_warm_up_power_consumed')
        obj.thruster_warm_up_power_consumed =
init_data.thruster_warm_up_power_consumed; % Watts
    else
        obj.thruster_warm_up_power_consumed =
obj.instantaneous_power_consumption; % Default to regular power
    end

    if isfield(init_data, 'command_actuation_power_consumed')
        obj.command_actuation_power_consumed =
init_data.command_actuation_power_consumed; % Watts
    else
        obj.command_actuation_power_consumed = 3 *
obj.instantaneous_power_consumption; % Default to 3x regular power
    end

    obj.instantaneous_data_generated =
init_data.instantaneous_data_generated_per_sample; % Kb
    obj.thruster_noise = init_data.chemical_thruster_noise; % Noise
level
    obj.command_wait_time = init_data.command_wait_time; % Seconds
between commands
    obj.location = init_data.location; % Meters (body frame)
    obj.orientation = init_data.orientation; % Unit vector (body
frame)

    obj.isp = init_data.chemical_thruster_ISP; % sec
```

```

obj.force_inertial = zeros(1,3);

obj.minimum_thrust = init_data.minimum_thrust;
obj.maximum_thrust = init_data.maximum_thrust;

% Ensure thrust variables are properly initialized as scalars
obj.commanded_thrust = 0; % Initialize as scalar 0, not empty
array
obj.true_commanded_thrust = 0; % Initialize as scalar 0, not empty
array
obj.control_force_CT = zeros(1,3); % Ensure it's a 1x3 array
obj.control_torque_CT = zeros(1,3); % Ensure it's a 1x3 array

% Initialize warm-up related properties
obj.thruster_warm_up_time = 30; % Default warm-up time (30
seconds)
obj.accumulated_warm_up_time = 0;
obj.warm_up_start_time = 0;
obj.ready_state_start_time = 0;
obj.flag_warming_up = false;
obj.thruster_state = 'idle';

% Initialize command properties
obj.command_actuation = false;
obj.command_executed = false;
obj.command_time = 0;
obj.flag_executive = false;
obj.pending_fire = false;
obj.pending_thrust = 0;

% Initialize data properties
obj.instantaneous_data_volume = obj.instantaneous_data_generated;
% Initialize to same value as data generated

% Initialize fuel consumption properties
obj.fuel_consumed_per_firing = 0;
obj.total_fuel_consumed = 0;

% Initialize Storage Variables
obj.store = [];
obj.store.commanded_thrust =
zeros(mission.storage.num_storage_steps,1);
obj.store.true_commanded_thrust =
zeros(mission.storage.num_storage_steps,1);
obj.store.force_inertial =
zeros(mission.storage.num_storage_steps,3); % Changed from 1 to 3 columns
obj.store.fuel_consumed_per_firing =
zeros(mission.storage.num_storage_steps,1);
obj.store.total_fuel_consumed =
zeros(mission.storage.num_storage_steps,1);
obj.store.thruster_state =
cell(mission.storage.num_storage_steps,1);
obj.store.warm_up_progress =
zeros(mission.storage.num_storage_steps,1);

```

```

        obj.store.pending_fire =
zeros(mission.storage.num_storage_steps,1);
        obj.store.pending_thrust =
zeros(mission.storage.num_storage_steps,1);
        obj.store.control_torque_CT =
zeros(mission.storage.num_storage_steps,3); % Add storage

        % Register with power system

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);
end

```

Update Storage

```

function obj = func_update_true_chemical_thruster_store(obj, mission)
    % Append new thrust data to the storage arrays

    % Ensure all values are valid before storing (defend against empty
arrays)
    if isempty(obj.commanded_thrust)
        obj.commanded_thrust = 0;
    end

    if isempty(obj.true_commanded_thrust)
        obj.true_commanded_thrust = 0;
    end

    % Store values with safety checks
    obj.store.commanded_thrust(mission.storage.k_storage,:) =
obj.commanded_thrust;
    obj.store.true_commanded_thrust(mission.storage.k_storage,:) =
obj.true_commanded_thrust;
    obj.store.force_inertial(mission.storage.k_storage,:) =
obj.force_inertial;
    obj.store.fuel_consumed_per_firing(mission.storage.k_storage,:) =
obj.fuel_consumed_per_firing;
    obj.store.total_fuel_consumed(mission.storage.k_storage,:) =
obj.total_fuel_consumed;
    obj.store.thruster_state{mission.storage.k_storage} =
obj.thruster_state;

    % Store pending fire information
    obj.store.pending_fire(mission.storage.k_storage) =
obj.pending_fire;
    obj.store.pending_thrust(mission.storage.k_storage) =
obj.pending_thrust;

    % Store control torque
    obj.store.control_torque_CT(mission.storage.k_storage,:) =
obj.control_torque_CT;

    % Calculate warm-up progress as a percentage

```

```

        if obj.thruster_warm_up_time > 0
            obj.store.warm_up_progress(mission.storage.k_storage) =
min(100, (obj.accumulated_warm_up_time / obj.thruster_warm_up_time) * 100);
        else
            obj.store.warm_up_progress(mission.storage.k_storage) = 100; %
No warm-up needed
        end
    end
end

```

Start Warm-Up

```

function obj = func_start_warm_up(obj, mission)
    % Start the thruster warm-up process
    if ~obj.flag_warming_up && strcmp(obj.thruster_state, 'idle') &&
obj.health
        obj.flag_warming_up = true;
        obj.thruster_state = 'warming_up';
        obj.warm_up_start_time = mission.true_time.time;
        obj.accumulated_warm_up_time = 0;
    end
end

```

Check Warm-Up Status

```

function is_ready = func_is_thruster_ready(obj)
    % Check if the thruster is ready to fire
    is_ready = strcmp(obj.thruster_state, 'ready') ||
strcmp(obj.thruster_state, 'firing');
end

```

Validate Properties

```

function obj = func_validate_thruster_properties(obj)
    % Safety function to ensure all properties are properly
initialized

    % Check and fix scalar properties
    if isempty(obj.commanded_thrust)
        obj.commanded_thrust = 0;
    end

    if isempty(obj.true_commanded_thrust)
        obj.true_commanded_thrust = 0;
    end

    if isempty(obj.instantaneous_power_consumption)
        obj.instantaneous_power_consumption = 0;
    end

    if isempty(obj.instantaneous_power_consumed)
        obj.instantaneous_power_consumed = 0;
    end

```

```

        end

        % Check and fix vector properties
        if isempty(obj.force_inertial) ||
~isequal(size(obj.force_inertial), [1,3])
            obj.force_inertial = zeros(1,3);
        end

        if isempty(obj.control_torque_CT) ||
~isequal(size(obj.control_torque_CT), [1,3])
            obj.control_torque_CT = zeros(1,3);
        end

        if isempty(obj.control_force_CT) ||
~isequal(size(obj.control_force_CT), [1,3])
            obj.control_force_CT = zeros(1,3);
        end

        % Check other critical properties
        if isempty(obj.thruster_state)
            obj.thruster_state = 'idle';
        end

        if isempty(obj.ready_state_start_time)
            obj.ready_state_start_time = 0;
        end

        if isempty(obj.pending_fire)
            obj.pending_fire = false;
        end

        if isempty(obj.pending_thrust)
            obj.pending_thrust = 0;
        end

        if isempty(obj.instantaneous_data_volume)
            obj.instantaneous_data_volume = 0;
        end
    end
end

```

Main Thruster Logic

```

function func_main_true_chemical_thruster(obj, mission, i_SC, ~)
    % Main function for True_SC_Chemical_Thruster
    % Controls thrust application, power consumption, and data
    generation.

    % Validate properties to ensure everything is properly initialized
    obj = func_validate_thruster_properties(obj);

    % Reset fuel consumed this firing (but NOT command_executed - that
    needs
    % to persist for one cycle so orbit control can see it)

```

```

obj.fuel_consumed_per_firing = 0;

% Initialize power consumption to idle value
obj.instantaneous_power_consumption = 0;
obj.instantaneous_power_consumed = 0;

% Save a local copy of command_executed for this cycle's
processing
was_executed_last_cycle = obj.command_executed;

% Check health status
if obj.health == 0
    obj.thruster_state = 'idle';
    obj.flag_warming_up = false;
    obj.command_executed = false;
    return;
end

% State machine for thruster operation
switch obj.thruster_state
    case 'idle'
        % Check if we need to start warming up
        if obj.flag_warming_up
            disp(['Thruster transitioning from idle to warming_up
at time ', num2str(mission.true_time.time), ' sec']);
            obj.thruster_state = 'warming_up';
            obj.accumulated_warm_up_time = 0;
        end

        % Minimal idle power consumption (fixed 0.1W in idle
state)
        obj.instantaneous_power_consumption = 0.1;

    case 'warming_up'
        % Update accumulated warm-up time
        obj.accumulated_warm_up_time =
obj.accumulated_warm_up_time + mission.true_time.time_step;

        % Check if warm-up is complete
        if obj.accumulated_warm_up_time >=
obj.thruster_warm_up_time
            disp(['Thruster completed warm-up and entering ready
state at time ', num2str(mission.true_time.time), ' sec']);
            obj.thruster_state = 'ready';
            obj.ready_state_start_time = mission.true_time.time; %
Set the start time of ready state
        end

        % Set warm-up power consumption
        obj.instantaneous_power_consumption =
obj.thruster_warm_up_power_consumed;

    case 'ready'
        % Thruster is ready to fire

```

```

        % Maintain warm-up power until firing or timeout
        obj.instantaneous_power_consumption =
obj.thruster_warm_up_power_consumed;

        % Check for command actuation
        if obj.flag_executive && obj.commanded_thrust > 0
            % Save the command for the next cycle
            obj.pending_fire = true;
            obj.pending_thrust = obj.commanded_thrust;

            disp(['Thruster transitioning from ready to firing at
time ', num2str(mission.true_time.time), ' sec']);
            disp([' Commanded thrust: ',
num2str(obj.commanded_thrust), ' N']);
            obj.thruster_state = 'firing';
        end

        % Check for timeout (5 minutes idle in ready state)
        % Use the ready_state_start_time as reference to measure
timeout period
        if (mission.true_time.time - obj.ready_state_start_time) >
300
            disp(['Thruster timing out from ready state after ',
num2str(mission.true_time.time - obj.ready_state_start_time), ' seconds']);
            obj.thruster_state = 'idle';
            obj.flag_warming_up = false;
        end

        case 'firing'
            % Process thruster firing - use either current command or
pending command from previous cycle
            if (obj.flag_executive && obj.commanded_thrust > 0) ||
obj.pending_fire
                try
                    % Use either active command or saved pending
command
                    if obj.flag_executive && obj.commanded_thrust > 0
                        thrust_to_use = obj.commanded_thrust;
                    else
                        thrust_to_use = obj.pending_thrust;
                        disp(['Using pending thrust from previous
cycle: ', num2str(thrust_to_use), ' N']);
                    end

                    % Update thrust with noise
                    obj.true_commanded_thrust = thrust_to_use +
obj.thruster_noise * (2*rand() - 1);

                    % Ensure thrust remains within bounds
                    obj.true_commanded_thrust =
max(min(obj.true_commanded_thrust, obj.maximum_thrust), obj.minimum_thrust);

                    % Get current attitude matrix to transform from
body to inertial frame

```

```

R =
quaternionToRotationMatrix(mission.true_SC{i_SC}.true_SC_adc.attitude);

% Calculate force vector in body frame first
force_body = obj.true_commanded_thrust *

obj.orientation;

% Transform to inertial frame
obj.force_inertial = R * force_body'; % Netwons

% Log the thruster firing
disp(['THRUSTER FIRING with thrust
of ', num2str(obj.true_commanded_thrust), ' N at time ',
num2str(mission.true_time.time), ' sec']);

% Make sure that the control orbit verifies if
% the deltaV has been performed correctly

mission.true_SC{i_SC}.software_SC_control_orbit.flag_executive = 1;

% Add thrust on the SC (in inertial frame)

mission.true_SC{i_SC}.true_SC_navigation.control_force =
mission.true_SC{i_SC}.true_SC_navigation.control_force + obj.force_inertial';

% Compute the disturbance torque
r = obj.location -
mission.true_SC{i_SC}.true_SC_body.location_COM;
obj.control_torque_CT = cross(r, force_body)';

% Apply torque to attitude system

mission.true_SC{i_SC}.true_SC_adc.disturbance_torque =
mission.true_SC{i_SC}.true_SC_adc.disturbance_torque + obj.control_torque_CT';

obj.control_torque_CT = obj.control_torque_CT';

% Update power consumption (warm-up + actuation)
obj.instantaneous_power_consumption =
obj.thruster_warm_up_power_consumed + obj.command_actuation_power_consumed;

% Set command execution flag - this will persist
until next cycle
obj.command_executed = true;

% Calculate DeltaV applied and directly update
orbit control
dt = mission.true_time.time_step;
sc_mass =
mission.true_SC{i_SC}.true_SC_body.total_mass;

% Calculate DeltaV applied (normalized direction *
magnitude)

```

```

        DeltaV_applied = (norm(obj.force_inertial) * dt /
sc_mass) * (obj.force_inertial / norm(obj.force_inertial));

        % Update orbit control's total executed DeltaV
directly

mission.true_SC{i_SC}.software_SC_control_orbit.total_DeltaV_executed =
mission.true_SC{i_SC}.software_SC_control_orbit.total_DeltaV_executed +
DeltaV_applied;

        % Log the actuation
        disp(['Applied DeltaV: ',
num2str(DeltaV_applied(1)), ', ', num2str(DeltaV_applied(2)), ', ',
num2str(DeltaV_applied(3)), ' m/s']);
        disp(['Total DeltaV now: ',
num2str(mission.true_SC{i_SC}.software_SC_control_orbit.total_DeltaV_executed(1)), ',
', ...
num2str(mission.true_SC{i_SC}.software_SC_control_orbit.total_DeltaV_executed(2)), ',
', ...
num2str(mission.true_SC{i_SC}.software_SC_control_orbit.total_DeltaV_executed(3)), ' m/s']]);

        % Calculate and display the remaining DeltaV to be
applied

        remaining_DeltaV =
mission.true_SC{i_SC}.software_SC_control_orbit.desired_control_DeltaV -
mission.true_SC{i_SC}.software_SC_control_orbit.total_DeltaV_executed;
        disp(['Remaining DeltaV: ',
num2str(norm(remaining_DeltaV)), ' m/s']);

        % Data generatio
        if isempty(obj.instantaneous_data_volume)
            obj.instantaneous_data_volume =
obj.instantaneous_data_generated;
        end

        % Calculate fuel consumption
        dt = mission.true_time.time_step;
        g0 = 9.80665; % m/s^2
        obj.fuel_consumed_per_firing =
(obj.true_commanded_thrust * dt) / (obj.isp * g0);
        obj.total_fuel_consumed = obj.total_fuel_consumed
+ obj.fuel_consumed_per_firing;

        % Consume fuel from the tank if fuel tank exists
        if
isfield(mission.true_SC{i_SC}, 'true_SC_fuel_tank') && ...

mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank > 0

        % Find the assigned fuel tank for this
thruster (assuming the first one for now)

```

```

        i_tank = 1; % In a more complex
implementation, each thruster might have a specific tank assigned

        % Consume fuel from the tank

mission.true_SC{i_SC}.true_SC_fuel_tank{i_tank}.func_consume_fuel(obj.fuel_consumed_per_f
end

        % Clear pending fire flag after successful firing
obj.pending_fire = false;
obj.pending_thrust = 0;

catch exception
    % Log error but continue execution
    warning('Error during thruster firing: %s',
exception.message);

    obj.thruster_state = 'ready';
    obj.command_executed = false;
    obj.pending_fire = false;
    obj.pending_thrust = 0;
end
else
    % No command to fire, go back to ready state
    if strcmp(obj.thruster_state, 'firing')
        disp(['Thruster returning to ready state from
firing at time ', num2str(mission.true_time.time), ' sec']);
    end
    obj.thruster_state = 'ready';
    % We don't reset command_executed here as it needs to
persist for one cycle
end
end

    % Always update power consumed for power tracking
obj.instantaneous_power_consumed =
obj.instantaneous_power_consumption;

    % Update the power system with this consumption

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update storage
obj.func_update_true_chemical_thruster_store(mission);

    % Reset executive flag and thrust command after processing
obj.flag_executive = false;
obj.commanded_thrust = 0;
obj.control_torque_CT = zeros(1,3);

    % Reset command_executed flag ONLY if it was set in the previous
cycle
    % (not the current one)
    if was_executed_last_cycle

```

```
        obj.command_executed = false;
    end

    obj.true_commanded_thrust = 0;
end
end
end
```

Published with MATLAB® R2022a

4.5 True_SC_Fuel_Tank

Table of Contents

Class: True_SC_Fuel_Tank	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Storage Variables	1
Methods	2
Constructor	2
Update Storage	3
Consume Fuel	3
Main Function	3

Class: True_SC_Fuel_Tank

Tracks the fuel state of the spacecraft propellant tank

```
classdef True_SC_Fuel_Tank < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
    name                % [string] 'Fuel Tank i'
    health               % [integer] Health of fuel tank (0: Off, 1:
On)
    temperature         % [deg C] Temperature of fuel tank

    instantaneous_power_consumed % [Watts] Power consumed by the fuel
tank (e.g., heaters)
    instantaneous_data_rate_generated % [kbps] Data rate generated during
current time step

    maximum_capacity    % [kg] Maximum fuel mass capacity
    instantaneous_fuel_mass % [kg] Current fuel mass in the tank
    fuel_density        % [kg/m^3] Density of the propellant

    location            % [m] Location of the tank in the body frame
    shape_model         % Structure containing shape model information

    flag_update_SC_body % [Boolean] Flag to signal when SC body mass
needs to be updated
```

[] Properties: Storage Variables

```
store % Structure to store historical data
```

end

Methods

methods

Constructor

```
function obj = True_SC_Fuel_Tank(init_data, mission, i_SC, i_HW)
    % Constructor for the fuel tank class

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Fuel Tank ', num2str(i_HW)];
    end

    obj.health = 1; % Default to healthy
    obj.temperature = 20; % Default temperature in Celsius

    % Power and data parameters
    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed;
    obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated;

    % Fuel parameters
    obj.maximum_capacity = init_data.maximum_capacity;
    obj.instantaneous_fuel_mass = init_data.initial_fuel_mass;
    obj.fuel_density = init_data.fuel_density;

    % Physical parameters
    obj.location = init_data.location;

    % Shape model if provided
    if isfield(init_data, 'shape_model')
        obj.shape_model = init_data.shape_model;
    else
        obj.shape_model = [];
    end

    % Flag for body update
    obj.flag_update_SC_body = 0;

    % Initialize storage variables
    obj.store = [];
    obj.store.instantaneous_fuel_mass =
zeros(mission.storage.num_storage_steps, 1);
    obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps, 1);

    % Register with power system
```

```
func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);
end
```

Update Storage

```
function obj = func_update_true_SC_fuel_tank_store(obj, mission)
    % Update storage variables for the fuel tank

    if mission.storage.flag_store_this_time_step == 1

obj.store.instantaneous_fuel_mass(mission.storage.k_storage, :) =
obj.instantaneous_fuel_mass;

obj.store.instantaneous_power_consumed(mission.storage.k_storage, :) =
obj.instantaneous_power_consumed;
    end
end
```

Consume Fuel

```
function obj = func_consume_fuel(obj, fuel_mass_consumed)
    % Consume fuel from the tank

    % Check if there's enough fuel
    if fuel_mass_consumed > obj.instantaneous_fuel_mass
        warning('Fuel tank %s: Attempted to consume more fuel than
available!', obj.name);
        fuel_mass_consumed = obj.instantaneous_fuel_mass;
    end

    % Update fuel mass
    obj.instantaneous_fuel_mass = obj.instantaneous_fuel_mass -
fuel_mass_consumed;

    % Set flag to update SC body
    obj.flag_update_SC_body = 1;
end
```

Main Function

```
function obj = func_main_true_SC_fuel_tank(obj, mission, i_SC)
    % Main function for the fuel tank

    % Update propellant mass in the spacecraft body
    if obj.flag_update_SC_body == 1
        % Instead of directly updating the mass, just set the flag for
the body to update
        % The body's func_update_SC_body_mass will grab the latest
fuel mass
    end
```

```
mission.true_SC{i_SC}.true_SC_body.flag_update_SC_body_total_mass_COM_MI = 1;

    % Reset update flag
    obj.flag_update_SC_body = 0;
end

    % Update power system with this tank's power consumption (heaters,
    valves, etc.)

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update storage
    obj = func_update_true_SC_fuel_tank_store(obj, mission);
end

end

end
```

Published with MATLAB® R2022a

4.6 True_SC_Generic_Sensor

Table of Contents

Class: True_SC_Generic_Sensor	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	1
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3

Class: True_SC_Generic_Sensor

Generic Sensor class

```
classdef True_SC_Generic_Sensor < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate, in kilo bits
per sec (kbps)
```

[] Properties: Variables Computed Internally

```
name % [string] 'Generic Sensor i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

data % Other useful data
```

[] Properties: Storage Variables

```
store

end
```

Methods

methods

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Generic_Sensor(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Generic Sensor ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated; % [kbps]

    obj.flag_executive = 0;

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    % Initialize Variables to store: measurement_vector
    obj.store = [];
    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
    obj.store.instantaneous_data_rate_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_generated)); % [kbps]
    obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_power_consumed)); % [W]

    % Update Storage
    obj = func_update_true_SC_generic_sensor_store(obj, mission);

    % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update SC Data Handling Class
```

```
func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,  
obj, mission);
```

```
end
```

[] Methods: Store

Update the store variable

```
function obj = func_update_true_SC_generic_sensor_store(obj, mission)  
  
    if mission.storage.flag_store_this_time_step == 1  
        obj.store.flag_executive(mission.storage.k_storage_attitude,:) =  
= obj.flag_executive; % [Boolean]  
  
        if obj.flag_executive == 1  
  
obj.store.instantaneous_data_rate_generated(mission.storage.k_storage,:) =  
obj.instantaneous_data_rate_generated; % [kbps]  
  
obj.store.instantaneous_power_consumed(mission.storage.k_storage,:) =  
obj.instantaneous_power_consumed; % [W]  
        end  
    end  
  
end
```

[] Methods: Main

Update Radar

```
function obj = func_main_true_SC_generic_sensor(obj, mission, i_SC)  
  
    if (obj.flag_executive == 1) && (obj.health == 1)  
        % Take measurement  
  
        if  
isfield(obj.data, 'instantaneous_power_consumed_per_SC_mode')  
            obj.instantaneous_power_consumed =  
obj.data.instantaneous_power_consumed_per_SC_mode(mission.true_SC{i_SC}.software_SC_execu  
[W]  
        end  
  
        % Update Power Consumed  
  
func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,  
obj, mission);  
  
        % Update Data Generated  
  
func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,  
obj, mission);
```

```
else
    % Do nothing
end

% Update Storage
obj = func_update_true_SC_generic_sensor_store(obj, mission);

% Reset Variables
obj.flag_executive = 0;

end

end

end
```

Published with MATLAB® R2022a

4.7 True_SC_IMU

Table of Contents

Class: True_SC_IMU	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3
[] Methods: Truth	4
[] Methods: Simple	5

Class: True_SC_IMU

Tracks the IMU measurements

```
classdef True_SC_IMU < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

mode_true_sc_imu_selector % [string]
% - Truth
% - Simple

measurement_noise % [rad/sec] (1-sigma standard deviation)

measurement_wait_time % [sec]

location % [m] : Location of sensor, in body frame B

orientation % [unit vector] : Normal vector from location
```

[] Properties: Variables Computed Internally

```
name % [string] 'Sun Sensor i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
```

```

    % - 1. Switched on, works nominally

    temperature % [deg C] : Temperature of sensor/actuator

    measurement_vector % [quaternion]

    measurement_time % [sec] SC time when this measurement was taken

    flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

    data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_SC_IMU(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['IMU ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_generated_per_sample =
init_data.instantaneous_data_generated_per_sample; % [kb]

    obj.mode_true_SC_imu_selector =
init_data.mode_true_SC_imu_selector; % [string]
    obj.measurement_wait_time = init_data.measurement_wait_time; %
[sec]
    obj.measurement_noise = init_data.measurement_noise; % [rad]

    obj.measurement_vector = zeros(1,3);

    obj.flag_executive = 1;

```

```

obj.measurement_time = -inf; % [sec]

obj.location = init_data.location; % [m]
obj.orientation = init_data.orientation; % [unit vector]

if isfield(init_data, 'data')
    obj.data = init_data.data;
else
    obj.data = [];
end

% Initialize Variables to store: measurement_vector
obj.store = [];
obj.store.measurement_vector =
zeros(mission.storage.num_storage_steps, length(obj.measurement_vector));

% Update Storage
obj = func_update_true_SC_imu_store(obj, mission);

% Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

% Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_imu_store(obj, mission)

    if mission.storage.flag_store_this_time_step_attitude == 1

obj.store.measurement_vector(mission.storage.k_storage_attitude,:) =
obj.measurement_vector; % [quaternion]
    end

end

```

[] Methods: Main

Update Camera

```

function obj = func_main_true_SC_imu(obj, mission, i_SC)

    if (obj.flag_executive == 1) && (obj.health == 1)

```

```

        % Take measurement

        if (mission.true_time.time_attitude - obj.measurement_time) >=
obj.measurement_wait_time

            % Sufficient time has elapsed for a new measurement
            obj.measurement_time = mission.true_time.time_attitude; %
[sec]

            switch obj.mode_true_SC_imu_selector

                case 'Truth'
                    obj = func_true_SC_imu_Truth(obj, mission, i_SC);

                case 'Simple'
                    obj = func_true_SC_imu_Simple(obj, mission, i_SC);

                otherwise
                    error('IMU mode not defined!')
            end

            % Update Data Generated

            func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
            obj, mission);

        else
            % Data not generated in this time step

        end

        % Update Power Consumed

        func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
        obj, mission);

    else
        % Do nothing

    end

    % Update Storage
    obj = func_update_true_SC_imu_store(obj, mission);

    % Reset Variables
    obj.flag_executive = 0;

end

```

[] Methods: Truth

IMU mode

```
function obj = func_true_SC_imu_Truth(obj, mission, i_SC)
    obj.measurement_vector =
mission.true_SC{i_SC}.true_SC_adc.angular_velocity;
end
```

[] Methods: Simple

IMU mode

```
function obj = func_true_SC_imu_Simple(obj, mission, i_SC)
    obj.measurement_vector =
mission.true_SC{i_SC}.true_SC_adc.angular_velocity +
obj.measurement_noise*randn(1,3) ;
end

end

end
```

Published with MATLAB® R2022a

4.8 True_SC_Micro_Thruster

Table of Contents

Class: True_SC_Micro_Thruster	1
Properties	1
[] Properties: Initialized Variables	1
Constructor	2

Class: True_SC_Micro_Thruster

Individual SC's micro thruster

```
classdef True_SC_Micro_Thruster < handle

    % True_SC_Micro_Thruster class represents a single micro-thruster
    % with various properties and methods for simulation and analysis.
```

Properties

properties

[] Properties: Initialized Variables

```
    instantaneous_power_consumption    %[Watts] : Instantaneous power
consumed by unit (if it is switched on)
    instantaneous_power_consumed        %[Watts]
    thruster_warm_up_power_consumed     %[Watts] : Power consumed during
warm-up time
    command_actuation_power_consumed    %[Watts] : Power consumed during
actuation
    name                                % Name of the thruster
    health                              % Health of the thruster (0 : Not
working - 1 : Working)
    temperature                         %[Celcius] Temerature of the thruster

    instantaneous_data_volume
    instantaneous_data_generated        %[kb] : Data generated during the
current time step, in kilo bits

    control_force_MT                    %[N] : Control force vector generated
by the unit that passes through Center of Mass of SC
    control_torque_MT                  %[Nm] : Control torque about Center
of Mass of SC, generated by the unit

    mode_true_sc_micro_thruster_selector% Mode (Truth/Simple)

    thruster_warm_up_time                %[sec] : How much warm-up time is
needed?
    accumulated_warm_up_time            %[sec]
    command_time                        %[sec] : Time when latest Command was
actuated
```

```

        command_wait_time                %[sec] : Seconds between commands

        command_actuation                %[Boolean] : Command actuation
during this time step (after actuator has warmed up)
        command_executed                %[Boolean] : Command executed during
this time step

        orientation                      %[Unit vector] : Normal vector of
thrust in body frame B
        location                        %[m] : Location of actuator, in body
frame B

        maximum_thrust                  %[N]
        minimum_thrust                  %[N]
        commanded_thrust                %[N] : Comes from the SC's software
        thruster_noise                  %[N] : Added to every dimension of
commanded_thrust
        true_commanded_thrust            %[N] : Add thruster noise to
commanded_thrust. This is the actual thrust of this thruster, which the SC
software doesn't know.

        % Fuel consumption properties
        isp                             %[s] : Specific impulse of the
thruster
        fuel_consumed_per_firing         %[kg] : Amount of fuel consumed in
current firing
        total_fuel_consumed              %[kg] : Total fuel consumed by this
thruster

        store

end

methods

```

Constructor

```

function obj = True_SC_Micro_Thruster(init_data, mission, i_SC, i_MT)
    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Micro Thruster ', num2str(i_MT)];
    end
    obj.health = true;
    obj.temperature = 20;
    obj.instantaneous_power_consumption =
init_data.instantaneous_power_consumption; % Watts
    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumption; % Watts
    obj.instantaneous_data_generated =
init_data.instantaneous_data_generated_per_sample; % Kb
    obj.mode_true_SC_micro_thruster_selector =
init_data.mode_true_SC_micro_thruster_selector; % Mode (Truth/Simple)

```

```

        obj.command_wait_time = init_data.command_wait_time; % Seconds
between commands
        obj.location = init_data.location; % Meters (body frame)

        obj.orientation = init_data.orientation; % Unit vector (body
frame)
        obj.orientation = obj.orientation / norm(obj.orientation); %
Normalize

        obj.thruster_noise = init_data.thruster_noise; % Noise level [N]
        obj.minimum_thrust = init_data.minimum_thrust;
        obj.maximum_thrust = init_data.maximum_thrust; % [N]

        % Set default values for warm-up parameters
        obj.thruster_warm_up_time = 0; % Default no warm-up
        obj.accumulated_warm_up_time = 0;
        obj.thruster_warm_up_power_consumed = 0;
        obj.command_actuation_power_consumed =
obj.instantaneous_power_consumption; % Default to regular consumption

        % Initialize fuel consumption properties
        obj.isp = init_data.micro_thruster_ISP; % Get ISP from init data
        obj.fuel_consumed_per_firing = 0;
        obj.total_fuel_consumed = 0;

        % Initialize Storage Variables
        obj.store = [];
        obj.store.true_commanded_thrust =
zeros(mission.storage.num_storage_steps_attitude,1);
        obj.store.control_torque_MT =
zeros(mission.storage.num_storage_steps_attitude,3); % Initialize as empty
numeric array
        obj.store.fuel_consumed_per_firing =
zeros(mission.storage.num_storage_steps_attitude,1);
        obj.store.total_fuel_consumed =
zeros(mission.storage.num_storage_steps_attitude,1);

        % Register with power system

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

end

function obj = func_update_true_SC_micro_thruster_store(obj, mission)
    % Append new thrust data to the storage arrays

    obj.store.true_commanded_thrust(mission.storage.k_storage_attitude,:) =
obj.true_commanded_thrust;
        obj.store.control_torque_MT(mission.storage.k_storage_attitude,:)
= obj.control_torque_MT;

    obj.store.fuel_consumed_per_firing(mission.storage.k_storage_attitude,:) =
obj.fuel_consumed_per_firing;

```

```

obj.store.total_fuel_consumed(mission.storage.k_storage_attitude,:) =
obj.total_fuel_consumed;
end

function func_reset_state(obj)
    % Reset temporary state variables after each time step
    % This ensures clean state for the next time step

    % Reset actuation flags
    obj.command_actuation = false;
    obj.command_executed = false;

    % Reset thruster state for next time step
    obj.commanded_thrust = 0;
    obj.true_commanded_thrust = 0;
    obj.control_torque_MT = [0, 0, 0];

    % Reset fuel consumption for this firing
    % Note: total_fuel_consumed is kept as it's cumulative
    obj.fuel_consumed_per_firing = 0;
end

function func_main_true_SC_micro_thruster(obj, mission, i_SC, ~)
    % Main function for True_SC_Micro_Thruster
    % Controls thrust application, power consumption, and data
generation.

    % Reset fuel consumed this firing
    obj.fuel_consumed_per_firing = 0;

    % Check if the thruster is healthy
    if obj.health == 0
        obj.commanded_thrust = 0; % Thruster is offline, no thrust can
be applied

        obj.command_executed = false; % Mark as not executed
        obj.instantaneous_power_consumption = 0;
        obj.instantaneous_power_consumed = 0;
        return;
    end

    % Fire the thruster if the actuation flag is true
    % (should be set in the function optimize_thruster_dart in sw
attitude control)
    if obj.command_actuation
        % Check warm-up time
        if obj.accumulated_warm_up_time < obj.thruster_warm_up_time
            % Thruster warming up
            obj.instantaneous_power_consumption =
obj.thruster_warm_up_power_consumed;
            obj.instantaneous_power_consumed =
obj.thruster_warm_up_power_consumed;
            obj.accumulated_warm_up_time =
obj.accumulated_warm_up_time + mission.time_step;

```

```

        obj.command_executed = false; % Not yet ready to fire
    else
        % Thruster ready to fire
        obj.instantaneous_power_consumption =
obj.command_actuation_power_consumed;
        obj.instantaneous_power_consumed =
obj.command_actuation_power_consumed;

        % Randomize noise
        current_thruster_noise = obj.thruster_noise * (2*rand() -
1); % Generates a number between 1.0e-04 and - 1.0e-04

        % Compute the true thrust with noise
        obj.true_commanded_thrust = obj.commanded_thrust +
current_thruster_noise;

        % Compute force vector
        force_vector = obj.orientation *
obj.true_commanded_thrust;

        % Compute the torque generated by the thruster
        obj.control_torque_MT = cross(obj.location -
mission.true_SC{1, 1}.true_SC_body.location_COM, force_vector);

        % Apply the generated torque to the spacecraft
        mission.true_SC{i_SC}.true_SC_adc.control_torque = ...
mission.true_SC{i_SC}.true_SC_adc.control_torque + obj.control_torque_MT';

        % Calculate fuel consumption
        g0 = 9.80665; % m/s^2
        obj.fuel_consumed_per_firing = (obj.true_commanded_thrust
* mission.true_time.time_step_attitude) / (obj.isp * g0);
        obj.total_fuel_consumed = obj.total_fuel_consumed +
obj.fuel_consumed_per_firing;

        % Consume fuel from the tank if fuel tank exists
        if isfield(mission.true_SC{i_SC}, 'true_SC_fuel_tank')
&& ...

mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank > 0
        % Find the assigned fuel tank (assuming the first one
for now)
            i_tank = 1;
            % Consume fuel from the tank

mission.true_SC{i_SC}.true_SC_fuel_tank{i_tank}.func_consume_fuel(obj.fuel_consumed_per_f
end

        % Generate data for this actuation step
        obj.instantaneous_data_generated =
obj.instantaneous_data_volume;

        % Mark as executed

```

```

        obj.command_executed = true;
    end

    % Update power system with consumption
    func_update_instantaneous_power_consumed_attitude(mission.true_SC{i_SC}.true_SC_power,
    obj, mission);
    else
        % Thruster not actuating
        obj.commanded_thrust = 0;
        obj.true_commanded_thrust = 0;
        obj.control_torque_MT = [0, 0, 0]; % No torque applied
        obj.command_executed = false;
        obj.instantaneous_power_consumption = 0; % No power used
        obj.instantaneous_power_consumed = 0;
        obj.instantaneous_data_generated = 0; % No data generated
    end

    func_update_true_SC_micro_thruster_store(obj, mission);

    % Add this line to properly reset state after the time step is
complete
    % and all data has been stored
    func_reset_state(obj); % Commented out to avoid multiple resets
end

end

end

```

Published with MATLAB® R2022a

4.9 True_SC_Onboard_Computer

Table of Contents

Class: True_SC_Onboard_Computer	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	1
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3

Class: True_SC_Onboard_Computer

Onboard Computer class for spacecraft

```
classdef True_SC_Onboard_Computer < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate, in kilo bits
per sec (kbps)

processor_utilization % [%] : CPU usage (0-100%)
```

[] Properties: Variables Computed Internally

```
name % [string] 'Onboard Computer i'

health % [integer] Health of computer
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of computer

flag_executive % [Boolean] Executive has told this computer to do its
job

data % Other useful data
```

