# Aquila Transonic Buffet and Drag Analysis

**Barty Wardell**

bjw68@cam.ac.uk

October 20, 2021

## Contents

## 1 Introduction

Buffet is a form of vibration, typically found in the wings of aircraft or in the fins of rockets. It is associated with vortices and separated flow. As such, it is prevalent in the transonic regime but not in the supersonic regimes. Given the high accelerations we experience during the flight, it is important that we analyse some of the forces that buffet could potentially develop, as this may influence the design of the fins.

In this document, we also seek to lay out the equations for calculating drag used in the ascent model. This was subsequently used to find values such as Max. Q and peak velocity for the purposes of Aeroelastic modelling. As such, it is of relevance to the design.

## 2 Buffet

### 2.1 References

Our analysis here has made great use of the following sources.

- On the Application of Statistical Concepts to the Buffeting Problem. Journal of the Aeronautical Sciences (Institute of the Aeronautical Sciences), 19(12), 793–800. W.H. Liepmann. (1952)
- An Analytical Approach to the Fuel Sloshing and Buffeting Problems of Aircraft. Journal of the Aeronautical Sciences 14(4). Harold Luskin and Ellis Lapin. (1952)

### 2.2 Theory

Buffet is an immensely complex phenomena, but may be thought of as a high frequency random forced vibration that occurs on fixed fins. This occurs due to impulses on the fin in turbulent flows, due to shock effects and separated airflow.

As such, there are a number of ways to model buffet. However, the vast majority employ CFD and complex coupled structural-fluid calculators, even before turbulence is introduced. For interest, this is usually done using the Unsteady Reynolds averaged Navier Stokes method (URANS).

However, this is often done in order to find the energy transfer rate to the aircraft. This has practical interests but is not the most useful output in our case. Here, we simply want to find the maximum expected displacement due to specified turbulence - which could occur in theory due to the flow over the rocket. However, much more likely, we want to know the response of the system if patches of atmosphere are themselves turbulent.

Using the first paper, this can be found from a statistical approach.

We first define the local angle of attack at the fin due to turbulence, $\alpha$

$$\alpha = \frac{w}{U}, \tag{1}$$

where $w$ is the turbulent velocity and $U$ is the free stream velocity.

We may then assume that all gusts can be modelled sinusoidally, or at least locally sinusoidally, with a given frequency. We then define an equivalent relation of the fin to random sinusoidal input. This relates the frequency to the angle of attack, and thus the turbulent velocity over the fin.

N.B. This function itself is derived from (Sears, W. R., Some Aspects of Nonstationary Airfoil Theory and Its Practical Applications, Journal of the Aeronautical Sciences, Vol. 8, No. 2, p. 104, February, 1941).

We find,

$$\frac{w(x,t)}{U} = \alpha_0 e^{i\omega(t-\frac{x}{U})}. \tag{2}$$

Where, $w$ and $U$ are as before. Then $t$ is simulation time, $\alpha_0$ is the initial angle of attack at $t = 0$ at the tip of the fin. Then $x$ is the distance parameter. Unfortunately, the paper only deals in one dimension - however given the scarcity of analytical approaches in this field and the complexity of CFD, we may use this later to somewhat generalise the result.

We then need to define the input to the fin system in the form of a power spectrum which relates, free-stream velocity, turbulent velocity with frequency and other parameters. Luckily, the first paper provides an energy spectrum which encapsulates the bulk of the energy.

We see,

$$f(\omega) = \frac{\overline{w^2}}{U^2}\frac{L}{\pi U}\frac{1+3(\frac{L^2\omega^2}{U^2})}{(1+(\frac{L^2\omega^2}{U^2}))^2}. \tag{3}$$

Where $f(\omega)$ is the power spectrum, $\omega$ is the angular frequency of the turbulent waves. Then, $L$ is the scale of turbulence, which acts like a representation of eddy size.

