MAJOR PROJECT # 2

Shaun Harris
Department of Mechanical and Aerospace Engineering
Utah State University
Email: shaun.r.harris@gmail.com

ABSTRACT

The solid motor propulsion rates for a small "Pike" missile was simulated. The chamber pressure, regression rate, massflow, choked massflow, mass depletion, thrust profile, and I_{sp} was analyzed for cylindrical port burning and for Bates grain burning.

CONTENTS

NOMENCLATURE

L_0, L_{port} Total length of propellant
D_0 Outer diameter of propellant
d_0 Inner diameter of propellant
$\rho, \rho_{propellant}$ density of propellant
A^* Throat area
A_{exit} Nozzle exit area
M_W Molecular weight
T_0 Flame temperature
$a, n, M_{crit,k}$ properties for calculating $\dot{r}$
$\dot{r}$ Linear change of propellant per time in direction perpendicular to surface
$\dot{P}_0$ Change in chamber pressure per time
r Inner radius of solid propellant
P_0 Chamber pressure
t time in seconds

1 INTRODUCTION

A small missle was simulated using the erosive and bates grain burning methods. Two simulations were run and compared. The first was a cylindrical port simulation, where erosive burning was used. The second simulation used three individual blocks of bates grains where the ends were not burn inhibited. It also did not use erosive burning. Many of the properties were shared. Eq. 1 shows the shared properties. The first simulation had a specific $k = 0.2$ and the second simulation set $k = 0$ to neglect the erosive burning. Eq. 2 shows the relevant equations for the first simulation and Eq. 3 shows the relevant equations for the second simulation.

$$
\begin{aligned}
L_0 &= 35cm \\
D_0 &= 6.6cm \\
d_0 &= 3cm \\
\rho &= 1260\frac{kg}{m^3} \\
A^* &= 1.887cm^2 \\
\frac{A_{exit}}{A^*} &= 4.0 \\
\theta_{exit} &= 20deg \\
\gamma &= 1.18 \\
M_W &= 23\frac{kg}{kg-mol} \\
T_0 &= 2900K \\
a &= 0.132\frac{cm}{s-kPa^n} \\
n &= 0.16 \\
M_{crit} &= 0.3
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
\dot{P}_0 &= \frac{A_{burn}\dot{r}}{V_c}\left(\rho_{propellant}R_gT_0 - P_0\right)P_0\sqrt{\gamma R_g T_0\left(\frac{2}{\gamma+1}\right)^{\frac{\gamma+1}{\gamma-1}}} \\
\dot{r} &= aP_0^n \\
P_0 &= P_{ambient} \\
r &= \frac{d_0}{2} \\
A_{burn} &= 2\pi r L_{port} \\
V_c &= \pi r^2 L_{port}
\end{aligned}
\tag{2}
$$

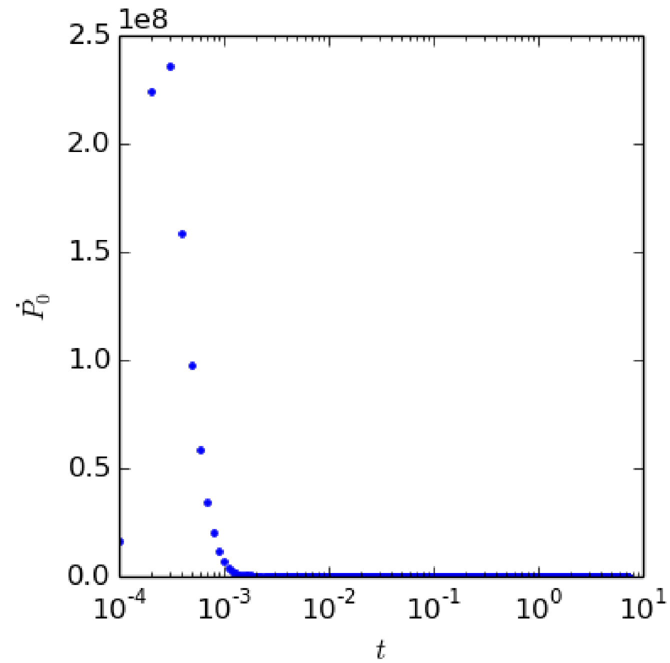
$$
\begin{aligned}
\dot{P}_0 &= \frac{A_{burn}\dot{r}}{V_c}\left(\rho_{propellant}R_gT_0 - P_0\right)P_0\sqrt{\gamma R_g T_0\left(\frac{2}{\gamma+1}\right)^{\frac{\gamma+1}{\gamma-1}}} \\
\dot{r} &= aP_0^n\left(\frac{1+k\frac{M_{port}}{M_{crit}}}{1+k}\right) \\
P_0 &= P_{ambient} \\
r &= \frac{d_0}{2} \\
A_{burn} &= N\pi\left[\frac{D_0^2-(d_0+2s)^2}{2} + (L_0-2s)(d_0+2s)\right] \\
V_c &= \frac{N\pi}{4}\left[(d_0+2s)^2(L_0-2s) + D_0^2 2s\right]
\end{aligned}
\tag{3}
$$

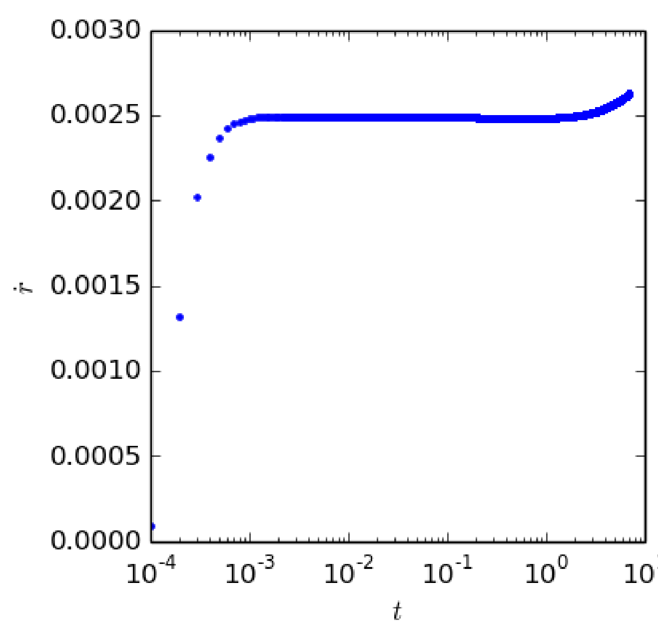
2 RESULTS

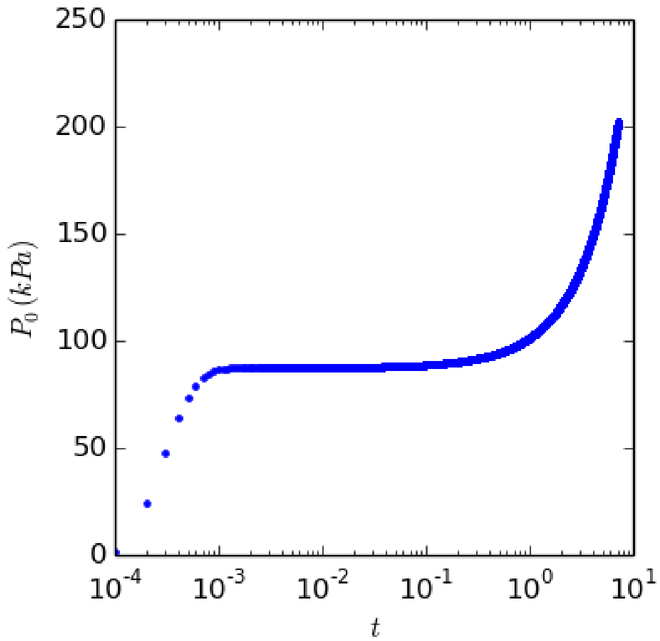
Each simulation had their own set of results. The subsections below will outline the results for each.