[] Properties: Storage Variables

```
store
```

```
end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Onboard_Computer(init_data, mission, i_SC,
i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Onboard Computer ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 15; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated; % [kbps]

    if isfield(init_data, 'processor_utilization')
        obj.processor_utilization = init_data.processor_utilization; %
[%]
    else
        obj.processor_utilization = 10; % [%] default utilization
    end

    obj.flag_executive = 0;

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    % Initialize Variables to store
    obj.store = [];
    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
    obj.store.instantaneous_data_rate_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_generated)); % [kbps]
    obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_power_consumed)); % [W]
```

```

        obj.store.processor_utilization =
zeros(mission.storage.num_storage_steps, 1); % [%]

        % Update Storage
        obj = func_update_true_SC_onboard_computer_store(obj, mission);

        % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_onboard_computer_store(obj,
mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage_attitude,:)
= obj.flag_executive; % [Boolean]

        if obj.flag_executive == 1

obj.store.instantaneous_data_rate_generated(mission.storage.k_storage,:) =
obj.instantaneous_data_rate_generated; % [kbps]

obj.store.instantaneous_power_consumed(mission.storage.k_storage,:) =
obj.instantaneous_power_consumed; % [W]

obj.store.processor_utilization(mission.storage.k_storage,:) =
obj.processor_utilization; % [%]
        end
    end

end

```

[] Methods: Main

Update Onboard Computer

```

function obj = func_main_true_SC_onboard_computer(obj, mission, i_SC)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % Operate computer - constant power and data rate regardless
of processor utilization
    end

```

```
        % Update power consumed for spacecraft power budget

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update data generated for spacecraft data handling

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    else
        % Computer is off or malfunctioning
    end

    % Update Storage
    obj = func_update_true_SC_onboard_computer_store(obj, mission);

    % Do not reset flag_executive because computer is always on

end

end

end
```

Published with MATLAB® R2022a

4.10 True_SC_Onboard_Clock

Table of Contents

Class: True_SC_Onboard_Clock	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3

Class: True_SC_Onboard_Clock

Tracks the time onboard the SC

```
classdef True_SC_Onboard_Clock < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate generated
during current time step, in kilo bits (kb) per sec

mode_true_SC_onboard_clock_selector % [string]
% - Simple

measurement_noise % [sec] (1-sigma standard deviation) (Optional)

measurement_wait_time % [sec]
```

[] Properties: Variables Computed Internally

```
name % [string] 'Clock i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

measurement_vector % [sec] [Time, Date]

measurement_time % [sec] SC time when this measurement was taken
```

```
flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job
```

[] Properties: Storage Variables

```
store

end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Onboard_Clock(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Clock ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated; % [kbps]

    if isfield(init_data, 'measurement_noise')
        obj.measurement_noise = init_data.measurement_noise; % [sec]
    else
        obj.measurement_noise = 0; % [sec]
    end

    obj.mode_true_SC_onboard_clock_selector =
init_data.mode_true_SC_onboard_clock_selector; % [string]
    obj.measurement_wait_time = init_data.measurement_wait_time; %
[sec]

    obj.flag_executive = 1;

    this_measurement_noise = obj.measurement_noise*2*(rand-0.5); %
[sec]
    obj.measurement_vector = [(mission.true_time.time +
this_measurement_noise) (mission.true_time.date + this_measurement_noise)]; %
[sec] [Time, Date]
    obj.measurement_time = obj.measurement_vector(1); % [sec]
```

```

        % Initialize Variables to store: measurement_vector
measurement_time
        obj.store = [];

        obj.store.measurement_vector =
zeros(mission.storage.num_storage_steps, length(obj.measurement_vector));
        obj.store.measurement_time =
zeros(mission.storage.num_storage_steps, length(obj.measurement_time));
        obj.store.measurement_noise = obj.measurement_noise; % [sec]

        % Update Storage
        obj = func_update_true_SC_onboard_clock_store(obj, mission);

        % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_onboard_clock_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.measurement_vector(mission.storage.k_storage,:) =
obj.measurement_vector; % [sec]
        obj.store.measurement_time(mission.storage.k_storage,:) =
obj.measurement_time; % [sec]
    end

end

```

[] Methods: Main

Update Clock Time

```

function obj = func_main_true_SC_onboard_clock(obj, mission, i_SC)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % Take measurement

        if (mission.true_time.time - obj.measurement_time) >=
obj.measurement_wait_time

```

```

        % Sufficient time has elapsed for a new measurement

        switch obj.mode_true_SC_onboard_clock_selector

            case 'Simple'
                this_measurement_noise =
obj.measurement_noise*2*(rand-0.5); % [sec]
                obj.measurement_vector = [(mission.true_time.time
+ this_measurement_noise) (mission.true_time.date +
this_measurement_noise)]; % [sec]
                obj.measurement_time = mission.true_time.time; %
[sec]

            otherwise
                error('Clock mode not defined!')
            end

        end

        % Update Power Consumed

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    else
        % Do nothing

    end

    % Update Storage
    obj = func_update_true_SC_onboard_clock_store(obj, mission);

    % Reset Variables
    obj.flag_executive = 0;

end

end

end

```

Published with MATLAB® R2022a

4.11 True_SC_Onboard_Memory

Table of Contents

Class: True_SC_Onboard_Memory	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	1
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3

Class: True_SC_Onboard_Memory

Tracks the onboard Memory state

```
classdef True_SC_Onboard_Memory < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate generated
during current time step, in kilo bits (kb) per sec

maximum_capacity % [kb] : Maximum data storage capacity of the Memeory
```

[] Properties: Variables Computed Internally

```
name % [string] 'Memory i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

instantaneous_capacity % [kb] : Instantaneous capacity of Memeory

state_of_data_storage % [percentage] : SoDS is defined by = 100×
instantaneous_capacity / maximum_capacity
```

[] Properties: Storage Variables

store

end

Methods

methods

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Onboard_Memory(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Memory ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated; % [kbps]

    obj.maximum_capacity = init_data.maximum_capacity; % [kb]
    obj.instantaneous_capacity = 0; % [kb]
    obj.state_of_data_storage = 100*obj.instantaneous_capacity/
obj.maximum_capacity; % [percentage]

    % Initialize Variables to store: instantaneous_capacity
state_of_charge
    obj.store = [];

    obj.store.instantaneous_capacity =
zeros(mission.storage.num_storage_steps, length(obj.instantaneous_capacity));
    obj.store.state_of_data_storage =
zeros(mission.storage.num_storage_steps, length(obj.state_of_data_storage));
    obj.store.maximum_capacity = obj.maximum_capacity; % [kb]

    obj = func_update_true_SC_onboard_memory_store(obj, mission);

    % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);
```

```
end
```

[] Methods: Store

Update the store variable

```
function obj = func_update_true_SC_onboard_memory_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.instantaneous_capacity(mission.storage.k_storage,:)
= obj.instantaneous_capacity; % [kb]
        obj.store.state_of_data_storage(mission.storage.k_storage,:) =
obj.state_of_data_storage; % [percentage]
    end

end
```

[] Methods: Main

Update Memory SoDS

```
function obj = func_main_true_SC_onboard_memory(obj, mission, i_SC)

    %           if obj.instantaneous_capacity <= 0
    %           obj.instantaneous_capacity = 1e-3; % [kb]
    %           end

    obj.state_of_data_storage = 100*obj.instantaneous_capacity/
obj.maximum_capacity; % [percentage]

    obj = func_update_true_SC_onboard_memory_store(obj, mission);

    % Update Power Consumed

    func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update Data Generated

    func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

end

end
```

Published with MATLAB® R2022a

4.12 True_SC_Radio_Antenna

Table of Contents

Class: True_SC_Radio_Antenna	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	2
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	4
[] Methods: Main	5

Class: True_SC_Radio_Antenna

Tracks the Radio Antennas

```
classdef True_SC_Radio_Antenna < handle
```

Properties

properties

[] Properties: Initialized Variables

```
    antenna_type % [string]
    % - 'Dipole'
    % - 'High Gain'

    location % [m] : Location of sensor, in body frame B

    orientation % [unit vector] : Normal vector from location

    mode_true_sc_radio_antenna_selector % [string]
    % - TX
    % - RX

    TX_power_consumed % [Watts] : Power consumed during TX

    RX_power_consumed % [Watts] : Power consumed during RX

    base_data_rate_generated % [kbps] : Data rate, in kilo bits per sec
(kbps) due to Health Keeping

    % Optional (only for Link Margin Calculations)

    antenna_gain % [dB] gain of Earth receiver

    noise_temperature % [K] temperature noise
```

```
beamwidth % [MHz] receiver beamwidth

energy_bit_required % [dB] Minimum energy bit required

line_loss % [dB] Loss due to pointing or others

coding_gain % [dB] Coding gain
```

[] Properties: Variables Computed Internally

```
name % [string] 'Radio Antenna i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate, in kilo bits
per sec (kbps) due to RX

instantaneous_data_rate_removed % [kbps] : Data rate, in kilo bits per
sec (kbps) due to TX

data % Other useful data

maximum_data_rate
```

[] Properties: Storage Variables

```
store

end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Radio_Antenna(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
```

```

        obj.name = init_data.name;
    else
        obj.name = ['Radio Antenna ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]
    obj.flag_executive = 0;
    %
    obj.antenna_type = init_data.antenna_type;
    %
    % obj.mode_true_SC_radio_antenna_selector =
init_data.mode_true_SC_radio_antenna_selector;
    %
    obj.TX_power_consumed = init_data.TX_power_consumed; % [Watts]
    obj.RX_power_consumed = init_data.RX_power_consumed; % [Watts]
    obj.instantaneous_power_consumed = obj.TX_power_consumed; %
[Watts] Will be modified dynamically but a value is needed to register

    obj.location = init_data.location; % [m]
    obj.orientation = init_data.orientation; % [unit vector]

    % switch obj.mode_true_SC_radio_antenna_selector
    %     case 'TX'
    %         obj.instantaneous_power_consumed =
obj.TX_power_consumed; % [Watts]
    %     case 'RX'
    %         obj.instantaneous_power_consumed =
obj.RX_power_consumed; % [Watts]
    %     otherwise
    %         error('Should not reach here!')
    % end

    if isfield(init_data, 'antenna_gain')

        obj.antenna_gain = init_data.antenna_gain; % [dB]
        obj.noise_temperature = init_data.noise_temperature; % [K]
        obj.beamwidth = init_data.beamwidth; % [MHz]
        obj.energy_bit_required = init_data.energy_bit_required; %
[dB]

        obj.coding_gain = init_data.coding_gain; % [dB]

    end
    %
    obj.base_data_rate_generated =
init_data.base_data_rate_generated; % [kbps]
    obj.instantaneous_data_rate_generated =
obj.base_data_rate_generated; % [kbps]
    obj.instantaneous_data_rate_removed = 0; % [kbps]
    obj.maximum_data_rate = obj.maximum_data_rate; % [kbps]

    % if isfield(init_data, 'data')
    %     obj.data = init_data.data;
    % else

```

```

        %      obj.data = [];
    % end
    %
    % Initialize Variables to store
    obj.store = [];

    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive)); %
[integer]
    obj.store.mode_TX_RX = zeros(mission.storage.num_storage_steps,
1); % [integer]
    obj.store.instantaneous_data_rate_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_generated)); % [kbps]
    obj.store.instantaneous_data_rate_removed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_removed)); % [kbps]
    obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_power_consumed)); % [W]

    % Update Storage
    obj = func_update_true_SC_radio_antenna_store(obj, mission);
    %
    % % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);
    %
    % % Update SC Data Handling Class (Generated and Removed)

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

func_initialize_list_HW_data_removed(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_radio_antenna_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage,:) =
obj.flag_executive; % [integer]

        if obj.flag_executive == 1

obj.store.instantaneous_data_rate_generated(mission.storage.k_storage,:) =
obj.instantaneous_data_rate_generated; % [kbps]

```

```

obj.store.instantaneous_data_rate_removed(mission.storage.k_storage,:) =
obj.instantaneous_data_rate_removed; % [kbps]

obj.store.instantaneous_power_consumed(mission.storage.k_storage,:) =
obj.instantaneous_power_consumed; % [W]

        switch obj.mode_true_SC_radio_antenna_selector
            case 'TX'
                obj.store.mode_TX_RX(mission.storage.k_storage,1)
= 1;
            case 'RX'
                obj.store.mode_TX_RX(mission.storage.k_storage,1)
= 2;
            otherwise
                error('[True_SC_Radio_Antenna] Should not reach
here!')
            end
        end
    end
end
end
end

```

[] Methods: Main

Update all variables

```

function obj = func_main_true_SC_radio_antenna(obj, mission, i_SC)

    if (obj.flag_executive == 1) && (obj.health == 1)

        switch obj.mode_true_SC_radio_antenna_selector
            case 'TX'
                obj.instantaneous_power_consumed =
obj.TX_power_consumed; % [Watts]
            case 'RX'
                obj.instantaneous_power_consumed =
obj.RX_power_consumed; % [Watts]
            otherwise
                error('[True_SC_Radio_Antenna] Should not reach
here!')
            end

        % Update SC Power Class

        func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,obj,
mission);

        func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);
    end
end

```

```

        % Update SC Data Handling Class (Generated and Removed)

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

func_update_instantaneous_data_removed(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    end

    % Update Storage
    obj = func_update_true_SC_radio_antenna_store(obj, mission);

    % Reset All Variables
    obj.flag_executive = 0;
    obj.instantaneous_data_rate_generated =
obj.base_data_rate_generated; % [kbps]
    obj.instantaneous_data_rate_removed = 0; % [kbps]

    end

end

end

```

Published with MATLAB® R2022a

4.13 True_SC_Reaction_Wheel

Table of Contents

Class: True_SC_Reaction_Wheel	1
Properties	1
[EXTERNAL] Parameters set by systems engineers	1
[INTERNAL] Computed internally	1
Methods	2
Constructor	2
Update Storage	4

Class: True_SC_Reaction_Wheel

Represents a single reaction wheel for spacecraft attitude control

```
classdef True_SC_Reaction_Wheel < handle
```

Properties

```
properties
```

[EXTERNAL] Parameters set by systems engineers

```
name                % Name of the reaction wheel
health              % Health status (0: Not working, 1:
Working)
location            %[m] : Location of the reaction wheel in
the body frame
orientation          %[Unit vector] : Axis of rotation in the
body frame
max_torque          %[Nm] : Maximum torque the wheel can
apply
max_angular_velocity %[rad/s] : Maximum angular velocity
moment_of_inertia   %[kg·m^2] : Moment of inertia of the
wheel
radius              %[m] radius of 1 RW
mass                %[kg] : Mass of the reaction wheel
power_consumed_angular_velocity_array % [power_array ; velocity_array]
angular_velocity_noise %[rad/s] : Random noise added to
commanded torque
rpm_values           % RPM values for interpolation
torque_values        % Torque values for interpolation
power_matrix         % Power consumption matrix for
interpolation
```

[INTERNAL] Computed internally

```
temperature         %[deg C]
total_momentum       %[kg·m^2/s] : Total momentum of the wheel
```

```

        inertia_matrix           %[kg·m^2] : Inertia matrix of the wheel
        commanded_angular_acceleration % [rad/s] : Desired angular velocity of
the wheel
        actual_angular_acceleration % [rad/s] : Actual realised angular
velocity of the wheel
        min_acceleration          % [rad/s^2]
        angular_velocity           %[rad/s] : Current angular velocity of
the wheel
        commanded_torque          %[Nm] : Torque commanded by the control
system
        actual_torque              %[Nm] : Actual torque considering
saturation and noise
        saturated                  %[Bool] Is the wheel currently
saturated ?
        instantaneous_power_consumption %[Watts] : Power consumed during
operation
        instantaneous_power_consumed   %[Watts] : Alias for power tracking
system
        command_actuation_power_consumed %[Watts] : Power consumed during
actuation
        instantaneous_data_generated %[Kb] : Data generated during the current
time step
        store                       %[Struct] : Store of the reaction wheel
        flag_executive              %[Bool] : Is the wheel currently
executing a command ?
        envelope_ratio              %[Ratio] : Ratio of the minimum distance
between wheels to the radius of the wheel
        maximum_torque              %[Nm] : Maximum torque the wheel can
apply
        momentum_capacity           %[kg·m^2/s] : Momentum capacity of the
wheel
        maximum_acceleration        %[rad/s^2] : Maximum acceleration the
wheel can apply
    end

```

Methods

methods

Constructor

```

function obj = True_SC_Reaction_Wheel(init_data, mission, i_SC, i_RW)

    obj.name = ['Reaction Wheel ', num2str(i_RW)];

    obj.health = true;
    obj.location = init_data.location;
    obj.orientation = init_data.orientation;
    obj.max_angular_velocity = init_data.max_angular_velocity;
    obj.radius = init_data.radius;
    obj.mass = init_data.mass;

```

```

        % Calculate moment of inertia
        % For a disk rotating around its center axis : 1/2*m*r^2
        obj.moment_of_inertia = 0.5 * obj.mass * obj.radius^2;

        obj.power_consumed_angular_velocity_array =
init_data.power_consumed_angular_velocity_array;

        % Torque envelope
        if isfield(init_data, 'max_torque')
            % Maximum Torque and Momentum Envelopes for Reaction Wheel
Arrays
            % https://ntrs.nasa.gov/api/citations/20110015369/
downloads/20110015369.pdf

assert(ismember(mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
[3 4 5 6]), 'Number of reaction wheel must be 3, 4, 5 or 6')
        switch
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
        case 3
            d_min = 1; % d_12
        case 4
            d_min = 1.633; % d_12 = d_13
        case 5
            d_min = 5/8 * 2.667; % d_24 = d_42
        case 6
            d_min = 2.667; % d_23
        end
        obj.envelope_ratio = d_min;
        obj.maximum_torque = init_data.max_torque;

        % Get maximum acceleration from init_data if provided,
otherwise calculate it
        if isfield(init_data, 'maximum_acceleration')
            obj.maximum_acceleration = init_data.maximum_acceleration;
        else
            obj.maximum_acceleration = obj.maximum_torque /
obj.moment_of_inertia; % rad/s^2
        end
    else
        obj.envelope_ratio = 1;
        obj.maximum_torque = Inf;
        obj.momentum_capacity = Inf;
        obj.maximum_acceleration = Inf;
    end
end

obj.min_acceleration = 1; % rad/s^2

obj.angular_velocity_noise = init_data.angular_velocity_noise;

% Initialize dynamic state
obj.angular_velocity = 0;
obj.commanded_angular_acceleration = 0;
obj.saturated = false;
obj.flag_executive = false;

```

```

obj.instantaneous_power_consumption = 0;
obj.instantaneous_power_consumed = 0;

obj.actual_torque = [0,0,0];

% Initialize storage
obj.total_momentum = 0;

obj.store = [];
obj.store.angular_velocity =
zeros(mission.storage.num_storage_steps_attitude, 1);
obj.store.torque =
zeros(mission.storage.num_storage_steps_attitude, 1);
obj.store.saturated =
zeros(mission.storage.num_storage_steps_attitude, 1);

obj.store.commanded_angular_acceleration =
zeros(mission.storage.num_storage_steps_attitude, 1);
obj.store.actual_angular_acceleration =
zeros(mission.storage.num_storage_steps_attitude, 1);
obj.store.actual_torque =
zeros(mission.storage.num_storage_steps_attitude, 3);

obj.store.max_angular_velocity = obj.max_angular_velocity;

% Register with power system

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

end

```

Update Storage

```

function obj = func_update_reaction_wheel_store(obj, mission)
    obj.store.angular_velocity(mission.storage.k_storage_attitude, :)
= obj.angular_velocity;
    obj.store.actual_torque(mission.storage.k_storage_attitude, :) =
obj.actual_torque;

obj.store.commanded_angular_acceleration(mission.storage.k_storage_attitude, :)
= obj.commanded_angular_acceleration;
    obj.store.saturated(mission.storage.k_storage_attitude, :) =
obj.saturated;
end

function func_main_true_reaction_wheel(obj, mission, i_SC, ~)
% Main function that runs the reaction wheel simulation
if ~obj.health
    obj.commanded_angular_acceleration = 0;
    obj.angular_velocity = 0;
    obj.instantaneous_power_consumption = 0;
    obj.instantaneous_power_consumed = 0;

```

```

        obj.instantaneous_data_generated = 0;
        obj.actual_torque = [0,0,0];
        obj.actual_angular_acceleration = 0;
        return;
    end

    if(obj.flag_executive && abs(obj.commanded_angular_acceleration) >
0)

        % Limit commanded acceleration
        if abs(obj.commanded_angular_acceleration) >
obj.maximum_acceleration
            obj.commanded_angular_acceleration =
sign(obj.commanded_angular_acceleration) * obj.maximum_acceleration;
        end

        % Calculate torque directly from acceleration (# = I*#)
        torque_magnitude = obj.moment_of_inertia *
obj.commanded_angular_acceleration;
        obj.actual_torque = torque_magnitude * obj.orientation;

        % Update velocity with noise and limits
        if obj.commanded_angular_acceleration ~= 0
            % Add noise only during active commands
            current_vel_noise = obj.angular_velocity_noise * (2*rand()
- 1);

            else
                current_vel_noise = 0; % No noise when idle
            end

            % Calculate the new velocity without limits
            unconstrained_velocity = obj.angular_velocity +
obj.commanded_angular_acceleration * mission.true_time.time_step_attitude +
current_vel_noise;

            % Apply rate limiting to prevent dramatic velocity changes in
a single step
            % Maximum allowed change in velocity per time step (10% of max
velocity is a reasonable value)
            max_velocity_change = 0.1 * obj.max_angular_velocity;

            % ENHANCED SAFETY: Add more conservative limits for direction
reversals
            % Detect potential reversal (wheel going one way, command
pushing it the complete other way)
            if sign(unconstrained_velocity) ~= sign(obj.angular_velocity)
&& abs(obj.angular_velocity) > 0.3 * obj.max_angular_velocity
                % This is a potential direction reversal and the wheel is
at significant speed
                % Reduce maximum allowed change dramatically for safer
deceleration
                max_velocity_change = 0.02 * obj.max_angular_velocity;

                % Log this event to help debugging

```

```

        disp(['REACTION WHEEL SAFETY: Detected potential rapid
direction reversal for ', obj.name]);
        disp(['Current velocity: ',
num2str(obj.angular_velocity), ' rad/s']);
        disp(['Commanded acceleration: ',
num2str(obj.commanded_angular_acceleration), ' rad/s^2']);
        disp(['Limiting velocity change to ',
num2str(max_velocity_change), ' rad/s per time step']);
    end

    % Limit the velocity change
    if abs(unconstrained_velocity - obj.angular_velocity) >
max_velocity_change
        limited_velocity = obj.angular_velocity +
sign(unconstrained_velocity - obj.angular_velocity) * max_velocity_change;
    else
        limited_velocity = unconstrained_velocity;
    end

    % Apply absolute velocity limits (don't exceed max angular
velocity)
    new_velocity = max(min(limited_velocity,
obj.max_angular_velocity), -obj.max_angular_velocity);

    % If the velocity didn't change much, leave it as is to avoid
numerical issues
    velocity_tolerance = 1e-4;
    if abs(new_velocity - obj.angular_velocity) <
velocity_tolerance
        new_velocity = obj.angular_velocity; % Clamp near-zero
velocities
    end

    % Calculate the actual acceleration that occurred (for
telemetry)
    obj.actual_angular_acceleration = (new_velocity -
obj.angular_velocity) / mission.true_time.time_step_attitude;

    % Update the angular velocity
    obj.angular_velocity = new_velocity;

    % Check for saturation
    % > If reaches 80% of max angular velocity
    obj.saturated = (abs(obj.angular_velocity) >=
obj.max_angular_velocity * 0.80);

    % Update spacecraft torque directly
    mission.true_SC{i_SC}.true_SC_adc.control_torque = ...
        mission.true_SC{i_SC}.true_SC_adc.control_torque +
obj.actual_torque';

    % Update power and data
    obj.instantaneous_power_consumption = abs(obj.actual_torque *
obj.angular_velocity);

```

```

        % Calculate individual wheel momentum:  $h = I * \# * \text{direction}$ 
        % The wheel momentum is a vector along the wheel's spin axis
        (orientation)
        momentum_magnitude = obj.moment_of_inertia *
obj.angular_velocity;
        obj.total_momentum = momentum_magnitude * obj.orientation;

        % Update power and data
        power_consumptions =
obj.power_consumed_angular_velocity_array(1, :);
        angular_velocities =
obj.power_consumed_angular_velocity_array(2, :);

        % Ensure the array is sorted by angular velocity
        [angular_velocities, sortIdx] = sort(angular_velocities);
        power_consumptions = power_consumptions(sortIdx);

        % Interpolate to find the power consumption for the current
        angular_velocity
        if obj.angular_velocity < min(angular_velocities)
            obj.instantaneous_power_consumption =
power_consumptions(1);
        elseif obj.angular_velocity > max(angular_velocities)
            obj.instantaneous_power_consumption =
power_consumptions(end);
        else
            obj.instantaneous_power_consumption
= interp1(angular_velocities, power_consumptions,
obj.angular_velocity, 'linear');
        end
        % Set property for power tracking system
        obj.instantaneous_power_consumed =
obj.instantaneous_power_consumption;

        % Update the power system with this consumption

func_update_instantaneous_power_consumed_attitude(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        obj.instantaneous_data_generated =
obj.instantaneous_power_consumption / 10;

    else
        obj.commanded_angular_acceleration = 0;
        obj.instantaneous_power_consumption = 0;
        obj.instantaneous_power_consumed = 0;
        obj.instantaneous_data_generated = 0;
    end

    % Update storage

```

```
        func_update_reaction_wheel_store(obj, mission);

        % CRITICAL FIX: Reset command flags at the end of each cycle
        % This prevents commands from persisting indefinitely between
cycles
        obj.flag_executive = false;
        obj.commanded_angular_acceleration = 0;
    end
end
end
```

Published with MATLAB® R2022a

4.14 True_SC_Science_Processor

Table of Contents

Class: True_SC_Science_Processor	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	4

Class: True_SC_Science_Processor

Tracks the Science Processor

```
classdef True_SC_Science_Processor < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate, in kilo bits
per sec (kbps)

instantaneous_data_removed_per_sample % [kb] : Data in kilo bits (kb)

flag_show_science_processor_plot % [Boolean] : 1 = Shows the Science
Processor plot

mode_true_sc_science_processor_selector % [string] Select which Mode
to run
```

[] Properties: Variables Computed Internally

```
name % [string] 'Generic Sensor i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator
```

```
        flag_executive % [Boolean] Executive has told this sensor/actuator to  
do its job
```

```
        data % Other useful data
```

[] Properties: Storage Variables

```
        store  
  
    end
```

Methods

```
    methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Science_Processor(init_data, mission, i_SC,  
i_HW)  
  
    if isfield(init_data, 'name')  
        obj.name = init_data.name;  
    else  
        obj.name = ['Science Processor ', num2str(i_HW)];  
    end  
  
    obj.health = 1;  
    obj.temperature = 10; % [deg C]  
  
    obj.instantaneous_power_consumed =  
init_data.instantaneous_power_consumed; % [W]  
    obj.instantaneous_data_rate_generated =  
init_data.instantaneous_data_rate_generated; % [kbps]  
    obj.instantaneous_data_removed_per_sample =  
init_data.instantaneous_data_removed_per_sample; % [kb]  
    obj.flag_show_science_processor_plot =  
init_data.flag_show_science_processor_plot; % [Boolean]  
    obj.mode_true_SC_science_processor_selector =  
init_data.mode_true_SC_science_processor_selector; % [string]  
  
    obj.flag_executive = 0;  
  
    if isfield(init_data, 'data')  
        obj.data = init_data.data;  
    else  
        obj.data = [];  
    end  
  
    % Initialize Variables to store: measurement_vector  
    obj.store = [];
```

```

        obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
        obj.store.instantaneous_data_rate_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_generated)); % [kbps]
        obj.store.instantaneous_data_removed_per_sample =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_removed_per_sample)); % [kb]
        obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_power_consumed)); % [W]

        % Additional Science Processor Variables
        switch obj.mode_true_SC_science_processor_selector

            case 'Nightingale'
                obj =
func_true_SC_science_processor_Nightingale_constructor(obj, mission, i_SC);

            otherwise
                % Do nothing!
        end

        % Update Storage
        obj = func_update_true_SC_science_processor_store(obj, mission);

        % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

func_initialize_list_HW_data_removed(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_science_processor_store(obj,
mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage_attitude,:)
= obj.flag_executive; % [Boolean]

        if obj.flag_executive == 1

```

```

obj.store.instantaneous_data_rate_generated(mission.storage.k_storage,:) =
obj.instantaneous_data_rate_generated; % [kbps]

obj.store.instantaneous_data_removed_per_sample(mission.storage.k_storage,:)
= obj.instantaneous_data_removed_per_sample; % [kb]

obj.store.instantaneous_power_consumed(mission.storage.k_storage,:) =
obj.instantaneous_power_consumed; % [W]
    end
end
end

```

[] Methods: Main

Update Science Processor

```

function obj = func_main_true_SC_science_processor(obj, mission, i_SC)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % Take measurement

        if
isfield(obj.data, 'instantaneous_power_consumed_per_SC_mode')
            obj.instantaneous_power_consumed =
obj.data.instantaneous_power_consumed_per_SC_mode(mission.true_SC{i_SC}.software_SC_execu
[W]
        end

        switch obj.mode_true_SC_science_processor_selector

            case 'Nightingale'
                obj = func_true_SC_science_processor_Nightingale(obj,
mission, i_SC);

            otherwise
                error('Should not reach here!')
            end

        % Update Power Consumed

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

        % Update Data Handling

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

func_update_instantaneous_data_removed(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

```

```

        else
            % If not active, set power to a low standby value
            obj.instantaneous_power_consumed =
obj.instantaneous_power_consumed * 0.1; % 10% of normal power when in standby

            % Still update power system even when in standby

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);
        end

        % Update Storage
obj = func_update_true_SC_science_processor_store(obj, mission);

        % Reset Variables
obj.flag_executive = 0;

    end

end

end

```

Published with MATLAB® R2022a

4.15 True_SC_Science_Radar

Table of Contents

Class: True_SC_Science_Radar	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	2
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	5
[] Methods: Main	5
[] Methods: DROID Radar	6
[] Methods: Visualize Radar Coverage	7

Class: True_SC_Science_Radar

Tracks the onboard Radar measurements

```
classdef True_SC_Science_Radar < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_rate_generated % [kbps] : Data rate, in kilo bits
per sec (kbps)

mode_true_SC_science_radar_selector % [string]
% - DROID

measurement_wait_time % [sec]

location % [m] : Location of sensor, in body frame B

orientation % [unit vector] : Normal vector from location

field_of_view % [deg] : Field of view (FOV) of the radar in deg
% Set to 0 to select the closest point

flag_show_radar_plot % [Boolean] : 1 = Shows the radar plot
wait_time_visualize_SC_radar_coverage_during_sim % [sec] (Optional)

num_points % [integer] Number of points in mesh
```

[] Properties: Variables Computed Internally

```
name % [string] 'Radar i'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

measurement_vector % [Image]

measurement_time % [sec] SC time when this measurement was taken

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

pos_points % [unit vector] Location of points in mesh

spherical_points % [unit vector] Location of points in Sphere

monostatic_observed_point % [array] Monostatic: which point is
observed how many times

monostatic_num_point_observed % [integer] Monostatic: total number of
points observed

prev_time_visualize_SC_radar_coverage_during_sim % [sec]

data % Other useful data
```

[] Properties: Storage Variables

```
store

end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = True_SC_Science_Radar(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Radar ', num2str(i_HW)];
    end
```

```

obj.health = 1;
obj.temperature = 10; % [deg C]

obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
obj.mode_true_SC_science_radar_selector =
init_data.mode_true_SC_science_radar_selector; % [string]
obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated; % [kbps]

obj.measurement_wait_time = init_data.measurement_wait_time; %
[sec]
obj.measurement_time = -inf; % [sec]

obj.flag_executive = 0;

obj.location = init_data.location; % [m]
obj.orientation = init_data.orientation; % [unit vector]

obj.field_of_view = init_data.field_of_view; % [deg]

obj.flag_show_radar_plot = init_data.flag_show_radar_plot; %
[Boolean]

if
isfield(init_data, 'wait_time_visualize_SC_radar_coverage_during_sim')
    obj.wait_time_visualize_SC_radar_coverage_during_sim =
init_data.wait_time_visualize_SC_radar_coverage_during_sim;
else
    obj.wait_time_visualize_SC_radar_coverage_during_sim = 0; %
[sec]
end
obj.prev_time_visualize_SC_radar_coverage_during_sim = -inf;

if isfield(init_data, 'data')
    obj.data = init_data.data;
else
    obj.data = [];
end

% load the science points
obj.num_points = init_data.num_points; % [integer]
obj.pos_points = func_load_science_points_v2(obj.num_points);

obj.spherical_points = zeros(obj.num_points, 2);
for i = 1:1:obj.num_points

    % sph =
    Cartesian2Spherical(obj.pos_points(i,:)); % [r, theta, phi] in radians
    % longitude = rad2deg(sph(3)); % [deg]
    % latitude = rad2deg(sph(2)); % [deg]
    % latitude = latitude - 90; % [deg]

```

```

        [radius, lon, lat] = cspice_reclat(obj.pos_points(i,:)); %
        [radius, longitude [rad], latitude [rad] ]

        obj.spherical_points(i,:) = [rad2deg(lon), rad2deg(lat)];
    end

    % monostatic data
    obj.monostatic_observed_point = zeros(1,obj.num_points);
    obj.monostatic_num_point_observed = 0;

    % Initialize Variables to store: monostatic_num_point_observed
    obj.store = [];

    obj.store.monostatic_num_point_observed =
zeros(mission.storage.num_storage_steps,
length(obj.monostatic_num_point_observed));
    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
    obj.store.instantaneous_data_rate_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_data_rate_generated)); % [kbps]
    obj.store.instantaneous_power_consumed =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_power_consumed)); % [W]

    % Update Storage
    obj = func_update_true_SC_science_radar_store(obj, mission);

    % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    % Store video of func_visualize_SC_orbit_during_sim
    if (mission.storage.plot_parameters.flag_save_video == 1) &&
(obj.flag_show_radar_plot == 1)
        obj.data.video_filename = [mission.storage.output_folder,
mission.name, '_SC', num2str(i_SC), '_Radar', num2str(i_HW), '.mp4'];
        obj.data.myVideo =
VideoWriter(obj.data.video_filename, 'MPEG-4');
        obj.data.myVideo.FrameRate = 30; % Default 30
        obj.data.myVideo.Quality = 100; % Default 75
        open(obj.data.myVideo);
    end
end
end

```

[] Methods: Store

Update the store variable

```
function obj = func_update_true_SC_science_radar_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1

obj.store.monostatic_num_point_observed(mission.storage.k_storage,:) =
obj.monostatic_num_point_observed; % [integer]
        obj.store.flag_executive(mission.storage.k_storage,:) =
obj.flag_executive; % [integer]

        if obj.flag_executive == 1

obj.store.instantaneous_data_rate_generated(mission.storage.k_storage,:) =
obj.instantaneous_data_rate_generated; % [kbps]

obj.store.instantaneous_power_consumed(mission.storage.k_storage,:) =
obj.instantaneous_power_consumed; % [W]
            end

        end

    end

end
```

[] Methods: Main

Update Radar

```
function obj = func_main_true_SC_science_radar(obj, mission, i_SC,
i_HW)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % Take measurement

        if (mission.true_time.time - obj.measurement_time) >=
obj.measurement_wait_time

            % Sufficient time has elapsed for a new measurement
obj.measurement_time = mission.true_time.time; % [sec]

            switch obj.mode_true_SC_science_radar_selector

                case 'DROID'
                    obj = func_true_SC_science_radar_DROID(obj,
mission, i_SC);

                case 'Nightingale'
                    obj = func_true_SC_science_radar_Nightingale(obj,
mission, i_SC);

                otherwise
```

```

                                error('Radar mode not defined!')
                                end

                                obj.prev_time_visualize_SC_radar_coverage_during_sim = -
inf;

                                else
                                    % Data not generated in this time step
                                    % Data is only generated when a radar measurement is
performed

                                end

                                if
isfield(obj.data, 'instantaneous_power_consumed_per_SC_mode')
                                    obj.instantaneous_power_consumed =
obj.data.instantaneous_power_consumed_per_SC_mode(mission.true_SC{i_SC}.software_SC_execu
[W]
                                end

                                % Update Power Consumed

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

                                else
                                    % Do nothing

                                end

                                % Plot Radar Coverage
                                if (obj.flag_show_radar_plot == 1) &&
(mission.true_SC{i_SC}.software_SC_executive.time -
obj.prev_time_visualize_SC_radar_coverage_during_sim >=
obj.wait_time_visualize_SC_radar_coverage_during_sim )
                                    obj = func_visualize_SC_radar_coverage_during_sim(obj,
mission, i_SC, i_HW);
                                end

                                % Update Storage
                                obj = func_update_true_SC_science_radar_store(obj, mission);

                                % Reset Variables
                                obj.flag_executive = 0;

                                end

```

[] Methods: DROID Radar

Execute DROID Radar code

```
function obj = func_true_SC_science_radar_DROID(obj, mission, i_SC)
```

```

        i_target =
mission.true_SC{i_SC}.true_SC_navigation.index_relative_target;

        this_pos_points = (mission.true_target{i_target}.rotation_matrix *
obj.pos_points')'; % Rotated unit vectors

        SC_pos_normalize =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_navigation.position -
mission.true_target{i_target}.position); % SC unit vector

        dot_prod_angle_array = real(acosd(this_pos_points *
SC_pos_normalize'))'; % [deg] angle between the vectors SC-SB and mesh_point-
SB

        if obj.field_of_view == 0
            % Use closest point
            [min_angle,index_point_science_mesh] =
min(dot_prod_angle_array);

        else
            % Use all points within field of view
            index_point_science_mesh = find(dot_prod_angle_array <=
obj.field_of_view);

        end

        % MONOSTATIC

        for i = 1:1:length(index_point_science_mesh)

            if obj.monostatic_observed_point(index_point_science_mesh(i))
== 0
                obj.monostatic_num_point_observed =
obj.monostatic_num_point_observed + 1;
                end

                obj.monostatic_observed_point(index_point_science_mesh(i)) =
obj.monostatic_observed_point(index_point_science_mesh(i)) + 1;

            end

            % Update Data Generated

        func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

        end

```

[] Methods: Visualize Radar Coverage

Visualize all SC attitude orbit during simulation

```

function obj = func_visualize_SC_radar_coverage_during_sim(obj,
mission, i_SC, i_HW)

    obj.prev_time_visualize_SC_radar_coverage_during_sim =
mission.true_SC{i_SC}.software_SC_executive.time; % [sec]

    plot_handle = figure( (7*i_SC) + i_HW);
    clf
    set(plot_handle,'Color',[1 1 1]);
    set(plot_handle,'units','normalized','outerposition',[0 0 1 1])
    set(plot_handle,'PaperPositionMode','auto');

    time_sim_elapsed = seconds(mission.true_time.time -
mission.true_time.t_initial);
    time_sim_elapsed.Format = 'dd:hh:mm:ss';

    sgtitle(['SC ',num2str(i_SC),', Radar ',num2str(i_HW),'
Coverage = ',num2str(round(100 * obj.monostatic_num_point_observed /
obj.num_points,1)), ' %, Simulation Time =
',char(time_sim_elapsed)], 'FontSize',mission.storage.plot_parameters.title_font_size,'Font

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%
% % 3D Radar Vizualization % %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%

    subplot(1,2,1)
    hold on

    for i_SC = 1:1:mission.num_SC

        plot3(mission.true_SC{i_SC}.true_SC_navigation.position_relative_target(1),
mission.true_SC{i_SC}.true_SC_navigation.position_relative_target(2),
mission.true_SC{i_SC}.true_SC_navigation.position_relative_target(3), 's','MarkerSize',15

            if (obj.flag_executive == 1)
                % Plot Radar Orientation
                this_location =
mission.true_SC{i_SC}.true_SC_navigation.position_relative_target;
                this_orientation =
(mission.true_SC{i_SC}.true_SC_adc.rotation_matrix * obj.orientation)';

                quiver3(this_location(1), this_location(2),
this_location(3), this_orientation(1), this_orientation(2),
this_orientation(3), ...