We then wish to find the response of the fin aeroelastically for a generalised input of impulse. This acts a little like a transfer fucntion. From the first source, we again find the displacement $y$,

$$\overline{y^2} = 4\pi^2\frac{\pi f(\omega_0)}{2\beta\omega_0(1+\pi(\frac{\omega_0 c}{U}))} \tag{4}$$

where $c$ is the aerofoil chord, and $\beta$ is a constant from the fin impedance. For aeroelastic effects we may find a 'structural impedance' of the fin to the input, which takes the form,

$$z = (\omega^2-\omega_0^2)^2+\beta^2\omega^2. \tag{5}$$

We can use this, along with the equation for the power spectrum from earlier, in order to find the displacement of the fin in one direction to a given turbulence. Overall, we find,

$$\overline{y^2} = \frac{\overline{w^2}}{U^2}\frac{2\pi L}{U}\frac{1+3(\frac{L^2\omega^2}{U^2})}{(1+\pi(\frac{\omega_0 c}{U}))(1+(\frac{L^2\omega^2}{U^2}))^2} \tag{6}$$

We can then generalise this by assuming the worst case occurs with identical turbulence in all three coordinate directions, where $c$ varies between the three models such that they are normally distributed,

$$y_T^2 = y_x^2 + y_y^2 + y_z^2. \tag{7}$$

We can then iterate over the possible inputs and find the corresponding worst case displacement. We find the worst case has the following characteristics.

- Free-stream velocity: 238 m/s
- Turbulent length scale = 0.16
- Angle of attack = 5 degrees
- Natural fin frequency = 137 Hz
- Maximum displacement = 1.84 mm

As discussed in our aeroelastic report, this deflection may be survived by our fins.

## 3 Drag Calculator

For modelling velocity, it was required to built a simulator which better modelled the likely flight characteristics of the model. Using thrust data etc. are computational issues, and as such will not be dealt with here. However, the drag model used most commonly is not suitable for our purposes we must compute a $C_d$ which varies with Mach number.

The exact drag models employed are laid out below, so they may be understood or re-implemented in the future.

### 3.1 References

Throughout, we heavily used the Open Rocket Drag implementation as a guide for our work. This may be found https://openrocket.info/documentation.html. Development of an Open Source model rocket simulation software (Helsinki University of Technology),Sampo Niskanen. (2009).

This report does not attempt to explain the theory behind these calculations, as many are just functions fitted to real data. However, it allows readers to understand what is included in the model, and what has been neglected.

### 3.2 Skin Friction

We begin this by calculating the Reynold's number for the flow over the body. This may be found by,

$$R = \frac{v_0 x}{\nu}. \tag{8}$$

Where, $v$ is the free-stream velocity, $x$ is length of the rocket and $\nu$ is the kinematic viscosity. For air, we find that $\nu = 1.5 \times 10^{-5}$. Regarding Reynold's number, we can also compute a critical Reynold's number
$R_{crit} = 51\Big(\frac{R_s}{L}\Big)^{-1.039}$,
where $R_s$ is the approximate roughness height of the surface, $L$ is the body length, and $R_{crit}$ represents a critical Reynolds number which dictates which set of equations is subsequently used, and is related to whether the flow is turbulent or laminar. Depending on the result we then use one of three equations.

If $R < 10^4$, then for the coefficient of skin friction can to be $C_f = 1.48 \times 10^{-2}$.

For $10^4 < R < R_{crit}$,
we can use $C_f = \frac{1}{(1.5\ln(R) - 5.6)^2}$.
Finally, for $R > R_{crit}$,
we find $C_f = 0.032\Big(\frac{R_s}{L}\Big)^{0.2}$.

#### 3.2.1 Correction for Comparability

We must then correct the $C_f$ term as the air is compressible at higher mach numbers ($> 0.3$). This is demonstrated below.

At subsonic velocities, we find the corrected skin coefficnet $C_{fc}$,

$C_{fc} = C_f(1 - 0.1M^2)$, where $M$ is the Mach number.

Then for supersonic flow there are two equations, the first is turbulence limited and the second roughness limited.
For turbulence, $C_{fc} = \frac{C_f}{1 + 0.15M^2}^{0.58}$.
For roughness, $C_{fc} = \frac{C_f}{1 + 0.18M^2}$.
In order to apply this, we use whichever coefficient yields a greater value in the supersonic region.

