Jeff’s Laboratory

NE08 - Solid Rocket Motor Databook

Comments	Revision	Date	Author
Initial Release	A	January 6, 2025	J. Mays

1. References

1. NE10 - Thrust Vector Assembly Databook
2. https://www.apogeerockets.com/Rocket_Motors/Cesaroni_Propellant_Kits/29mm_Motors/3-Grain_Motors/Cesaroni_P29-3G_Mellow_G33
3. https://www.apogeerockets.com/Rocket_Motors/Cesaroni_Casings/29mm_Casings/Cesaroni_29mm_3-Grain_Case

2. Purpose

This document describes the solid rocket motor (SRM) computational model at a high level as it pertains to the New Mays (NM) Model Rocket. While it does cover valuable information that may be of interest to the reader, this document is not intended to discuss every aspect of the model. The model was intentionally designed to be simple, only containing enough fidelity for the purposes of Monte-Carlo (MC) simulations of the New Mays vehicle to test and validate the flight software algorithms and verify requirements. This document discusses the respective simulation model written in Matlab/Simulink.

3. Hardware Context

The NM rocket's primary effector is a gimballed solid rocket motor using the thrust vector assembly [1] under thrust vector control. SRMs are rocket motors which use solid propellants, and consist of a casing, nozzle, grain (propellant), and an ignitor. Once the ignitor is lit, it starts a predictable chain reaction with the propellant, generating high-velocity exhaust gas out of the nozzle, producing thrust. This thrust from a control's perspective is "open-loop," meaning once the reaction is started, it cannot be altered without external effects.

The SRM chosen for NM such that we meet performance requirements is the Cesaroni – P29-3G (G33). Figure 1 shows the SRM in its various components, and Table 1 show generic specifications about the motor. This will all be tested in a static hotfire test and the data will be presented later in this document.

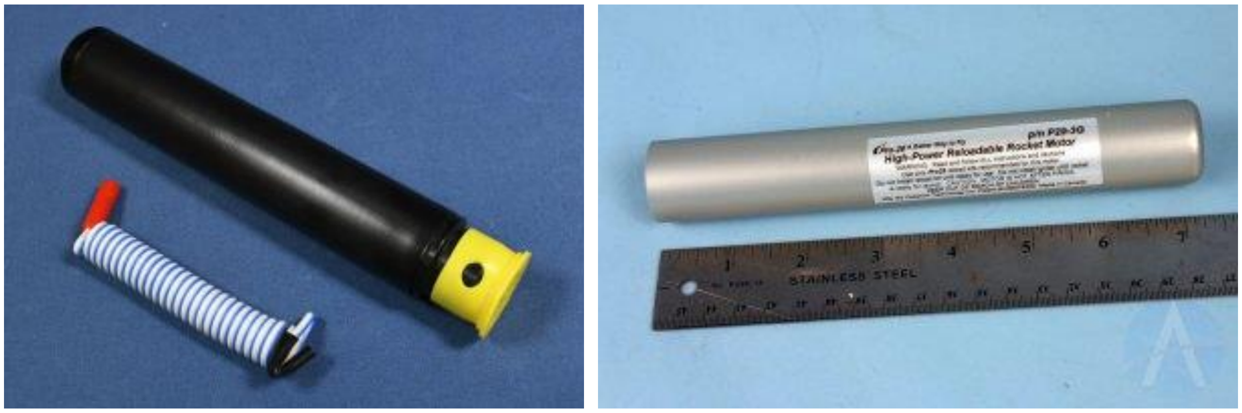

Figure 1: G33 3-grain SRM with ignitor and housing [2,3]

Table 1: Cesaroni – P29-3G (G33) general parameters [2,3]

Parameter	Spec
Grain Diameter	29 mm
Approx. Burn Time	4.4 sec
Total Impulse	143.2 N-s
Max Thrust	66.1 N
Total Mass	0.197 kg
Propellant Mass	0.0791 kg

The SRM grain and housing fit inside the TVA's inner SRM sleeve, as illustrated in Figure 2.

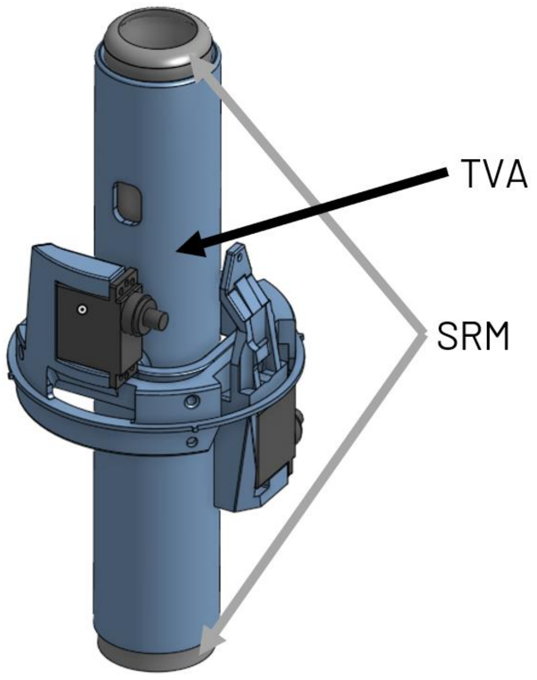

Figure 2: TVA with SRM mounted inside the SRM sleeve [1]

