Dynamic Bending Rig

Introduction & Context

There are two points in flight that we are most concerned about: the rocket as it has just left the launch rail(as it will be at it's slowest), and the rocket at burnout(max Q).

We will be using the continuous gust model(dryden or von karman) for forces at max Q(burnout), and discrete gust model(1-cosine) for worst case loading just after the rocket leaves the launch rail.

Source: http://www.aspirespace.org.uk/downloads/Rocket%20vehicle%20loads%20and%20airframe%20design.pdf

Original source for gust is here: https://ntrs.nasa.gov/api/citations/19990008476/downloads/19990008476.pdf

ASSUMPTIONS MADE FOR ANALYSIS OF SHEAR FORCE FROM THE (1-cosine gust) MIL-F-8785B

Note that once the (1-cosine gust) MIL-F-8785B velocity is calculated, and used in the calculation for shear, these important assumptions are made.

For calculating Cna(which is used to calculate lift), the follow assumptions apply from Barrowman's 1967 equations:

2.40 GENERAL ASSUMPTIONS

1. The angle-of-attack is very near zero.
2. The flow is steady and irrotational.
3. The vehicle is a rigid body.
4. The nose tip is a sharp point.

NOTE FROM 2/17/26: Also note that there was a mistake below in definining static margin, which decreased the amount of lift force, and thus shear, shown below. Will be trying to adjust before meeting with Peter on Wednesday(2/18/26), but timeline is tight with homework

Additionally, I messed up beta by using the mach freestream values at motor burnout, instead of where the angle of attack = 0

NASA Discrete Gust Model(1-cosine gust) MIL-F-8785B

https://ntrs.nasa.gov/api/citations/20090022159/downloads/20090022159.pdf

$$V_g(x) = \frac{V_m}{2}(1 - cos(\frac{\pi d}{d_m})) \qquad for\ \ 0 \leq x \leq 2d_m$$

$V_m(d)$ - gust magnitude (ft/s or m/s)

V_m - design gust velocity (set by altitude & category)

d - Distance flown into the gust

d_m - Gust gradient length (half-length of gust)

Note that for design gust velocity (from NASA):

$$\omega_{gust} \sim \frac{\pi V}{H}$$

After Rail: Interpolate the half-length of gust

Assuming that the worst loading case for the cosine gust is

CALCULATE THE Vdm from

LV3.1 launch rail length = 4.505706 meters

LV3.1 rocket length = 3.849378 meters

Interested in the wind speed at (3.849378 meters + 4.505706 meters) = 8.355084 meters

```
rocketH = 3.849378;
launchrailH = 4.505706
```

```
launchrailH =
4.5057
```

```
%
heightRocketAfterRail = rocketH + launchrailH
```

```
heightRocketAfterRail =
8.3551
```

~ round up to 8.4 meters.

Load from Open Rocket CSV

```
%% LOAD CSV
filePath = "OpenRocket.csv";
T = readtable(filePath);
```

```
Warning: Column headers from the file were modified to make them valid MATLAB identifiers before creating
variable names for the table. The original column headers are saved in the VariableDescriptions property.
Set 'VariableNamingRule' to 'preserve' to use the original column headers as table variable names.
```

```
disp("Columns found in CSV:");
```

```
Columns found in CSV:
```

```
disp(string(T.Properties.VariableNames));
```

```
    "x_Time_s_"    "Altitude_m_"    "VerticalVelocity_m_s_"    "TotalVelocity_m_s_"    "LateralAcceleration_m_s__"
```

```
%% HELPER: find a column by common name patterns (case-insensitive)
findCol = @(patterns) find( ...
    cellfun(@(v) any(contains(lower(v), patterns)), T.Properties.VariableNames), ...
    1, 'first');

% Locate key columns
altCol   = findCol(["alt", "height"]);                          % altitude / height
velCol   = findCol(["vel", "velocity"]);                        % (rocket) velocity
massCol  = findCol(["mass"]);                                   % mass
timeCol  = findCol(["time", "sec", "t"]);                       % optional time

% Acceleration (generic)
accCol   = findCol(["accel", "acceleration", "g-force", "gforce"]);

% Lateral acceleration (Ay)
latAccCol = findCol(["lateral", "side", "sideways", "ay", "a_y", "accy", "acc y",
"accel y", "acceleration y", "y-acc"]);

% Angle of Attack (AoA)
aoaCol = findCol(["angle of attack", "aoa", "alpha", "a_o_a"]);

% Wind velocity (scalar wind speed)
windCol = findCol(["wind speed", "wind velocity", "windspeed", "wind"]);

% OPTIONAL: wind components (if your export includes them)
windXCol = findCol(["wind x", "windx", "wind_u", "wind east", "wind_e"]);
windYCol = findCol(["wind y", "windy", "wind_v", "wind north", "wind_n"]);
windZCol = findCol(["wind z", "windz", "wind_w", "wind up", "wind_u_p"]);

% NEW: Mach number
machCol = findCol(["mach", "mach number", "machnumber", "m"]);
```

```
% Validate required columns
if isempty(altCol),  error("Could not find altitude/height column (needs 'alt' or
'height' in name)."); end
if isempty(velCol),  error("Could not find velocity column (needs 'vel' or
'velocity' in name)."); end
if isempty(massCol), error("Could not find mass column (needs 'mass' in name).");
end

altName   = T.Properties.VariableNames{altCol};
velName   = T.Properties.VariableNames{velCol};
massName  = T.Properties.VariableNames{massCol};

alt  = double(T.(altName)(:));
vel  = double(T.(velName)(:));
mass = double(T.(massName)(:));

% Optional time
hasTime = ~isempty(timeCol);
if hasTime
    timeName = T.Properties.VariableNames{timeCol};
    t = double(T.(timeName)(:));
else
    t = NaN(size(alt));
end

%% ---- ACC (generic / axial-ish) ----
hasAcc = ~isempty(accCol);
if hasAcc
    accName = T.Properties.VariableNames{accCol};
    acc = double(T.(accName)(:));
else
    if hasTime
        acc = gradient(vel, t);
        disp("No generic acceleration column found; computed acc = d(vel)/d(t) from
time + velocity.");
    else
        acc = NaN(size(alt));
        disp("No generic acceleration column found and no time column available;
acc set to NaN.");
    end
end

%% ---- LATERAL ACC (Ay) ----
hasLatAcc = ~isempty(latAccCol);
if hasLatAcc
    latAccName = T.Properties.VariableNames{latAccCol};
    ay = double(T.(latAccName)(:));
else
    ay = NaN(size(alt));
```

```
    disp("No lateral acceleration (Ay) column found; ay set to NaN (cannot infer
from speed alone).");
end

%% ---- ANGLE OF ATTACK (AoA) ----
hasAoA = ~isempty(aoaCol);
if hasAoA
    aoaName = T.Properties.VariableNames{aoaCol};
    aoa = double(T.(aoaName)(:));   % deg or rad depending on export
else
    aoa = NaN(size(alt));
    disp("No Angle of Attack column found; aoa set to NaN.");
end
```

```
No Angle of Attack column found; aoa set to NaN.
```

```
%% ---- WIND VELOCITY (wind speed) ----
hasWind = ~isempty(windCol);
if hasWind
    windName = T.Properties.VariableNames{windCol};
    wind = double(T.(windName)(:));
else
    wind = NaN(size(alt));
    disp("No wind speed/velocity column found; wind set to NaN.");
end

%% ---- OPTIONAL wind components (if present) ----
hasWindXYZ = ~isempty(windXCol) && ~isempty(windYCol) && ~isempty(windZCol);
if hasWindXYZ
    windX = double(T.(T.Properties.VariableNames{windXCol})(:));
    windY = double(T.(T.Properties.VariableNames{windYCol})(:));
    windZ = double(T.(T.Properties.VariableNames{windZCol})(:));
end

%% ---- MACH NUMBER ----
hasMach = ~isempty(machCol);
if hasMach
    machName = T.Properties.VariableNames{machCol};
    mach = double(T.(machName)(:));
else
    mach = NaN(size(alt));
    disp("No Mach column found; mach set to NaN.");
end

%% 1) PICK OFF VALUES AT ALTITUDE CLOSEST TO heightRocketAfterRail
[~, idxRail] = min(abs(alt - heightRocketAfterRail));

rail.alt      = alt(idxRail);
rail.vel      = vel(idxRail);
rail.acc      = acc(idxRail);
```

```
rail.ay       = ay(idxRail);
rail.aoa      = aoa(idxRail);
rail.wind     = wind(idxRail);
rail.mach     = mach(idxRail);          % NEW
rail.mass     = mass(idxRail);
rail.idx      = idxRail;
rail.time     = t(idxRail);

%% 2) APOGEE (MAX ALTITUDE) + VALUES AT THAT ROW
clear apogee
apogee = struct();

[apogeeAlt, idxApo] = max(alt);

apogee.alt  = apogeeAlt;
apogee.idx  = idxApo;
apogee.vel  = vel(idxApo);
apogee.acc  = acc(idxApo);
apogee.ay   = ay(idxApo);
apogee.aoa  = aoa(idxApo);
apogee.wind = wind(idxApo);
apogee.mach = mach(idxApo);             % NEW
apogee.mass = mass(idxApo);
apogee.time = t(idxApo);

%% 3) "MAX Q" PROXY: LARGEST SPEED (MAX |VELOCITY|)
clear maxQ
maxQ = struct();

[~, idxMaxV] = max(abs(vel));
maxQ.idx  = idxMaxV;
maxQ.alt  = alt(idxMaxV);
maxQ.vel  = vel(idxMaxV);
maxQ.acc  = acc(idxMaxV);
maxQ.ay   = ay(idxMaxV);
maxQ.aoa  = aoa(idxMaxV);
maxQ.wind = wind(idxMaxV);
maxQ.mach = mach(idxMaxV);              % NEW
maxQ.mass = mass(idxMaxV);
maxQ.time = t(idxMaxV);

%% PRINT RESULTS
fprintf("\n=== Rail-exit height lookup ===\n");
```

```
=== Rail-exit height lookup ===
```

```
fprintf("Requested heightRocketAfterRail: %.6g\n", heightRocketAfterRail);
```

```
Requested heightRocketAfterRail: 8.35508
```

```
fprintf("Closest altitude in CSV:         %.6g (row %d)\n", rail.alt, rail.idx);
```

```
Closest altitude in CSV:        8.416 (row 38)
```

```
fprintf("Velocity at that row:            %.6g\n", rail.vel);
```

```
Velocity at that row:           35.075
```

```
fprintf("Acceleration (acc) at that row:  %.6g\n", rail.acc);
```

```
Acceleration (acc) at that row:  0.109
```

```
fprintf("Lateral accel (ay) at that row:  %.6g\n", rail.ay);
```

```
Lateral accel (ay) at that row:  0.109
```

```
fprintf("AoA at that row:                 %.6g\n", rail.aoa);
```

```
AoA at that row:                NaN
```

```
fprintf("Wind speed at that row:          %.6g\n", rail.wind);
```

```
Wind speed at that row:         2.263
```

```
fprintf("Mach at that row:                %.6g\n", rail.mach);     % NEW
```

```
Mach at that row:               0.516
```

```
fprintf("Mass at that row:                %.6g\n", rail.mass);
```

```
Mass at that row:               37820
```

```
if hasTime
    fprintf("Time at that row:                %.6g\n", rail.time);
end
```

```
Time at that row:               0.516
```

```
fprintf("\n=== Apogee ===\n");
```

```
=== Apogee ===
```

```
fprintf("Apogee altitude (max alt):        %.6g (row %d)\n", apogee.alt,
apogee.idx);
```

```
Apogee altitude (max alt):       4315.11 (row 622)
```

```
fprintf("Velocity at apogee row:           %.6g\n", apogee.vel);
```

```
Velocity at apogee row:          0.181
```

```
fprintf("Acceleration (acc) at apogee row: %.6g\n", apogee.acc);
```

```
Acceleration (acc) at apogee row: 0.102
```

```
fprintf("Lateral accel (ay) at apogee row: %.6g\n", apogee.ay);
```

```
Lateral accel (ay) at apogee row: 0.102
```

```
fprintf("AoA at apogee row:                %.6g\n", apogee.aoa);
```

```
AoA at apogee row:               NaN
```

```
fprintf("Wind speed at apogee row:         %.6g\n", apogee.wind);
```

```
Wind speed at apogee row:        1.767
```

```
fprintf("Mach at apogee row:               %.6g\n", apogee.mach);  % NEW
```

```
Mach at apogee row:              29.376
```

```
fprintf("Mass at apogee row:               %.6g\n", apogee.mass);
```

```
Mass at apogee row:              29990
```

```
if hasTime
    fprintf("Time at apogee row:               %.6g\n", apogee.time);
end
```

```
Time at apogee row:              29.376
```

```

fprintf("\n=== Max-Q proxy (max speed = max |velocity|) ===\n");
```

```
=== Max-Q proxy (max speed = max |velocity|) ===
```

```
fprintf("Altitude at max speed row:        %.6g (row %d)\n", maxQ.alt, maxQ.idx);
```

```
Altitude at max speed row:       1068.82 (row 145)
```

```
fprintf("Velocity at max speed row:        %.6g\n", maxQ.vel);
```

```
Velocity at max speed row:       331.825
```

```
fprintf("Speed magnitude (|v|):            %.6g\n", abs(maxQ.vel));
```

```
Speed magnitude (|v|):           331.825
```

```
fprintf("Acceleration (acc) at max speed:  %.6g\n", maxQ.acc);
```

```
Acceleration (acc) at max speed:  0.929
```

```
fprintf("Lateral accel (ay) at max speed:  %.6g\n", maxQ.ay);
```

```
Lateral accel (ay) at max speed:  0.929
```

```
fprintf("AoA at max speed row:             %.6g\n", maxQ.aoa);
```

```
AoA at max speed row:            NaN
```

```
fprintf("Wind speed at max speed row:      %.6g\n", maxQ.wind);
```

```
Wind speed at max speed row:     2.107
```

```
fprintf("Mach at max speed row:            %.6g\n", maxQ.mach);    % NEW
```

```
Mach at max speed row:           5.611
```

```
fprintf("Mass at max speed row:            %.6g\n", maxQ.mass);
```

```
Mass at max speed row:           30170
```

```
if hasTime
    fprintf("Time at max speed row:            %.6g\n", maxQ.time);
end
```

```
Time at max speed row:           5.611
```

```
%% OPTIONAL: workspace variables
windAfterRail = rail.wind;
apogeeWind    = apogee.wind;
maxQWind      = maxQ.wind;

machAfterRail = rail.mach;     % NEW
apogeeMach    = apogee.mach;   % NEW
maxQMach      = maxQ.mach;     % NEW
```

As well as the wind speed at 10 meters (as this is already given to us by the Salt Creek-Prineville station):

https://raws.dri.edu/cgi-bin/rawMAIN.pl?orOSAL

After Rail: Calculations for getting 'distance in gust'.

From the RAWS page at the Salt Creek-Prinveville Oregon, the max gust between 3/1/25 - 6/31/25 was **22.8 m/s - measured at 10 meters above ground.**

Extrapolate the math for the 8.4 meters from the wind power law. Cited as located here: https://csl.noaa.gov/projects/lamar/windshearformula.html

Assuming the shear exponent(alpha) is equal to 0.2 .

$$V_m = V_{ref} * (\frac{z}{z_{ref}})^{\alpha}$$

$$V_m = 22.8 * (\frac{8.4}{10})^{0.2}$$

```
% vm = 22.8*(heightRocketAfterRail/10)^.2
```

vm = 21.9951

For dm, use general NASA guidelines..

https://ntrs.nasa.gov/api/citations/19990008476/downloads/19990008476.pdf?utm_source=chatgpt.com

where

$$d = Vt_{vehicle}$$

Pick, conservatively, t_g = 0.5s . for the full gust length. so, from this gust time length, you can calculate a gust distance, as we are looking at the time it takes to travel between the launch rail, and an unspecified altitude at 0.5s after the rocket leaves the rail

```
%% ===== GUST WINDOW FROM RAIL EXIT (ROBUST + DENSE MACH + AoA ARRAYS) =====
gustDt = 1; % [s]

%% ---- Robust column finding ----
varNames    = string(T.Properties.VariableNames);
lowerNames  = lower(varNames);

findFirst = @(patterns) find( ...
    arrayfun(@(nm) any(contains(nm, patterns)), lowerNames), ...
    1, 'first');

% TIME
timeCandidates = find(contains(lowerNames, "time") | contains(lowerNames, "sec"));
if isempty(timeCandidates)
    timeCandidates = find( lowerNames=="t" | contains(lowerNames,"t(") |
contains(lowerNames,"t_") | endsWith(lowerNames,"_t") );
end
if isempty(timeCandidates)
    error("No time column detected. Columns are: %s", strjoin(varNames, ", "));
end
timeName = T.Properties.VariableNames{timeCandidates(1)};
fprintf("Detected time column: %s\n", string(timeName));
```

```
Detected time column: x_Time_s_
```

```
% ALT / VEL
altCol = findFirst(["alt","height"]);
velCol = findFirst(["vel","velocity"]);
```

```
if isempty(altCol), error("Could not find altitude/height column."); end
if isempty(velCol), error("Could not find velocity column."); end
altName = T.Properties.VariableNames{altCol};
velName = T.Properties.VariableNames{velCol};

% MACH (optional)
machCol = findFirst(["mach","mach number","machnumber"]);
hasMach = ~isempty(machCol);
if hasMach
    machName = T.Properties.VariableNames{machCol};
    fprintf("Detected Mach column: %s\n", string(machName));
else
    machName = "";
    disp("No Mach column detected; gust Mach will be NaN.");
end
```

```
Detected Mach column: MachNumber___
```

```
% AoA (optional)
aoaCol = findFirst(["angle of attack","aoa","a_o_a","alpha"]);
hasAoA = ~isempty(aoaCol);
if hasAoA
    aoaName = T.Properties.VariableNames{aoaCol};
    fprintf("Detected AoA column: %s\n", string(aoaName));
else
    aoaName = "";
    disp("No AoA column detected; gust AoA will be NaN.");
end
```

```
No AoA column detected; gust AoA will be NaN.
```

```
%% ---- Robust conversion helper ----
toDoubleCol = @(x) local_toDouble(x);

t   = toDoubleCol(T.(timeName));
alt = toDoubleCol(T.(altName));
vel = toDoubleCol(T.(velName));

if hasMach
    machRaw = toDoubleCol(T.(machName));
else
    machRaw = NaN(size(t));
end

if hasAoA
    aoaRaw = toDoubleCol(T.(aoaName));
else
    aoaRaw = NaN(size(t));
end
```

```
% Force columns
t       = t(:);
alt     = alt(:);
vel     = vel(:);
machRaw = machRaw(:);
aoaRaw  = aoaRaw(:);

%% ---- Clean data: ONLY require t/alt/vel to be valid ----
good = isfinite(t) & isfinite(alt) & isfinite(vel);

t       = t(good);
alt     = alt(good);
vel     = vel(good);
machRaw = machRaw(good);
aoaRaw  = aoaRaw(good);

if numel(t) < 2
    error("Not enough valid (t, alt, vel) samples after cleaning.");
end

%% ---- Sort by time + unique time ----
[t, sidx] = sort(t);
alt     = alt(sidx);
vel     = vel(sidx);
machRaw = machRaw(sidx);
aoaRaw  = aoaRaw(sidx);

[tU, uidx] = unique(t, 'stable');
altU     = alt(uidx);
velU     = vel(uidx);
machUraw = machRaw(uidx);
aoaUraw  = aoaRaw(uidx);

%% ---- Gust window endpoints ----
tRail = rail.time;
tEnd  = tRail + gustDt;

%% ---- Alt/V at endpoints ----
altRail = interp1(tU, altU, tRail, "linear", "extrap");
velRail = interp1(tU, velU, tRail, "linear", "extrap");
altEnd  = interp1(tU, altU, tEnd,  "linear", "extrap");
velEnd  = interp1(tU, velU, tEnd,  "linear", "extrap");

%% ---- Build gust time arrays (dense & includes endpoints) ----
mask = (tU >= min(tRail,tEnd)) & (tU <= max(tRail,tEnd));
tSeg = tU(mask);
vSeg = velU(mask);

% add endpoints explicitly
```

```
tSeg = [tRail; tSeg; tEnd];
vSeg = [velRail; vSeg; velEnd];

% sort + unique again
[tSeg, segIdx] = sort(tSeg);
vSeg = vSeg(segIdx);

[tSeg, segUidx] = unique(tSeg, 'stable');
vSeg = vSeg(segUidx);

%% ---- Distance traveled over gust ----
gustDistance = trapz(tSeg, vSeg);
gustDeltaAlt = altEnd - altRail;

%% ---- Mach: interpolate from VALID Mach samples only ----
if hasMach
    machValid = isfinite(machUraw);
    if nnz(machValid) >= 2
        tMach   = tU(machValid);
        machVal = machUraw(machValid);

        machRail = interp1(tMach, machVal, tRail, "linear", "extrap");
        machEnd  = interp1(tMach, machVal, tEnd,  "linear", "extrap");
        machSeg  = interp1(tMach, machVal, tSeg,  "linear", "extrap");

        machMin = min(machSeg(isfinite(machSeg)));
        machMax = max(machSeg(isfinite(machSeg)));
    else
        machRail = NaN; machEnd = NaN;
        machSeg  = NaN(size(tSeg));
        machMin  = NaN; machMax = NaN;
        disp("Mach column exists but has <2 finite samples; cannot interpolate Mach
in gust.");
    end
else
    machRail = NaN; machEnd = NaN;
    machSeg  = NaN(size(tSeg));
    machMin  = NaN; machMax = NaN;
end

%% ---- AoA: interpolate from VALID AoA samples only ----
if hasAoA
    aoaValid = isfinite(aoaUraw);
    if nnz(aoaValid) >= 2
        tAoA   = tU(aoaValid);
        aoaVal = aoaUraw(aoaValid);

        aoaRail = interp1(tAoA, aoaVal, tRail, "linear", "extrap");
        aoaEnd  = interp1(tAoA, aoaVal, tEnd,  "linear", "extrap");
        aoaSeg  = interp1(tAoA, aoaVal, tSeg,  "linear", "extrap");
```

```
        aoaMin = min(aoaSeg(isfinite(aoaSeg)));
        aoaMax = max(aoaSeg(isfinite(aoaSeg)));
    else
        aoaRail = NaN; aoaEnd = NaN;
        aoaSeg  = NaN(size(tSeg));
        aoaMin  = NaN; aoaMax = NaN;
        disp("AoA column exists but has <2 finite samples; cannot interpolate AoA
in gust.");
    end
else
    aoaRail = NaN; aoaEnd = NaN;
    aoaSeg  = NaN(size(tSeg));
    aoaMin  = NaN; aoaMax = NaN;
end

%% ---- Package ----
gust = struct();
gust.dt        = gustDt;
gust.t_start   = tRail;
gust.t_end     = tEnd;

gust.alt_start = altRail;
gust.alt_end   = altEnd;
gust.delta_alt = gustDeltaAlt;

gust.vel_start = velRail;
gust.vel_end   = velEnd;
gust.distance  = gustDistance;

gust.t_array    = tSeg;
gust.vel_array  = vSeg;

gust.mach_array = machSeg;
gust.mach_start = machRail;
gust.mach_end   = machEnd;
gust.mach_min   = machMin;
gust.mach_max   = machMax;

gust.aoa_array  = aoaSeg;     % <-- FULL AoA array inside gust
gust.aoa_start  = aoaRail;
gust.aoa_end    = aoaEnd;
gust.aoa_min    = aoaMin;
gust.aoa_max    = aoaMax;

%% ---- Print ----
fprintf("\n=== Gust travel (dt = %.3f s) ===\n", gust.dt);
```

```
=== Gust travel (dt = 1.000 s) ===
```

```
fprintf("Rail exit time:                    %.6g s\n", gust.t_start);
```

```
Rail exit time:                 0.516 s
```

```
fprintf("Gust end time:                     %.6g s\n", gust.t_end);
```

```
Gust end time:                  1.516 s
```

```
fprintf("Altitude at rail (interp/extr):  %.6g\n", gust.alt_start);
```

```
Altitude at rail (interp/extr):  8.416
```

```
fprintf("Altitude at end (interp/extr):   %.6g\n", gust.alt_end);
```

```
Altitude at end (interp/extr):   79.929
```

```
fprintf("Altitude change over gust:       %.6g\n", gust.delta_alt);
```

```
Altitude change over gust:       71.513
```

```
fprintf("Distance traveled over gust:     %.6g\n", gust.distance);
```

```
Distance traveled over gust:     71.5131
```

```

fprintf("Mach at rail:                    %.6g\n", gust.mach_start);
```

```
Mach at rail:                   0.105
```

```
fprintf("Mach at end:                     %.6g\n", gust.mach_end);
```

```
Mach at end:                    0.323
```

```
fprintf("Mach min/max in gust:            %.6g / %.6g\n", gust.mach_min,
gust.mach_max);
```

```
Mach min/max in gust:           0.105 / 0.323
```

```

fprintf("AoA at rail:                     %.6g\n", gust.aoa_start);
```

```
AoA at rail:                    NaN
```

```
fprintf("AoA at end:                      %.6g\n", gust.aoa_end);
```

```
AoA at end:                     NaN
```

```
fprintf("AoA min/max in gust:             %.6g / %.6g\n", gust.aoa_min,
gust.aoa_max);
```

```
AoA min/max in gust:            NaN / NaN
```

```

%% ---- Local helper function ----
function y = local_toDouble(x)
    if isnumeric(x)
        y = double(x(:));
```

```
        return;
    end
    s = string(x(:));
    s = regexprep(s, "[^0-9eE\+\-\.]", ""); % keep digits, +/-, decimal, e/E
    y = str2double(s);
end
```