2.1 Part 1 Cylindrical Port with Erosive burning

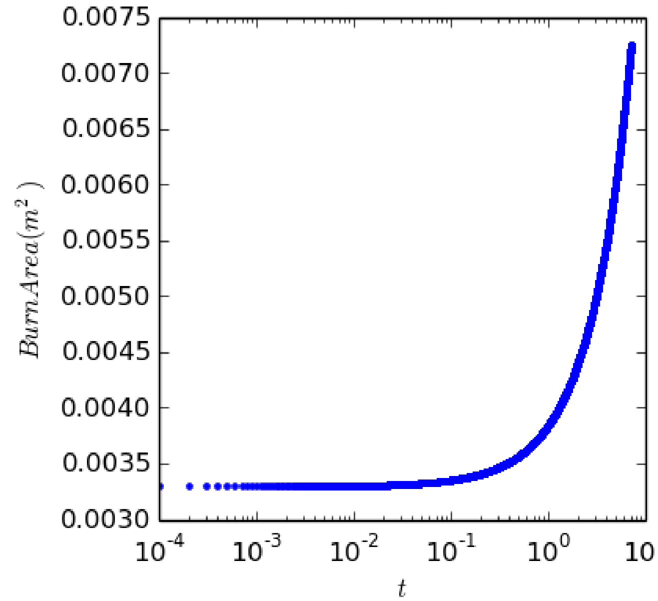
The following figures outline the results for each of the quantities of interest.



