TRANSIENT ANALYSIS OF A PULSED DETONATION COMBUSTOR USING THE NUMERICAL PROPULSION SYSTEM SIMULATION

By

Anthony Scott Hasler

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Aerospace Engineering

The University of Texas at Arlington

December 2012

Abstract

TRANSIENT ANALYSIS OF A PULSED DETONATION COMBUSTOR USING THE NUMERICAL PROPULSION SYSTEM SIMULATION

Anthony Scott Hasler, M.S.

The University of Texas at Arlington, 2012

Supervising Professor: Donald Wilson

The performance of a hybrid mixed flow turbofan (with detonation tubes installed in the bypass duct) is investigated in this study and compared with a baseline model of a mixed flow turbofan with a standard combustion chamber as a duct burner. Previous studies have shown that pulsed detonation combustors have the potential to be more efficient than standard combustors, but they also present new challenges that must be overcome before they can be utilized. The Numerical Propulsion System Simulation (NPSS) will be used to perform the analysis with a pulsed detonation combustor model based on a numerical simulation done by Endo, Fujiwara, et. al. Three different cases will be run using both models representing a take-off situation, a subsonic cruise and a supersonic cruise situation. Since this study investigates a transient analysis, the pulse detonation combustor is run in a rig setup first and then its pressure and temperature are averaged for the cycle to obtain quasi-steady results.

Table of Contents

List of Symbols

ρ – Density

p – Pressure

γ – Specific Heat Ratio

q – Heat Addition

u_{CJ} – Chapman-Jouguet Detonation Wave Velocity

a – Speed of Sound in a Fluid

$C_P \equiv \frac{\partial h}{\partial T}\Big|_P$ – Specific Heat at Constant Pressure

$C_V \equiv \frac{\partial e}{\partial T}\Big|_V$ – Specific Heat at Constant Volume

M_{CJ} – Chapman-Jouguet Mach Number

h – Enthalpy

A_{tube} – Detonation Tube Cross–Sectional Area

l_{tube} – Tube Length

MFP – Mass Flow Parameter

$TSFC$ – Thrust Specific Fuel Consumption

List of Abbreviations

PDE – Pulsed Detonation Engine

PDC – Pulsed Detonation Combustor

NPSS – Numerical Propulsion System Simulation

CJ – Chapman – Jouguet

ZND – Zeldovich, Von Neumann and Döring

DDT – Deflagration to Detonation Transition

CEA – Chemical Equilibrium with Applications

SLS – Sea Level Static

List of Figures

List of Tables

Chapter 1

Introduction

In the past ten to fifteen years, a significant amount of research has been done in trying to make detonation a feasible means of increasing the efficiency of combustion engines for aerospace propulsion. Since a detonation wave provides both the heat release due to chemical combustion as well as a large pressure rise, it has the potential to be much more efficient than a standard combustion engine as well as removing the necessity of having a large series of compressor blades in order to compress the fuel-air mixture. In addition to reducing the complexity of designing an efficient compressor, reducing the amount of delicate moving parts would also lead to lower maintenance costs and lower maintenance time on the engine itself.

Although detonation engines are promising in these aspects, there are still technical problems which must be solved in order to make a detonation engine viable. When a fuel-air mixture is burned in a tube with both ends open, the deflagration wave will move with a constant velocity, usually small compared to the sonic velocity of the gas (order of 0.01%). However, if one end is closed and a deflagration wave is ignited at this end the combustion wave will be observed to accelerate until it reaches the speed of a detonation wave. This is the “thermal initiation” of a detonation wave and the overall process is known as deflagration to detonation transition (DDT). In experiments, it has been shown that the DDT process can be sometimes difficult to induce and there is not yet a good way of predicting whether a transition will take place with a specific geometry and chemistry. Another large problem with detonation engines is the noise that they produce. With traditional engines, steps are already taken to reduce the amount of noise produced by the

engine exhaust, and a detonation engine produces *significantly* more noise than a traditional turbofan/turbojet engine.

In the aerospace community, it is hoped that a hybrid of a detonation combustor within a more traditional turbomachinery setup will result in an engine that is both quieter than a pure detonation engine and more efficient than a pure turbomachine/combustion engine.

1.1 Purpose

The purpose of this thesis is to create a model of a pulse detonation combustor (henceforth PDC), integrate the PDC into a turbofan engine in the bypass ducts and then compare the PDC/Turbofan hybrid to a turbofan engine with a traditional combustor in the bypass ducts. The goal of the comparison is to demonstrate the potential benefit of detonation combustion over traditional deflagration combustion, specifically taking advantange of the self-compressing nature of detonation in the bypass ducts (i.e. flow not coming out of a compressor).

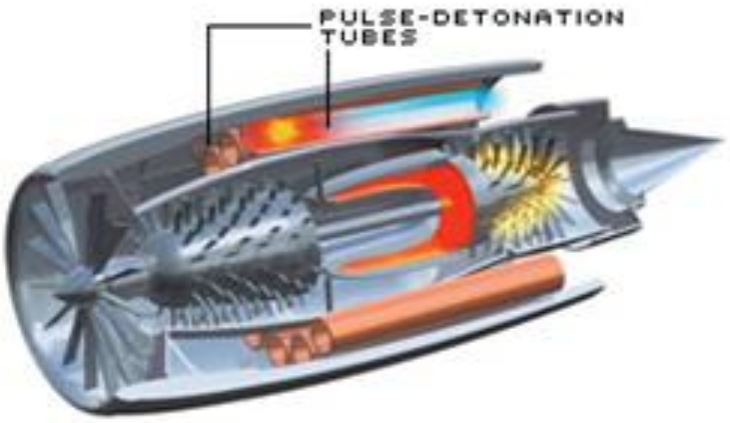

Figure 1.1 – Detonation Tubes in the bypass section of a turbfan engine (Ref. 5).

1.2 Procedure

The Numerical Propulsion System Simulation (NPSS) was developed in a cooperative effort between the major aerospace industry companies and NASA to predict and analyze the performance of air breathing engines in various configurations, and can easily be modified and tweaked by the user (Ref. 11). NPSS will be the tool used to model and analyze the performance of the hybrid PDC turbofan and the "baseline" turbofan engine with a conventional combustor. Additionally, NASA's Chemical Equilibrium with Applications (CEA) will be used to calculate the upper Chapman-Jouguet conditions based on an initial pressure, temperature and chemical composition (Ref. 10). For this thesis, the fuel will be liquid Hydrogen (H_2) and the oxidizer will be air from the atmosphere (Std. Air by molar fraction: O_2 + 3.76 N_2, CEA will include trace elements as well).

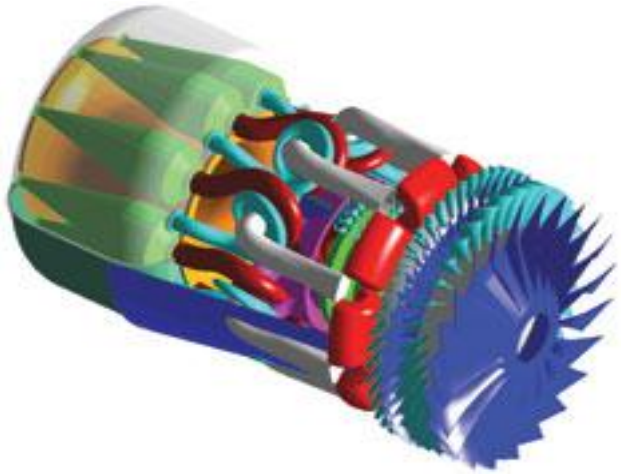

Figure 1.2 – A concept for a hybrid engine combining conventional turbomachinery with pulsed detonation combustion (Ref. 5).

Chapter 2

Thermodynamic Models of Detonation

2.1 Thermodynamic Models

2.1.1 Chapman-Jouguet

Experimentally, it has been found that the detonation velocity is uniquely constant for a given mixture (Ref. 1). By using the conservation of mass, momentum and energy the Rayleigh line can be derived (Ref. 1):

$$u_1^2 = \frac{1}{\rho_1^2}\left(\frac{P_2 - P_1}{\frac{1}{\rho_1} - \frac{1}{\rho_2}}\right) \ (Rayleigh\ Line)$$

State 1 corresponds to the upstream, unburnt gases and state 2 refers to the downstream, burned gases. After adding the equation of state, the Rankine-Hugoniot relationship is given by (Ref. 1):

$$\frac{\gamma}{\gamma - 1}\left(\frac{P_2}{\rho_2} - \frac{P_1}{\rho_1}\right) - \frac{1}{2}(P_2 - P_1)\left(\frac{1}{\rho_1} + \frac{1}{\rho_2}\right) = q\ (Hugoniot\ curve)$$

Where the heat release per unit mass is given by the enthalpy difference between states 1 and 2:

$$q = h_1^o - h_2^o\ (heat\ release\ per\ unit\ mass)$$

$$h \equiv C_p T + h^o (total\ enthalpy\ per\ unit\ mass)$$

The intersection of the Raleigh line and the Hugoniot curve are the upper and lower Chapman-Jouguet (CJ) points. By controlling the value of heat release q we can control the offset of the Hugoniot curve, and therefore change the CJ wave velocity. When the pressure behind the combustion wave P_2 is plotted as a function of specific volume $\frac{1}{\rho_2}$, the shock Hugoniot curve is the result:

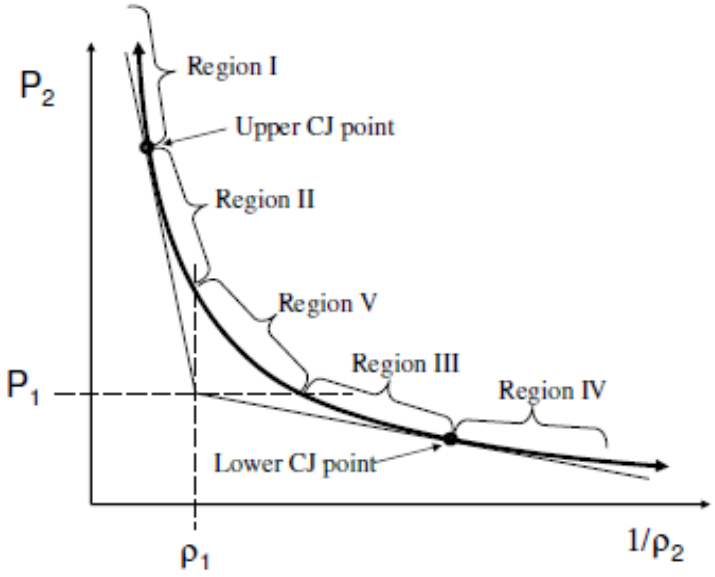

Figure 2.1 – Diagram of Hugoniot Curve and Rayleigh Line

There are five regions on the Hugoniot curve. Region V can be immediately eliminated since it would require a transition from subsonic to supersonic flow across a normal shock, which is impossible in a constant area duct. Regions III and IV are weak and strong deflagrations, respectively, and give expansion waves (i.e. $P_2 < P_1$). Finally, Regions I and II are strong and weak detonations giving compression waves with high velocities. Of primary interest to this work are the detonation regions, specifically, the upper CJ point between regions I and II. At the CJ points, the following is true (Ref. 1):

$$\frac{P_2 - P_1}{\frac{1}{\rho_2} - \frac{1}{\rho_1}} = -\gamma \rho_2 P_2$$

When combined with

$$\rho_1^2 u_1^2 = \frac{P_2 - P_1}{\frac{1}{\rho_2} - \frac{1}{\rho_1}} = \dot{m}^2$$

Results in the following (Ref. 1):

$$u_2^2 = \frac{\gamma P_2}{\rho_2} = a_2^2 \; or \; |u_2| = a_2$$

Where a_2 is the speed of sound in the downstream gases. In region II ("weak" detonations) the wave speed is below $M_2 = 1$ which is rarely observed. Rather, we will focus on wave speeds corresponding to the upper CJ point and above ($M_2 \geq 1$).

2.1.2 Zeldovich-von Neumann-Döring

The ZND theory states that the detonation wave consists of a planar shock (1 dimensional) that moves at the detonation velocity and leaves heated and compressed gas behind it. After a short induction period during which the gas is adiabatically compressed, the chemical reactions begin and the temperature rises and the pressure and density fall to the CJ values. The short pressure spike before the chemical reactions start is called the Von Neumann spike.

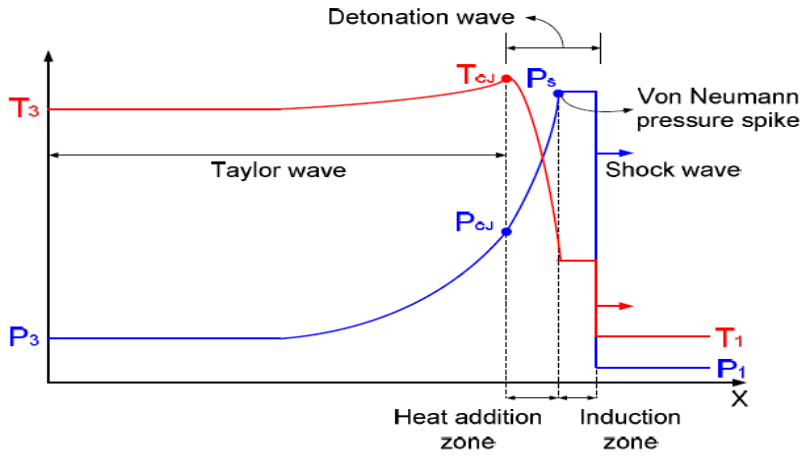

Figure 2.2 – Diagram showing the ZND Model of Detonation (Ref. 8).

In order to visualize the ZND theory of detonation on the Hugoniot, first start with unreacted, undisturbed gas (i.e. q = 0) into which the shock propagates, heating it up to the Von Neumann spike at which point the chemical reactions begin and moves the gas to the upper CJ point at a new Hugoniot curve with a new value for heat release. Along the path followed by the gas from the initial point to the CJ point if the pressure and temperature after the 'jump' from the initial point to the Von Neumann spike are not high enough, the detonation may not have enough energy to sustain itself. In other words, the heat released by combustion must be enough to keep the wave propagating at sonic velocity in order to maintain a detonation. It is important to note that the sonic velocity (i.e. M = 1) occurs behind the detonation wave in the moving frame of reference, and in the stationary (or "lab") reference frame, the flow following the detonation wave can still be supersonic.

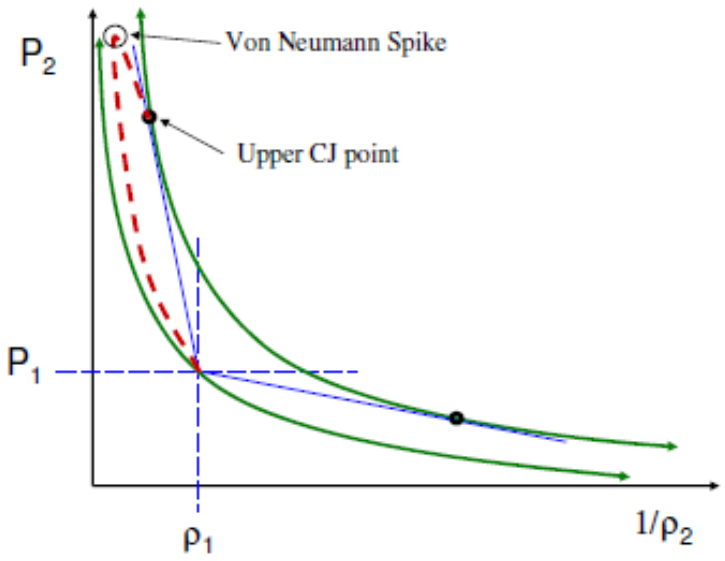

Figure 2.3 – Diagram showing the transition of the gas from undisturbed to Von Neumann to the upper Chapman-Jouguet point (Refs. 4 & 5).

The Von Neumann conditions are calculated by assuming a normal shock moves into the fuel-air mixture at a speed equal to the CJ velocity. Assuming a calorically perfect gas (valid for the upstream gas and the adiabatic transition to Von Neumann spike), the conditions after the Von Neumann spike are given by:

$$\frac{P_1}{P_0} = 1 + \frac{2\gamma_0}{\gamma_0 + 1}(M_0^2 - 1)$$

$$\frac{\rho_1}{\rho_0} = \frac{(\gamma_0 + 1)M_0^2}{2 + (\gamma_0 - 1)M_0^2}$$

$$\frac{T_1}{T_0} = \frac{P_1}{P_0}\frac{\rho_0}{\rho_1}$$

Where state 0 is the undisturbed gas and 1 is the Von Neumann condition. M_0 is the detonation Mach number, calculated using NASA CEA code. It is important to note that the above equations are the normal shock equations assuming a calorically perfect gas, whereas CEA performs the calculations based on a real gas model. For small Mach numbers, the difference between these two models should be small. In this project, the Mach number calculated by CEA was typically in the range of M = 3 – 4, with a maximum of 4.8288 in the (unused) case of a takeoff with a fan pressure ratio of 1.0. For the cases analyzed in the project the Chapman-Jouguet Mach number was less than 4.

Table 2.1 – Pressure Ratio from CEA, CPG and Percent Error.

Upstream Mach No.	**CEA Result**	**CPG Result**	**Percent Error**
1	1.000	1.000	0%
2	4.528	4.500	0.62%
2.5	7.198	7.125	1.01%
3	10.483	10.333	1.40%
3.5	14.390	14.125	1.84%
4	18.926	18.500	2.25%
4.5	24.105	23.4583	2.68%
4.8288	27.867	27.0369	2.98%

Table 2.2 – Density Ratio from CEA, CPG and Percent Error.

Upstream Mach No.	CEA Result	CPG Result	Percent Error
1	1.0003	1.0000	0.03%
2	2.7540	2.6667	3.17%
2.5	3.5235	3.3333	5.40%
3	4.1833	3.8571	7.80%
3.5	4.7475	4.2609	10.25%
4	5.2392	4.5714	12.75%
4.5	5.6816	4.8119	15.31%
4.8288	5.9562	4.9406	17.05%

Table 2.3 – Temperature Ratio from CEA, CPG and Percent Error.

Upstream Mach No.	CEA Result	CPG Result	Percent Error
1	1.0000	1.0000	0%
2	1.6440	1.6875	2.65%
2.5	2.0430	2.1375	4.63%
3	2.5060	2.6790	6.90%
3.5	3.0310	3.3151	9.37%
4	3.6130	4.0469	12.01%
4.5	4.2420	4.8751	14.92%
4.8288	4.6770	5.4724	17.01%

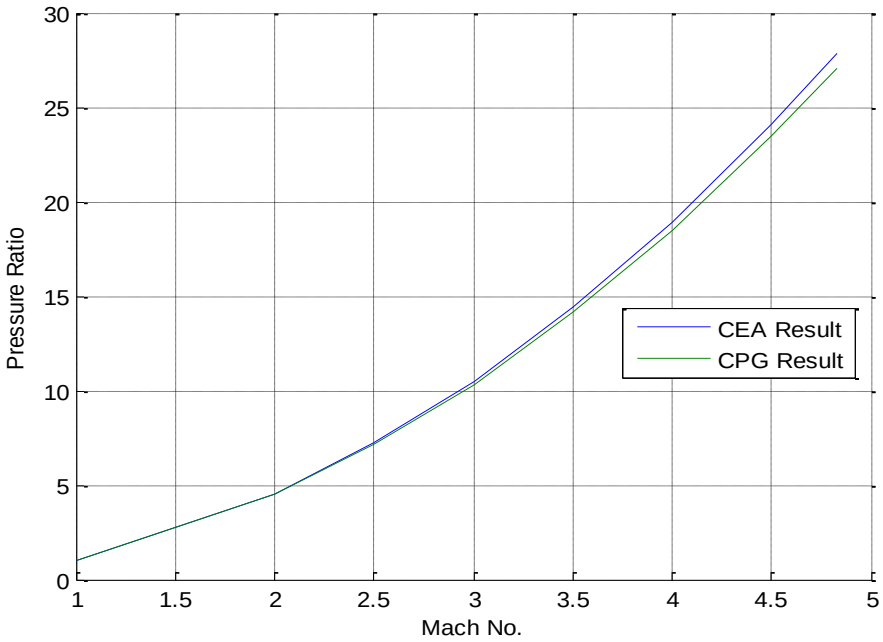

Figure 2.4 – Pressure Ratio from CEA, CPG.

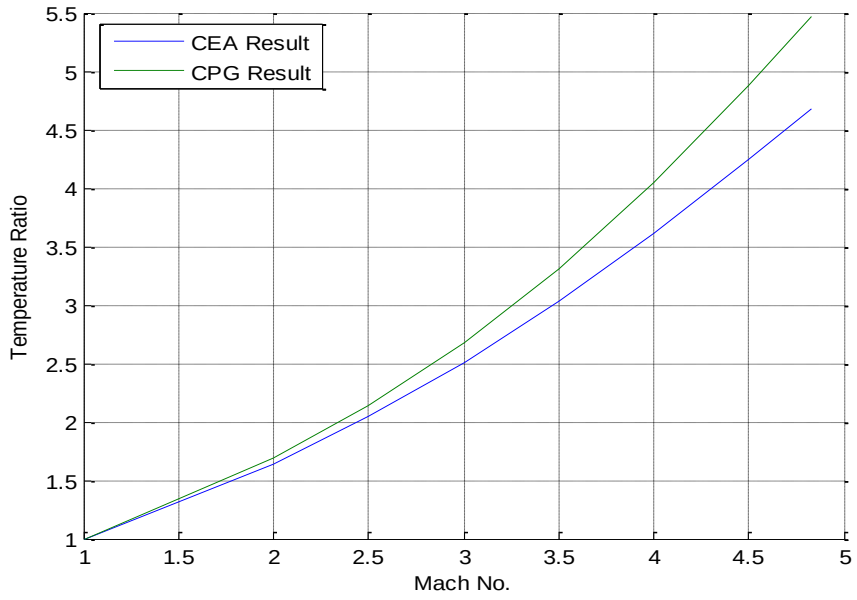

Figure 2.5 – Temperature Ratio from CEA, CPG.