                'LineWidth',3,'DisplayName',obj.name,'Color',rgb('Orange'), 'AutoScaleFactor',200*mission
            end

        end
    end

```

```

        i_target =
mission.true_SC{i_SC}.true_SC_navigation.index_relative_target;
        func_plot_target_shape(i_target, mission);

        this_pos_points = mission.true_target{i_target}.radius *
(mission.true_target{i_target}.rotation_matrix * obj.pos_points)'; % [km]

        % Define a colormap
        cmap = jet(max(obj.monostatic_observed_point)+1); % Use a colormap
with max(obj.monostatic_observed_point)+1 colors

        % Map the values to colors
        colors = cmap(1+obj.monostatic_observed_point', :);

        scatter3(this_pos_points(:,1), this_pos_points(:,2),
this_pos_points(:,3), 10, colors, 'filled','DisplayName','Radar Points'); %
50 is the size of the markers

        grid on

        axis equal
        legend('Location','southwest')
        xlabel('X axis [km]')
        ylabel('Y axis [km]')
        zlabel('Z axis [km]')

set(gca, 'FontSize',mission.storage.plot_parameters.standard_font_size,'FontName',mission
        title('3D Radar Coverage in Target-centered Rotating
Frame','FontSize',mission.storage.plot_parameters.standard_font_size)

        %               view(3)
        view(-40,-30)
        axis equal

        hold off

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % % 2D Radar Vizualization % %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        subplot(1,2,2)
        hold on

        scatter(obj.spherical_points(:,2), obj.spherical_points(:,1), 10,
colors, 'filled');

        for i_SC = 1:1:mission.num_SC

                %               sph =
Cartesian2Spherical(mission.true_target{i_target}.rotation_matrix' *

```

```

mission.true_SC{i_SC}.true_SC_navigation.position_relative_target'); % [r,
theta, phi] in radians
    %             longitude = rad2deg(sph(3)); % [deg]
    %             latitude = rad2deg(sph(2)); % [deg]
    %             latitude = latitude - 90; % [deg]

    [radius, lon, lat] =
cspice_reclat(mission.true_target{i_target}.rotation_matrix' *
mission.true_SC{i_SC}.true_SC_navigation.position_relative_target'); %
[radius, longitude [rad], latitude [rad] ]
    plot(rad2deg(lat),
rad2deg(lon), 's', 'MarkerSize', 15, 'MarkerFaceColor', rgb('Gray'), 'DisplayName', mission.t
end

    % Add colorbar to show mapping
    colorbar;
    caxis([0 max(obj.monostatic_observed_point)+1]);

    axis equal
    ylabel('Longitude [deg]')
    xlabel('Latitude [deg]')

set(gca, 'FontSize', mission.storage.plot_parameters.standard_font_size, 'FontName', mission
    title('2D Radar Coverage in Target-centered Static
Frame', 'FontSize', mission.storage.plot_parameters.standard_font_size)

drawnow limitrate

    if (mission.storage.plot_parameters.flag_save_video == 1) &&
(mission.flag_stop_sim == 0)
        open(obj.data.myVideo);
        writeVideo(obj.data.myVideo, getframe(plot_handle));
    end

    if (mission.storage.plot_parameters.flag_save_plots == 1) &&
(mission.flag_stop_sim == 1)
        saveas(plot_handle, [mission.storage.output_folder,
mission.name, '_SC', num2str(i_SC), '_Radar', num2str(i_HW), '.png'])
    end

end

end

end
end

```

4.16 True_SC_Solar_Panel

Table of Contents

Class: True_SC_Solar_Panel	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	2
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	5
[] Methods: Main	5
[] Methods: Instantaneous Power Generated	6

Class: True_SC_Solar_Panel

SC's Solar Panels

```
classdef True_SC_Solar_Panel < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed  
(irrespective of whether it is generating power or not)
```

```
instantaneous_data_rate_generated % [kbps] : Data rate generated  
during current time step, in kilo bits (kb) per sec
```

```
mass % [kg] : Mass of ith solar panel
```

```
shape_model % : Shape model of Solar Panel  
% - Vertices [m] : Position of vertices in body frame B  
% - Faces : Triplet of vertex indices define a face  
% - Face_reflectance_factor_solar_cell_side : # [0, 1] for ith face  
(used for SRP)  
% - Face_reflectance_factor_opposite_side : # [0, 1] for ith face  
(used for SRP)  
% - Face_orientation_solar_cell_side [unit vector] : Normal vector in  
body frame B  
% - # Face_center [m] : Center of this Face  
% - # Face_area [m^2]  
% - type [string] : Type of shape is used for MI and volume  
calculations
```

```
type % [string] : Solar panel type
```

```

        % 'body_mounted' : Stuck to SC side (only solar cell side is used for
SRP)
        % 'passive_deployed' : Passively deployed (orientation in body frame B
does not change, i.e. it is static)
        % 'active_deployed_gimballed' : Actively gimballed (orientation in
body frame B changes)

        packing_fraction % [float] # [0, 1] Packing fraction of solar cells in
solar panel

        solar_cell_efficiency % [float] # [0, 1] Efficiency of each solar cell

```

[] Properties: Variables Computed Internally

```

name % [string] 'SP i_SP'

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

instantaneous_power_generated % [Watts] : Instantaneous power produced
by ith solar panel

maximum_power % [Watts] : Maximum power that could have been produced
by ith solar panel is the Sun was exactly along SP's orientation

Sun_incidence_angle % [deg] : Angle between Sun vector and
Face_orientation_solar_cell_side

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_SC_Solar_Panel(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else

```

```

        obj.name = ['Solar Panel ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_rate_generated =
init_data.instantaneous_data_rate_generated; % [kbps]

    % Initialize Shape
    obj.shape_model = init_data.shape_model;

    Face_center = zeros(size(obj.shape_model.Faces));
    for i=1:size(obj.shape_model.Faces,1)
        % face center : [(V1x+V2x+V3x)/3 ; (V1y+V2y+V3y)/3 : (V1z+V2z
+V3z)/3 ]
        Face_center(i,:) = [
            (obj.shape_model.Vertices(obj.shape_model.Faces(i,1),1)
+ obj.shape_model.Vertices(obj.shape_model.Faces(i,2),1) +
obj.shape_model.Vertices(obj.shape_model.Faces(i,3),1))/3;
            (obj.shape_model.Vertices(obj.shape_model.Faces(i,1),2)
+ obj.shape_model.Vertices(obj.shape_model.Faces(i,2),2) +
obj.shape_model.Vertices(obj.shape_model.Faces(i,3),2))/3;
            (obj.shape_model.Vertices(obj.shape_model.Faces(i,1),3)
+ obj.shape_model.Vertices(obj.shape_model.Faces(i,2),3) +
obj.shape_model.Vertices(obj.shape_model.Faces(i,3),3))/3];
    end
    obj.shape_model.Face_center = Face_center;

    % SP area from vertices
    area = 0; % [m^2]
    for f=1:size(obj.shape_model.Faces,1)
        vertex_index = obj.shape_model.Faces(f,:); % index of
vertices for this face
        a = norm(obj.shape_model.Vertices(vertex_index(1),:) -
obj.shape_model.Vertices(vertex_index(2),:));
        b = norm(obj.shape_model.Vertices(vertex_index(2),:) -
obj.shape_model.Vertices(vertex_index(3),:));
        c = norm(obj.shape_model.Vertices(vertex_index(3),:) -
obj.shape_model.Vertices(vertex_index(1),:));
        s = (a+b+c)/2; % semi perimeter
        area = area + sqrt(s*(s-a)*(s-b)*(s-c)); % Heron formula
    end

    obj.shape_model.Face_area = area;

    % Center of mass
    obj.shape_model.r_CM = mean(obj.shape_model.Vertices, 1); % [m]

    switch obj.shape_model.type

        case 'cuboid'

```

```

        % Inertia matrix (assume cuboid)
        L = max(obj.shape_model.Vertices(:,1)) -
min(obj.shape_model.Vertices(:,1)); % [m]
        W = max(obj.shape_model.Vertices(:,2)) -
min(obj.shape_model.Vertices(:,2)); % [m]
        H = max(obj.shape_model.Vertices(:,3)) -
min(obj.shape_model.Vertices(:,3)); % [m]
        obj.shape_model.I_through_r_CM = diag([1/12*(W^2+H^2),
1/12*(L^2+H^2), 1/12*(L^2+W^2)]); % [m^2]

        % Volume
        obj.shape_model.volume = L*W*H; % [m^3]

    otherwise
        error('Havent written yet!')

end

obj.mass = init_data.mass; % [kg]

obj.packing_fraction = init_data.packing_fraction;

obj.type = init_data.type;

obj.solar_cell_efficiency = init_data.solar_cell_efficiency;

obj.instantaneous_power_generated = 0; % [W]

obj.maximum_power = 0; % [W]

obj.Sun_incidence_angle = 180; % [deg]

obj = func_update_SP_instantaneous_power_generated(obj, mission,
i_SC);

    % Initialize Variables to store: instantaneous_power_generated
maximum_power Sun_incidence_angle
    obj.store = [];

    obj.store.instantaneous_power_generated =
zeros(mission.storage.num_storage_steps,
length(obj.instantaneous_power_generated));
    obj.store.maximum_power = zeros(mission.storage.num_storage_steps,
length(obj.maximum_power));
    obj.store.Sun_incidence_angle =
zeros(mission.storage.num_storage_steps, length(obj.Sun_incidence_angle));

    obj = func_update_true_SC_SP_store(obj, mission);

    % Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

```

```
func_initialize_list_HW_energy_generated(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end
```

[] Methods: Store

Update the store variable

```
function obj = func_update_true_SC_SP_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1

obj.store.instantaneous_power_generated(mission.storage.k_storage,:) =
obj.instantaneous_power_generated; % [W]
        obj.store.maximum_power(mission.storage.k_storage,:) =
obj.maximum_power; % [W]
        obj.store.Sun_incidence_angle(mission.storage.k_storage,:) =
obj.Sun_incidence_angle; % [deg]
    end

end
```

[] Methods: Main

Update SP's instantaneous_power_generated

```
function obj = func_main_true_SC_solar_panel(obj, mission, i_SC)

    obj = func_update_SP_instantaneous_power_generated(obj, mission,
i_SC);

    obj = func_update_true_SC_SP_store(obj, mission);

    % Update Power Generated Consumed

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

func_update_instantaneous_power_generated(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);
```

```
end
```

[] Methods: Instantaneous Power Generated

Update SP's instantaneous_power_generated

```
function obj = func_update_SP_instantaneous_power_generated(obj,
mission, i_SC)

    if (obj.health == 1) &&
(mission.true_SC{i_SC}.true_SC_navigation.flag_visible_Sun == 1)

        Sun_vector =
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.position
- mission.true_SC{i_SC}.true_SC_navigation.position; % [km]

        Sun_vector_normalized = func_normalize_vec(Sun_vector); %
[unit vector]

        obj.Sun_incidence_angle =
real(acosd(dot(Sun_vector_normalized',
mission.true_SC{i_SC}.true_SC_adc.rotation_matrix *
obj.shape_model.Face_orientation_solar_cell_side'))); % [deg]

        obj.maximum_power =
mission.true_solar_system.solar_constant_AU *
(mission.true_solar_system.AU_distance/norm(Sun_vector))^2
* obj.shape_model.Face_area * obj.packing_fraction *
obj.solar_cell_efficiency; % [W]

        if obj.Sun_incidence_angle <= 90 % [deg]

            obj.instantaneous_power_generated = obj.maximum_power *
cosd(obj.Sun_incidence_angle); % [W]

        else

            % No power generated
            obj.instantaneous_power_generated = 0; % [W]

        end

    else

        % Unhealthy Solar Panel

        obj.maximum_power = 0; % [W]

        obj.instantaneous_power_generated = 0; % [W]

        obj.Sun_incidence_angle = inf; % [deg]

    end

end

end
```

```
end  
end
```

Published with MATLAB® R2022a

4.17 True_SC_Star_Tracker

Table of Contents

Class: True_SC_Star_Tracker	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3
[] Methods: Truth	5
[] Methods: Simple	5

Class: True_SC_Star_Tracker

Tracks the Star Tracker measurements

```
classdef True_SC_Star_Tracker < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

mode_true_SC_star_tracker_selector % [string]
% - Truth
% - Simple
% - Simple with Sun outside FOV

measurement_noise % [rad] (1-sigma standard deviation) (Optional)

measurement_wait_time % [sec]

location % [m] : Location of sensor, in body frame B

orientation % [unit vector] : Normal vector from location

field_of_view % [deg] : Field of view (FOV) of the camera in deg (No
measurement if Sun is within this FOV)
```

[] Properties: Variables Computed Internally

```
name % [string] 'Sun Sensor i'
```

```

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

measurement_vector % [quaternion]

measurement_time % [sec] SC time when this measurement was taken

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_SC_Star_Tracker(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Star Tracker ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_generated_per_sample =
init_data.instantaneous_data_generated_per_sample; % [kb]

    obj.mode_true_SC_star_tracker_selector =
init_data.mode_true_SC_star_tracker_selector; % [string]
    obj.measurement_wait_time = init_data.measurement_wait_time; %
[sec]
    obj.measurement_noise = init_data.measurement_noise; % [rad]

```

```

obj.measurement_vector = zeros(1,4);

obj.flag_executive = 1;

obj.measurement_time = -inf; % [sec]

obj.location = init_data.location; % [m]
obj.orientation = init_data.orientation; % [unit vector]
obj.field_of_view = init_data.field_of_view; % [deg]

if isfield(init_data, 'data')
    obj.data = init_data.data;
else
    obj.data = [];
end

% Initialize Variables to store: measurement_vector
obj.store = [];
obj.store.measurement_vector =
zeros(mission.storage.num_storage_steps, length(obj.measurement_vector));

% Update Storage
obj = func_update_true_SC_star_tracker_store(obj, mission);

% Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

% Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_star_tracker_store(obj, mission)

    if mission.storage.flag_store_this_time_step_attitude == 1

obj.store.measurement_vector(mission.storage.k_storage_attitude,:) =
obj.measurement_vector; % [quaternion]
    end

end

```

[] Methods: Main

Update Camera

```

function obj = func_main_true_SC_star_tracker(obj, mission, i_SC)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % Take measurement

        if (mission.true_time.time_attitude - obj.measurement_time) >=
obj.measurement_wait_time

            % Sufficient time has elapsed for a new measurement
            obj.measurement_time = mission.true_time.time_attitude; %
[sec]

            switch obj.mode_true_SC_star_tracker_selector

                case 'Truth'
                    obj = func_true_SC_star_tracker_Truth(obj,
mission, i_SC);

                case 'Simple'
                    obj = func_true_SC_star_tracker_Simple(obj,
mission, i_SC);

                case 'Simple with Sun outside FOV'

                    this_orientation =
(mission.true_SC{i_SC}.true_SC_adc.rotation_matrix * obj.orientation)'; %
[unit vector]

                    Sun_vector =
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.position
- mission.true_SC{i_SC}.true_SC_navigation.position; % [km]
                    Sun_vector_normalized =
func_normalize_vec(Sun_vector); % [unit vector]

                    if func_angle_between_vectors(this_orientation,
Sun_vector_normalized) >= deg2rad(obj.field_of_view)
                        obj = func_true_SC_star_tracker_Simple(obj,
mission, i_SC);
                    else
                        % Measurement doesn't exist
                        obj.measurement_vector = nan(1,4);
                    end

                    otherwise
                        error('Star Tracker mode not defined!')
                    end

                % Update Data Generated

            func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

            else
                % Data not generated in this time step

```

```

        end

        % Update Power Consumed

func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    else
        % Do nothing

    end

    % Update Storage
    obj = func_update_true_SC_star_tracker_store(obj, mission);

    % Reset Variables
    obj.flag_executive = 0;

end

```

[] Methods: Truth

Star Tracker mode

```

function obj = func_true_SC_star_tracker_Truth(obj, mission, i_SC)
    obj.measurement_vector =
func_quaternion_properize( mission.true_SC{i_SC}.true_SC_adc.attitude);
end

```

[] Methods: Simple

Star Tracker mode

```

function obj = func_true_SC_star_tracker_Simple(obj, mission, i_SC)
    % OLD Incorrect Method
    % obj.measurement_vector =
func_quaternion_properize( mission.true_SC{i_SC}.true_SC_adc.attitude +
obj.measurement_noise*randn(1,4) );

    % New Correct Method

    % Define the error quaternion (small rotation) in axis-angle
format
    error_angle = obj.measurement_noise; % [rad]
    error_axis = randn(1,3);
    error_axis = error_axis/norm(error_axis); % [unit vector]

    % Convert axis-angle to quaternion
    error_quaternion = [sin(error_angle/2) * error_axis,
cos(error_angle/2)];
    error_quaternion = func_quaternion_properize(error_quaternion);

```

```
        % Apply the error by quaternion multiplication (Hamilton product)
        obj.measurement_vector =
func_quaternion_multiply(mission.true_SC{i_SC}.true_SC_adc.attitude,
error_quaternion);
        obj.measurement_vector =
func_quaternion_properize(obj.measurement_vector);

    end

end

end
```

Published with MATLAB® R2022a

4.18 True_SC_Sun_Sensor

Table of Contents

Class: True_SC_Sun_Sensor	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3
[] Methods: Truth	5
[] Methods: Simple	5

Class: True_SC_Sun_Sensor

Tracks the Sun Sensor measurements

```
classdef True_SC_Sun_Sensor < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_power_consumed % [Watts] : Instantaneous power consumed

instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

mode_true_SC_sun_sensor_selector % [string]
% - Truth
% - Simple
% - Simple with Sun in FOV

measurement_noise % [rad] (1-sigma standard deviation)

measurement_wait_time % [sec]

location % [m] : Location of sensor, in body frame B

orientation % [unit vector] : Normal vector from location

field_of_view % [deg] : Field of view (FOV) of the camera in deg (No
measurement if Sun is outside this FOV)
```

[] Properties: Variables Computed Internally

```
name % [string] 'Sun Sensor i'
```

```

health % [integer] Health of sensor/actuator
% - 0. Switched off
% - 1. Switched on, works nominally

temperature % [deg C] : Temperature of sensor/actuator

measurement_vector % [quaternion]

measurement_time % [sec] SC time when this measurement was taken

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = True_SC_Sun_Sensor(init_data, mission, i_SC, i_HW)

    if isfield(init_data, 'name')
        obj.name = init_data.name;
    else
        obj.name = ['Sun Sensor ', num2str(i_HW)];
    end

    obj.health = 1;
    obj.temperature = 10; % [deg C]

    obj.instantaneous_power_consumed =
init_data.instantaneous_power_consumed; % [W]
    obj.instantaneous_data_generated_per_sample =
init_data.instantaneous_data_generated_per_sample; % [kb]

    obj.mode_true_SC_sun_sensor_selector =
init_data.mode_true_SC_sun_sensor_selector; % [string]
    obj.measurement_wait_time = init_data.measurement_wait_time; %
[sec]
    obj.measurement_noise = init_data.measurement_noise; % [rad]

```

```

obj.measurement_vector = zeros(1,4);

obj.flag_executive = 1;

obj.measurement_time = -inf; % [sec]

obj.location = init_data.location; % [m]
obj.orientation = init_data.orientation; % [unit vector]
obj.field_of_view = init_data.field_of_view; % [deg]

if isfield(init_data, 'data')
    obj.data = init_data.data;
else
    obj.data = [];
end

% Initialize Variables to store: measurement_vector
obj.store = [];
obj.store.measurement_vector =
zeros(mission.storage.num_storage_steps, length(obj.measurement_vector));

% Update Storage
obj = func_update_true_SC_sun_sensor_store(obj, mission);

% Update SC Power Class

func_initialize_list_HW_energy_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

% Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_true_SC_sun_sensor_store(obj, mission)

    if mission.storage.flag_store_this_time_step_attitude == 1

obj.store.measurement_vector(mission.storage.k_storage_attitude,:) =
obj.measurement_vector; % [quaternion]
        end

    end

```

[] Methods: Main

Update Camera

```

function obj = func_main_true_SC_sun_sensor(obj, mission, i_SC)

    if (obj.flag_executive == 1) && (obj.health == 1)
        % Take measurement

        if (mission.true_time.time_attitude - obj.measurement_time) >=
obj.measurement_wait_time

            % Sufficient time has elapsed for a new measurement
            obj.measurement_time = mission.true_time.time_attitude; %
[sec]

            switch obj.mode_true_SC_sun_sensor_selector

                case 'Truth'
                    obj = func_true_SC_sun_sensor_Truth(obj, mission,
i_SC);

                case 'Simple'
                    obj = func_true_SC_sun_sensor_Simple(obj, mission,
i_SC);

                case 'Simple with Sun in FOV'

                    this_orientation =
(mission.true_SC{i_SC}.true_SC_adc.rotation_matrix * obj.orientation)'; %
[unit vector]

                    Sun_vector =
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.position
- mission.true_SC{i_SC}.true_SC_navigation.position; % [km]
                    Sun_vector_normalized =
func_normalize_vec(Sun_vector); % [unit vector]

                    if func_angle_between_vectors(this_orientation,
Sun_vector_normalized) <= deg2rad(obj.field_of_view)
                        obj = func_true_SC_sun_sensor_Simple(obj,
mission, i_SC);
                    else
                        % Measurement doesn't exist
                        obj.measurement_vector = nan(1,4);
                    end

                    otherwise
                        error('Sun Sensor mode not defined!')
                    end

                    % Update Data Generated

                    func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

                else
                    % Data not generated in this time step

```

```

        end

        % Update Power Consumed
func_update_instantaneous_power_consumed(mission.true_SC{i_SC}.true_SC_power,
obj, mission);

    else
        % Do nothing

    end

    % Update Storage
obj = func_update_true_SC_sun_sensor_store(obj, mission);

    % Reset Variables
obj.flag_executive = 0;

end

```

[] Methods: Truth

Sun Sensor mode

```

function obj = func_true_SC_sun_sensor_Truth(obj, mission, i_SC)
    obj.measurement_vector =
func_quaternion_properize( mission.true_SC{i_SC}.true_SC_adc.attitude);
end

```

[] Methods: Simple

Sun Sensor mode

```

function obj = func_true_SC_sun_sensor_Simple(obj, mission, i_SC)
    % OLD Incorrect Method
    % obj.measurement_vector =
func_quaternion_properize( mission.true_SC{i_SC}.true_SC_adc.attitude +
obj.measurement_noise*randn(1,4) );

    % New Correct Method

    % Define the error quaternion (small rotation) in axis-angle
format
error_angle = obj.measurement_noise; % [rad]
error_axis = rand(1,3) - 0.5;
error_axis = error_axis/norm(error_axis); % [unit vector]

    % Convert axis-angle to quaternion
error_quaternion = [sin(error_angle/2) * error_axis,
cos(error_angle/2)];
error_quaternion = func_quaternion_properize(error_quaternion);

```

```
        % Apply the error by quaternion multiplication (Hamilton product)
        obj.measurement_vector =
func_quaternion_multiply(mission.true_SC{i_SC}.true_SC_adc.attitude,
error_quaternion);
        obj.measurement_vector =
func_quaternion_properize(obj.measurement_vector);

    end

end

end
```

Published with MATLAB® R2022a

Chapter 5

SC System-Level and Functional-Level Autonomy Software Layer Classes

5.1 Software_SC_Communication

Table of Contents

Class: Software_SC_Communication	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3

Class: Software_SC_Communication

Tracks the Communication Links of the Spacecraft

```
classdef Software_SC_Communication < handle
```

Properties

properties

[] Properties: Initialized Variables

```
    instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

    mode_software_SC_communication_selector % [string] Different
communication modes
    % - 'DART'
    % - 'Nightingale'

    attitude_error_threshold % [rad]
```

[] Properties: Variables Computed Internally

```
    name % [string] = SC j for jth SC + SW Communication

    flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

    this_attitude_error % [rad] current attitude error

    data % Other useful data

    last_communication_time % [sec]
    wait_time_comm_dte % [sec] time to wait for communicating info to
earth
```

[] Properties: Storage Variables

```
store  
  
end
```

Methods

```
methods
```

[] Methods: Constructor

Construct an instance of this class

```
function obj = Software_SC_Communication(init_data, mission, i_SC)  
  
    obj.name = [mission.true_SC{i_SC}.true_SC_body.name, ' SW  
Communication']; % [string]  
    obj.flag_executive = 0;  
  
    obj.instantaneous_data_generated_per_sample =  
init_data.instantaneous_data_generated_per_sample; % [kb]  
    obj.mode_software_SC_communication_selector =  
init_data.mode_software_SC_communication_selector; % [string]  
  
    obj.attitude_error_threshold =  
deg2rad(init_data.attitude_error_threshold_deg); % [rad]  
  
    obj.this_attitude_error = inf;  
  
    if isfield(init_data, 'data')  
        obj.data = init_data.data;  
    else  
        obj.data = [];  
    end  
  
    % Initialize communication time tracker  
    if isfield(init_data, 'last_communication_time')  
        obj.last_communication_time =  
init_data.last_communication_time;  
    else  
        obj.last_communication_time = 0; % Start at 0 so first check  
will attempt communication  
    end  
  
    % Initialize wait time for DTE communications if provided  
    if isfield(init_data, 'wait_time_comm_dte')  
        obj.wait_time_comm_dte = init_data.wait_time_comm_dte;  
    else  
        obj.wait_time_comm_dte = 3600; % Default to 1 hour (3600  
seconds)
```

```

end

% Initialize Variables to store
obj.store = [];

obj.data.transmission_complete = false;

obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
obj.store.this_attitude_error =
zeros(mission.storage.num_storage_steps, length(obj.this_attitude_error));

% Update Storage
obj = func_update_software_SC_communication_store(obj, mission);

% Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variables

```

function obj = func_update_software_SC_communication_store(obj,
mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage,:) =
obj.flag_executive; % [Boolean]
        obj.store.this_attitude_error(mission.storage.k_storage,:) =
obj.this_attitude_error; % [rad]
    end

end

```

[] Methods: Main

Main Function

```

function obj = func_main_software_SC_communication(obj, mission, i_SC)

    if (obj.flag_executive == 1)

        switch obj.mode_software_SC_communication_selector

            case 'DART'
                obj = func_update_software_SC_communication_Dart(obj,
mission, i_SC);

```

```

        case 'Nightingale'
            obj =
func_update_software_SC_communication_Nightingale(obj, mission, i_SC);

        otherwise
            disp('Communication mode not defined!')
        end

        % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    end

    % Update Storage
    obj = func_update_software_SC_communication_store(obj, mission);

    % Reset Variables
    obj.flag_executive = 0;
    obj.this_attitude_error = inf;

end

function obj = func_update_software_SC_communication_Dart(obj,
mission, i_SC)

    if
strcmp(mission.true_SC{i_SC}.software_SC_executive.this_sc_mode, 'DTE Comm')

        % When transmission is complete (as signaled by communication
module), reset memory
        if obj.data.transmission_complete
            % Empty memory after transmission is complete

mission.true_SC{i_SC}.software_SC_data_handling.mean_state_of_data_storage =
0;

mission.true_SC{i_SC}.software_SC_data_handling.total_data_storage = 0;

            % Log successful data transmission
            if ~isfield(obj.data, 'successful_transmissions')
                obj.data.successful_transmissions = 1;
            else
                obj.data.successful_transmissions =
obj.data.successful_transmissions + 1;
            end

            % Update the timestamp after successful transmission
completion

mission.true_SC{i_SC}.software_SC_communication.last_communication_time =
mission.true_time.time;

```

```

        disp(['Executive: Memory cleared after successful
transmission at time ', ...
            num2str(mission.true_time.time), ' seconds']);

        % IMPORTANT: Reset the transmission_complete flag so we
can start a new communication
        % cycle when conditions are met again
        obj.data.transmission_complete = false;

        return;
    end

    % Block other operations during communication
    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_camera
        mission.true_SC{i_SC}.true_SC_camera{i_HW}.flag_executive
= 0;
    end

    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_science_radar
        if
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_science_radar >=
i_HW
mission.true_SC{i_SC}.true_SC_science_radar{i_HW}.flag_executive = 0;
        end
    end

    % Reset communication link flags to avoid "TX link already on"
errors
    % This should be done regardless of previous state

mission.true_SC{i_SC}.true_SC_communication_link{1}.flag_executive = 0;
mission.true_SC{i_SC}.true_SC_communication_link{2}.flag_executive = 0;

    % Let the communication link manage these instead of setting
them directly
    mission.true_SC{i_SC}.true_SC_radio_antenna{1}.flag_executive
= 0;

    % Switch SC communication from Rx to Tx, and the contrary for
GS

mission.true_SC{i_SC}.true_SC_radio_antenna{1}.mode_true_SC_radio_antenna_selector
= "TX"; % Switching SC Antenna to TX

mission.true_GS_radio_antenna{1}.mode_true_GS_radio_antenna_selector
= "RX"; % Switching GS Antenna to RX

    % Check Attitude
    this_orientation =
(mission.true_SC{i_SC}.true_SC_adc.rotation_matrix *

```

```

mission.true_SC{i_SC}.true_SC_radio_antenna{1}.orientation'); % [unit
vector]
        vec_Earth_from_SC_normalized =
func_normalize_vec(mission.true_solar_system.SS_body{mission.true_solar_system.index_Earth
- mission.true_SC{i_SC}.software_SC_estimate_orbit.position}); % [unit vector]
        obj.this_attitude_error =
func_angle_between_vectors(this_orientation',
vec_Earth_from_SC_normalized'); % [rad]

        if obj.this_attitude_error <= obj.attitude_error_threshold
            % Attitude is good enough for communication

            % Switch on Communication Link for Dart (SC to Earth)

mission.true_SC{i_SC}.true_SC_communication_link{1}.flag_executive = 1;

            % Calculate data rate and transmission time
            total_data =
mission.true_SC{i_SC}.software_SC_data_handling.total_data_storage;

            % Get data rate directly - no need to use isfield since we
know it's a property
            data_rate_kbps =
mission.true_SC{i_SC}.true_SC_communication_link{1}.this_data_rate;

            % Add safeguard for division by zero
            if data_rate_kbps > 0 && total_data > 0
                transmission_time = total_data / data_rate_kbps; %
Time in seconds

                % Store communication start time and expected duration
if not already tracking
                    if ~isfield(obj.data, 'current_transmission') ||
isempty(obj.data.current_transmission)
                        obj.data.current_transmission =
struct('start_time', mission.true_time.time, ...

'estimated_duration', transmission_time, ...

'current_transmission', obj.data.current_transmission, ...
'data_size',
total_data, ...
'data_rate',
data_rate_kbps);
                        disp(['Started transmission of ',
num2str(total_data), ' kb at ', ...
num2str(data_rate_kbps), ' kbps (estimated
time: ', ...
num2str(transmission_time), ' seconds)']);
                    end

                    % Check memory usage percentage against threshold
memory_percentage =
mission.true_SC{i_SC}.software_SC_data_handling.mean_state_of_data_storage;
memory_threshold = 0.1; % 0.1% threshold

```

```

        % Check if memory is nearly empty (below 1%)
        if memory_percentage <= memory_threshold
            % Transmission complete - memory is sufficiently
emptied

            % Log successful communication attempt
            if ~isfield(obj.data, 'successful_communications')
                obj.data.successful_communications = 1;
            else
                obj.data.successful_communications =
obj.data.successful_communications + 1;
            end

            % Reset transmission tracking
            obj.data.current_transmission = [];

            % Signal completion to the executive to reset
memory
            obj.data.transmission_complete = true;

            disp(['Completed transmission at time ',
num2str(mission.true_time.time), ...
                ' seconds. Memory usage now at ',
num2str(memory_percentage), '%']);
        else
            % Still transmitting - memory not sufficiently
emptied yet
            obj.data.transmission_complete = false;

            % Optional: print status update occasionally
            if mod(mission.true_time.time, 60) <
mission.true_time.time_step
                disp(['Transmission in progress. Memory usage:
', num2str(memory_percentage), ...
                    '% (target: below ',
num2str(memory_threshold), '%)']);
            end
        end
    else
        % Can't calculate transmission time, set flag to false
        obj.data.transmission_complete = false;

        if total_data <= 0
            % Memory is already empty, consider transmission
complete
            obj.data.transmission_complete = true;
            obj.last_communication_time =
mission.true_time.time;
            disp('No data to transmit, considering
transmission complete. ');
        end
    end
end
else