After Rail: Final Calculation of Vg(x) (WIP)

Calculate Vg(x) after rail

For the rocket, since $x = Vt$; and $t = x/V$ where V is velocity of launch vehicle...

$$V_g(x) = \frac{V_m}{2}(1 - cos(\frac{\pi d}{d_m})) \qquad for\ \ 0 \leq x \leq 2d_m$$

$V_m(d)$ - gust magnitude (ft/s or m/s)

U_{ds} - design gust velocity (set by altitude & category)

d - Distance flown into the gust

d_m - Gust gradient length (half-length of gust)

Calculate range of values for d, as well as get the arrays for corresponding time stamps:

below for function for time:

Max Q MIL-F-8785B

Estimates for Max Q gusts are here:

https://ntrs.nasa.gov/api/citations/20200000867/downloads/20200000867.pdf

$$V_g(x) = \frac{V_m}{2}(1 - cos(\frac{\pi d}{d_m})) \qquad for\ \ 0 \leq x \leq 2d_m$$

$V_m(d)$ - gust magnitude (ft/s or m/s)

V_m - design gust velocity (set by altitude & category)

d - Distance flown into the gust

d_m - Gust gradient length (half-length of gust)

Note that for design gust velocity (from NASA):

$$\omega_{gust} \sim \frac{\pi V}{H}$$

```
fprintf("Altitude at max speed row: %.6g (row %d)\n", maxQ.alt, maxQ.idx);
```

```
Altitude at max speed row: 1068.82 (row 145)
```

Table 3.2.5-1. Discrete Longitudinal Gust Magnitude (m/s) as a Function of Altitude (km) and Gust Half-width d_m (m) Based on Moderate Turbulence

Gust Magnitude (m/s)										
Alt. (km)	Gust Half-Width (m)									
	30.0	60.0	90.0	120.0	150.0	180.0	210.0	240.0	270.0	300.0
1.0	1.12	1.56	1.87	2.13	2.34	2.52	2.68	2.81	2.94	3.05
2.0	1.08	1.50	1.81	2.05	2.26	2.44	2.59	2.73	2.85	2.96
4.0	1.24	1.73	2.09	2.39	2.63	2.84	3.03	3.19	3.34	3.48
6.0	1.30	1.81	2.19	2.49	2.75	2.97	3.16	3.34	3.49	3.63
8.0	1.31	1.83	2.21	2.51	2.77	3.00	3.19	3.37	3.52	3.67
10.0	1.25	1.75	2.12	2.42	2.67	2.89	3.09	3.27	3.42	3.57
12.0	1.15	1.62	1.96	2.25	2.49	2.71	2.90	3.08	3.24	3.39
14.0	0.98	1.38	1.68	1.93	2.14	2.34	2.51	2.67	2.82	2.95
16.0	0.83	1.17	1.43	1.64	1.83	1.99	2.14	2.28	2.41	2.53
18.0	0.62	0.88	1.07	1.23	1.37	1.50	1.62	1.72	1.82	1.91
20.0	0.48	0.68	0.84	0.96	1.08	1.18	1.27	1.35	1.43	1.51

Choose a shorter gust half-width for a more conservative estimate - in this case, the smallest denomination is `30.0m` as shown in the table above. Also pick the 1 km gust maginitude, 1.12m/s

```
%user inputs
dmMQ = 30;
vm = 1.12
```

```
vm =
1.1200
```

Setting the range of values for the altitude(according to the decided `30.0m` half width, as well as distance into gust):

```
function [gust, gustArray] = buildGustFromAltRange(tU, altU, velU, altRange_m, N)
% buildGustFromAltRange
% Creates gust time/alt/distance-into-gust arrays using an altitude window.
%
% Inputs
%   tU, altU, velU   : time (s), altitude (m), velocity (m/s); must be same length,
sorted, unique in time
%   altRange_m       : [alt_start, alt_end] in meters (order doesn't matter)
```

```
%   N                : number of sample points in the gust window (default 200)
%
% Outputs
%   gust             : struct with t_start, t_end, alt_start, alt_end, distance
%   gustArray        : numeric array [time, altitude, dist_into_gust, frac]

    if nargin < 5 || isempty(N)
        N = 200;
    end
    if numel(altRange_m) ~= 2
        error("altRange_m must be a 2-element vector: [alt_start, alt_end] in
meters.");
    end

    % --- ensure ascending alt range
    alt0 = min(altRange_m);
    alt1 = max(altRange_m);

    % --- quick checks
    if any(isnan(tU)) || any(isnan(altU)) || any(isnan(velU))
        error("Inputs contain NaNs. Clean your arrays first.");
    end

    % --- find times when altitude crosses alt0 and alt1
    % assumes altU is mostly monotonic in the segment you're targeting (ascent).
    % If your sim includes descent, we restrict to the first time it hits each
altitude.
    t_start = interpTimeAtAlt(tU, altU, alt0);
    t_end   = interpTimeAtAlt(tU, altU, alt1);

    if isempty(t_start) || isempty(t_end)
        error("Could not find both altitude crossings within the provided data
range.");
    end
    if t_end <= t_start
        error("Computed t_end <= t_start. Check that your altitude range is on the
ascent portion.");
    end

    % --- sample times across the window
    tSamples = linspace(t_start, t_end, N).';

    % --- interpolate altitude & velocity at those times
    altSamples = interp1(tU, altU, tSamples, "linear", "extrap");
    velSamples = interp1(tU, velU, tSamples, "linear", "extrap");

    % --- distance into gust: cumulative integral of v dt from t_start
    distIntoGust = cumtrapz(tSamples, velSamples);
    gustDistance = distIntoGust(end);
```

```
    % --- fraction through gust
    if abs(gustDistance) > 0
        fracIntoGust = distIntoGust ./ gustDistance;
    else
        fracIntoGust = zeros(size(distIntoGust));
    end

    % clamp
    fracIntoGust = max(0, min(1, fracIntoGust));
    distIntoGust = max(0, min(gustDistance, distIntoGust));

    % --- outputs
    gust = struct();
    gust.t_start   = t_start;
    gust.t_end     = t_end;
    gust.alt_start = alt0;
    gust.alt_end   = alt1;
    gust.distance  = gustDistance;

    gustProfile = struct();
    gustProfile.time           = tSamples;
    gustProfile.alt            = altSamples;
    gustProfile.dist_into_gust = distIntoGust;
    gustProfile.frac           = fracIntoGust;

    gustArray = [tSamples, altSamples, distIntoGust, fracIntoGust];

    % --- sanity prints
    fprintf("\n=== Gust profile arrays created (from altitude range) ===\n");
    fprintf("Alt range: %.3f to %.3f m\n", alt0, alt1);
    fprintf("t range:   %.6g to %.6g s\n", tSamples(1), tSamples(end));
    fprintf("dist end:  %.6g m\n", distIntoGust(end));
end


function t_cross = interpTimeAtAlt(tU, altU, altTarget)
% Returns first time altitude crosses altTarget (linear interpolation).
% If no crossing exists, returns [].

    % Find indices where sign changes or exact hit occurs
    a = altU - altTarget;

    % exact hit
    idxExact = find(a == 0, 1, "first");
    if ~isempty(idxExact)
        t_cross = tU(idxExact);
        return;
    end

    % sign change across adjacent samples
```

```
    idx = find(a(1:end-1) .* a(2:end) < 0, 1, "first");
    if isempty(idx)
        t_cross = [];
        return;
    end

    % linear interpolation between idx and idx+1
    t1 = tU(idx);    t2 = tU(idx+1);
    a1 = a(idx);     a2 = a(idx+1);

    % a(t) linear -> solve for a=0
    t_cross = t1 + (t2 - t1) * (-a1) / (a2 - a1);
end
```

Now needing the vel

Using the code above but for the distances of max Q and such:

```
altRange_m = [maxQ.alt - dmMQ, maxQ.alt + dmMQ];
N = 200;
[gust, gustArray] = buildGustFromAltRange(tU, altU, velU, altRange_m, N)
```

```
=== Gust profile arrays created (from altitude range) ===
Alt range: 1038.818 to 1098.818 m
t range:   5.52055 to 5.70142 s
dist end:  59.9997 m
gust = struct with fields:
      t_start: 5.5206
        t_end: 5.7014
    alt_start: 1.0388e+03
      alt_end: 1.0988e+03
     distance: 59.9997
gustArray = 200×4
10^3 ×
    0.0055    1.0388         0         0
    0.0055    1.0391    0.0003    0.0000
    0.0055    1.0394    0.0006    0.0000
    0.0055    1.0397    0.0009    0.0000
    0.0055    1.0400    0.0012    0.0000
    0.0055    1.0403    0.0015    0.0000
    0.0055    1.0406    0.0018    0.0000
    0.0055    1.0409    0.0021    0.0000
    0.0055    1.0412    0.0024    0.0000
    0.0055    1.0415    0.0027    0.0000
    0.0055    1.0418    0.0030    0.0001
    0.0055    1.0421    0.0033    0.0001
    0.0055    1.0424    0.0036    0.0001
    0.0055    1.0427    0.0039    0.0001
    0.0055    1.0430    0.0042    0.0001
      ⋮
```

```
% buildGustFromAltRange
% Creates gust time/alt/distance-into-gust arrays using an altitude window.
%
% Inputs
```

```
%   tU, altU, velU   : time (s), altitude (m), velocity (m/s); must be same length,
sorted, unique in time
%   altRange_m       : [alt_start, alt_end] in meters (order doesn't matter)
%   N                : number of sample points in the gust window (default 200)
%
% Outputs
%   gust             : struct with t_start, t_end, alt_start, alt_end, distance
%   gustProfile      : struct with .time, .alt, .dist_into_gust, .frac
%   gustArray        : numeric array [time, altitude, dist_into_gust, frac]
```

Now, I need the velocity of the vehicle in the gust:

```
%% Gust time bounds (already known)
t1 = gust.t_start;
t2 = gust.t_end;

%% Generate 200 evenly spaced time samples inside gust
t_query = linspace(t1, t2, 200);

%% Ensure time vector is monotonic + unique
[t_u, iu] = unique(t, "stable");
vel_u = vel(iu);

%% Interpolate velocity at gust times
vel_query = interp1(t_u, vel_u, t_query, "linear", "extrap");

%% Store results
vehicleVelTimeGustCOS = table(t_query(:), vel_query(:), ...
    'VariableNames', ["Time","Velocity"]);

disp(vehicleVelTimeGustCOS)
```

Time	Velocity
5.5206	331.49
5.5215	331.49
5.5224	331.5
5.5233	331.5
5.5242	331.51
5.5251	331.51
5.526	331.52
5.5269	331.52
5.5278	331.53
5.5287	331.53
5.5296	331.54
5.5306	331.54
5.5315	331.55
5.5324	331.55
5.5333	331.56
5.5342	331.56
5.5351	331.57
5.536	331.57
5.5369	331.58
5.5378	331.58

5.5387	331.59
5.5396	331.59
5.5405	331.6
5.5415	331.6
5.5424	331.6
5.5433	331.61
5.5442	331.61
5.5451	331.62
5.546	331.62
5.5469	331.63
5.5478	331.63
5.5487	331.64
5.5496	331.64
5.5505	331.65
5.5515	331.65
5.5524	331.66
5.5533	331.66
5.5542	331.67
5.5551	331.67
5.556	331.68
5.5569	331.68
5.5578	331.69
5.5587	331.69
5.5596	331.7
5.5605	331.7
5.5615	331.7
5.5624	331.71
5.5633	331.71
5.5642	331.71
5.5651	331.71
5.566	331.72
5.5669	331.72
5.5678	331.72
5.5687	331.72
5.5696	331.72
5.5705	331.73
5.5715	331.73
5.5724	331.73
5.5733	331.73
5.5742	331.74
5.5751	331.74
5.576	331.74
5.5769	331.74
5.5778	331.74
5.5787	331.75
5.5796	331.75
5.5805	331.75
5.5814	331.75
5.5824	331.76
5.5833	331.76
5.5842	331.76
5.5851	331.76
5.586	331.76
5.5869	331.77
5.5878	331.77
5.5887	331.77
5.5896	331.77
5.5905	331.78
5.5914	331.78
5.5924	331.78
5.5933	331.78
5.5942	331.78
5.5951	331.79
5.596	331.79

5.5969	331.79
5.5978	331.79
5.5987	331.8
5.5996	331.8
5.6005	331.8
5.6014	331.8
5.6024	331.8
5.6033	331.81
5.6042	331.81
5.6051	331.81
5.606	331.81
5.6069	331.81
5.6078	331.82
5.6087	331.82
5.6096	331.82
5.6105	331.82
5.6114	331.82
5.6124	331.82
5.6133	331.82
5.6142	331.82
5.6151	331.82
5.616	331.82
5.6169	331.82
5.6178	331.82
5.6187	331.82
5.6196	331.82
5.6205	331.82
5.6214	331.82
5.6223	331.82
5.6233	331.82
5.6242	331.82
5.6251	331.82
5.626	331.82
5.6269	331.82
5.6278	331.82
5.6287	331.82
5.6296	331.82
5.6305	331.82
5.6314	331.82
5.6323	331.82
5.6333	331.82
5.6342	331.82
5.6351	331.82
5.636	331.82
5.6369	331.82
5.6378	331.81
5.6387	331.81
5.6396	331.81
5.6405	331.81
5.6414	331.81
5.6423	331.81
5.6433	331.81
5.6442	331.81
5.6451	331.81
5.646	331.81
5.6469	331.81
5.6478	331.81
5.6487	331.81
5.6496	331.81
5.6505	331.81
5.6514	331.81
5.6523	331.81
5.6532	331.81
5.6542	331.81

```
    5.6551     331.81
     5.656     331.81
    5.6569     331.81
    5.6578     331.81
    5.6587     331.81
    5.6596     331.81
    5.6605     331.81
    5.6614      331.8
    5.6623      331.8
    5.6632      331.8
    5.6642      331.8
    5.6651     331.79
     5.666     331.79
    5.6669     331.79
    5.6678     331.78
    5.6687     331.78
    5.6696     331.78
    5.6705     331.78
    5.6714     331.77
    5.6723     331.77
    5.6732     331.77
    5.6742     331.76
    5.6751     331.76
     5.676     331.76
    5.6769     331.76
    5.6778     331.75
    5.6787     331.75
    5.6796     331.75
    5.6805     331.74
    5.6814     331.74
    5.6823     331.74
    5.6832     331.74
    5.6842     331.73
    5.6851     331.73
     5.686     331.73
    5.6869     331.72
    5.6878     331.72
    5.6887     331.72
    5.6896     331.72
    5.6905     331.71
    5.6914     331.71
    5.6923     331.71
    5.6932      331.7
    5.6941      331.7
    5.6951      331.7
     5.696     331.69
    5.6969     331.69
    5.6978     331.69
    5.6987     331.69
    5.6996     331.68
    5.7005     331.68
    5.7014     331.68
```

As well as the lateral acceleration:

```
%% Gust time bounds
t1 = gust.t_start;
t2 = gust.t_end;

%% Generate 200 evenly spaced time samples inside gust
```

```
t_query = linspace(t1, t2, 200);

%% Ensure time vector is monotonic + unique
[t_u, iu] = unique(t, "stable");
acc_u = acc(iu);   % lateral accel aligned with time

%% Interpolate lateral acceleration at gust times
acc_query = interp1(t_u, acc_u, t_query, "linear", "extrap");

%% Store results
vehicleLAccTimeGustCOS = table(t_query(:), acc_query(:), ...
    'VariableNames', ["Time","LateralAcceleration"]);

disp(vehicleLAccTimeGustCOS)
```

Time	LateralAcceleration
5.5206	0.80879
5.5215	0.80106
5.5224	0.79334
5.5233	0.78561
5.5242	0.77789
5.5251	0.77016
5.526	0.76244
5.5269	0.75471
5.5278	0.74699
5.5287	0.73926
5.5296	0.73153
5.5306	0.72381
5.5315	0.71608
5.5324	0.70836
5.5333	0.70063
5.5342	0.69291
5.5351	0.68518
5.536	0.67746
5.5369	0.66973
5.5378	0.66201
5.5387	0.65428
5.5396	0.64655
5.5405	0.63883
5.5415	0.6311
5.5424	0.62338
5.5433	0.61565
5.5442	0.60793
5.5451	0.6002
5.546	0.59248
5.5469	0.58475
5.5478	0.57703
5.5487	0.5693
5.5496	0.56158
5.5505	0.55385
5.5515	0.54612
5.5524	0.5384
5.5533	0.53067
5.5542	0.52295
5.5551	0.51522
5.556	0.5075
5.5569	0.49977
5.5578	0.49205

5.5587	0.48432
5.5596	0.4766
5.5605	0.46887
5.5615	0.46349
5.5624	0.46045
5.5633	0.45741
5.5642	0.45438
5.5651	0.45134
5.566	0.44831
5.5669	0.44527
5.5678	0.44224
5.5687	0.4392
5.5696	0.43616
5.5705	0.43313
5.5715	0.43009
5.5724	0.42706
5.5733	0.42402
5.5742	0.42099
5.5751	0.41795
5.576	0.41491
5.5769	0.41188
5.5778	0.40884
5.5787	0.40581
5.5796	0.40277
5.5805	0.39974
5.5814	0.3967
5.5824	0.39367
5.5833	0.39063
5.5842	0.38759
5.5851	0.38456
5.586	0.38152
5.5869	0.37849
5.5878	0.37545
5.5887	0.37242
5.5896	0.36938
5.5905	0.36634
5.5914	0.36331
5.5924	0.36027
5.5933	0.35724
5.5942	0.3542
5.5951	0.35117
5.596	0.34813
5.5969	0.3451
5.5978	0.34206
5.5987	0.33902
5.5996	0.33599
5.6005	0.33295
5.6014	0.32992
5.6024	0.32688
5.6033	0.32385
5.6042	0.32081
5.6051	0.31777
5.606	0.31474
5.6069	0.3117
5.6078	0.30867
5.6087	0.30563
5.6096	0.3026
5.6105	0.29956
5.6114	0.29939
5.6124	0.30224
5.6133	0.30509
5.6142	0.30795
5.6151	0.3108
5.616	0.31366

