

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.

t time in seconds

CONTENTS

1 INTRODUCTION	1
2 RESULTS	2
2.1 Part 1 Cylindrical Port with Erosive burning . . .	2
2.2 Part 2 Bates grain with non-erosive burning . . .	5
2.3 Sensitivity and effect of flame temperature	7
3 CONCLUSION	8
A Appendix A: Code	8

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

1 INTRODUCTION

A small missile 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.

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

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

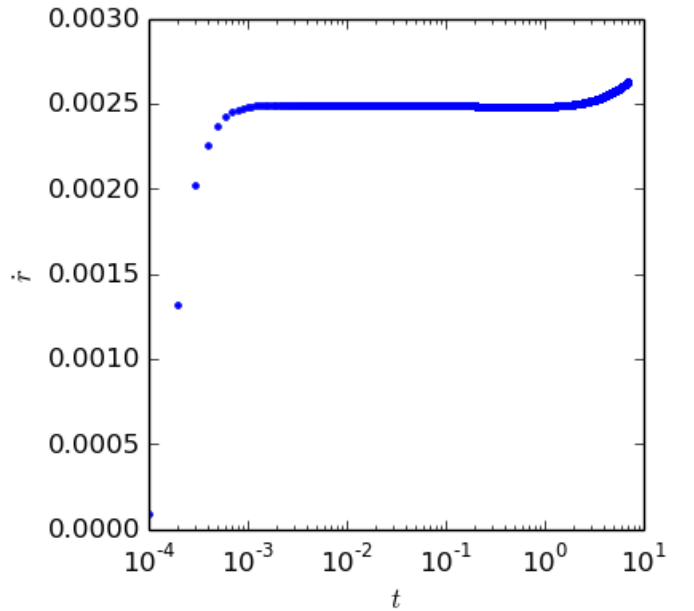
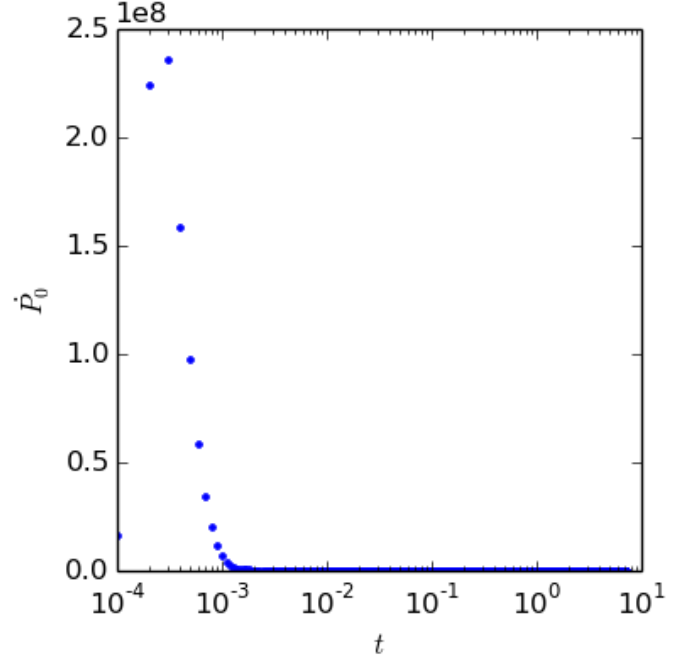
$$\begin{aligned}
\dot{P}_0 &= \frac{A_{burn}\dot{r}}{V_c} (\rho_{propellant}R_gT_0 - P_0) P_0 \sqrt{\gamma R_g T_0 \left(\frac{2}{\gamma+1}\right)^{\frac{\gamma+1}{\gamma-1}}} \\
\dot{r} &= aP_0^n \left(\frac{1+k\frac{M_{port}}{M_{crit}}}{1+k}\right) \\
P_0 &= P_{ambient} \\
r &= \frac{d_0}{2} \\
A_{burn}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