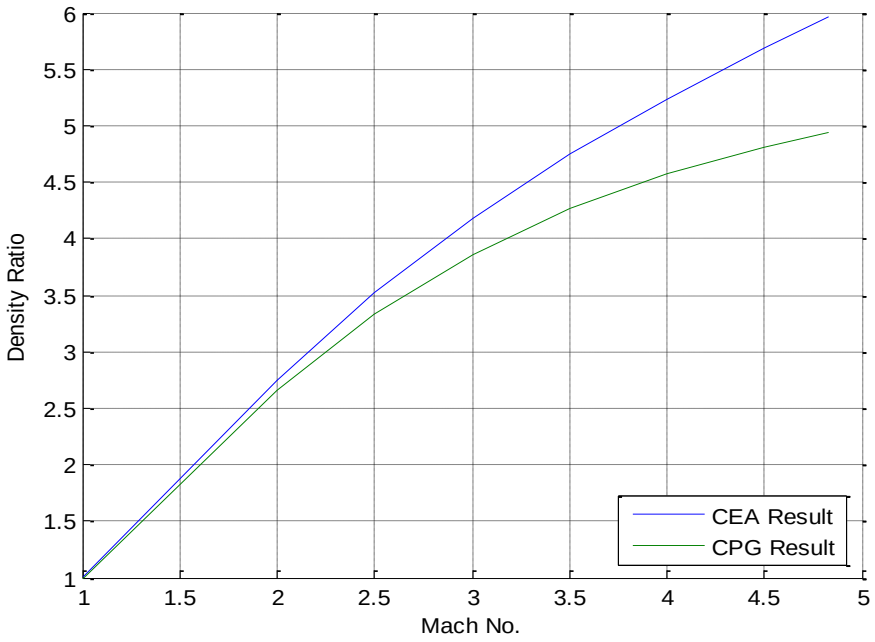

Figure 2.6 – Density Ratio from CEA and CPG.

2.1.3 Performance Model

The theoretical model developed by Endo, Fujiwara, et. al. is one of the most widely used and accepted models of the gas dynamics inside a pulse detonation engine (Refs. 6 & 7). They treat the PDE as a straight tube with fixed cross section in which one end of the tube is closed and the other end is open. At time t = 0, a detonation wave is initiated at the closed end (position x = 0) and begins propagating toward the open end of the tube. Since the detonation wave is a compression wave, the gas velocity behind the compression wave must be non-zero. However, since the gas velocity at the closed end must be at rest, the gas is decelerated through a self-similar rarefaction wave (known as the Taylor Rarefaction wave) following the detonation wave which also propagates toward the open end. The time at which the

detonation wave reaches the open end is called t_{CJ} and is the time at which another rarefaction wave starts to propagate from the open end toward the closed end. This second rarefaction wave reaches the closed end of the tube at time $t_{plateau}$, at which point it reflects once more toward the open end. Finally, once the rarefaction reaches the open end it will begin to reflect toward the closed end, but will be interrupted by the purging of the air remaining in the tube before being refilled and the process starting from the beginning.

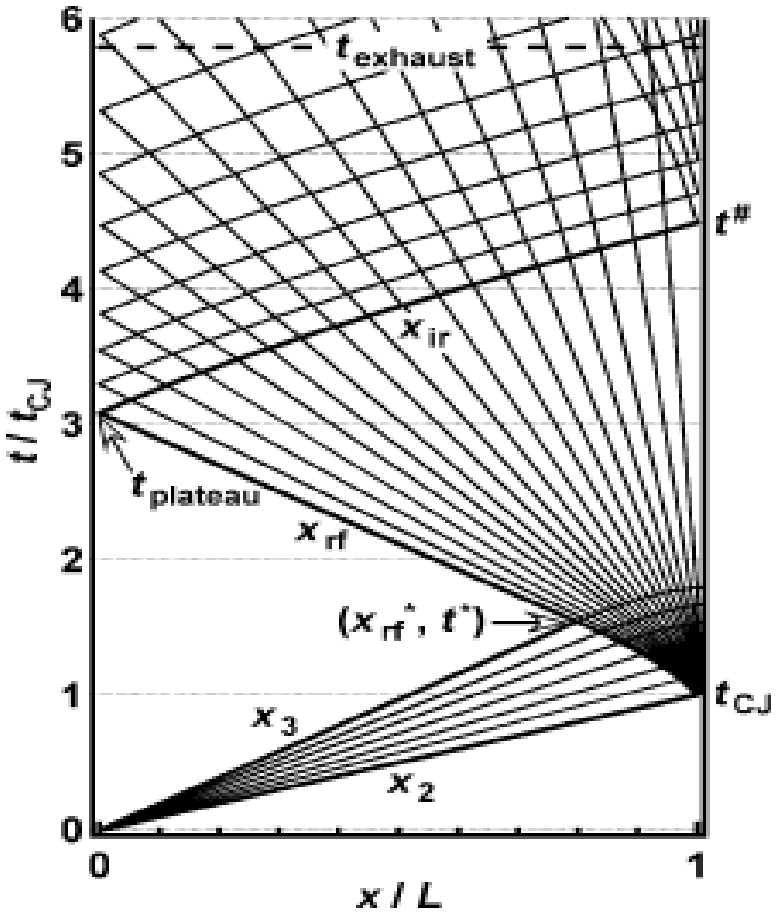

Figure 2.7 – The x-t diagram for the characteristics inside the detonation tube (Ref. 6).

The conditions in the initial “shock-rarefaction” region are given by:

$$P = \left(\frac{1}{\gamma_2} + \frac{\gamma_2 - 1}{\gamma_2}\frac{x}{x_2}\right)^{\frac{2\gamma_2}{(\gamma_2-1)}} P_2$$

$$\rho = \left(\frac{1}{\gamma_2} + \frac{\gamma_2 - 1}{\gamma_2}\frac{x}{x_2}\right)^{\frac{2}{(\gamma_2-1)}} \rho_2$$

$$T = \frac{P}{\rho R_2}$$

Where $P_2,\ \rho_2$ refer to the Von Neumann spike conditions. The conditions inside the rarefaction wave’s “first reflection” region:

$$P = \left(1 + \frac{\gamma_2 - 1}{D_{CJ}}\frac{L - x}{t - t_1}\right)^{\frac{2\gamma_2}{(\gamma_2-1)}} \frac{\gamma_1}{\gamma_2^{\frac{(\gamma_2+1)}{(\gamma_2-1)}}(\gamma_2 + 1)} M_{CJ}^2 P_1$$

$$\rho = \left(1 + \frac{\gamma_2 - 1}{D_{CJ}}\frac{L - x}{t - t_1}\right)^{\frac{2}{(\gamma_2-1)}} \left(\frac{\gamma_2 + 1}{\gamma_2^{\frac{(\gamma_2+1)}{(\gamma_2-1)}}}\right) \rho_1$$

However, these conditions do not affect the flow at the closed end and are usually referred to as the “plateau” conditions since they represent a reflected shock moving from the open end to the closed end (refer to Fig. 2.7, the open end is at position $\frac{x}{L} = 1$ from time $t^*\ to\ t^{\#}$). The final rarefaction wave which is referred to as the “blowdown phase” or decay of the pressure at the thrust wall is given in the Endo-Fujiwara model at the closed end of the tube, and had to be modified in order to give the correct results at the open end of the tube. In this project it is referred to as “second reflection” and whose conditions are given by:

$$P = 2.1348 \cdot P_{decay} \cdot P_3 \cdot (1 - 175(t - t_{\#}))$$

$$\rho = P_{decay} \cdot \rho_3$$

Where P_{decay} is given as a curve fit to experimental data in Endo, Fujiwara et. al. as:

$$P_{decay} = 0.6066e^{-\left(2.991\frac{a_3}{L}(t-t_{\#})\right)} + (1 - 0.6066)e^{-\left(0.5014\frac{a_3}{L}(t-t_{\#})\right)}$$

$$t_{\#} = \left(\delta_B + \delta_{A2}\left\{\left[\frac{2}{(\gamma_2 + 1)}\right]^{\frac{-(\gamma_2+1)}{2(\gamma_2-1)}} - 1\right\}\right)t_{CJ}$$

$$\delta_B = 2\left(\left[\frac{\gamma_1 M_{CJ}^2 + \gamma_2}{\gamma_1 M_{CJ}^2 + 1}\right]\left[\frac{\gamma_2 + 1}{2\gamma_2}\right]\right)^{\frac{-(\gamma+1)}{2(\gamma-1)}}$$

$$\delta_{A2} = 2\frac{\gamma_1 M_{CJ}^2}{\gamma_1 M_{CJ}^2 + \gamma_2}$$

2.2 Transient Analysis

Initially, this project was intended to analyze a standard turbofan with the traditional combustion chamber changed to a pulse detonation combustor. Doing this, however, caused several problems in NPSS which would not allow a solution to be found. One of the major problems in analyzing a pulse detonation combustor in a turbomachinery based engine is the purge-fill portion of the cycle during which essentially cold air (no chemical energy added through combustion) is blown through the turbine. A study cited by Andrus (Ref. 4) involved a series of computational and

experimental studies which examined how detonation waves interacted with a 2-dimensional cascade of rotor blades. The work showed that there is a significant fluctuation in mass flow rate, pressure and temperature across the rotor blades during the detonation cycle. One possible solution to such a problem is to fire the detonation tubes out of phase so that quasi-steady flow could be obtained at the turbine inlet. In NPSS this type of approach does not work since there cannot be any amount of time at which cool air is ejected from the combustor, since the turbine would be attempting to extract work out of air which has had no chemical energy added to it. Instead the transient analysis will be carried out in a "rig" setup in which the combustor is the only engine element present. The pressure and temperature over the cycle will be numerically integrated at each point using the trapezoidal method. After the cycle ends, the average pressure and temperature will be obtained using the Mean Value Theorem. This approach will allow a pulsed detonation combustor to be used in any type of engine setup in NPSS.

$$\int_{t_1}^{t_2} P(t)dt \approx \frac{P(t_1)+P(t_2)}{2}(t_2-t_1) \quad [Trapezoidal\ Method]$$

$$\int_{t_1}^{t_2} T(t)dt \approx \frac{T(t_1)+T(t_2)}{2}(t_2-t_1)$$

$$\overline{P(t)} = \frac{\int P(t)dt}{\Delta t} \quad [Mean\ Value\ Theorem]$$

$$\overline{T(t)} = \frac{\int T(t)dt}{\Delta t}$$

2.2.1 PDE Cycle

A pulse detonation engine cycle essentially contains seven stages. Figure 2.5 shows the major portions of the cycle in order.

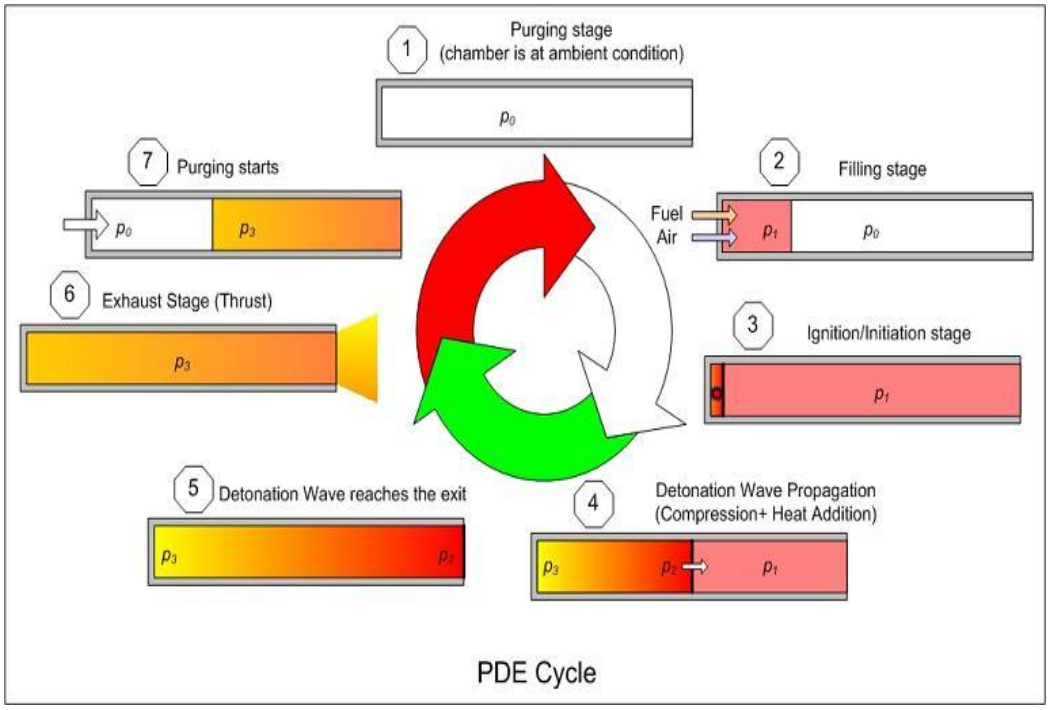

Figure 2.8 – Diagram containing the seven major stages of a PDE cycle.

(Image from Ref. 8)

Stage one shows the initial condition of the tube, containing only static air at the ambient conditions upstream of the tube. Stage two is the “fill” portion of the cycle during which upstream air is mixed with fuel and is assumed to be homogenous by the end of the fill portion of the cycle. Stage three is the ignition stage during which a deflagration wave is started at the closed end of the tube and transitions to a detonation wave at some point before reaching the open end of the tube during stage four. The time that the transition takes is known as the deflagration to detonation

transition time and can be changed in the combustor element file in NPSS. At stage five, the detonation wave, assumed to be at the Chapman-Jouguet condition, reaches the open end of the tube and begins exhausting into the downstream element of the engine. This is known as the exhaust phase during which the thrust of the detonation tube is generated. Finally, the “purge” portion is the part of the cycle when the valve opens the closed end of the tube and fresh air replaces the burned fuel-air mixture. Pressure and temperature in the tube return to ambient (or upstream) conditions during this portion of the cycle and velocity in the tube drops to zero.

2.2.2 NPSS Transient

In NPSS, running a transient analysis is essentially determining the behavior of the desired system at a series of discrete time steps over a certain interval. This allows for time varying inputs, component configurations (i.e. shaft inertia), initial conditions and various termination criteria. In this work, the “rig” setup will be run in the transient condition in order to determine the pressure, temperature and enthalpy profile for the detonation tubes over one full cycle. After the cycle is run, each value will be averaged for the given time period and substituted into the values for the steady-state burner output. In this case, the time is reported as a multiple of the CJ time, which is the time taken for the initial detonation wave to reach the open end of the tube. For this particular setup, the CJ time is 0.00222463 seconds, or 2.22463 ms (and an overall cycle time of 0.0757394 seconds).

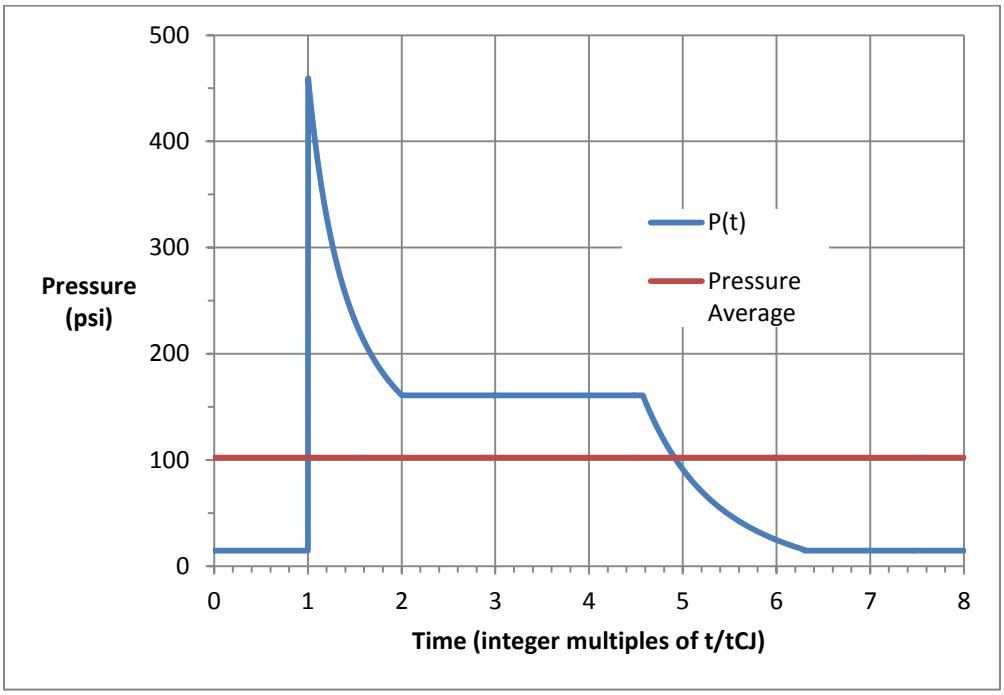

Figure 2.9 – Typical result of pressure as a function of time in a rig setup.

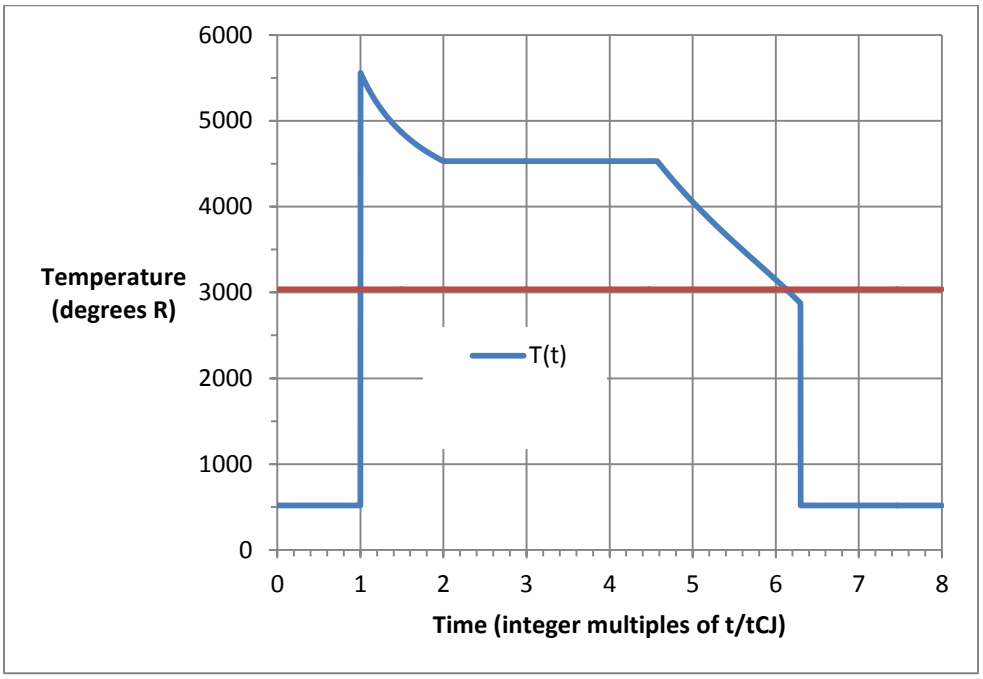

Figure 2.10 - Typical result of temperature as a function of time in a rig setup.

In order to obtain the average of these results, at each discrete time step a running average is updated according to the numerical integration process of the trapezoidal rule. At each time step, n:

$$\bar{P}_{n+1} = \bar{P}_n + \frac{(P_{n+1} - P_n)(t_{n+1} - t_n)}{2}$$

$$\bar{T}_{n+1} = \bar{T}_n + \frac{(T_{n+1} - T_n)(t_{n+1} - t_n)}{2}$$

Where $\bar{P}_n$ and $\bar{T}_n$ refer to the running averages, and the n subscript refers to the current time (n is the previous step, n+1 is the current step). After the final time step, these values are then divided by the time interval in order to find their average values over the cycle. A typical profile of a running pressure average is shown in Figure 2.11.

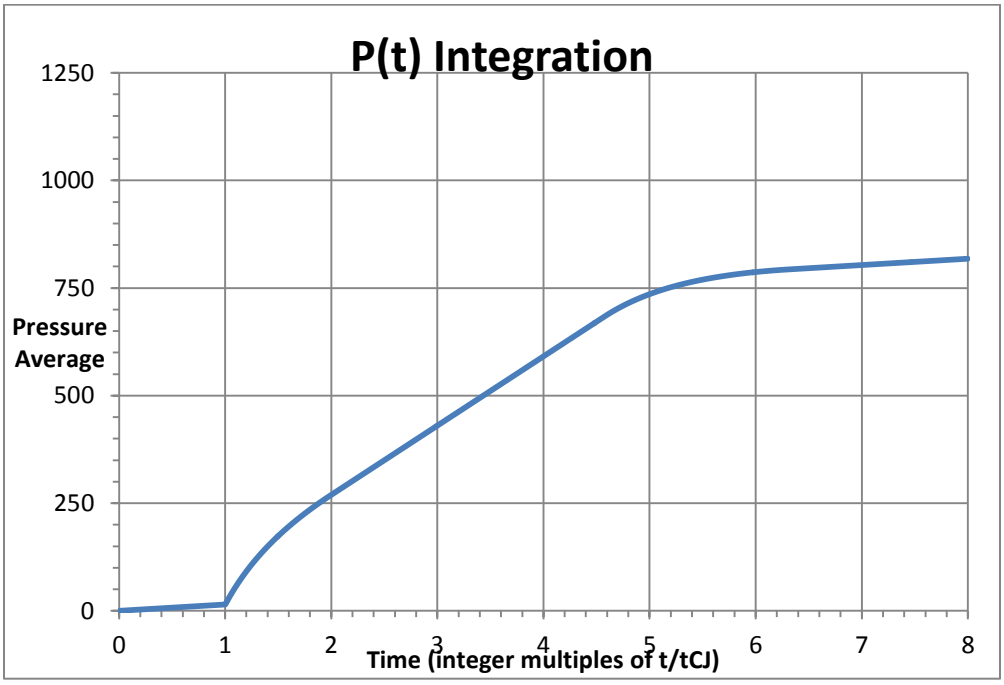

Figure 2.11 – Typical pressure integration running average.

The final value of the pressure integration is 818.0731354, which when divided by the time interval (in terms of multiples of t_{CJ}, the time interval is 8) gives the average pressure over the cycle of 102.271 psi. The same process is used to find the average temperature with a final value of 24289.474 and an average of 3036.532 °R. Since the pressure and temperature as a function of time are straight lines connecting two discrete values, it can be considered a polynomial of degree 1. As such, the trapezoidal rule gives an exact result for the integration (since the area bounded by a straight line is a trapezoid).