```

```

                                % Attitude error is too large, log failed communication
attempt
                                if ~isfield(obj.data, 'failed_communications')
                                    obj.data.failed_communications = 1;
                                else
                                    obj.data.failed_communications =
obj.data.failed_communications + 1;
                                end

                                % Issue warning about poor pointing
                                warning('Communication attempt failed due to poor
pointing. Attitude error: %.2f degrees, threshold: %.2f degrees', ...
                                    rad2deg(obj.this_attitude_error),
rad2deg(obj.attitude_error_threshold));

                                % Ensure we don't have a completed flag set
                                obj.data.transmission_complete = false;
                            end
                        else
                            % Not in communication mode, switch SC communication to Rx,
and GS to Tx

                            % Reset communication link flags

mission.true_SC{i_SC}.true_SC_communication_link{1}.flag_executive = 0;
mission.true_SC{i_SC}.true_SC_communication_link{2}.flag_executive = 0;

                            % Reset radio antenna flags - let the communication link
handle these
                            mission.true_SC{i_SC}.true_SC_radio_antenna{1}.flag_executive
= 0;

mission.true_SC{i_SC}.true_SC_radio_antenna{1}.mode_true_SC_radio_antenna_selector
= "RX"; % Switching SC Antenna to RX

mission.true_GS_radio_antenna{1}.mode_true_GS_radio_antenna_selector
= "TX"; % Switching GS Antenna to TX

                            % Switch on Communication Link for GS to SC (for receiving
commands)

mission.true_SC{i_SC}.true_SC_communication_link{2}.flag_executive = 1;

                            % Clear any ongoing transmission data
                            if isfield(obj.data, 'current_transmission')
                                obj.data.current_transmission = [];
                            end

                            % Make sure transmission_complete is reset when not in
communication mode
                            obj.data.transmission_complete = false;
                        end
                    end

```

```
end  
end  
end
```

Published with MATLAB® R2022a

5.2 Software_SC_Control_Attitude

Table of Contents

Class: Software_SC_Control_Attitude	1
Properties	2
[] Properties: Initialized Variables	2
[] Properties: Variables Computed Internally	2
[] Properties: Storage Variables	2
Methods	3
[] Methods: Constructor	3
Initialization of thruster and RW optimization data	3
[] Methods: Store	4
Initialize Thruster Contribution Matrix	5
Initialize Reaction Wheel Contribution Matrix for Pinverse Optimization	6
[] Methods: Main	6
[] Methods: Control Attitude DART Oracle	7
[] Methods: Control Attitude DART Oracle	9
[] Methods: Get Attitude Error	10
[] Methods: Predict wheel saturation	12
[] Methods: Reset Micro Thrusters	14
[] Methods: Reset Reaction Wheels	14
[] Methods: Compute Reaction Wheels Command using pseudo inverse	15
[] Methods: Manage Wheel Momentum to Avoid Direction Reversals	15
[] Methods: Safely Apply Reaction Wheels Command	18
[] Methods: Handle wheel desaturation	18
[] Methods: Checks if desaturation is needed	20
[] Methods: Optimize Thrusters using Pseudo Inverse	20
Step 1: Identify microthrusters with the same resultant torques	21
Step 2: Identify microthrusters with opposite resultant torques	23
Step 3: Formulate the optimization problem	24
Step 4: Store optimization data in obj	24
[] Methods: Optimize Thrusters using KKT Condition	25
Step 1: Reset thruster commands	25
Step 2: Retrieve pre-computed optimization data	25
Step 3: Solve the optimization problem using the KKT system	26
Step 4: Undo the torque pair reduction	26
Step 5: Assign computed thrust and torque to each microthruster	27
[] Methods: Control Attitude Oracle	29
[] Methods: Desired Control Torque Asymptotically Stable	29
[] Methods: Desired Control Torque PD	30
[] Methods: Desired Attitude CVX	31
[] Methods: Desired Attitude Array Search	31
[] Methods: Update torque capabilities	32
[] Methods: Calculate maximum reaction wheel torque capability	32
[] Methods: Calculate maximum thruster torque capability	33
Helper function: Calculate angle between quaternions	34

Class: Software_SC_Control_Attitude

Control the Attitude of the Spacecraft

```
classdef Software_SC_Control_Attitude < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

mode_software_SC_control_attitude_selector % [string] Different
attitude control modes
% - 'DART Oracle' : Directly change True_SC_ADC.attitude values for
DART mission
% - 'Nightingale Oracle' : Directly change True_SC_ADC.attitude values
for Nightingale mission
% - 'Oracle with Control' : Use True_SC_ADC values + Noise
```

[] Properties: Variables Computed Internally

```
name % [string] = SC j for jth SC + SW Control Attitude

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

desired_attitude % [quaternion] : Orientation of inertial frame I
with respect to the body frame B

desired_angular_velocity % [rad/sec] : Angular velocity of inertial
frame I with respect to the body frame B

desired_control_torque % [Nm] : Desired control torque

data % Other useful data

desaturation_procedure % [Bool] - Is the procedure started ?
thruster_contribution_matix % thruster contribution
pinv_reaction_wheel_contribution_matrix % reaction wheel contribution
matrix
reaction_wheel_attitude_control_threshold % [rad] Angle from which
wheels can take over the correction
optim_data

max_thrust % [N] : Maximum thrust of the thruster
min_thrust % [N] : Minimum thrust of the thruster

% Cached torque capabilities
max_rw_torque % [Nm] : Maximum torque capability of reaction wheels
max_mt_torque % [Nm] : Maximum torque capability of micro thrusters
```

[] Properties: Storage Variables

store

end

Methods

methods

[] Methods: Constructor

Construct an instance of this class

```
function obj = Software_SC_Control_Attitude(init_data, mission, i_SC)

    obj.name = [mission.true_SC{i_SC}.true_SC_body.name, ' SW Control
Attitude']; % [string]
    obj.flag_executive = 1;

    obj.instantaneous_data_generated_per_sample = 0; % [kb]
    obj.mode_software_SC_control_attitude_selector =
init_data.mode_software_SC_control_attitude_selector; % [string]

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);
```

Initialization of thruster and RW optimization data

```
func_initialize_optimization_data(obj, mission, i_SC);

% Initialize matrices and hardware data
obj = obj.initialize_thruster_contribution(mission, i_SC);
obj = obj.initialize_reaction_wheel_contribution(mission, i_SC);

% Calculate and cache max torque capabilities
obj.max_rw_torque =
obj.calculate_max_reaction_wheel_torque(mission, i_SC);
obj.max_mt_torque = obj.calculate_max_thruster_torque(mission,
i_SC);

% All the zeros
obj.desaturation_procedure = 0;
obj.desired_attitude = zeros(1,4); % [quaternion]
```

```

        obj.desired_control_torque = zeros(1,3); % [Toraue Vector Nm]
        obj.desired_angular_velocity = zeros(1,3); % [rad/sec]
        obj.data.integral_error = zeros(3,1); % Initialize as 3x1 vector
    for X/Y/Z axes

        % Initialize control gains
        obj.data.control_gain = init_data.control_gain;

        % Initialize Variables to store
        obj.store = [];

        obj.store.flag_executive =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.flag_executive));
        obj.store.desaturation_procedure =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.desaturation_procedure));
        obj.store.desired_attitude =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.desired_attitude));
        obj.store.desired_angular_velocity =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.desired_angular_velocity));
        obj.store.desired_control_torque =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.desired_control_torque));

        % Update Storage
        obj = func_update_software_SC_control_attitude_store(obj,
mission);

        if isfield(init_data, 'reaction_wheel_attitude_control_threshold')
            obj.reaction_wheel_attitude_control_threshold =
init_data.reaction_wheel_attitude_control_threshold;
        else
            obj.reaction_wheel_attitude_control_threshold = 0.1; % [Rad]-
Nominal is 0.05
        end

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

    end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_software_SC_control_attitude_store(obj,
mission)
    if mission.storage.flag_store_this_time_step_attitude == 1

```

```

        obj.store.flag_executive(mission.storage.k_storage_attitude,:)
= obj.flag_executive;

obj.store.desaturation_procedure(mission.storage.k_storage_attitude,:) =
obj.desaturation_procedure; % [quaternion]

obj.store.desired_attitude(mission.storage.k_storage_attitude,:) =
obj.desired_attitude; % [quaternion]

obj.store.desired_angular_velocity(mission.storage.k_storage_attitude,:) =
obj.desired_angular_velocity; % [rad/sec]

obj.store.desired_control_torque(mission.storage.k_storage_attitude,:) =
obj.desired_control_torque; % [Nm]
    end
end

function obj = initialize_from_init_data(obj, mission, i_SC,
init_data)

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    if isfield(init_data, 'control_gain')
        obj.data.control_gain = init_data.control_gain;
    else
        obj.data.control_gain = [0.1 1]; % [Kr ; Lambda_r] for RWA/
MT control
    end
end

```

Initialize Thruster Contribution Matrix

```

function obj = initialize_thruster_contribution(obj, mission, i_SC)
    % Build thruster torque contribution matrix
    num_thrusters =
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster;
    obj_thruster_contribution_matix = zeros(3, num_thrusters);
    obj.max_thrust = zeros(1, num_thrusters);
    obj.min_thrust = zeros(1, num_thrusters);

    for i = 1:num_thrusters
        thruster = mission.true_SC{i_SC}.true_SC_micro_thruster{i};
        r = thruster.location -
mission.true_SC{i_SC}.true_SC_body.location_COM;
        obj_thruster_contribution_matix(:,i) = cross(r,
thruster.orientation);
        obj.max_thrust(i) = thruster.maximum_thrust;
        obj.min_thrust(i) = thruster.minimum_thrust;
    end
end

```

```
end
end
```

Initialize Reaction Wheel Contribution Matrix for Pinverse Optimization

```
function obj = initialize_reaction_wheel_contribution(obj, mission,
i_SC)
    % Build RWA momentum contribution matrix
    reaction_wheel_contribution_matrix = zeros(3,
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel);

    for i = 1:
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
        wheel = mission.true_SC{i_SC}.true_SC_reaction_wheel{i};
        if wheel.health
            reaction_wheel_contribution_matrix(:,i) =
wheel.orientation' * wheel.moment_of_inertia;
        end
    end

    obj.pinv_reaction_wheel_contribution_matrix =
pinv(reaction_wheel_contribution_matrix);
end
```

[] Methods: Main

Main Function

```
function obj = func_main_software_SC_control_attitude(obj, mission,
i_SC)

    if (obj.flag_executive == 1)

        switch obj.mode_software_SC_control_attitude_selector

            case {'DART Oracle', 'DART Control Asymptotically Stable
send to ADC directly', 'DART Control PD', 'DART Control Asymptotically Stable
send to actuators', 'DART Control Asymptotically Stable send to thrusters'}
                obj =
func_update_software_SC_control_attitude_DART(obj, mission, i_SC);

            case {'Nightingale Oracle', 'Nightingale Control
Asymptotically Stable send to actuators', 'Nightingale Control Asymptotically
Stable send to rwa', 'Nightingale Control Asymptotically Stable send to
thrusters'}
                obj =
func_update_software_SC_control_attitude_Nightingale(obj, mission, i_SC);

            otherwise
                error('Attitude Control mode not defined!')
```

```

        end

        % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);
    end

    % Update Storage
    obj = func_update_software_SC_control_attitude_store(obj,
mission);

    % DO NOT SWITCH OFF FUNCTIONS USING flag_executive INSIDE Attitude
Dynamics Loop (ADL)
end

```

[] Methods: Control Attitude DART Oracle

Use Truth Data

```

function obj = set_pointing_vectors(obj, mission, i_SC)

    % Navigation and Guidance
    switch mission.true_SC{i_SC}.software_SC_executive.this_sc_mode

        case 'Point camera to Target'
            % Point camera to target
            obj.data.primary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_camera{1}.orientation); % In
body frame
            obj.data.desired_primary_vector =
func_normalize_vec(mission.true_SC{i_SC}.software_SC_estimate_orbit.position_target
- mission.true_SC{i_SC}.software_SC_estimate_orbit.position); % In Inertial
frame
            % Optimize for solar pannels orientation to sun
            obj.data.secondary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_solar_panel{1}.shape_model.Face_orientat
In body frame
            obj.data.desired_secondary_vector =
func_normalize_vec(mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}
- mission.true_SC{i_SC}.software_SC_estimate_orbit.position); % In Inertial
frame

        case 'DTE Comm'
            % Point antenna to earth
            % DART has one antenna
            obj.data.primary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_radio_antenna{1}.orientation); %
In body frame
            obj.data.desired_primary_vector = func_normalize_vec( ...

mission.true_solar_system.SS_body{mission.true_solar_system.index_Earth}.position
- ...

```

```

mission.true_SC{i_SC}.software_SC_estimate_orbit.position); % In Inertial
frame

        % Optimize for solar pannels orientation to sun
        obj.data.secondary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_solar_panel{1}.shape_model.Face_orientat
In body frame
        obj.data.desired_secondary_vector =
func_normalize_vec(mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}
- mission.true_SC{i_SC}.software_SC_estimate_orbit.position); % In Inertial
frame

        case 'Point Thruster along DeltaV direction'
            % Point thruster in direction of desired deltaV
            obj.data.primary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_chemical_thruster.orientation); %
In body frame

            % Use the deltaV vector direction instead of target
direction
            deltaV =
mission.true_SC{i_SC}.software_SC_control_orbit.desired_control_DeltaV;
            obj.data.desired_primary_vector =
func_normalize_vec(deltaV)'; % In Inertial frame

            % Secondary vector remains optimized for solar pannels
            obj.data.secondary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_solar_panel{1}.shape_model.Face_orientat
In body frame
            obj.data.desired_secondary_vector =
func_normalize_vec(mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}
- mission.true_SC{i_SC}.software_SC_estimate_orbit.position); % In Inertial
frame

        case 'Maximize SP Power'
            % Primary vector is the solar pannels to sun
            obj.data.primary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_solar_panel{1}.shape_model.Face_orientat
In body frame
            obj.data.desired_primary_vector =
func_normalize_vec(mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}
- mission.true_SC{i_SC}.software_SC_estimate_orbit.position); % In Inertial
frame

            % [Optional] Optimize for DTE to earth
            obj.data.secondary_vector =
func_normalize_vec(mission.true_SC{i_SC}.true_SC_radio_antenna{1}.orientation); %
In body frame
            obj.data.desired_secondary_vector =
func_normalize_vec(mission.true_solar_system.SS_body{mission.true_solar_system.index_Eart

```

```

- mission.true_SC{i_SC}.software_SC_estimate_orbit.position); % In Inertial
frame

        otherwise
            error('Attitude Control mode for this SC mode not
defined!')
        end

    end

end

```

[] Methods: Control Attitude DART Oracle

Use Truth Data

```

function obj = func_update_software_SC_control_attitude_DART(obj,
mission, i_SC)

    obj.instantaneous_data_generated_per_sample = (1e-3)*8*7; % [kb]
    i.e. 7 Bytes per sample

    % Navigation and Guidance
    % Set pointing vectors based on SC mode
    obj = obj.set_pointing_vectors(mission, i_SC);

    % Compute Desired Attitude using Array Search
    obj = func_compute_desired_attitude_Array_Search(obj);
    obj.desired_angular_velocity = zeros(1,3); % [rad/sec]

    % Control
    switch obj.mode_software_SC_control_attitude_selector

        case 'DART Oracle'
            % Use Oracle
            obj = func_update_software_SC_control_attitude_Oracle(obj,
mission, i_SC);

        case 'DART Control Asymptotically Stable send to ADC directly'
            obj =
function_update_desired_control_torque_asymptotically_stable(obj, mission,
i_SC);

            mission.true_SC{i_SC}.true_SC_adc.control_torque =
obj.desired_control_torque;

        case 'DART Control Asymptotically Stable send to actuators'
            obj =
function_update_desired_control_torque_asymptotically_stable(obj, mission,
i_SC);

            obj = func_actuator_selection(obj, mission, i_SC);

        case 'DART Control PD'
            obj = function_update_desired_control_torque_PD(obj,
mission, i_SC);

```

```

        obj = func_actuator_selection(obj, mission, i_SC);

        case 'DART Control Asymptotically Stable send to thrusters'
            obj =
function_update_desired_control_torque_asymptotically_stable(obj, mission,
i_SC);

func_decompose_control_torque_into_thrusters_optimization_kkt(obj,mission,
i_SC);

        otherwise
            error('DART Attitude Control mode not defined!')
        end
    end
end

```

[] Methods: Get Attitude Error

```

function error = get_attitude_error_euler(obj, mission, i_SC)
    % This method is used by the orbit control
    % Decide which actuator to use
    actual_euler =
    ConvertAttitude(mission.true_SC{i_SC}.software_SC_estimate_attitude.attitude'/
norm(mission.true_SC{i_SC}.software_SC_estimate_attitude.attitude)', 'quaternion','321');
    desired_euler = ConvertAttitude(obj.desired_attitude'/
norm(obj.desired_attitude)', 'quaternion','321');
    error = abs(actual_euler - desired_euler);
end

function error = get_attitude_error(obj, mission, i_SC)
    % This method calculates attitude error more efficiently
    % by working directly in quaternion space

    % Get quaternions and ensure proper format/normalization
    q_actual =
mission.true_SC{i_SC}.software_SC_estimate_attitude.attitude;
    q_desired = obj.desired_attitude;

    % Ensure column vectors and normalize once
    if size(q_actual,1) == 1
        q_actual = q_actual';
    end
    if size(q_desired,1) == 1
        q_desired = q_desired';
    end

    q_actual = q_actual / norm(q_actual);
    q_desired = q_desired / norm(q_desired);

    % Compute quaternion error: q_error = q_desired^(-1) * q_actual
    % For quaternion conjugate, negate the vector part (first 3
elements)
    q_desired_conj = q_desired;
    q_desired_conj(1:3) = -q_desired_conj(1:3);

```

```

        % Quaternion multiplication (using standard quaternion
multiplication formula)
        % This is q_error = q_desired_conj # q_actual
        q_error = zeros(4,1);
        q_error(1) = q_desired_conj(4)*q_actual(1) +
q_desired_conj(1)*q_actual(4) + q_desired_conj(2)*q_actual(3) -
q_desired_conj(3)*q_actual(2);
        q_error(2) = q_desired_conj(4)*q_actual(2) +
q_desired_conj(2)*q_actual(4) + q_desired_conj(3)*q_actual(1) -
q_desired_conj(1)*q_actual(3);
        q_error(3) = q_desired_conj(4)*q_actual(3) +
q_desired_conj(3)*q_actual(4) + q_desired_conj(1)*q_actual(2) -
q_desired_conj(2)*q_actual(1);
        q_error(4) = q_desired_conj(4)*q_actual(4) -
q_desired_conj(1)*q_actual(1) - q_desired_conj(2)*q_actual(2) -
q_desired_conj(3)*q_actual(3);

        % Extract error angle from quaternion (in radians)
        % The scalar part of the quaternion (q_error(4)) is cos(#/2)
        % So # = 2*acos(q_error(4)) is the rotation angle
        error_angle = 2 * acos(min(1, max(-1, q_error(4))));

        % Return error as a 3x1 vector (for compatibility with existing
code)

        % Scale by rotation axis to get component errors
        if error_angle > 1e-10 % Avoid division by zero
            axis = q_error(1:3) / sin(error_angle/2);
            error = error_angle * axis;
        else
            % For very small errors, just return the vector part (scaled)
            error = 2 * q_error(1:3);
        end
    end

function obj = func_actuator_selection(obj, mission, i_SC)

    % Get current attitude error
    attitude_error = obj.get_attitude_error(mission, i_SC);

    % STEP 0: Check if wheels need desaturation
    if obj.is_desaturation_needed(mission, i_SC)
        % If wheels are already saturated, handle desaturation
        disp('Performing wheel desaturation');
        obj.handle_desaturation(mission, i_SC);
        return;
    end

    % STEP 1: Check if attitude error is too large for wheels
    if norm(attitude_error) >
obj.reaction_wheel_attitude_control_threshold
        % For large angle corrections, always use thrusters - simple
rule

```

```

func_decompose_control_torque_into_thrusters_optimization_kkt(obj, mission,
i_SC);
    return;
end

% STEP 2: Check if torque request exceeds wheel capability
if norm(obj.desired_control_torque) > obj.max_rw_torque
    % If torque is too high for wheels, use thrusters

func_decompose_control_torque_into_thrusters_optimization_kkt(obj, mission,
i_SC);
    return;
end

% STEP 3: Check for predicted wheel saturation (simplified)
[will_saturate_quickly, ~] = obj.predict_wheel_saturation(mission,
i_SC);
if will_saturate_quickly
    % If wheels will saturate soon, use thrusters
    disp('Using thrusters due to predicted wheel saturation');

func_decompose_control_torque_into_thrusters_optimization_kkt(obj, mission,
i_SC);
    return;
end

% STEP 4: If we reached here, use reaction wheels (all conditions
satisfied)
reset_thrusters(obj, mission, i_SC);
func_compute_rw_command_pinv(obj, mission, i_SC);
end

```

[] Methods: Predict wheel saturation

```

function [will_saturate_quickly, time_steps_to_saturation] =
predict_wheel_saturation(obj, mission, i_SC)
    % Simplified prediction of reaction wheel saturation

    % Initialize output
    will_saturate_quickly = false;
    time_steps_to_saturation = Inf;

    % Minimum acceptable time steps before saturation
    min_acceptable_time_steps = 5;

    % Skip calculation if desired torque is negligible
    if norm(obj.desired_control_torque) < 1e-6
        return;
    end

    % Calculate required wheel accelerations using pseudo-inverse

```

```

        wheel_accelerations = obj.pinv_reaction_wheel_contribution_matrix
* obj.desired_control_torque';

    % Check each wheel
    for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
        wheel = mission.true_SC{i_SC}.true_SC_reaction_wheel{i};

        % Skip unhealthy wheels
        if ~wheel.health
            continue;
        end

        % Skip if the commanded acceleration is negligible
        if abs(wheel_accelerations(i)) < 1e-6
            continue;
        end

        % Limit command to maximum acceleration
        cmd_accel = wheel_accelerations(i);
        if abs(cmd_accel) > wheel.maximum_acceleration
            cmd_accel = sign(cmd_accel) * wheel.maximum_acceleration;
        end

        % Current angular velocity
        current_velocity = wheel.angular_velocity;

        % Saturation threshold (80% of max, matching the wheel's
internal check)
        saturation_threshold = wheel.max_angular_velocity * 0.8;

        % Two main checks:

        % 1. Direction reversal check - these are problematic
        if abs(current_velocity) > 0.3 * wheel.max_angular_velocity
&& ...
            sign(current_velocity) ~= sign(current_velocity +
cmd_accel * mission.true_time.time_step_attitude)
            % Large direction reversal detected
            will_saturate_quickly = true;
            time_steps_to_saturation = 1;
            return;
        end

        % 2. Acceleration toward saturation
        if sign(cmd_accel) == sign(current_velocity)
            % Wheel accelerating in same direction (toward saturation)
            remaining_velocity = saturation_threshold -
abs(current_velocity);

            if remaining_velocity <= 0
                % Already at or beyond saturation threshold
                will_saturate_quickly = true;
                time_steps_to_saturation = 0;
            end
        end
    end
end

```

```

        return;
    end

    % Time steps until saturation at current acceleration
    steps = remaining_velocity / (abs(cmd_accel) *
mission.true_time.time_step_attitude);

    if steps < min_acceptable_time_steps
        will_saturate_quickly = true;
        time_steps_to_saturation = steps;
        return;
    end

    % Keep track of minimum time to saturation across all
wheels

    if steps < time_steps_to_saturation
        time_steps_to_saturation = steps;
    end
end
end

    % Check for significant disturbance torque (simplified version)
    if
isfield(mission.true_SC{i_SC}.true_SC_adc, 'disturbance_torque')
        disturbance_torque =
mission.true_SC{i_SC}.true_SC_adc.disturbance_torque;
        desired_torque_mag = norm(obj.desired_control_torque);

        % If significant disturbance present relative to command
        if desired_torque_mag > 1e-6 && norm(disturbance_torque) > 0.5
* desired_torque_mag
            will_saturate_quickly = true;
            time_steps_to_saturation = min_acceptable_time_steps; %
Conservative estimate
        end
    end
end
end

```

[] Methods: Reset Micro Thrusters

```

function reset_thrusters(~, mission, i_SC)
    for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.command_actuation = 0;

mission.true_SC{i_SC}.true_SC_micro_thruster{i}.commanded_thrust = 0;
    end
end

```

[] Methods: Reset Reaction Wheels

```

function reset_wheels(~, mission, i_SC)

```

```

        for i =
1:mission.true_SC{i_SC}.true_SC_body.numHardware_exists.num_reaction_wheel
        mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.flag_executive
= 0;

mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.commanded_angular_acceleration
= 0;

    end
end

```

[] Methods: Compute Reaction Wheels Command using pseudo inverse

```

function func_compute_rw_command_pinv(obj, mission, i_SC)
    % Compute reaction wheel commands to achieve desired control
    torque

    % Use scaled torque for wheel commands
    wheel_accelerations = obj.pinv_reaction_wheel_contribution_matrix
* obj.desired_control_torque';

    % Apply momentum management to avoid direction reversals
    wheel_accelerations = obj.manage_wheel_momentum(mission, i_SC,
wheel_accelerations);

    % Apply commands to reaction wheels
    obj.apply_reaction_wheel_commands(mission, i_SC,
wheel_accelerations);
end

```

[] Methods: Manage Wheel Momentum to Avoid Direction Reversals

```

function wheel_accelerations = manage_wheel_momentum(obj, mission,
i_SC, wheel_accelerations)
    % Apply momentum management to avoid abrupt direction changes
    for i =
1:mission.true_SC{i_SC}.true_SC_body.numHardware_exists.num_reaction_wheel

        % Skip unhealthy wheels
        if ~mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.health
            wheel_accelerations(i) = 0;
            continue;
        end

        % Get current wheel velocity
        current_velocity =
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.angular_velocity;

        % Calculate absolute velocity as percentage of max

```

```

        velocity_percentage = abs(current_velocity) /
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.max_angular_velocity;

        % ADDED: Special handling for mode transitions
        if isfield(obj.data, 'mode_transition_detected') &&
obj.data.mode_transition_detected
            % During mode transitions, be extremely conservative with
commands

            % Calculate expected velocity after one time step
            predicted_vel = current_velocity + wheel_accelerations(i)
* mission.true_time.time_step_attitude;

            % Check for problematic direction reversal
            if sign(predicted_vel) ~=
sign(current_velocity) && abs(current_velocity) > 0.3 *
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.max_angular_velocity
                % Direction would reverse and current speed is
significant

                % Instead of allowing the direction change, apply
gradual deceleration
                % Set deceleration to a fraction of what would be
needed to stop in 5 time steps
                safe_decel = -sign(current_velocity) *
min(abs(current_velocity) / (5 * mission.true_time.time_step_attitude), 0.05
* mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.maximum_acceleration);

                % Limit the change and log
                wheel_accelerations(i) = safe_decel;

                % Log this limiting with detailed diagnostics
                disp(['MODE TRANSITION SAFETY: Limiting wheel ',
num2str(i), ' acceleration during mode change.']);
                disp(['Current velocity: ',
num2str(current_velocity), ' rad/s (', num2str(velocity_percentage*100), '%
of max)']);
                disp(['Original command: ',
num2str(wheel_accelerations(i)), ' rad/s^2']);
                disp(['Safe deceleration: ', num2str(safe_decel), '
rad/s^2']);

                continue; % Skip the rest of the checks for this wheel
            end

            % Even when not reversing direction, limit acceleration
during transitions
            max_safe_accel = 0.1 *
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.maximum_acceleration;
            if abs(wheel_accelerations(i)) > max_safe_accel
                wheel_accelerations(i) = sign(wheel_accelerations(i))
* max_safe_accel;
                disp(['Limiting wheel ', num2str(i), ' acceleration to
', num2str(max_safe_accel), ' rad/s^2 during mode transition']);

```

```

        end

        continue; % Skip further checks during mode transitions
    end

    % Check for direction reversal (wheel going at high speed in
one direction
    % and acceleration trying to go in the opposite)
    if abs(current_velocity) > 0.5 *
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.max_angular_velocity &&
sign(current_velocity) ~= sign(wheel_accelerations(i))
        % Calculate expected velocity after one time step
        predicted_vel = current_velocity + wheel_accelerations(i)
        * mission.true_time.time_step_attitude;

        % If direction would reverse, limit the acceleration to
begin deceleration
        % rather than attempting a full reversal
        if sign(predicted_vel) ~= sign(current_velocity)
            % Set acceleration to a safe deceleration value (~ 5%
of max velocity per second)
            safe_decel = -sign(current_velocity) *
min(abs(wheel_accelerations(i)), 0.05 *
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.max_angular_velocity /
mission.true_time.time_step_attitude);

            % Limit the change
            wheel_accelerations(i) = safe_decel;

            % Log this limiting
            if abs(wheel_accelerations(i)) > 0.01
                disp(['Limiting wheel ', num2str(i), '
acceleration to prevent direction reversal. Current vel: ', ...
                    num2str(current_velocity), ', cmd accel: ',
num2str(wheel_accelerations(i))]);
            end
        end
    end

    % ADDED: Additional safety for high-speed wheels
    if velocity_percentage > 0.7
        % For wheels already at high speeds, be more conservative
with acceleration
        max_safe_accel = (1 - velocity_percentage) *
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.maximum_acceleration;

        if abs(wheel_accelerations(i)) > max_safe_accel &&
sign(wheel_accelerations(i)) == sign(current_velocity)
            % Only limit acceleration in the same direction as
current velocity
            wheel_accelerations(i) = sign(wheel_accelerations(i))
        * max_safe_accel;
        end
    end
end

```

```
    end
end
```

[] Methods: Safely Apply Reaction Wheels Command

```
function apply_reaction_wheel_commands(~, mission, i_SC,
wheel_accelerations)
    % Safely apply calculated commands to the reaction wheels

    for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
        if mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.health &&
abs(wheel_accelerations(i)) > 0
            % Additional safety check before applying command
            if abs(wheel_accelerations(i)) >
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.maximum_acceleration
                % Limit command to maximum acceleration while
preserving direction
                    wheel_accelerations(i) = sign(wheel_accelerations(i))
* mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.maximum_acceleration;
            end

mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.flag_executive = 1;

mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.commanded_angular_acceleration
= wheel_accelerations(i);
        else

mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.flag_executive = 0;

mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.commanded_angular_acceleration
= 0;
        end
    end
end
```

[] Methods: Handle wheel desaturation

```
function handle_desaturation(obj, mission, i_SC)
    % Perform wheel desaturation if required

    % Define desaturation threshold (in radians per second)
    % Change from 10 RPM to 100 RPM (~10.47 rad/s)
    desaturation_threshold = 100 * 2 * pi / 60; % 100 RPM

    wheels = mission.true_SC{i_SC}.true_SC_reaction_wheel;

    % Check if desaturation is complete for all wheels
    all_wheels_safe = true;
```

```

        was_in_desaturation = obj.desaturation_procedure;

        for i =
1:mission.true_SC{i_SC}.true_SC_body.numHardwareExists.numReactionWheel
            wheel = wheels{i};
            if abs(wheel.angular_velocity) > desaturation_threshold
                all_wheels_safe = false;
                break;
            end
        end

        if all_wheels_safe
            if was_in_desaturation
                % Just exited desaturation mode - update capabilities
                obj.desaturation_procedure = false;
                obj = updateTorqueCapabilities(obj, mission, i_SC);
            end
            return;
        else
            if ~was_in_desaturation
                % Just entered desaturation mode - update capabilities
                obj.desaturation_procedure = true;
                obj = updateTorqueCapabilities(obj, mission, i_SC);
            else
                obj.desaturation_procedure = true;
            end
        end

        wheel_torque = zeros(1,3);

        % Desaturate individual wheels and compute thruster torques
        for i =
1:mission.true_SC{i_SC}.true_SC_body.numHardwareExists.numReactionWheel
            wheel = wheels{i};

            % Calculate command to reduce wheel velocity - typically 10%
per time step
            cmd = -wheel.angular_velocity * 0.1; %

            % Apply safety limit to desaturation command
            if abs(cmd) > wheel.maximum_acceleration
                % Limit the commanded acceleration to maximum safe value
                cmd = sign(cmd) * wheel.maximum_acceleration;
            end

            wheel.commanded_angular_acceleration = cmd;
            wheel.flag_executive = true;

            % Compute the resulting torque from the reaction wheel
            wheel_torque = wheel_torque + (wheel.moment_of_inertia * cmd *
wheel.orientation);
        end

        % Compute the required thruster torque to compensate

```

```

        residual_torque = obj.desired_control_torque - wheel_torque;
        % Optimize thruster operation

obj.func_decompose_control_torque_into_thrusters_optimization_kkt(mission,
i_SC, residual_torque);
end

```

[] Methods: Checks if desaturation is needed

```

function needed = is_desaturation_needed(obj, mission, i_SC)
    % Simplified function to check if wheel desaturation is needed
    % Returns true if any wheel is saturated or desaturation is in
progress

    needed = false;

    % If desaturation is already in progress, continue it
    if obj.desaturation_procedure
        needed = true;
        return;
    end

    % Check each reaction wheel for saturation
    for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel

        % Only consider healthy wheels
        if mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.health &&
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.saturated
            needed = true;
            return;
        end
    end
end
end

```

[] Methods: Optimize Thrusters using Pseudo Inverse

```

function optimize_thruster_pinv(obj, mission, i_SC, desired_torque)

    % Optimize the use of micro-thrusters for the desired control
torque

    if nargin < 4
        % Allows to pass another torque as parameter
        desired_torque = obj.desired_control_torque';
    end

    % Solve for thrusts using linear least squares
    thrusts = pinv(obj.thruster_contribution_matix) * desired_torque;

    % Apply thrust min limit

```

```

        for i=1:length(thrusts)
            if(thrusts(i) <
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.minimum_thrust)
                if (rem(i, 2) == 0)
                    % Add on the next one
                    thrusts(i-1) = thrusts(i-1)+abs(thrusts(i));
                else
                    thrusts(i+1) = thrusts(i+1)+abs(thrusts(i));
                end
                thrusts(i) = 0;
            end
        end

        % Scale all thrusts proportionally if any exceed the maximum limit
        if max(thrusts) > max(obj.max_thrust)
            scaling_factor = max(thrusts) / max(obj.max_thrust);
            thrusts = thrusts / scaling_factor;
        end

        % Assign the thrust individually
        for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster
            if thrusts(i) >
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.minimum_thrust

mission.true_SC{i_SC}.true_SC_micro_thruster{i}.command_actuation = 1;

mission.true_SC{i_SC}.true_SC_micro_thruster{i}.commanded_thrust =
thrusts(i);

            else

mission.true_SC{i_SC}.true_SC_micro_thruster{i}.command_actuation = 0;

mission.true_SC{i_SC}.true_SC_micro_thruster{i}.commanded_thrust = 0;

            end
        end
    end

    function obj = func_initialize_optimization_data(obj, mission, i_SC)

        % Assumption: The thrust of each microthruster can be compensated
by the
        % thrust of another microthruster with an equal and opposite
resultant torque.