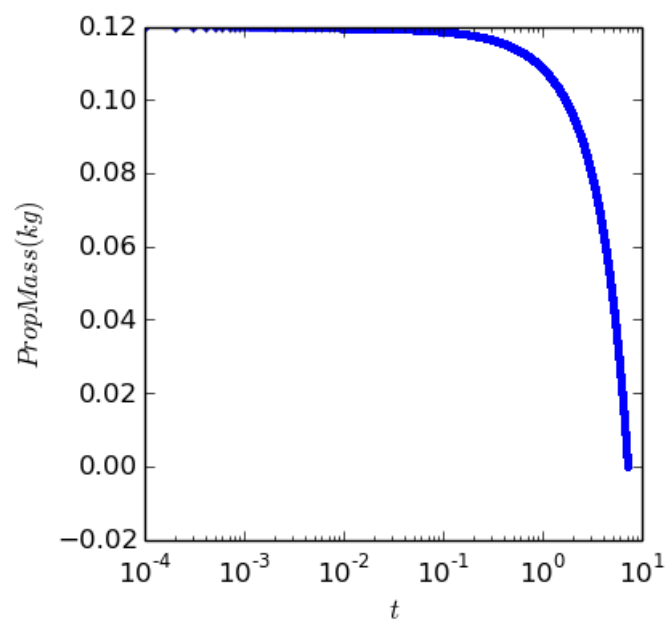
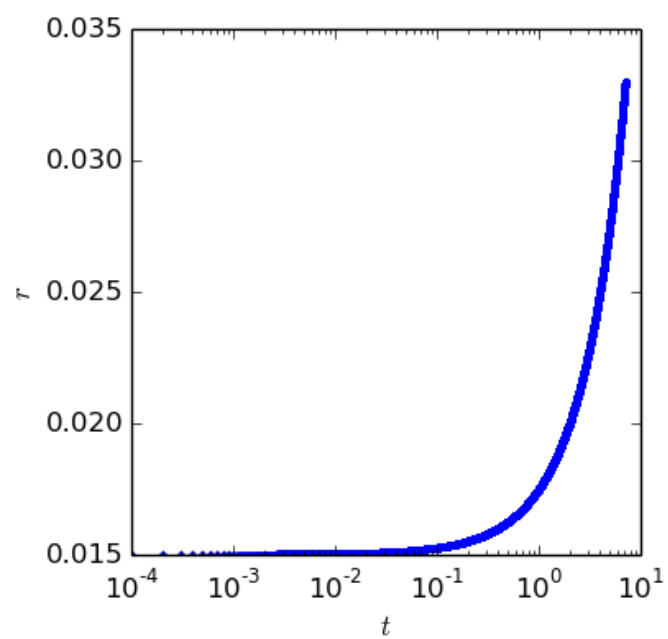
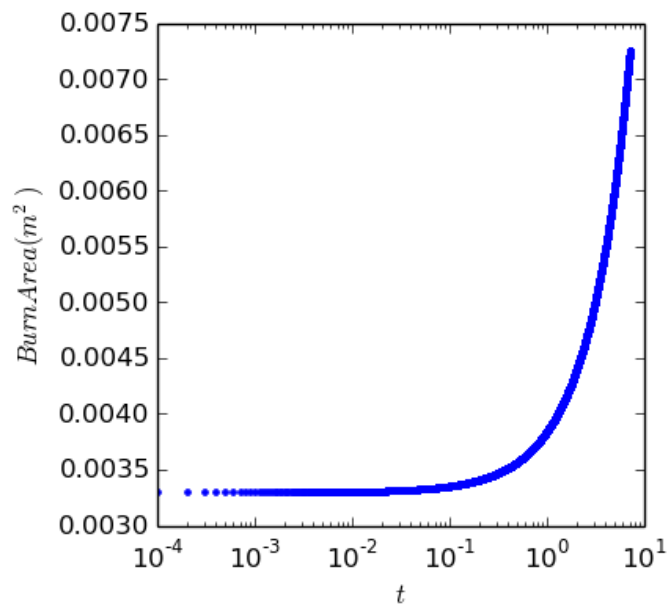
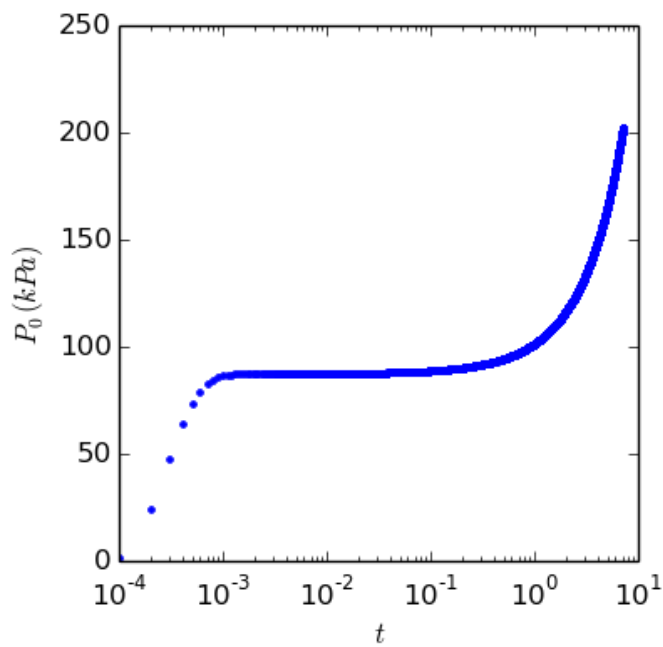
2 RESULTS