The main flight computer (MFC) contains a few circuits that can ignite pyrotechnic channels: 2 chute channels and 1 SRM channel. These channels use the onboard lipo-battery and a power switching N-channel mosfet to route current to the pyrotechnic charge, causing the chemical reaction that ignites the SRM or deploys the parachute. Figure 3 illustrates this circuit within the MFC.

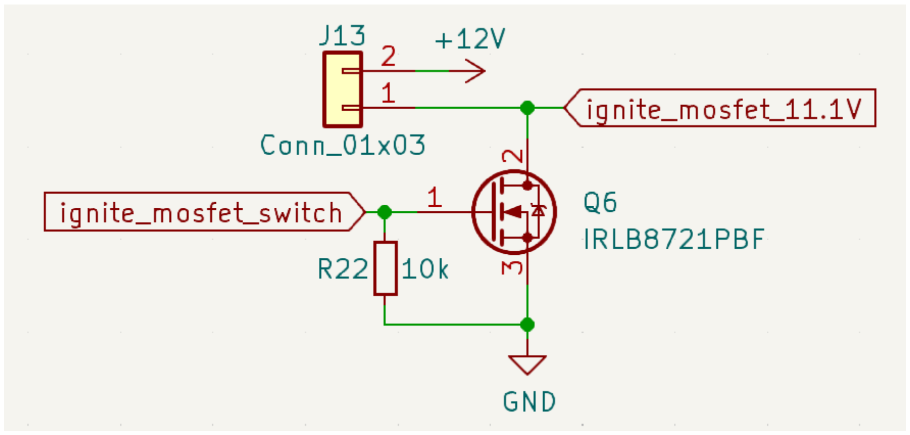

Figure 3: SRM mosfet channel circuit

4. Hotfire / SRM Thrust Test

To characterize the SRM used for flight, a launch pad with load cells that could record thrust data over time was developed. It was able to capture the time series thrust profile and pyro-ignition delay time. This data was then used as a baseline set for the Monte-Carlo integrated non-linear simulation and linear stability analysis. This data is shown later in Figure *9*. Below in Figure 4 is an image taken from the thrust test.