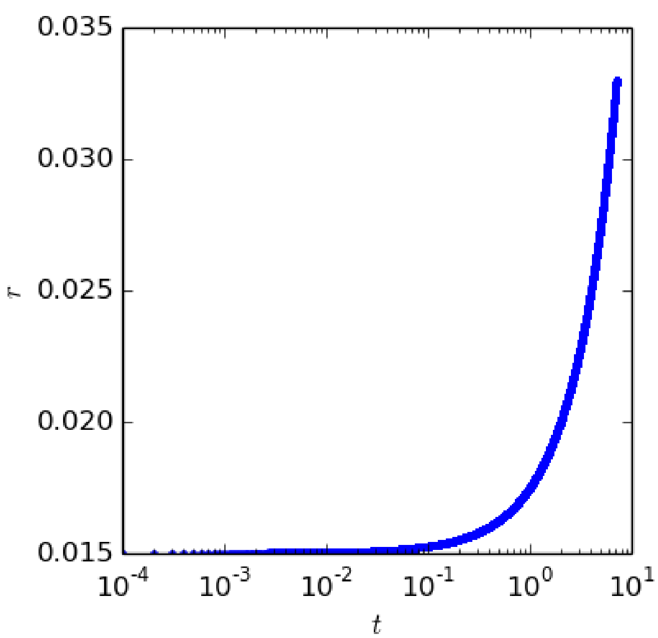




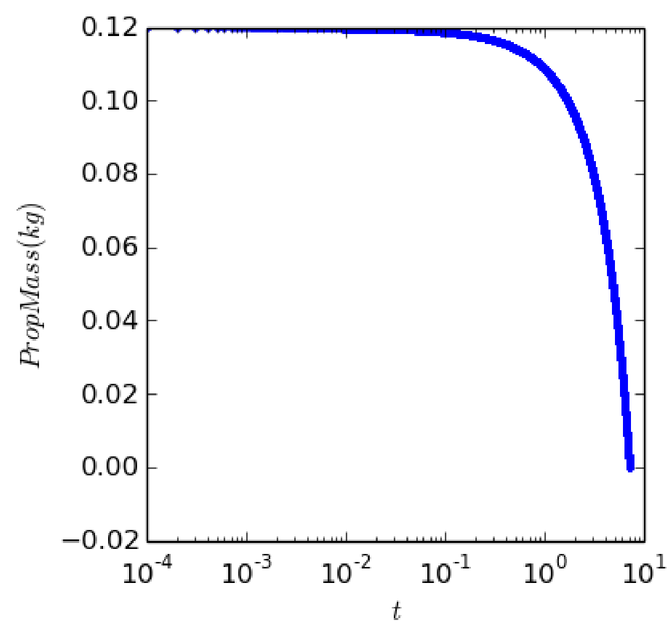
$P_0\,(kPa)$
t



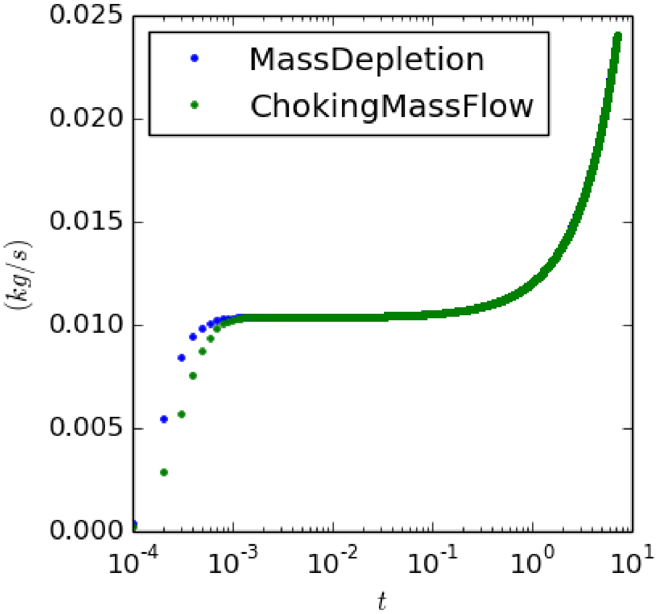
$BurnArea(m^2)$
t

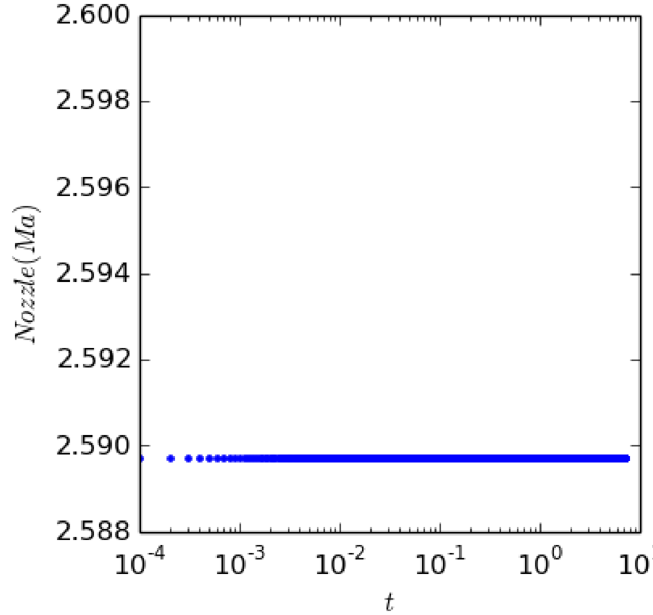


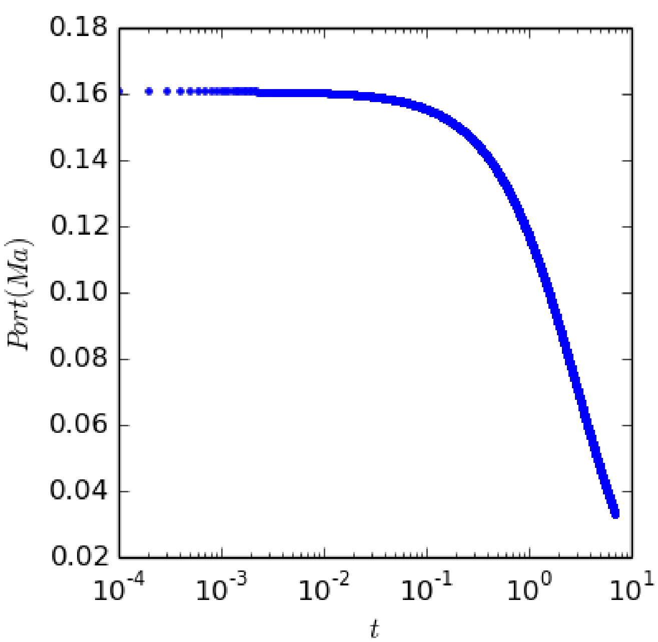
r
t

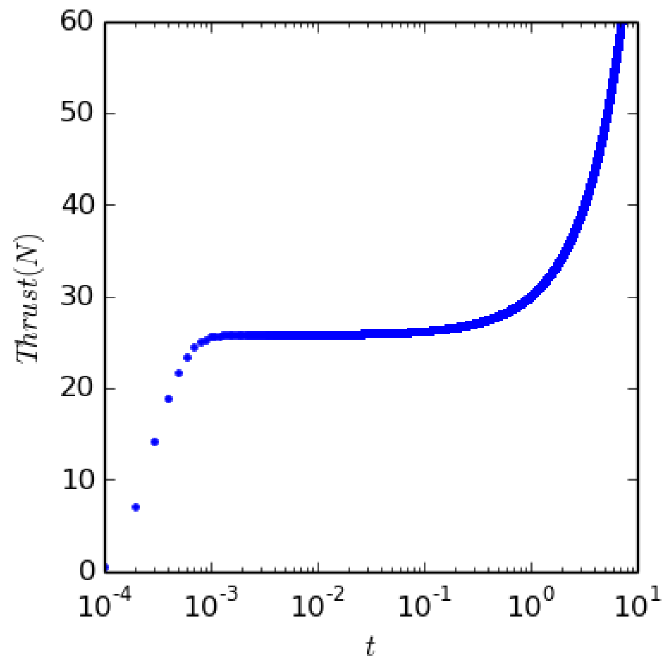


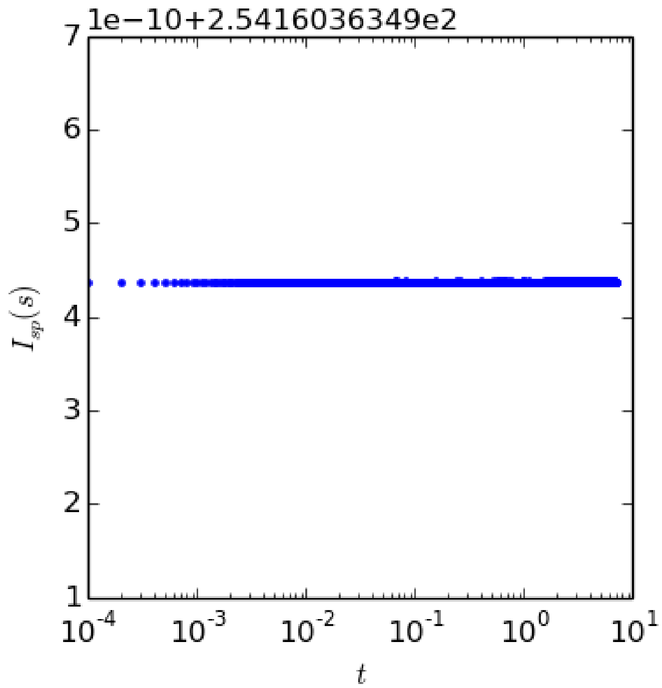
$PropMass(kg)$
t

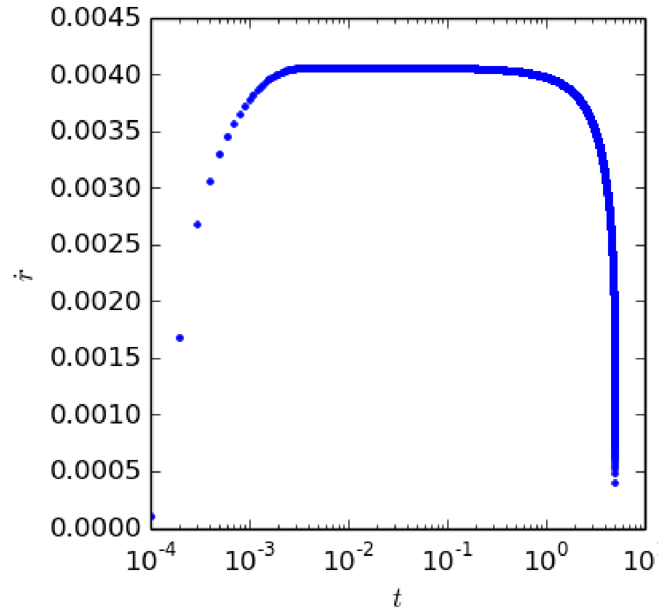

MassDepletion
ChokingMassFlow
(kg/s)
t
0.025
0.020
0.015
0.010
0.005
0.000
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}


$Nozzle(Ma)$
t
2.600
2.598
2.596
2.594
2.592
2.590
2.588
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}


$Port(Ma)$
t
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}


$Thrust(N)$
t
60
50
40
30
20
10
0
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}

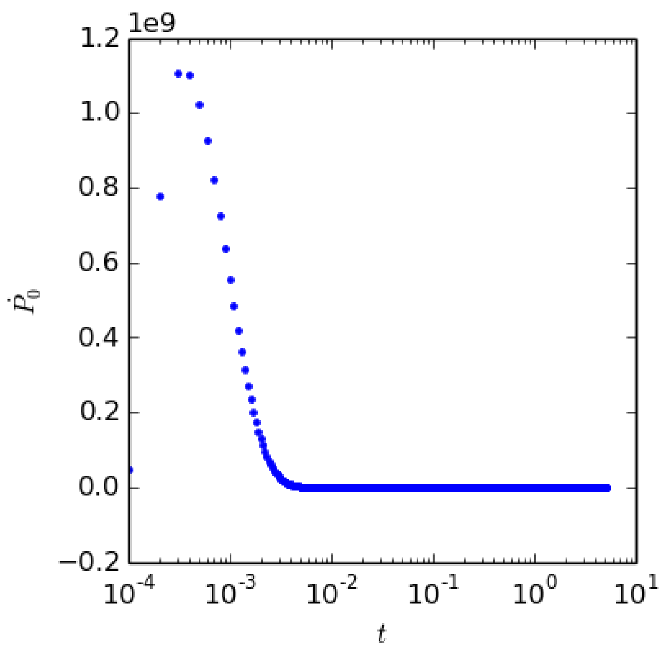


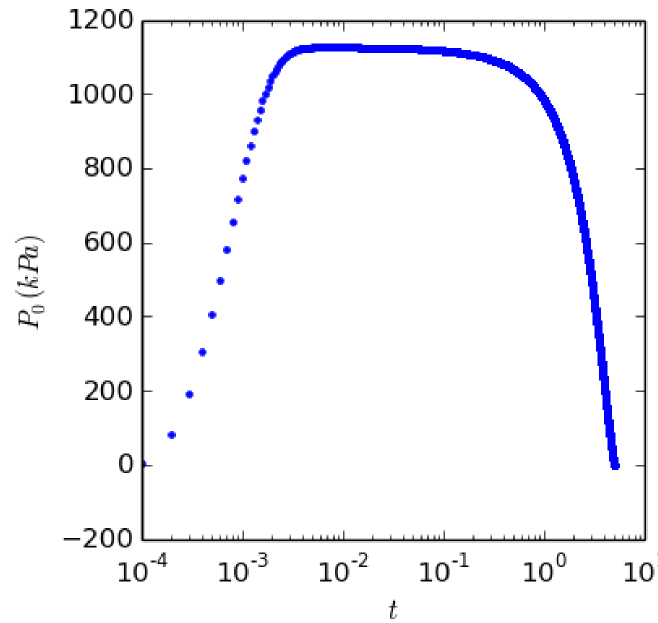


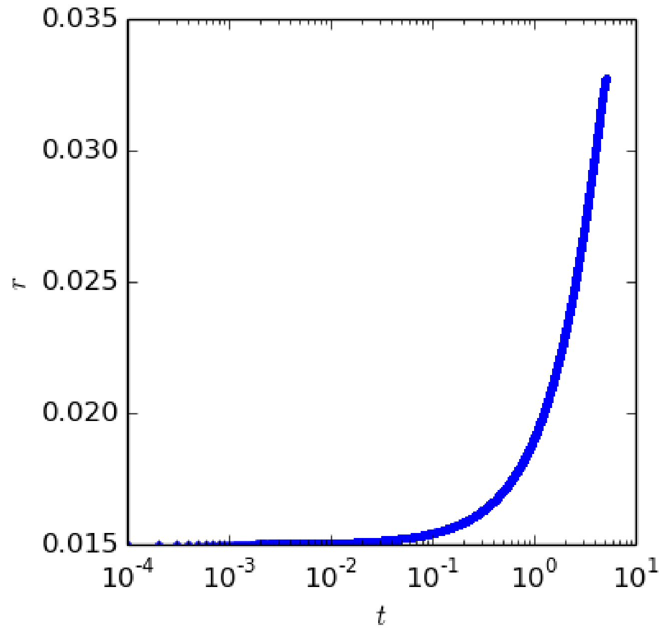
Where the effective mean $I_{sp} = 254.16s$.