```

Step 1: Identify microthrusters with the same resultant torques

Extract the thruster input matrix, where each row represents a microthruster's contribution. Columns correspond to force/torque contributions in each axis.

```

    % Initialize the thruster input array

```

```

        thruster_input_array =
zeros(mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster,
3);

        % Get the center of mass
        center_of_mass = mission.true_SC{1, 1}.true_SC_body.location_COM;

        % Loop through each thruster
        for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster
            % Calculate the relative position vector
            relative_position =
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.location - center_of_mass;

            % Compute the torque using the cross product
            torque = cross(relative_position,
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.orientation);

            % Round the result to three decimal places
            torque_rounded = round(torque, 3);

            % Store the result in the thruster input array
            thruster_input_array(i, :) = torque_rounded;
        end

        % Number of microthrusters
        N_mt =
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster;

        % Flags to track checked microthrusters
        flag_mt_checked = zeros(N_mt, 1);

        % Preallocate for storing indices and matrix of same torque
directions
        idxs_bool_same_torque = []; % Logical array for MTs with the same
torque
        matrix_same_torque = []; % Reduced input matrix for MTs with
same torque

        % Iterate through each microthruster
        for i_mt = 1:N_mt
            if flag_mt_checked(i_mt)
                continue; % Skip already processed microthrusters
            end

            % Extract the torque direction vector for the current
microthruster
            torque_dir_i = thruster_input_array(i_mt, :);

            % Identify microthrusters with the same torque direction
            idxs_bool = ismember(thruster_input_array,
torque_dir_i, 'rows');

```

```

        % Append the identified torque direction and indices to the
results      matrix_same_torque = [matrix_same_torque; torque_dir_i];
              idxs_bool_same_torque = [idxs_bool_same_torque; idxs_bool'];

              % Mark these microthrusters as processed
              flag_mt_checked(idxs_bool) = 1;
end

              % Convert the logical indices into a boolean array
              idxs_bool_same_torque = boolean(idxs_bool_same_torque);

```

Step 2: Identify microthrusters with opposite resultant torques

Extract the size of the reduced matrix containing same torques

```

N_mt_same = size(matrix_same_torque, 1);

% Reset flags to track processed microthrusters
flag_mt_checked = zeros(N_mt, 1);

% Preallocate for storing indices and matrix of opposite torque
directions
idxs_num_opposite_torque = []; % Numerical indices for MT pairs
with opposite torque
matrix_opposite_torque = []; % Reduced input matrix for opposite
torques

% Iterate through the reduced set of same torque microthrusters
for i_mt = 1:N_mt_same
    if flag_mt_checked(i_mt)
        continue; % Skip already processed microthrusters
    end

    % Extract the torque direction vector for the current
microthruster
    torque_dir_i = matrix_same_torque(i_mt, :);

    % Identify microthrusters with opposite torque direction
    idxs_bool = ismember(matrix_same_torque, -
torque_dir_i, 'rows');

    % Ensure there is exactly one matching opposite torque
    assert(sum(idxs_bool) == 1, 'There must be a unique opposite
torque match.');
```

```

    % Find the index of the matching opposite torque
    j_mt = find(idxs_bool);

    % Append the torque direction and indices to the results

```

```

        matrix_opposite_torque = [matrix_opposite_torque;
matrix_same_torque(i_mt, :)];
        idxs_num_opposite_torque = [idxs_num_opposite_torque; [i_mt,
j_mt]];

        % Mark these microthrusters as processed
        flag_mt_checked([i_mt, j_mt]) = 1;
end

```

Step 3: Formulate the optimization problem

```

% The goal is to distribute thrusts across the
% microthrusters such that the net torque matches the desired
control torque
% This is a least-squares problem

% The optimization minimizes ||Ax - b||_2^2 subject to Cx = d.

% Number of microthrusters with opposite torques
N_mt_opposite = size(matrix_opposite_torque, 1);

% Problem dimensions
n = N_mt_opposite; % Number of variables (thruster pairs)
p = 3;              % Number of constraints (force/torque balance
in 3D)

% Constraint matrix: Each column corresponds to the torque
contribution of a thruster pair
C = matrix_opposite_torque';

% Quadratic cost function: Ax approximates b, initialized as
identity for simplicity
A = eye(n);
b = zeros(n, 1);

% Form the Karush-Kuhn-Tucker (KKT) system
% E = [A'A  C'];
%      C    O]
O = zeros(p, p); % Zero block matrix for constraints
E = [A' * A, C';
     C,      O];

% Perform LU decomposition of the KKT matrix for efficient
solution
[L, U, P] = lu(E);

```

Step 4: Store optimization data in obj

```

obj.optim_data.A = A;
obj.optim_data.b = b;
obj.optim_data.U = U;
obj.optim_data.L = L;

```

```

obj.optim_data.P = P;
obj.optim_data.n = n;
obj.optim_data.N_mt = N_mt;
obj.optim_data.N_mt_opposite = N_mt_opposite;
obj.optim_data.N_mt_same = N_mt_same;
obj.optim_data.idx_num_opposite_torque =
idxs_num_opposite_torque;
obj.optim_data.idx_bool_same_torque = idxs_bool_same_torque;

end

```

[] Methods: Optimize Thrusters using KKT Condition

```

function obj =
func_decompose_control_torque_into_thrusters_optimization_kkt(obj, mission,
i_SC, desired_torque)

    % Optimize the use of micro-thrusters for the desired control
torque
    if nargin < 4
        % Allows to pass another torque as parameter
        desired_torque = obj.desired_control_torque;
    end

```

Step 1: Reset thruster commands

Reset the commanded thrust and torque for each microthruster to ensure a clean start

```
reset_thrusters(obj,mission, i_SC);
```

Step 2: Retrieve pre-computed optimization data

Extract optimization data that was initialized during the preparation phase.

```

A = obj.optim_data.A; % Quadratic cost matrix
b = obj.optim_data.b; % Right-hand side of the cost function
U = obj.optim_data.U; % Upper triangular matrix from LU decomposition
L = obj.optim_data.L; % Lower triangular matrix from LU decomposition
P = obj.optim_data.P; % Permutation matrix from LU decomposition
n = obj.optim_data.n; % Number of optimization variables
N_mt = obj.optim_data.N_mt; % Total number of microthrusters

```

```

        N_mt_opposite = obj.optim_data.N_mt_opposite;           % Number
of thruster pairs with opposite torque
        N_mt_same = obj.optim_data.N_mt_same;                   % Number
of unique torque directions
        idxs_num_opposite_torque =
obj.optim_data.idxs_num_opposite_torque; % Indices of opposite torque pairs
        idxs_bool_same_torque = obj.optim_data.idxs_bool_same_torque;
        % Logical array of same torque indices

```

Step 3: Solve the optimization problem using the KKT system

Formulate the KKT system and solve for the optimal thrust values.

```

        d = desired_torque';                                     % Desired control
torque for the spacecraft
        f = [A'*b; d];                                         % Combine
cost and constraint terms into a single vector
        xz = U\ (L\ (P*f));                                     % Solve the KKT
system using LU decomposition
        x_opposite = xz(1:n);                                   % Extract
the solution for thruster pairs

```

Step 4: Undo the torque pair reduction

Recover the thrust values for all microthrusters from the reduced solution.

```

        x_same = zeros(N_mt_same, 1);                           %
Preallocate thrust values for unique torque directions
        for i_mt = 1:N_mt_opposite
            idxs_num_i = idxs_num_opposite_torque(i_mt, :);    % Get the
indices of the current pair
            x_same(idxs_num_i(1)) = max(0, +x_opposite(i_mt)); % Assign
positive thrust to the first thruster
            x_same(idxs_num_i(2)) = max(0, -x_opposite(i_mt)); % Assign
negative thrust to the second thruster
        end

        % Distribute the thrust values among all microthrusters with the
same torque
        x = zeros(N_mt, 1);                                     %
Preallocate full thrust vector
        for i_mt = 1:N_mt_same
            idxs_bool_i = idxs_bool_same_torque(i_mt, :);      % Logical
indices for thrusters with same torque
            x(idxs_bool_i) = x_same(i_mt) / sum(idxs_bool_i);  % Evenly
distribute thrust among the group
        end

        % Scale all thrusts proportionally if any exceed the maximum limit
        if max(x) > max(obj.max_thrust)
            scaling_factor = max(x) / max(obj.max_thrust);

```

```
        x = x / scaling_factor;
    end
```

Step 5: Assign computed thrust and torque to each microthruster

Assign the thrust individually and safely

```
        for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster
            if x(i) >
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.minimum_thrust

mission.true_SC{i_SC}.true_SC_micro_thruster{i}.command_actuation = 1;
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.commanded_thrust = x(i);
            else

mission.true_SC{i_SC}.true_SC_micro_thruster{i}.command_actuation = 0;
mission.true_SC{i_SC}.true_SC_micro_thruster{i}.commanded_thrust = 0;
            end
        end

    end

    function func_compute_rw_command_kkt(obj, mission, i_SC)
        % Compute reaction wheel commands to achieve desired control
torque

        if norm(obj.desired_control_torque) > 0

            % Get the number of reaction wheels
            N =
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel;

            % Compute required wheel accelerations using optimization
            n = 4 * N; % number of variables
            p = 3 * (N + 1); % number of constraints

            % Constraint vector initialization
            d = zeros(p, 1);
            for i = 1:N
                idx1 = (i-1)*3 + 1;
                idx2 = (i-1)*3 + 3;
                % Assuming dot_angular_velocity is correctly accessed
                d(idx1:idx2, 1) =
mission.true_SC{i_SC}.software_SC_estimate_attitude.dot_angular_velocity';
            end

            % Scale desired torque vector
```

```

        torque_desired = obj.desired_control_torque;
        max_torque =
mission.true_SC{i_SC}.true_SC_reaction_wheel{1}.max_torque; % Assuming all
RWs have the same max torque
        if max_torque < Inf
            factor = min(1, (max_torque * 1) /
norm(torque_desired)); % envelope_ratio assumed as 1
        else
            factor = 1;
        end
        d(3*N + 1:p, 1) = factor * torque_desired;

% Objective matrix setup
A = diag([zeros(3*N, 1); ones(n - 3*N, 1)]);
b = zeros(n, 1);

% Constraint matrix setup
C = zeros(p, n);
for i = 1:N
    idx1 = (i-1)*3 + 1;
    idx2 = (i-1)*3 + 3;
    % Assuming orientation gives rotation matrix R (transposed
for usage here)
    R =
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.orientation';

    % Moment of inertia handling
    moment_of_inertia =
mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.moment_of_inertia;
    if any(diag(moment_of_inertia) == 0)
        % Regularization for singular inertia matrices
        moment_of_inertia = moment_of_inertia +
eye(size(moment_of_inertia)) * 1e-6;
    end

    C(idx1:idx2, idx1:idx2) = inv(moment_of_inertia);
    C(idx1:idx2, 3*N + i) = -R(:, 1);
    C(3*N + 1:p, idx1:idx2) = eye(3, 3);
end

% Solve optimization problem (constrained least squares)
% Form Karush-Kuhn-Tucker (KKT) system
O = zeros(p, p);
E = [A'*A, C'; C, O];
E = E + eye(size(E)) * 1e-6; % Regularization for numerical
stability

[L, U, P] = lu(E);

f = [A'*b; d];
xz = U\(L\(P*f));
x = xz(1:n);

% Results storing
for i = 1:N

```

```

        idx1 = (i-1)*3 + 1;
        idx2 = (i-1)*3 + 3;
        % Assign computed torque and angular acceleration

mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.commanded_torque =
x(idx1:idx2);

        % disp("RWA Commanded Torque")
        % disp(x(idx1:idx2))

mission.true_SC{i_SC}.true_SC_reaction_wheel{i}.commanded_angular_acceleration
= x(3*N + i, 1);

        % disp("RWA Commanded Angular Acceleration")
        % disp(x(3*N + i, 1))

    end

end

end

```

[] Methods: Control Attitude Oracle

Oracle directly moves the SC's attitude and rotation matrix, without the control actuators

```

function obj = func_update_software_SC_control_attitude_Oracle(obj,
mission, i_SC)
    % Oracle Move!
    mission.true_SC{i_SC}.true_SC_adc.attitude =
obj.desired_attitude; % [quaternion]
    mission.true_SC{i_SC}.true_SC_adc.angular_velocity =
obj.desired_angular_velocity; % [rad/sec]

    % Compute Rotation Matrix

func_update_true_SC_ADC_rotation_matrix(mission.true_SC{i_SC}.true_SC_adc);
end

```

[] Methods: Desired Control Torque Asymptotically Stable

```

function obj =
function_update_desired_control_torque_asymptotically_stable(obj, mission,
i_SC)

    omega_est =
mission.true_SC{i_SC}.software_SC_estimate_attitude.angular_velocity; % [rad/
sec]
    omega_desired = obj.desired_angular_velocity; % [rad/sec]
    beta_est =
func_quaternion_properize(mission.true_SC{i_SC}.software_SC_estimate_attitude.attitude);
[quaternion]
    beta_desired = func_quaternion_properize(obj.desired_attitude); %
[quaternion]

```

```

Zbeta = zeros(4,3);
Zbeta(1:3,1:3) = beta_est(4) * eye(3) + skew(beta_est(1:3));
Zbeta(4, 1:3) = -beta_est(1:3)';

% output desired control torque
omega_r = omega_desired + ( pinv(0.5 * Zbeta) *
obj.data.control_gain(2) * (beta_desired - beta_est)' )';
obj.desired_control_torque = - obj.data.control_gain(1)*(omega_est
- omega_r);

end

```

[] Methods: Desired Control Torque PD

```

function obj = function_update_desired_control_torque_PD(obj, mission,
i_SC)

% Parameters
J = mission.true_SC{i_SC}.true_SC_body.total_MI; % [kg m^2]
Spacecraft inertia matrix (excluding wheels)

Kp = eye(3) * obj.data.control_gain(1);
Kd = eye(3) * obj.data.control_gain(2);

% Estimated and final values
omega_est =
mission.true_SC{i_SC}.software_SC_estimate_attitude.angular_velocity; % [rad/
sec]

omega_desired = obj.desired angular_velocity; % [rad/sec]
delta_omega = omega_desired - omega_est;

beta_est =
func_quaternion_properize(mission.true_SC{i_SC}.software_SC_estimate_attitude.attitude);
[quaternion]
beta_desired = func_quaternion_properize(obj.desired_attitude); %
[quaternion]

% Calculate Delta q and Delta theta
delta_beta = func_quaternion_multiply(beta_desired,
func_quaternion_conjugate(beta_est));
delta_theta = 2 * delta_beta(1:3); % Assuming small angle
approximations

% Calculate control torques
uc = (J * (Kd * delta_omega' + Kp * delta_theta' ) )';

% Gyroscopic terms
vc = cross(omega_est, (J * omega_est')' ); % meas_rw_momentum

% Total control torque
obj.desired_control_torque = uc + vc;

```

end

[] Methods: Desired Attitude CVX

Compute Desired Attitude using CVX

```
function obj = func_compute_desired_attitude_CVX(obj)
    disp('Compute Desired Attitude using CVX')

    cvx_begin
        variable r(3,3)
        minimize ( norm(r * obj.data.secondary_vector'
- obj.data.desired_secondary_vector') + norm(r' *
obj.data.desired_secondary_vector' - obj.data.secondary_vector'))
        subject to

            r * obj.data.primary_vector' == obj.data.desired_primary_vector';
            r' * obj.data.desired_primary_vector' == obj.data.primary_vector';

    cvx_end

    [SC_e_desired, SC_Phi_desired] = func_decompose_Rot_matrix(r);

    SC_beta_v_desired = SC_e_desired*sin(SC_Phi_desired/2);
    SC_beta_4_desired = cos(SC_Phi_desired/2);
    obj.desired_attitude = [SC_beta_v_desired; SC_beta_4_desired];

end
```

[] Methods: Desired Attitude Array Search

Compute Desired Attitude using Array Search The array search component iteratively searches for an optimized secondary alignment by testing different rotation angles around the primary vector.

```
function obj = func_compute_desired_attitude_Array_Search(obj)

    % 1) Aligning the Primary Vector
    primary_vector = obj.data.primary_vector;
    desired_primary_vector = obj.data.desired_primary_vector;

    % Compute rotation to rotate pointing aligned with target (in
J2000)
    v = cross(primary_vector, desired_primary_vector);
    Rot_primary = eye(3) + skew(v)+skew(v)^2*(1/(1+dot(primary_vector,
desired_primary_vector)));

    % 2) Optimizing the Secondary Vector
    theta = 0:2*pi/60:2*pi;
    optimized_error_SP_pointing = zeros(1,length(theta));

    desired_secondary_vector = obj.data.desired_secondary_vector;
    secondary_vector = obj.data.secondary_vector;
```

```

        for i=1:length(theta)
            U = desired_primary_vector;
            R = func_axis_angle_rot(U, theta(i));
            optimized_error_SP_pointing(i) =
func_angle_between_vectors(desired_secondary_vector, R*secondary_vector');
        end

        [~,index_optimized_theta] = min(optimized_error_SP_pointing);
        Rot_secondary = func_axis_angle_rot(U,
theta(index_optimized_theta));

        % 3) Combine Rotations
        r = Rot_secondary * Rot_primary;

        % 4) Convert Rotation Matrix to Quaternion
        [SC_e_desired, SC_Phi_desired] =
func_decompose_rotation_matrix(r);

        SC_beta_v_desired = SC_e_desired' * sin(SC_Phi_desired/2);
        SC_beta_4_desired = cos(SC_Phi_desired/2);

        obj.desired_attitude =
func_quaternion_properize([SC_beta_v_desired, SC_beta_4_desired]);

    end

```

[] Methods: Update torque capabilities

Should be called if hardware configuration/health changes

```

function obj = update_torque_capabilities(obj, mission, i_SC)
    % Recalculate maximum torque capabilities after hardware changes
    obj.max_rw_torque =
obj.calculate_max_reaction_wheel_torque(mission, i_SC);
    obj.max_mt_torque = obj.calculate_max_thruster_torque(mission,
i_SC);
end

```

[] Methods: Calculate maximum reaction wheel torque capability

```

function max_torque = calculate_max_reaction_wheel_torque(obj,
mission, i_SC)
    % Calculate the maximum torque that can be provided by the
reaction wheel array

    % Initialize maximum torque
    max_torque = 0;

    % Count active (healthy) wheels
    active_wheels = 0;

```

```

        % Check each reaction wheel
        for i =
1:mission.true_SC{i_SC}.true_SC_body.numHardwareExists.num_reaction_wheel
            wheel = mission.true_SC{i_SC}.true_SC_reaction_wheel{i};

            if wheel.health
                % For each healthy wheel, add its maximum torque
capability
                wheel_max_torque = wheel.maximum_torque;

                % Calculate projection of wheel torque onto each axis
wheel_max_torque;
                wheel_torque_projection = wheel.orientation *

                % Add to overall torque capability
                max_torque = max_torque + norm(wheel_torque_projection);

                active_wheels = active_wheels + 1;
            end
        end

        % Apply a configuration factor based on wheel geometry
        % This is a conservative estimate of what the wheel array can
actually provide
        if active_wheels > 0
            % The factor accounts for the fact that not all wheels can
contribute equally
            % to torque in all directions
            config_factor = 1/sqrt(active_wheels);
            max_torque = max_torque * config_factor;
        end

        return;
    end

```

[] Methods: Calculate maximum thruster torque capability

```

function max_torque = calculate_max_thruster_torque(obj, mission,
i_SC)
    % Calculate the maximum torque that can be provided by micro
thrusters

    % Use the thruster contribution matrix and maximum thrust values
    % to calculate maximum possible torque

    % Initialize maximum torque
    max_torque = 0;

    % Calculate maximum torque for each principal axis
    for axis = 1:3

```

```

        % Calculate maximum positive torque
        pos_torque = 0;
        neg_torque = 0;

        for i =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster
            thruster =
mission.true_SC{i_SC}.true_SC_micro_thruster{i};

            % Skip unhealthy thrusters
            if ~thruster.health
                continue;
            end

            % Get torque contribution of this thruster for this axis
            torque_contrib = obj.thruster_contribution_matix(axis, i);

            % Add to positive or negative torque as appropriate
            if torque_contrib > 0
                pos_torque = pos_torque + torque_contrib *
thruster.maximum_thrust;
            elseif torque_contrib < 0
                neg_torque = neg_torque + abs(torque_contrib) *
thruster.maximum_thrust;
            end
        end

        % The maximum torque for this axis is the minimum of positive
and negative capability
        % (since we need to be able to control in both directions)
        axis_max_torque = min(pos_torque, neg_torque);

        % The overall maximum torque is the maximum across all axes
        max_torque = max(max_torque, axis_max_torque);
    end

    return;
end

```

Helper function: Calculate angle between quaternions

```

function angle = func_angle_between_quaternions(obj, q1, q2)
    % Calculates the angle between two quaternions in radians
    % This is the minimum rotation angle to get from one orientation
to the other

    % Ensure quaternions are normalized
    q1 = q1 / norm(q1);
    q2 = q2 / norm(q2);

    % Calculate the quaternion product  $q1^{-1} * q2$ 

```

```

        % For a quaternion, the inverse is the conjugate if it's
normalized
    q1_conj = q1;
    q1_conj(1:3) = -q1_conj(1:3); % Conjugate: negate vector part

    % Quaternion multiplication
    q_diff = zeros(4,1);
    if length(q1) == 4 && length(q2) == 4
        q_diff(1) = q1_conj(4)*q2(1) + q1_conj(1)*q2(4) +
q1_conj(2)*q2(3) - q1_conj(3)*q2(2);
        q_diff(2) = q1_conj(4)*q2(2) + q1_conj(2)*q2(4) +
q1_conj(3)*q2(1) - q1_conj(1)*q2(3);
        q_diff(3) = q1_conj(4)*q2(3) + q1_conj(3)*q2(4) +
q1_conj(1)*q2(2) - q1_conj(2)*q2(1);
        q_diff(4) = q1_conj(4)*q2(4) - q1_conj(1)*q2(1) -
q1_conj(2)*q2(2) - q1_conj(3)*q2(3);

        % The rotation angle is 2*acos(q_diff(4))
        % Clamp scalar part to [-1,1] to avoid numerical errors
        angle = 2 * acos(min(1, max(-1, q_diff(4))));
    else
        % Handle invalid quaternions by returning a large angle
        angle = pi;
        warning('Invalid quaternion dimensions in
func_angle_between_quaternions');
    end
end

end

end
end

```

Published with MATLAB® R2022a

5.3 Software_SC_Control_Orbit

Table of Contents

Class: Software_SC_Control_Orbit	1
Properties	1
Constructor	3
Methods	4
Update Storage	4

Class: Software_SC_Control_Orbit

Control the Orbit of the Spacecraft

```
classdef Software_SC_Control_Orbit < handle

    % Software_SC_Control_Orbit: Manages spacecraft control parameters for
    % orbiting small bodies.
    % Includes functionalities for trajectory calculations, intercept
    % predictions,
    % and delta-V computations tailored for spacecraft mission scenarios.
```

Properties

```
properties

    data % Other useful data

    mode_software_SC_control_orbit_selector % [string] Different orbit
    control modes
    % - 'DART Oracle' : Directly change True_SC_ADC.attitude values for
    DART mission
    % - 'Nightingale Oracle' : Directly change True_SC_ADC.attitude values
    for Nightingale mission
    % - 'Oracle with Control' : Use True_SC_ADC values + Noise

    last_time_control % [s]
    flag_executive % [bool]
    max_time_before_control % [s]

    % Maneuver tracking
    maneuver_start_time % [s] Time when current maneuver began execution
    thruster_fired_successfully % [bool] Flag to track if thruster
    actually fired

    % Delta-V control parameters
    desired_control_DeltaV % [m/s]: Desired Delta-V vector
    desired_control_DeltaV_units % [string]: Units for Delta-V
    total_DeltaV_executed % [m/s]: Accumulated executed Delta-V

    burn_duration % [s]

    % Thrust control parameters
```

```

desired_control_thrust % [N]: Desired thrust for trajectory control
desired_control_thrust_units % [string]: Units for thrust (e.g., N,
kN)

% Time horizons and intercept data
time_horizon % [sec]: Time horizon for trajectory planning
time_intercept % [sec]: Time of intercept with target
time_horizon_DeltaV % [sec]: Time horizon for Delta-V execution
time_DeltaV % [sec]: Time of Delta-V execution
time_horizon_data_cutoff % [sec]: Time cutoff for data integration
time_data_cutoff % [sec]: Data cutoff time

% Intercept and trajectory details
intercept_SB_position % [km]: Intercept position relative to small
body
intercept_SB_velocity % [km/s]: Intercept velocity relative to small
body
intercept_distance % [km]: Distance at intercept
desired_intercept_distance % [km]: Desired distance at intercept

% Options for numerical integrations
options % [struct]: Options for ODE solvers

% Flags and execution controls
desired_DeltaV_needs_to_be_executed % [boolean]: Whether Delta-V needs
execution
desired_DeltaV_computed % [boolean]: Whether Delta-V is computed
desired_attitude_for_DeltaV_achieved % [boolean]: Attitude flag for
Delta-V execution
desired_DeltaV_achieved % [boolean]: Whether Delta-V has been achieved

% Desired trajectory details
desired_time_array % [sec]: Array of planned trajectory times
desired_SC_pos_vel_current_SBcentered % [km, km/s]: Desired spacecraft
state in small body frame
flag_position_velocity_burn % [int]: Indicator for position or
velocity-based burns
desired_control_DeltaV_position_burn % [m/s]: Position-based Delta-V
time_DeltaV_position_burn % [sec]: Time for position-based burn
desired_control_DeltaV_velocity_burn % [m/s]: Velocity-based Delta-V
time_DeltaV_velocity_burn % [sec]: Time for velocity-based burn
threshold_minimum_deltaV % [m/s] - the minimum deltaV that is worth
executing, usually 0.001 m/s

% Fuel management properties
min_fuel_threshold % [kg]: Minimum fuel threshold to allow maneuvers
estimated_fuel_required % [kg]: Estimated fuel required for maneuver
flag_insufficient_fuel % [boolean]: Flag indicating insufficient fuel

store % Structure to store historical data
end

```

Constructor

```
methods
function obj = Software_SC_Control_Orbit(init_data, mission, i_SC)
    % Initialize the spacecraft control orbit class

    obj.mode_software_SC_control_orbit_selector =
init_data.mode_software_SC_control_orbit_selector;
    obj.max_time_before_control = init_data.max_time_before_control;

    obj.flag_executive = 0;
    obj.last_time_control = 0;

    obj.desired_control_thrust = 0; % Initialize thrust to zero
    obj.options = odeset('RelTol', 1e-14, 'AbsTol', 1e-14);

    obj.time_DeltaV = 0; % Initialize Delta-V time
    obj.time_horizon = 10 * 24 * 60 * 60; % Default planning horizon
(10 days)

    obj.desired_DeltaV_needs_to_be_executed = false;
    obj.desired_DeltaV_computed = false;
    obj.desired_attitude_for_DeltaV_achieved = 0;

    obj.intercept_distance = 0; % Initialize intercept distance
    obj.desired_intercept_distance = 0; % Initialize desired intercept
distance

    obj.desired_control_DeltaV = zeros(3, 1); % Initialize Delta-V
vector

    obj.desired_DeltaV_achieved = false;

    obj.total_DeltaV_executed = zeros(3, 1); % Initialize Delta-V
vector

    obj.time_horizon_DeltaV = 30 * 60; % Default Delta-V planning
horizon (30 minutes)
    obj.time_horizon_data_cutoff = 0; % No data cutoff initially

    obj.threshold_minimum_deltaV = 0.01;

    % Initialize fuel management properties
    obj.min_fuel_threshold = 0.1; % Minimum fuel threshold in kg
    obj.estimated_fuel_required = 0; % Estimated fuel required for
maneuver

    obj.flag_insufficient_fuel = false; % Flag indicating insufficient
fuel

    % Initialize Storage Variables
    obj.store = [];
```

```

        obj.store.intercept_distance =
zeros(mission.storage.num_storage_steps_attitude,1); % Initialize as empty
numeric array
        obj.store.desired_DeltaV_computed =
zeros(mission.storage.num_storage_steps_attitude,1);
        obj.store.desired_attitude_for_DeltaV_achieved =
zeros(mission.storage.num_storage_steps_attitude,1);

        obj.store.desired_control_DeltaV =
zeros(mission.storage.num_storage_steps_attitude, 3);
        obj.store.total_DeltaV_executed =
zeros(mission.storage.num_storage_steps_attitude, 3);
        obj.store.attitude_error =
zeros(mission.storage.num_storage_steps_attitude, 3); % Euler angles error
        obj.store.deltaV_magnitude_desired =
zeros(mission.storage.num_storage_steps_attitude, 1);
        obj.store.deltaV_magnitude_executed =
zeros(mission.storage.num_storage_steps_attitude, 1);
        obj.store.flag_insufficient_fuel =
zeros(mission.storage.num_storage_steps_attitude, 1);
        obj.store.estimated_fuel_required =
zeros(mission.storage.num_storage_steps_attitude, 1);

        obj.desired_DeltaV_achieved = 0;
        obj.total_DeltaV_executed = [0 0 0]'; % [m/sec]
        obj.desired_attitude_for_DeltaV_achieved = 0;

        % Initialize maneuver tracking properties
        obj.maneuver_start_time = 0;
        obj.thruster_fired_successfully = false;
    end
end

```

Methods

methods

Update Storage

```

function obj = reset_variables(obj)
    % Reset time and control flags
    %obj.last_time_control = 0;
    obj.flag_executive = 0;
    %obj.desired_DeltaV_needs_to_be_executed = false;
    %obj.desired_DeltaV_computed = false;
    %obj.desired_attitude_for_DeltaV_achieved = 0;
    %obj.desired_DeltaV_achieved = false;

    % Reset Delta-V and thrust parameters
    % obj.desired_control_DeltaV = zeros(3, 1);
    %obj.total_DeltaV_executed = zeros(3, 1);
    obj.desired_control_thrust = 0;

```

```

        % Reset intercept and trajectory details
        %obj.intercept_SB_position = zeros(3, 1);
        %obj.intercept_SB_velocity = zeros(3, 1);
        %obj.intercept_distance = 0;
        %obj.desired_intercept_distance = 0;
        %obj.time_intercept = 0;
        %obj.time_DeltaV = 0;
        %obj.time_data_cutoff = 0;

        % Reset fuel management
        obj.flag_insufficient_fuel = false;
    end

    function obj = func_update_software_SC_Control_Orbit_store(obj,
mission)
        obj.store.intercept_distance(mission.storage.k_storage,:) =
obj.intercept_distance;
        obj.store.desired_DeltaV_computed(mission.storage.k_storage,:) =
obj.desired_DeltaV_computed;

        obj.store.desired_attitude_for_DeltaV_achieved(mission.storage.k_storage,:) =
obj.desired_attitude_for_DeltaV_achieved;

        % New variables
        obj.store.desired_control_DeltaV(mission.storage.k_storage, :) =
obj.desired_control_DeltaV;
        obj.store.total_DeltaV_executed(mission.storage.k_storage, :) =
obj.total_DeltaV_executed;

        % % Compute attitude error
        % actual_quat =
mission.true_SC{i_SC}.software_SC_estimate_attitude.attitude;
        % desired_quat =
mission.true_SC{i_SC}.software_SC_control_attitude.desired_attitude;
        % error_quat = quatmultiply(quatconj(actual_quat), desired_quat);
        % euler_error = quat2eul(error_quat, 'ZYX'); % Convert to Euler
angles (rad)
        % obj.store.attitude_error(mission.storage.k_storage, :) =
euler_error;

        % DeltaV magnitudes
        obj.store.deltaV_magnitude_desired(mission.storage.k_storage) =
norm(obj.desired_control_DeltaV);
        obj.store.deltaV_magnitude_executed(mission.storage.k_storage) =
norm(obj.total_DeltaV_executed);

        % Fuel management data
        obj.store.flag_insufficient_fuel(mission.storage.k_storage) =
obj.flag_insufficient_fuel;
        obj.store.estimated_fuel_required(mission.storage.k_storage) =
obj.estimated_fuel_required;
    end

```

```

        % This function computes the intercept position, velocity, and time
        for a spacecraft
        % to intercept a target body. The computation involves simulating the
        orbital
        % trajectories of both the spacecraft and the target over a defined
        planning horizon
        % and finding the point of minimum distance between their positions.

        function func_estimate_target_intercept_location_time(obj, mission,
        i_SC)

            % Retrieve the current position and velocity of the target body as
            estimated
            % by the spacecraft's software.
            this_target_pos_vel_current =
            [mission.true_SC{i_SC}.software_SC_estimate_orbit.position_target(:);
            mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity_target(:)];

            this_time_array = 0:60:obj.time_horizon;

            % Get the position of the Sun at each time step in the J2000 frame
            % for the defined time array.
            Sun_pos_t0_tf = cspice_spkezr('10', mission.true_time.date +
            this_time_array, 'J2000', 'NONE', '10');

            % Simulate the trajectory of the target body using an ODE solver.
            % The function `func_orbit_SB_body` models the orbital motion of
            the body.
            [T, X] = ode113(@(t, X) func_orbit_SB_body(t, X, ...

            mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.mu, ...
            Sun_pos_t0_tf, this_time_array), this_time_array,
            this_target_pos_vel_current, obj.options);

            % Store the target body's position and velocity over the
            simulation period.
            target_pos_t0_tf = X';

            % Retrieve the current position and velocity of the spacecraft
            % as estimated by its onboard software.
            this_SC_pos_vel_current =
            [mission.true_SC{i_SC}.software_SC_estimate_orbit.position(:);
            mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity(:)];

            % Simulate the spacecraft's trajectory over the same time horizon.
            [T, X] = ode113(@(t, X) func_orbit_SB_body(t, X, ...

            mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.mu, ...
            Sun_pos_t0_tf, this_time_array), this_time_array,
            this_SC_pos_vel_current, obj.options);

            % Store the spacecraft's position and velocity over the simulation
            period.

```

```

SC_pos_t0_tf = X';

% Compute the Euclidean distance between the spacecraft and the
target body
% at each time step. The result is an array of distances.
Distance_array = vecnorm(SC_pos_t0_tf(1:3, :) -
target_pos_t0_tf(1:3, :), 2, 1);

% Find the minimum distance and the corresponding index in the
time array.
% `min_index` tells us the time step where the closest approach
occurs.
[min_distance, min_index] = min(Distance_array);

% Determine the time of interception based on the index of minimum
distance.
if min_index == 1
    % Case 1: If the minimum distance occurs at the first time
step, it implies
    % that the orbits are diverging (the closest point is now, and
they are moving apart).
    obj.time_intercept = mission.true_time.time +
obj.time_horizon/5 + ...
    obj.time_horizon_DeltaV + obj.time_horizon_data_cutoff;
elseif (min_index > 1) && (min_index < length(Distance_array))
    % Case 2: The minimum distance occurs somewhere within the
planning horizon.
    obj.time_intercept = mission.true_time.time +
this_time_array(min_index);
    % If the calculated intercept time is too soon (before certain
operational
    % thresholds like `time_horizon_DeltaV`), adjust the intercept
time.
    if (obj.time_intercept - mission.true_time.time) < ...
(obj.time_horizon_DeltaV +
obj.time_horizon_data_cutoff)
        obj.time_intercept = mission.true_time.time +
obj.time_horizon/5 + ...
        obj.time_horizon_DeltaV +
obj.time_horizon_data_cutoff;
    end
else
    % Case 3: If the minimum distance occurs at the last time
step, it suggests
    % that the spacecraft and target body are converging slowly
but don't intercept
    % within the planning horizon.
    obj.time_intercept = mission.true_time.time +
obj.time_horizon;
end

% Record the target body's position and velocity at the time of
interception.