2.3.1 Turbine Inlet Temperature

In an engine operating on the Brayton cycle such as a gas-turbine, one extremely important design parameter is the turbine inlet temperature (also called the turbine entry temperature).

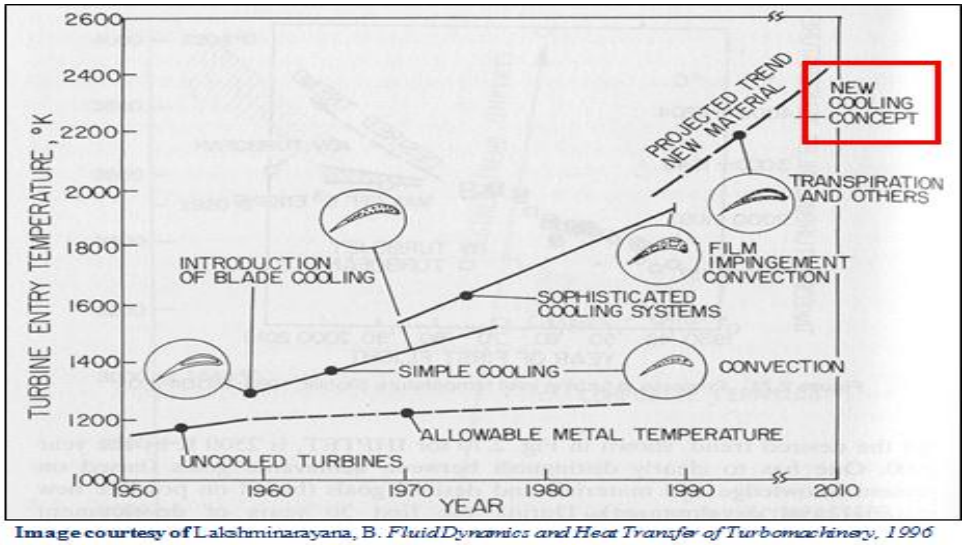

Image courtesy of Lakshminarayana, B. *Fluid Dynamics and Heat Transfer of Turbomachinery, 1996*

Figure 2.12 – A 1996 Projection of Turbine Inlet Temperatures.

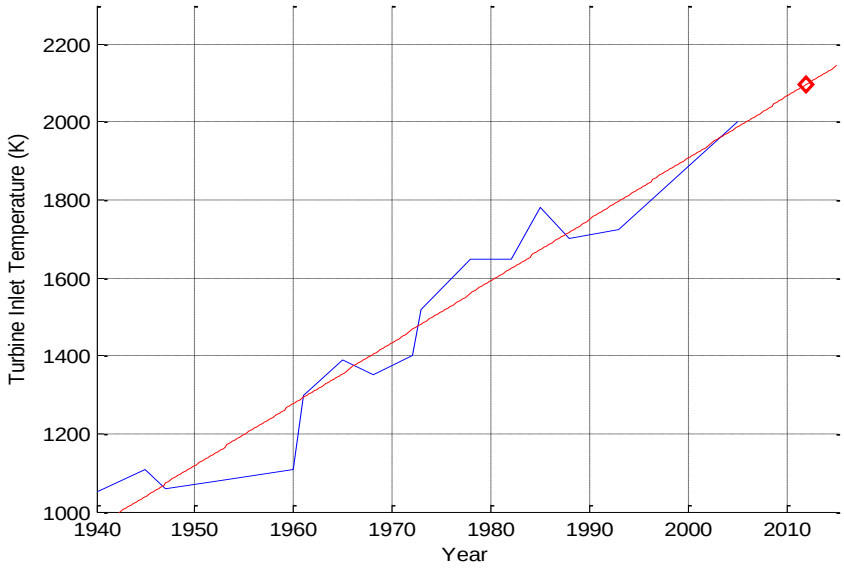

Figure 2.13 – Projection of Turbine Inlet Temperatures Using Actual Engine Data.

Initially when gas-turbine engines were first produced (roughly 1940), the turbine blades were completely uncooled and couldn't sustain high temperatures for very long. With the introduction of turbine cooling in the 1960's, the turbine inlet temperature began to rise. Since then, advances in materials, thermal coatings and more sophisticated cooling methods have allowed for an increasing amount of heat to reach the turbine blades. Using a linear projection of the turbine inlet temperatures of several commercial and military engines produced by Rolls-Royce from 1940 until 1993, the predicted turbine inlet temperature in 2012 is approximately 2100 K (3780 °R). However, it is important to note that the linear projection uses actual data, and should represent a very conservative value for the turbine inlet temperature. As of 1993, the maximum turbine inlet temperature that could be realistically cooled was

higher than 2200 K (3960 °R), however, and would be more than 2400 K (4320 °R) by the year 2012. As such, the maximum turbine inlet temperature in this project (4238.44 °R) should be within the cooling capabilities of a modern gas-turbine engine.

2.3.2 NPSS Design Process

Images property of Wolverine Ventures, Inc. ©Copyright, 2011. (Ref. 11)

Figure 2.14 – NPSS Design Step One

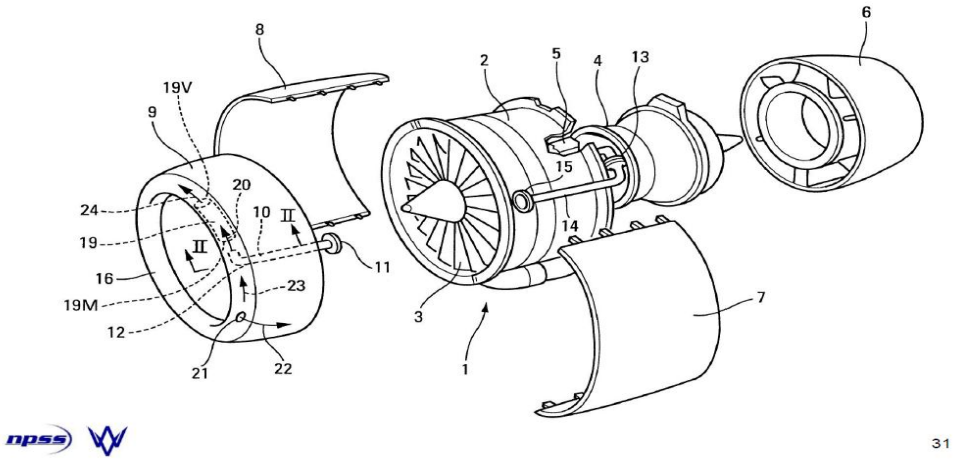

Figure 2.15 – NPSS Design Step Two

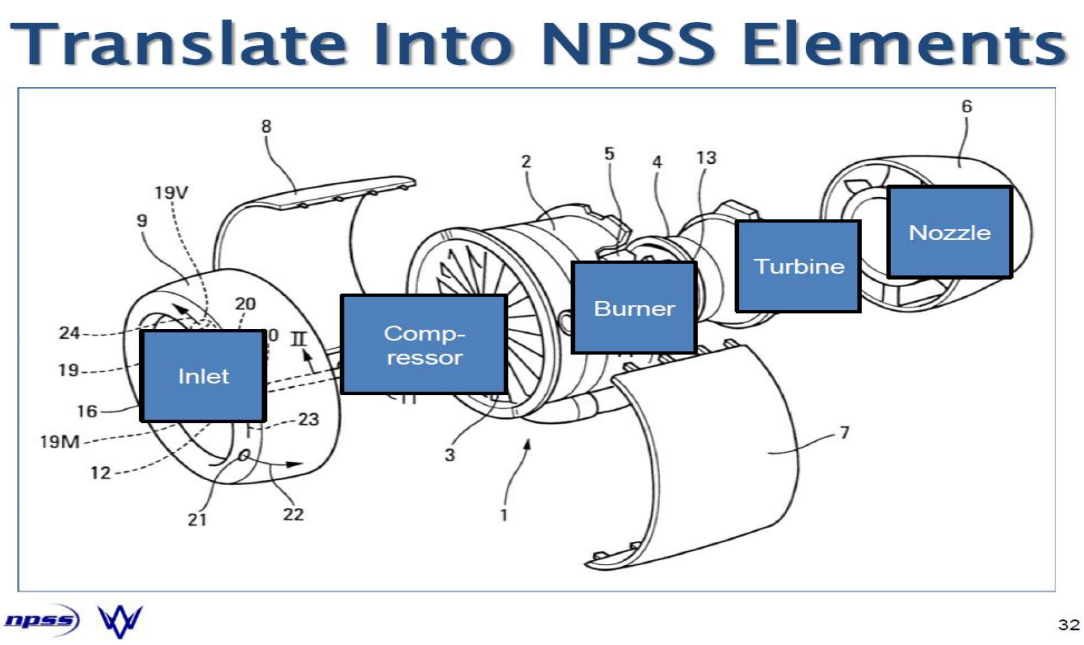

Figure 2.16 – NPSS Design Step Three

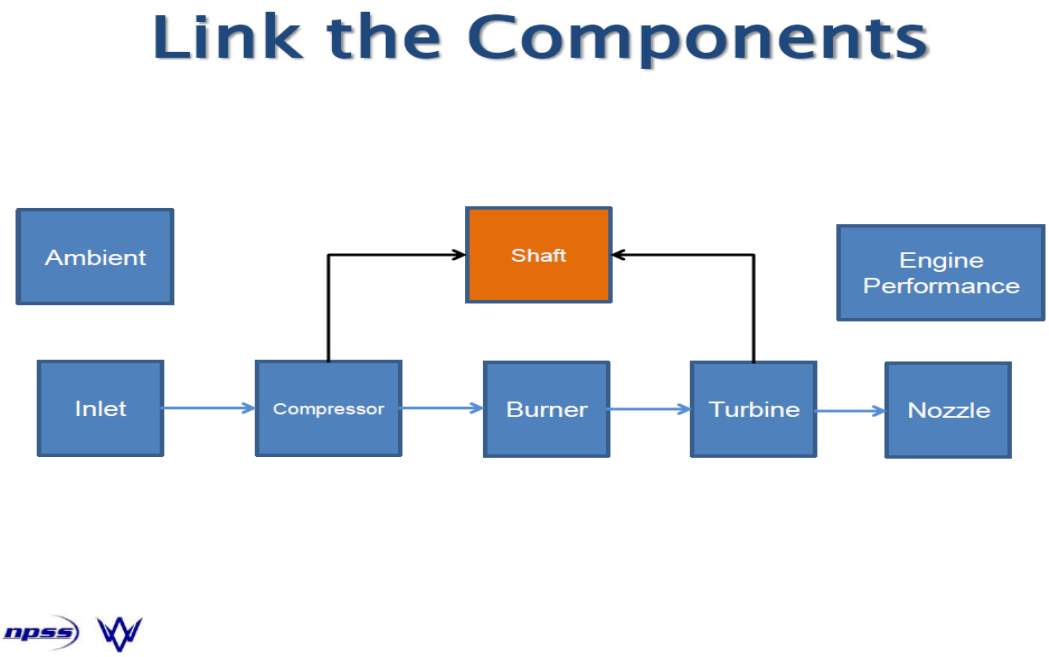

Figure 2.17 – NPSS Design Step Four

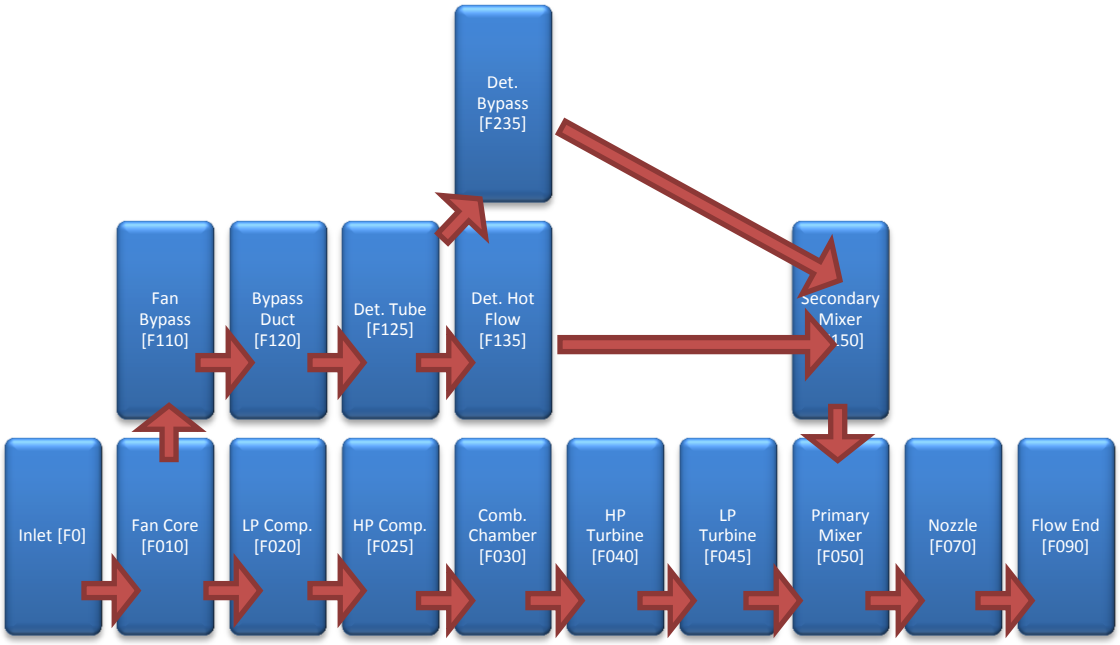

Figure 2.18 – Flow Station Diagram (PDC Setup)

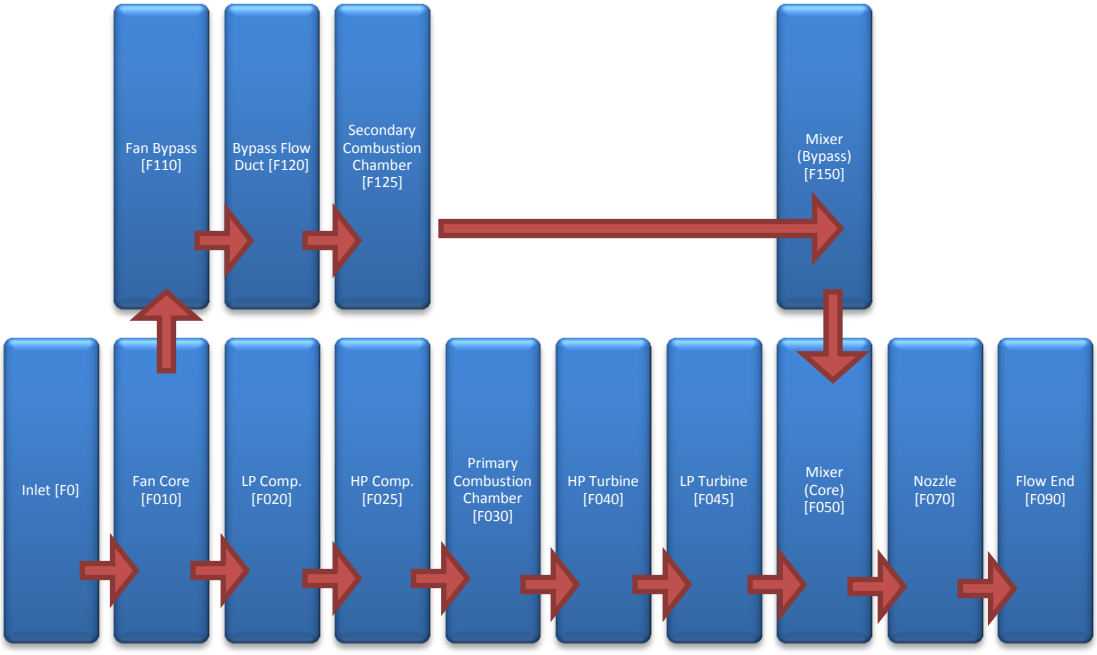

Figure 2.19 – Flow Station Diagram (Standard Combustor Setup)

Chapter 3

PDC and Baseline Performance Data

Table 3.1 – PDC entrance and exit conditions at altitude = 0 kft, M = 0.0

	Total Temperature (°R)	**Total Pressure (psia)**	**Total Enthalpy (BTU/lbm)**
Upstream Conditions	730.5	44.09	174.9
Cycle Average	2537.8	190.1	748.7
Det. Tube Max.	5181.9	847.7	1609.3

Table 3.2 – PDC entrance and exit conditions at altitude = 25 kft, M = 0.8

	Total Temperature (°R)	**Total Pressure (psia)**	**Total Enthalpy (BTU/lbm)**
Upstream Conditions	683.0	24.95	163.5
Cycle Average	2691.6	114.8	801.9
Det. Tube Max.	5475.6	511.1	1712.5

Table 3.3 – PDC entrance and exit conditions at altitude = 50 kft, M = 2.3

	Total Temperature (°R)	**Total Pressure (psia)**	**Total Enthalpy (BTU/lbm)**
Upstream Conditions	1118.8	63.2	270.7
Cycle Average	1854.9	171.1	510.4
Det. Tube Max.	3490.5	790.7	1026.3

Table 3.4 – PDE at altitude = 0 kft, M = 0.0

Flow Station	Total Temperature (°R)	Total Pressure (psia)	Mass Flow (lbm/s)
F0	518.67	14.6959	300
F010	518.67	14.6225	300
F110	518.67	14.6225	245.455
F020	518.67	14.6225	54.5455
F120	730.5	43.8674	245.455
F025	735.28	43.8674	54.5455
F125	730.5	43.2093	245.455
F030	1270.18	263.204	54.5455
F135	2397.27	186.31	11.7074
F235	730.5	43.2093	236.208
F040	3709.13	250.044	57.0455
F045	3317.1	138.317	57.0455
F050	2688.49	46.1055	57.0455
F150	823.22	61.8238	247.916
F070	1221.68	51.4168	304.961
F090	1221.68	51.4168	304.961

Table 3.5 – PDE at altitude = 25 kft, M = 0.8

Flow Station	Total Temperature (°R)	Total Pressure (psia)	Mass Flow (lbm/s)
F0	484.68	8.31607	300
F010	484.68	8.27449	300
F110	484.68	8.27449	245.455
F020	484.68	8.27449	54.5455
F120	683.00	24.8235	245.455
F025	687.49	24.8235	54.5455
F125	683.00	24.4511	245.455
F030	1191.96	148.941	54.5455
F135	2534.20	112.513	6.85979
F235	683.00	24.4511	240.037
F040	3650.55	141.494	57.0455
F045	3282.83	80.6931	57.0455
F050	2659.88	26.8977	57.0455
F150	744.22	19.4788	246.897
F070	1156.26	17.885	303.942
F090	1156.26	17.885	303.942

Table 3.6 – PDE at altitude = 50 kft, M = 2.3

Flow Station	Total Temperature (°R)	Total Pressure (psia)	Mass Flow (lbm/s)
F0	801.68	21.0581	300
F010	801.68	20.9528	300
F110	801.68	20.9528	245.455
F020	801.68	20.9528	54.5455
F120	1118.82	62.8584	245.455
F025	1125.89	62.8584	54.5455
F125	1118.82	61.9155	245.455
F030	1887.12	377.15	54.5455
F135	1787.8	167.7	13.6686
F235	1118.82	61.9155	234.659
F040	4186.57	358.293	57.0455
F045	3601.39	159.458	57.0455
F050	2925.09	53.1527	57.0455
F150	1160.03	61.537	248.328
F070	1531.97	57.0603	305.373
F090	1531.97	57.0603	305.373

Table 3.7 – Mixed Turbofan at altitude = 0 kft, M = 0.0

Flow Station	Total Temperature (°R)	Total Pressure (psia)	Mass Flow (lbm/s)
F0	518.67	14.6959	300
F010	518.67	14.6225	300
F110	518.67	14.6225	245.455
F020	518.67	14.6225	54.5455
F120	765.61	51.1786	245.455
F025	735.28	43.8674	54.5455
F125	765.61	50.4109	245.455
F030	1270.18	263.204	54.5455
F040	3761.64	250.044	57.0455
F045	3370.28	139.543	57.0455
F050	2403.42	23.7443	57.0455
F150	1332.37	47.8904	247.455
F070	1551.89	30.2027	304.5
F090	1551.89	30.2027	304.5

Table 3.8 – Mixed Turbofan at altitude = 25 kft, M = 0.8

Flow Station	Total Temperature (°R)	Total Pressure (psia)	Mass Flow (lbm/s)
F0	484.69	8.31607	300
F010	484.68	8.27449	300
F110	484.68	8.27449	245.455
F020	484.68	8.27449	54.5455
F120	715.96	28.9607	245.455
F025	687.50	24.8235	54.5455
F125	715.96	28.5263	245.455
F030	1191.96	148.941	54.5455
F040	3703.16	141.494	57.0455
F045	3336.09	81.3816	57.0455
F050	2432.39	15.5926	57.0455
F150	1286.71	27.1	247.455
F070	1522.33	19.1215	304.5
F090	1522.33	19.1215	304.5

Table 3.9 – Mixed Turbofan at altitude = 50 kft, M = 2.3

Flow Station	Total Temperature (°R)	Total Pressure (psia)	Mass Flow (lbm/s)
F0	801.69	21.0581	300
F010	801.68	20.9528	300
F110	801.68	20.9528	245.455
F020	801.68	20.9528	54.5455
F120	1170.41	73.3348	245.455
F025	1125.89	62.8584	54.5455
F125	1170.41	72.2348	245.455
F030	1887.12	377.15	54.5455
F040	4238.44	358.293	57.0455
F045	3654.14	161.205	57.0455
F050	2165.45	10.0521	57.0455
F150	1703.76	68.623	247.455
F070	1796.01	15.9855	304.5
F090	1796.01	15.9855	304.5

Table 3.10 – Mixed Turbofan with PDC Performance

	Gross Thrust (lbf)	Net Thrust (lbf)	TSFC (lbm/(lbf*hour))	Primary Fuel Flow (lbm/hour)	Duct Fuel Flow (lbm/hour)
Alt. = 0 kft M = 0.0	19770.0	19770.0	0.600842	9000	2878.62
Alt = 25 kft M = 0.8	18787.6	11205.6	0.953695	9000	1686.69
Alt = 50 kft M = 2.3	28057.2	7287.33	1.69621	9000	3360.82

Table 3.11 – Baseline Mixed Turbofan Performance

	Gross Thrust (lbf)	Net Thrust (lbf)	TSFC (lbm/(lbf*hour))	Primary Fuel Flow (lbm/hour)	Duct Fuel Flow (lbm/hour)
Alt. = 0 kft M = 0.0	17716.7	17716.7	0.812792	5400	9000
Alt = 25 kft M = 0.8	22105.8	14523.8	0.991477	5400	9000
Alt = 50 kft M = 2.3	28497.6	7727.72	1.86342	5400	9000

From Tables 3.10 and 3.11, it may seem strange that both the primary fuel flow and duct fuel flow vary significantly between the PDC turbofan and the baseline turbofan. After I had completed the PDC turbofan performance, I wanted to compare it to a baseline turbofan which offered similar performance in terms of thrust and thrust specific fuel consumption. This led to essentially a "guess-and-check" with the fuel flows of the baseline model until the performance was satisfactory. Though the performance is similar, the net fuel flow of the baseline turbofan was *much* higher than that of the PDC turbofan (greater than 1000 lbm/hour).