We then perform a area correction to find the total coefficient of skin friction, $C_{Df}$,

$$C_{Df} = C_{fc}\frac{(1 + \frac{1}{2f_B})A_{bodywet} + (1 + \frac{2t}{\bar{c}})A_{finswet}}{A_{ref}}. \tag{9}$$

Where, $f_b$ is the fineness ratio of the rocket which is the ratio of the length of the body divided by its width, $t$ is the fin thickness, $A_{wet}$ is the area in contact with the airflow in both the case of the body and the fins. Finally $\bar{c}$ is the mean aerodynamic chord length of the rocket. Then, $A_{ref}$ is the cross sectional area of the body.

### 3.3 Nosecone Pressure Drag

Well designed nosecones may produce a drop in the overall pressure drag of the rocket. Thus, since an ogive nosecone is used commonly we take the pressure coefficient of the nosecone to be $C_{np} = 0$ for $M < 0.8$.

In the transonic regime ($0.8 < M < 1.3$), we find the coefficient for nosecone pressure, $C_{np} = \sin(\epsilon)$.

Where, $\epsilon = \dfrac{d}{2l}$.

Here, $l$ is the nosecone length and $d$ is the maximum nosecone diameter. In general, $\epsilon$ is the half-apex angle of the nosecone.

In the supersonic regime, $M > 1.3$, we find,

$C_{np} = 2.1\sin(\epsilon)^2 + 0.5\dfrac{\sin(\epsilon)}{\sqrt{M^2-1}}$.

### 3.4 Fin Pressure Drag

We are using fins that are specified as per NACA 0012. This means they possess an aerofoil shape, with a rounded edge and tapered rear edge. This greatly simplifies this analysis, and helps to lower the drag of the rocket.

At this point the Open Rocket documentation directly quotes Barrowman. We will also use his equation for pressure drag at the leading edge of the fin.

We find, for $M < 0.9$, the coefficient of fin leading edge pressure drag, $C_{LE}$,

$C_{LE} = (1 - M^2)^{-0.417} - 1$.

As before, $M$ is the Mach number.

Then, in the transonic regime, $0.9 < M < 1$,

$C_{LE} = 1 - 1.785(M - 0.9)$.

Finally for $M > 1$,

$C_{LE} = 1.214 - \dfrac{0.502}{M^2} + \dfrac{0.1095}{M^4}$.

However, all of this analysis assumes that the fin edge is parallel to the flow. Thus, we must then correct for the slanted edge of the fin. Overall, $C_f t$,

$C_f t = C_{LE}\cos(\Gamma_L)^2$. Here, $\Gamma_L$ is the average leading edge angle.

We then find that in the case of a fin with a tapered edge, the base pressure drag is equal to zero. Thus, we have found the overall pressure drag coefficient for the fin.

### 3.5 Base Pressure Drag

The end of the rocket creates a low pressure zone, which itself creates drag on the rocket. We find find the following relationships allow us to find the coefficient of base pressure drag, $C_B$.

For $M < 1$,

$C_B = 0.12 + 0.13M^2$.

Then, for $M > 1$,

$C_B = \dfrac{0.25}{M}$

Thus, we have found all the important coefficients of drag for the rocket.

### 3.6 Total Drag

We may combine the coefficient of drag using the following formula, in order to find an overall coefficient of drag for the rocket $C_D$.

We find,

$$C_D = C_{Df} + C_{np} + C_B + C_{LE}\frac{A_T}{A_{ref}} \tag{10}$$

We must remember to account for both fin sets when implementing these equations. In the case of the fins, we find $A_T$, $A_T = 4ts$. Here $t$ is the fin thickness and $s$ is the fin height.

## A Code

The code is reproduced here, simply so the reader can see the implementation of the above functions. They have not been fully explained here and contain additional functionality to what is derived above.