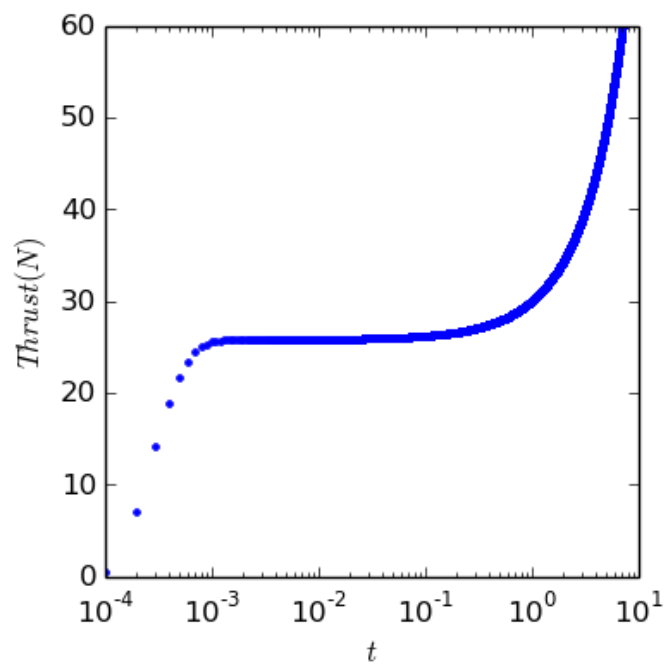
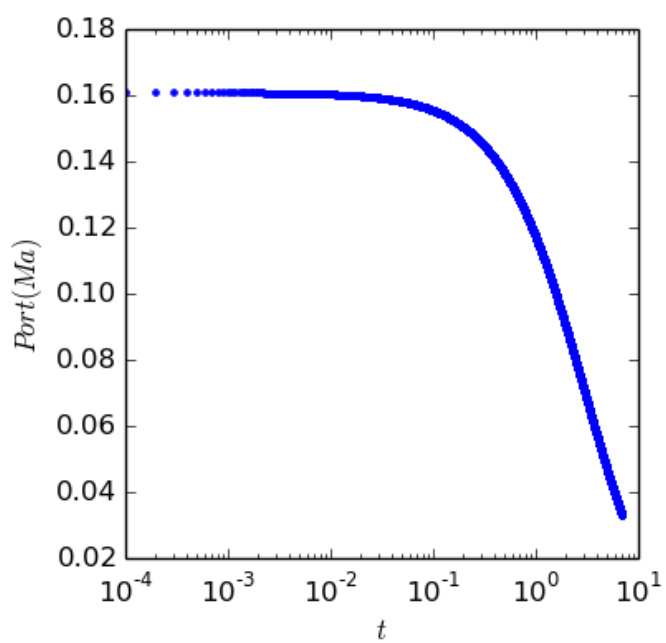
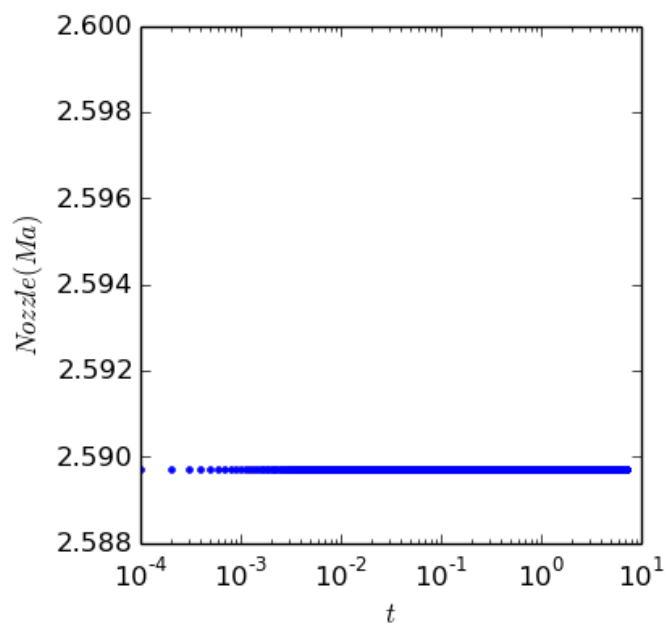
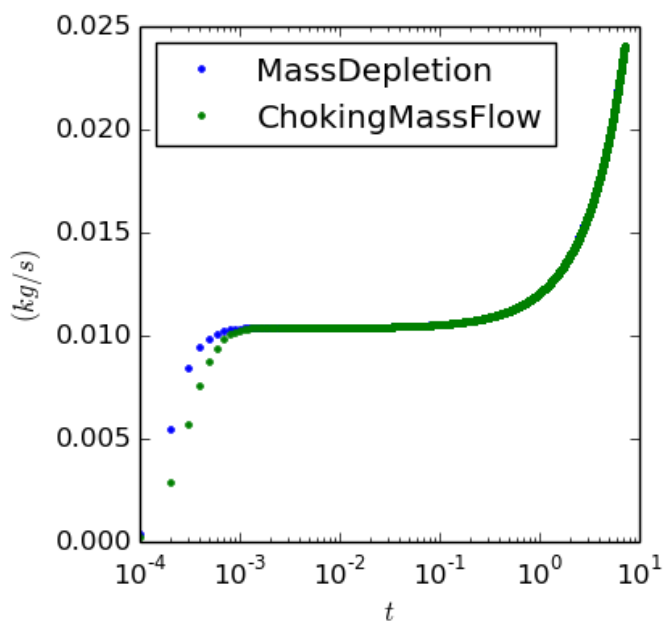
Each simulation had their own set of results. The subsections below will outline the results for each.

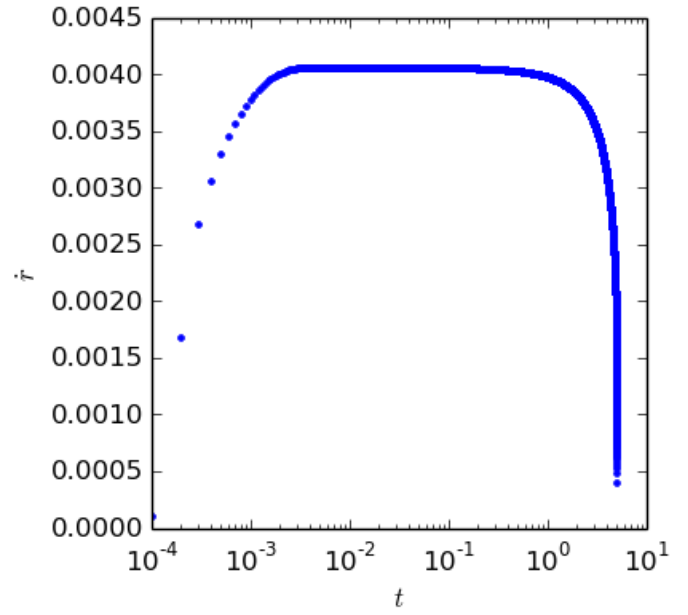
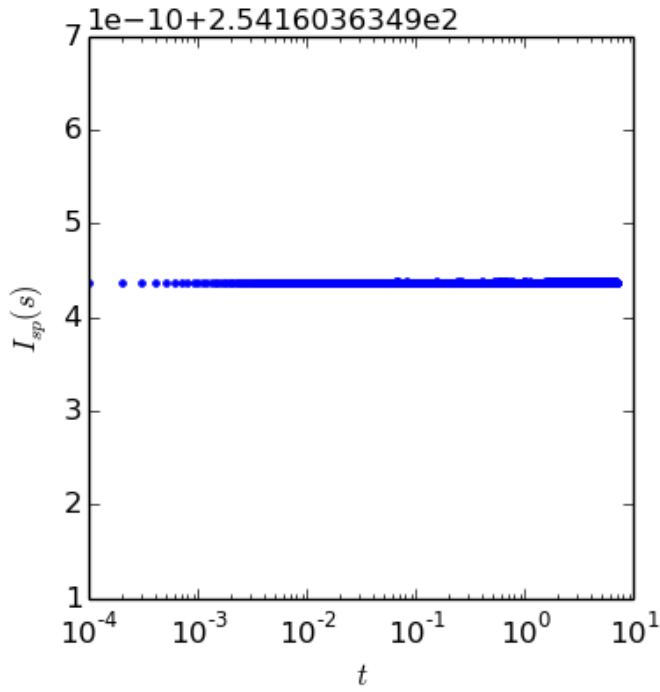
2.1 Part 1 Cylindrical Port with Erosive burning