Table 3.12 – Detonation properties for various fan pressure ratios.

Fan PR	P_{burned} (psia)	T_{burned} (°R)	H_{burned} (BTU/lbm)
1.0	162.18	5135.976	579.647
1.5	244.56	5178.024	582.92
2.0	327.22	5296.41	585.099
2.5	410.05	5227.56	586.71
3.0	493.04	5244.264	587.96
5.0	825.93	5288.256	591.225

For various fan pressure ratios, several detonation parameters were determined using CEA (i.e. they represent the Chapman-Jouguet properties). Although the initial pressure changes significantly, both the temperature and enthalpy values changed very little. As such, a fan pressure ratio of 3.0 was used in this project in order to represent an average turbofan engine without taking too much power out of the turbine (although modern turbofans may be able to provide much more compression).

Table 3.13 – PDC properties initialized in NPSS.

PDC Properties	
Burning Efficiency	0.995
Fuel-Air Ratio	0.0683
Equivalence Ratio	0.85
Purge Fraction	0.2
Fill Fraction	0.8
Tube Length (inches)	36
Number of Tubes	24
Tube Diameter (inches)	2

The values chosen for the PDC in Table 3.12 were the same as those used in the Andrus (Ref. 4) and Thorn (Ref. 5) papers, and represent average values expected for a pulsed detonation combustor in the real world. These values were not altered for any of the test cases and no optimization was attempted in this project, although customization of these parameters would alter the results.

Table 3.14 – CJ conditions for each design point.

Chapman-Jouguet Conditions	**0 kft, M = 0.0**	**25 kft, M = 0.8**	**50 kft, M = 2.3**
M/M_1	4.0949	4.2172	3.3140
P/P_1	11.183	11.868	7.422
T/T_1	7.719	7.579	4.836
U_{CJ} (ft/s)	6231.63	6208.99	6203.74
ρ/ρ_1	1.7704	1.7775	1.7364
γ_{CJ}	1.1821	1.1779	1.1750
R_{CJ} [(ft*lbf)/(lbm*°R)]	53.3507	53.3507	53.3507

Chapter 4

Results

Initial results for the mixed flow turbofan engine with a pulsed detonation combustor as a duct burner look extremely promising. The fuel efficiency in the PDC model was an improvement over the baseline model in each case. In a real engine setting, the pulse detonation combustor must be well designed in order to offer an improvement over the baseline case since the losses in a real detonation combustor could begin to outweigh the advantages that detonation offers. In NPSS, in order to differentiate between a "design-point" run and an "off-design" run various sub-elements become important (i.e. compressor element, compressor performance map sub-element). In each of these three cases, the PDC has no mechanism to provide for an off-design consideration other than the variation in Chapman-Jouguet conditions based on the inlet flow conditions. However, the core elements of the turbofan do correct for off-design since the performance of compressors and turbines have been documented extremely well and a performance map is widely available (NPSS includes performance maps for all of its turbomachinery elements). As detonation tubes become more widely used, a performance map will likely be developed, at which point an off-design consideration would be much more feasible and accurate.

4.1 Take-Off

Table 4.1 – Performance of Baseline and PDC Model in the Takeoff Condition

	Gross Thrust (lbf)	**Net Thrust (lbf)**	**TSFC (lbm/(lbf*hour))**	**Primary Fuel Flow (lbm/hour)**	**Duct Fuel Flow (lbm/hour)**
Baseline Model	17716.7	17716.7	0.812792	5400	9000
PDC Model	19770.0	19770.0	0.600842	9000	2878.62

In the take-off case, the thrust specific fuel consumption of the pulse detonation model was 73.92% that of the baseline case with the total fuel flow reduced from 14,400 pounds per hour to 11,878 pounds per hour with a gain in net thrust from 17,717 pounds in the baseline case to 19,770 pounds in the PDC case. This case offers the most drastic improvement over the baseline case since the efficiency of the detonations is highest when the initial pressure is the smallest (i.e. when the ram pressure increase due to the motion of the aircraft relative to the air is the smallest).

4.2 Transonic Cruise

Table 4.2 – Performance of Baseline and PDC Model in the Subsonic Cruise Condition

	Gross Thrust (lbf)	**Net Thrust (lbf)**	**TSFC (lbm/(lbf*hour))**	**Primary Fuel Flow (lbm/hour)**	**Duct Fuel Flow (lbm/hour)**
Baseline Model	22105.8	14523.8	0.991477	5400	9000
PDC Model	18787.6	11205.6	0.953695	9000	1686.69

In the transonic cruise case (freestream Mach number of 0.8 at an altitude of 25 kft), both engines offer similar performance. While the baseline mixed turbofan model has a higher net thrust than the PDC model, the fuel efficiency of the PDC is better, as in the previous case. The thrust specific fuel consumption of the PDC model in this case is improved by 3.96% and the overall fuel flow is reduced from 14,400 pounds per hour to 10,686 pounds per hour.

4.3 Supersonic Cruise

Table 4.3 – Performance of Baseline and PDC Model in the Supersonic Cruise Condition

	Gross Thrust (lbf)	**Net Thrust (lbf)**	**TSFC (lbm/(lbf*hour))**	**Primary Fuel Flow (lbm/hour)**	**Duct Fuel Flow (lbm/hour)**
Baseline Model	28497.6	7727.72	1.86342	5400	9000
PDC Model	28057.2	7287.33	1.69621	9000	3360.82

Finally, in a supercruise condition (freestream Mach number of 2.3 at an altitude of 50 kft) the PDC model is again much more efficient than the baseline model. The thrust specific fuel consumption of the PDC model is 9.86% better than the baseline model with the overall fuel flow reduced from 14,400 pounds per hour to 12,360 pounds per hour.

4.4 Future Work Recommendations

Improvements to this work could include a more flexible computational model which can accurately handle a wider range of input conditions. Also, a mechanism for including the deflagration to detonation transition would most likely improve the accuracy of the results. Though at the time of this writing a full chemically reacting CFD simulation would be too expensive in terms of computational time, future works may not be limited in this manner and would offer the potential to fully exploit the advantages offered by a detonation combustor through optimization. Additional work could include extending the detonation model or developing a performance map for a PDC to account for off-design runs.

Chapter 5

Code Listing

Transient Model Definition – TransModel.mdl

```
//--------------------------------------------------------------------------

//                        Model Definition

//--------------------------------------------------------------------------



// Instantiate the Ambient element

// This element sets the ambient conditions

// and is refered to by the InletStart and Nozzle elements



Element Ambient AmbT {

        switchMode = "ALDTMN"; // Set ambient conditions based on altitude, Mach

number, and standard day delta T

        alt_in = 0.; // ft, input altitude

        MN_in = 0.0;// input, Mach number

        dTs_in = 0.; // Rankine, input temperature delta from standard day conditions

} // END Amb

Element InletStart FsEngT {

        AmbientName = "AmbT"; // Name of the Ambient element

        W_in = 257.143; // lbm/s, input air flow flow rate
```

```
} // End FsEng

Element Compressor CmpT {
        // Load file that instantiates a subelement and plugs into compressor socket
(S_map) and contains
        // compressor performance map
        #include "lpcE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set compressor design point values in S_map subelement
        S_map {
                PRdes = 3.0; // design point pressure ratio
                effDes = 0.9; // design point efficiency
        }
} // End Cmp

Element FuelStart FusEngT {
        Wfuel = 2.0; // lbm/s, fuel flow rate (Used ONLY when Burner switchBurn =
WFUEL)
        LHV = 18400; // BTU/lbm, Hydrogen - 61,000, Default - 18,400
} // End FusEng

// Instantiate the BrnPri element
Element PulseDetonationCombustor BrnPriT {
        effBase = .995; // burning efficiency
        dPqPBase = 0; //1.0-0.96; // pressure loss across valves/through bypass
        switchBurn = FAR; // set fuel-air ratio (vs equivalence ratio)
```

```
        //switchHotLoss = CALCULATE; // hot pressure loss will be calculated
        FAR = 0.0683; // 0.0683 is ~85% of stoichiometric conditions
        purgeFrac = 0.2; // designate purge fraction
        fillFrac = 0.8; // designate fill fraction
        lTube = 36; // length of tube in inches
        n_tubes = 24; // number of tubes
        dTube = 2.0; // inside diameter of tubes
        tCycle = .016776271641; // cycle time
}
Element Shaft ShT {
        //  Mechanical Ports.  These are created as needed on the shaft.
        ShaftInputPort Sh_ICmp; // create a shaft port for a mechanical connection
between the shaft and the compressor
        Nmech = 10000.; // rpm, shaft speed
} // End Sh

Element FlowEnd FePriT {
} // End combustion air flow

Element FlowEnd FeSecT {
} // End bypass air flow

//------------------------------------------------------------------------
//                      Component Linkages
//------------------------------------------------------------------------
```

```
// Link Fluid Ports
linkPorts( "FsEngT.Fl_O"        , "CmpT.Fl_I"        , "F0T"   );
linkPorts( "CmpT.Fl_O"          , "BrnPriT.Fl_I"     , "F030T" );
linkPorts( "BrnPriT.Fl_O1"      , "FePriT.Fl_I"      , "F090T" );
linkPorts( "BrnPriT.Fl_O2"      , "FeSecT.Fl_I"      , "F190T" );

//Link Fuel Ports
linkPorts( "FusEngT.Fu_O"       , "BrnPriT.Fu_I"     , "Fu_InT" );

// Link Shaft Ports
linkPorts( "CmpT.Sh_O"          ,"ShT.Sh_ICmp"       ,"MeCmp" );

// Set power applied to shaft (HPX is in horsepower)
// To match the power consumed by the compressor
ShT.HPX = -CmpT.pwr;
```

Mixed Turbofan Design Model – DesignModel.mdl

```
//-----------------------------------------------------------------------
//                          Model Definition
//-----------------------------------------------------------------------

// Instantiate the Ambient element
// This element sets the ambient conditions
// and is refered to by the InletStart and Nozzle elements
real MN;

Element Ambient Amb {
  switchMode = "ALDTMN"; // Set ambient conditions based on altitude, Mach number,
and standard day delta T
  alt_in = 0.; // ft, input altitude
  MN_in = 0.0;// input, Mach number
  MN = MN_in;
  dTs_in = 0.; // Rankine, input temperature delta from standard day conditions
} // END Amb
```

```
// Instantiate the InletStart element

Element InletStart FsEng {
  AmbientName = "Amb"; // Name of the Ambient element
  W_in = 300.; // lbm/s, input air flow flow rate
} // End InletStart

// Instantiate the Inlet element

Element Inlet InEng {
        if(MN > 1) {
                eRamBase = 1 - (0.075*(MN - 1)**1.35); // Ram pressure recovery
        }
        else {
                eRamBase = 0.995;
        }
} // END InEng


// Instantiate the bypass splitter
Element Splitter SpltFan {
        BPRdes = 4.5;
} // END SpltFan

// Instantiate the fan OD element
```

```
Element Compressor CmpFSec {
        // Load file that instantiates a subelement and plugs into compressor socket
(S_map) and contains
        // compressor performance map
        #include "fanE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set compressor design point values in S_map subelement
        S_map{
                PRdes = 3.0; // design point pressure ratio
                effDes = 0.9; // design point efficiency
        }
} // End CmpL
// Instantiate the fan duct for the bypass flow

Element Duct D130 {
        switchDP = "INPUT"; // allow the user to input the relative pressure drop (deltP /
Pin)
        dPqP_in = 0.015; // user-input relative pressure drop
} // END Dfan

Element FuelStart FusEngDuct {
        Wfuel = 2.0; // lbm/s, fuel flow rate (Used ONLY when Burner switchBurn =
WFUEL)
        LHV = 18400; // BTU/lbm, Hydrogen - 61,000, Default - 18,400
} // End FusEng
```

```
Element PulseDetonationCombustor BrnPri {
        effBase = .995; // burning efficiency
        dPqPBase = 0; //1.0-0.96; // pressure loss across valves/through bypass
        switchBurn = FAR; // set fuel-air ratio (vs equivalence ratio)
        //switchHotLoss = CALCULATE; // hot pressure loss will be calculated
        FAR = 0.0683; // 0.0683 is ~85% of stoichiometric conditions
        purgeFrac = 0.2; // designate purge fraction
        fillFrac = 0.8; // designate fill fraction
        lTube = 36; // length of tube in inches
        n_tubes = 24; // number of tubes
        dTube = 2.0; // inside diameter of tubes
        tCycle = .016776271641; // cycle time
}

// Instantiate the Compressor element

Element Compressor CmpL {
        // Load file that instantiates a subelement and plugs into compressor socket
(S_map) and contains
        // compressor performance map
        #include "lpcE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set compressor design point values in S_map subelement
        S_map{
                PRdes = 3.0; // design point pressure ratio
                effDes = 0.88; // design point efficiency
```

```
        }
} // End CmpL

Element Compressor CmpH {
        // Load file that instantiates a subelement and plugs into compressor socket
(S_map) and contains
        // compressor performance map
        #include "hpcE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set compressor design point values in S_map subelement
        S_map{
                PRdes = 6.0; // design point pressure ratio
                effDes = 0.88; // design point efficiency
        }
} // End CmpH

// Start the flow of fuel

Element FuelStart FusEng {
        Wfuel = 2.0; // lbm/s, fuel flow rate (Used ONLY when Burner switchBurn =
WFUEL)
        LHV = 18000; // BTU/lbm, user input fuel lower heating value (LHV).  Default is
18400 BTU/lbm
} // End FusEng
// Instantiate the Burner element
```

```
Element Burner BrnSec {
        //dPqPfBase         = 0.05; // user input friction relative pressure drop (Pin -
Pout)/Pin
        dPqP_dmd          = 0.05; // user input friction relative pressure drop (Pin -
Pout)/Pin
        effBase           = 0.98; // user input burner adiabatic efficiency

        // The value for switchBurn determines how burner fuel flow rate is set
        // if switchBurn = FUEL, then use Wfuel that is set in the burner element
        // if switchBurn = WFUEL, then use fuel start element flow rate to set Wfuel
(Fu_I.Wfuel inherited from fuel start Fu_O.Wfuel)
        // if switchBurn = FAR, then use air inlet FAR value (Fl_I.FAR) to calculate Wfuel
        switchBurn = "FUEL"; // FUEL, WFUEL, or FAR
        Wfuel = 2.5; // lbm/s, user input fuel flow rate.  Used only whey switchBurn =
FUEL
} // End BrnPri

// Instantiate the High Pressure Turbine element

Element Turbine TrbH {
        // Load file that instantiates a subelement and plugs into turbine socket (S_map)
and contains
        // turbine performance map
        #include "hptE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set turbine design point values in S_map subelement
```

```
        S_map {
                effDes      = 0.9;  // user-specified Design efficiency
                parmMap     = 6.0;  // pressure ratio initial guess
        }
} // End TrbH

// Instantiate the High Pressure Turbine element

Element Turbine TrbL {
        // Load file that instantiates a subelement and plugs into turbine socket (S_map)
and contains
        // turbine performance map
        #include "lptE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set turbine design point values in S_map subelement
        S_map {
                effDes      = 0.9;  // user-specified Design efficiency
                parmMap     = 3.0;  // pressure ratio initial guess
        }
} // End TrbL

// Instantiate the Low Pressure Shaft element

Element Shaft ShL {
  //  Mechanical Ports.  These are created as needed on the shaft.
```

```
  ShaftInputPort Sh_ICmpFSec; // create a shaft port for a mechanical connection
between the shaft and the fan OD element
  ShaftInputPort Sh_ICmp; // create a shaft port for a mechanical connection between the
shaft and the low pressure compressor
  ShaftInputPort Sh_ITrb; // create a shaft port for a mechanical connection between the
shaft and the low pressure turbine
  Nmech = 5000.; // rpm, shaft speed
} // End ShL


// Instantiate the Low Pressure Shaft element
Element Shaft ShH {
  //  Mechanical Ports.  These are created as needed on the shaft.
  ShaftInputPort Sh_ICmp; // create a shaft port for a mechanical connection between the
shaft and the high pressure compressor
  ShaftInputPort Sh_ITrb; // create a shaft port for a mechanical connection between the
shaft and the high pressure turbine
  Nmech = 10000.; // rpm, shaft speed
} // End ShH

Element Mixer MixPri {
        Fl_I1.MN = 0.1;
        Fl_I2.MN = 0.1;
}

Element Mixer MixSec {
```

```
            //Fl_I1.MN = 0.15;
            //Fl_I2.MN = 0.15;
            Fl_I1.A = 256;
            Fl_I2.A = 256;
}

Element Nozzle NozPri {
  PsExhName = "Amb.Ps"; // Model variable for ambient static pressure
  switchType = "CONIC"; // conic nozzle
} //END NozPri

// End the flow of air for primary stream

Element FlowEnd FePri {
} // End FrPri

// Instantiate the EngPerf element
// This element makes some basic engine performance calculations

Element EngPerf Perf {
} //End Perf


//--------------------------------------------------------------------
//                        Component Linkages
//--------------------------------------------------------------------
```

```
// Link Fluid Ports

// Primary Hot Section
linkPorts( "FsEng.Fl_O"        , "InEng.Fl_I"      , "F0" );
linkPorts( "InEng.Fl_O"        , "SpltFan.Fl_I"    , "F010" );
linkPorts( "SpltFan.Fl_O1"     , "CmpL.Fl_I"       , "F020" );
linkPorts( "CmpL.Fl_O"                , "CmpH.Fl_I"       , "F025" );
linkPorts( "CmpH.Fl_O"                , "BrnSec.Fl_I"     , "F030" );
linkPorts( "BrnSec.Fl_O"       , "TrbH.Fl_I"       , "F040" );
linkPorts( "TrbH.Fl_O"         , "TrbL.Fl_I"       , "F045" );
linkPorts( "TrbL.Fl_O"                , "MixPri.Fl_I1"    , "F050" );


// Fan duct section
linkPorts( "SpltFan.Fl_O2",   "CmpFSec.Fl_I",      "F110"   );
linkPorts( "CmpFSec.Fl_O",    "D130.Fl_I",         "F120"   );
linkPorts( "D130.Fl_O",       "BrnPri.Fl_I",       "F125"   );
linkPorts( "BrnPri.Fl_O1",    "MixSec.Fl_I1",      "F135"   );
linkPorts( "BrnPri.Fl_O2",    "MixSec.Fl_I2",      "F235"   );
linkPorts( "MixSec.Fl_O",     "MixPri.Fl_I2",      "F150"   );

// Mix Primary/Duct flows
linkPorts( "MixPri.Fl_O"       , "NozPri.Fl_I"     , "F070" );
linkPorts( "NozPri.Fl_O"       , "FePri.Fl_I"      , "F090" );
```

```
// Link Fuel Ports
linkPorts( "FusEng.Fu_O"              , "BrnPri.Fu_I"     , "Fu_In"  );
linkPorts( "FusEngDuct.Fu_O"   , "BrnSec.Fu_I"     , "Fu_In2" );

// Link Shaft Ports
linkPorts( "CmpFSec.Sh_O"       ,"ShL.Sh_ICmpFSec"   ,"MeCmpFSec" );
linkPorts( "CmpL.Sh_O"          ,"ShL.Sh_ICmp"       ,"MeCmpL" );
linkPorts( "TrbL.Sh_O"          ,"ShL.Sh_ITrb"       ,"MeTrbL" );
linkPorts( "CmpH.Sh_O"          ,"ShH.Sh_ICmp"       ,"MeCmpH" );
linkPorts( "TrbH.Sh_O"          ,"ShH.Sh_ITrb"       ,"MeTrbH" );

//--------------------------------------------------------------------------
//                        Solver Sequence
//--------------------------------------------------------------------------
// If solver execution sequence is not set by the user (below), the default sequence will be
the order of the element instantiation above

        solver.executionSequence = {
                                "Amb",
                                "FsEng",
                                "InEng",
                                "SpltFan",
                                "CmpL",
                                "CmpH",
                                "CmpFSec",
                                "D130",
```

```
                    "FusEngDuct",
                    "BrnPri",
                    "MixSec",
                    "FusEng",
                    "BrnSec",
                    "TrbH",
                    "TrbL",
                    "MixPri",
                    "ShH",
                    "ShL",
                    "NozPri",
                    "FePri",
                    "Perf"
          }
```