5.6169	0.31651
5.6178	0.31936
5.6187	0.32222
5.6196	0.32507
5.6205	0.32793
5.6214	0.33078
5.6223	0.33363
5.6233	0.33649
5.6242	0.33934
5.6251	0.34219
5.626	0.34505
5.6269	0.3479
5.6278	0.35076
5.6287	0.35361
5.6296	0.35646
5.6305	0.35932
5.6314	0.36217
5.6323	0.36503
5.6333	0.36788
5.6342	0.37073
5.6351	0.37359
5.636	0.37644
5.6369	0.3793
5.6378	0.38215
5.6387	0.385
5.6396	0.38786
5.6405	0.39071
5.6414	0.39356
5.6423	0.39642
5.6433	0.39927
5.6442	0.40213
5.6451	0.40498
5.646	0.40783
5.6469	0.41069
5.6478	0.41354
5.6487	0.4164
5.6496	0.41925
5.6505	0.4221
5.6514	0.42496
5.6523	0.42781
5.6532	0.43066
5.6542	0.43352
5.6551	0.43637
5.656	0.43923
5.6569	0.44208
5.6578	0.44493
5.6587	0.44779
5.6596	0.45064
5.6605	0.4535
5.6614	0.45765
5.6623	0.46325
5.6632	0.46884
5.6642	0.47444
5.6651	0.48004
5.666	0.48564
5.6669	0.49124
5.6678	0.49684
5.6687	0.50244
5.6696	0.50804
5.6705	0.51363
5.6714	0.51923
5.6723	0.52483
5.6732	0.53043
5.6742	0.53603

```
    5.6751          0.54163
     5.676          0.54723
    5.6769          0.55282
    5.6778          0.55842
    5.6787          0.56402
    5.6796          0.56962
    5.6805          0.57522
    5.6814          0.58082
    5.6823          0.58642
    5.6832          0.59202
    5.6842          0.59761
    5.6851          0.60321
     5.686          0.60881
    5.6869          0.61441
    5.6878          0.62001
    5.6887          0.62561
    5.6896          0.63121
    5.6905           0.6368
    5.6914           0.6424
    5.6923            0.648
    5.6932           0.6536
    5.6941           0.6592
    5.6951           0.6648
     5.696           0.6704
    5.6969            0.676
    5.6978          0.68159
    5.6987          0.68719
    5.6996          0.69279
    5.7005          0.69839
    5.7014          0.70399
```

As well as the AoA:

```
%% Gust time bounds
t1 = gust.t_start;
t2 = gust.t_end;

%% Generate 200 evenly spaced time samples inside gust
t_query = linspace(t1, t2, 200);

%% Ensure time vector is monotonic + unique
[t_u, iu] = unique(t, "stable");
aoa_u = aoa(iu);

%% Interpolate AoA at gust times
aoa_query = interp1(t_u, aoa_u, t_query, "linear", "extrap");

%% Store results
vehicleAoATimeGustCOS = table(t_query(:), aoa_query(:), ...
    'VariableNames', ["Time","AoA"]);

disp(vehicleAoATimeGustCOS)
```

```
     Time      AoA
    ______     ___

    5.5206     NaN
    5.5215     NaN
```

5.5224	NaN
5.5233	NaN
5.5242	NaN
5.5251	NaN
5.526	NaN
5.5269	NaN
5.5278	NaN
5.5287	NaN
5.5296	NaN
5.5306	NaN
5.5315	NaN
5.5324	NaN
5.5333	NaN
5.5342	NaN
5.5351	NaN
5.536	NaN
5.5369	NaN
5.5378	NaN
5.5387	NaN
5.5396	NaN
5.5405	NaN
5.5415	NaN
5.5424	NaN
5.5433	NaN
5.5442	NaN
5.5451	NaN
5.546	NaN
5.5469	NaN
5.5478	NaN
5.5487	NaN
5.5496	NaN
5.5505	NaN
5.5515	NaN
5.5524	NaN
5.5533	NaN
5.5542	NaN
5.5551	NaN
5.556	NaN
5.5569	NaN
5.5578	NaN
5.5587	NaN
5.5596	NaN
5.5605	NaN
5.5615	NaN
5.5624	NaN
5.5633	NaN
5.5642	NaN
5.5651	NaN
5.566	NaN
5.5669	NaN
5.5678	NaN
5.5687	NaN
5.5696	NaN
5.5705	NaN
5.5715	NaN
5.5724	NaN
5.5733	NaN
5.5742	NaN
5.5751	NaN
5.576	NaN
5.5769	NaN
5.5778	NaN
5.5787	NaN
5.5796	NaN

5.5805	NaN
5.5814	NaN
5.5824	NaN
5.5833	NaN
5.5842	NaN
5.5851	NaN
5.586	NaN
5.5869	NaN
5.5878	NaN
5.5887	NaN
5.5896	NaN
5.5905	NaN
5.5914	NaN
5.5924	NaN
5.5933	NaN
5.5942	NaN
5.5951	NaN
5.596	NaN
5.5969	NaN
5.5978	NaN
5.5987	NaN
5.5996	NaN
5.6005	NaN
5.6014	NaN
5.6024	NaN
5.6033	NaN
5.6042	NaN
5.6051	NaN
5.606	NaN
5.6069	NaN
5.6078	NaN
5.6087	NaN
5.6096	NaN
5.6105	NaN
5.6114	NaN
5.6124	NaN
5.6133	NaN
5.6142	NaN
5.6151	NaN
5.616	NaN
5.6169	NaN
5.6178	NaN
5.6187	NaN
5.6196	NaN
5.6205	NaN
5.6214	NaN
5.6223	NaN
5.6233	NaN
5.6242	NaN
5.6251	NaN
5.626	NaN
5.6269	NaN
5.6278	NaN
5.6287	NaN
5.6296	NaN
5.6305	NaN
5.6314	NaN
5.6323	NaN
5.6333	NaN
5.6342	NaN
5.6351	NaN
5.636	NaN
5.6369	NaN
5.6378	NaN

5.6387	NaN
5.6396	NaN
5.6405	NaN
5.6414	NaN
5.6423	NaN
5.6433	NaN
5.6442	NaN
5.6451	NaN
5.646	NaN
5.6469	NaN
5.6478	NaN
5.6487	NaN
5.6496	NaN
5.6505	NaN
5.6514	NaN
5.6523	NaN
5.6532	NaN
5.6542	NaN
5.6551	NaN
5.656	NaN
5.6569	NaN
5.6578	NaN
5.6587	NaN
5.6596	NaN
5.6605	NaN
5.6614	NaN
5.6623	NaN
5.6632	NaN
5.6642	NaN
5.6651	NaN
5.666	NaN
5.6669	NaN
5.6678	NaN
5.6687	NaN
5.6696	NaN
5.6705	NaN
5.6714	NaN
5.6723	NaN
5.6732	NaN
5.6742	NaN
5.6751	NaN
5.676	NaN
5.6769	NaN
5.6778	NaN
5.6787	NaN
5.6796	NaN
5.6805	NaN
5.6814	NaN
5.6823	NaN
5.6832	NaN
5.6842	NaN
5.6851	NaN
5.686	NaN
5.6869	NaN
5.6878	NaN
5.6887	NaN
5.6896	NaN
5.6905	NaN
5.6914	NaN
5.6923	NaN
5.6932	NaN
5.6941	NaN
5.6951	NaN
5.696	NaN

```
    5.6969    NaN
    5.6978    NaN
    5.6987    NaN
    5.6996    NaN
    5.7005    NaN
    5.7014    NaN
```

As well as the wind and gust velocity:

```
%% Gust time bounds
t1 = gust.t_start;
t2 = gust.t_end;

%% Generate 200 evenly spaced time samples inside gust
t_query = linspace(t1, t2, 200);

%% Ensure time vector is monotonic + unique
[t_u, iu] = unique(t, "stable");
wind_u = wind(iu);

%% Interpolate wind velocity at gust times
wind_query = interp1(t_u, wind_u, t_query, "linear", "extrap");

%% Store results
vehicleWindTimeGustCOS = table(t_query(:), wind_query(:), ...
    'VariableNames', ["Time","WindVelocity"]);

disp(vehicleWindTimeGustCOS)
```

```
     Time      WindVelocity
    ______     ____________

    5.5206        1.9362
    5.5215        1.9378
    5.5224        1.9395
    5.5233        1.9411
    5.5242        1.9427
    5.5251        1.9444
     5.526         1.946
    5.5269        1.9476
    5.5278        1.9493
    5.5287        1.9509
    5.5296        1.9526
    5.5306        1.9542
    5.5315        1.9558
    5.5324        1.9575
    5.5333        1.9591
    5.5342        1.9607
    5.5351        1.9624
     5.536         1.964
    5.5369        1.9656
    5.5378        1.9673
    5.5387        1.9689
    5.5396        1.9706
    5.5405        1.9722
```

5.5415	1.9738
5.5424	1.9755
5.5433	1.9771
5.5442	1.9787
5.5451	1.9804
5.546	1.982
5.5469	1.9836
5.5478	1.9853
5.5487	1.9869
5.5496	1.9885
5.5505	1.9902
5.5515	1.9918
5.5524	1.9935
5.5533	1.9951
5.5542	1.9967
5.5551	1.9984
5.556	2
5.5569	2.0016
5.5578	2.0033
5.5587	2.0049
5.5596	2.0065
5.5605	2.0082
5.5615	2.0098
5.5624	2.0113
5.5633	2.0128
5.5642	2.0143
5.5651	2.0159
5.566	2.0174
5.5669	2.0189
5.5678	2.0205
5.5687	2.022
5.5696	2.0235
5.5705	2.025
5.5715	2.0266
5.5724	2.0281
5.5733	2.0296
5.5742	2.0311
5.5751	2.0327
5.576	2.0342
5.5769	2.0357
5.5778	2.0372
5.5787	2.0388
5.5796	2.0403
5.5805	2.0418
5.5814	2.0434
5.5824	2.0449
5.5833	2.0464
5.5842	2.0479
5.5851	2.0495
5.586	2.051
5.5869	2.0525
5.5878	2.054
5.5887	2.0556
5.5896	2.0571
5.5905	2.0586
5.5914	2.0602
5.5924	2.0617
5.5933	2.0632
5.5942	2.0647
5.5951	2.0663
5.596	2.0678
5.5969	2.0693
5.5978	2.0708
5.5987	2.0724

5.5996	2.0739
5.6005	2.0754
5.6014	2.0769
5.6024	2.0785
5.6033	2.08
5.6042	2.0815
5.6051	2.0831
5.606	2.0846
5.6069	2.0861
5.6078	2.0876
5.6087	2.0892
5.6096	2.0907
5.6105	2.0922
5.6114	2.0932
5.6124	2.0937
5.6133	2.0942
5.6142	2.0947
5.6151	2.0952
5.616	2.0957
5.6169	2.0962
5.6178	2.0967
5.6187	2.0972
5.6196	2.0977
5.6205	2.0981
5.6214	2.0986
5.6223	2.0991
5.6233	2.0996
5.6242	2.1001
5.6251	2.1006
5.626	2.1011
5.6269	2.1016
5.6278	2.1021
5.6287	2.1026
5.6296	2.1031
5.6305	2.1035
5.6314	2.104
5.6323	2.1045
5.6333	2.105
5.6342	2.1055
5.6351	2.106
5.636	2.1065
5.6369	2.107
5.6378	2.1075
5.6387	2.108
5.6396	2.1085
5.6405	2.1089
5.6414	2.1094
5.6423	2.1099
5.6433	2.1104
5.6442	2.1109
5.6451	2.1114
5.646	2.1119
5.6469	2.1124
5.6478	2.1129
5.6487	2.1134
5.6496	2.1139
5.6505	2.1143
5.6514	2.1148
5.6523	2.1153
5.6532	2.1158
5.6542	2.1163
5.6551	2.1168
5.656	2.1173
5.6569	2.1178

```
    5.6578       2.1183
    5.6587       2.1188
    5.6596       2.1193
    5.6605       2.1197
    5.6614       2.1198
    5.6623       2.1194
    5.6632       2.1189
    5.6642       2.1185
    5.6651        2.118
     5.666       2.1176
    5.6669       2.1172
    5.6678       2.1167
    5.6687       2.1163
    5.6696       2.1159
    5.6705       2.1154
    5.6714        2.115
    5.6723       2.1146
    5.6732       2.1141
    5.6742       2.1137
    5.6751       2.1132
     5.676       2.1128
    5.6769       2.1124
    5.6778       2.1119
    5.6787       2.1115
    5.6796       2.1111
    5.6805       2.1106
    5.6814       2.1102
    5.6823       2.1098
    5.6832       2.1093
    5.6842       2.1089
    5.6851       2.1085
     5.686        2.108
    5.6869       2.1076
    5.6878       2.1071
    5.6887       2.1067
    5.6896       2.1063
    5.6905       2.1058
    5.6914       2.1054
    5.6923        2.105
    5.6932       2.1045
    5.6941       2.1041
    5.6951       2.1037
     5.696       2.1032
    5.6969       2.1028
    5.6978       2.1023
    5.6987       2.1019
    5.6996       2.1015
    5.7005        2.101
    5.7014       2.1006
```

Since there is no explicit gust value between 1.0km and 2.0 km, interpolate the values from the `Table 3.2.5-3. Discrete Longitudinal Gust Magnitude (m/s) as a Function of Altitude (km) and Gust Half-width dm (m) Based on Severe Turbulence`

```
%% Discrete Longitudinal Gust Magnitude Table (SEVERE Turbulence)
% Table 3.2.5-3: Discrete Longitudinal Gust Magnitude (m/s) vs Altitude (km)
% and Gust Half-Width d_m (m)
```

```
% Altitude grid (km)
alt_km = [ ...
     1
     2
     4
     6
     8
    10
    12
    14
    16
    18
    20
    25
    30
    35
    40
    45
    50
    55
    60
    65
    70
    75
    80 ];

% Gust half-width grid d_m (m)
dm_m = [30 60 90 120 150 180 210 240 270 300];

% Gust magnitude (m/s): rows = alt_km, cols = dm_m
G = [ ...
    2.65 3.68 4.43 5.03 5.53 5.95 6.32 6.65 6.94 7.20;  %  1 km
    2.84 3.95 4.77 5.42 5.96 6.43 6.84 7.20 7.52 7.81;  %  2 km
    3.81 5.31 6.41 7.30 8.05 8.69 9.26 9.77 10.23 10.64; %  4 km
    4.37 6.09 7.35 8.37 9.23 9.98 10.63 11.21 11.74 12.21; %  6 km
    4.63 6.45 7.79 8.88 9.79 10.58 11.27 11.89 12.44 12.94; %  8 km
    4.34 6.06 7.34 8.37 9.25 10.02 10.70 11.30 11.85 12.36; % 10 km
    3.68 5.16 6.27 7.18 7.96 8.65 9.27 9.83 10.35 10.82; % 12 km
    2.59 3.64 4.44 5.10 5.67 6.18 6.64 7.06 7.45 7.81;  % 14 km
    1.70 2.40 2.92 3.36 3.74 4.08 4.39 4.67 4.94 5.18;  % 16 km
    1.14 1.61 1.97 2.27 2.53 2.76 2.98 3.17 3.35 3.53;  % 18 km
    0.82 1.16 1.42 1.64 1.83 2.00 2.16 2.31 2.44 2.57;  % 20 km
    0.79 1.12 1.36 1.57 1.76 1.92 2.07 2.21 2.35 2.47;  % 25 km
    0.66 0.93 1.14 1.32 1.47 1.61 1.74 1.86 1.97 2.08;  % 30 km
    0.73 1.03 1.26 1.46 1.63 1.79 1.93 2.06 2.18 2.30;  % 35 km
    0.76 1.08 1.32 1.52 1.70 1.86 2.01 2.15 2.28 2.40;  % 40 km
    0.83 1.18 1.44 1.66 1.86 2.03 2.20 2.35 2.49 2.62;  % 45 km
    0.90 1.28 1.56 1.81 2.02 2.21 2.39 2.55 2.71 2.85;  % 50 km
    1.01 1.43 1.76 2.03 2.27 2.48 2.68 2.86 3.04 3.20;  % 55 km
```

```
    1.10 1.56 1.91 2.21 2.47 2.70 2.92 3.12 3.31 3.49;  % 60 km
    1.17 1.65 2.02 2.34 2.61 2.86 3.09 3.30 3.50 3.69;  % 65 km
    1.56 2.21 2.71 3.12 3.49 3.82 4.13 4.42 4.68 4.93;  % 70 km
    1.79 2.53 3.10 3.58 4.01 4.39 4.74 5.07 5.37 5.66;  % 75 km
    2.02 2.86 3.50 4.05 4.52 4.95 5.35 5.72 6.07 6.39   % 80 km
];



%% Build interpolant once
% 'linear' interpolation, 'nearest' extrap outside grid (safer than guessing)
F = griddedInterpolant({alt_km, dm_m}, G, "linear", "nearest");

%% ---- YOU ONLY CHANGE THESE LINES ----
alt_query_km = 1.11468;  % your altitude in km
dm_query_m   = 30;       % gust half-width (m)
% --------------------------------------

gust_mag_ms = F(alt_query_km, dm_query_m);

fprintf("Gust magnitude at alt = %.5f km, d_m = %.1f m: %.4f m/s\n", ...
    alt_query_km, dm_query_m, gust_mag_ms);
```

```
Gust magnitude at alt = 1.11468 km, d_m = 30.0 m: 2.6718 m/s
```

From above:

```
=== Gust profile arrays created (from altitude range) ===

Alt range: 1032.074 to 1092.074 m
```

Using the code above

$$V_g(x) = \frac{V_m}{2}(1 - cos(\frac{\pi d}{d_m})) \qquad for\ \ 0 \leq x \leq 2d_m$$

$V_m(d)$ - gust magnitude (ft/s or m/s)

V_m - design gust velocity (set by altitude & category)

d - Distance flown into the gust

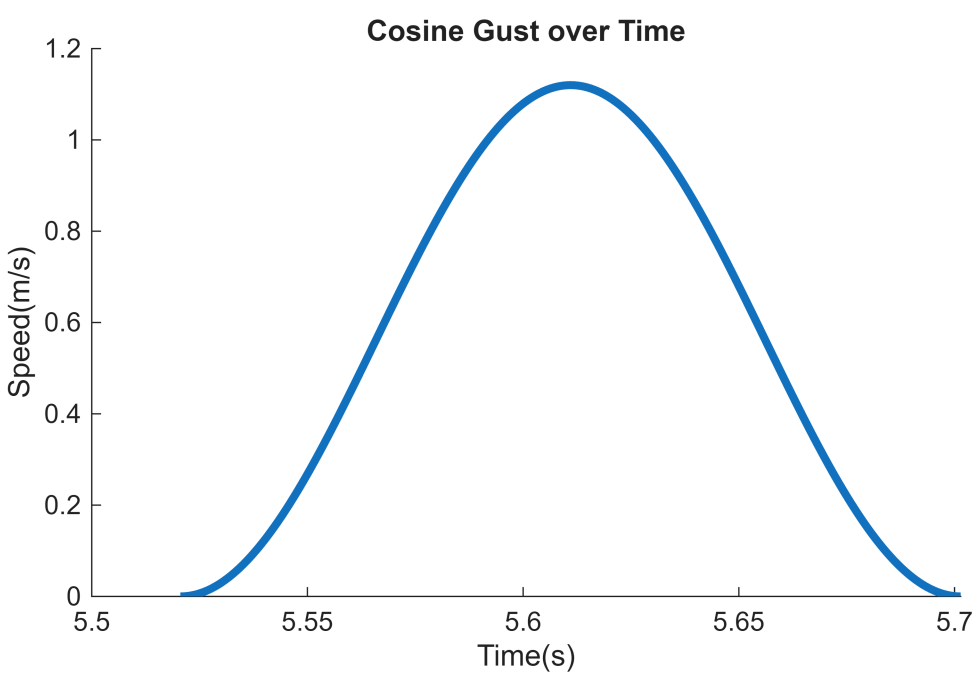
d_m - Gust gradient length (half-length of gust)

```
% Distance into gust
```

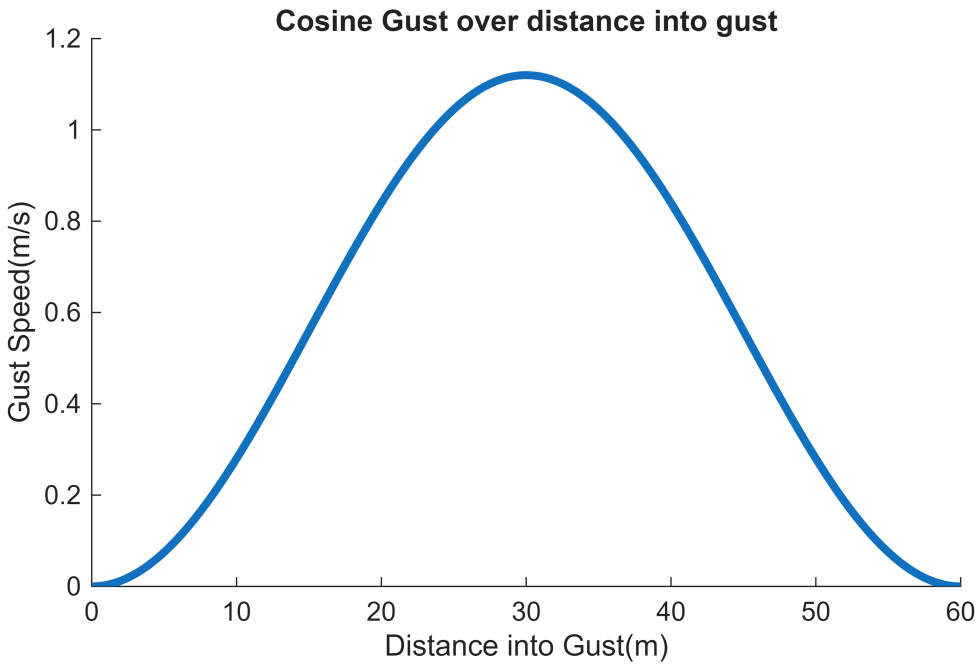
```
s = gustArray(:,3);

% Gust length so peak occurs at launchRailH
dmMQ;

Lgust = gust.distance;          % NOT (gust.alt_end - gust.alt_start)
inside = (s >= 0) & (s <= Lgust);
vg = zeros(size(s));


vg(inside) = (vm/2) .* (1 - cos(pi * s(inside) / dmMQ));

vm = gust_mag_ms;

% vg(inside) = (vm/2) .* (1 - cos(pi * s(inside) / dmMQ));
% equivalently: (vm/2)*(1 - cos(2*pi*s/Lgust))

%
```

```
Alt range: 1032.074 to 1092.074 m
```

```
% Outside gust region vg remains zero
figure;
hold on;
title('Cosine Gust over Time')
xlabel('Time(s)')
ylabel('Speed(m/s)')
plot(gustArray(:,1), vg, 'LineWidth', 3);
```



```
figure;
hold on;
title('Cosine Gust over distance into gust')
xlabel('Distance into Gust(m)')
ylabel('Gust Speed(m/s)')
plot(gustArray(:,3), vg, 'LineWidth', 3);
```




```
figure;
```

Requirements for Estimating CnA for Fins

Note these are the assumptions made by Barrowman:

2.40 GENERAL ASSUMPTIONS

1. The angle-of-attack is very near zero.
2. The flow is steady and irrotational.
3. The vehicle is a rigid body.
4. The nose tip is a sharp point.