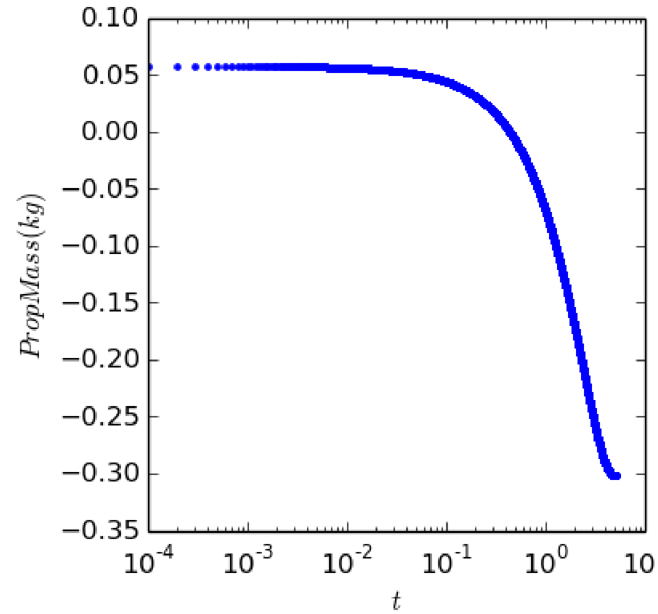
2.2 Part 2 Bates grain with non-erosive burning

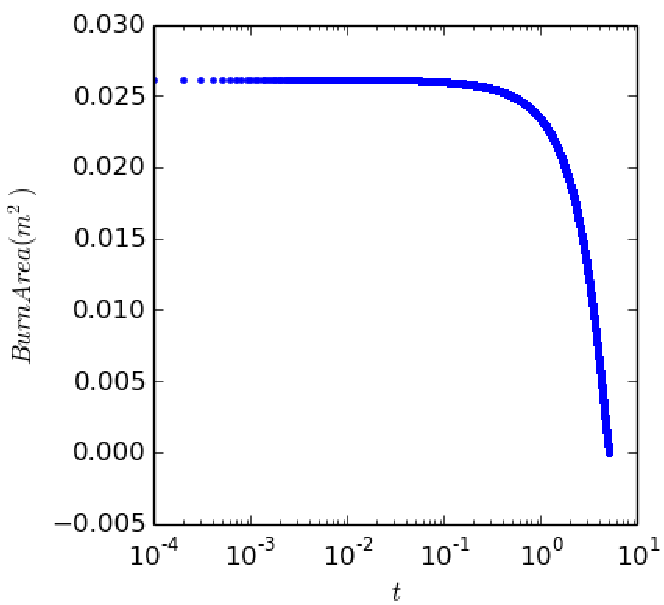
The following figures outline the results for each of the quantities of interest.

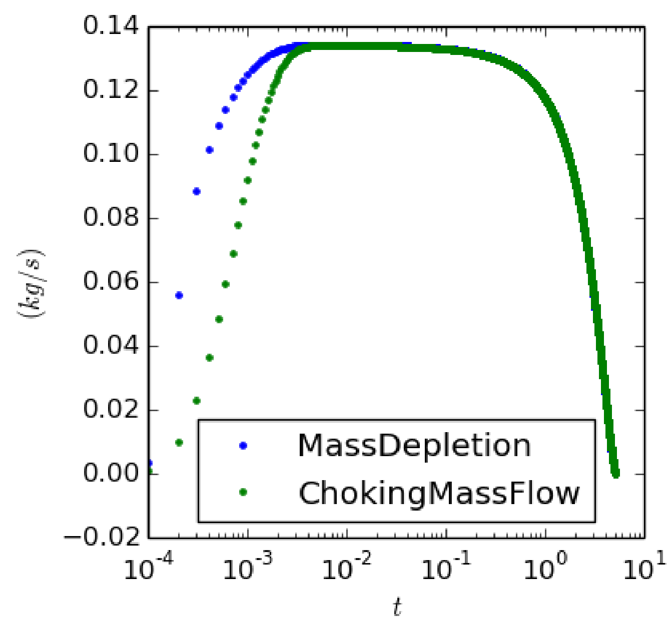


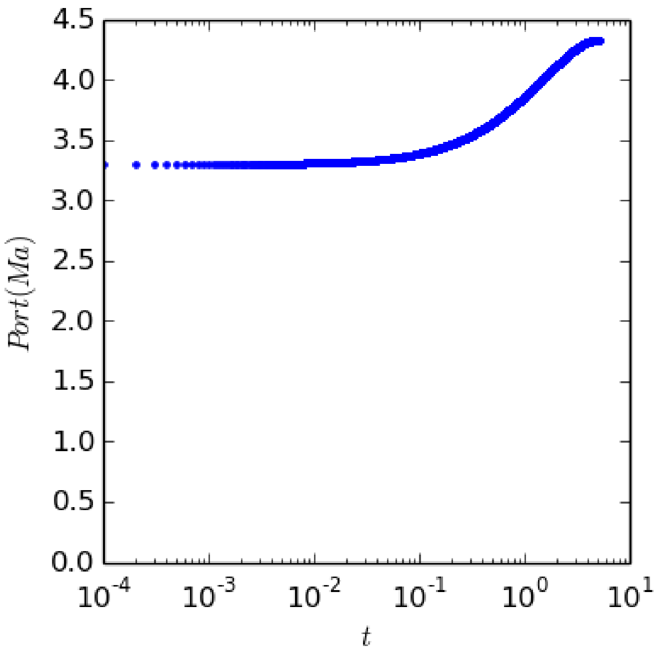


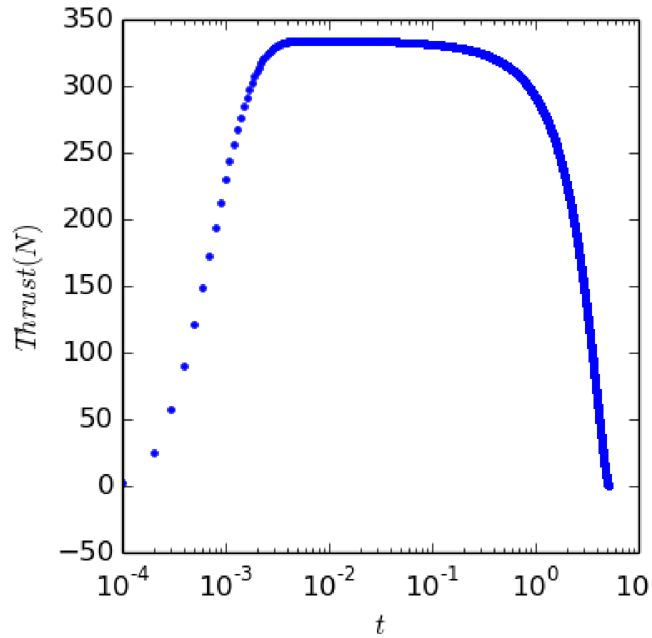

0.035
0.030
0.025
0.020
0.015
r
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}
t

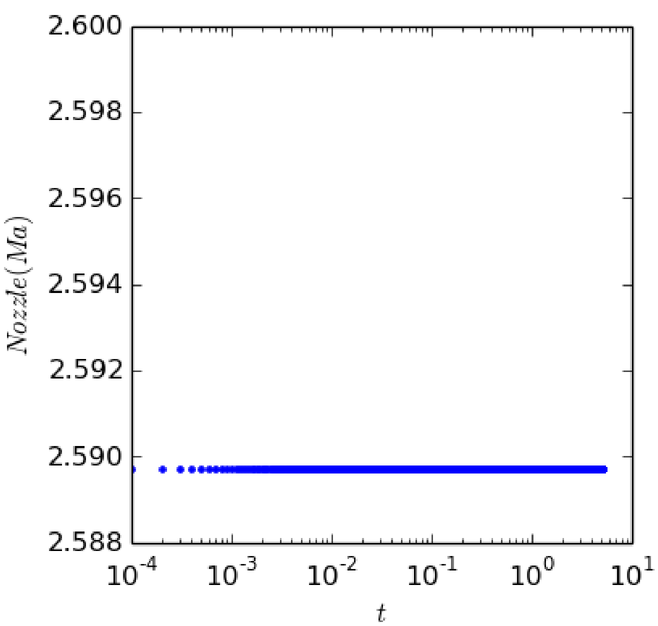

0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
−0.30
−0.35
PropMass(kg)
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}
t