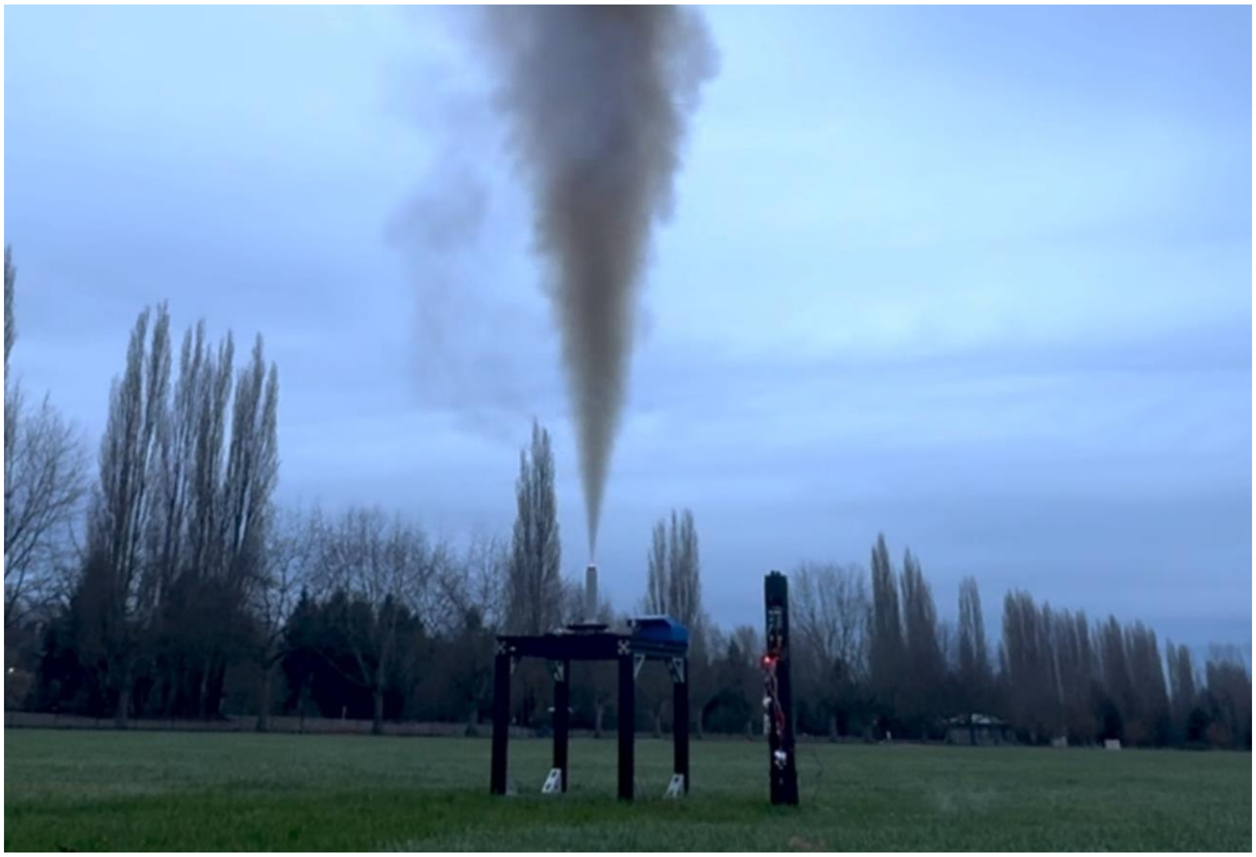

Figure 4: Thrust testing of the G33 SRM

5. Modeling & Simulation

5.1. Model Initialization

For each Monte-Carlo index, a loading script is performed to parameterize the SRM model for that particular simulation index. The following snip of the `loadSRMParameters()` is used to initialize all variables required for the SRM simulation model. It assumes there is an available timeseries matlab data file on the path, which represents the nominal thrust profile.

```
function [srm_constant, srm_tunable] = loadSRMParameters(mc, config)
% Load Solid Rocket Motor (SRM) parameters
%
% Author: Jeff Mays

%% SRM choice
switch lower(config.engine)
    case 'f25'
        srm = load('F25.mat');
    case 'g33'
        srm = load('G33.mat');
    otherwise
        error([config.engine ' is not an available SRM model...']);
end
% Disperse

%% Thrust Lookup
% Force scale factor for the SRM burn
force_scale = mc.disperse('force_scale', ...
                          'Dispersion', 'Normal', ...
                          'Nominal',     1.0, ...
                          'StdDev',      0.05/3);

% SRM force slope factor that augments the profile of the burn
thrust_augment_slope = mc.disperse('thrust_augment_slope', ...
                                   'Dispersion', 'Normal', ...
                                   'Nominal',     0.0, ...
                                   'StdDev',      0.02);

% Delay in chemical reaction of the burning SRM propellant
burn_delay = mc.disperse('burn_delay', ...
                         'Dispersion', 'Uniform', ...
                         'Nominal',     1.0*C_SEC, ...
                         'Min',         0.75*C_SEC, ...
                         'Max',         1.25*C_SEC);

% Scale the time vector
time_scale = mc.disperse('time_scale', ...
                         'Dispersion', 'Normal', ...
                         'Nominal',     1.0, ...
                         'StdDev',      0.07/3);

% Finalize lookup
srm_tunable.time_lookup   = srm.performance.srm_time * time_scale + burn_delay;

% Determine number of elements in the lookup table and use that to slope
% the output
elems = 1:numel(srm_tunable.time_lookup);

% Create a thrust augment that will be added to the base thrust lookup
thrust_augment = linspace(0, (numel(elems)-1)*thrust_augment_slope, numel(elems));

% Zero the first values in the lookup table so that we dont get zero thrust
% until the srm ignites from the burn delay
no_thrust_i = find(srm.performance.srm_thrust == 0);
thrust_augment(no_thrust_i) = 0; %#ok<FNDSB>

% Generate the thrust lookup table
thrust_lookup = (srm.performance.srm_thrust * force_scale) + thrust_augment;
```

```matlab
neg_thrust_i = find(thrust_lookup <= 0);