From Barrowman's The Practical Calculation of the Aerodynamic Characteristics of Slender Finned Vehicles(1967)

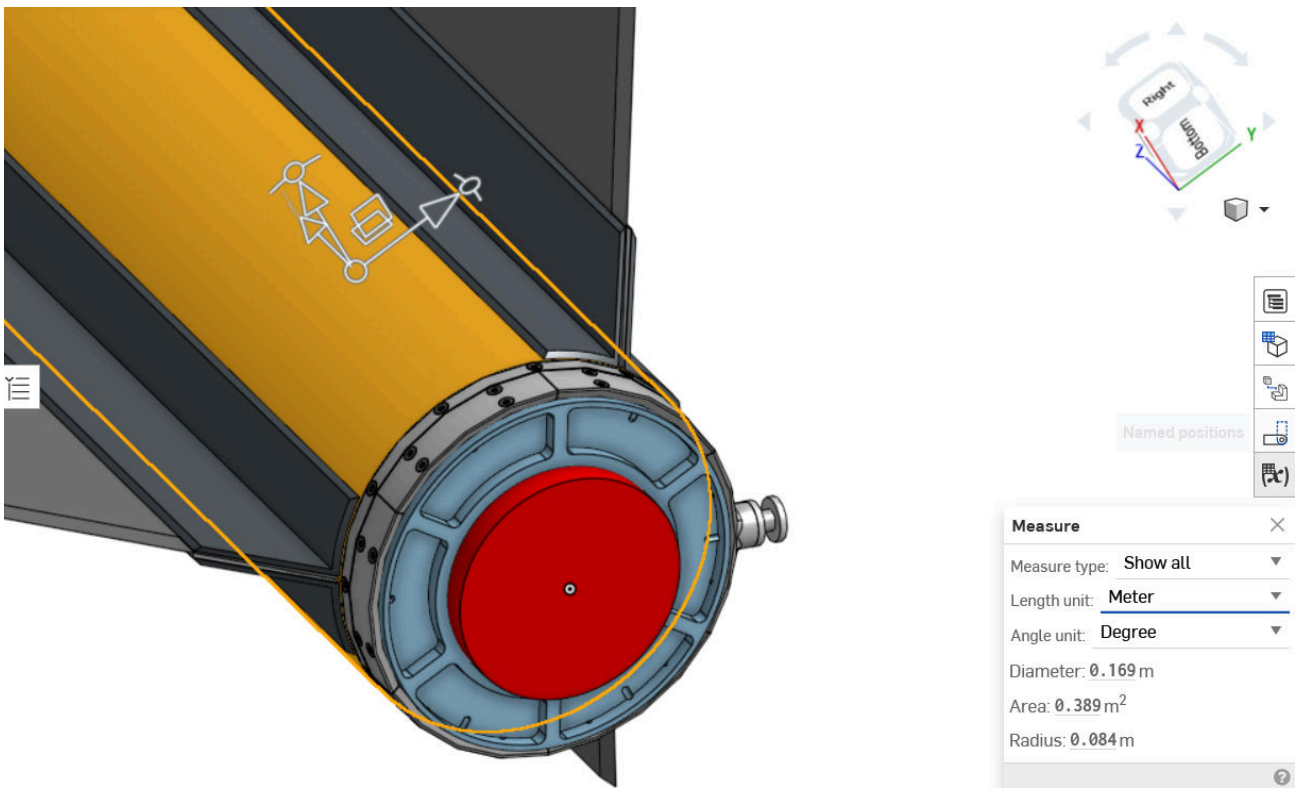
https://ntrs.nasa.gov/api/citations/20010047838/downloads/20010047838.pdf

$$C_{n_\alpha} = \frac{2\pi(A_f/A_r)}{2 + \sqrt{4 + (\frac{\beta AR}{cos\Gamma_c})}}$$

For β in the equation, for C_{n_α}

Easily found values for Cna Calculation

Ar = reference area of rocket body (based off body tube diameter)



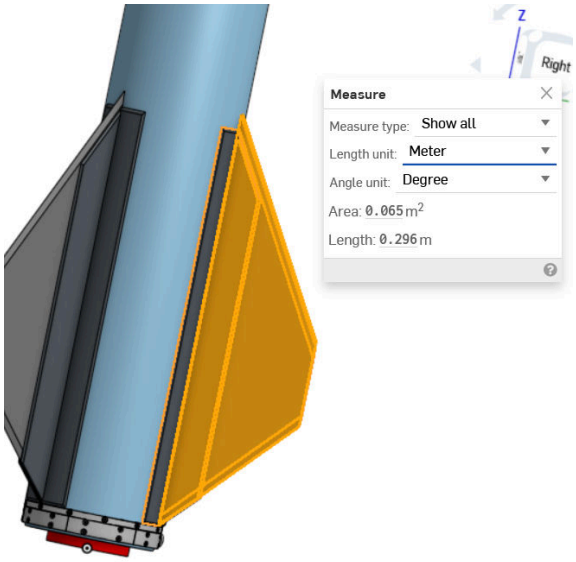
From the figure above:

```
dBodyTube = 0.169
```

```
dBodyTube =
0.1690
```

```
bodyTubeArea = (pi*dBodyTube^2)/4;
```

Af = area of fins

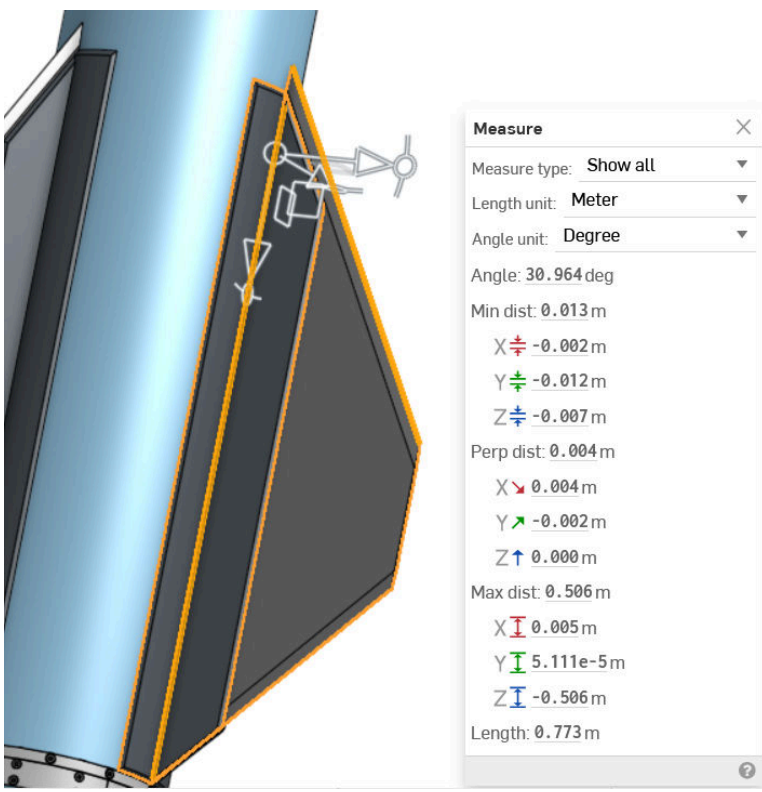


For Af:

From the figure above:

```
finA = 0.042;
```

GAMMA_c - Midchord Line Sweep Angle



The midchord line sweep angle is measured above, at 30.964 degrees.

```
finGammaC = deg2rad(59)
```

```
finGammaC =
1.0297
```

For beta:

https://en.wikipedia.org/wiki/Prandtl%E2%80%93Glauert_transformation?

From above. $\beta = \sqrt{1 - M_\infty^2}$

$M_\infty = freestream\ \ mach\ \ number$

Calculating Beta for Fins CnA

https://en.wikipedia.org/wiki/Freestream?

For simplicity(and speed), only calculating the mach number for the velocity at max Q. Which is:

$$M_\infty = \frac{V_\infty}{c_\infty}$$

$$V_\infty = V_{vehicle} - V_{air}$$

Extrapolating the wind data from the RAWS to get max wind speed at the altitude of max Q:

Assuming the shear exponent(alpha) is equal to 0.2 .

$$V_m = V_{ref} * (\frac{z}{z_{ref}})^\alpha$$

$$V_m = 22.8 * (\frac{8.4}{10})^{0.2}$$

```
maxQ.alt
```

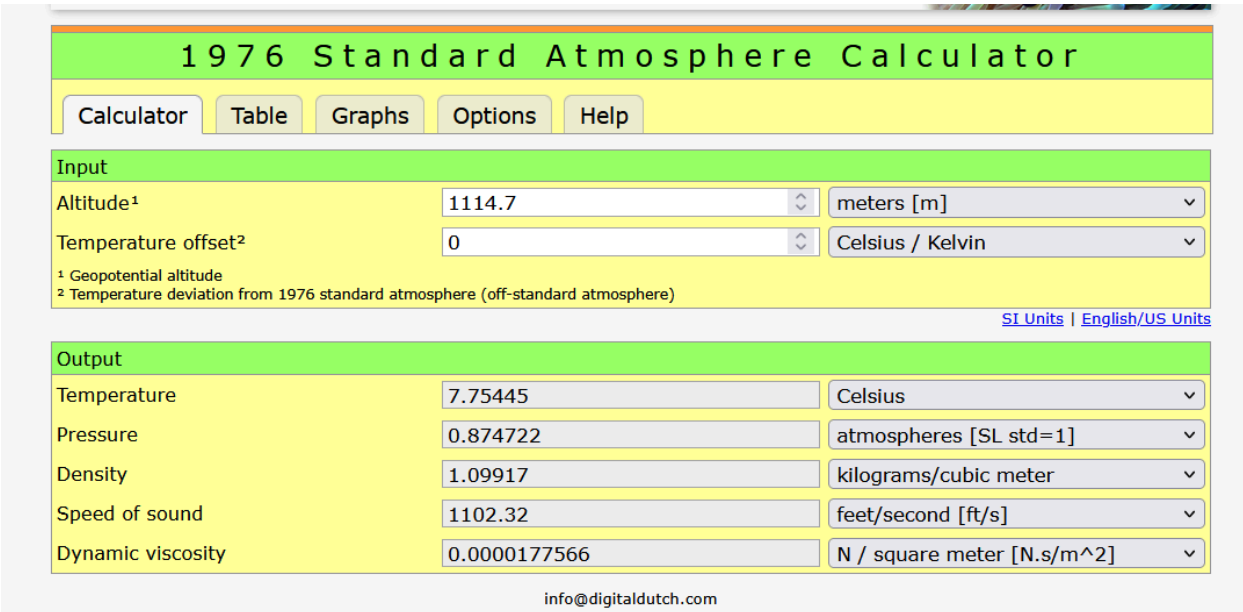
```
ans =
1.0688e+03
```

```
v_WindVehicleMaxQ = 22.8*(maxQ.alt/10)^.2
```

```
v_WindVehicleMaxQ =
58.0384
```

To estimate Cna, get the Speed of Sound from the "Tables of the U.S Standard Atmosphere"(1976) at max Q

https://www.digitaldutch.com/atmoscalc/




```
cforCNA = 1102.32;
```

cforCNA = 320.55 m/s at z = 5km.

Note that from previous discrete gust modeling, V =

```
v_air = (v_WindVehicleMaxQ + max(vm));
```

And then for V_{inf} :

```
v_inf4CNA = abs((v_WindVehicleMaxQ - v_air));

% Note that, I origainlly did v_inf = v_air - V_WindVehicle
```

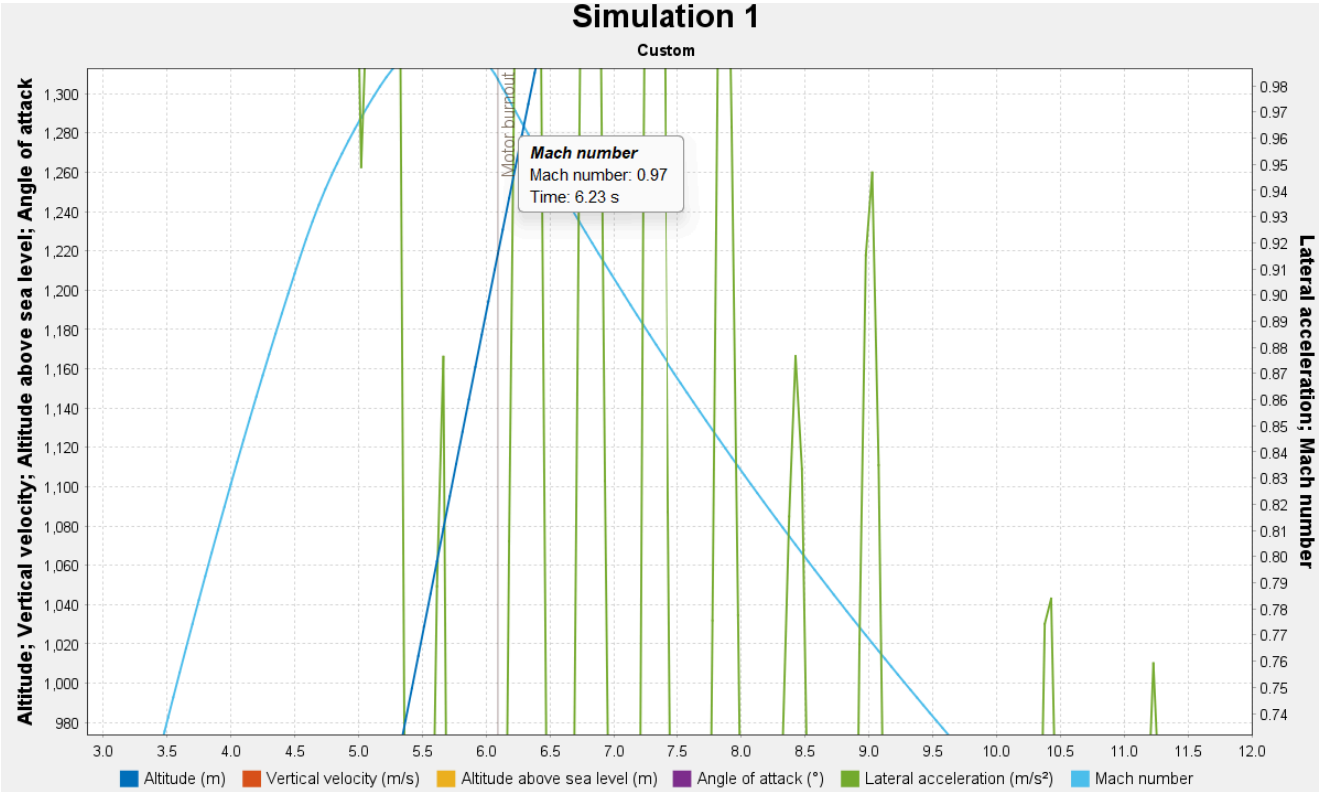
Now that we have v_air and c, can calc freestream mach number:

```
Ma_4CNA = v_inf4CNA/cforCNA
```

```
Ma_4CNA =
0.0024
```

```
% Ma_4CNA = machSeg
```

Instead of the above, due to Peter's feedback on 2/18/26



Recall:

$$\beta = \sqrt{1 - M_\infty^2}$$

$$M_\infty = freestream \;\; mach \;\; number$$

```
beta = sqrt(1-.97^2)
```

```
beta =
0.2431
```

Calculating Aspect Ratio for Fins

From:

https://en.wikipedia.org/wiki/Aspect_ratio_(aeronautics)

Definition [edit]

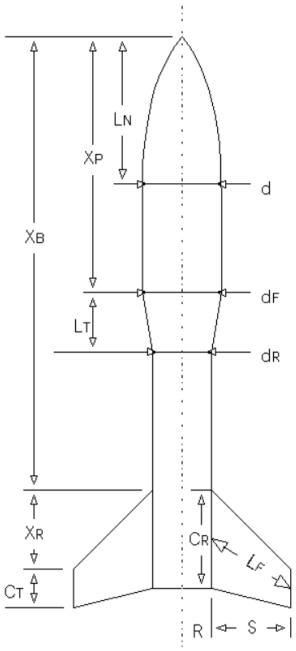
The aspect ratio AR is the ratio of the square of the wingspan b to the projected[3] wing area S,[4][5] which is equal to the ratio of the wingspan b to the standard mean chord SMC:[6]

$$\mathrm{AR} \equiv \frac{b^2}{S} = \frac{b}{\mathrm{SMC}}$$

Note that Root Chord from this equation is

Source that helps understand how to calculate the effective aspect ratio

https://www.rocketmime.com/rockets/Barrowman.html?



Computing Center of Pressure

The Barrowman equations permit you to determine the stability of your rocket by finding the location of the center of pressure (CP). The value computed is the distance from the tip of the rocket's nose to the CP. In order for your rocket to be stable, you would like the CP to be aft of the center of gravity (CG).

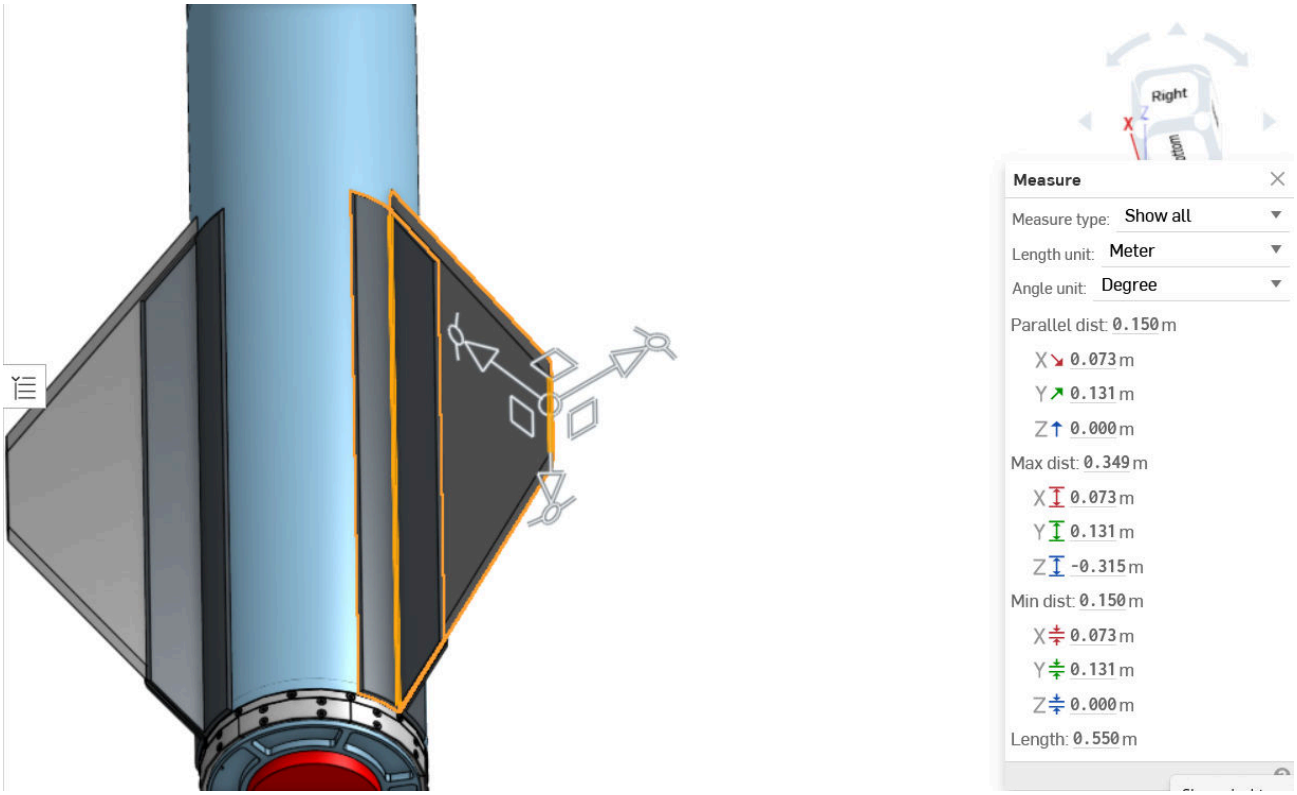
The computation of CP isn't as hard as it looks at first. Check out the spreadsheet example at the bottom of this page.

You can find the CG of your rocket by simply finding the balance point after loading recovery system and motor. (Literally - balance the rocket on your hand - or finger - and that's the CG). You can then measure from the tip of the rocket's nose to the CG. The calculated CP distance should be greater than the measured CG distance by one rocket diameter. This is called "one caliber stability".