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





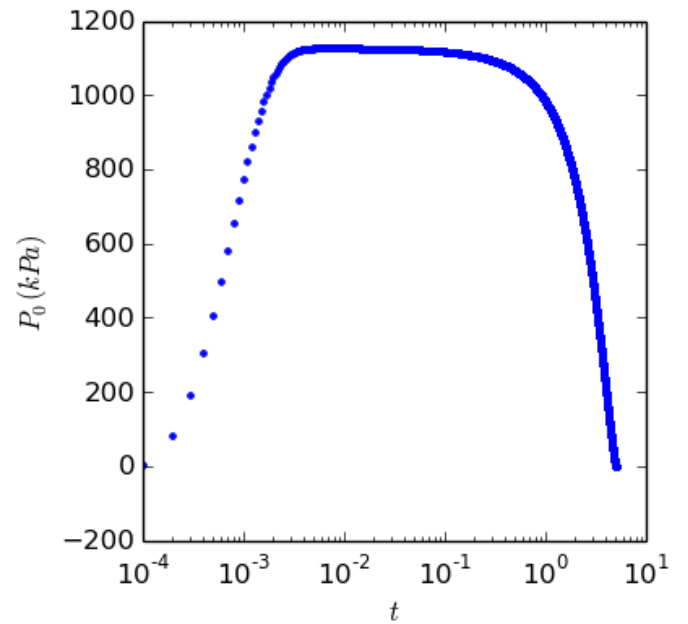
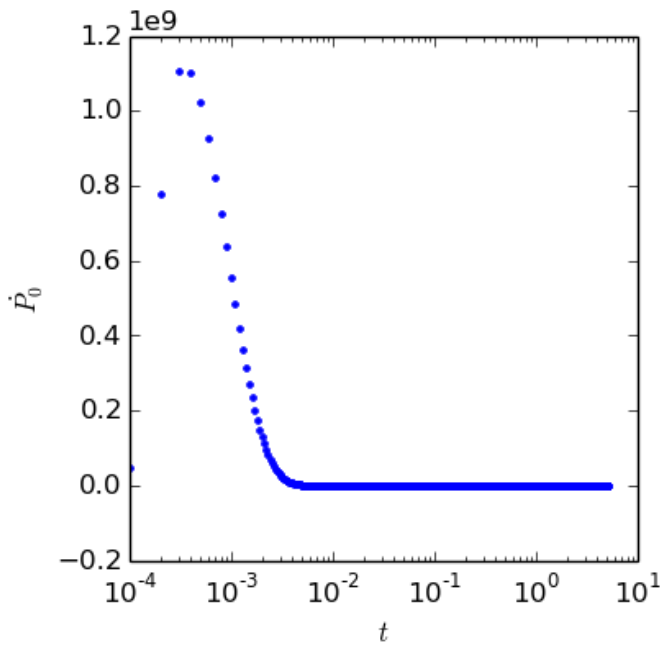