thrust_lookup(neg_thrust_i) = 0; %#ok<FNDSB>
srm_tunable.thrust_lookup = thrust_lookup;

% Assert the tables are aligned in number of elements
assert(numel(srm_tunable.time_lookup) == numel(srm_tunable.thrust_lookup), ...
    'Number of SRM time lookups are not eq. to SRM thrust lookups')

%% Thrust Alignment

% Misalign the thrust (constant for now)
srm_tunable.C_tube_to_vector = eye(3);

%% Thrust Swirl

% Swirl the thrust to produce srm body frame moments
srm_tunable.swirl_factor = 0.0; % gets multiplied by total thrust

%% Mass
% The mass burn should not be linear, but a function of the impulse
% produced

% Disperse mass of the SRM grain and housing
grain_mass = mc.disperse('grain_mass', ...
                         'Dispersion', 'Normal', ...
                         'Nominal',     srm.mass.grain * C_KG, ...
                         'StdDev',      0.005/3 * C_KG); % Off my 5 grams 1 sigma
housing_mass = mc.disperse('grain_mass', ...
                           'Dispersion', 'Normal', ...
                           'Nominal',     srm.mass.srm_housing * C_KG, ...
                           'StdDev',      0.005/3 * C_KG); % Off my 5 grams 1 sigma

% Inertia
R = 0.6 * C_IN; % Radius of SRM
L = 8 * C_IN; % Length of SRM
Ixx = @(M,R) 1/2 * M * R^2;
Iyyzz = @(M,R,L) 1/4 * M * R^2 + 1/12 * M * L^2;

% Inertia of the housing
inertia6_housing = [Ixx(housing_mass, R); ...
                    Iyyzz(housing_mass, R, L); ...
                    Iyyzz(housing_mass, R, L); ...
                    0; 0; 0];
inertia6_grain = [Ixx(grain_mass, R); ...
                  Iyyzz(grain_mass, R, L); ...
                  Iyyzz(grain_mass, R, L); ...
                  0; 0; 0];

% Find mass burn profile from the SRM impulse
int_notime = [];
for ii = 1:numel(srm_tunable.thrust_lookup)
    int_notime(end+1) = trapz(srm.performance.srm_thrust(1:ii)); %#ok<AGROW>
end

% The profile needs to burn the entire grain, so adjust the profile
burn_mass_scale_factor = int_notime(end) / grain_mass;
grain_mass_burn_profile = -1.0 * int_notime / burn_mass_scale_factor;

% Inertia portion
burn_inertia_scale_factor = int_notime(end) ./ inertia6_grain;
burn_inertia_scale_factor(isinf(burn_inertia_scale_factor)) = 0;
grain_inertia_burn_profile = -1.0 * int_notime ./ burn_inertia_scale_factor;
grain_inertia_burn_profile(isinf(grain_inertia_burn_profile)) = 0; % Remove inf
grain_inertia_burn_profile(isnan(grain_inertia_burn_profile)) = 0; % remove nan

% Make a grain burn table (add back SRM housing mass)
srm_tunable.mass_lookup = grain_mass_burn_profile + (housing_mass + grain_mass) * C_KG;
srm_tunable.inertia_lookup = grain_inertia_burn_profile + (inertia6_housing + inertia6_grain) *
C_KG*C_M^2;
```

```
srm_constant = 0;
end
```