```

```

        % These values correspond to the time step with the minimum
distance.
        obj.intercept_SB_position = target_pos_t0_tf(1:3, min_index);
        obj.intercept_SB_velocity = target_pos_t0_tf(4:6, min_index);

        % Record the minimum distance at interception.
        obj.intercept_distance = min_distance;
    end

    function verify_fuel_availability(obj, mission, i_SC)

        % Get distance to target
        target_pos =
mission.true_SC{i_SC}.software_SC_estimate_orbit.position_target;
        sc_pos =
mission.true_SC{i_SC}.software_SC_estimate_orbit.position;
        distance_to_target = norm(target_pos - sc_pos);

        % Check if we're already so close that we can consider this an
intercept
        % For an impactor mission, if we're within 5km, we're essentially
on an intercept course already
        % This prevents unnecessary maneuvers when we're extremely close
        if distance_to_target < 5.0
            % Get relative velocity to check if trajectory is favorable
            target_vel =
mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity_target;
            sc_vel =
mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity;
            rel_vel = sc_vel - target_vel;

            % Calculate dot product to see if we're approaching or moving
away
            approach_direction = dot(rel_vel, (target_pos - sc_pos) /
distance_to_target);

            if approach_direction < 0 % Negative means approaching
                disp(['Already on intercept course at distance of ',
num2str(distance_to_target), ...
                    ' km with approach velocity of ', num2str(-
approach_direction), ' km/s. Skipping maneuver calculation.']);

                % Skip the maneuver since we're already approaching the
target
                obj.flag_insufficient_fuel = false; % Not a fuel issue
                obj.desired_DeltaV_needs_to_be_executed = false;
                obj.desired_DeltaV_computed = false;
                return;
            end
        end
    end
end
end
end

```

```

    % Compute Transfer Correction Maneuver (TCM) using Lambert-Battin
method
    function func_compute_TCM_Lambert_Battin(obj, mission, i_SC)

        % Verify fuel availability
        obj.verify_fuel_availability(mission, i_SC);

        % We are solving the Lambert Battin problem using the position
        % of the spacecraft after 'time_horizon_DeltaV' has passed.
        this_time_array = 0:10:obj.time_horizon_DeltaV
+obj.time_horizon_data_cutoff;
        Sun_pos_t0_tf = cspice_spkezr('10', mission.true_time.date +
        this_time_array, 'J2000', 'NONE', '10');
        new_this_SC_pos_vel_current =
[mission.true_SC{i_SC}.software_SC_estimate_orbit.position(:);
mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity(:)]; % [km, km/sec]

        % Estimated SC orbit
        [T,X] = ode113(@(t,X) func_orbit_SB_body(t,X,
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.mu,
Sun_pos_t0_tf,
this_time_array),this_time_array,new_this_SC_pos_vel_current,obj.options);
        SC_pos_t_DeltaV = X(end,:);
        r1 = 1e3*SC_pos_t_DeltaV(1:3); % [m]

        % Extract the positions
        r2 = obj.intercept_SB_position * 1e3; % [m]

        % Time of flight [sec]
        tof = obj.time_intercept - mission.true_time.time -
obj.time_horizon_DeltaV - obj.time_horizon_data_cutoff;

        % Safety check: Ensure time of flight is reasonable
        if tof <= 0
            warning('Time of flight is negative or zero. Adjusting
intercept time. ');
            tof = 3600; % Set a default 1 hour time of flight
            obj.time_intercept = mission.true_time.time +
obj.time_horizon_DeltaV + obj.time_horizon_data_cutoff + tof;
        end

        % Solve Lambert's problem
        [V1, ~] = LAMBERTBATTIN(r1, r2, 'pro', tof);

        % Compute required Delta-V
        obj.desired_control_DeltaV = V1' - 1e3*SC_pos_t_DeltaV(4:6); % [m/
s]

        % Now that we have the new deltaV we can update the intercept
distance.
        % Estimate SB orbit
        this_target_pos_vel_current =
[mission.true_SC{i_SC}.software_SC_estimate_orbit.position_target(:);

```

```

mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity_target(:)];
    [~, X] = ode113(@(t, X) func_orbit_SB_body(t, X, ...

mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.mu, ...
    Sun_pos_t0_tf, this_time_array), this_time_array,
this_target_pos_vel_current, obj.options);
    SB_pos_t0_tf = X';

    % Estimate SC orbit
    % We propagate using the time after the deltaV begins (after
time_horizon_DeltaV)
    new_this_time_array = obj.time_horizon_DeltaV
+obj.time_horizon_data_cutoff : 10 : obj.time_intercept -
mission.true_time.time; % [sec]
    new_this_SC_pos_vel_current = [SC_pos_t_DeltaV(1:3);
SC_pos_t_DeltaV(4:6) + (1e-3*obj.desired_control_DeltaV)]; % [km, km/sec]
    Sun_pos_t0_tf = cspice_spkezr('10', mission.true_time.date +
new_this_time_array, 'J2000', 'NONE', '10');

    [~,X] = ode113(@(t,X) func_orbit_SB_body(t,X,
mission.true_solar_system.SS_body{mission.true_solar_system.index_Sun}.mu,
Sun_pos_t0_tf,
new_this_time_array),new_this_time_array,new_this_SC_pos_vel_current,obj.options);

    SC_pos_tDeltaV_tf = X';

    new_intercept_distance = norm(SC_pos_tDeltaV_tf(1:3,end) -
SB_pos_t0_tf(1:3,end));
    obj.desired_intercept_distance = new_intercept_distance; % [km]

    obj.time_DeltaV = mission.true_time.time + obj.time_horizon_DeltaV
+ obj.time_horizon_data_cutoff; % [sec] since t0
    obj.time_data_cutoff = mission.true_time.time +
obj.time_horizon_data_cutoff; % [sec] since t0

    % Calculate estimated fuel required for the maneuver
    % Using the rocket equation: m_fuel = m_sc * (1 - exp(-deltaV/g0/
Isp))
    % But for small deltaV, we can approximate: m_fuel # m_sc *
deltaV/(g0*Isp)
    g0 = 9.80665; % m/s^2

    % Get average ISP from all available chemical thrusters
    total_isp = 0;
    num_thrusters = 0;
    for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster
        if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health
            total_isp = total_isp +
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).isp;
            num_thrusters = num_thrusters + 1;

```

```

        end
    end

    % Calculate average ISP if any thruster is healthy
    if num_thrusters > 0
        avg_isp = total_isp / num_thrusters;
    else
        avg_isp = 200; % Default ISP if no healthy thrusters
        warning('No healthy thrusters available. Using default ISP of
200s.');
```

200s.');

```

    end

    % Estimate fuel required
    sc_mass = mission.true_SC{i_SC}.true_SC_body.total_mass;
    deltaV_magnitude = norm(obj.desired_control_DeltaV);
    obj.estimated_fuel_required = sc_mass * deltaV_magnitude / (g0 *
avg_isp);

    % Check if we have sufficient fuel
    total_available_fuel = 0;
    if isfield(mission.true_SC{i_SC}, 'true_SC_fuel_tank') && ...

mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank > 0
        for i_tank =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank
            total_available_fuel = total_available_fuel + ...

mission.true_SC{i_SC}.true_SC_fuel_tank{i_tank}.instantaneous_fuel_mass;
        end
    end

    % Set flags based on fuel availability
    if total_available_fuel < (obj.estimated_fuel_required +
obj.min_fuel_threshold) &&
~mission.true_SC{i_SC}.true_SC_navigation.flag_SC_crashed
        obj.flag_insufficient_fuel = true;
        warning(['Insufficient fuel for maneuver. Required: ', ...
            num2str(obj.estimated_fuel_required), ' kg, Available:
', ...
            num2str(total_available_fuel), ' kg']);

    % Don't execute the maneuver if we don't have enough fuel
    obj.desired_DeltaV_needs_to_be_executed = false;
    obj.desired_DeltaV_computed = false;
else
    obj.flag_insufficient_fuel = false;
    obj.desired_DeltaV_needs_to_be_executed = true;
    obj.desired_DeltaV_computed = true;

end

obj.desired_DeltaV_achieved = false;
obj.total_DeltaV_executed = [0 0 0]'; % [m/sec]
obj.desired_attitude_for_DeltaV_achieved = false;

```

```

end

function func_command_DeltaV(obj, mission, i_SC)
    % Initialize desired control thrust to zero
    obj.desired_control_thrust = 0;

    % Get spacecraft parameters
    sc_mass = mission.true_SC{i_SC}.true_SC_body.total_mass;
    dt = mission.true_time.time_step;

    % Check for healthy thrusters
    healthy_thrusters = [];
    for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster
        if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health
            healthy_thrusters = [healthy_thrusters, i_thruster];
        end
    end

    if isempty(healthy_thrusters)
        warning('No healthy thrusters available for DeltaV
execution. ');
        return;
    end

    % Check execution conditions
    if (mission.true_time.time < obj.time_DeltaV) ||
obj.desired_DeltaV_achieved || obj.flag_insufficient_fuel
        return;
    end

    % Check if thruster is ready for firing
    all_thrusters_ready = true;
    for i = 1:length(healthy_thrusters)
        if
~mission.true_SC{i_SC}.true_SC_chemical_thruster(healthy_thrusters(i)).func_is_thruster_r
            all_thrusters_ready = false;
            break;
        end
    end

    if ~all_thrusters_ready
        % Thrusters still warming up, wait until next cycle
        return;
    end

    % Calculate remaining Delta-V
    remaining_DeltaV = obj.desired_control_DeltaV -
obj.total_DeltaV_executed;
    remaining_magnitude = norm(remaining_DeltaV);

    if remaining_magnitude > 0

```

```

        % Record start time if this is the first thrust of a maneuver
        if obj.maneuver_start_time == 0
            obj.maneuver_start_time = mission.true_time.time;
            obj.thruster_fired_successfully = false;
            disp(['Starting maneuver execution at time ',
num2str(mission.true_time.time), ' s']);
        end

        % Get thruster parameters
        max_thrusts = zeros(1, length(healthy_thrusters));
        for i = 1:length(healthy_thrusters)
            max_thrusts(i) =
mission.true_SC{i_SC}.true_SC_chemical_thruster(healthy_thrusters(i)).maximum_thrust;
        end

        % Get current attitude matrix
        current_attitude = mission.true_SC{i_SC}.true_SC_adc.attitude;
        R = quaternionToRotationMatrix(current_attitude);

        % Rotate thruster directions to inertial frame
        thruster_body_directions = zeros(3,
length(healthy_thrusters));
        for i = 1:length(healthy_thrusters)
            thruster_body_directions(:, i) =
mission.true_SC{i_SC}.true_SC_chemical_thruster(healthy_thrusters(i)).orientation';
        end
        orientations = R * thruster_body_directions;
        A = orientations;

        % Calculate maximum available thrust in desired direction
        max_thrust_dir = A * max_thrusts';
        max_DeltaV_per_step = (norm(max_thrust_dir) * dt) / sc_mass;

        % Calculate required thrust scaling factors
        if remaining_magnitude > max_DeltaV_per_step
            % Use maximum thrust for each thruster
            thrust_vector = max_thrusts;
        else
            % Calculate required thrust to achieve the desired delta-V
in this step
            required_thrust_magnitude = (remaining_magnitude *
sc_mass) / dt;

            % Distribute thrust among thrusters (simplified approach -
equal distribution)
            thrust_vector = zeros(1, length(healthy_thrusters));
            for i = 1:length(healthy_thrusters)
                thrust_vector(i) = required_thrust_magnitude /
length(healthy_thrusters);

                % Clamp to thruster limits
                thrust_vector(i) = min(max(thrust_vector(i), ...

mission.true_SC{i_SC}.true_SC_chemical_thruster(healthy_thrusters(i)).minimum_thrust), ..

```

```

mission.true_SC{i_SC}.true_SC_chemical_thruster(healthy_thrusters(i)).maximum_thrust);
    end
end

    % Apply thrust commands to thrusters
    for i = 1:length(healthy_thrusters)

mission.true_SC{i_SC}.true_SC_chemical_thruster(healthy_thrusters(i)).commanded_thrust
= thrust_vector(i);

mission.true_SC{i_SC}.true_SC_chemical_thruster(healthy_thrusters(i)).flag_executive
= 1;

        % Save that we commanded a thrust (needed for completion
checks)
        obj.desired_control_thrust = obj.desired_control_thrust +
thrust_vector(i);
    end

        % Note: We no longer calculate applied DeltaV here - that's
now done in the thruster itself
        % which will directly update obj.total_DeltaV_executed
    end
end

function reset_after_completion(obj, mission, i_SC)

    % IMPORTANT: Clear all maneuver flags and reset values to zero
    obj.desired_DeltaV_achieved = true;
    obj.desired_DeltaV_needs_to_be_executed = false;
    obj.desired_control_DeltaV = [0 0 0]';
    obj.desired_DeltaV_computed = false;
    obj.last_time_control = mission.true_time.time;
    obj.total_DeltaV_executed = [0 0 0]';

    % Reset maneuver tracking
    obj.maneuver_start_time = 0;
    obj.thruster_fired_successfully = false;
    obj.flag_executive = 0;

    % Add: Clear pending thruster commands
    for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster
        if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).pending_fire =
false;

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).pending_thrust =
0;
    end
end

```

```

        end
    end

    function func_main_software_SC_control_orbit(obj, mission, i_SC)

        % Main control loop for spacecraft orbit
        if obj.flag_executive

            first_pass = false;

            % Always verify the capacity of the fuel tanks.
            total_fuel = 0;
            if
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank > 0
                for i_tank =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank
                    total_fuel = total_fuel +
mission.true_SC{i_SC}.true_SC_fuel_tank{i_tank}.instantaneous_fuel_mass;
                end

                % Log fuel level if it falls below certain thresholds
                if total_fuel < 0.5
                    warning('CRITICAL: Spacecraft fuel level below 0.5 kg.
Remaining: %.3f kg', total_fuel);
                elseif total_fuel < 1.0
                    warning('WARNING: Spacecraft fuel level below 1.0 kg.
Remaining: %.3f kg', total_fuel);
                end
            end

            if ~obj.desired_DeltaV_computed

                % First, assess whether a maneuver calculation is actually
needed
                [is_maneuver_needed, reason] =
obj.func_assess_maneuver_necessity(mission, i_SC);

                if ~is_maneuver_needed
                    % Log the reason we're skipping the maneuver
calculation
                    disp(['Skipping maneuver calculation: ', reason]);

                    % If we're skipping, make sure to reset the flags
                    obj.reset_after_completion(mission, i_SC);
                    return;
                end

                % Maneuver seems necessary, proceed with calculations
                obj.func_estimate_target_intercept_location_time(mission,
i_SC);
            end
        end
    end
end

```

```

        obj.func_compute_TCM_Lambert_Battin(mission, i_SC);

        % Log the computed maneuver for mission awareness
        if obj.desired_DeltaV_computed
            disp(['Computed maneuver: DeltaV magnitude = ',
num2str(norm(obj.desired_control_DeltaV)), ' m/s']);
            disp(['Execution time: ', datestr(datetime('now') +
seconds(obj.time_DeltaV - mission.true_time.time))]);
            disp(['Estimated fuel required: ',
num2str(obj.estimated_fuel_required), ' kg']);
        end

        first_pass = true;
    end

    % Validate attitude and execute DeltaV if conditions are met
    if obj.desired_DeltaV_needs_to_be_executed && ...
        (mission.true_time.time >= obj.time_DeltaV) && ...
        ~first_pass && ...

(norm(mission.true_SC{i_SC}.software_SC_control_attitude.get_attitude_error(mission,i_SC)
< 0.03) && ... % rad
        ~obj.flag_insufficient_fuel
        obj.func_command_DeltaV(mission, i_SC);
    end

    % Handle thruster warm-up logic if maneuver is planned
    if obj.desired_DeltaV_needs_to_be_executed &&
obj.desired_DeltaV_computed
        % Check if remaining DeltaV is above threshold
        remaining_DeltaV = obj.desired_control_DeltaV -
obj.total_DeltaV_executed;
        if norm(remaining_DeltaV) < obj.threshold_minimum_deltaV
            % If DeltaV is below threshold, consider it complete
            disp(['Skipping maneuver - remaining DeltaV (',
num2str(norm(remaining_DeltaV)), ' ...
                ' m/s) below threshold (',
num2str(obj.threshold_minimum_deltaV), ' m/s)']);

            % Stop warming up thruster if needed
            for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster
                if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).flag_warming_up =
false;

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).flag_executive =
false;

                end
            end
        end
    end

```

```

        obj.reset_after_completion(mission, i_SC);
        return;
    end

    % Check for sufficient fuel before proceeding
    if
mission.true_SC{i_SC}.true_SC_body.num hardware_exists.num_fuel_tank > 0
        total_fuel = 0;
        for i_tank =
1:mission.true_SC{i_SC}.true_SC_body.num hardware_exists.num_fuel_tank
            total_fuel = total_fuel +
mission.true_SC{i_SC}.true_SC_fuel_tank{i_tank}.instantaneous_fuel_mass;
        end

        % Add safety margin
        fuel_with_margin = obj.estimated_fuel_required *
1.1; % 10% margin

        if total_fuel < fuel_with_margin
            warning('Insufficient fuel for DeltaV execution.
Required: %.3f kg (with margin), Available: %.3f kg', ...
                fuel_with_margin, total_fuel);

            % Cancel the maneuver
            obj.flag_insufficient_fuel = true;
            obj.desired_DeltaV_needs_to_be_executed = false;
            obj.desired_DeltaV_computed = false;

            % Stop warming up thruster
            for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num hardware_exists.num_chemical_thruster
                if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).flag_warming_up =
false;

                    end
                end
            end
            return;
        end
    end

    % Get time until DeltaV execution
    time_to_DeltaV = obj.time_DeltaV - mission.true_time.time;

    % Get thruster warm-up time
    thruster_warm_up_time = 30; % Default value
    for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num hardware_exists.num_chemical_thruster
        if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health
            thruster_warm_up_time =
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).thruster_warm_up_time;

```

```

        break;
    end
end

% Start thruster warm-up with a safety margin (45 sec
before needed)
thruster_warming = false;
for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster
    if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).flag_warming_up
        thruster_warming = true;
        break;
    end
end

    if time_to_DeltaV > 0 && time_to_DeltaV <=
(thruster_warm_up_time + 45) && ~thruster_warming
        % Time to start warming up the thruster
        for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster
            if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).flag_warming_up =
true;

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).func_start_warm_up(mission);
            end
        end
        disp(['Starting thruster warm-up at T-',
num2str(time_to_DeltaV), ' seconds before maneuver']);
    end
end

% Check if maneuver is complete - simplified approach
if obj.desired_DeltaV_needs_to_be_executed &&
obj.desired_DeltaV_computed
    % Check remaining DeltaV - this is updated directly by the
thruster
    remaining_DeltaV = obj.desired_control_DeltaV -
obj.total_DeltaV_executed;

    % Check timeout conditions
maneuver_timeout = false;
time_since_planned = mission.true_time.time -
obj.time_DeltaV;

    % Timeout if running too long since start
    if obj.maneuver_start_time > 0 && (mission.true_time.time
- obj.maneuver_start_time) > 120
        maneuver_timeout = true;
        warning('Maneuver timeout reached after %d seconds
from maneuver start', ...

```

```

        mission.true_time.time - obj.maneuver_start_time);
    end

    % Timeout if waiting too long after planned execution
    if time_since_planned > 120 && ~maneuver_timeout
        maneuver_timeout = true;
        warning('Maneuver timeout: %d seconds since planned
execution time', time_since_planned);
    end

    % Check if we should complete the maneuver
    is_deltaV_complete = norm(remaining_DeltaV) <
obj.threshold_minimum_deltaV;
    if is_deltaV_complete || maneuver_timeout ||
time_since_planned > 60
        % Report reason for completion
        if is_deltaV_complete
            disp(['Maneuver completed: Target DeltaV achieved
within ', ...
                num2str(obj.threshold_minimum_deltaV), ' m/s
threshold']);
        elseif maneuver_timeout
            disp(['Maneuver timeout after ',
num2str(mission.true_time.time - obj.maneuver_start_time), ' seconds']);
        else
            disp(['Maneuver forced to complete after ',
num2str(time_since_planned), ' seconds from planned time']);
        end

        % Reset everything
        obj.reset_after_completion(mission, i_SC);

        % Turn off thruster warm-up
        for i_thruster =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster
            if
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).health
mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).flag_warming_up =
false;

mission.true_SC{i_SC}.true_SC_chemical_thruster(i_thruster).commanded_thrust
= 0;

            end
        end

        disp('Maneuver completed and all flags reset');
    end
end
end

% Store and reset
obj.func_update_software_SC_Control_Orbit_store(mission);

```

```

        % Reset the executive flag
        obj.flag_executive = 0;
    end

    function [is_needed, reason] = func_assess_maneuver_necessity(obj,
mission, i_SC)
        % This function evaluates whether a maneuver calculation is
actually needed
        % based on current trajectory and intercept parameters

        % Default to needing a maneuver
        is_needed = true;
        reason = '';

        % Get orbit estimation data
        position_sc =
mission.true_SC{i_SC}.software_SC_estimate_orbit.position;
        velocity_sc =
mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity;

        position_target =
mission.true_SC{i_SC}.software_SC_estimate_orbit.position_target;
        velocity_target =
mission.true_SC{i_SC}.software_SC_estimate_orbit.velocity_target;

        % Calculate distance to target
        rel_position = position_target - position_sc;
        distance_to_target = norm(rel_position);

        % Calculate relative velocity
        rel_velocity = velocity_target - velocity_sc;
        relative_speed = norm(rel_velocity);

        approach_projection = dot(rel_position, rel_velocity) /
(distance_to_target * relative_speed);

        % Check if we're already very close to target (less than 3 km)
        if distance_to_target < 3
            % Check if we're on a reasonable approach (approaching, not
departing)

            % If projection is negative, we're approaching
            if approach_projection < 0
                % Calculate miss distance approximation using simple
projection
                is_needed = false;
                reason = ['Already on successful intercept trajectory.
Distance: ', num2str(distance_to_target), ' km'];
            end
        end

        % Check available fuel vs expected requirements

```

```

        if is_needed &&
mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank > 0
            % Estimate using very rough heuristic - actual calculation
            will happen in Lambert-Battin
            % This is just a pre-check to avoid unnecessary calculation
            available_fuel =
mission.true_SC{i_SC}.true_SC_fuel_tank{1}.instantaneous_fuel_mass;

            % Very rough estimate of fuel needed (not accurate, just for
pre-screening)
            approximate_fuel_required = relative_speed * 0.005 *
mission.true_SC{i_SC}.true_SC_body.total_mass / 3000;

            % If clearly insufficient fuel (with large margin to be safe)
            if approximate_fuel_required > 10 * available_fuel
                is_needed = false;
                reason = ['Insufficient fuel for likely maneuver.
Est. required: ', num2str(approximate_fuel_required), ' kg, Available: ',
num2str(available_fuel), ' kg'];
            end
        end

        % Check if time-to-intercept is very short
        if is_needed && approach_projection < 0
            % Estimate time to closest approach
            time_to_closest_approach = distance_to_target /
relative_speed;

            % If very close to intercept (less than 1 minute)
            if time_to_closest_approach < 60
                is_needed = false;
                reason = ['Intercept imminent (',
num2str(time_to_closest_approach), ' seconds). Maneuver computation
skipped.'];
            end
        end
    end
end
end
end
end
end

```

Published with MATLAB® R2022a

5.4 Software_SC_Data_Handling

Table of Contents

Class: Software_SC_Data_Handling	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	1
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	3
[] Methods: Update Mean SoDS	4

Class: Software_SC_Data_Handling

Track the data_handling status onboard the spacecraft

```
classdef Software_SC_Data_Handling < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

mode_software_SC_data_handling_selector % [string] Different Data
Handling modes
% - 'Generic'
```

[] Properties: Variables Computed Internally

```
name % [string] = SC j for jth SC + SW Data Handling

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

total_data_storage % [kb] : Total data in all memories

mean_state_of_data_storage % [percentage] : SoDS is defined by = 100×
Sum (instantaneous_capacity) / Sum (maximum_capacity)

data % Other useful data
```

[] Properties: Storage Variables

```
store
```

end

Methods

methods

[] Methods: Constructor

Construct an instance of this class

```
function obj = Software_SC_Data_Handling(init_data, mission, i_SC)

    obj.name = [mission.true_SC{i_SC}.true_SC_body.name, ' SW Data
Handling']; % [string]
    obj.flag_executive = 0;

    obj.instantaneous_data_generated_per_sample =
init_data.instantaneous_data_generated_per_sample; % [kb]

    obj.mode_software_SC_data_handling_selector =
init_data.mode_software_SC_data_handling_selector;

    obj.mode_software_SC_data_handling_selector = 'Generic';

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    % Update Mean SoC
    obj = func_update_mean_state_of_data_storage(obj, mission, i_SC);

    obj.mean_state_of_data_storage = 0;

    % Initialize Variables to store
    obj.store = [];

    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
    obj.store.mean_state_of_data_storage =
zeros(mission.storage.num_storage_steps,
length(obj.mean_state_of_data_storage));
    obj.store.total_data_storage =
zeros(mission.storage.num_storage_steps, length(obj.total_data_storage));

    % Update Storage
    obj = func_update_software_SC_data_handling_store(obj, mission);

    % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);
```

```
end
```

[] Methods: Store

Update the store variables

```
function obj = func_update_software_SC_data_handling_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage,:) = obj.flag_executive; % [Boolean]

    obj.store.mean_state_of_data_storage(mission.storage.k_storage,:) = obj.mean_state_of_data_storage; % [percentage]
        obj.store.total_data_storage(mission.storage.k_storage,:) = obj.total_data_storage; % [kb]
    end

end
```

[] Methods: Main

Main Function

```
function obj = func_main_software_SC_data_handling(obj, mission, i_SC)

    switch obj.mode_software_SC_data_handling_selector

        case 'Generic'
            % Update Mean SoC
            obj = func_update_mean_state_of_data_storage(obj, mission, i_SC);

        case 'Nightingale'
            % Update Mean SoC for (num_onboard_memory-1) only
            obj = func_update_mean_state_of_data_storage_Nightingale(obj, mission, i_SC);

        otherwise
            error('Data Handling mode not defined!')
        end

        % Update Data Generated

    func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling, obj, mission);

    % Update Storage
    obj = func_update_software_SC_data_handling_store(obj, mission);
```

```
    % Reset Variables
    obj.flag_executive = 0;

end
```

[] Methods: Update Mean SoDS

Updates mean_state_of_data_storage

```
function obj = func_update_mean_state_of_data_storage(obj, mission,
i_SC)

    total_instantaneous_capacity = 0; % [kb]
    total_maximum_capacity = 0; % [kb]

    for i_memory =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_memory

        total_instantaneous_capacity = total_instantaneous_capacity +
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.instantaneous_capacity; %
[kb]
        total_maximum_capacity = total_maximum_capacity +
mission.true_SC{i_SC}.true_SC_onboard_memory{i_memory}.maximum_capacity; %
[kb]

    end

    obj.mean_state_of_data_storage = 100 *
total_instantaneous_capacity / total_maximum_capacity; % [percentage]

    obj.total_data_storage = total_instantaneous_capacity; % [kb]

end

end

end
```

Published with MATLAB® R2022a

5.5 Software_SC_Estimate_Attitude

Table of Contents

Class: Software_SC_Estimate_Attitude	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	4
[] Methods: Truth Estimate Attitude	5
[] Methods: Estimate Attitude using Kalman Filter (KF)	5

Class: Software_SC_Estimate_Attitude

Estimates the Attitude of the Spacecraft

```
classdef Software_SC_Estimate_Attitude < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_data_generated_per_sample % [kb] : Data generated per  
sample, in kilo bits (kb)
```

```
mode_software_SC_estimate_attitude_selector % [string] Different  
attitude estimation modes  
% - 'Truth' : Use True_SC_ADC values  
% - 'Truth with Noise' : Use True_SC_ADC values + Noise  
% - 'KF' : Use Kalman Filter
```

[] Properties: Variables Computed Internally

```
name % [string] = SC j for jth SC + SW Estimate Attitude
```

```
flag_executive % [Boolean] Executive has told this sensor/actuator to  
do its job
```

```
attitude % [quaternion] : Orientation of inertial frame I with respect  
to the body frame B
```

```
angular_velocity % [rad/sec] : Angular velocity of inertial frame I  
with respect to the body frame B
```

```
dot_angular_velocity % [rad/sec^2] : Time derivative of  
angular_velocity (needed by RWA)
```

```

attitude_uncertainty % [error in quaternion]

angular_velocity_uncertainty % [rad/sec]

dot_angular_velocity_uncertainty % [rad/sec^2]

data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = Software_SC_Estimate_Attitude(init_data, mission, i_SC)

    obj.name = [mission.true_SC{i_SC}.true_SC_body.name, ' SW Estimate
Attitude']; % [string]
    obj.flag_executive = 1;

    obj.instantaneous_data_generated_per_sample = 0; % [kb]
    obj.mode_software_SC_estimate_attitude_selector =
init_data.mode_software_SC_estimate_attitude_selector; % [string]

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    obj = func_update_software_SC_estimate_attitude_Truth(obj,
mission, i_SC);

    % Initialize Variables to store: attitude angular_velocity
dot_angular_velocity and uncertainties
    obj.store = [];

    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.flag_executive));

    obj.store.attitude =
zeros(mission.storage.num_storage_steps_attitude, length(obj.attitude));

```

```

        obj.store.attitude_uncertainty =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.attitude_uncertainty));

        obj.store.angular_velocity =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.angular_velocity));
        obj.store.angular_velocity_uncertainty
= zeros(mission.storage.num_storage_steps_attitude,
length(obj.angular_velocity_uncertainty));

        obj.store.dot_angular_velocity =
zeros(mission.storage.num_storage_steps_attitude,
length(obj.dot_angular_velocity));
        obj.store.dot_angular_velocity_uncertainty
= zeros(mission.storage.num_storage_steps_attitude,
length(obj.dot_angular_velocity_uncertainty));

        % Update Storage
        obj = func_update_software_SC_estimate_attitude_store(obj,
mission);

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_software_SC_estimate_attitude_store(obj,
mission)

    if mission.storage.flag_store_this_time_step_attitude == 1
        obj.store.flag_executive(mission.storage.k_storage_attitude,:)
= obj.flag_executive;

        obj.store.attitude(mission.storage.k_storage_attitude,:) =
obj.attitude; % [quaternion]

        obj.store.attitude_uncertainty(mission.storage.k_storage_attitude,:) =
obj.attitude_uncertainty;

        obj.store.angular_velocity(mission.storage.k_storage_attitude,:) =
obj.angular_velocity; % [rad/sec]

        obj.store.angular_velocity_uncertainty(mission.storage.k_storage_attitude,:)
= obj.angular_velocity_uncertainty;

```

```

obj.store.dot_angular_velocity(mission.storage.k_storage_attitude,:) =
obj.dot_angular_velocity; % [rad/sec^2]

obj.store.dot_angular_velocity_uncertainty(mission.storage.k_storage_attitude,:)
= obj.dot_angular_velocity_uncertainty;
    end

end

```

[] Methods: Main

Main Function

```

function obj = func_main_software_SC_estimate_attitude(obj, mission,
i_SC)

    if (obj.flag_executive == 1)

        switch obj.mode_software_SC_estimate_attitude_selector

            case 'Truth'
                obj =
func_update_software_SC_estimate_attitude_Truth(obj, mission, i_SC);

            case 'KF'
                obj =
func_update_software_SC_estimate_attitude_KF(obj, mission, i_SC);

            otherwise
                error('Attitude Estimation mode not defined!')
            end

            % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

        end

        % Update Storage
obj = func_update_software_SC_estimate_attitude_store(obj,
mission);

        % Reset Variables
if abs(mission.true_time.time - mission.true_time.time_attitude)
<= 1e-6
    obj.flag_executive = 0;
else
    % DONOT SWITCH OFF FUNCTIONS USING flag_executive INSIDE
Attitude Dynamics Loop (ADL)
end

```

end

[] Methods: Truth Estimate Attitude

Use Truth Data

```
function obj = func_update_software_SC_estimate_attitude_Truth(obj,
mission, i_SC)

    obj.instantaneous_data_generated_per_sample = (1e-3)*16*20; % [kb]
    i.e. 20 values per sample, each of 16-bit depth

    obj.attitude = mission.true_SC{i_SC}.true_SC_adc.attitude; %
    [quaternion]
    obj.attitude_uncertainty = zeros(1,4);

    obj.angular_velocity =
    mission.true_SC{i_SC}.true_SC_adc.angular_velocity; % [rad/sec]
    obj.angular_velocity_uncertainty = zeros(1,3);

    obj.dot_angular_velocity =
    mission.true_SC{i_SC}.true_SC_adc.dot_angular_velocity; % [rad/sec^2]
    obj.dot_angular_velocity_uncertainty = zeros(1,3);

end
```

[] Methods: Estimate Attitude using Kalman Filter (KF)

Use KF

```
function obj = func_update_software_SC_estimate_attitude_KF(obj,
mission, i_SC)

    obj.instantaneous_data_generated_per_sample = (1e-3)*16*20; % [kb]
    i.e. 20 values per sample, each of 16-bit depth

    % Take in measurements from SS, ST, IMU

    % Perform KF

    % Output Data
```

```
        obj.attitude = mission.true_SC{i_SC}.true_SC_adc.attitude; %  
[quaternion]  
        obj.attitude_uncertainty = zeros(1,4);  
  
        obj.angular_velocity =  
mission.true_SC{i_SC}.true_SC_adc.angular_velocity; % [rad/sec]  
        obj.angular_velocity_uncertainty = zeros(1,3);  
  
        obj.dot_angular_velocity =  
mission.true_SC{i_SC}.true_SC_adc.dot_angular_velocity; % [rad/sec^2]  
        obj.dot_angular_velocity_uncertainty = zeros(1,3);  
  
    end  
  
end  
  
end
```

Published with MATLAB® R2022a

5.6 Software_SC_Estimate_Orbit

Table of Contents

Class: Software_SC_Estimate_Orbit	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	4
[] Methods: Main	5
[] Methods: Estimate Orbit Truth	6
[] Methods: Estimate Orbit Truth With Error Growth	7

Class: Software_SC_Estimate_Orbit

Estimates the Orbits of the Spacecraft and Target

```
classdef Software_SC_Estimate_Orbit < handle
```

Properties

properties

[] Properties: Initialized Variables

```
instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

mode_software_SC_estimate_orbit_selector % [string] Different attitude
dynamics modes
% - 'Truth' : Use True values
% - 'Truth with Noise' : Use True values + Noise
% - 'TruthWithErrorGrowth' : Use True values with error growth when
target is not visible

compute_wait_time % [sec]
```

[] Properties: Variables Computed Internally

```
name % [string] = SC j for jth SC + SW Estimate Orbit

flag_executive % [Boolean] Executive has told this sensor/actuator to
do its job

position % [km] : Current position of SC in inertial frame I
position_uncertainty % [km]
```

```

velocity % [km/sec] : Current velocity of SC in inertial frame I
velocity_uncertainty % [km/sec]

position_relative_target % [km] : Current position of SC relative to
SB-center J2000 inertial frame
position_relative_target_uncertainty % [km]

velocity_relative_target % [km/sec] : Current velocity of SC relative
to SB-center J2000 inertial frame
velocity_relative_target_uncertainty % [km/sec]

name_relative_target % [string] : Name of the target, relative to
which position and velocity are specified
index_relative_target % [integer] : Index of the target, relative to
which position and velocity are specified

position_target % [km] Current position of Target wrt Sun-centered
J2000
position_target_uncertainty % [km]

velocity_target % [km/sec] Current velocity of Target wrt Sun-centered
J2000
velocity_target_uncertainty % [km/sec]

compute_time % [sec] SC time when this measurement was taken

data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = Software_SC_Estimate_Orbit(init_data, mission, i_SC)

    obj.name = [mission.true_SC{i_SC}.true_SC_body.name, ' SW Estimate
Orbit']; % [string]
    obj.flag_executive = 1;

    obj.instantaneous_data_generated_per_sample = 0; % [kb]
    obj.mode_software_SC_estimate_orbit_selector =
init_data.mode_software_SC_estimate_orbit_selector; % [string]

```

```

        obj.name_relative_target =
mission.true_SC{i_SC}.true_SC_navigation.name_relative_target; % [string]
        obj.index_relative_target =
mission.true_SC{i_SC}.true_SC_navigation.index_relative_target; % [integer]

        if isfield(init_data, 'data')
            obj.data = init_data.data;
        else
            obj.data = [];
        end

        % Initialize data fields for error growth mode
        if
strcmp(obj.mode_software_SC_estimate_orbit_selector, 'TruthWithErrorGrowth')
            obj.data.last_target_visible_time = -inf; % [sec] Initialize
to a value that ensures we start with uncertainty
            obj.data.position_error_growth_rate = 0.01; % [km/sec] Rate at
which position uncertainty grows
            obj.data.velocity_error_growth_rate = 0.001; % [km/sec^2] Rate
at which velocity uncertainty grows
        end

        if isfield(init_data, 'compute_wait_time')
            obj.compute_wait_time = init_data.compute_wait_time; % [sec]
        else
            obj.compute_wait_time = 0; % [sec]
        end

        obj.compute_time = -inf; % [sec]

        obj = func_update_software_SC_estimate_orbit_Truth(obj, mission,
i_SC);

        % Initialize Variables to store: position velocity of SC and
Target
        obj.store = [];

        obj.store.position = zeros(mission.storage.num_storage_steps,
length(obj.position));
        obj.store.position_uncertainty =
zeros(mission.storage.num_storage_steps, length(obj.position_uncertainty));

        obj.store.velocity = zeros(mission.storage.num_storage_steps,
length(obj.velocity));
        obj.store.velocity_uncertainty =
zeros(mission.storage.num_storage_steps, length(obj.velocity_uncertainty));

        obj.store.position_relative_target =
zeros(mission.storage.num_storage_steps, length(obj.position));
        obj.store.position_relative_target_uncertainty =
zeros(mission.storage.num_storage_steps,
length(obj.position_relative_target_uncertainty));

```

```

        obj.store.velocity_relative_target =
zeros(mission.storage.num_storage_steps, length(obj.velocity));
        obj.store.velocity_relative_target_uncertainty =
zeros(mission.storage.num_storage_steps,
length(obj.velocity_relative_target_uncertainty));

        obj.store.position_target =
zeros(mission.storage.num_storage_steps, length(obj.position_target));
        obj.store.position_target_uncertainty =
zeros(mission.storage.num_storage_steps,
length(obj.position_target_uncertainty));

        obj.store.velocity_target =
zeros(mission.storage.num_storage_steps, length(obj.velocity_target));
        obj.store.velocity_target_uncertainty =
zeros(mission.storage.num_storage_steps,
length(obj.velocity_target_uncertainty));

        % Update Storage
        obj = func_update_software_SC_estimate_orbit_store(obj, mission);

        % Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_software_SC_estimate_orbit_store(obj,
mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.position(mission.storage.k_storage,:) =
obj.position; % [km]
        obj.store.position_uncertainty(mission.storage.k_storage,:) =
obj.position_uncertainty;

        obj.store.velocity(mission.storage.k_storage,:) =
obj.velocity; % [km/sec]
        obj.store.velocity_uncertainty(mission.storage.k_storage,:) =
obj.velocity_uncertainty;

obj.store.position_relative_target(mission.storage.k_storage,:) =
obj.position_relative_target; % [km]

obj.store.position_relative_target_uncertainty(mission.storage.k_storage,:) =
obj.position_relative_target_uncertainty;