Standard Turbofan Model Definition – MixedTFComparison.mdl

```
//-------------------------------------------------------------------------
//                      Model Definition
//-------------------------------------------------------------------------

// Instantiate the Ambient element
// This element sets the ambient conditions
// and is refered to by the InletStart and Nozzle elements
```

```
Element Ambient Amb {
  switchMode = "ALDTMN"; // Set ambient conditions based on altitude, Mach number,
and standard day delta T
  alt_in = 0.; // ft, input altitude
  MN_in = 0.0;// input, Mach number
  dTs_in = 0.; // Rankine, input temperature delta from standard day conditions
} // END Amb

// Instantiate the InletStart element

Element InletStart FsEng {
  AmbientName = "Amb"; // Name of the Ambient element
  W_in = 300.; // lbm/s, input air flow flow rate
} // End InletStart

// Instantiate the Inlet element

Element Inlet InEng {
  if(MN > 1) {
                eRamBase = 1 - (0.075*(MN - 1)**1.35); // Ram pressure recovery
        }
        else {
                eRamBase = 0.995;
        }
} // END InEng
```

```
// Instantiate the bypass splitter

Element Splitter SpltFan {
        BPRdes = 4.5;
} // END SpltFan

// Instantiate the fan OD element

Element Compressor CmpFSec {
        // Load file that instantiates a subelement and plugs into compressor socket
(S_map) and contains
        // compressor performance map
        #include "fanE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set compressor design point values in S_map subelement
        S_map{
                PRdes = 3.5; // design point pressure ratio
                effDes = 0.9; // design point efficiency
        }
} // End CmpL

// Instantiate the fan duct for the bypass flow
Element Duct D130 {
```

```
        switchDP = "INPUT"; // allow the user to input the relative pressure drop (deltP /
Pin)
        dPqP_in = 0.015; // user-input relative pressure drop
} // END Dfan

Element FuelStart FusEngDuct {
        Wfuel = 2.5; // lbm/s, fuel flow rate (Used ONLY when Burner switchBurn =
WFUEL)
        LHV = 18400; // BTU/lbm, Hydrogen - 61,000, Default - 18,400
} // End FusEng

Element Burner BrnPri {
        //dPqPfBase        = 0.05; // user input friction relative pressure drop (Pin -
Pout)/Pin
        dPqP_dmd        = 0.05; // user input friction relative pressure drop (Pin -
Pout)/Pin
        effBase          = 0.98; // user input burner adiabatic efficiency

        // The value for switchBurn determines how burner fuel flow rate is set
        // if switchBurn = FUEL, then use Wfuel that is set in the burner element
        // if switchBurn = WFUEL, then use fuel start element flow rate to set Wfuel
(Fu_I.Wfuel inherited from fuel start Fu_O.Wfuel)
        // if switchBurn = FAR, then use air inlet FAR value (Fl_I.FAR) to calculate Wfuel
        switchBurn = "FUEL"; // FUEL, WFUEL, or FAR
        Wfuel = 2.0; // lbm/s, user input fuel flow rate.  Used only whey switchBurn =
FUEL
```

```
}

// Instantiate the Compressor element
Element Compressor CmpL {
        // Load file that instantiates a subelement and plugs into compressor socket
(S_map) and contains
        // compressor performance map
        #include "lpcE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set compressor design point values in S_map subelement
        S_map{
                PRdes = 3.0; // design point pressure ratio
                effDes = 0.88; // design point efficiency
        }
} // End CmpL

Element Compressor CmpH {
        // Load file that instantiates a subelement and plugs into compressor socket
(S_map) and contains
        // compressor performance map
        #include "hpcE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set compressor design point values in S_map subelement
        S_map{
                PRdes = 6.0; // design point pressure ratio
                effDes = 0.88; // design point efficiency
```

```
        }
} // End CmpH

// Start the flow of fuel
Element FuelStart FusEng {
        Wfuel = 1.5; // lbm/s, fuel flow rate (Used ONLY when Burner switchBurn =
WFUEL)
        //LHV = 18400; // BTU/lbm, user input fuel lower heating value (LHV).  Default is
18400 BTU/lbm
} // End FusEng


// Instantiate the Burner element
Element Burner BrnSec {
        //dPqPfBase        = 0.05; // user input friction relative pressure drop (Pin -
Pout)/Pin
        dPqP_dmd         = 0.05; // user input friction relative pressure drop (Pin -
Pout)/Pin
        effBase          = 0.98; // user input burner adiabatic efficiency

        // The value for switchBurn determines how burner fuel flow rate is set
        // if switchBurn = FUEL, then use Wfuel that is set in the burner element
        // if switchBurn = WFUEL, then use fuel start element flow rate to set Wfuel
(Fu_I.Wfuel inherited from fuel start Fu_O.Wfuel)
        // if switchBurn = FAR, then use air inlet FAR value (Fl_I.FAR) to calculate Wfuel
        switchBurn = "WFUEL"; // FUEL, WFUEL, or FAR
```

```
        // Wfuel = 0.2; // lbm/s, user input fuel flow rate.  Used only whey switchBurn =
FUEL
}

// Instantiate the High Pressure Turbine element
Element Turbine TrbH {
        // Load file that instantiates a subelement and plugs into turbine socket (S_map)
and contains
        // turbine performance map
        #include "hptE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set turbine design point values in S_map subelement
        S_map {
                effDes       = 0.9;  // user-specified Design efficiency
                parmMap      = 5.0;  // pressure ratio initial guess
        }
} // End TrbH

// Instantiate the High Pressure Turbine element
Element Turbine TrbL {
        // Load file that instantiates a subelement and plugs into turbine socket (S_map)
and contains
        // turbine performance map
        #include "lptE3.map"; // when scoping, refer to the subelement by its socket
name: S_map
        // Set turbine design point values in S_map subelement
```

```
        S_map {
                effDes      = 0.9;  // user-specified Design efficiency
                parmMap     = 2.0;  // pressure ratio initial guess
        }
} // End TrbL

// Instantiate the Low Pressure Shaft element
Element Shaft ShL {
  //  Mechanical Ports.  These are created as needed on the shaft.
  ShaftInputPort Sh_ICmpFSec; // create a shaft port for a mechanical connection
between the shaft and the fan OD element
  ShaftInputPort Sh_ICmp; // create a shaft port for a mechanical connection between the
shaft and the low pressure compressor
  ShaftInputPort Sh_ITrb; // create a shaft port for a mechanical connection between the
shaft and the low pressure turbine
  Nmech = 5000.; // rpm, shaft speed
} // End ShL

// Instantiate the Low Pressure Shaft element
Element Shaft ShH {
  //  Mechanical Ports.  These are created as needed on the shaft.
  ShaftInputPort Sh_ICmp; // create a shaft port for a mechanical connection between the
shaft and the high pressure compressor
  ShaftInputPort Sh_ITrb; // create a shaft port for a mechanical connection between the
shaft and the high pressure turbine
  Nmech = 10000.; // rpm, shaft speed
```

```
} // End ShH

Element Mixer MixPri {
        Fl_I1.MN = 0.1;
        Fl_I2.MN = 0.1;
}

Element Nozzle NozPri {
  PsExhName = "Amb.Ps"; // Model variable for ambient static pressure
  switchType = "CONIC"; // conic nozzle
} //END NozPri

// End the flow of air for primary stream

Element FlowEnd FePri {
} // End FrPri

// Instantiate the EngPerf element
// This element makes some basic engine performance calculations

Element EngPerf Perf {
} //End Perf

//--------------------------------------------------------------------------
//                      Component Linkages
```

```
//--------------------------------------------------------------------------

// Link Fluid Ports

// Primary Hot Section
linkPorts( "FsEng.Fl_O"        , "InEng.Fl_I"      , "F0" );
linkPorts( "InEng.Fl_O"        , "SpltFan.Fl_I"    , "F010" );
linkPorts( "SpltFan.Fl_O1"     , "CmpL.Fl_I"       , "F020" );
linkPorts( "CmpL.Fl_O"                , "CmpH.Fl_I"       , "F025" );
linkPorts( "CmpH.Fl_O"                , "BrnSec.Fl_I"     , "F030" );
linkPorts( "BrnSec.Fl_O"       , "TrbH.Fl_I"       , "F040" );
linkPorts( "TrbH.Fl_O"         , "TrbL.Fl_I"       , "F045" );
linkPorts( "TrbL.Fl_O"                , "MixPri.Fl_I1"    , "F050" );

// Fan duct section
linkPorts( "SpltFan.Fl_O2",   "CmpFSec.Fl_I",      "F110"   );
linkPorts( "CmpFSec.Fl_O",    "D130.Fl_I",         "F120"   );
linkPorts( "D130.Fl_O",       "BrnPri.Fl_I",       "F125"   );
linkPorts( "BrnPri.Fl_O",     "MixPri.Fl_I2",      "F150"   );

// Mix Primary/Duct flows
linkPorts( "MixPri.Fl_O"       , "NozPri.Fl_I"     , "F070" );
linkPorts( "NozPri.Fl_O"       , "FePri.Fl_I"      , "F090" );

// Link Fuel Ports
linkPorts( "FusEng.Fu_O"              , "BrnPri.Fu_I"     , "Fu_In"  );
```

```
linkPorts( "FusEngDuct.Fu_O"    , "BrnSec.Fu_I"     , "Fu_In2" );


// Link Shaft Ports
linkPorts( "CmpFSec.Sh_O"       ,"ShL.Sh_ICmpFSec"   ,"MeCmpFSec" );
linkPorts( "CmpL.Sh_O"          ,"ShL.Sh_ICmp"       ,"MeCmpL" );
linkPorts( "TrbL.Sh_O"          ,"ShL.Sh_ITrb"       ,"MeTrbL" );
linkPorts( "CmpH.Sh_O"          ,"ShH.Sh_ICmp"       ,"MeCmpH" );
linkPorts( "TrbH.Sh_O"          ,"ShH.Sh_ITrb"       ,"MeTrbH" );


//-------------------------------------------------------------------------
//                          Solver Sequence
//-------------------------------------------------------------------------
// If solver execution sequence is not set by the user (below), the default sequence will be
the order of the element instantiation above


        solver.executionSequence = {
                                "Amb",
                                "FsEng",
                                "InEng",
                                "SpltFan",
                                "CmpL",
                                "CmpH",
                                "CmpFSec",
                                "D130",
                                "FusEngDuct",
                                "BrnPri",
```

```
                "FusEng",
                "BrnSec",
                "TrbH",
                "TrbL",
                "MixPri",
                "ShH",
                "ShL",
                "NozPri",
                "FePri",
                "Perf"
}
```

Standard Turbofan Run File – MixedTurbofan.run

```
#include <InterpIncludes.ncp> // file contains unit names, socket types, error statements,
and some constants

//-------------------------------------------------------------------------
//                        Set Thermodynamic Package
//-------------------------------------------------------------------------
setThermoPackage("GasTbl");  // air properties, developed by Pratt and Whitney
//setThermoPackage("Janaf"); // air properties, developed by Honeywell
//setThermoPackage("FPT"); // custom fluid property tables, developed by the user

//-------------------------------------------------------------------------
//                        Model Definition
//-------------------------------------------------------------------------
// Switch 1 contains a standard mixed turbofan
// Switch 2 contains a mixed turbofan with a duct burner

//#include "MixedTurbofan.mdl"; // switch = 1
#include "MixedTFComparison.mdl"; // switch = 2
real switch = 2;
//-------------------------------------------------------------------------
```

```
//              Print Desired Values to the Command Prompt
//-------------------------------------------------------------------------

// Create a custom function to print model some results to the command window.
// Results will only be printed whenever this function is called.

void printResults() {
        if(switch == 1) {
                // Print out Ambient conditions
                cout << "Altitude = " << Amb.alt << " ft" << endl;
                cout << "Mach Number = " << Amb.MN << endl;
                cout << "Engine Air Flow = " << F0.W << " lbm/s" << endl;
                cout << endl;
                cout << "HP Compressor Power = " << CmpH.pwr << " " <<
CmpH.pwr.units << endl;
                cout << "LP Compressor Power = " << CmpL.pwr << " " <<
CmpL.pwr.units << endl;
                cout << "LP Turbine Power = " << TrbL.pwr << " " << TrbL.pwr.units <<
endl;
                cout << "HP Turbine Power = " << TrbH.pwr << " " << TrbH.pwr.units <<
endl << endl;
                cout << "F0.Tt   = " << F0.Tt << " " << F0.Tt.units << endl;
                cout << "F010.Tt = " << F010.Tt << " " << F010.Tt.units << "     F110.Tt =
" << F110.Tt << " " << F110.Tt.units << endl;
                cout << "F020.Tt = " << F020.Tt << " " << F020.Tt.units << endl;
                cout << "F025.Tt = " << F025.Tt << " " << F025.Tt.units << endl;
```

```
                    cout << "F030.Tt = " << F030.Tt << " " << F030.Tt.units << "    F130.Tt =
" << F130.Tt << " " << F130.Tt.units << endl;
                    cout << "F040.Tt = " << F040.Tt << " " << F040.Tt.units << endl;
                    cout << "F045.Tt = " << F045.Tt << " " << F045.Tt.units << endl;
                    cout << "F050.Tt = " << F050.Tt << " " << F050.Tt.units << "    F150.Tt =
" << F150.Tt << " " << F150.Tt.units << endl;
                    cout << "F070.Tt = " << F070.Tt << " " << F070.Tt.units << endl;
                    cout << "F090.Tt = " << F090.Tt << " " << F090.Tt.units << endl;
                    cout << endl;
                    cout << "F0.Pt   = " << F0.Pt << " " << F0.Pt.units << endl;
                    cout << "F010.Pt = " << F010.Pt << " " << F010.Pt.units << "    F110.Pt =
" << F110.Pt << " " << F110.Pt.units << endl;
                    cout << "F020.Pt = " << F020.Pt << " " << F020.Pt.units << endl;
                    cout << "F025.Pt = " << F025.Pt << " " << F025.Pt.units << endl;
                    cout << "F030.Pt = " << F030.Pt << " " << F030.Pt.units << "    F130.Pt =
" << F130.Pt << " " << F130.Pt.units << endl;
                    cout << "F040.Pt = " << F040.Pt << " " << F040.Pt.units << endl;
                    cout << "F045.Pt = " << F045.Pt << " " << F045.Pt.units << endl;
                    cout << "F050.Pt = " << F050.Pt << " " << F050.Pt.units << "    F150.Pt =
" << F150.Pt << " " << F150.Pt.units << endl;
                    cout << "F070.Pt = " << F070.Pt << " " << F070.Pt.units << endl;
                    cout << "F090.Pt = " << F090.Pt << " " << F090.Pt.units << endl;
                    cout << endl;
                    cout << "F0.W    = " << F0.W << " " << F0.W.units << endl;
                    cout << "F010.W = " << F010.W << " " << F010.W.units << "    F110.W =
" << F110.W << " " << F110.W.units << endl;
```

```
                cout << "F020.W = " << F020.W << " " << F020.W.units << endl;
                cout << "F025.W = " << F025.W << " " << F025.W.units << endl;
                cout << "F030.W = " << F030.W << " " << F030.W.units << "    F130.W =
" << F130.W << " " << F130.W.units << endl;
                cout << "F040.W = " << F040.W << " " << F040.W.units << endl;
                cout << "F045.W = " << F045.W << " " << F045.W.units << endl;
                cout << "F050.W = " << F050.W << " " << F050.W.units << "    F150.W =
" << F150.W << " " << F150.W.units << endl;
                cout << "F070.W = " << F070.W << " " << F070.W.units << endl;
                cout << "F090.W = " << F090.W << " " << F090.W.units << endl;
                cout << endl;
                cout << "Primary Wfuel = " << FusEng.Wfuel*3600 << " lbm/hr" << endl;
                cout << "SFC = " << Perf.SFC << " " << Perf.SFC.units <<  endl;
                cout << "Gross Thrust = " << NozPri.Fg << " " << NozPri.Fg.units <<
endl;
                cout << "Net Thrust = " << Perf.Fn << " " << Perf.Fn.units <<  endl <<
endl;
        }
        else {
                // Print out Ambient conditions
                cout << "Solver Indep and Dep variables" << endl;
                cout << endl;
                cout << solver.list( "Independent" );
                cout << endl;
                cout << solver.list( "Dependent" );
                cout << endl;
```

```
                cout << "Altitude = " << Amb.alt << " ft" << endl;
                cout << "Mach Number = " << Amb.MN << endl;
                cout << "Engine Air Flow = " << F0.W << " lbm/s" << endl;
                cout << endl;
                cout << "HP Compressor Power = " << CmpH.pwr << " " <<
CmpH.pwr.units << endl;
                cout << "LP Compressor Power = " << CmpL.pwr << " " <<
CmpL.pwr.units << endl;
                cout << "LP Turbine Power = " << TrbL.pwr << " " << TrbL.pwr.units <<
endl;
                cout << "HP Turbine Power = " << TrbH.pwr << " " << TrbH.pwr.units <<
endl << endl;
                cout << "HP Turbine PR = " << TrbL.S_map.parmMap << endl;
                cout << "LP Turbine PR = " << TrbH.S_map.parmMap << endl << endl;
                cout << "F0.Tt   = " << F0.Tt << " " << F0.Tt.units << endl;
                cout << "F010.Tt = " << F010.Tt << " " << F010.Tt.units << "     F110.Tt =
" << F110.Tt << " " << F110.Tt.units << endl;
                cout << "F020.Tt = " << F020.Tt << " " << F020.Tt.units << "     F120.Tt =
" << F120.Tt << " " << F120.Tt.units << endl;
                cout << "F025.Tt = " << F025.Tt << " " << F025.Tt.units << "     F125.Tt =
" << F125.Tt << " " << F125.Tt.units << endl;
                cout << "F030.Tt = " << F030.Tt << " " << F030.Tt.units << endl;
                cout << "F040.Tt = " << F040.Tt << " " << F040.Tt.units << endl;
                cout << "F045.Tt = " << F045.Tt << " " << F045.Tt.units << endl;
                cout << "F050.Tt = " << F050.Tt << " " << F050.Tt.units << "     F150.Tt =
" << F150.Tt << " " << F150.Tt.units << endl;
```

```
                cout << "F070.Tt = " << F070.Tt << " " << F070.Tt.units << endl;
                cout << "F090.Tt = " << F090.Tt << " " << F090.Tt.units << endl;
                cout << endl;
                cout << "F0.Pt   = " << F0.Pt << " " << F0.Pt.units << endl;
                cout << "F010.Pt = " << F010.Pt << " " << F010.Pt.units << "    F110.Pt =
" << F110.Pt << " " << F110.Pt.units << endl;
                cout << "F020.Pt = " << F020.Pt << " " << F020.Pt.units << "    F120.Pt =
" << F120.Pt << " " << F120.Pt.units << endl;
                cout << "F025.Pt = " << F025.Pt << " " << F025.Pt.units << "    F125.Pt =
" << F125.Pt << " " << F125.Pt.units << endl;
                cout << "F030.Pt = " << F030.Pt << " " << F030.Pt.units << endl;
                cout << "F040.Pt = " << F040.Pt << " " << F040.Pt.units << endl;
                cout << "F045.Pt = " << F045.Pt << " " << F045.Pt.units << endl;
                cout << "F050.Pt = " << F050.Pt << " " << F050.Pt.units << "    F150.Pt =
" << F150.Pt << " " << F150.Pt.units << endl;
                cout << "F070.Pt = " << F070.Pt << " " << F070.Pt.units << endl;
                cout << "F090.Pt = " << F090.Pt << " " << F090.Pt.units << endl;
                cout << endl;
                cout << "F0.W    = " << F0.W << " " << F0.W.units << endl;
                cout << "F010.W = " << F010.W << " " << F010.W.units << "       F110.W
= " << F110.W << " " << F110.W.units << endl;
                cout << "F020.W = " << F020.W << " " << F020.W.units << "    F120.W =
" << F120.W << " " << F120.W.units << endl;
                cout << "F025.W = " << F025.W << " " << F025.W.units << "    F125.W =
" << F125.W << " " << F125.W.units << endl;
                cout << "F030.W = " << F030.W << " " << F030.W.units << endl;
```

```
                cout << "F040.W = " << F040.W << " " << F040.W.units << endl;
                cout << "F045.W = " << F045.W << " " << F045.W.units << endl;
                cout << "F050.W = " << F050.W << " " << F050.W.units << "    F150.W =
" << F150.W << " " << F150.W.units << endl;
                cout << "F070.W = " << F070.W << " " << F070.W.units << endl;
                cout << "F090.W = " << F090.W << " " << F090.W.units << endl;
                cout << endl;
                cout << "Wfuel = " << FusEng.Wfuel*3600 << " lbm/hr" << endl;
                cout << "Duct Wfuel = " << FusEngDuct.Wfuel*3600 << " lbm/hr" <<
endl;
                cout << "Overall TSFC = " <<
(3600*(FusEng.Wfuel+FusEngDuct.Wfuel))/Perf.Fn << " lbm/hr" << endl;
                cout << "Gross Thrust = " << NozPri.Fg << " " << NozPri.Fg.units <<
endl;
                cout << "Net Thrust = " << Perf.Fn << " " << Perf.Fn.units <<  endl <<
endl;
        }
}

//---------------------------------------------------------------------
//                      Running the Model
//---------------------------------------------------------------------
cout << endl;
cout << "======================================= \n";
cout << "=====    RUNNING DESIGN POINT     ===== \n";
cout << "======================================= \n";
```

```
cout << endl;

// set the model design/offdesign switch to design mode
setOption("switchDes", "DESIGN"); // DESIGN or OFFDESIGN
// set the model solution mode switch to steady state
setOption( "solutionMode", "STEADY_STATE" ); // STEADY_STATE (default) or
TRANSIENT

autoSolverSetup();
// Run the model
run();

// Call function (defined earlier in this file) to print results to the command window
printResults();
```

Pulse Detonation Combustor Code – pdc.int

```
#ifndef __PDC__
#define __PDC__

#include <InterpIncludes.ncp>

class PulseDetonationCombustor extends Element {
// ---------------------------------------------------------
// ******* DOCUMENTATION *******
// ---------------------------------------------------------
title = "";
description = isA () + " will calculate performance for pulsed detonation combustor .";
usageNotes = "The burner element performs high level burner performance calculations .
This element works with an entrance fluid and fuel stream. It mixes the two flows together
and then performs the burn calculations. Please note that the burner has no control over
the actual fuel stream conditions -- fuel type, LHV, etc. These values are properties of the
fuel flow itself
```

and are usually set in the FuelStart element. There are two ways to specify the burner exit conditions. The first way is specify the burner fuel -to -air ratio. The second way is to set equivalence ratio. The type of input used is controlled by an option switch. The burner tracks several different pressure losses. The first, dPqP, accounts for duct friction pressure drops and approximates the pressure loss through valves. The second, dPqPRayleigh, accounts for the Rayleigh pressure drop. dPRayleigh is input or 50 calculated - see switchHotLoss, an iteration is necessary since the pressure loss itself is a function of the exit conditions. The burner also allow two efficiencies to be input. The first efficiency, eff, refers to the efficiency based on enthalpy
55 change. The second efficiency, effChem, refers to the efficiency based on temperature change . Both terms can be input. However, the enthalpy efficiency is always applied first. Additionally, the user can request a pre burner pressure loss dPqP. The pressure loss calculations are performed before all the other calculations are done . This means that the combustion entrance
pressure will not match the value indicated by the burner entrance. The user can request a heat transfer Qhx. The heat transfer calculations are performed after all the other calculations are
done. This means that if heat transfer is being used, the exit temperature will not match the value indicated by the burner calculations.";
background = "";