5.2. Simulink Model

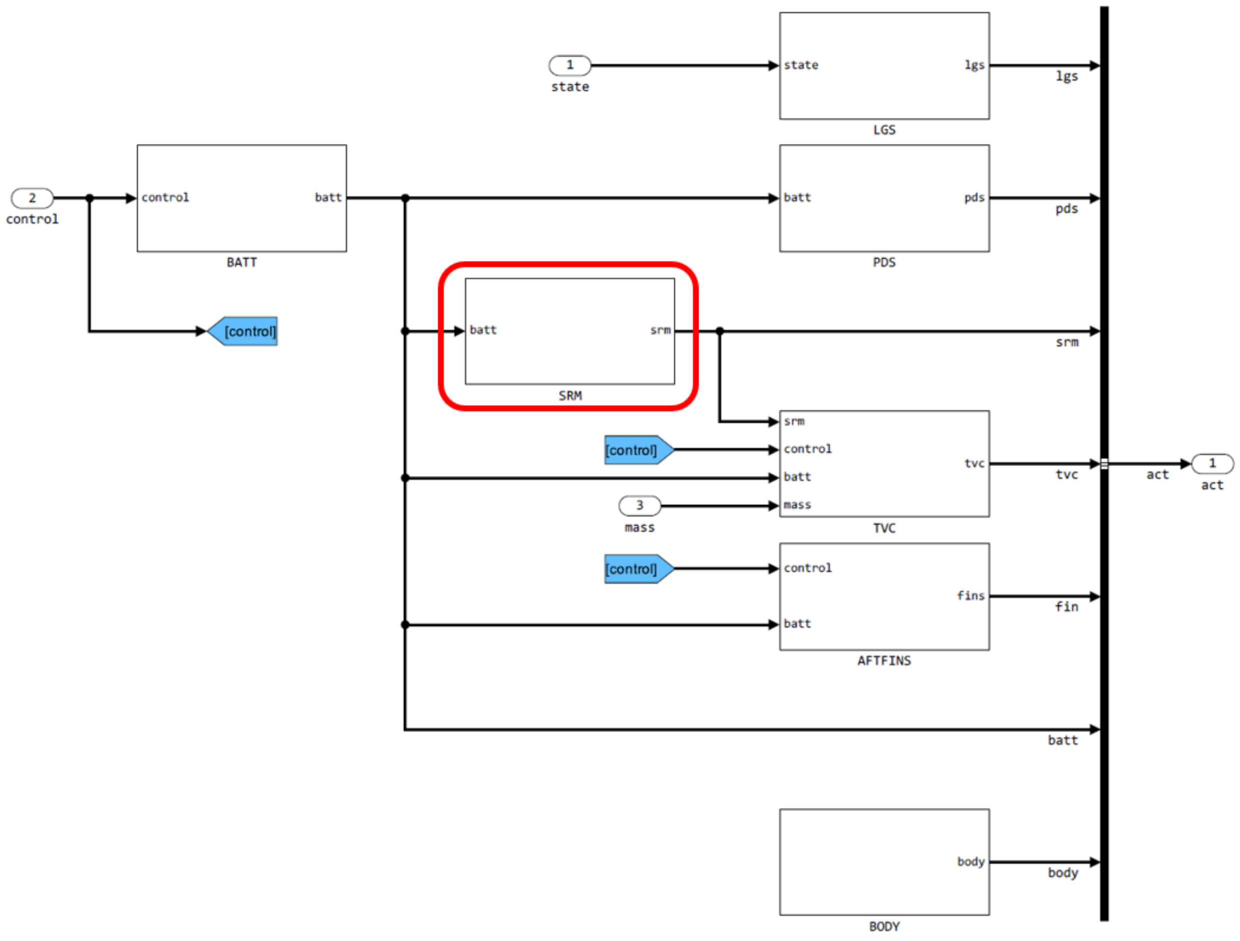

Figure 5: nm_sim/ROCKET/MODELS/ACTUATORS

Within the SRM model exists the library reference of the SRM. It takes in two inputs: pyrotechnic voltage and the center of mass of the SRM in the FAB frame. Since the center of mass of the SRM is not changing location (or is negligibly small) we can assume it is a single value. Once the Pyrotechnic voltage exceeds a certain limit, the model will kick off a pre-define thrust profile that is manipulated depending on the monte-carlo index