```

```

obj.store.velocity_relative_target(mission.storage.k_storage,:) =
obj.velocity_relative_target; % [km/sec]

obj.store.velocity_relative_target_uncertainty(mission.storage.k_storage,:) =
obj.velocity_relative_target_uncertainty;

                obj.store.position_target(mission.storage.k_storage,:) =
obj.position_target; % [km]

obj.store.position_target_uncertainty(mission.storage.k_storage,:) =
obj.position_target_uncertainty;

                obj.store.velocity_target(mission.storage.k_storage,:) =
obj.velocity_target; % [km/sec]

obj.store.velocity_target_uncertainty(mission.storage.k_storage,:) =
obj.velocity_target_uncertainty;

                end

        end

```

[] Methods: Main

Main Function

```

function obj = func_main_software_SC_estimate_orbit(obj, mission,
i_SC)

    if (obj.flag_executive == 1)

        switch obj.mode_software_SC_estimate_orbit_selector

            case 'Truth'
                obj =
func_update_software_SC_estimate_orbit_Truth(obj, mission, i_SC);

            case 'TruthWithErrorGrowth'
                obj =
func_update_software_SC_estimate_orbit_TruthWithErrorGrowth(obj, mission,
i_SC);

            otherwise
                error('Orbit Estimation mode not defined!')
            end

            % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

        end

```

```

    % Update Storage
    obj = func_update_software_SC_estimate_orbit_store(obj, mission);

    % Reset Variables
    obj.flag_executive = 0;

end

```

[] Methods: Estimate Orbit Truth

Main Function

```

function obj = func_update_software_SC_estimate_orbit_Truth(obj,
mission, i_SC)

    % Update compute time
    obj.compute_time =
mission.true_SC{i_SC}.software_SC_executive.time;

    obj.instantaneous_data_generated_per_sample = (1e-3)*8*36; % [kb]
i.e. 36 Bytes per sample

    obj.position =
mission.true_SC{i_SC}.true_SC_navigation.position; % [km]
    obj.position_uncertainty = zeros(1,3);

    obj.velocity =
mission.true_SC{i_SC}.true_SC_navigation.velocity; % [km/sec]
    obj.velocity_uncertainty = zeros(1,3);

    obj.position_relative_target =
mission.true_SC{i_SC}.true_SC_navigation.position_relative_target; % [km]
    obj.position_relative_target_uncertainty = zeros(1,3);

    obj.velocity_relative_target =
mission.true_SC{i_SC}.true_SC_navigation.velocity_relative_target; % [km/sec]
    obj.velocity_relative_target_uncertainty = zeros(1,3);

    obj.position_target =
mission.true_target{obj.index_relative_target}.position; % [km]
    obj.position_target_uncertainty = zeros(1,3);

    obj.velocity_target =
mission.true_target{obj.index_relative_target}.velocity; % [km/sec]
    obj.velocity_target_uncertainty = zeros(1,3);

end

```

[] Methods: Estimate Orbit Truth With Error Growth

Realistic model with error growth when camera doesn't have target in view

```
function obj =  
func_update_software_SC_estimate_orbit_TruthWithErrorGrowth(obj, mission,  
i_SC)  
    % Update compute time  
    obj.compute_time =  
mission.true_SC{i_SC}.software_SC_executive.time;  
  
    obj.instantaneous_data_generated_per_sample = (1e-3)*8*36; % [kb]  
    i.e. 36 Bytes per sample  
  
    % Get the basic true values first  
    obj.position =  
mission.true_SC{i_SC}.true_SC_navigation.position; % [km]  
    obj.velocity =  
mission.true_SC{i_SC}.true_SC_navigation.velocity; % [km/sec]  
    obj.position_relative_target =  
mission.true_SC{i_SC}.true_SC_navigation.position_relative_target; % [km]  
    obj.velocity_relative_target =  
mission.true_SC{i_SC}.true_SC_navigation.velocity_relative_target; % [km/sec]  
    obj.position_target =  
mission.true_target{obj.index_relative_target}.position; % [km]  
    obj.velocity_target =  
mission.true_target{obj.index_relative_target}.velocity; % [km/sec]  
  
    % Check if any camera has the target in view  
    target_is_visible = false;  
    for i_HW =  
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_camera  
        if  
mission.true_SC{i_SC}.true_SC_camera{i_HW}.flag_target_visible == 1  
            target_is_visible = true;  
            break;  
        end  
    end  
  
    % Update the uncertainty based on target visibility  
    if target_is_visible  
        % Reset uncertainties when target becomes visible  
        obj.data.last_target_visible_time = obj.compute_time;  
  
        % Set low base uncertainty when target is visible  
        obj.position_uncertainty = [0.01, 0.01, 0.01]; % [km]  
        obj.velocity_uncertainty = [0.001, 0.001, 0.001]; % [km/sec]  
        obj.position_relative_target_uncertainty = [0.01, 0.01,  
0.01]; % [km]  
        obj.velocity_relative_target_uncertainty = [0.001, 0.001,  
0.001]; % [km/sec]
```

```

        obj.position_target_uncertainty = [0.01, 0.01, 0.01]; % [km]
        obj.velocity_target_uncertainty = [0.001, 0.001, 0.001]; %
[km/sec]
    else
        % Calculate time since target was last visible
        time_since_target_visible = obj.compute_time -
obj.data.last_target_visible_time; % [sec]

        % Ensure time is positive (handles initial case)
        if time_since_target_visible < 0
            time_since_target_visible = 3600; % Default to 1 hour if
no prior visibility
        end

        % Calculate position uncertainty growth based on time since
last visible
        base_position_uncertainty = 0.01; % [km] Base uncertainty when
target is visible
        position_uncertainty_growth =
obj.data.position_error_growth_rate * time_since_target_visible; % [km]

        % Calculate velocity uncertainty growth
        base_velocity_uncertainty = 0.001; % [km/sec] Base uncertainty
when target is visible
        velocity_uncertainty_growth =
obj.data.velocity_error_growth_rate * time_since_target_visible; % [km/sec]

        % Apply uncertainty to all position and velocity components
        obj.position_uncertainty = [1, 1, 1] *
(base_position_uncertainty + position_uncertainty_growth);
        obj.velocity_uncertainty = [1, 1, 1] *
(base_velocity_uncertainty + velocity_uncertainty_growth);
        obj.position_relative_target_uncertainty = [1, 1, 1] *
(base_position_uncertainty + position_uncertainty_growth);
        obj.velocity_relative_target_uncertainty = [1, 1, 1] *
(base_velocity_uncertainty + velocity_uncertainty_growth);
        obj.position_target_uncertainty = [1, 1, 1] *
(base_position_uncertainty + position_uncertainty_growth * 0.5); % Target
position known better
        obj.velocity_target_uncertainty = [1, 1, 1] *
(base_velocity_uncertainty + velocity_uncertainty_growth * 0.5); % Target
velocity known better
    end
end
end
end
end

```

Published with MATLAB® R2022a

5.7 Software_SC_Executive

Table of Contents

Class: Software_SC_Executive	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	2
Methods	2
[] Methods: Constructor	2
[] Methods: Store	3
[] Methods: Main	4
[] Methods: Find SC Mode Value	5
[] Methods: DART Executive	5
2. Power Check - Prioritize Survival if under 30%	6
3. Periodic Data Transmission (Every 5 Hour)	6
4. Collision Check & Orbit Control	7
5. Execute DeltaV to Ensure Collision	8
6. Default Mode: Camera Pointing	8
Update Mode Value	8

Class: Software_SC_Executive

Tracks the tasks performed by Executive

```
classdef Software_SC_Executive < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
sc_modes % [Cells of strings] All Spacecraft Modes

mode_software_SC_executive_selector % [string] Select which Executive
to run

instantaneous_data_generated_per_sample % [kb] : Data generated per
sample, in kilo bits (kb)

compute_wait_time % [sec]
```

[] Properties: Variables Computed Internally

```
name % [string] = SC j for jth SC + SW Executive

time % [sec] : Current SC time
```

```

date % [sec from J2000] : Current SC date

this_sc_mode % [string] : Current SC mode
this_sc_mode_value % [integer] : Current SC mode

compute_time % [sec] SC time when this measurement was taken
time_SB_visible % [sec]

data % Other useful data

```

[] Properties: Storage Variables

```

store

end

```

Methods

```

methods

```

[] Methods: Constructor

Construct an instance of this class

```

function obj = Software_SC_Executive(init_data, mission, i_SC)

    obj.name = [mission.true_SC{i_SC}.true_SC_body.name, ' SW
Executive']; % [string]
    obj.time = mission.true_time.time; % [sec]
    obj.date = mission.true_time.date; % [sec from J2000]

    obj.sc_modes = init_data.sc_modes;
    obj.mode_software_SC_executive_selector =
init_data.mode_software_SC_executive_selector;
    obj.instantaneous_data_generated_per_sample = 0; % [kb]

    obj.this_sc_mode = obj.sc_modes{1};
    obj.this_sc_mode_value = func_find_this_sc_mode_value(obj,
obj.this_sc_mode);

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    if isfield(init_data, 'compute_wait_time')
        obj.compute_wait_time = init_data.compute_wait_time; % [sec]
    else
        obj.compute_wait_time = 0; % [sec]
    end

```

```

obj.compute_time = -inf; % [sec]

% Initialize Variables to store: this_sc_mode_value time date data
obj.store = [];

obj.store.this_sc_mode_value =
zeros(mission.storage.num_storage_steps, length(obj.this_sc_mode_value));
obj.store.time = zeros(mission.storage.num_storage_steps,
length(obj.time));
obj.store.date = zeros(mission.storage.num_storage_steps,
length(obj.date));
obj.store.sc_modes = obj.sc_modes; % [sec]

% Additional Executive Variables
switch obj.mode_software_SC_executive_selector

    case 'DART'
        obj.func_software_SC_executive_Dart_constructor(mission,
i_SC);

    case 'Nightingale'
        obj =
func_software_SC_executive_Nightingale_constructor(obj, mission, i_SC);

    otherwise
        % Do nothing!
        disp('Using only DART Executive variables!')
end

% Update Storage
obj = func_update_software_SC_executive_store(obj, mission);

% Update SC Data Handling Class

func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end

```

[] Methods: Store

Update the store variable

```

function obj = func_update_software_SC_executive_store(obj, mission)
    if mission.storage.flag_store_this_time_step == 1
        obj.store.this_sc_mode_value(mission.storage.k_storage,:) =
obj.this_sc_mode_value; % [integer]
        obj.store.time(mission.storage.k_storage,:) = obj.time; %
[sec]
        obj.store.date(mission.storage.k_storage,:) = obj.date; %
[sec]

        % Additional Storage Variables

```

```

        switch obj.mode_software_SC_executive_selector
            case 'Nightingale'
                obj =
func_update_software_SC_executive_store_Nightingale(obj, mission);

            otherwise
                % Do nothing!
            end
        end
    end
end

```

[] Methods: Main

Main Function

```

function obj = func_main_software_SC_executive(obj, mission, i_SC)

    % Update Time
    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_clock

mission.true_SC{i_SC}.true_SC_onboard_clock{i_HW}.flag_executive = 1; % Make
sure time is measured
        obj.time =
mission.true_SC{i_SC}.true_SC_onboard_clock{i_HW}.measurement_vector(1); %
[sec]
        obj.date =
mission.true_SC{i_SC}.true_SC_onboard_clock{i_HW}.measurement_vector(2); %
[sec from J2000]
    end

    switch obj.mode_software_SC_executive_selector

        case 'DART'
            obj = func_software_SC_executive_DART(obj, mission, i_SC);
            obj.this_sc_mode_value = func_find_this_sc_mode_value(obj,
obj.this_sc_mode);

        case 'Nightingale'
            obj = func_software_SC_executive_Nightingale(obj, mission,
i_SC);

        otherwise
            error('Executive mode not defined!')
        end

    end

    % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

```

```

    % Update Storage
    obj = func_update_software_SC_executive_store(obj, mission);

end

```

[] Methods: Find SC Mode Value

Finds the value of a given SC mode

```

function val = func_find_this_sc_mode_value(obj, this_sc_mode)
    % Find the index of the given SC mode
    IndexC = strcmp(obj.sc_modes, this_sc_mode);
    val = find(IndexC);

    % Check if the result is empty
    if isempty(val)
        % Raise an error if nothing is found
        error('SC mode "%s" not found in obj.sc_modes.',
this_sc_mode);
    end
end

```

[] Methods: DART Executive

Main Executive Function for DART mission

```

function obj = func_software_SC_executive_DART(obj, mission, i_SC)

    % Executive logic for DART mission

    mission.true_SC{i_SC}.software_SC_estimate_attitude.flag_executive
= 1;
    mission.true_SC{i_SC}.software_SC_control_attitude.flag_executive
= 1;

    %% 1. Check for Power Emergency
    % if mission.true_SC{i_SC}.true_SC_power.power_emergency
    %     warning('POWER EMERGENCY: Power deficit of %.2f W-hr
detected. Entering safe mode.', ...
    %         mission.true_SC{i_SC}.true_SC_power.power_deficit);
    %
    %     % Enter power-saving mode by maximizing solar panel exposure
    %     obj.this_sc_mode = 'Maximize SP Power';
    %
    %     % Disable non-critical subsystems
    %     % - Keep only essential attitude determination and control
active
    %     % - Disable science instruments
    %     for i_HW =
1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_camera
    %
mission.true_SC{i_SC}.true_SC_camera{i_HW}.flag_executive = 0;
    %     end

```

```

        %
        %      % Disable orbit control and planned maneuvers
        %
mission.true_SC{i_SC}.software_SC_control_orbit.flag_executive = 0;
        %
mission.true_SC{i_SC}.software_SC_control_orbit.desired_DeltaV_needs_to_be_executed
= false;
        %      return;
        % end

% Check if this is a new mode transition
if ~strcmp(obj.this_sc_mode, obj.data.previous_mode)
    obj.data.last_mode_change_time = mission.true_time.time;
    disp(['Mode transition: ', obj.data.previous_mode, ' -> ',
obj.this_sc_mode, ' at t=', num2str(mission.true_time.time)]);

    % Store previous mode
    obj.data.previous_mode = obj.this_sc_mode;
end

% Add a small delay to prevent rapid mode switching
% Only allow mode changes after minimum time threshold, to not
% overload the attitude control system
time_since_last_mode_change = mission.true_time.time -
obj.data.last_mode_change_time;
if time_since_last_mode_change < 100 &&
obj.data.last_mode_change_time > 0
    % Skip evaluation of mode changes to avoid rapid oscillation
    % Just maintain the current mode
    return;
end

```

2. Power Check - Prioritize Survival if under 30%

```

if mission.true_SC{i_SC}.software_SC_power.mean_state_of_charge <
30
    obj.this_sc_mode = 'Maximize SP Power';
    return; % Highest priority - exit early
end

```

3. Periodic Data Transmission (Every 5 Hour)

```

time_since_last_comm = mission.true_time.time -
mission.true_SC{i_SC}.software_SC_communication.last_communication_time;

% Check if we're already in communication mode with an active link
already_communicating = strcmp(obj.this_sc_mode, 'DTE Comm') &&
mission.true_SC{i_SC}.true_SC_communication_link{1}.flag_executive == 1;

```

```

        % Check if transmission just completed (to exit comm mode)
        transmission_complete =
isfield(mission.true_SC{i_SC}.software_SC_communication.data, 'transmission_complete')
&& ...

mission.true_SC{i_SC}.software_SC_communication.data.transmission_complete;

        % Start comm if:
        % 1. Time threshold is met (3600 seconds since last communication)
AND memory usage is significant OR
        % 2. We're already communicating AND haven't completed yet
        % 3. Earth is visible AND
        % 4. No orbit maneuver is in progress
        if ((time_since_last_comm >= 3600 * 5) || ...
            (already_communicating && ~transmission_complete)) && ...

(mission.true_SC{i_SC}.true_SC_navigation.flag_visible_Earth) && ...

~mission.true_SC{i_SC}.software_SC_control_orbit.desired_DeltaV_needs_to_be_executed

        obj.this_sc_mode = 'DTE Comm';

        % Make sure data handling is activated to track data storage
mission.true_SC{i_SC}.software_SC_data_handling.flag_executive
= 1;

        % Activate communication software last to ensure proper
sequencing
        mission.true_SC{i_SC}.software_SC_communication.flag_executive
= 1;

        return;
end

```

4. Collision Check & Orbit Control

```

        mission.true_SC{i_SC}.software_SC_control_orbit =
mission.true_SC{i_SC}.software_SC_control_orbit;
        time_since_last_orbit_check = mission.true_time.time -
mission.true_SC{i_SC}.software_SC_control_orbit.last_time_control;

        if (time_since_last_orbit_check >=
mission.true_SC{i_SC}.software_SC_control_orbit.max_time_before_control)
&& ...

~mission.true_SC{i_SC}.software_SC_control_orbit.desired_DeltaV_needs_to_be_executed
&& ...

        ~mission.true_SC{i_SC}.true_SC_navigation.flag_SC_crashed

        % Activate orbit control system to compute necessary
corrections
        mission.true_SC{i_SC}.software_SC_control_orbit.flag_executive
= 1;

```

```

mission.true_SC{i_SC}.software_SC_estimate_orbit.flag_executive = 1;
    end

```

5. Execute DeltaV to Ensure Collision

```

    if
mission.true_SC{i_SC}.software_SC_control_orbit.desired_DeltaV_needs_to_be_executed
&& ...

mission.true_SC{i_SC}.software_SC_control_orbit.desired_DeltaV_computed
&& ...
        ~mission.true_SC{i_SC}.true_SC_navigation.flag_SC_crashed

        % Keep turning the spacecraft to the right orientation
        obj.this_sc_mode = 'Point Thruster along DeltaV direction';

        % Make sure orbit control stays active
        mission.true_SC{i_SC}.software_SC_control_orbit.flag_executive
= 1;

mission.true_SC{i_SC}.software_SC_estimate_orbit.flag_executive = 1;

        return;
    end

```

6. Default Mode: Camera Pointing

```

        obj.this_sc_mode = 'Point camera to Target';
        mission.true_SC{i_SC}.true_SC_camera{1}.flag_executive = 1; %
Activate primary camera
        mission.true_SC{i_SC}.software_SC_estimate_orbit.flag_executive =
1; % Regular updates

```

Update Mode Value

```

    end

    function func_software_SC_executive_Dart_constructor(obj,
mission,i_SC)
        % Add mode transition management
        if ~isfield(obj.data, 'last_mode_change_time')
            obj.data.last_mode_change_time = 0;
            obj.data.previous_mode = obj.this_sc_mode;
        end
    end

end

end

end

```

Published with MATLAB® R2022a

5.8 Software_SC_Power

Table of Contents

Class: Software_SC_Power	1
Properties	1
[] Properties: Initialized Variables	1
[] Properties: Variables Computed Internally	1
[] Properties: Storage Variables	1
Methods	2
[] Methods: Constructor	2
[] Methods: Store	2
[] Methods: Main	3
[] Methods: Update Mean SoC	4

Class: Software_SC_Power

Track the power status onboard the spacecraft

```
classdef Software_SC_Power < handle
```

Properties

```
properties
```

[] Properties: Initialized Variables

```
instantaneous_data_generated_per_sample % [kb] : Data generated per  
sample, in kilo bits (kb)
```

```
mode_software_SC_power_selector % [string] Different power modes  
% - 'Generic'
```

[] Properties: Variables Computed Internally

```
name % [string] = SC j for jth SC + SW Power
```

```
flag_executive % [Boolean] Executive has told this sensor/actuator to  
do its job
```

```
mean_state_of_charge % [percentage] : Mean SoC is defined by = 100×  
Sum (instantaneous_capacity) / Sum(maximum_capacity)
```

```
data % Other useful data
```

[] Properties: Storage Variables

```
store
```

end

Methods

methods

[] Methods: Constructor

Construct an instance of this class

```
function obj = Software_SC_Power(init_data, mission, i_SC)

    obj.name = [mission.true_SC{i_SC}.true_SC_body.name, ' SW
Power']; % [string]
    obj.flag_executive = 0;

    obj.instantaneous_data_generated_per_sample =
init_data.instantaneous_data_generated_per_sample; % [kb]

    obj.mode_software_SC_power_selector =
init_data.mode_software_SC_power_selector;

    if isfield(init_data, 'data')
        obj.data = init_data.data;
    else
        obj.data = [];
    end

    % Update Mean SoC
    obj = func_update_mean_state_of_charge(obj, mission, i_SC);

    % Initialize Variables to store
    obj.store = [];

    obj.store.flag_executive =
zeros(mission.storage.num_storage_steps, length(obj.flag_executive));
    obj.store.mean_state_of_charge =
zeros(mission.storage.num_storage_steps, length(obj.mean_state_of_charge));

    % Update Storage
    obj = func_update_software_SC_power_store(obj, mission);

    % Update SC Data Handling Class

    func_initialize_list_HW_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

end
```

[] Methods: Store

Update the store variables

```

function obj = func_update_software_SC_power_store(obj, mission)

    if mission.storage.flag_store_this_time_step == 1
        obj.store.flag_executive(mission.storage.k_storage,:) =
obj.flag_executive; % [Boolean]
        obj.store.mean_state_of_charge(mission.storage.k_storage,:) =
obj.mean_state_of_charge; % [percentage]
    end

end

```

[] Methods: Main

Main Function

```

function obj = func_main_software_SC_power(obj, mission, i_SC)

    if (obj.flag_executive == 1)

        switch obj.mode_software_SC_power_selector

            case 'Generic'

                % Update Mean SoC
                obj = func_update_mean_state_of_charge(obj, mission,
i_SC);

            case 'Nightingale'
                % obj =
func_update_software_SC_power_Nightingale(obj,mission,i_SC);

                % Update Mean SoC
                obj = func_update_mean_state_of_charge(obj, mission,
i_SC);

            otherwise
                error('Power mode not defined!')
            end

            % Update Data Generated

func_update_instantaneous_data_generated(mission.true_SC{i_SC}.true_SC_data_handling,
obj, mission);

        end

        % Update Storage
        obj = func_update_software_SC_power_store(obj, mission);

        % Reset Variables
        obj.flag_executive = 0;

    end

```

[] Methods: Update Mean SoC

Updates mean_state_of_charge

```
function obj = func_update_mean_state_of_charge(obj, mission, i_SC)

    total_instantaneous_capacity = 0; % [Watts * hr]
    total_maximum_capacity = 0; % [Watts * hr]

    for i_batt =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_battery

        total_instantaneous_capacity = total_instantaneous_capacity
+ mission.true_SC{i_SC}.true_SC_battery{i_batt}.instantaneous_capacity; %
[Watts * hr]
        total_maximum_capacity = total_maximum_capacity +
mission.true_SC{i_SC}.true_SC_battery{i_batt}.maximum_capacity; % [Watts *
hr]

    end

    obj.mean_state_of_charge = 100 * total_instantaneous_capacity /
total_maximum_capacity; % [percentage]

end

end

end
```

Published with MATLAB® R2022a

Chapter 6

Main File

6.1 main_v3

Table of Contents

main_v3	2
Time Loop	2
Update Time, Date	2
Update Storage	2
Update Solar System	2
Update Target Position Velocity	2
For Each Spacecraft	2
Update Solar Radiation Pressure	2
Update Gravity Gradient	2
[] Update SC Body	3
[] Update SC Position Velocity	3
[] Update SC Onboard Clock	3
[] Update SC Onboard Computer	3
[] Update Software SC Executive	3
[] Update SC Camera	3
Attitude Dynamics Loop (ADL)	3
[] Update Time in ADL	4
[] Update Storage in ADL	4
[] For Each Spacecraft	4
[] [] Update SC Attitude in ADL	4
[] [] Update SC Sun Sensor in ADL	4
[] [] Update SC Star Tracker in ADL	4
[] [] Update SC IMU in ADL	4
[] [] Update Software SC Attitude Estimation in ADL	5
[] [] Update Software SC Attitude Control	5
[] Update Micro Thrusters	5
[] Update Reaction wheels	5
For Each Spacecraft	5
[] Update SC Science Radar	5
[] Update SC Science Processor	6
[] Update Software SC Communication	6
[] Update SC Communication Link	6
[] Update SC Radio Antenna	6
[] Update SC Generic Sensor	6
[] Update SC Solar Panels	6
[] Update Software SC Orbit Estimation in ADL	7
[] Update Software SC Orbit Control in ADL	7
[] Update Chemical Thruster	7
[] Update Fuel Tanks	7
[] Update SC Battery	7
[] Update Software SC Power	7
[] Update SC Data Handling	7
[] Update SC Onboard Memory	8
[] Update Software SC Data Handling	8
[] Update SC Power	8
Update Ground Stations's Radio Antenna	8
Update Ground Station	8
Fix Warnings	8
TEST: Force spacecraft rotation for visualization testing	8

Update real-time visualization	9
Stop Sim	9
Close all SPICE files	9

main_v3

This is the main time loop that runs everything!

tic

Time Loop

```
disp('Starting Main Time Loop')
for k = 1:1:mission.true_time.num_time_steps
```

Update Time, Date

```
func_update_true_time_date(mission.true_time, k);
```

Update Storage

```
func_update_storage_flag(mission.storage, mission);
func_update_time_store(mission.true_time, mission);
```

Update Solar System

```
func_main_true_solar_system(mission.true_solar_system, mission);
```

Update Target Position Velocity

```
for i_target = 1:1:mission.num_target
    func_main_true_target(mission.true_target{i_target}, mission);
end
```

For Each Spacecraft

```
for i_SC = 1:1:mission.num_SC
```

Update Solar Radiation Pressure

```
func_main_true_SRP(mission.true_SC{i_SC}.true_SRP, mission, i_SC);
```

Update Gravity Gradient

```
func_update_disurbance_torque_G2(mission.true_SC{i_SC}.true_gravity_gradient,
mission, i_SC);
```

[] Update SC Body

```
func_main_true_SC_body(mission.true_SC{i_SC}.true_SC_body, mission,
i_SC);
```

[] Update SC Position Velocity

```
func_main_true_SC_navigation(mission.true_SC{i_SC}.true_SC_navigation,
mission, i_SC);
```

[] Update SC Onbaord Clock

```
for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_clock

func_main_true_SC_onboard_clock(mission.true_SC{i_SC}.true_SC_onboard_clock{i_HW},
mission, i_SC);
end
```

[] Update SC Onboard Computer

```
for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_computer

func_main_true_SC_onboard_computer(mission.true_SC{i_SC}.true_SC_onboard_computer{i_HW},
mission, i_SC);
end
```

[] Update Software SC Executive

```
func_main_software_SC_executive(mission.true_SC{i_SC}.software_SC_executive,
mission, i_SC);
```

[] Update SC Camera

```
for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_camera

func_main_true_SC_camera(mission.true_SC{i_SC}.true_SC_camera{i_HW}, mission,
i_SC, i_HW);
end

end
```

Attitude Dynamics Loop (ADL)

```
for k_attitude = 1:1:mission.true_time.num_time_steps_attitude
```

[] Update Time in ADL

```
func_update_true_time_attitude(mission.true_time, k_attitude);
```

[] Update Storage in ADL

```
func_update_storage_flag_attitude(mission.storage, mission);  
func_update_time_store_attitude(mission.true_time, mission);
```

[] For Each Spacecraft

```
for i_SC = 1:1:mission.num_SC
```

[] [] Update SC Attitude in ADL

```
    func_main_true_SC_attitude(mission.true_SC{i_SC}.true_SC_adc,  
mission, i_SC);
```

[] [] Update SC Sun Sensor in ADL

```
        for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_sun_sensor  
  
func_main_true_SC_sun_sensor(mission.true_SC{i_SC}.true_SC_sun_sensor{i_HW},  
mission, i_SC);  
        end
```

[] [] Update SC Star Tracker in ADL

```
        for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_star_tracker  
  
func_main_true_SC_star_tracker(mission.true_SC{i_SC}.true_SC_star_tracker{i_HW},  
mission, i_SC);  
        end
```

[] [] Update SC IMU in ADL

```
        for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_imu  
            func_main_true_SC_imu(mission.true_SC{i_SC}.true_SC_imu{i_HW},  
mission, i_SC);  
        end
```

[] [] Update Software SC Attitude Estimation in ADL

```
func_main_software_SC_estimate_attitude(mission.true_SC{i_SC}.software_SC_estimate_attitude, mission, i_SC);
```

[] [] Update Software SC Attitude Control

```
func_main_software_SC_control_attitude(mission.true_SC{i_SC}.software_SC_control_attitude, mission, i_SC);
```

[] Update Micro Thrusters

```
        for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster  
  
func_main_true_SC_micro_thruster(mission.true_SC{i_SC}.true_SC_micro_thruster{i_HW},  
mission, i_SC, i_HW);  
        end
```

[] Update Reaction wheels

```
        for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel  
  
func_main_true_reaction_wheel(mission.true_SC{i_SC}.true_SC_reaction_wheel{i_HW},  
mission, i_SC, i_HW);  
        end  
  
    end  
  
end
```

For Each Spacecraft

```
    for i_SC = 1:1:mission.num_SC
```

[] Update SC Science Radar

```
        for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_science_radar  
  
func_main_true_SC_science_radar(mission.true_SC{i_SC}.true_SC_science_radar{i_HW},  
mission, i_SC, i_HW);  
        end
```

[] Update SC Science Processor

```
    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_science_processor

func_main_true_SC_science_processor(mission.true_SC{i_SC}.true_SC_science_processor{i_HW},
mission, i_SC);
    end
```

[] Update Software SC Communication

```
func_main_software_SC_communication(mission.true_SC{i_SC}.software_SC_communication,
mission, i_SC);
```

[] Update SC Communication Link

```
    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_communication_link

func_main_true_SC_communication_link(mission.true_SC{i_SC}.true_SC_communication_link{i_HW},
mission, i_SC);
    end
```

[] Update SC Radio Antenna

```
    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_radio_antenna

func_main_true_SC_radio_antenna(mission.true_SC{i_SC}.true_SC_radio_antenna{i_HW},
mission, i_SC);
    end
```

[] Update SC Generic Sensor

```
    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_generic_sensor

func_main_true_SC_generic_sensor(mission.true_SC{i_SC}.true_SC_generic_sensor{i_HW},
mission, i_SC);
    end
```

[] Update SC Solar Panels

```
    for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_solar_panel

func_main_true_SC_solar_panel(mission.true_SC{i_SC}.true_SC_solar_panel{i_HW},
mission, i_SC);
    end
```

[] Update Software SC Orbit Estimation in ADL

```
func_main_software_SC_estimate_orbit(mission.true_SC{i_SC}.software_SC_estimate_orbit,  
mission, i_SC);
```

[] Update Software SC Orbit Control in ADL

```
func_main_software_SC_control_orbit(mission.true_SC{i_SC}.software_SC_control_orbit,  
mission, i_SC);
```

[] Update Chemical Thruster

```
    for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster  
  
func_main_true_chemical_thruster(mission.true_SC{i_SC}.true_SC_chemical_thruster,  
mission, i_SC, i_HW);  
    end
```

[] Update Fuel Tanks

```
    for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank  
  
func_main_true_SC_fuel_tank(mission.true_SC{i_SC}.true_SC_fuel_tank{i_HW},  
mission, i_SC);  
    end
```

[] Update SC Battery

```
    for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_battery  
  
func_main_true_SC_battery(mission.true_SC{i_SC}.true_SC_battery{i_HW},  
mission, i_SC);  
    end
```

[] Update Software SC Power

```
func_main_software_SC_power(mission.true_SC{i_SC}.software_SC_power,  
mission, i_SC);
```

[] Update SC Data Handling

```
func_main_true_SC_data_handling(mission.true_SC{i_SC}.true_SC_data_handling,  
mission, i_SC);
```

[] Update SC Onboard Memory

```
for i_HW =  
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_memory  
  
func_main_true_SC_onboard_memory(mission.true_SC{i_SC}.true_SC_onboard_memory{i_HW},  
mission, i_SC);  
end
```

[] Update Software SC Data Handling

```
func_main_software_SC_data_handling(mission.true_SC{i_SC}.software_SC_data_handling,  
mission, i_SC);
```

[] Update SC Power

```
func_main_true_SC_power(mission.true_SC{i_SC}.true_SC_power, mission,  
i_SC);  
  
end
```

Update Ground Stations's Radio Antenna

```
for i_HW = 1:1:mission.true_ground_station.num_GS_radio_antenna  
    func_main_true_GS_radio_antenna(mission.true_GS_radio_antenna{i_HW},  
mission);  
end
```

Update Ground Station

```
func_main_true_ground_station(mission.true_ground_station, mission, i_SC);
```

Fix Warnings

```
w = warning('query', 'last');  
if ~isempty(w)  
    warning('off', w.identifier);  
end
```

TEST: Force spacecraft rotation for visualization testing

This section is just for testing the visualization - remove once confirmed working

```
if isfield(mission.true_SC{i_SC}, 'true_SC_adc') &&  
isfield(mission.true_SC{i_SC}.true_SC_adc, 'rotation_matrix')  
    % Create a small rotation around Z axis
```

```

    angle_increment = 0.01; % Small angle in radians
    cos_ang = cos(angle_increment);
    sin_ang = sin(angle_increment);
    rot_z = [cos_ang, -sin_ang, 0;
             sin_ang, cos_ang, 0;
             0, 0, 1];

    % Apply small rotation to current rotation matrix
    mission.true_SC{i_SC}.true_SC_adc.rotation_matrix = rot_z *
mission.true_SC{i_SC}.true_SC_adc.rotation_matrix;

    % Update quaternion if it exists
    if isfield(mission.true_SC{i_SC}.true_SC_adc, 'quaternion')
        % Convert rotation matrix to quaternion
        % Simple conversion for this test
        mission.true_SC{i_SC}.true_SC_adc.quaternion =
mission.true_SC{i_SC}.true_SC_adc.quaternion + 0.01 * [0, 0,
sin(angle_increment/2), cos(angle_increment/2)];
        mission.true_SC{i_SC}.true_SC_adc.quaternion
= mission.true_SC{i_SC}.true_SC_adc.quaternion /
norm(mission.true_SC{i_SC}.true_SC_adc.quaternion);
    end
end

```

Update real-time visualization

```
func_update_realtime_plot(mission.storage, mission);
```

Stop Sim

```

    if mission.storage.flag_stop_sim == 1
        disp('Stopping Sim!')
        break
    end
end

```

Close all SPICE files

```
cspice_kclear
```

Published with MATLAB® R2022a

Chapter 7

Mission Classes

7.1 Mission_DART

Table of Contents

DART Mission	1
Mission Definition	2
Time Configuration	2
Storage Configuration	2
Star Catalog Configuration	3
Solar System Configuration	3
Target Body Configuration	3
Ground Station Configuration	3
Ground Station Radio Antenna Configuration	3
Spacecraft Initialization	4
Spacecraft Body Configuration	4
Initialize First Spacecraft's Position and Velocity	5
Initialize First Spacecraft's Attitude	6
Initialize First Spacecraft's Power	6
Initialize First Spacecraft's Data	6
Initialize First Spacecraft's Radio Antenna	6
Spacecraft Fuel Tank Configuration	7
Initialize First Spacecraft's Solar Panels	8
Initialize First Spacecraft's Battery	9
Initialize First Spacecraft's Onboard Memory	10
Initialize First Spacecraft's Onboard Clock	10
Initialize First Spacecraft's Cameras	10
Initialize First Spacecraft's Sun Sensors	11
Initialize First Spacecraft's Star Tracker	12
Initialize First Spacecraft's IMU	13
Initialize First Spacecraft's Micro Thrusters	13
Initialize the Reaction Wheels	15
Chemical Thruster Configuration	16
Onboard Computer Configuration	17
Spacecraft Communication Links Configuration	17
Initialize Solar Radiation Pressure	18
Initialize Gravity Gradient for Earth	18
Spacecraft Software: Executive Configuration	18
Spacecraft Software: Attitude Estimation Configuration	19
Spacecraft Software: Orbit Estimation Configuration	19
Spacecraft Software: Orbit Control Configuration	19
Spacecraft Software: Attitude Control Configuration	19
Spacecraft Software: Communication Configuration	19
Spacecraft Software: Power Management Configuration	20
Spacecraft Software: Data Handling Configuration	20
Final Things to Do Before Running the Simulation	20
Save All Data	21
Execute Main File	21
Save All Data	21
Plots	21