// ---

// ******* SETUP VARIABLES ********

// ---

real x2 {

```
        value = 0.0; IOstatus = "output"; units = "ft";
        description = "Location of leading shock characteristic";
}

real x3 {
        value = 0.0; IOstatus = "output"; units = "ft";
        description = "Location of trailing rarefaction characteristic";
}

real xir {
        value = 0.0; IOstatus = "output"; units = "ft";
        description = "Location of first reflected rarefaction characteristic";
}

real xrf {
        value = 0.0; IOstatus = "output"; units = "ft";
        description = "Location of second reflected rarefaction characteristic";
}

real Tlast {
        value = 0.0; IOstatus = "output"; units = "R";
        description = "Previous value of burner temp. for determining max/avg value";
}
```

```
real Plast {
        value = 0.0; IOstatus = "output"; units = "psia";
        description = "Previous value of burner pressure for determining max/avg value";
}

real Hlast {
        value = 0.0; IOstatus = "output"; units = "BTU/lbm";
        description = "Previous value of burner enthalpy for determining max/avg value";
}

real Tmax {
        value = 0.0; IOstatus = "output"; units = "R";
        description = "Maximum value of temperature up to time t";
}

real Pmax {
        value = 0.0; IOstatus = "output"; units = "psia";
        description = "Maximum value of pressure up to time t";
}

real Hmax {
        value = 0.0; IOstatus = "output"; units = "BTU/lbm";
        description = "Maximum value of enthalpy up to time t";
}

real pCJ {
```

```
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Pressure ratio given for the detonation by CEA";
}

real TCJ {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Temperature ratio given for the detonation by CEA";
}

real uCJ {
        value = 0.0; IOstatus = "input"; units = "ft/s";
        description = "Wave speed given for the detonation by CEA";
}

real rhoCJ {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Density ratio given for the detonation by CEA";
}


real gamCJ {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Specific heat ratio given for the detonation by CEA";
}

real RCJ {
```

```
        value = 0.0; IOstatus = "input"; units = "(ft*lbf)/(lbm*R)";
        description = "Gas constant given for the detonation by CEA";
}

real a3 {
        value = 0.0; IOstatus = "input"; units = "ft/s^2";
        description = "Speed of sound behind detonation wave";
}

real a_dPqP {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Duct friction pressure drop adder";
}

real a_dPqPAud {
        value = 0.0; IOstatus = "unset"; units = "psia";
        description = "Audit factor adder applied to pressure ratio";
}

real a_eff {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Adiabatic efficiency adder";
}

real a_effChem {
        value = 0.0; IOstatus = "input"; units = " none ";
```

```
        description = "Chemical efficiency adder";
}

real ARvalve {
        value = 0.5; IOstatus = "input"; units = "none";
        description = "Ratio of valve throat area to tube cross section area";
}

real deltaS {
        value = 0.0; IOstatus = "output"; units = "none";
        description = "Change in entropy due to detonation";
}

real DDT {
        value = 0.0005; IOstatus = "input"; units = "seconds";
        description = "Detonation to deflaration time in seconds";
}

real dPqP {
        value = 0.0; IOstatus = "output"; units = "none";
        description = "Adjusted duct friction pressure drop";
}


real dPqPBase {
        value = 0.0; IOstatus = "input"; units = "none";
```

```
        description = "Duct friction pressure drop";
}
real dPqPRayleigh {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Adjusted Rayleigh pressure drop";
}

real dTube {
        value = 2.0; IOstatus = "input"; units = "inches";
        description = "Inside diameter of the detonation tube";
}

real eff {
        value = 1.0; IOstatus = "output"; units = "none";
        description = "Adjusted adiabatic burner efficiency";
}

real effBase {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Adiabatic burner efficiency, from socket";
}

real effChem {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Adjusted chemical efficiency";
}
```

```
real effChemBase {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Chemical efficiency, from socket";
}

real eqRatio {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Equivalence ratio for fuel-air mixture";
}

real FAR {
        value = 0.0; IOstatus = "output"; units = "none";
        description = "Fuel-air ratio";
}

real FARDes {
        value = 0.0; IOstatus = "output"; units = "none";
        description = "Fuel-to-air ratio at design";
}

real fillFrac {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Fill fraction";
}
```

```
real freq {
        value = 0.0; IOstatus = "output"; units = "Hz";
        description = "Detonation frequency";
}

real fuelFractV {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Fraction of the incoming flow velocity fuel enters the burner";
}

real Haverage {
        value = 0.0; IOstatus = "output"; units = "(ft*lbf)/lbm";
        description = "Continuously updated enthalpy average";
}

real Hinput {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Input cycle average enthalpy value for design mode";
}

real iBPR {
        value = 1.4; IOstatus = "output"; units = "none";
        description = "Bypass ratio internal to the PDC";
}
```

```
real iBPRdes {
        value = 1.0; IOstatus = "output"; units = "none";
        description = "Bypass ratio internal to the PDC at design conditions";
}

real lTube {
        value = 36; IOstatus = "input"; units = "inches";
        description = "Length of the individual detonation tubes";
}

real n_tubes {
        value = 36; IOstatus = "input"; units = "none";
        description = "Total number of detonation tubes used in the PDC";
}

real MCJ {
        value = 3.0; IOstatus = "output"; units = "none";
        description = "Chapman-Jouguet Mach number of the detonation wave";
}

real Mvalve {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Mach number of flow passing through the valve throat";
}
```

```
real qadd {
        value = 0.0; IOstatus = "output"; units = "none";
        description = "Heat addition due to fuel combustion";
}


real Qhx {
        value = 0.0; IOstatus = "input"; units = "Btu/sec";
        description = "Heat loss to thermal mass storage";
}

string region {
        value = " "; IOstatus = "output"; units = "none";
        description = "Detonation region at point x at time t";
}

real out {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Output value from discrete solver function";
}

real Paverage {
        value = 0.0; IOstatus = "output"; units = "psia";
        description = "Continuously updated pressure average";
}
```

```
real Pinput {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Input cycle average pressure value for design mode";
}

real PqPRayleigh {
        value = 1.0; IOstatus = "output"; units = "none";
        description = "Adjusted Rayleigh pressure drop";
}

real PqPRayleighDelta {
        value = 0.0; IOstatus = "output"; units = "none";
        description = "Bounded Rayleigh pressure drop - for loop only";
}

real PqPRayleighError {
        value = 1.0; IOstatus = "output"; units = "none";
        description = "Adjusted Rayleigh pressure drop error";
}

real PqPRayleighMin {
        value = 0.05; IOstatus = "input"; units = "none";
        description = "Rayleigh pressure drop lower limit - for loop only";
}

real PqPRayleighStep {
```

```
        value = 0.05; IOstatus = "input"; units = "none";
        description = "Maximum step for Rayleigh pressure drop - for loop only";
}

real PqPRayleighNew {
        value = 1.0; IOstatus = "output"; units = "none";
        description = "Previous adjusted Rayleigh pressure drop - for loop only";
}


real purgeFrac {
        value = 0.25; IOstatus = "input"; units = "none";
        description = "Purge fraction coefficient for flow";
}

real s_dPqP {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Duct friction pressure drop scalar";
}

real s_dPqPAud {
        value = 1.0; IOstatus = "unset"; units = "none";
        description = "Audit factor scalar applied to pressure ratio";
}

real s_eff {
```

```
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Adiabatic efficiency scalar";
}

real s_effChem {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Chemical efficiency scalar";
}

real tauBlDn {
        value = 5.; IOstatus = "input"; units = "none";
        description = "Blowdown time constant";
}

real tauValveOpen {
        value = 0.33333; IOstatus = "output"; units = "none";
        description = "Time valve open/time cycle - from 0 to 1";
}

real Taverage {
        value = 0.0; IOstatus = "output"; units = "Rankine";
        description = "Continuously updated temperature average";
}

real tCycle {
        value = 0.01; IOstatus = "output"; units = "seconds";
```

```
        description = "Detonation engine cycle time (= 1/frequency)";
}

real time {
        value = 0.0; IOstatus = "input"; units = "seconds";
        description = "Time used in transient calculation";
}

real tInterval {
        value = 0.0; IOstatus = "input"; units = "seconds";
        description = "Time interval used in pressure/temp average";
}


real tExhaust {
        value = 0.0; IOstatus = "output"; units = "seconds";
        description = "Time at which the valve opens and the tube is purged";
}

real tolRayleigh {
        value = 4e-05; IOstatus = "input"; units = "none";
        description = "Iteration tolerance on momentum pressure drop";
}

real tolWfuel {
        value = 1e-05; IOstatus = "input"; units = "none";
```

```
        description = "Iteration tolerance on temperature burn";
}

real TtCombOut {
        value = 0.0; IOstatus = "input"; units = "R";
        description = "Exit temperature";
}

real TtLast {
        value = 0.0; IOstatus = "input"; units = "R";
        description = "Previous exit temperature - for loop only";
}

real TTSSeff {
        value = 1.0; IOstatus = "input"; units = "none";
        description = "Efficiency factor for the transition device";
}

real TTSSdPqP {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Change in Pressure divided by Pressure for transistion to steady
state calculation";
}

real tBlowdown {
        value = 0.0; IOstatus = "input"; units = "seconds";
```

```
        description = "Blowdown time";
}

real tBlowdownComp {
        value = 0.0; IOstatus = "input"; units = "seconds";
        description = "Blowdown time comparison";
}

real tValve {
        value = 0.0002; IOstatus = "input"; units = "seconds";
        description = "Time for valves to open/close";
}

real u3 {
        value = 0.0; IOstatus = "input"; units = "ft/s";
        description = "Velocity behind detonation wave";
}


real Wfuel {
        value = 0.0; IOstatus = "input"; units = "lbm/sec";
        description = "Combustor fuel flow";
}

real WfuelError {
        value = 0.0; IOstatus = "input"; units = "lbm/sec";
```

```
        description = "Combustor fuel flow error";
}

real WfuelLast {
        value = 0.0; IOstatus = "input"; units = "lbm/sec";
        description = "Previous combustor fuel flow - for loop only";
}

real WfuelNew {
        value = 0.0; IOstatus = "input"; units = "lbm/sec";
        description = "Next combustor fuel flow - for loop only";
}

real xval {
        value = 0.0; IOstatus = "input"; units = "none";
        description = "Non-dimensional location along tube";
}

int countFuel {
        value = 0; IOstatus = "output";
        description = "Fuel loop counter";
}

int countFuelMax {
        value = 50; IOstatus = "input";
        description = "Fuel loop maximum counter";
```

```
}

int countRayleigh {
        value = 0; IOstatus = "output";
        description = "Rayleigh loop counter";
}

int countRayleighMax {
        value = 25; IOstatus = "input";
        description = "Rayleigh loop maximum counter";
}

int flagRayleighLossTooMuch {
        value = 0; IOstatus = "output";
        description = "If true, Rayleigh loop results in too much loss";
}

int flagRayleighChoked {
        value = 0; IOstatus = "output";
        description = "If true, Rayleigh loop results in supersonic flow";
}


// for backward compatibilty with old " aud "
FunctVariable a_dPqPaud {
        units = "none"; IOstatus = "input";
```

```
        getFunction = "get_aAud"; setFunction = "set_aAud";
}

real get_aAud() {
        return a_dPqPAud;
}

void set_aAud(real userValue) {
        a_dPqPAud = userValue;
}

FunctVariable s_dPqPaud {
        units = "none"; IOstatus = "input";
        getFunction = "get_sAud"; setFunction = "set_sAud";
}

real get_sAud() {
        return s_dPqPAud;
}

void set_sAud(real userValue) {
        s_dPqPAud = userValue;
}
// ---------------------------------------------------------
// ******* OPTION VARIABLE SETUP *******
// ---------------------------------------------------------
```

```
Option switchAud {
        allowedValues = { "BASE" , "AUDIT" }
        description = "Determines if the audit factors are used";
        IOstatus = "input";
        trigger = TRUE ;
}

Option switchBurn {
        allowedValues = { "FAR" , "EQRATIO", "FUEL", "WFUEL", "TEMPERATURE" };
        description = "Switch determines if burner is running to fuel flow , FAR , or T4 .
Setting option to FUEL  will burn using the burner value as an input . Setting the option to
WFUEL will burn using the value coming in from the fuel station .";
        trigger = TRUE ;
}

Option switchDes {
        allowedValues = { "DESIGN" , "TRANSIENT" , "ORIGINAL" };
        description = "Design switch";
        trigger = FALSE ;
}

// input kept in for backward compatible ( remove later )
Option switchHotLoss {
        allowedValues = { "INPUT" , "CALCULATE", "input" };
        description = "Switch determines if the hot pressure loss is input or iterated on";
        trigger = TRUE ;
```

```
}

// ---------------------------------------------------------
// **** SETUP PORTS , FLOW STATIONS , SOCKETS , TABLES ******
// ---------------------------------------------------------
// FLUID PORTS
FluidInputPort Fl_I {
        description = "Incoming flow";
}

FluidOutputPort Fl_O1 {
        description = "Exiting combustion flow";
}

FluidOutputPort Fl_O2 {
        description = "Exiting bypass flow";
}

// FUEL PORTS
FuelInputPort Fu_I {
        description = "Incoming fuel flow";
}

// BLEED PORTS
// THERMAL PORTS
// MECHANICAL PORTS
```

```
// FLOW STATIONS

FlowStation Fl_Icomb {
        description = "Inlet station to detonation tube section of burner ( after the initial
pressure loss is applied )";
}

FlowStation Fl_IcombAir {
        description = "Copy of the inlet station to detonation tube section of burner ( after
the initial pressure loss is applied, before flow is split and partitioned )";
}

FlowStation Fl_Iprg {
        description = "Station containing detonation tube purge fluid ";
}

FlowStation Fl_Ocomb {
        description = "Exit station to combustion section of burner (before thermal
storage heat transfer is calculated )";
}

FlowStation Fl_Vit {
        description = "Vitiated Fluid flow station before detonation (cold)";
}

// SOCKETS
```

```
Socket S_dPqP {
        allowedValues = { "dPqPBase" };
        description = "Dry duct and valve pressure loss "; // __ mod - socketType = "
dPqP ";
}

Socket S_eff {
        allowedValues = { "effBase" , "effChemBase" };
        description = "PulseDetonationCombustor adiabatic efficiency";
        socketType = "BURN_EFFICIENCY";
}

Socket S_Qhx {
        allowedValues = { "Qhx" };
        description = "Thermal storage socket";
        socketType = "HEATTRANSFER";
}

// TABLES
// -------------------------------------------------------
// ******* INTERNAL SOLVER SETUP *******
// -------------------------------------------------------
// -------------------------------------------------------
// ****** ADD SOLVER INDEPENDENTS & DEPENDENTS ******
// -------------------------------------------------------
```

```
// ---------------------------------------------------------
// ******* VARIABLE CHANGED METHODOLOGY *******
// ---------------------------------------------------------
void variableChanged(string name, any oldVal) {
// Check to see what variables were changed ....
// Change input / output status as necessary
if(name == "switchBurn") {
        if(switchBurn == "FAR") {
                FAR.IOstatus = "input";
                Wfuel.IOstatus = "output";
                TtCombOut.IOstatus = "output";
                eqRatio.IOstatus = "output";
        }

        else if(switchBurn == "FUEL") {
                FAR.IOstatus = "output";
                Wfuel.IOstatus = "input";
                TtCombOut.IOstatus = "output";
        }

        else if(switchBurn == "WFUEL") {
                FAR.IOstatus = "output";
                Wfuel.IOstatus = "output";
                TtCombOut.IOstatus = "output";
        }
```

```
		else if(switchBurn == "EQRATIO") {
			FAR.IOstatus = "output";
			Wfuel.IOstatus = "output";
			TtCombOut.IOstatus = "output";
			eqRatio.IOstatus = "input";
		}
}

else if(name == "switchHotLoss") {
		if(switchHotLoss == "INPUT") {
			dPqPRayleigh.IOstatus = "input";
		}
		else if(switchHotLoss == "input"){
			switchHotLoss = "INPUT";
		}
		else {
			dPqPRayleigh.IOstatus = "output";
		}
}

else if(name == "switchAud") {
		a_dPqPAud.IOstatus = "inactive";
		s_dPqPAud.IOstatus = "inactive";
		if(switchAud == "AUDIT") {
			a_dPqPAud.IOstatus = "input";
			s_dPqPAud.IOstatus = "input";
```

```
		}
}
}



// ---------------------------------------------------------
// ******* PERFORM ENGINEERING CALCULATIONS *******
// ---------------------------------------------------------
void calcPreLoss() {


// ---------------------------------------------------------
// Check to see if the pressure sockets are empty , if not then execute
// ---------------------------------------------------------
if(!S_dPqP.isEmpty()) {
		S_dPqP.execute();
}
dPqP = dPqPBase*s_dPqP + a_dPqP ; // calculate pressure losses ( dry duct and Valve )


if(switchDes == "OFFDESIGN") {
		if(switchAud == "AUDIT") {
			dPqP = dPqP*s_dPqPAud + a_dPqPAud;
		}
}


// Collect total enthalpy at inlet
real hin = Fl_I.ht;
```

```
real Pin = (1 - dPqP)*Fl_I.Pt;

// copy flow to combustor flow
Fl_Icomb.copyFlowStatic("Fl_I");
Fl_Icomb.setTotal_hP(hin, Pin);
}



void calcBurn() {
        real TtCombOutTemp;
        real htStoich;
        real WFuelLimit;
        real WFuelHeat;
        Fl_Ocomb.copyFlow("Fl_Icomb");

// ---------------------------------------------------------
// Efficiency
// ---------------------------------------------------------
if(!S_eff.isEmpty()) {
        S_eff.execute();
}
eff = effBase*s_eff + a_eff ;
effChem = effChemBase*s_effChem + a_effChem ;

// ---------------------------------------------------------
// Burn
```

```
// ---------------------------------------------------------
Fl_Ocomb.burn("Fu_l", eff);

// ---------------------------------------------------------
// if inputting a PW type of efficiency adjust the temperature
// ---------------------------------------------------------
if(effChem < 1.0) {
        TtCombOutTemp = effChem*(Fl_Ocomb.Tt - Fl_Icomb.Tt) + Fl_Icomb.Tt;
        Fl_Ocomb.setTotalTP(TtCombOutTemp, Fl_Icomb.Pt); // use Pin
}
}


void calcRayleighLoss() {
        flagRayleighChoked = 0;
        flagRayleighLossTooMuch = 0;
        PqPRayleigh = 1.0;
        PqPRayleighError = 0.0;

// ---------------------------------------------------------
// self - convergent iteration loop for internal momentum pressure drop calc
// ---------------------------------------------------------
for(countRayleigh = 0; countRayleigh <= countRayleighMax; countRayleigh++) {
// -----------------------------------------------------
// input or output dPqPRayleigh
// -----------------------------------------------------
```

```
if(switchHotLoss == "INPUT") {
        PqPRayleigh = 1.0 - dPqPRayleigh;
}

else if(switchHotLoss == "CALCULATE") {
        dPqPRayleigh = 1.0 - PqPRayleigh;
}

// -------------------------------------------------------
// calculate momentum pressure drop
// -------------------------------------------------------
real PtCombOut = PqPRayleigh*Fl_Icomb.Pt;
Fl_Ocomb.setTotal_hP(Fl_Ocomb.ht, PtCombOut);

// -------------------------------------------------------
// Check momentum pressure drop
// -------------------------------------------------------
PqPRayleighNew = PqPRayleigh;

if(switchHotLoss == "CALCULATE") {
// -------------------------------------------------------
// make this thing a constant area burner
// -------------------------------------------------------
Fl_Ocomb.A = Fl_Icomb.A;
flagRayleighChoked = 0;
```

```
// when MN > 1.0 FlowStation static calc is not consistent with Area
if(Fl_Ocomb.MN > 1.0) {
        //Fl_Ocomb.MN = 0.6; // do not do this - creates major iteration problems
        flagRayleighChoked = 1;
}

// ---------------------------------------------------------
// Calculate the exit static pressure from the momentum eqn
// assume the fuel has the same velocity as the entrance flow
// ---------------------------------------------------------
real PsMomMeth1;
PsMomMeth1 = Fl_Icomb.W*Fl_Icomb.V - Fl_Ocomb.W*Fl_Ocomb.V;
PsMomMeth1 = PsMomMeth1/C_GRAVITY;
PsMomMeth1 = PsMomMeth1 + Fl_Icomb.Ps*Fl_Icomb.A;
PsMomMeth1 = PsMomMeth1/Fl_Ocomb.A;

real PsMomMeth2;
// PsMomMeth2 = Fl_Ocomb.W*Fl_Icomb.V;
PsMomMeth2 = Fl_Icomb.W*Fl_Icomb.V + Wfuel*Fl_Icomb.V*fuelFractV;
PsMomMeth2 = PsMomMeth2/C_GRAVITY;
PsMomMeth2 = PsMomMeth2 + Fl_Icomb.Ps*Fl_Icomb.A;
PsMomMeth2 = PsMomMeth2/Fl_Ocomb.A;
PsMomMeth2 = PsMomMeth2/(1.0 + Fl_Ocomb.gams*Fl_Ocomb.MN*Fl_Ocomb.MN);
// PsMomMeth1 = PsMonMeth2;
// ---------------------------------------------------------
// Note Meth1 = Meth2 when MN <= 1.0
```

```
// Use Meth2 - seems more stable the Meth1 when MN > 1.0
// ----------------------------------------------------
PqPRayleighNew = (PsMomMeth2/Fl_Ocomb.Ps)*PqPRayleigh;
}

// Check against tolerance
PqPRayleighError = PqPRayleighNew - PqPRayleigh;
if(abs(PqPRayleighError) < tolRayleigh) {
        break;
}

// Bounding of PqPRayleigh movement to PqPRayleighStep
real sign;
sign = PqPRayleighError/abs(PqPRayleighError);
PqPRayleighDelta = sign*min(abs(PqPRayleighError), PqPRayleighStep);
PqPRayleighNew = PqPRayleigh + PqPRayleighDelta;

// Lower limit of PqPRayleigh - limit too much loss to PqPRayleighMin
if(PqPRayleighNew < PqPRayleighMin) {
        if(flagRayleighLossTooMuch == 1) {
                ESOreport(1023901, "Rayleigh pressure loss limited, too much loss" ,
FALSE);
                break ;
        }
        PqPRayleighNew = PqPRayleighMin;
        flagRayleighLossTooMuch = 1;
```

```
}
else {
        flagRayleighLossTooMuch = 0;
}

/*
// debug info
cout << "Fl_Ocomb.A = " << Fl_Ocomb.A << endl;
cout << "Fl_Ocomb.MN = " << Fl_Ocomb.MN << endl;
cout << "Fl_Ocomb.Ps = " << Fl_Ocomb.Ps << endl;
cout << "PsMomMeth1 = " << PsMomMeth1 << endl;
cout << "PsMomMeth2 = " << PsMomMeth2 << endl;
cout << "PqPRayleigh = " << PqPRayleigh << endl;
cout << "PqPRayleighNew = " << PqPRayleighNew << endl;
cout << endl;
*/

// ------------------------------------------------------
// check for convergence
// ------------------------------------------------------
if(countRayleigh >= countRayleighMax) {
        ESOreport(1023901 , "Rayleigh iteration failed to converge, counter exceed
max", FALSE);
        break ;
}
```

```
PqPRayleigh = PqPRayleighNew;
}

if(flagRayleighChoked == 1) {
        ESOreport(1023901 , "Rayleigh Fl_Ocomb.MN exceed choked condition" ,
FALSE);
}
}


void calculate() {
// ---------------------------------------------------------
// Preburning pressure loss
// ---------------------------------------------------------
calcPreLoss(); // creates Fl_Icomb, applies pre - losses
real FARin = Fl_Icomb.FAR;
real WARin = Fl_Icomb.WAR;

/*
cout << "Fl_I.MN = " << Fl_I.MN << endl;
cout << "Fl_I.Aphy = " << Fl_I.Aphy << endl;
cout << "Fl_I.ht = " << Fl_I.ht << endl;
cout << "Fl_I.Pt = " << Fl_I.Pt << endl;
cout << "Fl_I.s = " << Fl_I.s << endl;
cout << endl;
*/
```

```
if(Fl_I.MN == 0. && Fl_I.Aphy == 0.) {
        Fl_Icomb.MN = 0.4;
        Fl_Icomb.setTotal_hP(Fl_Icomb.ht, Fl_Icomb.Pt);
}

// ---------------------------------------------------------
// Pre - calculate Burning to obtain enthalpy ,etc .
// ---------------------------------------------------------
if(switchBurn == "FAR") {
// ---------------------------------------------------
// determine the fuel weight flow from the input FAR
// ---------------------------------------------------
        Wfuel = (Fl_Icomb.W/(1 + FARin + WARin))*(FAR - FARin);
        Fu_I.Wfuel = Wfuel;
        eqRatio = FAR/Fu_I.FARst;

calcBurn();
calcRayleighLoss();
TtCombOut = Fl_Ocomb.Tt;
}


// do an equivalence ratio calculation
else if(switchBurn == "EQRATIO") {
        FAR = eqRatio*Fu_I.FARst ;
```

```
        Wfuel = (Fl_Icomb.W/(1 + FARin + WARin))*(FAR - FARin);
        Fu_I.Wfuel = Wfuel;
        calcBurn();
        calcRayleighLoss();
        TtCombOut = Fl_Ocomb.Tt;
}

// -------------------------------------------------------
// make a flow station that has props of cold vitiated air
// -------------------------------------------------------
Fl_Vit.copyFlowStatic("Fl_Ocomb");
Fl_Vit.setTotalTP(Fl_Icomb.Tt, Fl_Icomb.Pt);

// -------------------------------------------------------
// copy inlet flow for pure air reference to be used later
// -------------------------------------------------------
// Take a snapshot of air after it has entered detn tubes
Fl_IcombAir.copyFlowStatic("Fl_Icomb");
// Copy input flow properties for internal bypass flow
Fl_O2.copyFlow("Fl_IcombAir");



// -------------------------------------------------
// On - design loop
// -------------------------------------------------
if(switchDes == "DESIGN") {
```

```
// ------------------------------------------------
// Initialize local variables
// ------------------------------------------------
        real uCJ, a_1, rhoVit, freq, PcqPi, errors;
        real gamt, Cpt, beta, MCJ2, PcqPi2;
        real Atube, Vtube, mCycle, Wtube;
        real MFP, Wvalve, gma_l;
        real mFillAir, mPurgeAir, mPureAir;
        real tDetonation, tDetProp, tBlowdown, tPurge, tFill, iVel;
        real gam_s, gmm_fc;
        real WtotAir, Wbypass;
        int count;

// ---- initiated but not iterated ----------------------
// static density of cool vitiated fluid
        rhoVit = Fl_Vit.rhot; //(lbm /ft ^3)

// speed of sound in pure air, stagnated in detonation
// tube that the detonation wave propogates in to
        a_1 = sqrt(Fl_Icomb.gamt*Fl_Icomb.Rt*Fl_Icomb.Tt*25037.);

// ================================================
// Calculate Chapman - Jouguet Mach number for wave
// as described in Heiser and Pratt
// ================================================
// *** input variables : //
```

```
// *** output variables : // MCJ , deltaS , qadd //
// *** Flow Stations : // Fl_Ocomb , Fl_Icomb //
// local variables : // gamt , Cpt , qadd , beta , MCJ2 //

// ------ Arithmetically average specific heats -----------
// arithmetic mean of gamma for stopped fluid
        gamt = (Fl_Ocomb.gamt + Fl_Icomb.gamt)/2.0;

// arithmetic mean of Cp for a stopped fluid
        Cpt = (Fl_Ocomb.Cpt + Fl_Icomb.Cpt)/2.0;
// ----- Calculate heat addition per Heiser - Pratt cycle ---
// calculate non - dimensional heat addition
        qadd = (Fl_Ocomb.ht - Fl_Icomb.ht)/(Cpt* Fl_Icomb.Tt);

// ------- Calculate Chapman - Jouget Mach number --------
        beta = (gamt + 1.0)*qadd+1.0;
        MCJ2 = beta + sqrt(beta **2 - 1.0);
        MCJ = sqrt(MCJ2);

// --------- Calculate Entropy gain based on CJ detonation -
        deltaS = Cpt*(-log(MCJ2*((gamt+1.0)/(1.0+gamt*MCJ2))**((gamt+1.0)/gamt)));