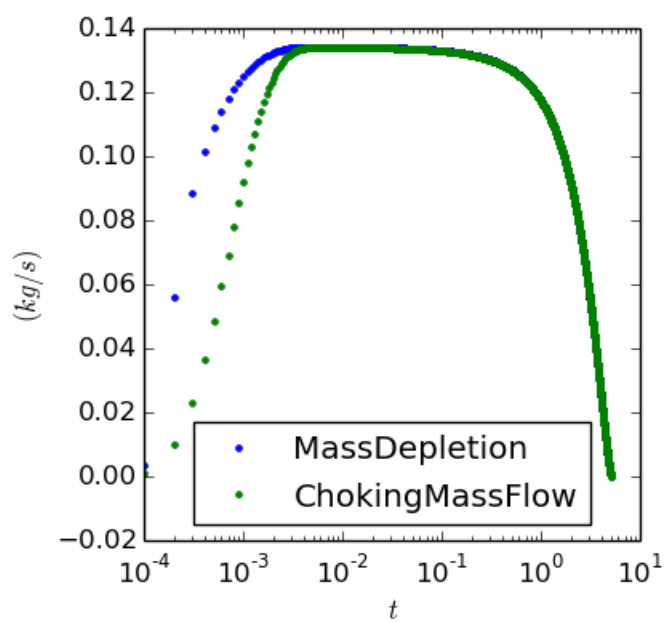
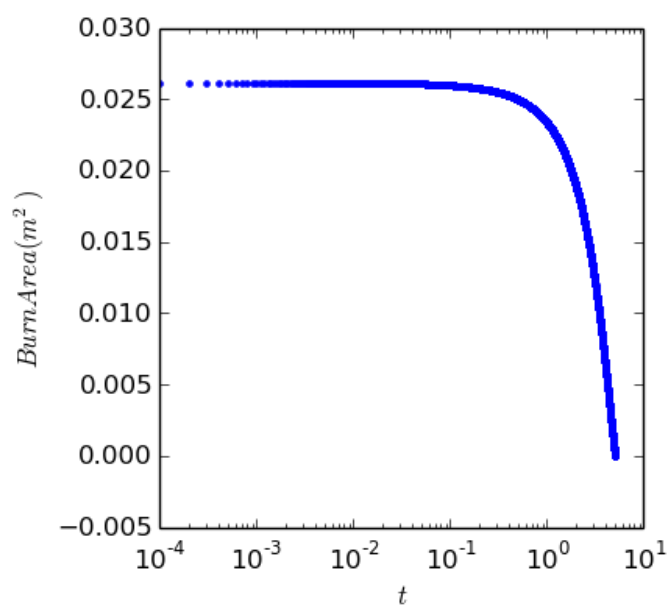
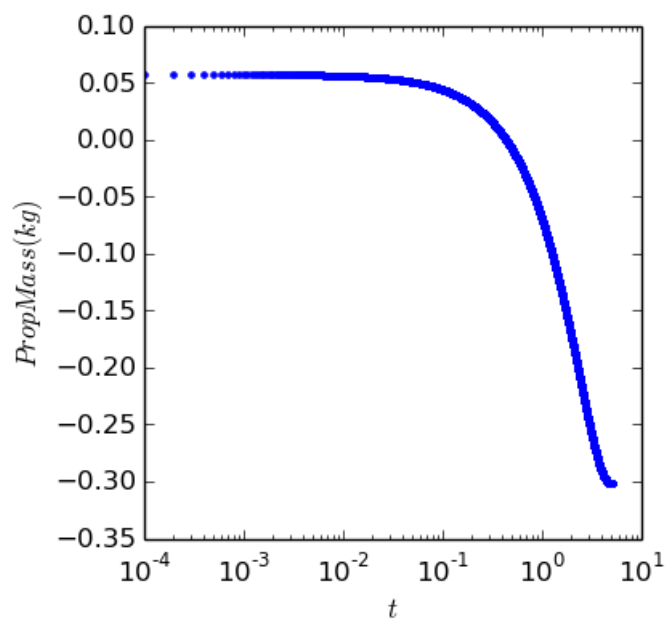
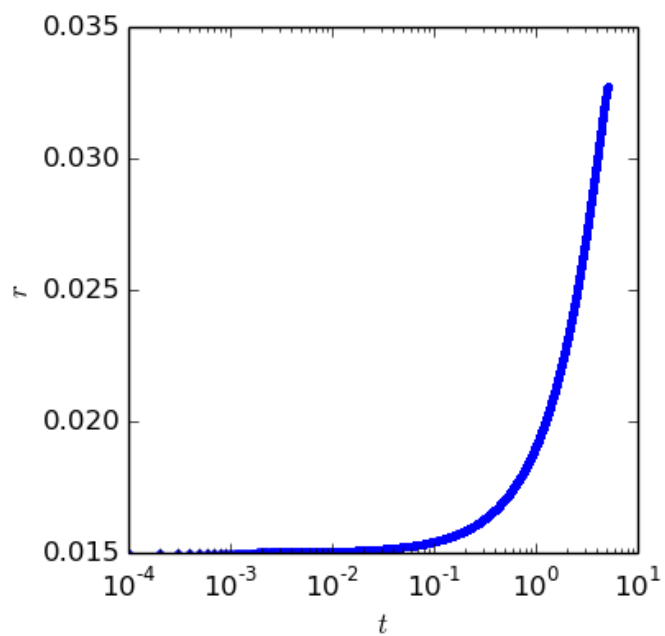