Terms in the equations are defined below (and in the diagram):

L_N = length of nose
d = diameter at base of nose
d_F = diameter at front of transition
d_R = diameter at rear of transition
L_T = length of transition
X_P = distance from tip of nose to front of transition
C_R = fin root chord
C_T = fin tip chord
S = fin semispan
L_F = length of fin mid-chord line
R = radius of body at aft end
X_R = distance between fin root leading edge and fin tip leading edge parallel to body
X_B = distance from nose tip to fin root chord leading edge
N = number of fins

OnShape: "LV3.1 Full Assembly": pulled from "Main" Branch 2/12/26:



For span of the fin:

From the figure above:

```
sfins = 0.150 % in meters
```

```
sfins =
0.1500
```

```
% Now that we have all of the fin data we wanted
AR = sfins*finA
```

```
AR =
0.0063
```

Estimating Cna for fins

From Barrowman's The Practical Calculation of the Aerodynamic Characteristics of Slender Finned Vehicles(1967).

https://ntrs.nasa.gov/api/citations/20010047838/downloads/20010047838.pdf

This particular equation uses a beta at ONLY the altitude of the Q, so this may cause the beta to be a more conservative value, and thus the Cna is a more conservative value.

For a singular fin:

$$C_{n_\alpha} = \frac{2\pi(A_f/A_r)AR}{2 + \sqrt{4 + (\frac{\beta AR}{cos\Gamma_c})}}$$

```
finBodR = finA/bodyTubeArea
```

```
finBodR =
1.8723
```

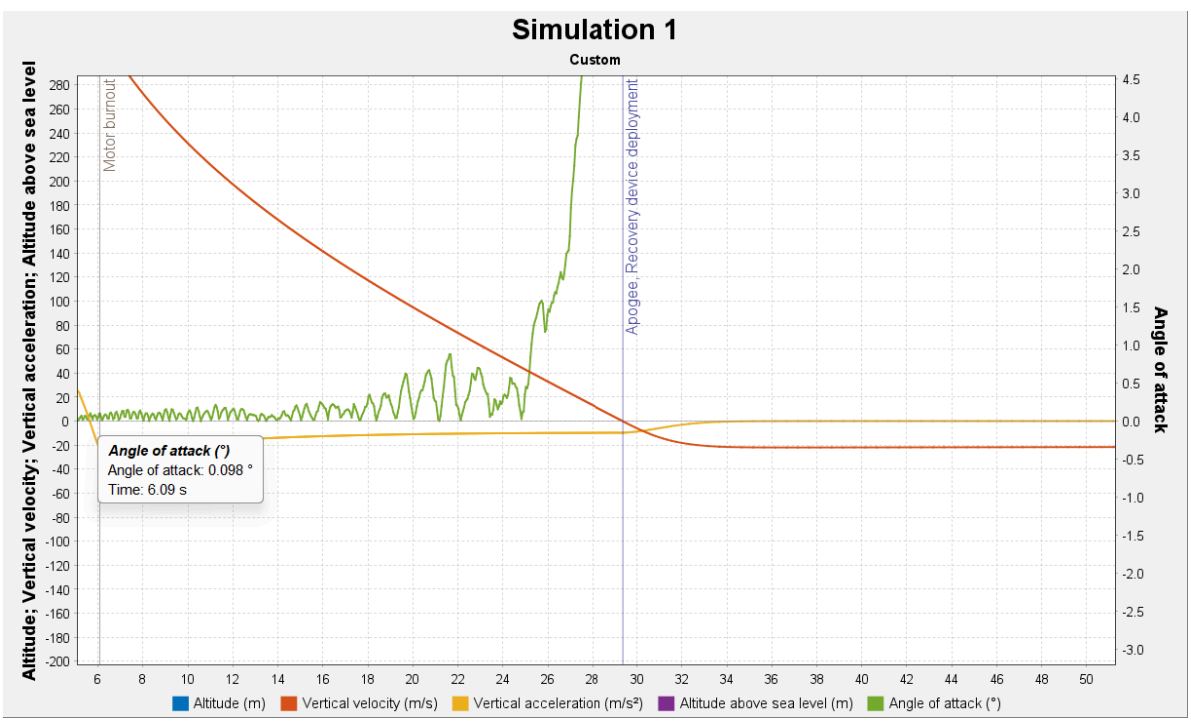
```
finRatInD = (beta*AR)/(cos(finGammaC))
```

```
finRatInD =
0.0030
```

```
fin1_Cna = (2*pi * finBodR*AR)/ (2 + sqrt(4 + finRatInD))
```

```
fin1_Cna =
0.0185
```

In order for the total Cna to equal the summation of fin Cna, the angle of attack must be sufficiently small (< 10 degrees). These conditions were verified by looking at the OpenRocket graph below:

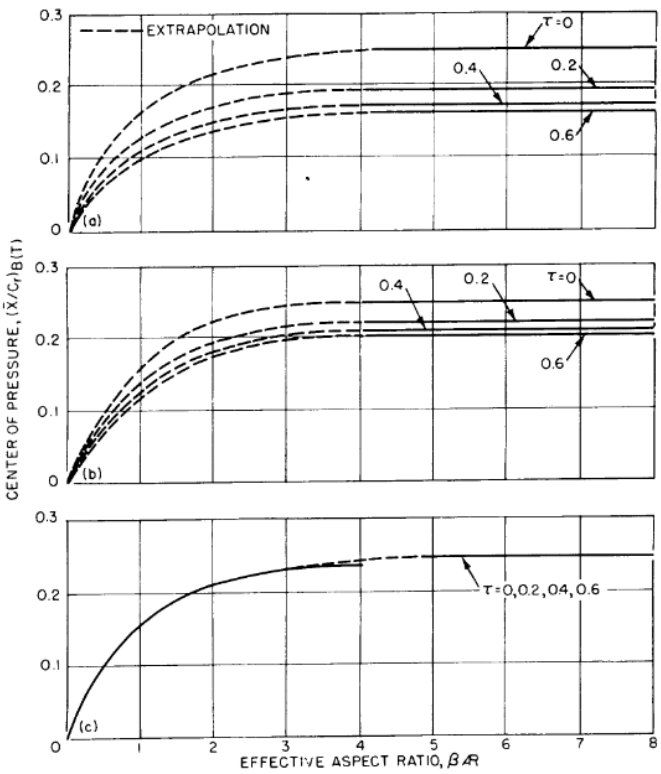



```
finsCna = fin1_Cna + fin1_Cna + fin1_Cna;
```

Calculating Boat Tail Ratio(Placeholder)

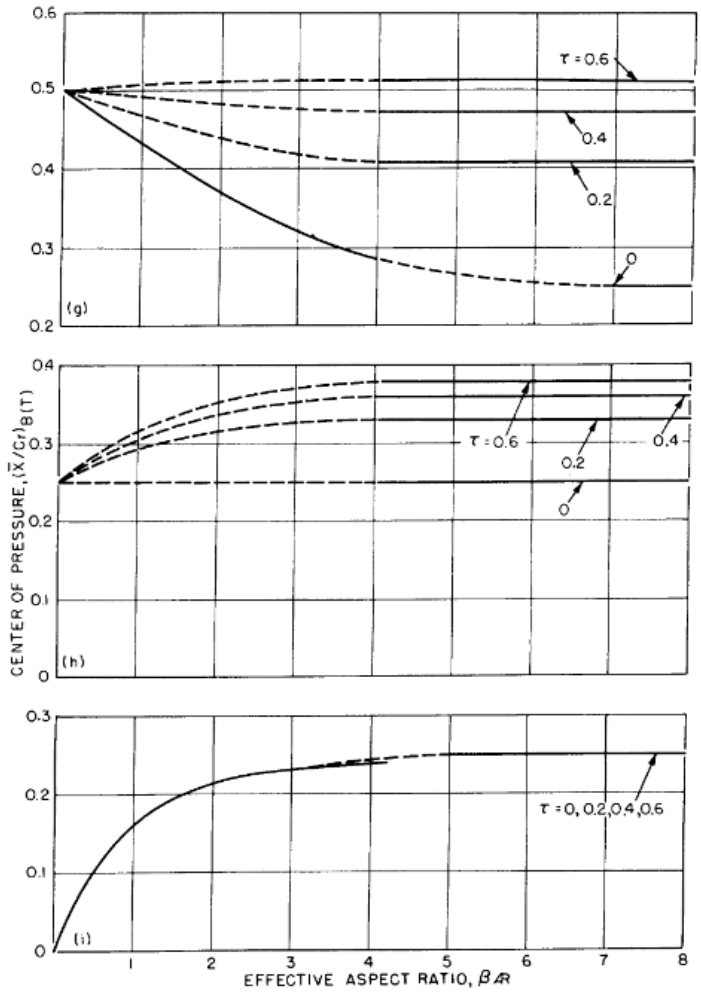
From Barrowman's The Practical Calculation of the Aerodynamic Characteristics of Slender Finned Vehicles(1967)

https://ntrs.nasa.gov/api/citations/20010047838/downloads/20010047838.pdf



a) NO LEADING-EDGE SWEEP, $\lambda=0$. (b) NO LEADING-EDGE SWEEP, $\lambda=\frac{1}{2}$. (c) NO LEADING-EDGE SWEEP, $\lambda=1$.

Figure 3-10–Subsonic $\bar{X}_{B(T)}$



(g) NO TRAILING-EDGE SWEEP, λ=0. (h) NO TRAILING-EDGE SWEEP, $\lambda=\frac{1}{2}$. (i) NO TRAILING EDGE SWEEP, λ = 1.

Figure 3-10 (Continued)—Subsonic $\bar{X}_{B(T)}$

36

τ $\frac{s + r_t}{r_t}$ Ratio

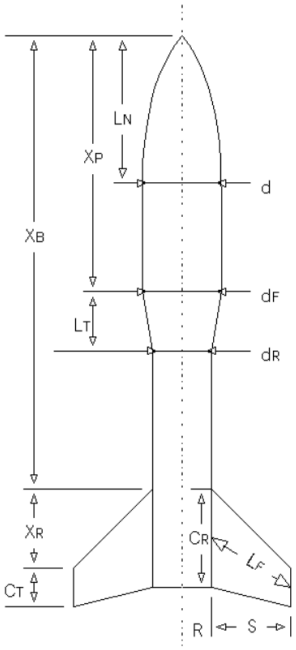
s Exposed Fin Semispan Measured from Root Chord

Note that Root Chord from this equation is

r_t Body Radius at Tail

Source that helps understand how to calculate the effective aspect ratio

https://www.rocketmime.com/rockets/Barrowman.html?



Computing Center of Pressure

The Barrowman equations permit you to determine the stability of your rocket by finding the location of the center of pressure (CP). The value computed is the distance from the tip of the rocket's nose to the CP. In order for your rocket to be stable, you would like the CP to be aft of the center of gravity (CG).

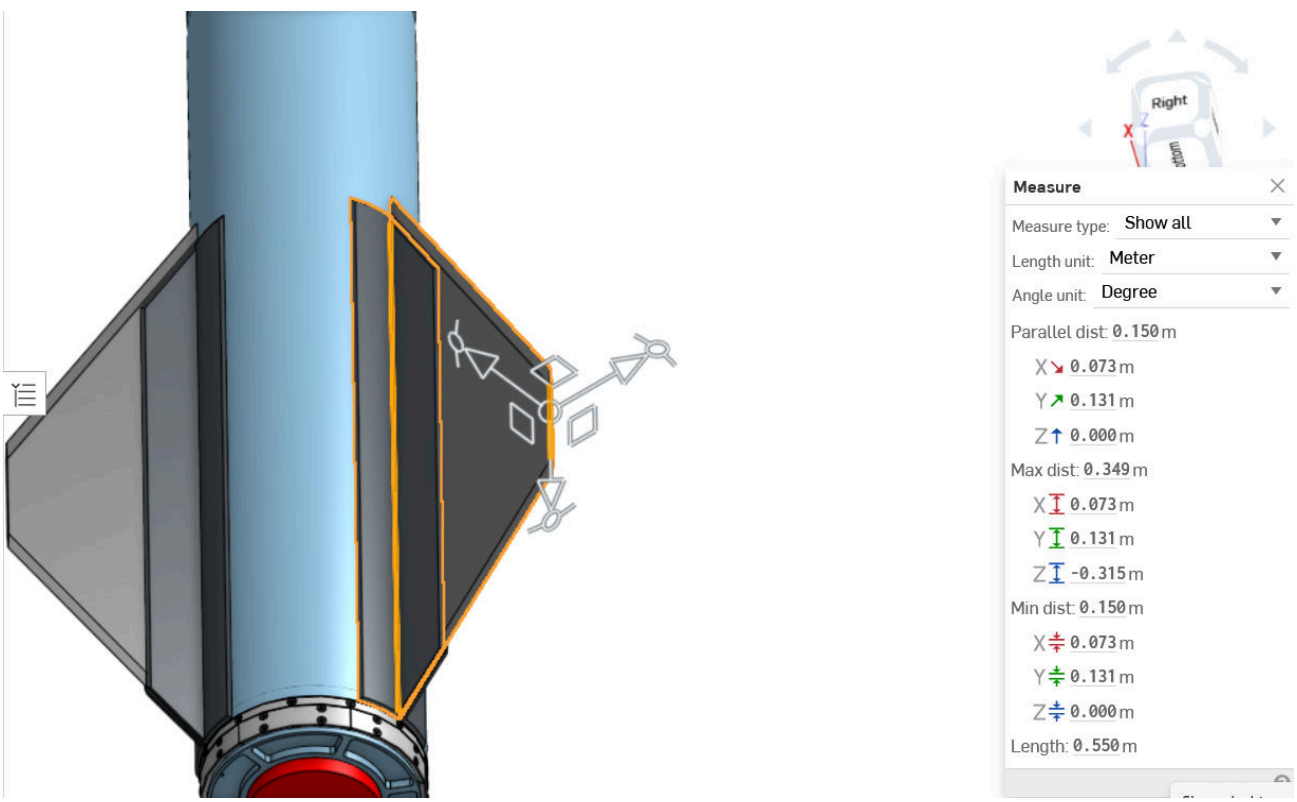
The computation of CP isn't as hard as it looks at first. Check out the spreadsheet example at the bottom of this page.

You can find the CG of your rocket by simply finding the balance point after loading recovery system and motor. (Literally - balance the rocket on your hand - or finger - and that's the CG). You can then measure from the tip of the rocket's nose to the CG. The calculated CP distance should be greater than the measured CG distance by one rocket diameter. This is called "one caliber stability".

Terms in the equations are defined below (and in the diagram):

L_N = length of nose
d = diameter at base of nose
d_F = diameter at front of transition
d_R = diameter at rear of transition
L_T = length of transition
X_P = distance from tip of nose to front of transition
C_R = fin root chord
C_T = fin tip chord
S = fin semispan
L_F = length of fin mid-chord line
R = radius of body at aft end
X_R = distance between fin root leading edge and fin tip leading edge parallel to body
X_B = distance from nose tip to fin root chord leading edge
N = number of fins

OnShape: "LV3.1 Full Assembly": pulled from "Main" Branch 2/12/26:

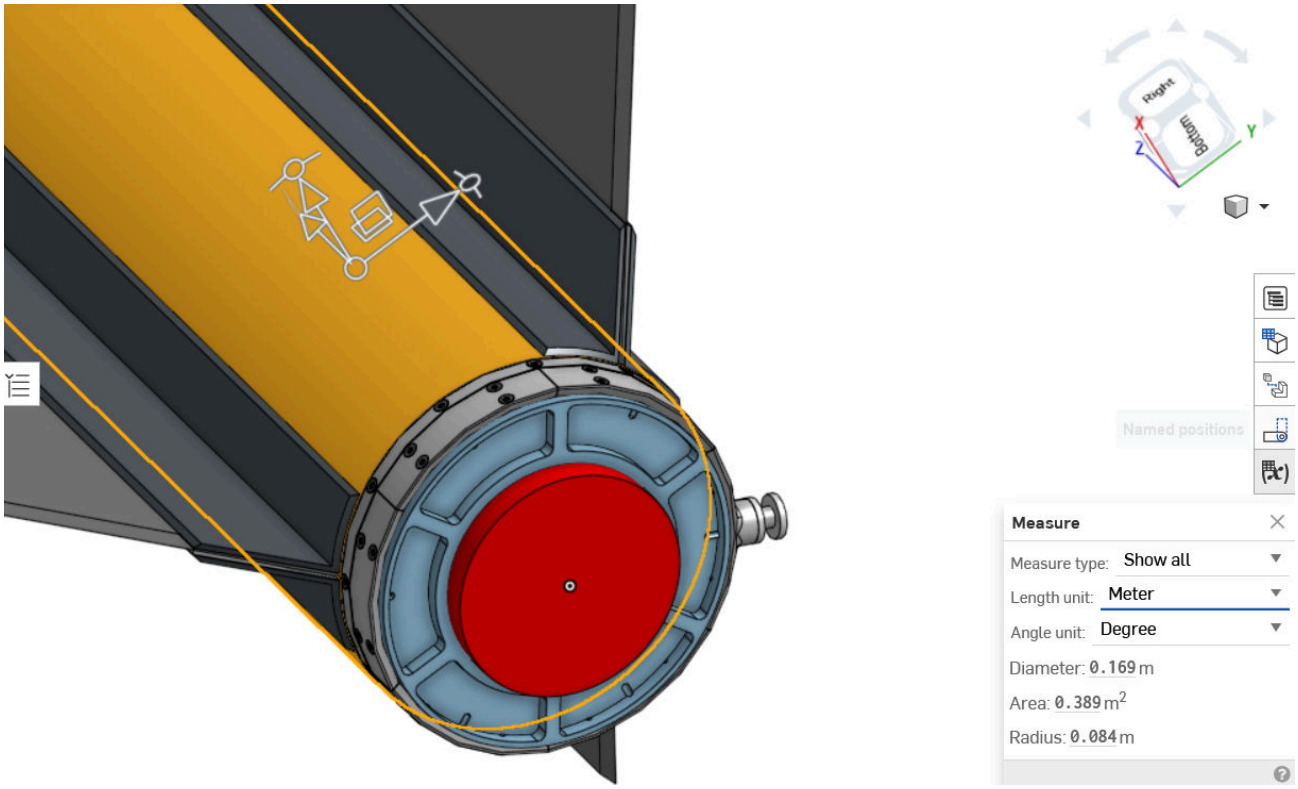


From the figure above:

```
sfins = 0.150 % in meters
```

```
sfins =
0.1500
```

Since larger tau correlates with smaller r_t, for radius we used the radius of the body tube, rather than at the fin chord.



From the figure above:

```
rtfins = 0.084 % in meters
```

```
rtfins =
0.0840
```

With the rtfins and sfins from the CAD model, can now calculate τ :

```
tauFins = (sfins + rtfins)/rtfins
```

```
tauFins =
2.7857
```

Estimating Cna for Boat Tail

Temporarily, since we do not use a boat tail, pick Cna =0

Note that this section is incomplete. That is because the fincan group has not yet completed their choice of boattail yet. This is a placeholder

From Barrowman's The Practical Calculation of the Aerodynamic Characteristics of Slender Finned Vehicles(1967)

https://ntrs.nasa.gov/api/citations/20010047838/downloads/20010047838.pdf

By definition of normal force derivative,

$$C_{N_\alpha} = \left.\frac{\partial C_N}{\partial \alpha}\right|_{\alpha=0} \quad (3$$

equation 3-63 provides,

$$C_{N_\alpha} = \frac{2}{A_r}\left[A(l_0) - A(0)\right] \quad (3$$

where Ar (reference area is =

Skinny of boat tail is A(l_0)

Fat end of boat tail is A(l_0)

Ar = area of the rocket body tube

```
boattailCna = 0
```

```
boattailCna =
0
```

Nose Cone: Normal Force Coefficient

From Barrowman's The Practical Calculation of the Aerodynamic Characteristics of Slender Finned Vehicles(1967). Page 21 - 22.

https://ntrs.nasa.gov/api/citations/20010047838/downloads/20010047838.pdf

$$C_{N_\alpha} = \frac{2}{A_r}\left[A(l_0) - A(0)\right] \qquad (3\text{-}65)$$

Immediately equation 3-65 indicates that C_{N_α} is independent of the body shape as long as the independent of the body shape as long as the integration of equation 3-62 is valid over the body. The

body components being analysed are slender and have no discontinuities in their crossectional area or its derivatives. Thus, equation 3-65 may be applied to them. Since the nose tip is assumed to be a sharp point,

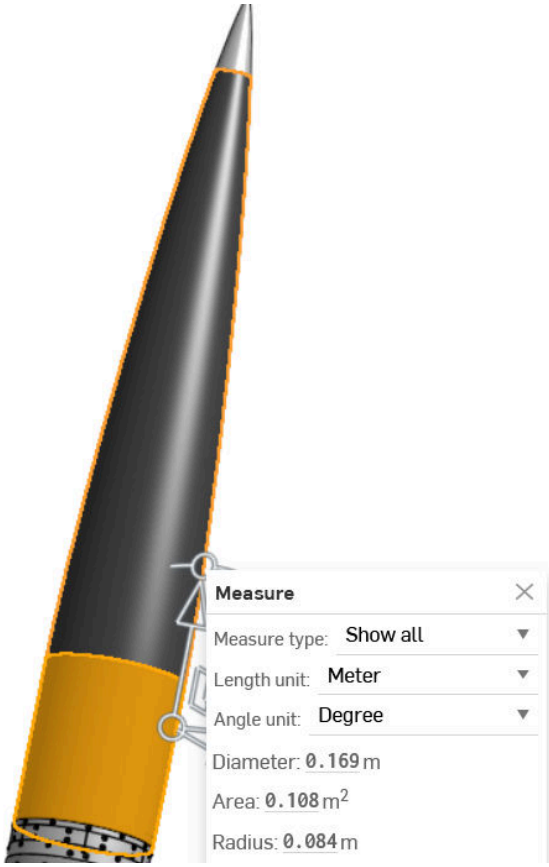
$$A(0) = 0$$

Thus, the body normal force coefficient derivative in subsonic flow is;

$$\left(C_{N_\alpha}\right)_B = 2\frac{A_{BN}}{A_r} \qquad (3\text{-}66)$$

From the quotation of Barrowman above, the analysis for the CnA of a cone is the same as the body of the rocket(with the area changes being captured in the A(l0) - A(0).