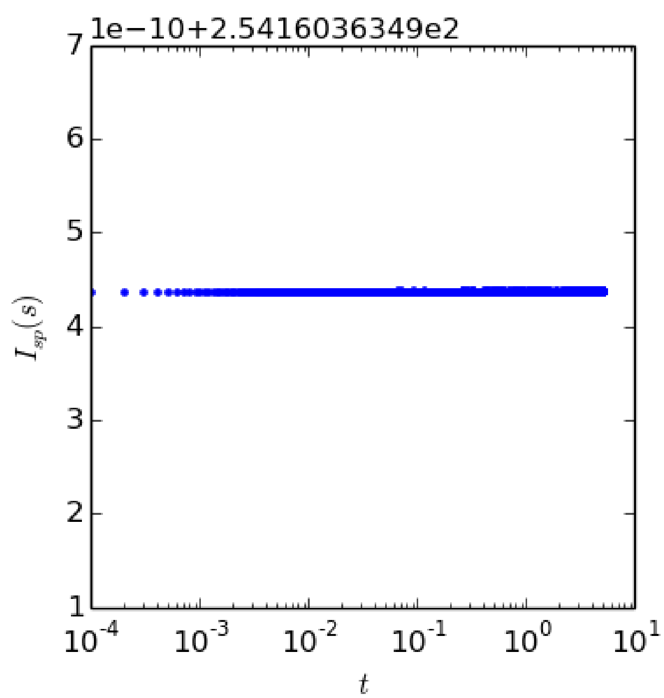

0.030
0.025
0.020
0.015
0.010
0.005
0.000
−0.005
BurnArea(m^2)
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}
t


0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
(kg/s)
MassDepletion
ChokingMassFlow
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1}
t









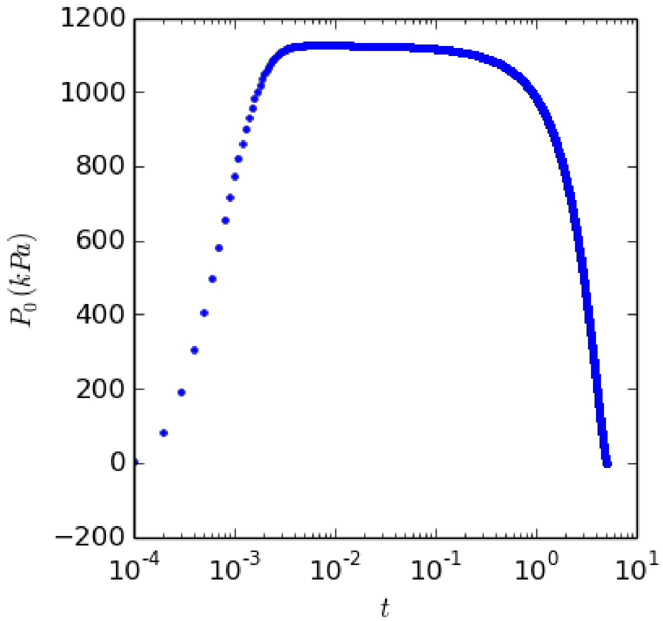
Where the effective mean $I_{sp} = 254.16s$.

2.3 Sensitivity and effect of flame temperature

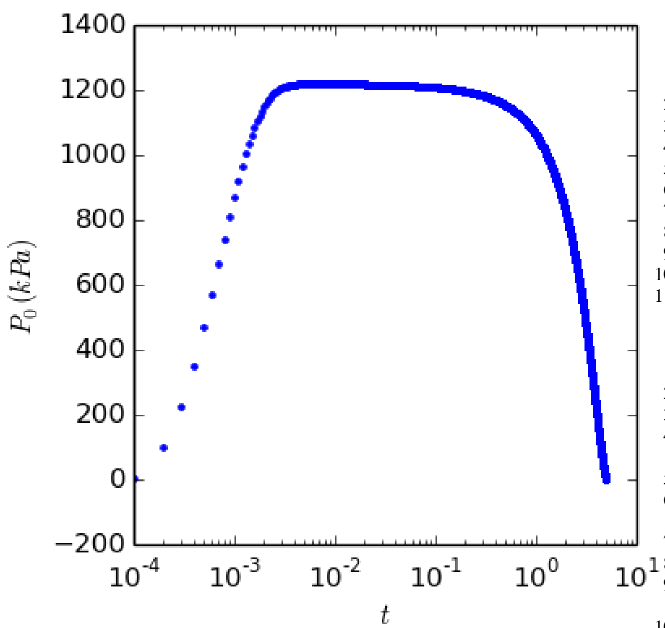
In addition to the above plots, analysis was run to see the sensitivity of the input parameters. It was found that small changes can lead to large changes in the output values. Thus, care must be taken to ensure correct and repeatable parameters are provided for rockets. For example, by changing $T_0 = 3300$

we can compare the two plots of P_0 vs time to see the change in the values. It is fairly drastic.

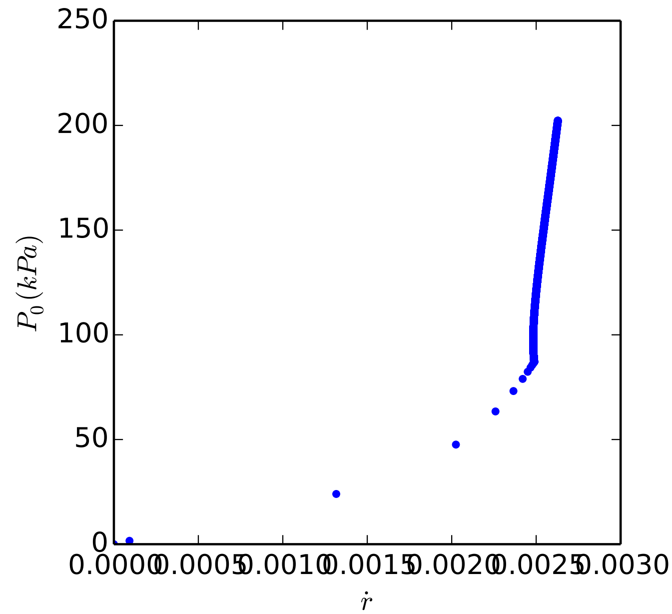
This is the original P_0 plot shown above in Part 2.



This is the value of P_0 with the increased flame temperature.



Additionally, the regression vs chamber pressure was shown in this next plot.



3 CONCLUSION

These calculations used compressible fluid flow as well as solid propellant simulations. We were able to show the many quantities of interest for the two situations.