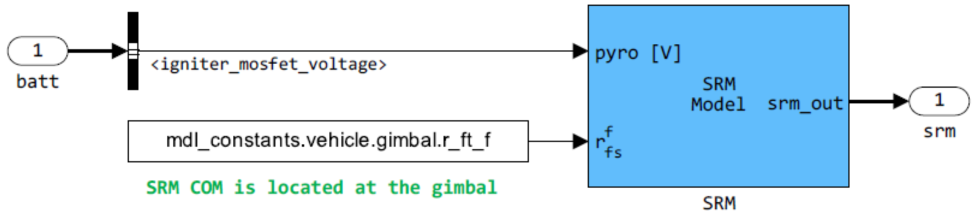

Figure 6: nm_sim/ROCKET/MODELS/ACTUATORS/SRM

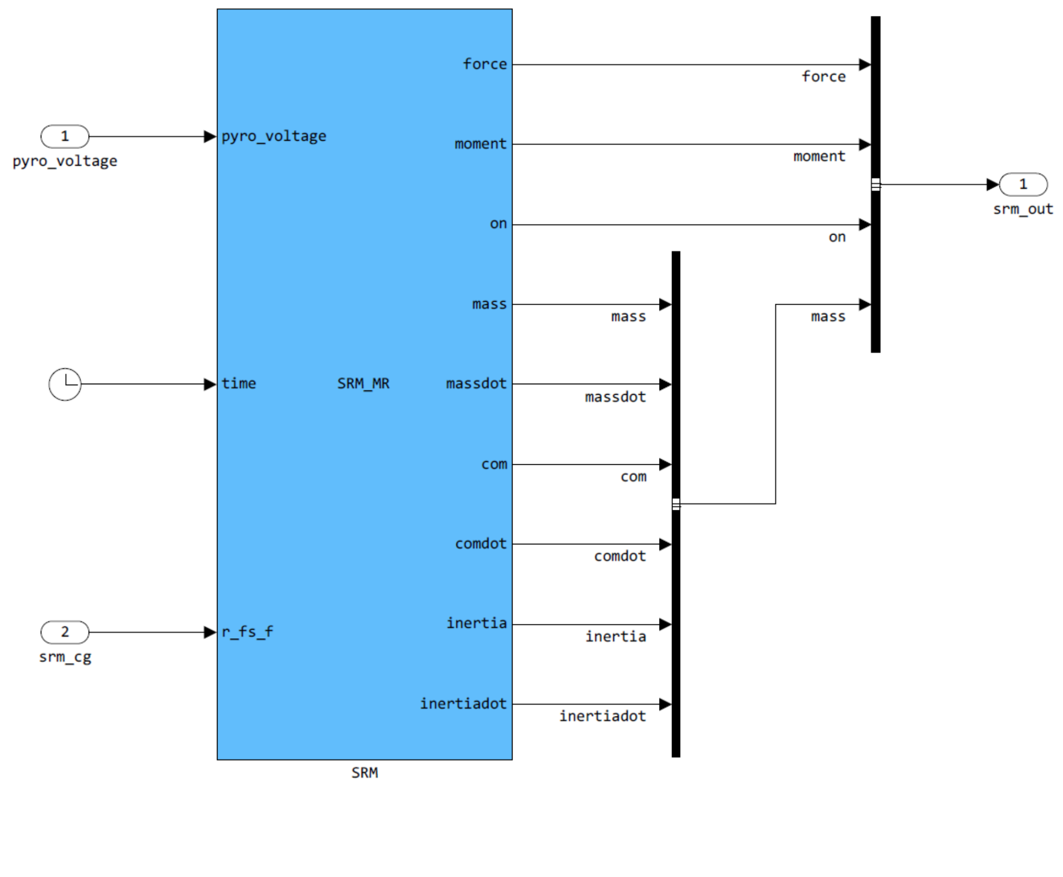

Figure 7: nm_sim/ROCKET/MODELS/ACTUATORS/SRM/SRM

The model is nothing fancy, but rather a simple lookup table of the thrust profile via time. Below is the `Matlab.system` object code which is used to provide the simulation with the true plant SRM states.

```
classdef SRM_MR < matlab.System
    % Solid Rocket Motor model
    %
    % Some assumptions:
    %   - The thrust profile is assumed to be at sea level.
    %   - We do not account for any atmospheric effects since this vehicle
    %     doesnt pass through a large altitude range.
    %   - Mass change is a function of the impulse from the dispersed
    %     thrust profile.
    %   - There is a delay time that is associated with the time taken for
    %     the chemical reaction to start inside the SRM. This usually is a
    %     function of how fast the igniter ignites and sits inside the SRM.
    %
    % Author: Jeff Mays

    % Public, tunable properties
    properties
        % Constant Values
        SRM = struct('constant',0);

        % Tunable Values
        SRM_TUNABLE = struct('tunable',0);

        % Time
        DT = 0.01;
    end

    properties (DiscreteState)
    end

    % Pre-computed constants or internal states
    properties (Access = private)
        ignite_sent = false;
        ignition_sent_time = 0.0;
        burn_complete = false;
        prev_mass = 0;
        firstRun = true;
        % output
    end

    methods (Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
        end

        function [force, moment, on, mass, massdot, com, comdot, inertia, inertiadot] = stepImpl(self, pyro_voltage,
time, r_fs_f)
            % Implement algorithm.
            % Inputs:
            %   pyro_fire   Boolean given by the sim (FM) indicating pyro is lit
            %   time        Time in the simulation, since this fcn uses lookups
            %   r_fs_f      Position of the SRM CG in the FAB frame
            %
            % Outputs:
            %   force       SRM force in local SRM frame
            %   moment      SRM moment in local SRM frame
            %   on          Boolean indicating if SRM is on or not
            %   mass        SRM mass
            %   massdot     SRM mass rate of change
            %   com         SRM CG in FAB
            %   comdot      SRM CG rate of change
            %   inertia     SRM inertia
            %   inertiadot  SRM inertia rate of change

            % Enforce
            pyro_fire = pyro_voltage > 0;
            time = double(time);

            % Pre fluff
            on      = false; %#ok<*NASGU>
            force   = [0; 0; 0];
            moment  = [0; 0; 0];
            mass    = self.SRM_TUNABLE.mass_lookup(1) * C_KG;
            massdot = 0 * C_KG/C_SEC;
            com     = [0;0;0] * C_M;
            comdot  = [0;0;0] * C_M/C_SEC;
            inertia    = self.SRM_TUNABLE.inertia_lookup(:,1) * C_KG*C_M^2;
            inertiadot = [0;0;0;0;0;0] * C_KG*C_M^2/C_SEC;

            % Setup prev_mass for derivative
            if (self.firstRun == true)
                self.prev_mass = mass;
                self.firstRun = false;
            end

            % Choose when to ignite the SRM
            if (pyro_fire == true && self.ignite_sent == false)
                self.ignite_sent = true; % Latch state so SRM continues even if pyro goes low...
                self.ignition_sent_time = time;
            end