### A.1 Buffet Code

```
import numpy as np

span = 0.2
chord = 0.16
thickness = 0.004
velocity = np.linspace(238,240, 3)
time = np.linspace(0,10, 100)
angle = [1, 2, 3, 4, 5]
freq = np.linspace(0,1000, 100)
distance = np.linspace(0, chord, 100)

def L(c):
  #C is the chord at a distance y
  scale = np.linspace(1,9, 9)
  l = c*scale
  return l

L = L(chord)

def ysfactor(L, U, c, omega_o):
  y_factor = 2*(1/(U**2))*(np.pi * L/U)*((1+(3*(L**2)*(omega_o**2)/(U**2))))/\
             ((1+(np.pi*omega_o*c/U))*((1+((L**2)*(omega_o**2)/(U**2)))**2))
  return y_factor

def wsquare(alpha_0, omega, t, x, U):
  w = alpha_0 * complex(np.cos(omega*(t - x/U)), np.sin(omega*(t - x/U)))
  w_2 = w**2
  return w_2

omega0 = 137 * 2 * np.pi

alpha = 5
for l in L:
    for U in velocity:
        max_y = 0
        frequency = 0
        for f in freq:
            max_w2 = 0
            for x in distance:
                for t in time:
                    omeg = 2*np.pi*f
                    w_out = wsquare(alpha, omeg, t, x, U)
                    max_w2 += w_out
                iterations = len(time)*len(distance)
                average = max_w2/iterations
                y_factor = ysfactor(l, U, chord, omega0) + ysfactor(l, U, span, omega0)\
                           + ysfactor(l, U, thickness, omega0)
                y = np.sqrt(y_factor*average)
                if y > max_y:
```

```
                max_y = y
                frequency = f
print(alpha, l, U, max_y)
```