// ---- calculate the pressure rise using the H &P method --
        PcqPi = (1.0+gamt*MCJ2 )/(gamt+1.0);
        uCJ = a_1* MCJ;
```

```
// ------ Calculate tube volume and Area -------------------

        Atube = (PI/4.)*dTube**2/144.; // ft ^2

        Vtube = Atube*(lTube/12) ; // ft ^3


// ------ calculate the valve inlet mass flow rate --------

        gma_I = Fl_IcombAir.gamt;

        MFP = Mvalve*sqrt((gma_I*32.174) /( Fl_IcombAir.Rt *778.16) )*(1.+( gma_I -1.)
/2.* Mvalve **2) **( ( gma_I +1.) /(2.*(1. - gma_I )));

        Wvalve = (Fl_IcombAir.Pt/sqrt(Fl_IcombAir.Tt))*(Atube*144.*ARvalve)*MFP;




// ------------------------------------------------------

// On - Design : Calculate bypass ratio

// ------------------------------------------------------

// *** input Variables : // dTube , lTube , n_tubes , fillFrac

// // purgeFrac ,

// *** iterated Variables // freq

// *** output Variables : // iBPR

// *** local variables : // WfillAir , WpurgeAir , WpureAir , WtotAir

// Wbypass , WpurgeAir , Wvit

// *** Flow Stations : // Fl_IcombAir , Fl_Icomb , Fl_Iprg , Fl_Vit


// ---- Calculate the split and partition of flow ----------

// amount of air that will be mixed with fuel - 1 tube

        mFillAir = Vtube*(rhoVit*fillFrac)/(1.+ FAR);
```

```
// amount of air that will purge during each cycle - 1 tube
        mPurgeAir = Vtube*(Fl_IcombAir.rhos*purgeFrac);

// total air per cycle flowing though one tube
        mPureAir = mFillAir + mPurgeAir;

/*
        cout << "mFillAir = " << mFillAir << endl;
        cout << "mPurgeAir = " << mPurgeAir << endl;
        cout << "mPureAir = " << mPureAir << endl;
        cout << endl;
*/

// ------------------------------------------------------
// Timing - calculate frequency
// ------------------------------------------------------
// *** input Variables : DDT, tValve, lTube, ff, pf, tCycle
// *** iterated Variables : uCJ, PcqPi
// *** output Variables tCycle, tauValveOpen, freq
// *** local variables : tDetonation, tDetProp, tBlowdown, tPurge, tFill
// ------------------------------------------------------

// ---------- Detonation time ----------
// DetProp time is relatively independent of fill fraction
        tDetProp = lTube/(uCJ*12);
```

```
// DDT is input , tDetonationPropogation calcd
// ( may need to iterate )
        tDetonation = DDT + tDetProp;

// ---------- Blowdown time ----------
// assume choked flow at tube exit and calculateon
// blowdown based draw - down time of a pressurized
// tank calculated on pressure differential
        gam_s = Fl_IcombAir.gams; // larger gamma is more conservative
        gmm_fc = ((gam_s + 1)/2)**(-(gam_s + 1)/(2*(gam_s - 1)));

// #### tBlowdown : Use ~1/2 calcd pressure ( to match experimental data )
// we 'll use CJ det wave velocity as the speed of sound in the gas
// since a cannot be directly calc 'd
// note tBlowdown is proportional to tube length
// tauBlDn is proportional to tube length
        tBlowdown = (log(0.4*PcqPi)/gmm_fc)*(lTube/uCJ);

// ---------- Fill and Purge time ----------
// Use the choked flow at valve inlet and the mass flow rate as
// calculated outside the loop to calculate fill time (m/ mdot)
        tPurge = tValve + (mPurgeAir/Wvalve); // seconds
        tFill = tValve + (mFillAir/Wvalve); // seconds

// Improvement could be made by calculating vitiated air velocity ...
```

```
// ----------- Cycle Time output calculation ------------
        tCycle = tDetonation + tBlowdown + tPurge + tFill;
        tauValveOpen = (tPurge + tFill)/tCycle;
        freq = 1./tCycle;
        cout << endl;

// ---------- Set total mass flow through tubes ----------
        WtotAir = mPureAir*n_tubes*freq;

// steady - state flow rate into tubes
// conservation of mass check
if(WtotAir > Fl_I.W) {
        fillFrac = fillFrac*(Fl_I.W/WtotAir);
        purgeFrac = purgeFrac*(Fl_I.W/WtotAir);
        mFillAir = Vtube*(rhoVit*fillFrac)/(1.+ FAR);

// amount of air that will be mixed with fuel - 1 tube
        mPurgeAir = Vtube*(Fl_IcombAir.rhos*purgeFrac);

// amount of air that will purge during each cycle -1 tube
        mPureAir = mFillAir + mPurgeAir;

// total air per cycle flowing though one tube
        WtotAir = Fl_I.W;
        cout << "purgeFrac changed to: " << purgeFrac << endl;
        cout << "fillFrac changed to: " << fillFrac << endl << endl;
```

```
}

// ---------- Set iBPR ----------

        Wbypass = Fl_I.W - WtotAir;

// steady - state flow rate sent to bypass

        iBPR = Wbypass/WtotAir;

// steady - state internal PDC bypass ratio

        iBPRdes = iBPR;

// --------- Set bypass exit flow SPLIT ----------

        Fl_O2.W = Wbypass;

// ----- Set purge and fill stations PARTITION -------

        Fl_Iprg.copyFlowStatic("Fl_IcombAir");

// copy flow for purge function

// ---------- PURGE AIR ----------

        Fl_Iprg.AphyDes = (Atube*144)*n_tubes; // Set phys area

        Fl_Iprg.W = mPurgeAir*freq*n_tubes; // set m dot

// ---------- FILL AIR ----------

        Fl_Icomb.copyFlow("Fl_IcombAir");

        Fl_Icomb.AphyDes = Atube*144.*n_tubes*tauValveOpen;
```

```
// Actual area is multiplied by tauVO to get equivalent
// area . - Fluid flows steadily through this area
        Fl_Icomb.W = WtotAir; //mFillAir*n_tubes*freq;
        Fl_Icomb.setTotal_hP(Fl_IcombAir.ht, Fl_IcombAir.Pt);

/*
        cout << "WtotAir = " << WtotAir << endl;
        cout << "Wbypass = " << Wbypass << endl << endl;
        cout << "mPureAir = " << mPureAir << endl;
        cout << "mPurgeAir = " << mPurgeAir << endl;
        cout << "mFillAir = " << mFillAir << endl;
        cout << endl;
*/

// sets time - averaged static conditions
// ------------------------------------------------
// Burning
// ------------------------------------------------
// FAR was calculated prior to entering this
// point - so we just need to modify
// Wfuel based on changed Fl_Icomb.W
        Wfuel = (Fl_Icomb.W/(1. + FARin + WARin))*(FAR - FARin);
        Fu_I.Wfuel = Wfuel;
        calcBurn();
        calcRayleighLoss();
        TtCombOut = Fl_Ocomb.Tt;
```

```
        //cout << "Fl_Ocomb.Tt = " << Fl_Ocomb.Tt << endl;
        //cout << "Fl_Ocomb.Pt = " << Fl_Ocomb.Pt << endl;

// ====================================================
// Apply Dyer - Kaemming correction to obtain tube flow
// at exit ( ignores the kinetic energy of shock wave .)
// ====================================================
/*
        cout << "deltaS = " << deltaS << endl;
        cout << "Fl_Icomb.s = " << Fl_Icomb.s << endl;
        cout << "Fl_Ocomb.s = " << Fl_Ocomb.s << endl;
        cout << endl;
*/
        Fl_Ocomb.setTotal_hP(Hinput*Fl_Icomb.ht, Pinput*Fl_Icomb.Pt);
}



// ***************************************************
// Transient Loop
// ***************************************************
if(switchDes == "TRANSIENT") {
// ---------------------------------------------
// Initialize local variables
// ---------------------------------------------
        real a_1, rhoVit, freq, PcqPi, errors;
```

```
        real gamt, Cpt, beta, MCJ2, PcqPi2;
        real Atube, Vtube, mCycle, Wtube;
        real MFP, Wvalve, gma_I;
        real mFillAir, mPurgeAir, mPureAir;
        real tDetonation, tDetProp, tBlowdown, tPurge, tFill, iVel;
        real gam_s, gmm_fc;
        real WtotAir, Wbypass;
        int count;

// ---- initiated but not iterated ----------------------
// static density of cool vitiated fluid
        rhoVit = Fl_Vit.rhot; //(lbm/ft^3)

// speed of sound in pure air, stagnated in detonation
// tube that the detonation wave propogates in to
        a_1 = sqrt(Fl_Icomb.gamt*Fl_Icomb.Rt*Fl_Icomb.Tt*25037.);



// ====================================================
// Calculate Chapman - Jouguet Mach number for wave
// as described in Heiser and Pratt
// ====================================================
// *** input variables :
// *** output variables : MCJ, deltaS, qadd
// *** Flow Stations : Fl_Ocomb, Fl_Icomb
// local variables : gamt, Cpt, qadd, beta, MCJ2
```

```
// ------ Arithmetically average specific heats -----------
// arithmetic mean of gamma for stopped fluid
        gamt = (Fl_Ocomb.gamt + Fl_Icomb.gamt)/2.0;

// arithmetic mean of Cp for a stopped fluid
        Cpt = (Fl_Ocomb.Cpt + Fl_Icomb.Cpt)/2.0;
        //Cpt = 2.3190*0.23885; // convert kJ/(kg*K) to Btu/(lbm*R)

        MCJ2 = MCJ**2;
        beta = ((MCJ2**2)+1)/(2*MCJ2);
        qadd = (beta-1)/(gamt+1);

// --------- Calculate Entropy gain based on CJ detonation -
        deltaS = Cpt*(-log(MCJ2*((gamt + 1.0)/(1.0 + (gamt*MCJ2)))**((gamt +
1.0)/gamt)));

// ---- calculate the pressure rise using the H&P method --
        PcqPi = (1.0 + gamt*MCJ2)/(gamt + 1.0);

// ------ Calculate tube volume and Area -------------------
        Atube = (PI*((dTube/2)**2))/144.; // ft^2
        Vtube = Atube*(lTube/12); // ft^3

// ------ calculate the valve inlet mass flow rate --------
```

```
        gma_I = Fl_IcombAir.gamt;
        MFP = Mvalve*sqrt((gma_I*32.174)/(Fl_IcombAir.Rt*778.16))*(1. + (gma_I -
1.)/2.*Mvalve**2)**((gma_I + 1.)/(2.*(1. - gma_I)));
        Wvalve = (Fl_IcombAir.Pt/sqrt(Fl_IcombAir.Tt))*(Atube*144.*ARvalve)*MFP;



// ------------------------------------------------------
// On - Design : Calculate bypass ratio
// ------------------------------------------------------
// *** input Variables : dTube, lTube, n_tubes, fillFrac, purgeFrac
// *** iterated Variables freq
// *** output Variables : iBPR
// *** local variables : WfillAir, WpurgeAir, WpureAir, WtotAir, Wbypass, WpurgeAir, Wvit
// *** Flow Stations : Fl_IcombAir, Fl_Icomb, Fl_Iprg, Fl_Vit

// ---- Calculate the split and partition of flow ----------
// amount of air that will be mixed with fuel - 1 tube
        mFillAir = (Vtube*rhoVit*fillFrac)/(1.+ FAR);

// amount of air that will purge during each cycle - 1 tube
        mPurgeAir = Vtube*Fl_IcombAir.rhos*purgeFrac;

// total air per cycle flowing though one tube
        mPureAir = mFillAir + mPurgeAir;
```

```
// -------------------------------------------------------
// Timing - calculate frequency
// -------------------------------------------------------
// *** input Variables : DDT, tValve, lTube, ff, pf, tCycle
// *** iterated Variables : uCJ, PcqPi
// *** output Variables tCycle, tauValveOpen, freq
// *** local variables : tDetonation, tDetProp, tBlowdown, tPurge, tFill
// -------------------------------------------------------

// ---------------- Detonation time ------------------
// DetProp time is relatively independent of fill fraction
        tDetProp = lTube/(uCJ*12);

// DDT is input , tDetonationPropogation calcd
// ( may need to iterate )
        tDetonation = DDT + tDetProp;

// -------------- Blowdown time ----------------------
// assume choked flow at tube exit and calculateon
// blowdown based draw - down time of a pressurized
// tank calculated on pressure differential
        gam_s = Fl_IcombAir.gams; // larger gamma is more conservative
        gmm_fc = ((gam_s + 1)/2)**(-(gam_s + 1)/(2*(gam_s - 1)));

// #### tBlowdown : Use ~1/2 calcd pressure ( to match experimental data )
// we 'll use CJ det wave velocity as the speed of sound in the gas
```

```
// since a cannot be directly calc 'd
// note tBlowdown is proportional to tube length
// tauBlDn is proportional to tube length
        tBlowdown = (log(0.4*PcqPi)/gmm_fc)*(lTube/uCJ);

// ------------- Fill and Purge time -----------------
// Use the choked flow at valve inlet and the mass flow rate as
// calculated outside the loop to calculate fill time (m/ mdot)
        tPurge = tValve + (mPurgeAir/Wvalve); // seconds
        tFill = tValve + (mFillAir/Wvalve); // seconds

// Improvement could be made by calculating vitiated air velocity ...
// ----------- Cycle Time output calculation ------------
        tCycle = tDetonation + tBlowdown + tPurge + tFill;
        tauValveOpen = (tPurge + tFill)/tCycle;
        freq = 1./tCycle;


// ---- Calculate the transient pressure ----
        int tTot, tC, tNew, xTot, xC, xNew;
        real timeCycle, pwall, func, dB, fn2, tNo, Tt3, para, xrf, xir, fn3, rhox, hx;
        real gam1, gam2, p2, a2, u2, dA1, dA2, p3, tref, xref, tplateau, px, ax, xPr, f2,
pDecay;
        real rho2, rho3, T2, T3, dCJ, p, fn, Tx, Icycle, Isp, IcyclePositive, IspPositive;
        real Ispf, IspfPositive, pex, rhoex, xPlat;
```

```
	if(time >= timeCycle) {
		tTot = (1e6*time);
		tC = (1e6*tCycle);
		tNew = tTot%tC;
		timeCycle = tNew/1e6;
	}
	else {
		timeCycle = time;
	}

	lTube = lTube/12; // change lTube to feet
	tCJ = tDetonation;
	gam1 = Fl_Icomb.gamt;
	gam2 = Fl_Ocomb.gamt;
	dCJ = lTube/tCJ;
	dA1 =
((gam1*MCJ2+gam2)/(2*gam2))*((((gam1*MCJ2+gam2)*(gam2+1))/((gam1*MCJ2+1)*(2*
gam2)))**((gam2+1)/(gam2-1)));
	dA2 = 2*((gam1*MCJ2)/(gam1*MCJ2+gam2));
	dB = 2*(((gam1*MCJ2+gam2)/(gam1*MCJ2+1))*((gam2+1)/(2*gam2)))**(-
(gam2+1)/(2*(gam2-1)));

// von Neumann spike conditions
	real pVN = 1+(((2*gam1)/(gam1+1))*(MCJ2-1));
	real rhoVN = ((gam1+1)*MCJ2)/(2+((gam1-1)*MCJ2));
	real TVN = pVN/rhoVN;
```

```
        real RVN =
(pVN*Fl_Icomb.Pt/(rhoVN*Fl_Icomb.rhot*TVN*Fl_Icomb.Tt))*173.700943; //VN Gas
constant in (ft*lbf)/(lbm*R)

// Calculate fluid properties behind detonation wave
        p2 = pVN*Fl_Icomb.Pt;
        a2 = ((gam1*MCJ2+1)/(gam1*MCJ2))*(gam2/(gam2+1))*dCJ;
        u2 = (((gam1*MCJ2-gam2)/(gam1*MCJ2))*(1/(gam2+1)))*dCJ;
        rho2 = rhoVN*Fl_Icomb.rhot;
        T2 = (144*p2)/(rho2*RVN);
        x2 = dCJ*timeCycle;

// Calculate fluid properties behind rarefaction wave
        x3 = x2/2;
        a3 = dCJ/2;
        p3 = (gam1/(2*gam2))*(((gam2+1)/(2*gam2))**((gam2+1)/(gam2-
1)))*MCJ2*Fl_Icomb.Pt;
        rho3 = 2*(((gam2+1)/(2*gam2))**((gam2+1)/(gam2-1)))*Fl_Icomb.rhot;
        T3 = (p3*144)/(rho3*Fl_Ocomb.Rt*778.17);

        if(timeCycle == 0) {
                px = Fl_Icomb.Pt;
                Tx = Fl_Icomb.Tt;
        }
```

```
        tref = tCJ*((((gam1*MCJ2+gam2)*(gam2+1))/((gam1*MCJ2+1)*(2*gam2)))**(-
(gam2+1)/(2*(gam2-1))));
        xref =
lTube*((1/dA2)*((((gam1*MCJ2+gam2)*(gam2+1))/((gam1*MCJ2+1)*(2*gam2)))**(-
(gam2+1)/(2*(gam2-1)))));
        tplateau = (tref + xref/a3);
        tExhaust = (dA2*(calcDiscrete(dA1,gam2)-1) + dB)*tCJ;
        tNo = (dB + dA2*(((2/(gam2+1))**(-(gam2+1)/(2*(gam2-1))))-1))*tCJ;
        para = timeCycle - tplateau + lTube/a3;

// fn2 = aw/a3
        fn2 = (0.6066*exp(-2.991*(a3/lTube)*(timeCycle-tplateau))+((1-0.6066)*exp(-
0.5014*(a3/lTube)*(timeCycle-tplateau))))**((gam2-1)/(2*gam2));

// fn = pw/p3
        //fn = (a3*(timeCycle-tNo)/lTube)+1;
        //pDecay = calcDiscrete(fn, gam2);
        pDecay = 0.6066*exp(-2.991*(a3/lTube)*(timeCycle-tNo))+((1-0.6066)*exp(-
0.5014*(a3/lTube)*(timeCycle-tNo)));

// Reflection from open end of tube
        xrf = ((dCJ*timeCycle)/(gam2-
1))*(((gam2*(gam1*MCJ2+1)/(gam1*MCJ2))*(tCJ/timeCycle)**((2*(gam2-1))/(gam2+1)))-
((gam1*MCJ2+gam2)/(gam1*MCJ2)));

// Reflection from thrust wall
```

```
        xir = lTube - ((gam2+1)/(gam2-1))*a3*para*(((para*a3/lTube)**(-2*(gam2-
1)/(gam2+1))) - (2/(gam2+1)));

// Undisturbed air ahead of detonation wave
        if((xval > x2) && (xval > x3)) {
                px = Fl_Icomb.Pt;
                Tx = Fl_Icomb.Tt;
                region = "Undisturbed";
        }

// Between detonation wave and rarefaction wave
        else if((xval > x3) && (xval <= x2)) {
                xPlat = xval/x2;
                px = p2*(((1/gam2)+(((gam2-1)/gam2)*(xval/x2)))**((2*gam2)/(gam2-1)));
                rhox = rho2*(((1/gam2)+(((gam2-1)/gam2)*(xval/x2)))**(2/(gam2-1)));
                u3 = u2 - ((2/(gam2+1))*((x2-xval)/timeCycle));
                a3 = a2 - (((gam2-1)/(gam2+1))*((x2-xval)/timeCycle));
                Tx = TCJ*((px/p3)**((gam2-1)/gam2));
                region = "Shock-Rarefaction";
        }

// Plateau pressure
        else if((xrf >= xval) && (x3 >= xval)) {
                px = (gam1/((gam2**((2*gam2)/(gam2-1)))*(gam2+1)))*MCJ2*pCJ;
                rhox = ((gam2+1)/(gam2**((gam2+1)/(gam2-1))))*rhoCJ;
                Tx = (144*px)/(rhox*Fl_Ocomb.Rt*778.17);
```

```
                region = "Plateau";
        }

// Rarefaction wave propogating toward thrust wall
        else if((xrf < xval) && (xrf >= 0) && (x2 > lTube)) {
                px = p2*(((1/gam2)+(((gam2-1)/gam2)*(0.5)))**((2*gam2)/(gam2-1)));
                rhox = rho2*(((1/gam2)+(((gam2-1)/gam2)*(0.5)))**(2/(gam2-1)));
                Tx = TCJ*((px/p3)**((gam2-1)/gam2));
                region = "First Reflection";
        }

// Rarefaction wave propogating toward open end
        else if((xir >= xval) && (xrf < 0)) {
                px = 2.13480013*pDecay*p3*(1-175*(timeCycle-tNo));
                rhox = pDecay*rho3; //Fl_Icomb.rhot;
                rhoex = ((gam2+1)/(gam2**((gam2+1)/(gam2-1))))*Fl_Icomb.rhot;
                real uex = dCJ/(gam2+1);
                real t1 = 2*lTube/dCJ;
                real t2 = 4*lTube/dCJ;
                real t3 = ((rho3*lTube)/(rhoex*uex))+t1;
                real pUTA = (p3-pCJ)*(1-((timeCycle-t2)/(t3-t2))) + pCJ;
                Tx = TCJ*((px/p3)**((gam2-1)/gam2));
                region = "Second Reflection";
        }

// Exhaust
```

```
        if(timeCycle >= tExhaust) {
                px = Fl_Icomb.Pt;
                Tx = Fl_Icomb.Tt;
                region = "Exhaust";
        }

        lTube = lTube*12; // change lTube back to inches

// ---------- Set total mass flow through tubes ----------
        WtotAir = mPureAir*freq;



// steady - state flow rate into tubes
// conservation of mass check
if(WtotAir > Fl_I.W) {
        fillFrac = fillFrac*(Fl_I.W/WtotAir);
        purgeFrac = purgeFrac*(Fl_I.W/WtotAir);
        mFillAir = Vtube*(rhoVit*fillFrac)/(1.+ FAR);

// amount of air that will be mixed with fuel - 1 tube
        mPurgeAir = Vtube*(Fl_IcombAir.rhos*purgeFrac);

// amount of air that will purge during each cycle -1 tube
        mPureAir = mFillAir + mPurgeAir;
```

```
// total air per cycle flowing though one tube
        WtotAir = Fl_I.W;
        cout << "purgeFrac changed to: " << purgeFrac << endl;
        cout << "fillFrac changed to: " << fillFrac << endl << endl;
}

// ---------- Set iBPR ----------
        WtotAir = mPureAir*freq;
        Wbypass = Fl_I.W - WtotAir;


// steady - state flow rate sent to bypass
        iBPR = Wbypass/WtotAir;

// steady - state internal PDC bypass ratio
        iBPRdes = iBPR;

// --------- Set bypass exit flow SPLIT ----------
        Fl_O2.W = Wbypass;

// ----- Set purge and fill stations PARTITION -------
        Fl_Iprg.copyFlowStatic("Fl_IcombAir");

// copy flow for purge function
// ---------- PURGE AIR ----------
        Fl_Iprg.AphyDes = (Atube*144); //*n_tubes; // Set phys area
```

```
        Fl_Iprg.W = mPurgeAir*freq; //*n_tubes; // set m dot


// ---------- FILL AIR ----------
        Fl_Icomb.copyFlow("Fl_IcombAir");
        Fl_Icomb.AphyDes = Atube*144.*tauValveOpen; // *n_tubes

// Actual area is multiplied by tauVO to get equivalent
// area . - Fluid flows steadily through this area
        Fl_Icomb.W = mFillAir*freq; // *n_tubes


        Fl_Icomb.setTotal_hP(Fl_IcombAir.ht, Fl_IcombAir.Pt);



// ------------------------------------------------
// Burning
// ------------------------------------------------
// FAR was calculated prior to entering this
// point - so we just need to modify
// Wfuel based on changed Fl_Icomb.W
        Wfuel = (Fl_Icomb.W/(1. + FARin + WARin))*(FAR - FARin);
        Fu_I.Wfuel = Wfuel;
        calcRayleighLoss();



        Fl_Ocomb.setTotalTP(Tx, px);
        updateAvg(Tx, px, Fl_Ocomb.ht);
```

```
		if(Tlast < Tx) {
			Tmax = Tx;
		}
		if(Plast < px) {
			Pmax = px;
		}
		if(Hlast < Fl_Ocomb.ht) {
			Hmax = Fl_Ocomb.ht;
		}
		Tlast = Tx; Plast = px; Hlast = Fl_Ocomb.ht;
}

// ---------------------------------------------------
// Add split flows back to combusted flow
// ---------------------------------------------------
		Fl_Ocomb.add("Fl_Iprg"); // add purge flow in ( uncorrected )



// =====================================================
// Apply corrections to the flow for transition to ...
// steady state ( TTSS )
// =====================================================
// *** Local Variables : Snew, Pnew
// *** Input Variables : deltaS, TTSSeff, TTSSdPqP
// *** Flow stations : Fl_Ocomb, Fl_Vit
		real hnew, Pnew, Snew;
```

```
// ------ Calculate new Entropy and Pressure ----------
// eff = ( dht ) TTSF /( dht ) comb + 1.
// current h - ( h gained )*(1. - eff )

        hnew = Fl_Ocomb.ht - ((Fl_Ocomb.ht - Fl_Icomb.ht)*(1.0 - TTSSeff));
        Pnew = Fl_Ocomb.Pt*(1.0 - TTSSdPqP);

        Fl_O1.copyFlow("Fl_Ocomb");

// -------------------------------------------------------
// Thermal storage calculations
// -------------------------------------------------------
if(!S_Qhx.isEmpty()) {
        S_Qhx.execute();
}

real hout = Fl_O1.ht - (Qhx/Fl_O1.W);
Fl_O1.setTotal_hP(hout, Fl_O1.Pt);

// -------------------------------------------------------
// store the design value of FAR for use in guessing
// -------------------------------------------------------
if( switchDes == "DESIGN" ) {
        FARDes = FAR;
}
```

```
}

void setX(real x) {
	//cout << "Xval set to " << x*lTube << " feet" << endl;
	xval = x*lTube/12;
}

void setTime(real t, real tInt) {
	//cout << "Time set to " << t << " seconds" << endl;
	time = t;
	tInterval = tInt;
}

void updateAvg(real temp, real pressure, real enthalpy) {
/*
	cout << "temp = " << temp << endl;
	cout << "pressure = " << pressure << endl;
	cout << "enthalpy = " << enthalpy << endl;
	cout << "Tlast = " << Tlast << endl;
	cout << "Plast = " << Plast << endl;
	cout << "Hlast = " << Hlast << endl;
*/
	Taverage = Taverage + tInterval*(Tlast+temp)/2;
	Paverage = Paverage + tInterval*(Plast+pressure)/2;
	Haverage = Haverage + tInterval*(Hlast+enthalpy)/2;
/*
```

```
        cout << "Tnew = " << Taverage << endl;
        cout << "Pnew = " << Paverage << endl;
        cout << "Hnew = " << Haverage << endl;
        cout << endl;
*/
}

void setCJ(real M, real p, real T, real u, real rho, real gamma, real R) {
        MCJ = M;
        pCJ = p*Fl_Icomb.Pt;
        TCJ = T*Fl_Icomb.Tt;
        uCJ = u;
        rhoCJ = rho*Fl_Icomb.rhot;
        gamCJ = gamma;
        RCJ = R;
/*
        cout << "MCJ = " << MCJ << endl;
        cout << "pCJ = " << pCJ << endl;
        cout << "TCJ = " << TCJ << endl;
        cout << "uCJ = " << uCJ << endl;
        cout << "rhoCJ = " << rhoCJ << endl;
        cout << "gamCJ = " << gamCJ << endl;
        cout << "RCJ = " << RCJ << endl << endl;
*/
}
```

```
real getT() {
	return Taverage;
}

real getP() {
	return Paverage;
}

real getH() {
	return Haverage;
}

real getTstat() {
	return Fl_Icomb.Tt;
}

real getPstat() {
	return Fl_Icomb.Pt;
}

real getHstat() {
	return Fl_Icomb.ht;
}

real getPMax() {
	return Pmax;
```

```
}

real getTMax() {
	return Tmax;
}

real getHMax() {
	return Hmax;
}

void setH(real in) {
	Hinput = in;
}

void setP(real in) {
	Pinput = in;
}

real calcDiscrete(real parameter, real gamma) {
	real na, nb, gamA, gamB, X, np, gamIn, fa, fb, para;
	gamIn = gamma;
	para = parameter;

	if((gamIn < (5/3)) && (gamIn >= (7/5))) {
		na = 1; nb = 2;
	}
```

```
        else if((gamIn < (7/5)) && (gamIn >= (9/7))) {
                na = 2; nb = 3;
        }
        else if((gamIn < (9/7)) && (gamIn >= (11/9))) {
                na = 3; nb = 4;
        }
        else if((gamIn < (11/9)) && (gamIn >= (15/13))) {
                na = 4; nb = 6;
        }
        else if((gamIn < (15/13)) && (gamIn >= (23/21))) {
                na = 6; nb = 10;
        }
        else {
                na = 6; nb = 10;
        }

        if(na == 1) {
                fa = 0.5*(para**0.2) + 0.5*(para**0.6);
        }
        else if(na == 2) {
                fa = 0.375*(para**0.14286) + 0.25*(para**0.42857) +
0.375*(para**0.714286);
        }
        else if(na == 3) {
                fa = 0.3125*(para**0.11111) + 0.1875*(para**0.33333) +
0.1875*(para**0.55555) + 0.3125*(para**0.77777);
```

```
        }
        else if(na == 4) {
                fa = 0.273438*(para**0.090909) + 0.15625*(para**0.272727) +
0.140625*(para**0.454545) + 0.15625*(para**0.636363) + 0.273438*(para**0.818181);
        }
        else if(na == 6) {
                fa = 0.22471*(para**(1/15)) + 0.123047*(para**(3/15)) +
0.10214*(para**(5/15)) + 0.097656*(para**(7/15)) + 0.10214*(para**(9/15)) +
0.123047*(para**(11/15)) + 0.224708*(para**(13/15));
        }
        else if(na == 10) {
                fa = 0.176197*(para**(1/23)) + 0.092734*(para**(3/23)) +
0.073643*(para**(5/23)) + 0.065460*(para**(7/23)) + 0.061683*(para**(9/23)) +
0.060562*(para**(11/23)) + 0.061683*(para**(13/23)) + 0.065460*(para**(15/23)) +
0.073643*(para**(17/23)) + 0.092734*(para**(19/23)) + 0.176197*(para**(21/23));
        }

        if(nb == 1) {
                fb = 0.5*(para**0.2) + 0.5*(para**0.6);
        }
        else if(nb == 2) {
                fb = 0.375*(para**0.14286) + 0.25*(para**0.42857) +
0.375*(para**0.714286);
        }
        else if(nb == 3) {
```

```
                fb = 0.3125*(para**0.11111) + 0.1875*(para**0.33333) +
0.1875*(para**0.55555) + 0.3125*(para**0.77777);
        }
        else if(nb == 4) {
                fb = 0.273438*(para**0.090909) + 0.15625*(para**0.272727) +
0.140625*(para**0.454545) + 0.15625*(para**0.636363) + 0.273438*(para**0.818181);
        }
        else if(nb == 6) {
                fb = 0.22471*(para**(1/15)) + 0.123047*(para**(3/15)) +
0.10214*(para**(5/15)) + 0.097656*(para**(7/15)) + 0.10214*(para**(9/15)) +
0.123047*(para**(11/15)) + 0.224708*(para**(13/15));
        }
        else if(nb == 10) {
                fb = 0.176197*(para**(1/23)) + 0.092734*(para**(3/23)) +
0.073643*(para**(5/23)) + 0.065460*(para**(7/23)) + 0.061683*(para**(9/23)) +
0.060562*(para**(11/23)) + 0.061683*(para**(13/23)) + 0.065460*(para**(15/23)) +
0.073643*(para**(17/23)) + 0.092734*(para**(19/23)) + 0.176197*(para**(21/23));
        }

        gamA = (2*na + 3)/(2*na + 1);
        gamB = (2*nb + 3)/(2*nb + 1);
        X = (gamIn-gamA)/(gamB-gamA);
        np = (3-gamIn)/(2*(gamIn-1));
        out = (1-X)*fa + X*fb;
/*
        cout << "Para = " << para << endl;
```

```
        cout << "f_a(Z) = " << fa << endl;
        cout << "f_b(Z) = " << fb << endl;
        cout << "gamIn = " << gamIn << endl;
        cout << "gamA = " << gamA << endl;
        cout << "gamB = " << gamB << endl;
        cout << "X = " << X << endl;
        cout << "np = " << np << endl;
        cout << "pw/p3 = " << out << endl;
        cout << endl;
*/

        return out;
}



// ------------------------------------------------------------
// register the appropriate errors at build time
// ------------------------------------------------------------
void VCinit()
{
        ESOregCreate(1023901, 8 , "", TRUE, FALSE, TRUE); // provisional
        ESOregCreate(1093901, 8 , "", TRUE, FALSE, TRUE); // provisional
}
}
#endif
```

```
close all; clear all; clc;


% Mach no.
M = [1 2 2.5 3 3.5 4 4.5 4.8288];
% P ratio
PC = [1 4.528 7.198 10.483 14.390 18.926 24.105 27.867];
% T ratio
TC = [1 1.644 2.043 2.506 3.031 3.613 4.242 4.677];
% Density ratio
pC = [1.0003 2.7540 3.5235 4.1833 4.7475 5.2392 5.6816 5.9562];


for i=1:length(M)
    P(i) = 1+((2.8/2.4)*((M(i)^2)-1));
    d(1,i) = 100*abs(1-(P(i)/PC(i)));
    p(i) = (2.4*(M(i)^2))/(2+(0.4*(M(i)^2)));
    d(2,i) = 100*abs(1-(p(i)/pC(i)));
    T(i) = P(i)/p(i);
    d(3,i) = 100*abs(1-(T(i)/TC(i)));
end


figure(1);
hold on; grid on;
xlabel('Mach No.')
ylabel('Pressure Ratio')
plot(M,PC,M,P)
```

```
legend('CEA Result','CPG Result','Location','Best')
figure(2);
hold on; grid on;
xlabel('Mach No.')
ylabel('Temperature Ratio')
plot(M,TC,M,T)
legend('CEA Result','CPG Result','Location','Best')
figure(3);
hold on; grid on;
xlabel('Mach No.')
ylabel('Density Ratio')
plot(M,pC,M,p)
legend('CEA Result','CPG Result','Location','Best')
disp('Pressure % Error')
disp(d(1,:))
disp('Density % Error')
disp(d(2,:))
disp('Temp. % Error')
disp(d(3,:))
```