Where A_{BN} = body-nosecone interface area



Then, from the figure above, calculate A_{BN}.

```
rCone = 0.169/2
```

```
rCone =
0.0845
```

```
Abn = pi*(rCone)^2
```

```
Abn =
0.0224
```

Now that we have A_{BN} , we can calculate the normal force coefficient of the nose cone.

```
noseCna = 2 * (Abn/bodyTubeArea)
```

```
noseCna =
2
```

Turning Gust Velocity into Shear Force

http://www.aspirespace.org.uk/downloads/Rocket%20vehicle%20loads%20and%20airframe%20design.pdf

From Aspire Space Paper:

$$V_s(@x_k) = L_{nose} + L_{fins} - L_{boat-tail} - a_y \sum_{i=1}^{x_k} m - R \sum_{i=1}^{x_k} m(x_{cg} - x_m)$$

d = fuselage diameter d

R = Static Margin

$$R = \frac{9}{2I}\rho V d$$

For the L values:

$$N_{nosecone} = \frac{9}{2}\rho V S \left(\frac{dC_{L_nose}}{d\alpha}\right)_{@zero\ \alpha} \quad \text{and} \quad N_{fins} = \frac{9}{2}\rho V S \left(\frac{dC_{L_fins}}{d\alpha}\right)_{@zero\ \alpha}$$

$$N_{boat-tail} = -\frac{9}{2}\rho V S \left(\frac{dC_{L_{boat-tail}}}{d\alpha}\right)_{@zero\ \alpha}$$

and $\left(\frac{dC_L}{d\alpha}\right)_{@zero\ \alpha}$ is the lift-curve slope (gradient of) the lift versus angle of attack graph at zero angle of attack.

In contrast to the axial loads, the sum is most easily started at station 0, the aft end of the vehicle.

Continuing the philosophy of the Method of Sections, the nosecone lift $L_{nosecone}$ term is only included if 'x' is nearer the nosetip than the nosecone's centre of pressure, and so on.

Note that boat-tail 'lift' occurs in the opposite direction to fin lift, so is negative.

The fin lift term is L_{fin} acting at the fins centre-of-pressure.

The a_y summation term is the inertial load contribution of all masses 'm' tailward of the fuselage section 'X_K', and is multiplied by the vehicle's lateral acceleration a_y (metres/second^2). Each component mass 'm' gets 'switched on' as X_K passes it, and acts at its own position 'x_m'.

Extracting the Mass Data

```
% --- OpenRocket CSV extraction: Mass + CG (distance from nose) ---
fn = "rocketComponentValuesOR.csv";

% Keep headers exactly as they appear in the CSV
T = readtable(fn, "TextType","string", "VariableNamingRule","preserve");
vars = string(T.Properties.VariableNames);
```

```
% ---- pick component/name column ----
if any(vars == "Component")
    nameCol = "Component";
elseif any(vars == "Name")
    nameCol = "Name";
else
    disp("Available columns:");
    disp(vars');
    error("No Component/Name column found.");
end
names = string(T.(nameCol));

% ---- pick mass column (prefer Instance) ----
if any(vars == "Instance Mass (g)")
    massCol = "Instance Mass (g)";
    wantPrefix = "instance";
elseif any(vars == "Aggregate Mass (g)")
    massCol = "Aggregate Mass (g)";
    wantPrefix = "aggregate";
else
    disp("Available columns:");
    disp(vars');
    error("No Mass column found (Instance Mass (g) or Aggregate Mass (g)).");
end

% ---- pick CG-location column (the header varies between OR exports) ----
vlow = lower(vars);

% First try: match same prefix + contains "cg" and "loc" and "(cm)"
cgIdx = contains(vlow, wantPrefix) & contains(vlow, "cg") & contains(vlow, "loc") &
contains(vlow, "(cm)");

% If not found, loosen: any column containing "cg" and "loc" and "(cm)"
if ~any(cgIdx)
    cgIdx = contains(vlow, "cg") & contains(vlow, "loc") & contains(vlow, "(cm)");
end

% If still not found, loosen more: any column containing "cg" and "cm"
if ~any(cgIdx)
    cgIdx = contains(vlow, "cg") & contains(vlow, "cm");
end

if ~any(cgIdx)
    disp("Available columns:");
    disp(vars');
    error("No CG-location column found. Look for something like '... CG ...
(cm)'.");
end
```

```
cgCol = vars(find(cgIdx,1,"first"));  % take first match

% ---- extract + convert units ----
m_kg    = T.(massCol) / 1000;     % g -> kg
xcg_m   = T.(cgCol)   / 100;      % cm -> m (OpenRocket axial position, typically
from nose tip)

% ---- clean rows ----
valid = ~isnan(m_kg) & ~isnan(xcg_m) & (m_kg > 0);
names = names(valid);
m_kg  = m_kg(valid);
xcg_m = xcg_m(valid);

% Drop any "Total" row (avoid double count)
isTotal = contains(lower(names), "total");
names(isTotal) = [];
m_kg(isTotal)  = [];
xcg_m(isTotal) = [];

% ---- output a clean table + save ----
Tout = table(names, m_kg, xcg_m, 'VariableNames',
{'Component','Mass_kg','Xcg_m_from_nose'});
disp("Using columns:");
```

```
Using columns:
```

```
disp("  Name: " + nameCol);
```

```
  Name: Component
```

```
disp("  Mass: " + massCol);
```

```
  Mass: Instance Mass (g)
```

```
disp("  Xcg : " + cgCol);
```

```
  Xcg : Aggregate CG Location (cm)
```

```
disp(Tout(1:min(15,height(Tout)),:));
```

Component	Mass_kg	Xcg_m_from_nose
"Nose cone"	4.3377	0.82899
"Drogue Can"	0.51767	0.8904
"Shock Cord"	0.54431	0.74415
"Metal Nose Cone Tip"	0.24	0.049212
"threaded rod"	0.163	0.09652
"drogue Parachute"	0.18826	0.889
"arc clamp system"	0.097567	2.1319
"ERS Micromodule Attached to Nose Cone"	2.0484	1.096
"arc clamp system"	0.097567	2.2521
"Main Can Isogrid Module"	6.4768	1.3307
"Main Can Nylon"	0.22856	2.6778
"Main Shock cord"	0.005395	1.1818
"Main Parachute"	1.111	1.3443

```
    "arc clamp system"                          0.0976         3.0585
    "ERS Micromodule Attched to Main"           2.0462           1.54
```

```
writetable(Tout, "OR_mass_xcg_clean.csv");
```

Now needs to calculate for specfically the arc clamp in the back

```
% --- OpenRocket CSV extraction + cumulative sums about a chosen component station
---
fn = "rocketComponentValuesOR.csv";

% Keep headers exactly as they appear in the CSV
T = readtable(fn, "TextType","string", "VariableNamingRule","preserve");
vars = string(T.Properties.VariableNames);
vlow = lower(vars);

% ---- pick component/name column ----
if any(vars == "Component")
    nameCol = "Component";
elseif any(vars == "Name")
    nameCol = "Name";
else
    disp("Available columns:");
    disp(vars');
    error("No Component/Name column found.");
end
names = string(T.(nameCol));

% ---- pick mass column (prefer Instance) ----
if any(vars == "Instance Mass (g)")
    massCol = "Instance Mass (g)";
    wantPrefix = "instance";
elseif any(vars == "Aggregate Mass (g)")
    massCol = "Aggregate Mass (g)";
    wantPrefix = "aggregate";
else
    disp("Available columns:");
    disp(vars');
    error("No Mass column found (Instance Mass (g) or Aggregate Mass (g)).");
end

% ---- pick CG-location column (distance from nose, in cm) ----
cgIdx = contains(vlow, wantPrefix) & contains(vlow,"cg") & contains(vlow,"loc") &
contains(vlow,"(cm)");
if ~any(cgIdx)
    cgIdx = contains(vlow,"cg") & contains(vlow,"loc") & contains(vlow,"(cm)");
end
```

```
if ~any(cgIdx)
    cgIdx = contains(vlow,"cg") & contains(vlow,"cm");
end
if ~any(cgIdx)
    disp("Available columns:");
    disp(vars');
    error("No CG Location column found (look for something like '...CG...Location...
(cm)')." );
end
cgCol = vars(find(cgIdx,1,"first"));

% ---- extract + convert units (OpenRocket axial CG is typically from nose tip) ----
m = T.(massCol)/1000;      % g -> kg
x = T.(cgCol)/100;         % cm -> m from nose tip

% ---- clean rows ----
valid = ~isnan(m) & ~isnan(x) & (m > 0);
m = m(valid); x = x(valid); names = names(valid);

isTotal = contains(lower(names), "total");
m(isTotal) = []; x(isTotal) = []; names(isTotal) = [];

% ---- sort nose -> tail ----
[x, order] = sort(x, "ascend");
m = m(order); names = names(order);

% ---- choose the station by picking a component (rear-most duplicate name) ----
target = "arc clamp system";    % change as needed
matches = contains(lower(names), lower(target));
assert(any(matches), "Target not found in component names.");

rows = find(matches);
[~, k] = max(x(rows));          % rear-most = max distance from nose
j = rows(k);

x_m = x(j);                     % station (m from nose tip)

% ---- include parts before/at that station ----
idx = (x <= x_m);

% ---- outputs used separately ----
M_before  = sum(m(idx));                    % Σ m_i
S_before  = sum(m(idx) .* (x(idx) - x_m));  % Σ m_i (x_cg,i - x_m)
Mx_before = sum(m(idx) .* x(idx));          % Σ m_i x_cg,i (about nose)

% ---- print ----
fprintf("Using columns:\n  Name: %s\n  Mass: %s\n  Xcg : %s\n", nameCol, massCol,
cgCol);
```

```
Using columns:
  Name: Component
  Mass: Instance Mass (g)
  Xcg : Aggregate CG Location (cm)
```

```
fprintf("Selected component (rear-most match): %s\n", names(j));
```

```
Selected component (rear-most match): arc clamp system
```

```
fprintf("x_m (distance from nose) = %.6f m\n", x_m);
```

```
x_m (distance from nose) = 3.788330 m
```

```
fprintf("Cumulative mass Σm = %.6f kg\n", M_before);
```

```
Cumulative mass Σm = 46.022962 kg
```

```
fprintf("Cumulative first moment about nose Σ(m*x) = %.6f kg*m\n", Mx_before);
```

```
Cumulative first moment about nose Σ(m*x) = 102.960371 kg*m
```

```
fprintf("Cumulative sum Σ m*(x_cg - x_m) = %.6f kg*m\n", S_before);
```

```
Cumulative sum Σ m*(x_cg - x_m) = -71.389788 kg*m
```

Now that we have the cumulations for the arc clamp:

```
% % After you have sorted x and m as above...
%
% n = numel(m);
%
% % Cumulative mass from nose up to each component
% cumMass = cumsum(m);    % cumMass(j) = Σ_{i<=j} m_i
%
% % Cumulative moment term for each station x_m = x(j):
% % S(j) = Σ_{i<=j} m_i * (x_i - x_j)
% %
% % Compute efficiently:
% cumMx = cumsum(m .* x);               % Σ m_i x_i up to j
% S = cumMx - x .* cumMass;             % Σ m_i x_i - x_j Σ m_i
%
% % Now cumMass(j) and S(j) are your two "separate calculation" inputs
% Tout = table(names, x, m, cumMass, S, ...
%     'VariableNames',
{'Component','Xcg_fromNose_m','Mass_kg','CumMass_kg','CumSum_momentTerm_kgm'});
%
```

```
% writetable(Tout, "OR_cumulative_mass_and_sum.csv");
% disp(Tout(1:min(10,height(Tout)),:))
```

Shear Force: Calculating Individual Component Lifts

Recall:

$$V_s(@x_k) = L_{nose} + L_{fins} - L_{boat-tail} - a_y \sum_{i=1}^{x_k} m - R \sum_{i=1}^{x_k} m(x_{cg} - x_m)$$

d = fuselage diameter d

R = Static Margin

$$R = \frac{9}{2I}\rho V d$$

For the L values:

$$N_{nosecone} = \frac{9}{2}\rho V S \left(\frac{dC_{L_nose}}{d\alpha}\right)_{@zero\ \alpha} \quad \text{and} \quad N_{fins} = \frac{9}{2}\rho V S \left(\frac{dC_{L_fins}}{d\alpha}\right)_{@zero\ \alpha}$$

$$N_{boat-tail} = -\frac{9}{2}\rho V S \left(\frac{dC_{L_{boat-tail}}}{d\alpha}\right)_{@zero\ \alpha}$$

and $\left(\frac{dC_L}{d\alpha}\right)_{@zero\ \alpha}$ is the lift-curve slope (gradient of) the lift versus angle of attack graph at zero angle of attack.

S = rocket body cross-sectional area = A_r

3.6.8 Surface Winds for Off-Nominal Landing

Design Limits

Same as Section 3.5.8 with the model inputs as specified here.

Model Input:

Minimum surface wind for all off-nominal cases is calm conditions, i.e. no wind.

Maximum off–nominal land landing surface winds are not defined.

For off-nominal water landings, the maximum 10.0 m (32.8 ft) height steady state design wind is 13.9 m/s (45.6 ft/s). This value is associated with the sea state conditions defined in Section 3.6.18, Sea State for Off-Nominal Water Landing. The peak wind value of 19.5 m/s (63.8 ft/s) was determined by multiplying the maximum steady state wind by the 1.4 gust factor given in Section 3.5.8, Surface Winds for Normal Landing.

Then when I tried this the graphs were large. So I did v = 2m/s instead for wind(default setting for open rocket)

```
windVelMax = 13.91 % in m/s
```

```
windVelMax =
13.9100
```

Note that it is pulled from onshape inspection. Unfortunately, onshape does not allow for the output of a mass vs. x-position table

```
% Recall that cosine gust velocity profile is stored here: vg
% From the U.S Standard Atmosphere (1976) at the altitude of Max Q (1114.7
% m) . Air Density = 1.09917 kg/m^3
%
rhoMaxQ = 1.09917
```

```
rhoMaxQ =
1.0992
```

```
vehicleVelTimeGustCOS = table2array(vehicleVelTimeGustCOS)
```

```
vehicleVelTimeGustCOS = 200×2
    5.5206  331.4894
    5.5215  331.4942
    5.5224  331.4990
    5.5233  331.5038
    5.5242  331.5086
    5.5251  331.5134
    5.5260  331.5182
    5.5269  331.5230
    5.5278  331.5278
    5.5287  331.5326
    5.5296  331.5374
    5.5306  331.5422
    5.5315  331.5470
    5.5324  331.5518
    5.5333  331.5566
      ⋮
```

```
vehicleWindTimeGustCOS = table2array(vehicleWindTimeGustCOS)
```

```
vehicleWindTimeGustCOS = 200×2
    5.5206    1.9362
    5.5215    1.9378
    5.5224    1.9395
    5.5233    1.9411
    5.5242    1.9427
    5.5251    1.9444
    5.5260    1.9460
    5.5269    1.9476
    5.5278    1.9493
    5.5287    1.9509
    5.5296    1.9526
    5.5306    1.9542
    5.5315    1.9558
    5.5324    1.9575
```

```
    5.5333    1.9591
      ⋮
```

```
% velocity for dynamic array
velRel = vehicleVelTimeGustCOS(:,2) - vehicleWindTimeGustCOS(:,2)
```

```
velRel = 200×1
  329.5532
  329.5564
  329.5596
  329.5627
  329.5659
  329.5691
  329.5722
  329.5754
  329.5786
  329.5817
  329.5849
  329.5880
  329.5912
  329.5944
  329.5975
      ⋮
```

```
% Lift Nosecone
Nnosecone = (9/2).*(rhoMaxQ.*velRel.*bodyTubeArea).*(noseCna)
```

```
Nnosecone = 200×1
   73.1301
   73.1308
   73.1315
   73.1322
   73.1329
   73.1336
   73.1343
   73.1350
   73.1357
   73.1364
   73.1371
   73.1378
   73.1385
   73.1392
   73.1399
      ⋮
```

```
% Lift of ALL 3 fins
Nfins = (9/2).*(rhoMaxQ.*velRel.*bodyTubeArea).*(finsCna)
```

```
Nfins = 200×1
    2.0321
    2.0322
    2.0322
    2.0322
    2.0322
    2.0322
    2.0323
```

```
    2.0323
    2.0323
    2.0323
    2.0323
    2.0324
    2.0324
    2.0324
    2.0324
      ⋮
```

```
% Lift of the Boat Tail
Nbt = 0 % 0 since there are no plans for boat-tail. additionally, no boat tail is
good for a conservative estimate.
```

```
Nbt =
0
```

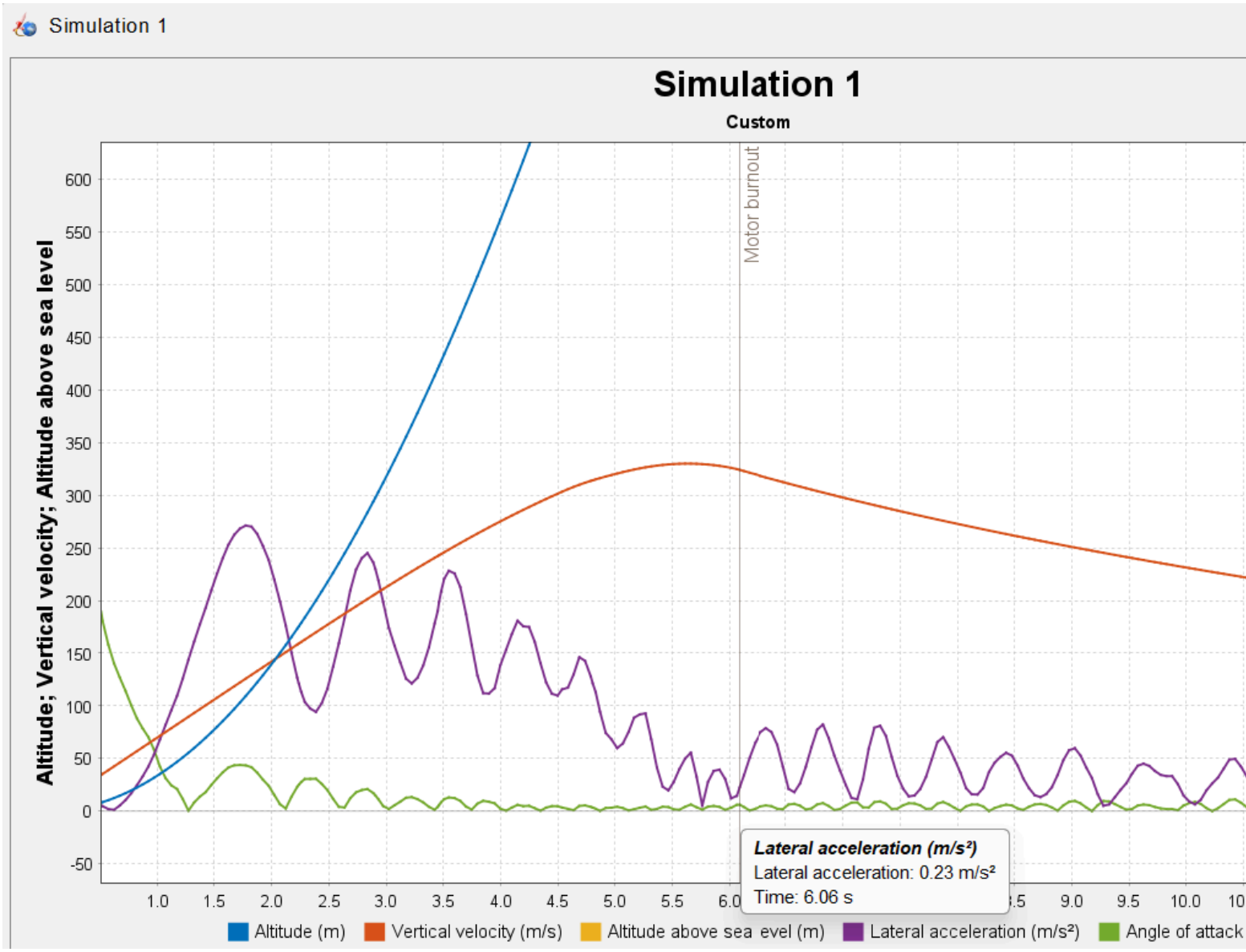
Shear Force: a_y sum(m)

$$V_s(@x_k) = L_{nose} + L_{fins} - L_{boat-tail} - a_y \sum_{i=1}^{x_k} m - R \sum_{i=1}^{x_k} m(x_{cg} - x_m)$$

d = fuselage diameter d

R = Static Margin

In the interest of data analysis time, the lateral acceleration is used exactly at the time of max Q. This lateral acceleration is applied acorss the board as an array the size of the 'gust velocity profile' array. In the next iteration, we will want to properly extract all of the vertical acceleration values, as they change over the duration of the gust




```
% 0.23 m/s^2 comes from the graph above
% t
% M_before
NlatAccel = vehicleLAccTimeGustCOS(:,2).*M_before
```

NlatAccel = 200×1 table

	LateralAcceleration
1	37.2229
2	36.8673
3	36.5118
4	36.1562
5	35.8007
6	35.4451

	LateralAcceleration
7	35.0896
8	34.7340
9	34.3785
10	34.0229
11	33.6674
12	33.3118
13	32.9563
14	32.6007
15	32.2452
16	31.8896
17	31.5341
18	31.1786
19	30.8230
20	30.4675
21	30.1119
22	29.7564
23	29.4008
24	29.0453
25	28.6897
26	28.3342
27	27.9786
28	27.6231
29	27.2675
30	26.9120
31	26.5564
32	26.2009
33	25.8453
34	25.4898
35	25.1343
36	24.7787
37	24.4232
38	24.0676
39	23.7121

	LateralAcceleration
40	23.3565
41	23.0010
42	22.6454
43	22.2899
44	21.9343
45	21.5788
46	21.3310
47	21.1912
48	21.0515
49	20.9118
50	20.7721
51	20.6324
52	20.4927
53	20.3530
54	20.2133
55	20.0736
56	19.9339
57	19.7942
58	19.6544
59	19.5147
60	19.3750
61	19.2353
62	19.0956
63	18.9559
64	18.8162
65	18.6765
66	18.5368
67	18.3971
68	18.2574
69	18.1176
70	17.9779
71	17.8382
72	17.6985

	LateralAcceleration
73	17.5588
74	17.4191
75	17.2794
76	17.1397
77	17.0000
78	16.8603
79	16.7206
80	16.5808
81	16.4411
82	16.3014
83	16.1617
84	16.0220
85	15.8823
86	15.7426
87	15.6029
88	15.4632
89	15.3235
90	15.1838
91	15.0440
92	14.9043
93	14.7646
94	14.6249
95	14.4852
96	14.3455
97	14.2058
98	14.0661
99	13.9264
100	13.7867

⋮

Shear Force: Static Margin

In this problem(to save time) we use the 'uniform cylinder approximation for the moment of inertia in the lateral axis. Note that the center of gravity is not centered on the longitude of the rocket, so it would idealy be calculated with the parallel axis theorem:

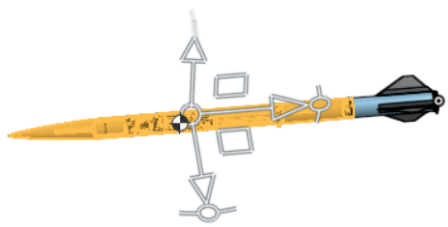
$$I = \sum (I_{component,own} + mr^2)$$

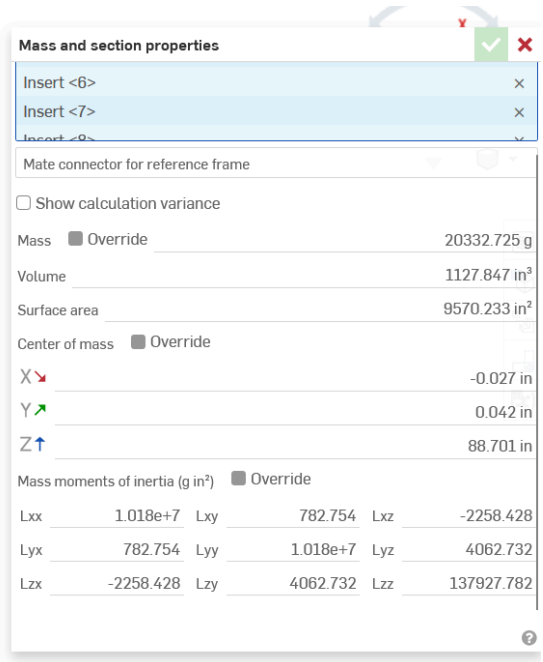
But for speed and simplicity we use the uniform cylinder approximation:

$$I = \frac{1}{12} m(3r^3 + L^2)$$

where r is the fuselage radius, and L is the length of the cylinder, and m is the mass.