A Appendix A: Code

Header files

```
#ifndef DEFS_H
#define DEFS_H
#include <iostream>
#include <vector>
#include <cmath>
#include <string>
#include <sstream>
#include <iomanip>
#include <fstream>
using namespace std;
#endif
```

```
// this is a newton solver for computing the area-Mach number
    relation
class Prandtl_Meyer{
public:
    // public values of gamma, A as a function of x, A at the
        throat, Mach number, the Newton iteration tolerance
    double gamma, A, A_star, M, tol;
    //public values for F = 0, dF/dM, and the number of
        iterations for the Newton solver
    double F, dFdM;
    int iter;
    //public values for P0 and T0 (stagnation pressure and
        temperature for an isentropic nozzle)
    //and local P and T (if an isentropic nozzle)
    double P0,T0,P,T;
    // non-isentropic values
    double P02;
    // x,yp,ym (positive and negative) values within the
        nozzle
```

```
    double x,yp,ym;
    // set a=1 for non-isentropic and a=0 for isentropic nozzle
    double isen;
    // ideal gas Rg property
    double Rg;
    // mass flow m_dot
    double m_dot;
    // oblique shock wave values of x,a,b,c and xnew and beta
    double xj,a,b,c,xnew,beta;
    // oblique shock wave density, pressure and temperature before and after
    double rho1,rho2,P2,T2;
    // M2 and theta and mu1 and mu2 of fan for Prandtl-Meyer equations (theta is reused for oblique shock waves as well)
    double M2,theta,mu1,mu2;
    //constructors
    Prandtl_Meyer(){
        gamma   =   1.18;
        A_star  =   0.0001887;
        A       =   4.*A_star;
        M       =   4.0; //initial assume it is supersonic desired region
        tol     =   0.00000001;
        iter    =   0;
        P0      =   24250000.;
        T0      =   2900.;
    }

    //set public values
    void    set_isentropic(int  a)  ;
        // set isen = 0 for isentropic relationships
        // set isen = 1 for non-isentropic relationships
        // set isen = 2 for Rayleigh Pitot Equation
        // set isen = 3 for Oblique Shock Waves
        // set isen = 4 for Prandtl-Meyer  Equations
    void    set_Rg(double a)  ;
    void    set_F();
    void    set_dFdM();

    //calculate the next iteration Mach number  in the newton solver iteration
    void Newton_iter();
    // Newton solver for oblique shock waves
    void Newton_iter_oblique();
    void Newton_iter_PM();

    //calculate the error (and set the new F and dFdM values) on each iteration of Mach number in the newton solver
    double error();

    //calculate the converged value of Mach number using the Newton iteration solver
    void Newton();

    //calculate the temperature for an isentropic nozzle
    void Temperature();
    void Pressure();
    // calculate the choked mass flow with the given Gamma, Rg, P0 and T0
    void calc_m_dot();
    // oblique shock wave M to M2
    void calc_M2_oblique();
    // Prandtl-Meyer equation V(M)
    double V(double Mach);
    void Stagnation_Pressure();//isentropic relation to get P0 from P
    void Pressure_Oblique();
    void Temperature_Oblique();
    void P0_Oblique();
    void calc_Oblique();
    void P2_Prandtl_Meyer();
    void T2_Prandtl_Meyer();
    void mu_Prandtl_Meyer();
    void calc_Prandtl_Meyer();
    double Thrust(double conical_nozzle_theta,double Pamb); // Thrust as a function of altitude
};
```

```
class CylindricalPort{
    public:
        //Fuel grain geometry
        double L0,D0,d0,rho_propellant;
        // Nozzle Geometry
        double A_star,A_exit,theta_exit;
        // Combustion gas properties
        double gamma,Mw,T0,Rg;
        // Burn Parameters
        double a,n,M_crit,k;
        // how many bates grains
        int N;
       // constructor
       CylindricalPort();
};
```

```
class RP{
    public:
        double r,r_dot,P0,P0_dot;
        RP();
//        double A_burn(CylindricalPort a);
        double A_burn(CylindricalPort a,int b);
//        double V_c(CylindricalPort a);
        double V_c(CylindricalPort a,int b);
};
```

```
class Solver{
    public:
        CylindricalPort CP;
        RP RP0,RP1;
        Prandtl_Meyer Port,Nozzle;
        double t0,t1;
        int bates; // if not bates grain then bates=0, bates grain otherwise
        // constructor
        Solver();
        Solver(CylindricalPort a, RP b,Prandtl_Meyer c, Prandtl_Meyer d);

        // solve P_dot and R_dot
        void calc_RP1_dot();
        // Solve P1 and R1 values
        void calc_RP1();
        // write out values
        void writeout();
        // reset values
        void reset();
        // next step
        void next_step();



};
```

Linked c++ files

```
#ifndef PRANDTL
#define PRANDTL
#include <iostream>
#include <cmath>
#include "Prandtl_Meyer.h"
using namespace std;


void    Prandtl_Meyer::set_isentropic(int  a)  { isen=a;   }
// set isen = 0 for isentropic relationships
// set isen = 1 for non-isentropic relationships
// set isen = 2 for Rayleigh Pitot Equation
// set isen = 3 for Oblique Shock Waves
// set isen = 4 for Prandtl-Meyer  Equations
void    Prandtl_Meyer::set_Rg       (double a)  { Rg=a;      }
void    Prandtl_Meyer::set_F(){
```

```
    if (isen==0){//isentropic nozzle
        F = (1./M)*
            pow(((2./(gamma+1))*(1.+((gamma-1.)/2.)*M*M)),
                ((gamma+1.)/(2.*(gamma-1.))))-
            (A/A_star);
    }
    else if (isen==1){//non-isentropic nozzle, ahead of shock wave
        F = 2.
            /((gamma+1.)*
                pow(gamma*M*M-((gamma-1.)/2.),
                    1./(gamma-1.)))
            *pow(pow((((gamma+1.)/2.)*M),2)/
                (1.+((gamma-1.)/2.)*M*M),
                gamma/(gamma-1.))
            -(P02/P0);
    }
    else if (isen==2){// rayleigh pitot equations
        if (M>1){
            F = (
                pow(((gamma+1.)/2.)*M*M,gamma/(gamma-1.))
                )
                /(
                    pow(((2.*gamma*M*M)/(gamma + 1.))
                        -((gamma - 1.)/(gamma+1.)),1./(gamma-1.))
                )
                - (P02/P)
                ;
        }
        else if (M<=1){
            F = pow(1. + (M*M*(gamma-1.))/(2.),gamma/(gamma-1.))
                - (P02/P)
                ;
        }
    }
    else if (isen==3){// oblique shock waves
        a = (1.+((gamma-1.)/2.)*M*M)*tan(theta);
        b = (M*M-1.);
        c = (1.+((gamma+1.)/2.)*M*M)*tan(theta);
        xj = tan(beta);
        F = 2.*a*xj*xj*xj
            - b*xj*xj - 1.;

    }
    else if (isen==4){// prandtl-meyer equations
        F = (theta + V(M) - V(M2));
    }
}
void Prandtl_Meyer::set_dFdM(){
    if (isen==0){
        //if isen=0 then assume isentropic nozzle
        dFdM= (pow(2.,
                (1.-3.*gamma)/(2.-2*gamma)))*
            (M*M-1.)/
            (M*M*(2.+M*M*(gamma-1.)))*
            pow(((1.+((gamma-1.)/2.)*M*M)/(gamma+1.)),
                ((gamma+1.)/(2*(gamma-1.))));
    }
    //if a=1 then non-isentropic nozzle, ahead of shock wave
    else if (isen==1){
        dFdM= -(pow(2.,
                3.-((2.*gamma)/(gamma-1.)))*
            gamma*
            pow(M*M-1.,2)*
            pow(pow((((gamma+1.)/2.)*M),2)/
                (1.+((gamma-1.)/2.)*M*M),
                gamma/(gamma-1.))*
            pow(0.5 + gamma*(M*M-0.5),-1./(gamma-1.)))
            /
            ((gamma+1.)*M*(2.+M*M*(gamma-1.))*(1.+gamma*(2*M*M-1.)));
    }
    else if (isen==2){// rayleigh pitot equations
        if (M>1){
            dFdM = (
                gamma * M * (2.*M*M-1.) * pow((M*M*(gamma+1.))/2.,1./(gamma-1.))
                )
                /(
                    pow(((2.*gamma)/(gamma+1.))*M*M - (gamma-1.)/(gamma+1.),(gamma/(gamma-1.)))
                );
            //          std::cout<<"M>1"<<std::endl;
        }
        else if (M<=1){
            dFdM = (M*(gamma))
                * pow(1.+((gamma-1.)/2.)*M*M,(1./(gamma-1.)))
                ;
            //          std::cout<<"M<=1"<<std::endl;
        }
    }
    else if (isen==3){
        dFdM = 3.*a*xj*xj
            - 2.*b*xj + c;
    }
    else if (isen==4){
        dFdM = (1./M2)
            *((sqrt(M2*M2 - 1.))
                /(1.+((gamma-1.)/2.) *M2*M2))
            ;
    }
}


//calculate the next iteration Mach number in the newton solver iteration
void Prandtl_Meyer::Newton_iter(){ M-=F/dFdM; }
// Newton solver for oblique shock waves
void Prandtl_Meyer::Newton_iter_oblique(){
    xnew = F/dFdM;
    beta = atan(xnew);
    //      cout<<iter<<" iteration and beta = "<<beta*180./M_PI<<endl;
}
void Prandtl_Meyer::Newton_iter_PM(){ M2+=F/dFdM; }

//calculate the error (and set the new F and dFdM values) on each iteration of Mach number in the newton solver
double Prandtl_Meyer::error(){
    set_F();    // set the F = 0 value for the current iteration
    set_dFdM(); // also calculate the dF/dM value for the current iteration
    iter++;     // each time this is run count up the iterations by one increment
    //      std::cout<<iter<<" iteration Ma = "<<M<<std::endl;
    //      return (std::fabs(F) /(A/A_star));     // alternative iterative solver convergence criteria
    //      cout<<"error = "<<F<<" / "<<dFdM<<endl;
    if (isen!=3){
        return (std::abs(F/dFdM));
    }
    else if (isen==3){
        return (std::abs((F/dFdM-xnew)/(F/dFdM)));
    }
    else
        return 999999999.;
}


//calculate the converged value of Mach number using the Newton iteration solver
void Prandtl_Meyer::Newton(){
    if (isen!=3 && isen!=4){
        // for loop to converge on the correct M value
        for (;error()>tol;) Newton_iter();
    }
    else if (isen==3){
```

```
        // for loop to converge on the correct beta value ( for oblique shock waves)
        for (;error()>tol;) Newton_iter_oblique();
    }
    else if (isen==4){
        // for loop to converge on the correct M2 value for Prandtl_Meyer equations
        for (;error()>tol;) Newton_iter_PM();

    }
    else {
        cout<<"you are in big trouble. Fix your isen value to Newton solve correctly."<<endl;
    }
}

//calculate the temperature for an isentropic nozzle
void Prandtl_Meyer::Temperature(){
    T = T0/
        (1.+(gamma-1.)/2.*M*M);
}
void Prandtl_Meyer::Pressure(){
    P = P0/
        pow((1.+(gamma-1.)/2.*M*M),
                (gamma/(gamma-1.)));
}
// calculate the choked mass flow with the given Gamma, Rg, P0 and T0
void Prandtl_Meyer::calc_m_dot(){
    m_dot = (P0/sqrt(T0))
        * A_star
        * sqrt((gamma/Rg)
                * pow(2./(gamma+1.),
                    (gamma+1.)/(gamma-1.)));
}
// oblique shock wave M to M2
void Prandtl_Meyer::calc_M2_oblique(){
    double Mn2;
    Mn2 = sqrt((1.+((gamma-1.)/2.)*pow(M*sin(beta),2))
            /(gamma*pow(M*sin(beta),2) - (gamma-1.)/2.));
    M2  =   Mn2/(sin(beta-theta));
}
// Prandtl-Meyer equation V(M)
double Prandtl_Meyer::V(double Mach){
    return sqrt((gamma+1.)/(gamma-1.))
        *atan(sqrt((gamma-1.)/(gamma+1.)*(Mach*Mach-1.)))
        - atan(sqrt(Mach*Mach-1.));
}
void Prandtl_Meyer::Stagnation_Pressure(){//isentropic relation to get P0 from P
    P0 = P*
        pow((1.+(gamma-1.)/2.*M*M),
                (gamma/(gamma-1.)));
}
void Prandtl_Meyer::Pressure_Oblique(){
    P2  =   P
        *(1.+(2.*gamma)/(gamma+1.)
                *(pow(M*sin(beta),2)-1.));
}
void Prandtl_Meyer::Temperature_Oblique(){
    T2  =   T*(1.+(2.*gamma)/(gamma+1.)*(pow(M*sin(beta),2)-1.))
        *((2.+(gamma-1.)*(pow(M*sin(beta),2)))
                /((gamma+1.)*pow(M*sin(beta),2)));
}
void Prandtl_Meyer::P0_Oblique(){
    P02   =   P0*2.
        /((gamma+1.)*
                pow(gamma*M*sin(beta)*M*sin(beta)-((gamma-1.)/2.),
                    1./(gamma-1.)))
        *pow(pow((((gamma+1.)/2.)*M*sin(beta)),2)/
                (1.+((gamma-1.)/2.)*M*sin(beta)*M*sin(beta)),
                gamma/(gamma-1.));
}
void Prandtl_Meyer::calc_Oblique(){
    Newton();
    calc_M2_oblique();
    Pressure_Oblique();
    Temperature_Oblique();
    Stagnation_Pressure();
    P0_Oblique();
}
void Prandtl_Meyer::P2_Prandtl_Meyer(){
    P2  =   P
        *pow((1.+((gamma-1.)/2.)*M*M)
                /(1.+((gamma-1.)/2.)*M2*M2)
                ,(gamma)/(gamma-1.));
}
void Prandtl_Meyer::T2_Prandtl_Meyer(){
    T2  =   T
        *(1.+((gamma-1.)/2.)*M*M)
        /(1.+((gamma-1.)/2.)*M2*M2)
        ;
}
void Prandtl_Meyer::mu_Prandtl_Meyer(){
    mu1 =   asin(1./M);
    mu2 =   asin(1./M2)-theta;
}
void Prandtl_Meyer::calc_Prandtl_Meyer(){
    Newton();           // calculate M2 using Prandtl-Meyer equations and a Newton solver iterative method
    P2_Prandtl_Meyer(); // Calculate P2 using Prandtl-Meyer equations
    T2_Prandtl_Meyer(); // Calculate T2 using Prandtl-Meyer equations
    mu_Prandtl_Meyer(); // Calculate mu values before and after using Prandtl-Meyer equations
}
double Prandtl_Meyer::Thrust(double conical_nozzle_theta, double Pamb){ // Thrust as a function of conical nozzle angle (degrees) and pressure at altitude

//    std::cout<<conical_nozzle_theta<<" "<<Pamb<<" "<<P0<<" "<<P<<" "<<gamma<<" "<<A_star<<" "<<std::endl;
    return ((1.+cos(conical_nozzle_theta*M_PI/180.))/2.)
        *
        gamma*P0*A_star
        *
        sqrt(
                (
                 2.
                 /
                 (gamma-1.)
                )
                *
                (
                 pow(2./(gamma+1.),
                     (gamma+1.)/(gamma-1.))
                )
            )
        *
        sqrt(
                (
                 1.
                 -
                 pow(P/P0,
                     (gamma-1.)/gamma)
                )
            )
        +
        A*(P-Pamb)
        ;
}



#endif
```

```
#ifndef CYLINDRICALPORT_H
#define CYLINDRICALPORT_H
#include <cmath>
#include "CylindricalPort.h"