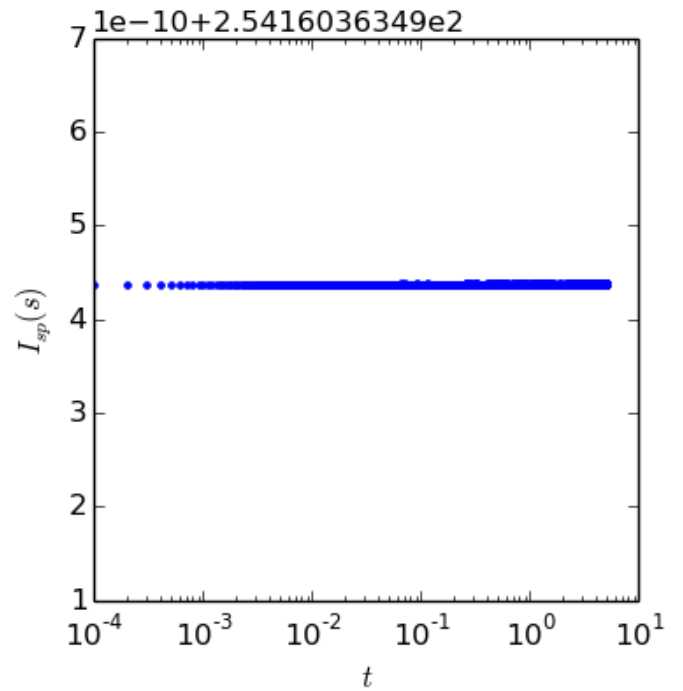
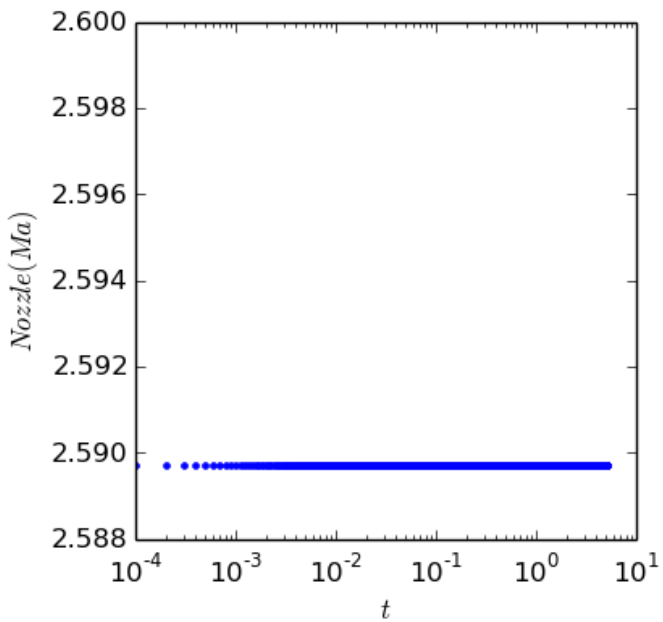
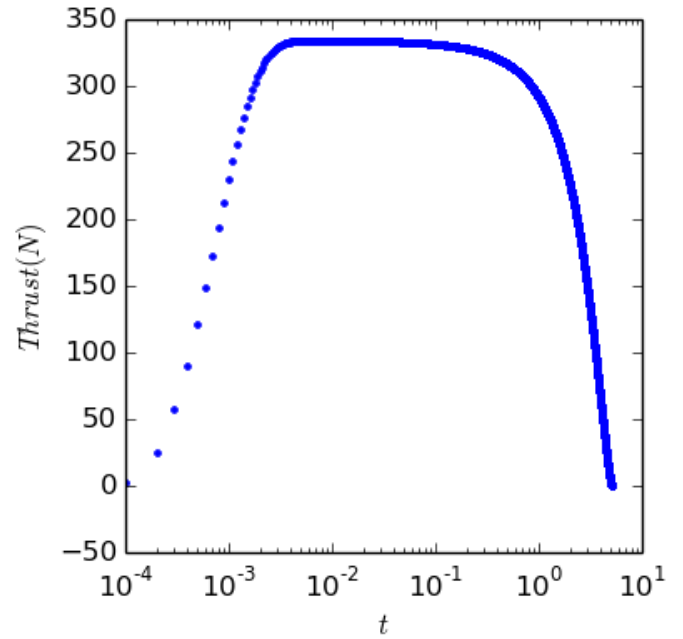
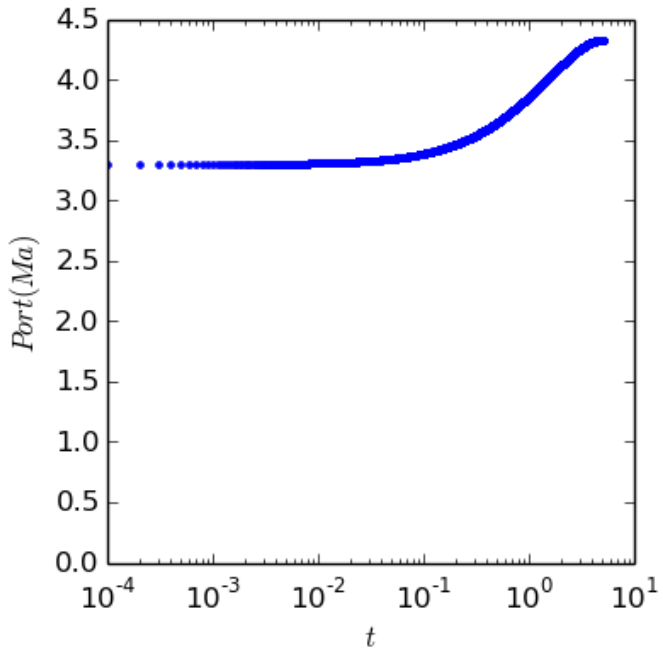
Where the effective mean $I_{sp} = 254.16s$.

2.2 Part 2 Bates grain with non-erosive burning

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







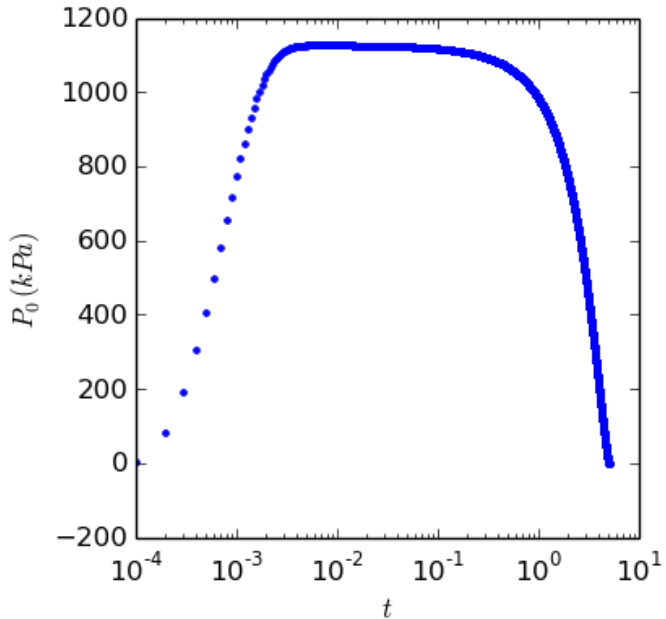
Where the effective mean $I_{sp} = 254.16s$.