Pulling those values from the onshape document below:





Using the information given from OnShape makes a simplifying assumption that bypasses individual component I, and assumes a uniform, perfrect cylinder from the tip of the nose cone to the back arc clamps. This makes for a conservative estimate for lift by increasing moment of inertia

Mass = 20.332725 kg

Length = 1.762 meters

r = 0.169 meters /2 = 0.0845 meters

```
mass2Arc = 20.332725;
lengthNose2Arc = 1.762;
fuselageR = dBodyTube/2;
momentOfInertia2BArc = (1/12)*(mass2Arc)*(3*fuselageR^3 + lengthNose2Arc^2)
```

```
momentOfInertia2BArc =
5.2636
```

$$V_s(@x_k) = L_{nose} + L_{fins} - L_{boat-tail} - a_y \sum_{i=1}^{x_k} m - R \sum_{i=1}^{x_k} m(x_{cg} - x_m)$$

d = fuselage diameter d

R = Angular Acceleration

If you have the L1, L2, and L3 are gien in meters(rather than calibres)

$$R = \frac{9}{2I} \rho V(static - margin)$$

Static Margin is this:

$$\left(-\frac{dC_{L_{nose}}}{d\alpha} L1 + \frac{dC_{L_{fins}}}{d\alpha} L2 - \frac{dC_{L_{boat-tail}}}{d\alpha} L3 \right)_{@zero\ \alpha}$$

L1, L2 and L3 are the moment arms of the nose, fins, and boat-tail from their respective centres of pressure to the C.G.

For speed, I used the values for the moment arms as shown in the spreadsheet below:

https://docs.google.com/spreadsheets/d/1YpCOBGyIhGbyl9DqY5GLDXi1CcHhjao9K5FwuqBK1_Q/edit?usp=sharing

```
part1OfREqn = (9/(2*momentOfInertia2BArc))*(rhoMaxQ*gustArray(:,1))
```

```
part1OfREqn = 200×1
    5.1878
    5.1886
    5.1895
    5.1903
    5.1912
    5.1920
    5.1929
    5.1937
    5.1946
    5.1955
    5.1963
```

```
    5.1972
    5.1980
    5.1989
    5.1997
      ⋮
```

```
% from the spreadsheet above (in meters):
L1 = 1.028330124
```

```
L1 =
1.0283
```

```
L2 = 1.568169876
```

```
L2 =
1.5682
```

```
L3 = 1.966669876
```

```
L3 =
1.9667
```

```
staticMargin = ((-1)*noseCna*L1 + finsCna*L2 + (-1)*boattailCna*L3)
```

```
staticMargin =
-1.9695
```

```
shearFrStaticMargin = part1OfREqn*staticMargin
```

```
shearFrStaticMargin = 200×1
  -10.2174
  -10.2190
  -10.2207
  -10.2224
  -10.2241
  -10.2258
  -10.2274
  -10.2291
  -10.2308
  -10.2325
  -10.2342
  -10.2359
  -10.2375
  -10.2392
  -10.2409
      ⋮
```

Shear Force: Stability Components, Vertical Acceleration, and Static Margin

Bringing all of the shear force component from above together;

```
% due to preious errors, turning the things into arrays
NlatAccel = table2array(NlatAccel)
```

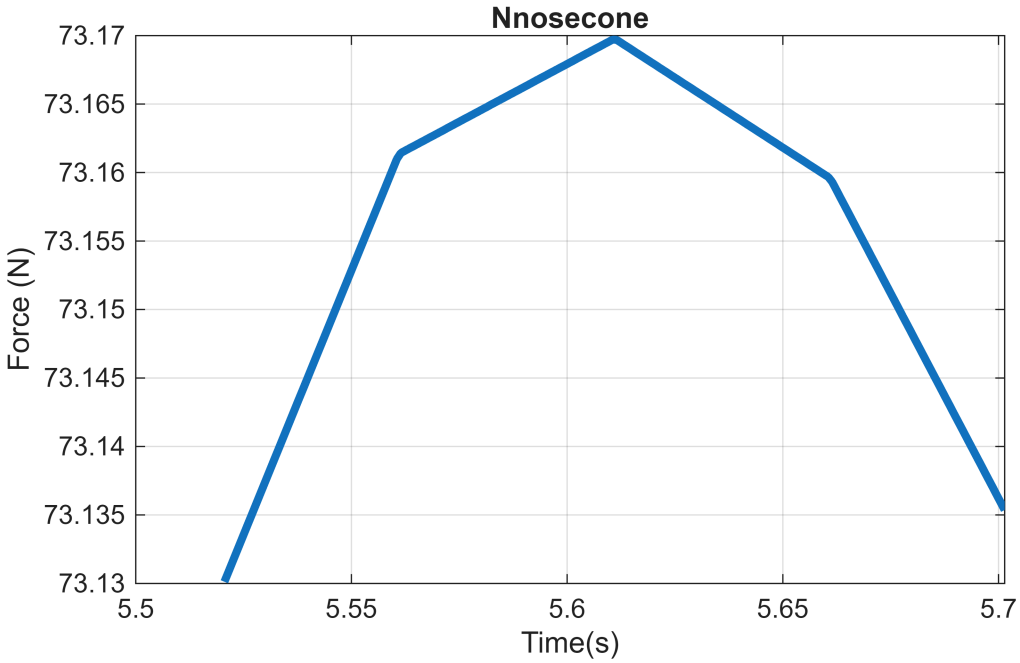
```
NlatAccel = 200×1
   37.2229
   36.8673
   36.5118
   36.1562
   35.8007
   35.4451
   35.0896
   34.7340
   34.3785
   34.0229
   33.6674
   33.3118
   32.9563
   32.6007
   32.2452
      ⋮
```

```
% Nfins = table2array(Nfins)
% Nnosecone = table2array(Nnosecone)
totalShearFnOfGustVel = Nnosecone + Nfins - Nbt - NlatAccel - shearFrStaticMargin
```

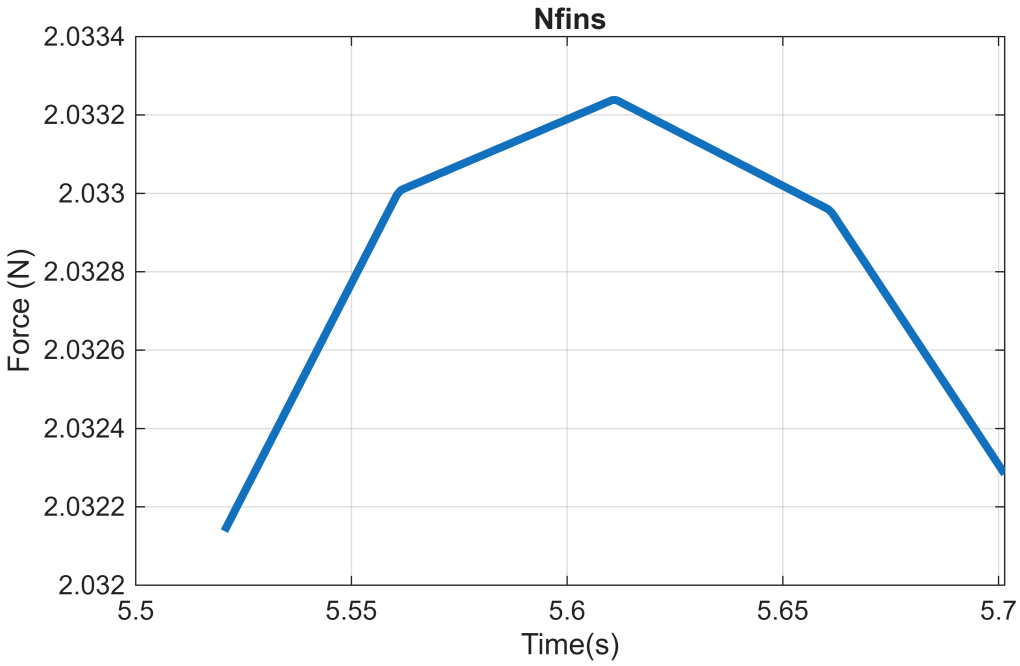
```
totalShearFnOfGustVel = 200×1
   48.1568
   48.5147
   48.8727
   49.2306
   49.5886
   49.9465
   50.3045
   50.6624
   51.0204
   51.3783
   51.7363
   52.0942
   52.4522
   52.8101
   53.1681
      ⋮
```

Stick the shear force on the back arc clamps on a graph related to time:

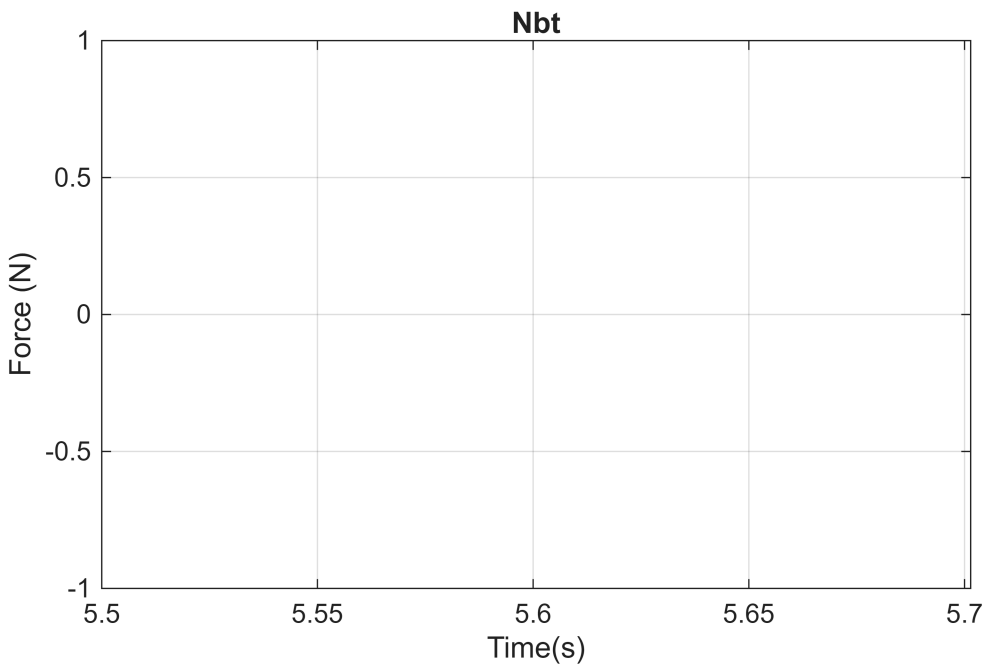
```
figure; plot(gustArray(:,1), Nnosecone, 'LineWidth', 3); xlabel('Time(s)');
ylabel('Force (N)'); title('Nnosecone'); grid on
```



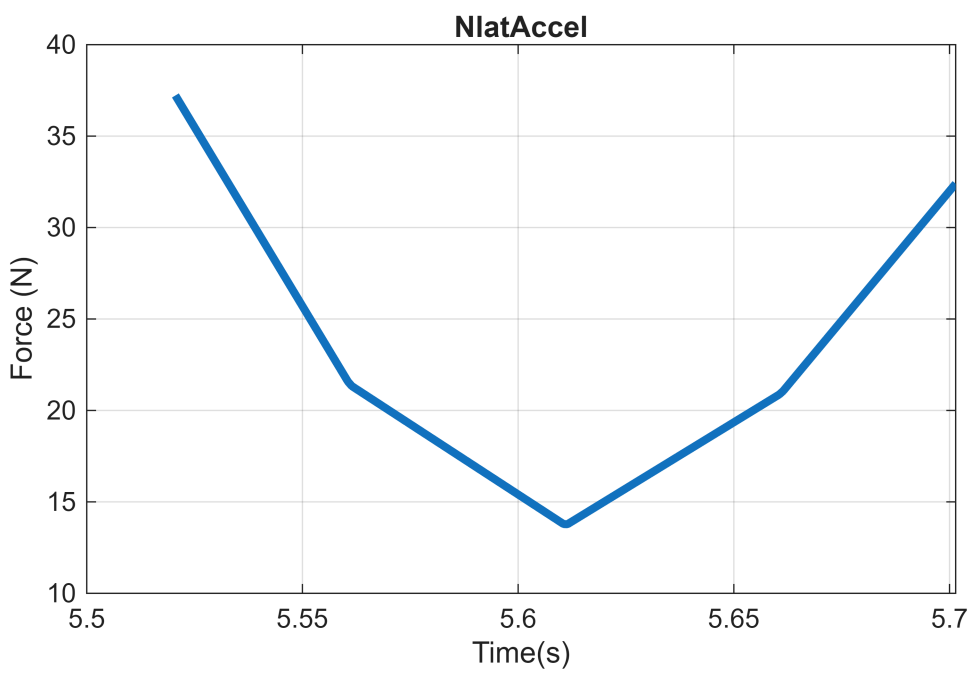

```
figure; plot(gustArray(:,1), Nfins, 'LineWidth', 3);  xlabel('Time(s)');
ylabel('Force (N)');   title('Nfins');     grid on
```



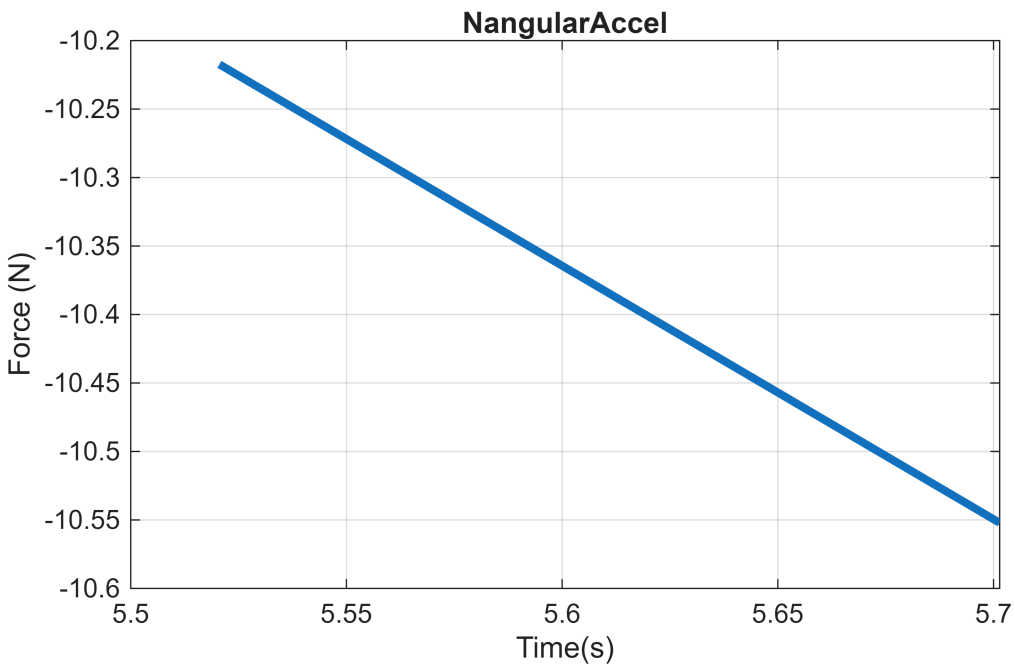

```
figure; plot(gustArray(:,1), Nbt, 'LineWidth', 3);  xlabel('Time(s)');
ylabel('Force (N)');     title('Nbt');       grid on
```



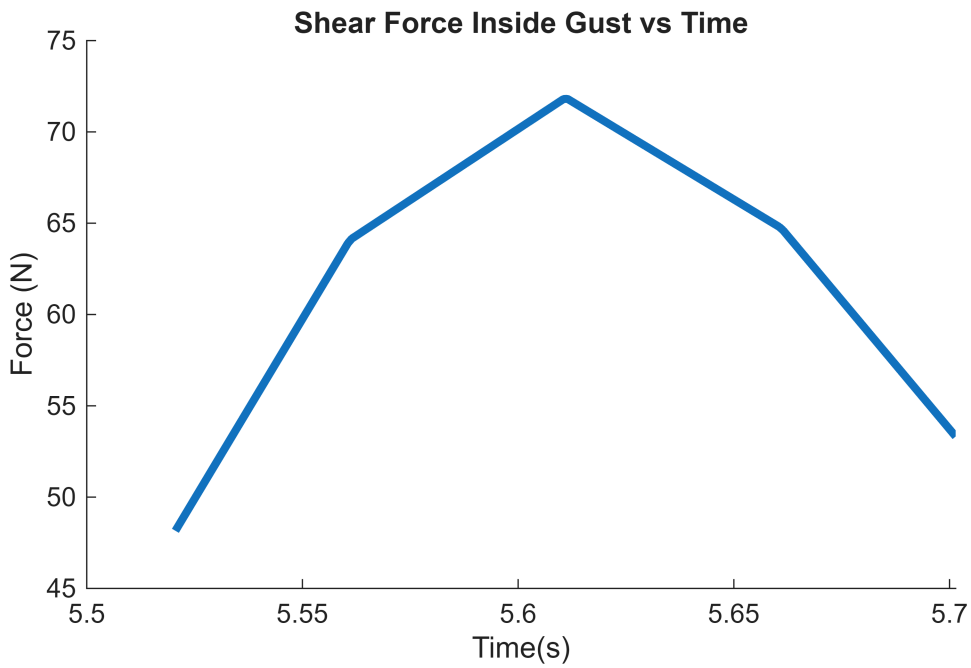

```
figure; plot(gustArray(:,1), NlatAccel, 'LineWidth', 3); xlabel('Time(s)');
ylabel('Force (N)');title('NlatAccel');grid on
```



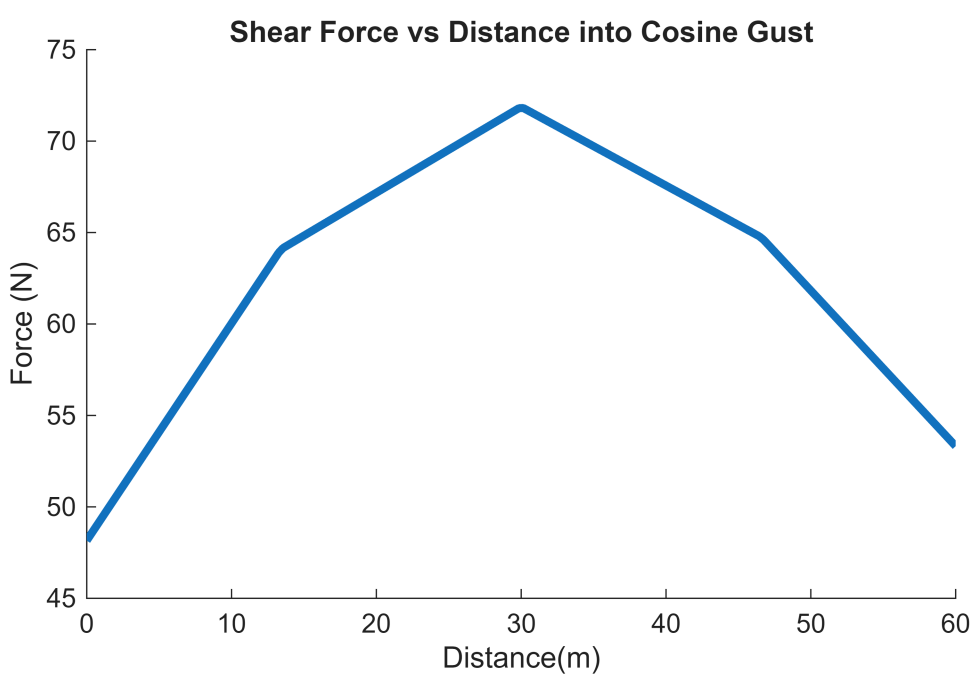

```
figure; plot(gustArray(:,1), shearFrStaticMargin, 'LineWidth', 3);
xlabel('Time(s)'); ylabel('Force (N)'); title('NangularAccel'); grid on
```




```
% figure; plot(gustArray(:,1), gust.aoa_array); title('Angle of Attack'); grid on


% Shear Force vs. Time
figure;
hold on;
title('Shear Force Inside Gust vs Time')
xlabel('Time(s)')
ylabel('Force (N)')

plot(gustArray(:,1), totalShearFnOfGustVel(:,1), 'LineWidth', 3);
```




```
% Shear Force vs. Distance
figure;
hold on;
title('Shear Force vs Distance into Cosine Gust')
xlabel('Distance(m)')
ylabel('Force (N)')
plot(gustArray(:,3), (totalShearFnOfGustVel(:,1)), 'LineWidth', 3);
```




```
fprintf('Max Shear is (N) : %f%', max(totalShearFnOfGustVel(:,1)))
```

```
Max Shear is (N) : 71.809829
```

```
save('shearDGustProfile.mat', 'totalShearFnOfGustVel')
save('shearDGustTimeDist.mat', 'gustArray')
```