```

```
CylindricalPort::CylindricalPort()
{
            L0                = .035;       // m
            D0                = .066;       // m
            d0                = .03;        // m
            rho_propellant    = 1260.;      // kg/m^3
            A_star            = .0001887;   // m^2
            A_exit            = 4.*A_star;  // m^2
            theta_exit        = 20.*M_PI/180.;   // radians
            gamma             = 1.18;       //
            Mw                = 23.;        // kg/kg-mol
            Rg                = 8.314/(Mw/1000.);
            T0                = 2900.;      // K
            n                 = 0.16;
            a                 = 0.00132/pow(1000.,n);       // m/(s*Pa^n)
            M_crit            = 0.3;
            k                 = 0.2;
            N                 = 3;
}




#endif
```

```
#ifndef RP_H
#define RP_H
#include <cmath>
#include "CylindricalPort.h"
#include "RP.h"

RP::RP(){        // default constructor
    r=0.015;      // m
    r_dot=0.0000001   ;// dm/ds
    P0=0.0001;// Pa
    P0_dot=0.0000001;  // dPa/ds
}
//double RP::A_burn  (CylindricalPort a) { return 2.*M_PI*r*a.L0;    }
double RP::A_burn  (CylindricalPort a,int b) {
    if (b == 0) return 2.*M_PI*r*a.L0;
    else {
//          double N=double(b);
        double s=r-a.d0/2.;
        return a.N*M_PI*
            (
             (a.D0*a.D0-(a.d0+2.*s)*(a.d0+2.*s))/2.
             +
             ((a.L0-2.*s)*(a.d0+2.*s))
            );
    }
}
//double RP::V_c     (CylindricalPort a) { return M_PI*r*r*a.L0;    }
double RP::V_c     (CylindricalPort a,int b) {
    if (b == 0) return M_PI*r*r*a.L0;
    else{
//          double N=double(b);
        double s=r-a.d0/2.;
        return a.N*M_PI/4.
            * ((a.d0+2.*s)*(a.d0+2.*s) * (a.L0-2.*s)
                    + a.D0*a.D0*2.*s
              );
    }
}


#endif
```

```
#ifndef SOLVER_H
#define SOLVER_H
#include "CylindricalPort.h"
#include "Prandtl_Meyer.h"
#include "RP.h"
#include "Solver.h"
#include <cmath>
#include <iostream>

Solver::Solver(){
    t0=0.;
    t1=0.;
    bates = 0;
}
Solver::Solver(CylindricalPort a, RP b,Prandtl_Meyer c,
    Prandtl_Meyer d){
    CP = a;
    RP0= b;
    RP1= b;
    Port = c;
    Nozzle = d;

    t0=0.;
    t1=0.;

    bates = 0;
}

void Solver::calc_RP1_dot(){
//    RP1.r_dot = CP.a * pow(RP0.P0,CP.n)/pow(1000.,CP.n);
    Port.A = RP0.V_c(CP,bates)/CP.L0;
    RP1.r_dot = CP.a * pow(RP0.P0,CP.n)
        * ((
            1.+CP.k*(Port.M/CP.M_crit))
        / (1.+CP.k));
    RP1.P0_dot = (RP0.A_burn(CP,bates) * RP1.r_dot / RP0.V_c(
        CP,bates) )
//    RP1.P0_dot = (2. * RP1.r_dot / RP0.r )
        * (CP.rho_propellant*CP.Rg*CP.T0-RP0.P0)
        - ((CP.A_star/RP0.V_c(CP,bates))
        * (RP0.P0)
        * sqrt(CP.gamma*CP.Rg*CP.T0
                * pow(2./(CP.gamma+1.),(CP.gamma+1.)/(CP.
                    gamma-1.))));
}
void Solver::calc_RP1(){
    RP1.P0 = RP0.P0 + (t1-t0) * RP1.P0_dot;
    RP1.r  = RP0.r  + (t1-t0) * RP1.r_dot;
}
void Solver::writeout(){
//    std::cout<<std::endl;
    if (isnan(RP1.P0)) {
        std::cerr<<"nan on RP1.P0"<<std::endl;
        return;
    }
    std::cout<<t1<<" "<<RP1.P0_dot  <<" "<<RP1.r_dot<<" "<<
        RP1.P0<<" "<<RP1.r<<" "<<RP1.A_burn(CP,bates)<<" "<<
         ((CP.D0*CP.D0*M_PI/4.*CP.L0)-RP1.V_c(CP,bates))*CP.
        rho_propellant<<" "<<((RP1.V_c(CP,bates)-RP0.V_c(CP,
        bates))*CP.rho_propellant)/(t1-t0)<<" "<<Port.m_dot
        <<" "<<Nozzle.Thrust(20.,0.)<<" "<<Nozzle.Thrust
        (20.,0.)/(Port.m_dot*9.81)<<" "<<Port.M<<" "<<" "<<
        Nozzle.M<<std::endl;
}
void Solver::reset(){
    RP0 = RP1;
    t0  = t1;
}
void Solver::next_step(){
    calc_RP1_dot();
    calc_RP1();
    Port.P0=RP1.P0;
    Nozzle.P0=RP1.P0;
    Port.Newton();
    Port.Pressure();
    Port.calc_m_dot();
    Nozzle.Newton();
    Nozzle.Pressure();
    Nozzle.calc_m_dot();
    writeout();
    reset();
}