2.3 Sensitivity and effect of flame temperature

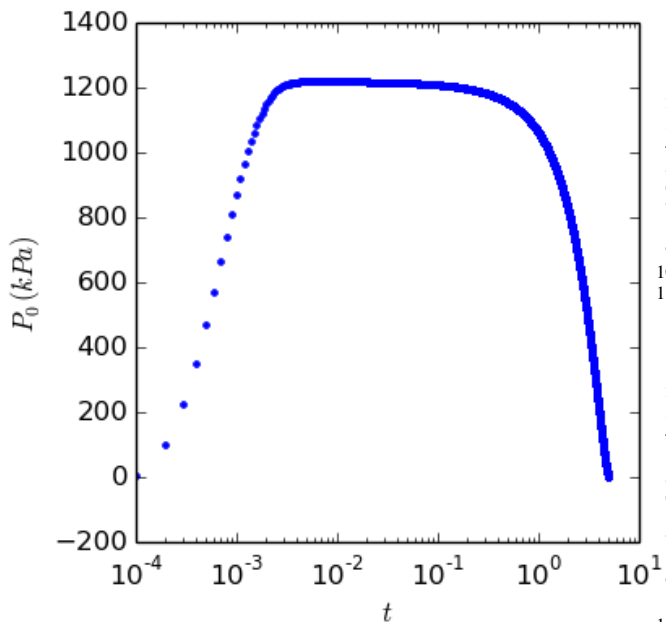
In addition to the above plots, analysis was run to see the sensitivity of the input parameters. It was found that small changes can lead to large changes in the output values. Thus, care must be taken to ensure correct and repeatable parameters are provided for rockets. For example, by changing $T_0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

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

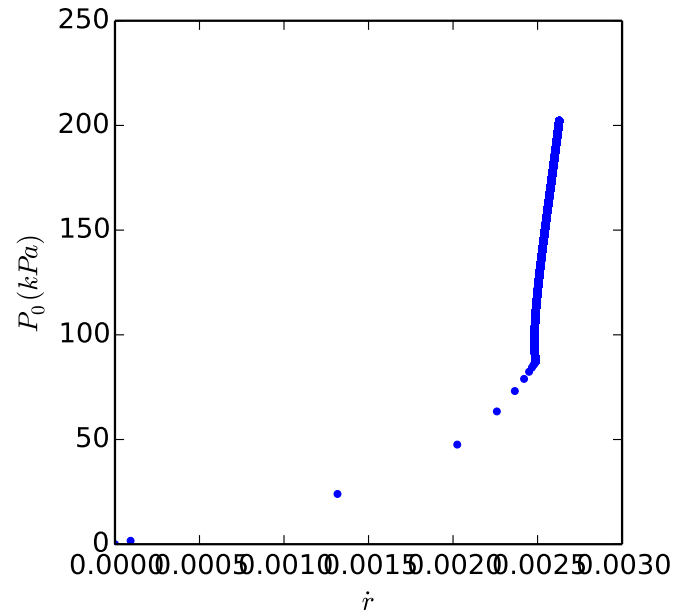
This is the original P_0 plot shown above in Part 2.



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

```

1 #ifndef DEFS.H
2 #define DEFS.H
3 #include <iostream>
4 #include <vector>
5 #include <cmath>
6 #include <string>
7 #include <sstream>
8 #include <iomanip>
9 #include <fstream>
10 using namespace std;
11 #endif

1 // this is a newton solver for computing the area-Mach number
  relation
2 class PrandtlMeyer{
3 public:
4     // public values of gamma, A as a function of x, A at the
      throat, Mach number, the Newton iteration tolerance
5     double gamma, A, A_star, M, tol;
6     //public values for F = 0, dF/dM, and the number of
      iterations for the Newton solver
7     double F, dFdM;
8     int iter;
9     //public values for P0 and T0 (stagnation pressure and
      temperature for an isentropic nozzle)
10    //and local P and T (if an isentropic nozzle)
11    double P0,T0,P,T;
12    // non-isentropic values
13    double P02;
14    // x,yp,ym (positive and negative) values within the
      nozzle