DART Mission

Initialization File for DART (Double Asteroid Redirection Test) mission simulation

```

% Clear workspace
clear
close all
clc

% Change workspace folder to this file location
mfile_name      = mfilename('fullpath');
[pathstr,name,ext] = fileparts(mfile_name);
cd(pathstr);
clear mfile_name pathstr name ext

% Add required paths
addpath(genpath('.././MuSCAT_Supporting_Files'))
addpath(genpath('.././'))

```

Mission Definition

```

mission = [];
mission.name = 'DART';           % Name of the Mission
mission.num_SC = 1;             % Number of Spacecraft
mission.num_target = 1;         % Number of Target bodies
mission.frame = 'Absolute';      % Frame type: 'Absolute', 'Relative', or
    'Combined'
mission.flag_stop_sim = 0;       % Boolean flag to stop simulation if needed

```

Time Configuration

```

init_data = [];
init_data.t_initial = 0;         % [sec] Initial
    time
init_data.t_final = 2000; %      % [sec] Final time
init_data.time_step = 5;         % [sec] Simulation
    time step
init_data.t_initial_date_string = '02-NOV-2018 00:00:00'; % Format = [DD-
MMM-YYYY HH:MM:SS]
init_data.time_step_attitude = 0.1; % [sec] Time step
    for attitude dynamics
mission.true_time = True_Time(init_data);

```

Storage Configuration

```

init_data = [];
init_data.time_step_storage = 1;
init_data.time_step_storage_attitude = 0.5;
init_data.flag_visualize_SC_attitude_orbit_during_sim = 0; % Don't show
    attitude during sim
init_data.flag_realtime_plotting = 0; % [Boolean] Show mission data and
    attitude during sim
init_data.flag_save_plots = 1; % [Boolean] 1: Save them (takes
    little time), 0: Doesnt save them
init_data.flag_save_video = 0; % [Boolean] 1: Save them (takes
    more time), 0: Doesnt save them

```

```
mission.storage = Storage(init_data, mission);

% Set font size for plots
mission.storage.plot_parameters.standard_font_size = 15;

% Initialize time storage
func_initialize_time_store(mission.true_time, mission);
```

Star Catalog Configuration

```
mission.true_stars = True_Stars(mission);
mission.true_stars.maximum_magnitude = 10; % Maximum star magnitude to
include
```

Solar System Configuration

```
init_data = [];
init_data.SS_body_names = ["Sun", "Earth"]; % Solar system bodies to include
mission.true_solar_system = True_Solar_System(init_data, mission);
```

Target Body Configuration

```
for i_target = 1:1:mission.num_target
    init_data = [];
    init_data.target_name = 'Bennu'; % Target asteroid name
    mission.true_target{i_target} = True_Target_SPICE(init_data, mission);
end
```

Ground Station Configuration

```
init_data = [];
init_data.num_GS_radio_antenna = 1; % Number of ground station
antennas
mission.true_ground_station = True_Ground_Station(init_data, mission);
```

Ground Station Radio Antenna Configuration

```
for i_HW = 1:1:mission.true_ground_station.num_GS_radio_antenna
    init_data = [];
    init_data.antenna_type = 'High Gain';
    init_data.mode_true_GS_radio_antenna_selector = 'RX';

    % Link Margin Calculation Parameters
    init_data.antenna_gain = 90; % [dB]
    init_data.noise_temperature = 100; % [K]
    init_data.beamwidth = 0.1; % [MHz]
    init_data.energy_bit_required = 4.2; % [dB]
    init_data.line_loss = 0; % [dB]
    init_data.coding_gain = 7.3; % [dB]
```

```
    mission.true_GS_radio_antenna{i_HW} = True_GS_Radio_Antenna(init_data,  
mission, i_HW);  
end
```

Spacecraft Initialization

```
for i_SC = 1:1:mission.num_SC  
    mission.true_SC{i_SC} = [];  
end
```

Spacecraft Body Configuration

```
i_SC = 1; % First spacecraft  
  
init_data = [];  
init_data.i_SC = i_SC;  
  
% Body shape model  
init_data.shape_model{1} = [];  
init_data.shape_model{1}.Vertices = [0 0 0;  
    0.3 0 0;  
    0.3 0 0.1;  
    0 0 0.1;  
    0 0.2 0;  
    0.3 0.2 0;  
    0.3 0.2 0.1;  
    0 0.2 0.1]; % [m]  
init_data.shape_model{1}.Faces = [1 2 3;  
    1 4 3;  
    2 3 7;  
    2 6 7;  
    3 4 8;  
    3 7 8;  
    1 4 8;  
    1 5 8;  
    1 2 6;  
    1 5 6;  
    5 6 7;  
    5 8 7];  
  
init_data.shape_model{1}.Face_reflectance_factor =  
    0.6*ones(size(init_data.shape_model{1}.Faces,1),1);  
init_data.shape_model{1}.type = 'cuboid';  
init_data.shape_model{1}.mass = 11; % [kg] Dry mass  
  
% Additional mass components  
init_data.mass.supplement{1}.mass = 0.5; % [kg]  
init_data.mass.supplement{1}.location = [0.1 0 0]; % [m]  
init_data.mass.supplement{1}.MI_over_m = zeros(3,3); % [m^2]  
  
init_data.mass.supplement{2}.mass = 0.5; % [kg]  
init_data.mass.supplement{2}.location = [0 0 0.1]; % [m]  
init_data.mass.supplement{2}.MI_over_m = zeros(3,3); % [m^2]
```

```

init_data.mode_COM_selector = 'update'; % Compute Center of Mass dynamically
init_data.mode_MI_selector = 'update'; % Compute Moment of Inertia
    dynamically

% Initialize hardware configuration
run init_num_hardware_exists
init_data.num_hardware_exists = num_hardware_exists;
clear num_hardware_exists

% Define hardware complement
init_data.num_hardware_exists.num_onboard_clock = 1;
init_data.num_hardware_exists.num_camera = 1;
init_data.num_hardware_exists.num_solar_panel = 3;
init_data.num_hardware_exists.num_battery = 2;
init_data.num_hardware_exists.num_onboard_memory = 2;
init_data.num_hardware_exists.num_sun_sensor = 6;
init_data.num_hardware_exists.num_star_tracker = 3;
init_data.num_hardware_exists.num_imu = 1;
init_data.num_hardware_exists.num_micro_thruster = 12;
init_data.num_hardware_exists.num_chemical_thruster = 1;
init_data.num_hardware_exists.num_reaction_wheel = 3;
init_data.num_hardware_exists.num_communication_link = 2;
init_data.num_hardware_exists.num_radio_antenna = 1;
init_data.num_hardware_exists.num_fuel_tank = 1;
init_data.num_hardware_exists.num_onboard_computer = 2;

mission.true_SC{i_SC}.true_SC_body = True_SC_Body(init_data, mission);

```

Initialize First Spacecraft's Position and Velocity

```

init_data = [];
init_data.spice_filename = '../..../MuSCAT_Supporting_Files/SC_data/DARE/
traj_daresim_simple.bsp'; % [string] : SC's SPICE FileName
cspice_furnsh(init_data.spice_filename)

% bandyopa@MT-319257 exe % ./brief ../..../SC_data/traj_daresim_simple.bsp
%
% BRIEF -- Version 4.0.0, September 8, 2010 -- Toolkit Version N0066
%
%
% Summary for: ../..../SC_data/traj_daresim_simple.bsp
%
% Body: -110
%      Start of Interval (ET)                End of Interval (ET)
%      -----
%      2018 OCT 27 21:36:30.000                2018 NOV 03 21:36:00.000

init_data.spice_name = '-110'; % [string] : SC's SPICE Name

% Sun centered - J2000 frame (inertial)

```

```
init_data.SC_pos_vel =
    cspice_spkezr(init_data.spice_name,mission.true_time.date,'J2000','NONE','SUN');
init_data.position = init_data.SC_pos_vel(1:3)'; % [km]
init_data.velocity = init_data.SC_pos_vel(4:6)'; % [km/sec]

init_data.mode_true_SC_navigation_dynamics_selector = 'Absolute Dynamics';

mission.true_SC{i_SC}.true_SC_navigation = True_SC_Navigation(init_data,
    mission);
```

Initialize First Spacecraft's Attitude

```
init_data = [];
init_data.SC_MRP_init = [0.1 0.2 0.3]; % MRP
init_data.SC_omega_init = [0 0 0.001]; % [rad/sec]

init_data.SC_e_init = init_data.SC_MRP_init/norm(init_data.SC_MRP_init);
init_data.SC_Phi_init = 4*atand(init_data.SC_MRP_init(1)/
    init_data.SC_e_init(1)); % [deg]
init_data.SC_beta_v_init = init_data.SC_e_init *
    sind(init_data.SC_Phi_init/2);
init_data.SC_beta_4_init = cosd(init_data.SC_Phi_init/2);

init_data.attitude = [init_data.SC_beta_v_init, init_data.SC_beta_4_init]; %
    [quaternion]
init_data.attitude = func_quaternion_properize(init_data.attitude); %
    [quaternion] properized
init_data.angular_velocity = init_data.SC_omega_init;

init_data.mode_true_SC_attitude_dynamics_selector = 'Rigid';

mission.true_SC{i_SC}.true_SC_adc = True_SC_ADC(init_data, mission);
```

Initialize First Spacecraft's Power

```
init_data = [];
init_data.power_loss_rate = 0.05; % [float] 5% power loss in distribution and
    conversion
mission.true_SC{i_SC}.true_SC_power = True_SC_Power(init_data, mission);
```

Initialize First Spacecraft's Data

```
init_data = [];
init_data.mode_true_SC_data_handling_selector = 'Generic';
mission.true_SC{i_SC}.true_SC_data_handling = True_SC_Data_Handling(init_data,
    mission);
```

Initialize First Spacecraft's Radio Antenna

```
for i_HW =
    1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_radio_antenna
```

```

init_data = [];
init_data.location = [0 1 0]; % [unit vector] antenna physical axis in
Body frame
init_data.orientation = [0 0 1]; % [unit vector] antenna pointing
direction for nominal gain in Body frame

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Antennas
init_data.antenna_type = "dipole"; % antenna type
init_data.antenna_gain = 28.1; % [dB] Gain of DTE antenna SC
init_data.antenna_axis = [0 1 0]; % [unit vector] antenna physical
axis in Body frame
init_data.antenna_pointing = [0 0 1]; % [unit vector] antenna pointing
direction for nominal gain in Body frame
init_data.antenna_frequency = 8450; % [MHz]
init_data.tx_line_loss = 1; % [dB]
init_data.noise_temperature = 100; % [K]
init_data.energy_bit_required = 4.2; % [dB]
init_data.coding_gain = 7.3; % [dB]
init_data.beamwidth = 0.1; % [MHz]

init_data.maximum_data_rate = 1000; % [kbps]
init_data.base_data_rate_generated = 10; % [kbps]

init_data.TX_power_consumed = 50; % [W] Based on 50 W RF output and 50%
efficiency
init_data.RX_power_consumed = 25; % [W] Typical for spacecraft receivers

mission.true_SC{i_SC}.true_SC_radio_antenna{i_HW} =
True_SC_Radio_Antenna(init_data, mission, i_SC, i_HW);
end

```

Spacecraft Fuel Tank Configuration

```

for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_fuel_tank
init_data = [];

% Basic properties
init_data.name = ['Fuel Tank ', num2str(i_HW)];
init_data.instantaneous_power_consumed = 2.0; % [W] for heaters,
valves, etc.
init_data.instantaneous_data_rate_generated = 0.1; % [kbps] for telemetry

% Fuel properties
init_data.maximum_capacity = 5.0; % [kg] total fuel capacity
init_data.initial_fuel_mass = 5.0; % [kg] initial fuel mass (full tank)
init_data.fuel_density = 1000; % [kg/m^3] typical hydrazine density

% Physical properties
init_data.location = [0.15, 0.1, 0.05]; % [m] tank location in body frame

```

```

% Shape model - simplified cuboid
init_data.shape_model = [];
init_data.shape_model.Vertices = [
    0.1, 0.05, 0.0;
    0.2, 0.05, 0.0;
    0.2, 0.15, 0.0;
    0.1, 0.15, 0.0;
    0.1, 0.05, 0.1;
    0.2, 0.05, 0.1;
    0.2, 0.15, 0.1;
    0.1, 0.15, 0.1
];
init_data.shape_model.Faces = [
    1, 2, 3;
    1, 3, 4;
    5, 6, 7;
    5, 7, 8;
    1, 2, 6;
    1, 6, 5;
    2, 3, 7;
    2, 7, 6;
    3, 4, 8;
    3, 8, 7;
    4, 1, 5;
    4, 5, 8
];
init_data.shape_model.type = 'cuboid';

% Create fuel tank object
mission.true_SC{i_SC}.true_SC_fuel_tank{i_HW} =
True_SC_Fuel_Tank(init_data, mission, i_SC, i_HW);

% Update spacecraft body with fuel mass properties
mission.true_SC{i_SC}.true_SC_body.mass.propellant{i_HW}.mass =
init_data.initial_fuel_mass;
mission.true_SC{i_SC}.true_SC_body.mass.propellant{i_HW}.location =
init_data.location;
mission.true_SC{i_SC}.true_SC_body.mass.propellant{i_HW}.MI_over_m =
zeros(3,3); % Simple approximation

% Trigger mass properties update
mission.true_SC{i_SC}.true_SC_body.flag_update_SC_body_total_mass_COM_MI =
1;
end

```

Initialize First Spacecraft's Solar Panels

```

for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_solar_panel

    init_data = [];
    init_data.instantaneous_power_consumed = 0.01; % [W] (irrespective of
whether it is generating power or not)

```

```

init_data.instantaneous_data_rate_generated = (1e-3)*8; % [kbps] i.e. 1
Byte per sec

init_data.shape_model = [];
init_data.shape_model.Vertices = [0 0 0; 0.2 0 0; 0.2 0 -0.6; 0 0 -0.6];
    % [m] vertices
init_data.shape_model.Faces = [1 2 3; 1 4 3];
    % faces
init_data.shape_model.Face_reflectance_factor_solar_cell_side = [0.01;
0.01]; % reflectance factor of solar cell side
init_data.shape_model.Face_reflectance_factor_opposite_side = [0.5; 0.5];
    % reflectance factor of solar cell side
init_data.shape_model.Face_orientation_solar_cell_side = [0 -1 0];
    % orientation normal vector of solar cell side
init_data.shape_model.type = 'cuboid';

init_data.mass = 0.24; % [kg] ~ 2 kg/m^2

init_data.type = 'passive_deployed';
% 'body_mounted' : Stuck to SC side (only solar cell side is used for SRP)
% 'passive_deployed' : Passively deployed (orientation in body frame B
does not change, i.e. it is static)
% 'active_deployed_gimballed' : Actively gimballed (orientation in body
frame B changes)

init_data.packing_fraction = 0.74; % Packing fraction of solar cells in
solar panel
init_data.solar_cell_efficiency = 0.28; % Efficiency of each solar cell

if i_HW == 2
    init_data.shape_model.Vertices = [0 0 0.1; 0.2 0 0.1; 0.2 0 0.7; 0 0
0.7]; % [m] vertices
end

if i_HW == 3
    init_data.shape_model.Vertices =
mission.true_SC{i_SC}.true_SC_body.shape_model{1}.Vertices;
    init_data.shape_model.Faces =
mission.true_SC{i_SC}.true_SC_body.shape_model{1}.Faces(1:2,:);
    init_data.type = 'body_mounted';
end

mission.true_SC{i_SC}.true_SC_solar_panel{i_HW} =
True_SC_Solar_Panel(init_data, mission, i_SC, i_HW);

end

```

Initialize First Spacecraft's Battery

```

for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_battery

```

```

init_data = [];
init_data.maximum_capacity = 40; % [W hr]
init_data.charging_efficiency = 0.96; % [float <= 1]
init_data.discharging_efficiency = 0.96; % [float <= 1]
init_data.instantaneous_power_consumed = 1e-4; % [W]
init_data.instantaneous_data_rate_generated = (1e-3)*8; % [kbps] i.e. 1
Byte per sec

mission.true_SC{i_SC}.true_SC_battery{i_HW} = True_SC_Battery(init_data,
mission, i_SC, i_HW);

end

```

Initialize First Spacecraft's Onboard Memory

```

for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_memory

    init_data = [];
    init_data.maximum_capacity = 1e6; % [kb]
    init_data.instantaneous_power_consumed = 1; % [W]
    init_data.instantaneous_data_rate_generated = (1e-3)*8; % [kbps] i.e. 1
    Byte per sec

    mission.true_SC{i_SC}.true_SC_onboard_memory{i_HW} =
    True_SC_Onboard_Memory(init_data, mission, i_SC, i_HW);

end

```

Initialize First Spacecraft's Onboard Clock

```

for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_clock

    init_data = [];
    init_data.instantaneous_power_consumed = 0.1; % [W]
    init_data.instantaneous_data_rate_generated = (1e-3)*16; % [kbps] i.e. 2
    Bytes per sec
    init_data.mode_true_SC_onboard_clock_selector = 'Simple';
    init_data.measurement_wait_time = 0; % [sec]

    mission.true_SC{i_SC}.true_SC_onboard_clock{i_HW} =
    True_SC_Onboard_Clock(init_data, mission, i_SC, i_HW);

end

```

Initialize First Spacecraft's Cameras

```

for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_camera

    init_data = [];

```

```

    init_data.instantaneous_power_consumed = 10; % [W] https://
dragonflyaerospace.com/products/caiman/
    init_data.mode_true_SC_camera_selector = 'Simple';
    init_data.measurement_wait_time = 60; % [sec] -> This was 10x60 but in the
    v1 is 60 only and that is how switch mode is implemented

    init_data.location = [0.3 0.1 0.05]; % [m]
    init_data.orientation = [1 0 0]; % [unit vector]
    init_data.orientation_up = [0 0 1]; % [unit vector]

    init_data.resolution = [512 512]; % [x y] pixel
    init_data.field_of_view = 10; % [deg]
    init_data.flag_show_camera_plot = 0;
    init_data.flag_show_stars = 1;

    init_data.instantaneous_data_generated_per_pixel = (1e-3)* 8; % [kb]

    mission.true_SC{i_SC}.true_SC_camera{i_HW} = True_SC_Camera(init_data,
    mission, i_SC, i_HW);

end

```

Initialize First Spacecraft's Sun Sensors

```

for i_HW =
    1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_sun_sensor

    init_data = [];
    init_data.instantaneous_power_consumed = 36e-3; % [W] https://
www.cubesatshop.com/wp-content/uploads/2016/06/SSOCA60-Technical-
Specifications.pdf
    init_data.instantaneous_data_generated_per_sample = (4 + 1)*16e-3; %
    [kb] : 4 quaternion + 1 time vector, each of 16-bit depth
    init_data.mode_true_SC_sun_sensor_selector = 'Simple with Sun in FOV';
    init_data.measurement_wait_time = 0.1; % [sec]
    init_data.measurement_noise = deg2rad(0.5); % [rad] 0.5 degrees
    init_data.field_of_view = 60; % [deg]

    switch i_HW

        case 1
            init_data.location = [0.3 0.05 0.05]; % [m]
            init_data.orientation = [1 0 0]; % [unit vector]

        case 2
            init_data.location = [0 0.05 0.05]; % [m]
            init_data.orientation = [-1 0 0]; % [unit vector]

        case 3
            init_data.location = [0.15 0.2 0.05]; % [m]
            init_data.orientation = [0 1 0]; % [unit vector]

        case 4

```

```

        init_data.location = [0.15 0 0.05]; % [m]
        init_data.orientation = [0 -1 0]; % [unit vector]

    case 5
        init_data.location = [0.15 0.1 0.1]; % [m]
        init_data.orientation = [0 0 1]; % [unit vector]

    case 6
        init_data.location = [0.15 0.1 0]; % [m]
        init_data.orientation = [0 0 -1]; % [unit vector]

    otherwise
        error('Should not reach here!')
    end

    mission.true_SC{i_SC}.true_SC_sun_sensor{i_HW} =
    True_SC_Sun_Sensor(init_data, mission, i_SC, i_HW);

end

```

Initialize First Spacecraft's Star Tracker

```

for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_star_tracker

    init_data = [];

    init_data.instantaneous_power_consumed = 1.5; %
[W] https://www.bluecanyontech.com/static/datasheet/
BCT_DataSheet_Components_StarTrackers.pdf
    init_data.instantaneous_data_generated_per_sample = (4 + 1)*16e-3; %
[kb] : 4 quaternion + 1 time vector, each of 16-bit depth
    init_data.mode_true_SC_star_tracker_selector = 'Simple with Sun outside
FOV';
    init_data.measurement_wait_time = 0.1; % [sec]
    init_data.measurement_noise = 2e-4; % [rad]
    init_data.field_of_view = 90; % [deg]

    switch i_HW

        case 1
            init_data.location = [0.3 0.15 0.05]; % [m]
            init_data.orientation = [1 0 0]; % [unit vector]

        case 2
            init_data.location = [0 0.15 0.05]; % [m]
            init_data.orientation = [-1 0 0]; % [unit vector]

        case 3
            init_data.location = [0.10 0.2 0.05]; % [m]
            init_data.orientation = [0 1 0]; % [unit vector]

        otherwise

```

```

        error('Should not reach here!')
    end

    mission.true_SC{i_SC}.true_SC_star_tracker{i_HW} =
    True_SC_Star_Tracker(init_data, mission, i_SC, i_HW);

end

```

Initialize First Spacecraft's IMU

```

for i_HW = 1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_imu

    init_data = [];

    init_data.instantaneous_power_consumed = 0.6; % [W] https://www.micro-a.net/imu-tmpl.html
    init_data.instantaneous_data_generated_per_sample = (3 + 1)*16e-3; %
    [kb] : 3 angular velocity + 1 time vector, each of 16-bit depth
    init_data.mode_true_SC_imu_selector = 'Simple';
    init_data.measurement_wait_time = 0.1; % [sec]
    init_data.measurement_noise = 9.7e-5; % [rad/sec]

    init_data.location = [0 0 0]; % [m]
    init_data.orientation = [1 0 0]; % [unit vector]

    mission.true_SC{i_SC}.true_SC_imu{i_HW} = True_SC_IMU(init_data, mission,
    i_SC, i_HW);

end

```

Initialize First Spacecraft's Micro Thrusters

```

for i_HW =
    1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_micro_thruster

    init_data = [];

    init_data.instantaneous_power_consumption = 10; % Watts
    init_data.instantaneous_data_generated_per_sample = 1; % Kb
    init_data.mode_true_SC_micro_thruster_selector = 'Simple'; % Mode (Truth/
Simple)
    init_data.thruster_noise = 100*(1e-6); % Noise level [N](unit depends on
implementation)

    init_data.micro_thruster_ISP = 700;

    init_data.minimum_thrust = 0.001; % [N]
    init_data.maximum_thrust = 10*(1e-2); % [N]

    init_data.command_wait_time = 0.5; % Seconds between commands

    switch i_HW

```

```

case 1
    init_data.location = [0.3 0.1 0.05]; % [m]
    init_data.orientation = [0 1 0]; % [unit vector]

case 2
    init_data.location = [0.3 0.1 0.05]; % [m]
    init_data.orientation = [0 -1 0]; % [unit vector]

case 3
    init_data.location = [0.3 0.1 0.05]; % [m]
    init_data.orientation = [0 0 1]; % [unit vector]

case 4
    init_data.location = [0.3 0.1 0.05]; % [m]
    init_data.orientation = [0 0 -1]; % [unit vector]

case 5
    init_data.location = [0 0.1 0.05]; % [m]
    init_data.orientation = [0 1 0]; % [unit vector]

case 6
    init_data.location = [0 0.1 0.05]; % [m]
    init_data.orientation = [0 -1 0]; % [unit vector]

case 7
    init_data.location = [0 0.1 0.05]; % [m]
    init_data.orientation = [0 0 1]; % [unit vector]

case 8
    init_data.location = [0 0.1 0.05]; % [m]
    init_data.orientation = [0 0 -1]; % [unit vector]

case 9
    init_data.location = [0.15 0.2 0.05]; % [m]
    init_data.orientation = [0 0 1]; % [unit vector]

case 10
    init_data.location = [0.15 0.2 0.05]; % [m]
    init_data.orientation = [0 0 -1]; % [unit vector]

case 11
    init_data.location = [0.15 0 0.05]; % [m]
    init_data.orientation = [0 0 1]; % [unit vector]

case 12
    init_data.location = [0.15 0 0.05]; % [m]
    init_data.orientation = [0 0 -1]; % [unit vector]

otherwise
    error('Should not reach here!')
end

mission.true_SC{i_SC}.true_SC_micro_thruster{i_HW} =
True_SC_Micro_Thruster(init_data, mission, i_SC, i_HW);

```

end

Initialize the Reaction Wheels

Dart can be simulated using 3 or 4 wheels as an example !

```
for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel

    init_data = [];
    init_data.location = [0,0,0];
    init_data.radius = (43e-3)/2; % [m] radius of 1 RW
    init_data.mass = 0.137; % [kg] mass of 1 RW
    init_data.max_angular_velocity = 6500*2*pi/60; % [rad/s] 6500 RPM
    init_data.angular_velocity_noise = 0.001*2*pi/60; % [rad/s] velocity
noise (reduced from 0.01)
    init_data.instantaneous_data_volume = (3)*16e-3; % [kb] : velocity +
health + temperature, each of 16-bit depth
    init_data.max_torque = 3.2*1e-3; % Nm

    % Calculate and set maximum acceleration
    % This is redundant with the calculation in the True_SC_Reaction_Wheel
constructor,
    % but makes it explicit and easier to adjust
    moment_of_inertia = 0.5 * init_data.mass * init_data.radius^2;
    init_data.maximum_acceleration = init_data.max_torque /
moment_of_inertia; % rad/s^2

    init_data.power_consumed_angular_velocity_array = [1e-3*[180 600
6000]; [0 1000*2*pi/60 init_data.max_angular_velocity]]; % [power_array ;
velocity_array]

    % 3 wheel configuration

    if(mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
== 3)
        switch i_HW
            case 1
                init_data.orientation = [1, 0, 0]; % X-axis
            case 2
                init_data.orientation = [0, 1, 0]; % Y-axis
            case 3
                init_data.orientation = [0, 0, 1]; % Z-axis
        end
    end

    % 4 wheel configuration

    if(mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_reaction_wheel
== 4)
        switch i_HW
            case 1
```

```

        init_data.orientation = [1, 1, 0]/sqrt(2); % Diagonal in XY-
plane
        case 2
            init_data.orientation = [1, -1, 0]/sqrt(2); % Diagonal in XY-
plane
        case 3
            init_data.orientation = [0, 1, 1]/sqrt(2); % Diagonal in YZ-
plane
        case 4
            init_data.orientation = [0, 1, -1]/sqrt(2); % Diagonal in YZ-
plane
        end
    end

    mission.true_SC{i_SC}.true_SC_reaction_wheel{i_HW} =
    True_SC_Reaction_Wheel(init_data, mission, i_SC, i_HW);

end

```

Chemical Thruster Configuration

```

init_data = [];
i_HW =
    mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_chemical_thruster;

% Power and data parameters
init_data.instantaneous_power_consumption = 2.0; % [W] Base power draw
    (standby)
init_data.thruster_warm_up_power_consumed = 5.0; % [W] Power during
    warm-up phase
init_data.command_actuation_power_consumed = 15.0; % [W] Power during
    active thrust
init_data.instantaneous_data_generated_per_sample = 10; % [kb] per sample
init_data.chemical_thruster_noise = 10e-4; % [N] Thrust noise
    level

% Thruster properties
init_data.chemical_thruster_ISP = 200; % [s] Specific impulse
init_data.command_wait_time = 1; % [s] Minimum time
    between commands
init_data.location = [0.3, 0.2/2, 0.1/2]; % [m] Thruster
    location in body frame
init_data.orientation = [-1, 0, 0]; % Thrust direction
    (unit vector)

init_data.maximum_thrust = 1; % [N] Maximum thrust
    level
init_data.minimum_thrust = 0.01; % [N] Minimum thrust
    level

% Create chemical thruster object
mission.true_SC{i_SC}.true_SC_chemical_thruster =
    True_SC_Chemical_Thruster(init_data, mission, i_SC, i_HW);

```

Onboard Computer Configuration

```
for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_computer
    init_data = [];

    % Basic properties
    init_data.name = ['Onboard Computer ', num2str(i_HW)];

    % Set different properties for primary and backup computers
    if i_HW == 1
        % Primary computer
        init_data.instantaneous_power_consumed = 8.0;      % [W] Main flight
computer
        init_data.instantaneous_data_rate_generated = 2.0; % [kbps] for
telemetry and logs
        init_data.processor_utilization = 25;              % [%] Fixed CPU
usage
    else
        % Backup computer
        init_data.instantaneous_power_consumed = 4.0;      % [W] Backup in
standby mode
        init_data.instantaneous_data_rate_generated = 0.5; % [kbps] minimal
telemetry
        init_data.processor_utilization = 5;               % [%] Fixed low
utilization in standby
    end

    % Create onboard computer object
    mission.true_SC{i_SC}.true_SC_onboard_computer{i_HW} =
True_SC_Onboard_Computer(init_data, mission, i_SC, i_HW);
end
```

Spacecraft Communication Links Configura- tion

```
for i_HW =
1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_communication_link
    init_data = [];

    % Configure appropriate link based on index
    if i_HW == 1
        % Downlink: Spacecraft to Earth
        init_data.TX_spacecraft = i_SC;          % Transmitter is this
spacecraft
        init_data.TX_spacecraft_Radio_HW = 1;    % Using antenna 1

        init_data.RX_spacecraft = 0;             % Receiver is Ground
Station (0)
        init_data.RX_spacecraft_Radio_HW = 1;    % Using GS antenna 1
    end
end
```

```

        init_data.flag_compute_data_rate = 0;           % Use given data rate
instead of computing
        init_data.given_data_rate = 360;               % [kbps] Downlink data
rate
    else
        % Uplink: Earth to Spacecraft
        init_data.TX_spacecraft = 0;                   % Transmitter is Ground
Station (0)
        init_data.TX_spacecraft_Radio_HW = 1;          % Using GS antenna 1

        init_data.RX_spacecraft = i_SC;                 % Receiver is this
spacecraft
        init_data.RX_spacecraft_Radio_HW = 1;          % Using antenna 1

        init_data.flag_compute_data_rate = 0;           % Use given data rate
instead of computing
        init_data.given_data_rate = 0;                 % [kbps] Set low to avoid
overfilling memory
    end

    % Create communication link object
    mission.true_SC{i_SC}.true_SC_communication_link{i_HW} =
True_SC_Communication_Link(init_data, mission, i_SC, i_HW);
end

```

Initialize Solar Radiation Pressure

```

init_data = [];
init_data.enable_SRP = 1; % Enable SRP calculations

mission.true_SC{i_SC}.true_SRP = True_SRP(init_data, mission, i_SC);

```

Initialize Gravity Gradient for Earth

```

init_data = [];
init_data.enable_G2 = 0; % Disable gravity gradient because Dart is an
interceptor
init_data.main_body = "Earth";

mission.true_SC{i_SC}.true_gravity_gradient = True_Gravity_Gradient(init_data,
mission, i_SC);

```

Spacecraft Software: Executive Configuration

```

init_data = [];
init_data.sc_modes = {'Point camera to Target', 'Maximize SP Power', 'Point
Thruster along DeltaV direction', 'DTE Comm'};
init_data.mode_software_SC_executive_selector = 'DART';

mission.true_SC{i_SC}.software_SC_executive = Software_SC_Executive(init_data,
mission, i_SC);

```

Spacecraft Software: Attitude Estimation Configuration

```
init_data = [];  
init_data.mode_software_SC_estimate_attitude_selector = 'Truth'; % Use true  
    attitude values  
  
mission.true_SC{i_SC}.software_SC_estimate_attitude =  
    Software_SC_Estimate_Attitude(init_data, mission, i_SC);
```

Spacecraft Software: Orbit Estimation Configuration

```
init_data = [];  
init_data.mode_software_SC_estimate_orbit_selector = 'TruthWithErrorGrowth';  
    % Use true orbit values with error growth when target not visible  
  
mission.true_SC{i_SC}.software_SC_estimate_orbit =  
    Software_SC_Estimate_Orbit(init_data, mission, i_SC);
```

Spacecraft Software: Orbit Control Configuration

```
init_data = [];  
init_data.max_time_before_control = 0.5*60*60 + 900; % 45 minutes  
init_data.mode_software_SC_control_orbit_selector = 'DART';  
  
mission.true_SC{i_SC}.software_SC_control_orbit =  
    Software_SC_Control_Orbit(init_data, mission, i_SC);
```

Spacecraft Software: Attitude Control Configuration

```
init_data = [];  
init_data.mode_software_SC_control_attitude_selector = 'DART Control  
    Asymptotically Stable send to actuators'; % 'DART Control Asymptotically  
    Stable send to actuators'  
init_data.control_gain = [1 0.2]; % Controller gain parameters  
  
mission.true_SC{i_SC}.software_SC_control_attitude =  
    Software_SC_Control_Attitude(init_data, mission, i_SC);
```

Spacecraft Software: Communication Configuration

```
init_data = [];
```

```

init_data.mode_software_SC_communication_selector = 'DART';
init_data.instantaneous_data_generated_per_sample = (1e-3)*8*2; % [kb] 2
    Bytes per sample
init_data.attitude_error_threshold_deg = 1; % [deg] Max attitude error for
    communication

init_data.data = [];
init_data.data.last_communication_time = 0;
init_data.data.wait_time_comm_dte = 0.7*60*60; % [sec] 42 minutes between DTE
    comms

mission.true_SC{i_SC}.software_SC_communication =
    Software_SC_Communication(init_data, mission, i_SC);

```

Spacecraft Software: Power Management Configuration

```

init_data = [];
init_data.mode_software_SC_power_selector = 'Generic';
init_data.instantaneous_data_generated_per_sample = (1e-3)*8*2; % [kb] 2
    Bytes per sample

mission.true_SC{i_SC}.software_SC_power = Software_SC_Power(init_data,
    mission, i_SC);

```

Spacecraft Software: Data Handling Configuration

```

init_data = [];
init_data.mode_software_SC_data_handling_selector = 'Generic';
init_data.instantaneous_data_generated_per_sample = (1e-3)*8*2; % [kb] 2
    Bytes per sample

mission.true_SC{i_SC}.software_SC_data_handling =
    Software_SC_Data_Handling(init_data, mission, i_SC);

```

Final Things to Do Before Running the Simulation

```

% Initialize mass, COM, MI
func_update_SC_body_total_mass_COM_MI(mission.true_SC{i_SC}.true_SC_body);

% Initialize store of Power
func_initialize_store_HW_power_consumed_generated(mission.true_SC{i_SC}.true_SC_power,
    mission);

% Initialize store of Data Handling
func_initialize_store_HW_data_generated_removed(mission.true_SC{i_SC}.true_SC_data_handling,
    mission);

```

```
% Initialize onboard computers
for i_HW =
    1:1:mission.true_SC{i_SC}.true_SC_body.num_hardware_exists.num_onboard_computer
        mission.true_SC{i_SC}.true_SC_onboard_computer{i_HW}.flag_executive = 1; %
        Start active
    end
```

Save All Data

```
clear init_data i_SC i_HW i_target

% Vizualise the SC in 3D + Dashboard
func_visualize_SC(mission.storage, mission, true);
save([mission.storage.output_folder, 'all_data.mat'], '-v7.3')
```

Execute Main File

```
run main_v3.m
```

Save All Data

```
close all
disp ('-----')
disp('Simulation Over. Starting saving data to disk...');

save([mission.storage.output_folder, 'all_data.mat'], '-v7.3')

disp(['Finished writing to file "all_data.mat" in folder : ',
    mission.storage.output_folder])
disp ('-----')
```

Plots

Use our memory-optimized visualization

```
fprintf('Starting memory-optimized visualization...');
memoryInfo = evalc('dispmemory()');
disp(['Current memory before visualisation - ', memoryInfo(1:end-1), ])

func_visualize_simulation_data(mission.storage, mission);

memoryInfo = evalc('dispmemory()');
fprintf('Visualization complete. ');
disp(['Current memory after visualisation - ', memoryInfo(1:end-1), ])
disp ('-----')
```

Published with MATLAB® R2022a

Bibliography

- [1] S. Bandyopadhyay, Y. K. Nakka, L. Fesq, and S. Ardito, “Design and development of MuSCAT: Multi-spacecraft concept and autonomy tool,” in *AIAA ASCEND Autonomous Operations in Space*, Las Vegas, NV, 2024.