            if (self.ignite_sent == false)
                mass = self.SRM_TUNABLE.mass_lookup(1) * C_KG;
                inerita = self.SRM_TUNABLE.inertia_lookup(:,1) * C_KG*C_M^2;
            end

            % If burn has yet to complete and we have ignited...
            thrust = 0.0 * C_N;
            if (self.burn_complete == false && self.ignite_sent == true)
                % Adjust time
                adjtime = time - self.ignition_sent_time;
                if (adjtime >= self.SRM_TUNABLE.time_lookup(1) && adjtime < self.SRM_TUNABLE.time_lookup(end))
                    thrust = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.thrust_lookup, adjtime, 'Linear')
* C_N;
```

```
                    mass = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.mass_lookup, adjtime, 'Linear') *
C_KG;
                    inertia(1,1) = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.inertia_lookup(1,:),
adjtime, 'Linear') * C_KG*C_M^2;
                    inertia(2,1) = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.inertia_lookup(2,:),
adjtime, 'Linear') * C_KG*C_M^2;
                    inertia(3,1) = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.inertia_lookup(3,:),
adjtime, 'Linear') * C_KG*C_M^2;
                    inertia(4,1) = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.inertia_lookup(4,:),
adjtime, 'Linear') * C_KG*C_M^2;
                    inertia(5,1) = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.inertia_lookup(5,:),
adjtime, 'Linear') * C_KG*C_M^2;
                    inertia(6,1) = interp1(self.SRM_TUNABLE.time_lookup, self.SRM_TUNABLE.inertia_lookup(6,:),
adjtime, 'Linear') * C_KG*C_M^2;
                else
                    thrust = 0.0 * C_N;
                end

                if(adjtime >= self.SRM_TUNABLE.time_lookup(end))
                    self.burn_complete = true;
                end

                force  = self.SRM_TUNABLE.C_tube_to_vector * [thrust;                                0; 0] * C_N;
                moment = self.SRM_TUNABLE.C_tube_to_vector * [thrust * self.SRM_TUNABLE.swirl_factor; 0; 0] * C_N*C_M;
            else
                force  = [0; 0; 0] * C_N;
                moment = [0; 0; 0] * C_N*C_M;
            end

            if (self.burn_complete == true)
                mass = self.SRM_TUNABLE.mass_lookup(end) * C_KG;
                inertia(1,1) = self.SRM_TUNABLE.inertia_lookup(1,end) * C_KG*C_M^2;
                inertia(2,1) = self.SRM_TUNABLE.inertia_lookup(2,end) * C_KG*C_M^2;
                inertia(3,1) = self.SRM_TUNABLE.inertia_lookup(3,end) * C_KG*C_M^2;
                inertia(4,1) = self.SRM_TUNABLE.inertia_lookup(4,end) * C_KG*C_M^2;
                inertia(5,1) = self.SRM_TUNABLE.inertia_lookup(5,end) * C_KG*C_M^2;
                inertia(6,1) = self.SRM_TUNABLE.inertia_lookup(6,end) * C_KG*C_M^2;
            end

            % Create mass dot
            massdot = (mass - self.prev_mass) / self.DT;

            % Com
            com = [r_fs_f(1); r_fs_f(2); r_fs_f(3)] * C_M/C_SEC;
            comdot = [0;0;0] * C_M/C_SEC;

            % Inertia6dot
            inertiadot = [0;0;0;0;0;0] * C_KG*C_M^2/C_SEC;

            % SRM on
            if norm(force) > 0
                on = true;
            else
                on     = false;
                force  = [0; 0; 0] * C_N;
                moment = [0; 0; 0] * C_N*C_M;
            end

        end

        function resetImpl(obj)
            % Initialize / reset internal properties
        end
    end
end
```