Shear Force: Including Points Outside of the Gust

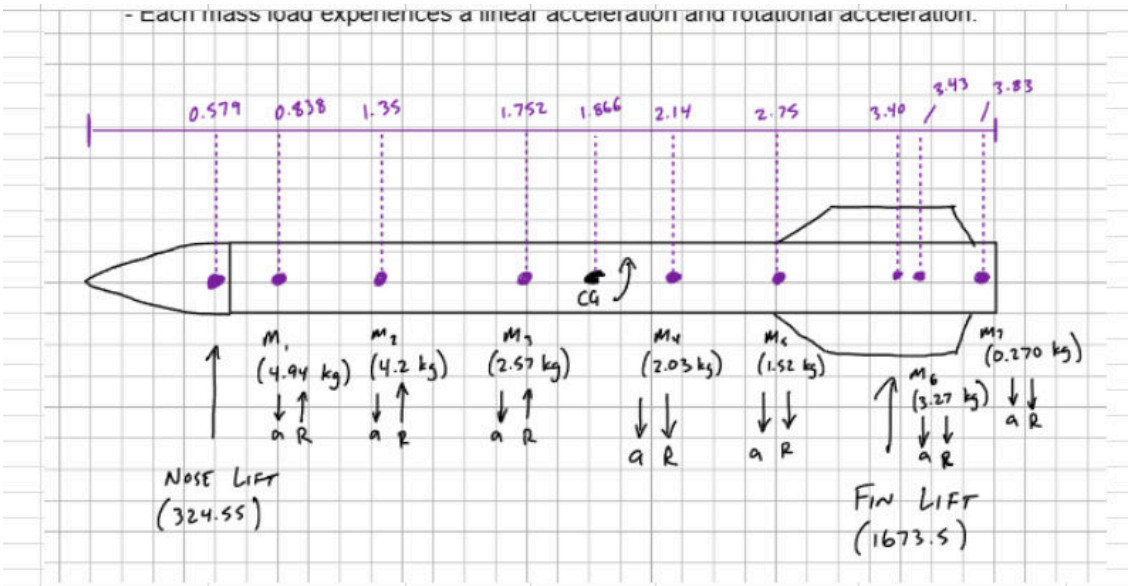
Shear Force: Vertical Acceleration at Points of Interest

Recall that we are looking at the forces on the ERS, as well as the arc clamps.

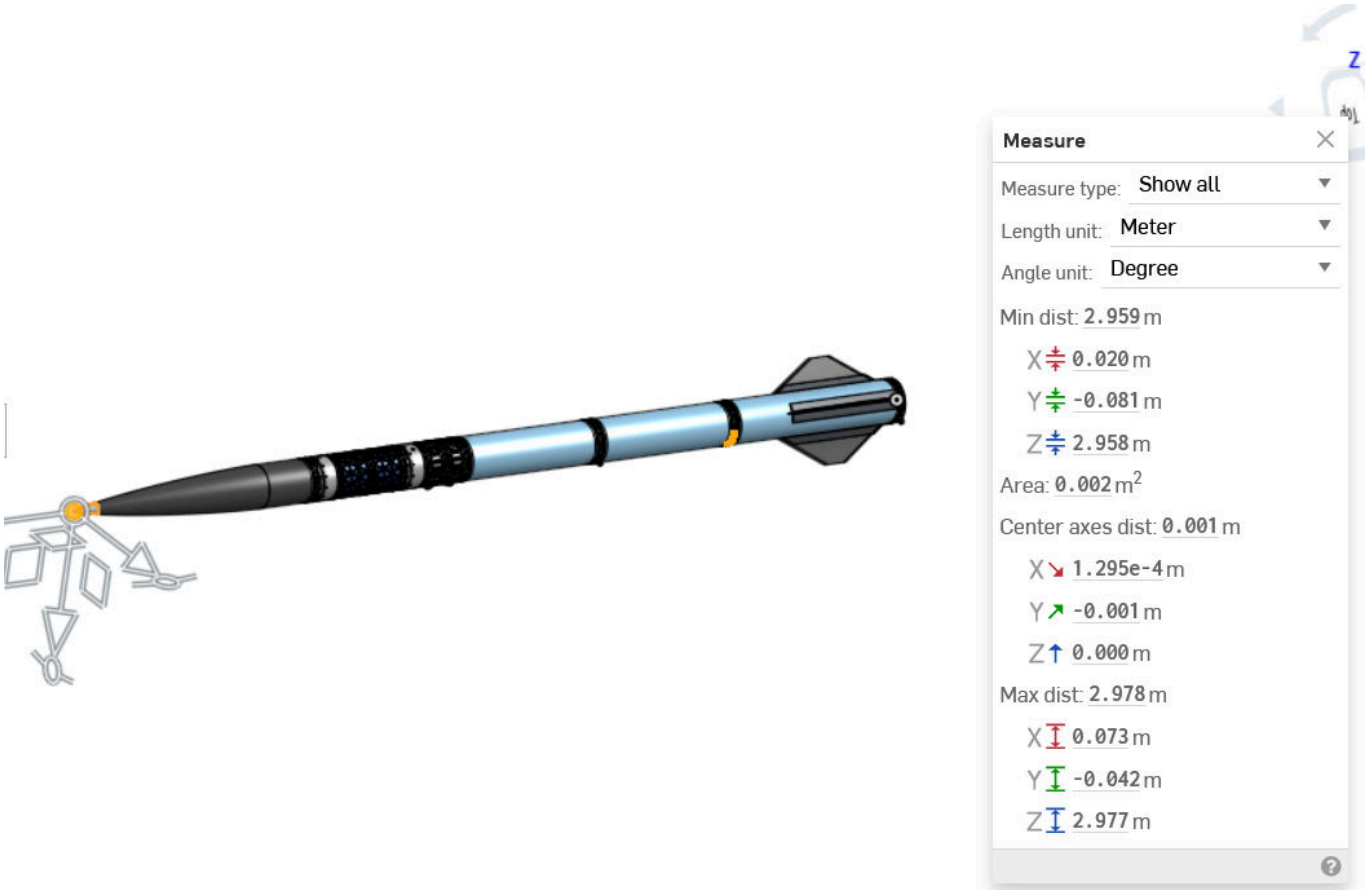
Arc Clamps

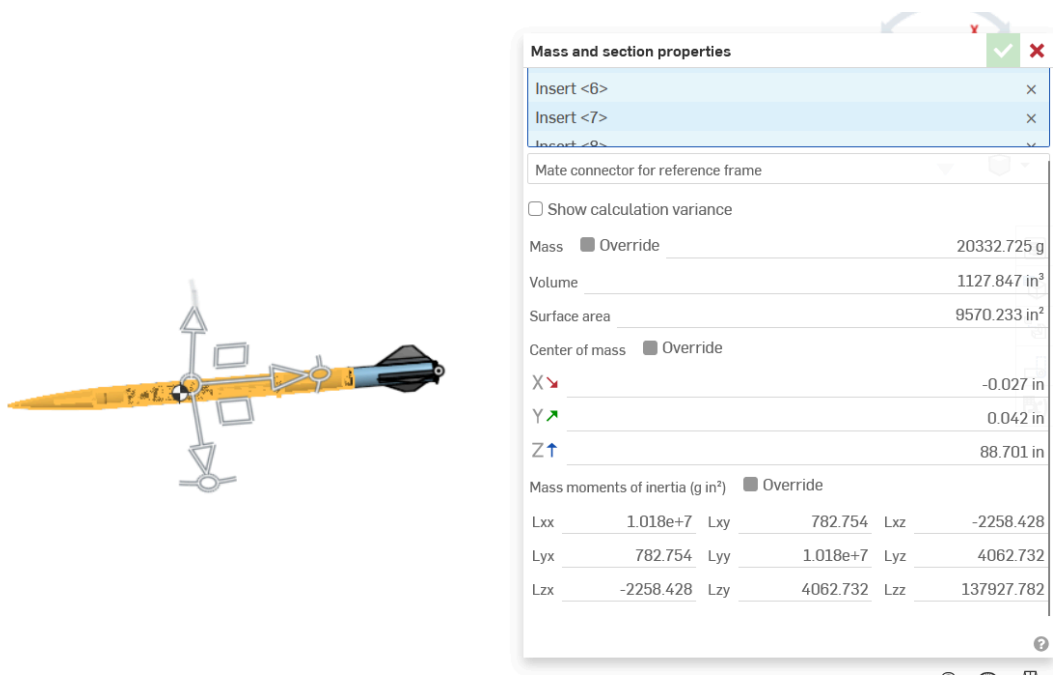
For the Arc Clamps, we know from the CFD that the largest shear at max Q is at around 3.40 meters.

https://docs.google.com/spreadsheets/d/1YpCOBGyIhGbyl9DqY5GLDXi1CcHhjao9K5FwuqBK1_Q/edit?usp=sharing



Closest arc clamps to this max shear force point is shown below:


Measure
Measure type: Show all
Length unit: Meter
Angle unit: Degree
Min dist: 2.959 m
X 0.020 m
Y -0.081 m
Z 2.958 m
Area: 0.002 m²
Center axes dist: 0.001 m
X 1.295e-4 m
Y -0.001 m
Z 0.000 m
Max dist: 2.978 m
X 0.073 m
Y -0.042 m
Z 2.977 m


Mass and section properties
Insert <6>
Insert <7>
Mate connector for reference frame
Show calculation variance
Mass Override 20332.725 g
Volume 1127.847 in³
Surface area 9570.233 in²
Center of mass Override
X -0.027 in
Y 0.042 in
Z 88.701 in
Mass moments of inertia (g in²) Override
Lxx 1.018e+7 Lxy 782.754 Lxz -2258.428
Lyx 782.754 Lyy 1.018e+7 Lyz 4062.732
Lzx -2258.428 Lzy 4062.732 Lzz 137927.782

Arc Clamps that Receive the Most Force

```
%yep placeholder
```

Bending Moments

$$M_B(@x_k) = L_{nose}(x_k - x_{nose}) + L_{boat}(x_k - x_{boat}) + L_{fin}(x_k - x_{fin}) - a_y \sum_{0}^{x_k}(x_k - x_m) - R \sum_{0}^{x_k} m(x_k - x_m)(x_{CG} - x_m)$$

where x_m is the position of each individual mass of the individual components

Bending Moments: Pulling Data for the x's

```
%% ============================================================
%  OpenRocket Component Position + Moment Extractor
% ============================================================

clear; clc;

%% LOAD FILE
fn = "rocketComponentValuesOR.csv";
T = readtable(fn,"TextType","string","VariableNamingRule","preserve");

vars = string(T.Properties.VariableNames);

disp("Detected columns:")
```

```
Detected columns:
```

```
disp(vars')
```

```
    "Component"
    "Instance Mass (g)"
    "Aggregate Mass (g)"
    "Aggregate CG Location (cm)"
    "CP (cm)"
    "Normal Force Coefficient"
```

```
%% ------------------------------------------------------------
% FIND NAME COLUMN
%% ------------------------------------------------------------
nameCol = vars(contains(lower(vars),"component") | contains(lower(vars),"name"));
if isempty(nameCol)
    error("No component/name column found.");
end
```

```
nameCol = nameCol(1);
names = T.(nameCol);

%% ---------------------------------------------------------
% FIND MASS COLUMN
%% ---------------------------------------------------------
massCol = vars(contains(lower(vars),"mass"));
if isempty(massCol)
    error("No mass column found.");
end
massCol = massCol(1);
m = T.(massCol);

%% ---------------------------------------------------------
% FIND CG COLUMN (xm)
%% ---------------------------------------------------------
cgCol = vars(contains(lower(vars),"cg") & contains(lower(vars),"location"));
if isempty(cgCol)
    error("No CG location column found.");
end
cgCol = cgCol(1);
xm = T.(cgCol);

%% ---------------------------------------------------------
% REMOVE TOTAL ROW
%% ---------------------------------------------------------
valid = ~contains(lower(names),"total");

names = names(valid);
xm    = xm(valid);
m     = m(valid);

%% ---------------------------------------------------------
% USER INPUT: STATION LOCATION x
%% ---------------------------------------------------------
x = 230;      % <-- CHANGE THIS VALUE (cm from nose)

%% ---------------------------------------------------------
% CALCULATIONS
%% ---------------------------------------------------------
dx = xm - x;             % (xm – x)
momentTerm = m .* dx;    % m(xm – x)

%% ---------------------------------------------------------
% FULL TABLE (ALL COMPONENTS)
%% ---------------------------------------------------------
fullTable = table(names, m, xm, dx, momentTerm, ...
    'VariableNames', ["Component","Mass_g","xm_cm","xm_minus_x_cm","m_dx"]);

% sort by absolute distance from x
```

```
[~,idx] = sort(abs(dx));
fullTable = fullTable(idx,:);

disp(" ")
```

```
disp("===== ALL COMPONENTS =====")
```

```
===== ALL COMPONENTS =====
```

```
disp(fullTable)
```

Component	Mass_g	xm_cm	xm_minus_x_cm	m_
"arc clamp system"	97.567	225.21	-4.7909	-
"arc clamp system"	97.6	240.93	10.931	
"nuts & washer"	45.359	242.18	12.181	
"Bulkhead"	141.1	242.36	12.365	
"Avionics Module (Third)"	2031.2	214.78	-15.217	
"arc clamp system"	97.567	213.19	-16.805	-
"avionics guts wooden frame and 3d printed spacer"	226.8	210.5	-19.498	
"threaded rod"	163	258.71	28.711	
"Main Can Nylon"	228.56	267.78	37.777	
"FG Body (second) Module"	1549.5	269.23	39.23	
"arc clamp system"	97.6	179.77	-50.232	-
"Bulkhead"	141	179.1	-50.903	-
"Umbilical Internals cameras, battery, board, wires, battery case"	195.61	173.7	-56.305	
"Camera Passthrough Micromodules"	3089.9	171.66	-58.337	-1.80
"Bulkhead"	141	171.12	-58.881	-
"midsection arc clamp system"	97.6	170.55	-59.452	-
"Bulkhead"	141	169.98	-60.023	-
"Argus Camera System"	572.66	164.5	-65.497	
"Bulkhead"	141	162	-68	-
"arc clamp system"	97.6	302.15	72.145	
"arc clamp system"	97.6	305.85	75.854	
"ERS Micromodule Attched to Main"	2046.2	154	-76	-1.55
"ERS Mechanism"	0	151.31	-78.686	
"Bulkhead"	141	318.93	88.935	
"arc clamp system"	97.6	322.95	92.948	
"Main Parachute"	1111	134.43	-95.571	-1.06
"Trapezoidal fin module"	1147.2	326.18	96.179	1.10
"Main Can Isogrid Module"	6476.8	133.07	-96.933	-6.27
"15227N2501-P"	13308	330.91	100.91	1.34
"Main Shock cord"	5.395	118.18	-111.82	-
"Fin Can Module"	3764.4	348.2	118.2	4.44
"ERS Micromodule Attached to Nose Cone"	2048.4	109.6	-120.4	-2.46
"ERS Mechanism"	0	106.91	-123.09	
"Drogue Can"	517.67	89.04	-140.96	
"drogue Parachute"	188.26	88.9	-141.1	
"Nose cone"	4337.7	82.899	-147.1	-6.38
"Bulkhead"	141.1	377.23	147.23	
"Motor Centering Ring"	156.59	378.2	148.2	
"arc clamp system"	97.6	378.83	148.83	
"Thrust Flange"	270.3	379.29	149.29	
"Shock Cord"	544.31	74.415	-155.59	
"threaded rod"	163	9.652	-220.35	
"Metal Nose Cone Tip"	240	4.9212	-225.08	

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

```
% IMPORTANT COMPONENT FILTER
%% ----------------------------------------------------------
importantKeywords = [
    "nose"
    "fin"
    "motor"
    "engine"
    "payload"
    "avionics"
    "recovery"
    "parachute"
    "tube"
    "arc rings"
];

importantMask = false(size(names));

for k = 1:length(importantKeywords)
    importantMask = importantMask | contains(lower(names),importantKeywords(k));
end

importantTable = fullTable(importantMask(idx),:);

disp(" ")
```

```
disp("===== IMPORTANT COMPONENTS ONLY =====")
```

```
===== IMPORTANT COMPONENTS ONLY =====
```

```
disp(importantTable)
```

Component	Mass_g	xm_cm	xm_minus_x_cm	m_dx
"Avionics Module (Third)"	2031.2	214.78	-15.217	-30908
"avionics guts wooden frame and 3d printed spacer"	226.8	210.5	-19.498	-4422
"Main Parachute"	1111	134.43	-95.571	-1.0618e+05
"Trapezoidal fin module"	1147.2	326.18	96.179	1.1034e+05
"Fin Can Module"	3764.4	348.2	118.2	4.4496e+05
"ERS Micromodule Attached to Nose Cone"	2048.4	109.6	-120.4	-2.4664e+05
"drogue Parachute"	188.26	88.9	-141.1	-26563
"Nose cone"	4337.7	82.899	-147.1	-6.3808e+05
"Motor Centering Ring"	156.59	378.2	148.2	23207
"Metal Nose Cone Tip"	240	4.9212	-225.08	-54019

```
%% ----------------------------------------------------------
% TOTAL SUMS (for bending equation use)
%% ----------------------------------------------------------
totalMoment = sum(momentTerm);
disp(" ")
```

```
disp("Total Σ m(xm − x) = " + totalMoment + " g·cm")
```

```
Total Σ m(xm − x) = -248890.4398 g·cm
```

Bending Moments: Math for Bending Using Component x-pos

$$M_B(@x_k) = L_{nose}(x_k - x_{nose}) + L_{boat}(x_k - x_{boat}) + L_{fin}(x_k - x_{fin})$$

$$-a_y \sum_0^{x_k} m(x_k - x_m) - R \sum_0^{x_k} m(x_k - x_m)\,(x_{CG} - x_m)$$

```
% x_m - x position(in meters) of the arc clamps in the very back (in m)
% x_m -


% deltXNose  = -1*importantTable(8,4)/100 % in meters, w/ conversion factor cm ->
m,
%         % and then multiplied by -1 to get x_cm - xm
% deltXboat; % currently this is undefined as the boattail is not attached
% deltXfin = -1*importantTable(5,4)/100 % in meters, w/ conversion factor
```

From the table outputted above, as of 2/18/26's revision, the corresponding row with the component that contains these delta x's

```
===== IMPORTANT COMPONENTS ONLY =====
```

Component	Mass_g	xm_cm	xm_minus_x_cm	m_dx
"Avionics Module (Third)"	2031.2	214.78	-15.217	-30908
"avionics guts wooden frame and 3d printed spacer"	226.8	210.5	-19.498	-4422
"Main Parachute"	1111	134.43	-95.571	-1.0618e+05
"Trapezoidal fin module"	1147.2	326.18	96.179	1.1034e+05
"Fin Can Module"	3764.4	348.2	118.2	4.4496e+05
"ERS Micromodule Attached to Nose Cone"	2048.4	109.6	-120.4	-2.4664e+05
"drogue Parachute"	188.26	88.9	-141.1	-26563
"Nose cone"	4337.7	82.899	-147.1	-6.3808e+05
"Motor Centering Ring"	156.59	378.2	148.2	23207
"Metal Nose Cone Tip"	240	4.9212	-225.08	-54019

```
Total Σ m(xm − x) = -248890.4398 g·cm
```

```
% Mbending_backArcClamp = Nnosecone*() + Nbt*(x_m -) +  Nfins*(x_m -) - ay
```

Continuous Gust Profile

From Aspire Space Paper:

"The vehicle accelerates in the direction of the total lift which builds up a lateral airspeed component that reduces the angle of attack. Simultaneously the fins rotate the vehicle to reduce the angle of attack further in a highly damped oscillatory motion."

So looking at the laterial continuous gust profile from Kopasakis(2010) "Atmospheric Turbulence Modeling for Aero Vehicles: Fractional Order Fits"

We assume Umax = 2sigma as a conservative estimate

Getting the table values, picked the one at 5km since it is more conservative of a dvisc. from (Tables of the U.S. Standard Atmosphere, 1976)

:

https://www.pdas.com/bigtables.html

```
% kinematic viscocity
dvisc = 2.2110E-05;

% l aka atmospheric region of size l
% use the same l(dm*2) from the discrete gust simulation above
% l = (dmMQ*2);


% vprime aka velocity fluctuation in atmospheric region side l.
% note that vprime =~u'
% assume 2sigma = Umax
% sigma = Umax/2 = vprime =
% note that from the discrete gust, vm = Umax
vprime = (vm/2)
```

```
Unrecognized function or variable 'vm'.
```

```
% eddy dissipation aka epsilon stuff
epsilon = (dvisc*vprime^2)/(l^2)

s = tf('s');
```

```
% traverse acoustic wave velocity
GVA = ( (56*epsilon^(2/9))*((s/9.2+1)*(s/55.0+1)*(s/335.5+1)) )/( (s/1.46+1)*(s/
30.1+1)*(s/85.7+1)* (s/1593.1+1) )

step(GVA)
```