References

1. Glassman, Irvin., Yetter, Richard. A., *Combustion*. Academic Press, New York, NY, fourth edition, 2008. ISBN 978-0-12-088573-2

2. Mattingly, Jack D. and von Ohain, Hans., *Elements of Propulsion: Gas Turbines and Rockets.* American Institute of Aeronautics and Astronautics (AIAA), Reston, Virginia, 2006. ISBN 1563477793

3. Andrus, Ionio Q., *Comparative Analysis of a High Bypass Turbofan Using a Pulsed Detonation Combustor.* MS thesis, AFIT/GAE/ENY/07-M02. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2007.

4. Thorn, Caitlin R., *Off-Design Analysis of a High Bypass Turbofan Using a Pulsed Detonation Combustor*. MS thesis, AFIT/GAE/ENY. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2010.

5. Kumar, Sivarai Amith., *Parameteric and Performance Analysis of a Hybrid Pulse Detonation/Turbofan Engine*. MS thesis, University of Texas at Arlington, Department of Mechanical & Aerospace Engineering, May 2011.

6. Endo, Takuma and Fujiwara, Toshi, *Analytical Estimation of Performance Parameters of an Ideal Pulse Detonation Engine*. Trans. Japan Soc. Aero. Space Sci., Vol. 45, No. 150, pp. 249-254, 2003.

7. Endo, Takuma, Kasahara, Jiro and Fujiwara, Toshi. *Pressure History at the Thrust Wall of a Simplified Pulse Detonation Engine.* AIAA Journal Vol. 42, No. 9, September 2004.

8. Wilson, Donald R., Lu, Frank K., Kim, JunHyun and Hekiri, Haider., *Analysis of an Ejector-Augmented Pulse Detonation Rocket.* 46th AIAA Aerospace Sciences Meeting and Exhibit, 7-10 January 2008, Reno, Nevada. AIAA 2008-114.

9. Browne, S., Ziegler, J. and Shepherd, J.E., *Numerical Solution Methods for Shock and Detonation Jump Conditions.* California Institute of Technology, GALCIT Report FM2006.006, July 2004-Revised August 29, 2008.

10. NASA Chemical Equilibrium with Applications (CEA), www.grc.nasa.gov/WWW/CEAWeb/, Last Updated: 02/03/2010. Curator and NASA Official: Dr. Michael J. Zehe.

11. NPSS Consortium, *NPSS Consortium,* http://www.npssconsortium.org/, Last Retrieved 11/13/2012.

Biographical Information

Anthony Hasler graduated from Texas Christian University in 2010 with a degree in Mechanical Engineering. From there he went to the University of Texas in Arlington to get a Master's degree in Aerospace Engineering (graduated in December of 2012). His research interests lie in propulsion, aerodynamics, high temperature gas-dynamics and numerical methods for solving differential equations. Projects that he has worked on include computational fluid dynamics in low and high Reynolds number situations with high temperature gas dynamics, a KIVA-based analysis of gasoline and ethanol in a four-stroke internal combustion engine, an NPSS analysis of a turbojet engine including transient effects and, most significantly, a transient analysis of a mixed-flow turbofan with detonation combustors acting as duct burners (and comparison with standard combustion duct burners). His future plans are to continue research in the field of Aerospace Engineering with the hopes of one day becoming an astronaut.