5.3. Model Validation

The library reference of the SRM model was taken and put into its own environment for checks with the ground hot fire test data. The goal of the model is to capture all possible and expected performance of the real system.

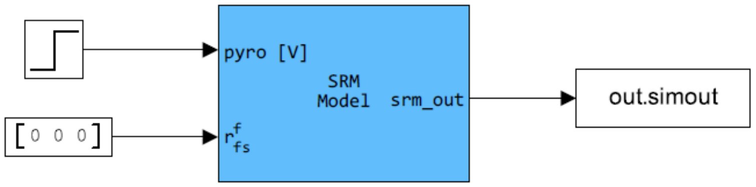

Figure 8: SRM model validation environment

On December 19, 2024, a load cell test was performed of the G33 SRM. This provided timeseries data such that the model could be updated and then dispersed to hopefully capture the SRM performance. The following figure illustrates the nominal data, the dispersed simulation sets (301 total simulations) and the test data captured from a set of load cells. Note that the thrust at the end of the test data states there is negative thrust, but this is due to the burning of the propellant while on the test stand.

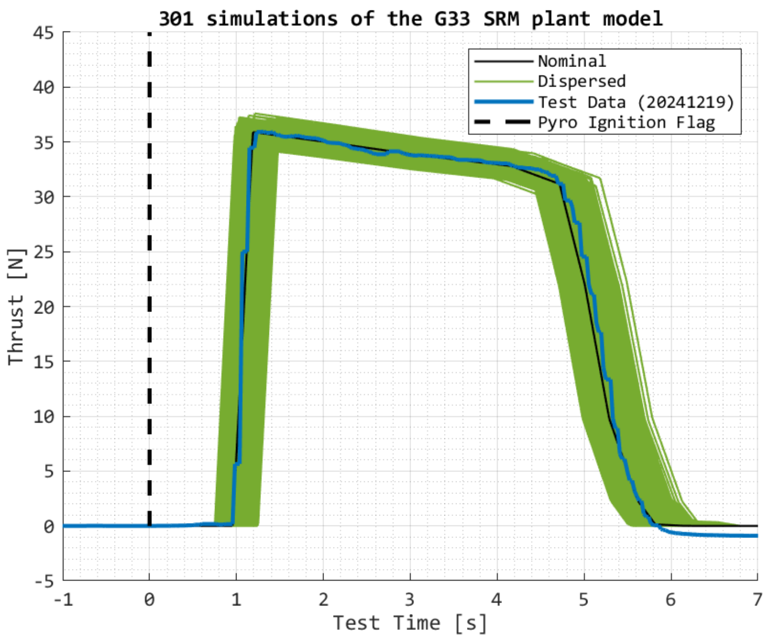

Figure 9: G33 thrust profile timeseries

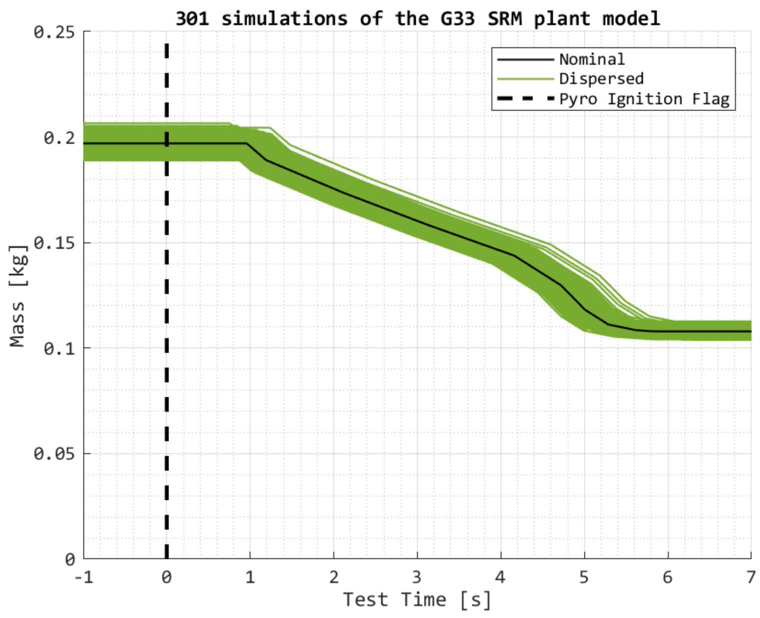

Figure 10: G33 mass timeseries

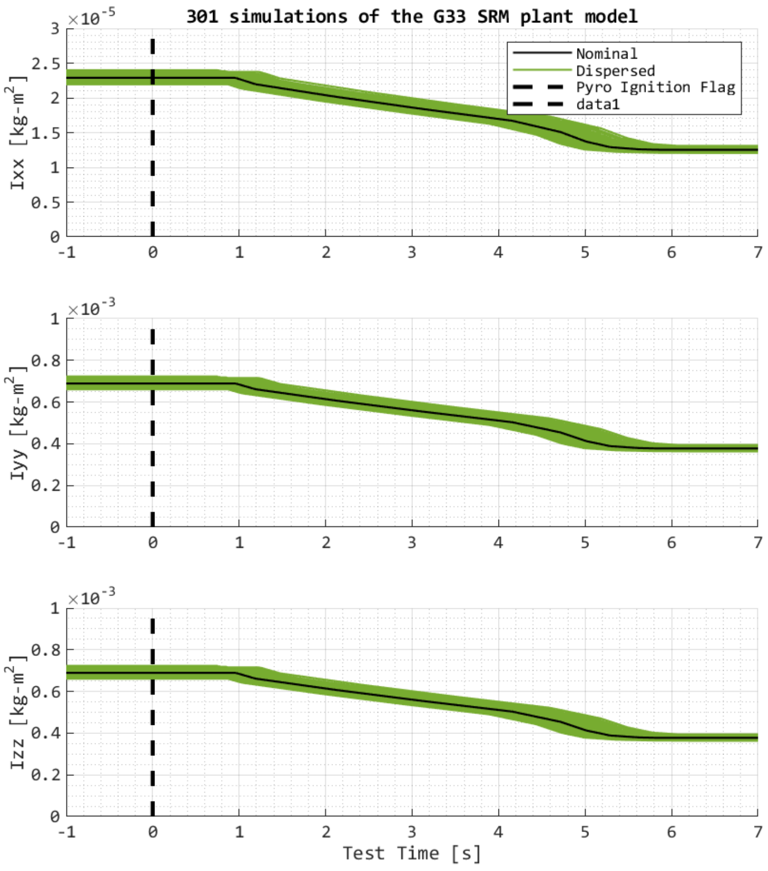

Figure 11: G33 inertia timeseries