## A.2 Ascent Code

```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01  # Change this to alter the timestep
iterations = 25000  # Change this to alter how many timesteps are calculated (Should be in

cddown = 0.4  # Change this to alter the Cd on descent
adown = 0.01539  # Change this to alter the effective area on descent
Length = np.arange(241)
A_Fin_base = 0.052
A_Fin_top = 2.4e-3
fin_thickness_base = 4e-3
fin_thickness_top = 1e-3
MAC_base = 27.282e-2
MAC_top = 8.16667e-2
Leadingangle_base = 26.6
Leadingangle_top = 33.7
Roughness =3e-6
totatlen = 2.4
noseconelen = 40e-2
maximumdiameter = 13e-2
A_ref = np.pi * (maximumdiameter/2)**2
A_body_wet = 2*np.pi*(maximumdiameter/2)*totatlen
A_fin_wet_base = 8 * A_Fin_base
A_fin_wet_top = 8 * A_Fin_top
fineness = totatlen/maximumdiameter

cddrogue = 0.5  # Change this to alter the Cd of the drogue
adrogue = 0.0818 # Change this to alter the area of the drogue
amain = 1 # Change this to alter the area of the main chute
cdmain = 1.2  # Change this to alter the area of the main chute
drogueheight = 10000  # Change this to alter the drogue deploy height
mainheight = 1000  # Change this to alter the main chute deploy height
droguedeploytime = 10  # Change this to alter the drogue deploy time
maindeploytime = 10  # Change this to alter the main chute deploy time
ratio = 0.6609  # Change this to alter the mass ratio (Total dry mass/ Total wet mass)
mass = [30]  # Change to alter the initial wet mass
densities = [1.225, 1.112, 1.007, 0.9093, 0.8194, 0.7364, 0.6601, 0.5900, 0.5258, 0.4671,
# Air density data
altitudes = [0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 15000, 20000,

time = []
droguecount = 0  # Variables for deploying times
maincount = 0

# Reads the interpolated thrust data for use, then formats it and sorts out the timebase
g = open("C:/Users/barty/PycharmProjects/CUSF/Altitude␣Analysis/thrustdata3.txt", "r")
force = g.read().split("\n")
g.close()
burntime = round(len(force) * dt, 2)
for i in range(0, iterations - len(force)):
  force.append(0)
for i in range(0, iterations + 100):
  time.append(dt * i)
force = np.array(force)
totalforce = 0
for forces in force:
  totalforce += float(forces)
time = np.array(time)

drag = np.zeros(iterations + 100)  # the +100 is to make sure all lists are longer than nee
height = np.zeros(iterations + 100)
velocity = np.zeros(iterations + 100)
acceleration = np.zeros(iterations + 100)
bfin_drag = np.zeros(iterations + 100)
dynamic_pressure = np.zeros(iterations + 100)

height[0] = 10000
velocity[0] = 824

def Reynolds(length, velocity, viscosity = 1.5e-5):
    R = (length * velocity)/viscosity
    return R

def R_crit(Roughness, Length):
    R_crit = 51 * (Roughness/Length)**(-1.039)
    return R_crit


for i in range(0, iterations + 100):
  mass.append((1 - ratio)*mass[0])
mass = np.array(mass)
force = np.array(force)

for i in range(0, iterations):

  if height[i] >= 0:

    for j in range(0, len(altitudes)):
       if height[i] >= altitudes[j] and height[i] < altitudes[j+1]:
        density = densities[j] + ((densities[j+1] - densities[j]) * ((height[i] - altitudes
        break

    if velocity[i] >= 0:
      #drag[i] = cdup * ((velocity[i]) ** 2) * 0.5 * density * aup
      Reynold = Reynolds(totatlen, velocity[i])
      R_critical = R_crit(Roughness, totatlen)
      Coeff_friction = 0
      if Reynold <= 1e4:
          Coeff_friction = 1.48e-2
      elif Reynold > 1e4 and Reynold < R_critical:
          Coeff_friction = ((1.5*np.log(Reynold))-5.6)**(-2)
      elif Reynold >= R_critical:
          Coeff_friction = 0.032 * ((Roughness/totatlen)**0.2)


      Machno = velocity[i]/295
      Skin_friction_coeff = 0
      if Machno < 0.8:
          Skin_friction_coeff = Coeff_friction * (1 - ((0.1) * (Machno ** 2)))
      elif Machno > 0.8:
          Supersonic_skin_friction_1 = Coeff_friction * (1 + ((0.15) * (Machno ** 2)))**(-
          Supersonic_skin_friction_2 = Coeff_friction * (1 + ((0.18) * (Machno ** 2)))**(-
          if Supersonic_skin_friction_1 > Supersonic_skin_friction_2:
              Skin_friction_coeff = Supersonic_skin_friction_1
          else:
              Skin_friction_coeff = Supersonic_skin_friction_2
      Coeff_friction_total = (Skin_friction_coeff/A_ref) * (((1 + (1/(2 * fineness))) * (A

      #D_friction = Coeff_friction_total * ((velocity[i]) ** 2) * 0.5 * density * (A_ref)

      Coeff_body_pressure = 0
      if Machno < 0.8:
          Coeff_body_pressure = 0
      elif Machno < 1.3 and Machno >= 0.8:
          eta = np.arctan(maximumdiameter/(2*noseconelen))
          Coeff_body_pressure = np.sin(eta)
      elif Machno >= 1.3:
          eta = np.arctan(maximumdiameter/(2*noseconelen))
          Coeff_body_pressure = (2.1 * ((np.sin(eta))**2)) + ((np.sin(eta))/(2 * np.sqrt((

      #D_body_pressure = Coeff_body_pressure * ((velocity[i]) ** 2) * 0.5 * density * A_re

      Coeff_fin_pressure_LE = 0  #Assuming a rounded leading edge
      if Machno < 0.9:
          Coeff_fin_pressure_LE = ((1 - (Machno**2))**(-0.417)) - 1
      elif Machno >= 0.9 and Machno <1:
          Coeff_fin_pressure_LE = 1 - (1.785 * (Machno-0.9))
      elif Machno >= 1:
          Coeff_fin_pressure_LE = 1.214 - (0.502/(Machno**2)) + (0.1095/(Machno**4))

      Coeff_fin_pressure_LE_base = Coeff_fin_pressure_LE * (np.cos(Leadingangle_base * np.
      Coeff_fin_pressure_LE_top = Coeff_fin_pressure_LE * (np.cos(Leadingangle_top * np.pi

      Coeff_fin_pressure_TE = 0
      if Machno < 1:
          Coeff_fin_pressure_TE = 0.5*(0.12 + (0.13 * (Machno**2)))
      elif Machno >= 1:
          Coeff_fin_pressure_TE = 0.5*(0.25/Machno)

      Coeff_fin_pressure_base = Coeff_fin_pressure_LE_base
      Coeff_fin_pressure_top = Coeff_fin_pressure_LE_top

      D_fin_pressure_base = Coeff_fin_pressure_base * ((velocity[i]) ** 2) * 0.5 * density
      #D_fin_pressure_top = Coeff_fin_pressure_top * ((velocity[i]) ** 2) * 0.5 * density


      Coeff_base_drag = Coeff_fin_pressure_TE
      #D_base = Coeff_base_drag * ((velocity[i]) ** 2) * 0.5 * density * A_ref

      #print("Friction: {}, Body pressure: {}, Fin base pressure: {}, Fin Top Pressure: {}

      Coefficient_of_drag = Coeff_friction_total + (Coeff_base_drag * A_ref/A_ref) + (Coef

      #drag[i] = D_friction + D_body_pressure + D_fin_pressure_base + D_fin_pressure_top +
      drag[i] = Coefficient_of_drag * ((velocity[i]) ** 2) * 0.5 * density * A_ref
      bfin_drag[i] = (D_fin_pressure_base/4)
      #print(time[i], height[i], velocity[i], bfin_drag[i])

    elif velocity[i] < 0:

      if height[i] > drogueheight:
        drag[i] = -1 * cddown * ((velocity[i]) ** 2) * 0.5 * density * adown
      elif height[i] < drogueheight and height[i] > mainheight:
        if droguecount <= droguedeploytime/dt:
          droguecount += 1
        drag[i] = -1 * cddrogue * ((velocity[i]) ** 2) * 0.5 * density * (droguecount/(drog

      elif height[i] < mainheight:
        if maincount <= maindeploytime/dt:
          maincount += 1
        drag[i] = -1 * cdmain * ((velocity[i]) ** 2) * 0.5 * density * (maincount/(maindep

    if float(dt*i) <= burntime:
      mass[i+1] = mass[i] - (float(force[i])/totalforce) * (ratio) * mass[0]
    acceleration[i+1] = (float(force[i]) - float(drag[i]))/float(mass[i]) - 9.81
# Gravity
    velocity[i+1] = float(velocity[i]) + float(acceleration[i]) * dt
    height[i+1] = float(height[i]) + float(velocity[i]) * dt

    dynamic_pressure[i+1] = density * (float(velocity[i]**2))

plt.plot(time[0:len(height):10], height[0:len(height):10])
plt.legend("height")
plt.show()

plt.plot(time[0:len(height):10], velocity[0:len(height):10])
plt.legend("velocity")
plt.show()

plt.plot(time[0:len(height):10], acceleration[0:len(height):10])
plt.legend("acceleration")
plt.show()

plt.plot(time[0:len(height):10], drag[0:len(height):10])
plt.legend("drag")
plt.show()

plt.plot(time[0:len(height):10], mass[0:len(height):10])
plt.legend("mass")
plt.show()

plt.plot(time[0:len(height):10], dynamic_pressure[0:len(height):10])
plt.legend("q␣dynamic␣pressure")
plt.show()


maxheight = np.sort(height)
print("Maximum␣height:␣", maxheight[-1])
maxvelocity = np.sort(velocity)
print("Maximum␣velocity:␣", maxvelocity[-1])
maxaccel = np.sort(acceleration)
print("Maximum␣acceleration:␣", maxaccel[-1])
maxbfin = np.sort(bfin_drag)
print("Maximum␣drag:␣", maxbfin[-1])
maxq = np.sort(dynamic_pressure)
print("")
print("Maximum␣Dynamic␣Pressure:", maxq[-1])

maxq = 0
for i in range(len(dynamic_pressure)):
    if float(dynamic_pressure[i]) > float(dynamic_pressure[maxq]):
        maxq = i
```

```
print("/nMax␣Q␣occurs␣at␣an␣altitude␣of", height[maxq], "and␣a␣speed␣of", velocity[maxq])

with open('Velocity.txt', 'w') as f_vel:
    for item in velocity:
        f_vel.write("%s\n" % item)

with open('Height.txt', 'w') as f_height:
    for item in height:
        f_height.write("%s\n" % item)

with open('OutputTime.txt', 'w') as f_time:
    for item in time:
        f_time.write("%s\n" % item)
```