```

```
#endif
```

Main Program

```
#include "DEFS.h"
#include "Prandtl_Meyer.h"
#include "CylindricalPort.h"
#include "RP.h"
#include "Solver.h"
int main(){

/*
    RP a;
    CylindricalPort b;
    Prandtl_Meyer PM,PM2;
    b.Mw=197.231;// g/mol
    b.Rg = 8.314/(b.Mw/1000.);
    b.A_exit=3.8*3.8*M_PI/4.;
    b.A_star=b.A_exit/7.78;
    b.gamma=1.02;
    b.L0=34.57;
    b.D0=3.66;
    b.d0=1.7;
    b.n=0.172;
    b.a=0.00192/pow(1000.,b.n); // m/(s*Pa^n)
    b.rho_propellant=1760.;
    a.r=b.d0/2.;
    b.T0=24000.;
    b.k=0;
//    cout<<"A_star = "<<b.A_star<<endl;
//    cout<<"A_exit = "<<b.A_exit<<endl;
//    cout<<"launch mass = "<<(b.D0*b.D0*M_PI/4.*b.L0-a.V_c(b))*b.rho_propellant<<endl;
    PM.A = a.V_c(b,0)/b.L0;
    PM.A_star=b.A_star;
    PM.gamma=b.gamma;
    PM.P0 = a.P0;
    PM.T0 = b.T0;
    PM.Rg = b.Rg;
    PM.M = 0.15; // guess subsonic region
    PM.set_isentropic(0);
    PM2=PM;
    PM2.A= b.A_exit;
    PM2.M= 2.;
    Solver S(b,a,PM,PM2);
//    cout<<"launch mass = "<<(b.D0*b.D0*M_PI/4.*b.L0-a.V_c(b,S.bates))*b.rho_propellant<<endl;
//    cout<<"Burn area = "<<(S.RP0.A_burn(S.CP,S.bates))<<endl;




    cout.precision(16);
    S.writeout();
//    while (S.t1<125.) {
    while (1){
        S.t1=S.t0+.01;
        S.next_step();
        if (S.RP0.r>=S.CP.D0/2.){ break; }
    }

*/

/*
    // non-erosive vs erosive burning
    RP a;
    CylindricalPort b;
    b.L0=0.35;
    b.D0=0.076;
    b.d0=0.03;
    b.rho_propellant=1314.;
    b.A_star=0.0001887;
    b.A_exit=4.*b.A_star;
    b.gamma=1.2;
    b.Mw=24.26;
    b.Rg = 8.314/(b.Mw/1000.);
    b.T0=2000.;
    b.M_crit=0.11;
    b.k=2.25;
    b.n=0.188;
    b.a=0.00178/pow(1000.,b.n);// m/(s*Pa^n)
    Prandtl_Meyer Port,Nozzle;
    Port.A = a.V_c(b,0)/b.L0;
    Port.A_star=b.A_star;
    Port.gamma=b.gamma;
    Port.P0 = a.P0;
    Port.T0 = b.T0;
    Port.Rg = b.Rg;
    Port.M = 0.15; // guess subsonic region
    Port.set_isentropic(0);
    Nozzle=Port;
    Nozzle.A= b.A_exit;
    Nozzle.M= 2.6;
    Solver S(b,a,Port,Nozzle);

    cout.precision(16);
    S.writeout();
//    S.CP.k=0.;

    while (1){
        S.t1=S.t0+.001;
        S.next_step();
        if (S.RP0.r>=S.CP.D0/2.){ break; }
    }
*/

///*
    // Part 1 cylindrical port
    // initialize parameters
    RP a;
    CylindricalPort b;
    Prandtl_Meyer Port,Nozzle;
    Port.A = a.V_c(b,0)/b.L0;
    Port.A_star=b.A_star;
    Port.gamma=b.gamma;
    Port.P0 = a.P0;
    Port.T0 = b.T0;
    Port.Rg = b.Rg;
    Port.M = 0.16; // guess subsonic region
    Port.set_isentropic(0);
    Nozzle=Port;
    Nozzle.A= b.A_exit;
    Nozzle.M= 2.6;
    Solver S(b,a,Port,Nozzle);
    S.bates=0;  // cylindrical port
    S.Port.Newton();
    // solve and output
    cout.precision(16);
//    S.CP.k=0.; // non-erosive burn
    S.writeout();
    while (1){
        S.t1=S.t0+.0001;
        S.next_step();
        if (S.RP0.r>=S.CP.D0/2.){ break; }
    }
//*/
/*
    // Part 2 bates grain
    // initialize parameters
    RP a;
    CylindricalPort b;
    Prandtl_Meyer Port,Nozzle;
    Port.A = a.V_c(b,0)/b.L0;
    Port.A_star=b.A_star;
    Port.gamma=b.gamma;
    Port.P0 = a.P0;
    Port.T0 = b.T0;
    Port.Rg = b.Rg;
    Port.M = 0.16; // guess subsonic region
    Port.set_isentropic(0);
    Nozzle=Port;
    Nozzle.A= b.A_exit;
```

```
    Nozzle.M= 2.6;
    Solver S(b,a,Port,Nozzle);
    S.bates=1;        // solve using bates grain
    S.Port.Newton();
    // solve and output
    cout.precision(16);
    S.CP.k=0.; // non-erosive burn
    S.writeout();
    while (1){
        S.t1=S.t0+.0001;
        S.next_step();
        if (S.RP0.r>=S.CP.D0/2.){ break; }
    }
*/

/*
    // Part 2 bates
    // initialize parameters
    RP a;
    CylindricalPort b;
    b.T0 = 3300.; // higher values of flame temp
    Prandtl_Meyer Port,Nozzle;
    Port.A = a.V_c(b,0)/b.L0;
    Port.A_star=b.A_star;
    Port.gamma=b.gamma;
    Port.P0 = a.P0;
    Port.T0 = b.T0;
    Port.Rg = b.Rg;
    Port.M = 0.16; // guess subsonic region
    Port.set_isentropic(0);
    Nozzle=Port;
    Nozzle.A= b.A_exit;
    Nozzle.M= 2.6;
    Solver S(b,a,Port,Nozzle);
    S.bates=1;
    S.Port.Newton();
    // solve and output
    cout.precision(16);
    S.CP.k=0.; // non-erosive burn
    S.writeout();
    while (1){
        S.t1=S.t0+.0001;
        S.next_step();
        if (S.RP0.r>=S.CP.D0/2.){ break; }
    }
*/




    return 0;
}
```