```



```

15  double x,yp,ym;
16  // set a=1 for non-isentropic and a=0 for isentropic
    nozzle
17  double isen;
18  // ideal gas Rg property
19  double Rg;
20  // mass flow m_dot
21  double m_dot;
22  // oblique shock wave values of x,a,b,c and xnew and beta
23  double xj,a,b,c,xnew,beta;
24  // oblique shock wave density, pressure and temperature
    before and after
25  double rho1,rho2,P2,T2;
26  // M2 and theta and mu1 and mu2 of fan for Prandtl-Meyer
    equations (theta is reused for oblique shock waves
    as well)
27  double M2,theta,mu1,mu2;
28  // constructors
29  Prandtl_Meyer(){
30      gamma    = 1.18;
31      A_star   = 0.0001887;
32      A        = 4.*A_star;
33      M        = 4.0; //initial assume it is supersonic
        desired region
34      tol      = 0.00000001;
35      iter     = 0;
36      P0       = 24250000.;
37      T0       = 2900.;
38  }
39
40  //set public values
41  void set_isentropic(int a) ;
42      // set isen = 0 for isentropic relationships
43      // set isen = 1 for non-isentropic relationships
44      // set isen = 2 for Rayleigh Pitot Equation
45      // set isen = 3 for Oblique Shock Waves
46      // set isen = 4 for Prandtl-Meyer Equations
47  void set_Rg(double a) ;
48  void set_F();
49  void set_dFdM();
50
51  //calculate the next iteration Mach number in the newton
    solver iteration
52  void Newton_iter();
53  // Newton solver for oblique shock waves
54  void Newton_iter_oblique();
55  void Newton_iter_PM();
56
57  //calculate the error (and set the new F and dFdM values)
    on each iteration of Mach number in the newton
    solver
58  double error();
59
60  //calculate the converged value of Mach number using the
    Newton iteration solver
61  void Newton();
62
63  //calculate the temperature for an isentropic nozzle
64  void Temperature();
65  void Pressure();
66  // calculate the choked mass flow with the given Gamma,
    Rg, P0 and T0
67  void calc_m_dot();
68  // oblique shock wave M to M2
69  void calc_M2_oblique();
70  // Prandtl-Meyer equation V(M)
71  double V(double Mach);
72  void Stagnation-Pressure();//isentropic relation to get
    P0 from P
73  void Pressure_Oblique();
74  void Temperature_Oblique();
75  void P0_Oblique();
76  void calc_Oblique();
77  void P2_Prandtl_Meyer();
78  void T2_Prandtl_Meyer();
79  void mu_Prandtl_Meyer();
80  void calc_Prandtl_Meyer();
81
    double Thrust(double conical_nozzle_theta,double Pamb);
    // Thrust as a function of altitude
82 };

1  class CylindricalPort{
2  public:
3      //Fuel grain geometry
4      double L0,D0,d0,rho_propellant;
5      // Nozzle Geometry
6      double A_star,A_exit,theta_exit;
7      // Combustion gas properties
8      double gamma,Mw,T0,Rg;
9      // Burn Parameters
10     double a,n,M_crit,k;
11     // how many bates grains
12     int N;
13     // constructor
14     CylindricalPort();
15 };

1  class RP{
2  public:
3      double r,r_dot,P0,P0_dot;
4      RP();
5      // double A_burn(CylindricalPort a);
6      double A_burn(CylindricalPort a,int b);
7      // double V_c(CylindricalPort a);
8      double V_c(CylindricalPort a,int b);
9  };

1  class Solver{
2  public:
3      CylindricalPort CP;
4      RP RP0,RP1;
5      Prandtl_Meyer Port,Nozzle;
6      double t0,t1;
7      int bates; // if not bates grain then bates=0, bates
        grain otherwise
8      // constructor
9      Solver();
10     Solver(CylindricalPort a, RP b,Prandtl_Meyer c,
        Prandtl_Meyer d);
11
12     // solve P_dot and R_dot
13     void calc_RP1_dot();
14     // Solve P1 and R1 values
15     void calc_RP1();
16     // write out values
17     void writeout();
18     // reset values
19     void reset();
20     // next step
21     void next_step();
22
23
24
25 };

Linked c++ files

1  #ifndef PRANDTL
2  #define PRANDTL
3  #include <iostream>
4  #include <cmath>
5  #include "Prandtl_Meyer.h"
6  using namespace std;
7
8
9  void Prandtl_Meyer::set_isentropic(int a) { isen=a; }
10 // set isen = 0 for isentropic relationships
11 // set isen = 1 for non-isentropic relationships
12 // set isen = 2 for Rayleigh Pitot Equation
13 // set isen = 3 for Oblique Shock Waves
14 // set isen = 4 for Prandtl-Meyer Equations
15 void Prandtl_Meyer::set_Rg (double a) { Rg=a; }
16 void Prandtl_Meyer::set_F(){

```

```

17 if (isen==0){//isentropic nozzle
18     F = (1./M)*
19         pow(((2./(gamma+1))* (1.+((gamma-1.)/2.)*M*M)),
20             ((gamma+1.)/(2.*(gamma-1.))))-
21             (A/A_star);
22 }
23 else if (isen==1){//non-isentropic nozzle, ahead of shock
24     wave
25     F = 2.
26         /((gamma+1.)*
27             pow(gamma*M*M-((gamma-1.)/2.),
28                 1./(gamma-1.))
29             *pow(pow(((gamma+1.)/2.)*M),2)/
30             (1.+((gamma-1.)/2.)*M*M),
31             gamma/(gamma-1.))
32     -(P02/P0);
33 }
34 else if (isen==2){// rayleigh pitot equations
35     if (M>1){
36         F = (
37             pow(((gamma+1.)/2.)*M*M, gamma/(gamma-1.))
38             )
39             /(
40                 pow(((2.*gamma*M*M)/(gamma+1.))
41                     -((gamma-1.)/(gamma+1.)), 1./
42                     (gamma-1.))
43             )
44             - (P02/P)
45             ;
46     }
47     else if (M<=1){
48         F = pow(1. + (M*M*(gamma-1.))/(2.), gamma/(
49             gamma-1.))
50             - (P02/P)
51             ;
52     }
53     else if (isen==3){// oblique shock waves
54         a = (1.+((gamma-1.)/2.)*M*M)*tan(theta);
55         b = (M*M-1.);
56         c = (1.+((gamma+1.)/2.)*M*M)*tan(theta);
57         xj = tan(beta);
58         F = 2.*a*xj*xj*xj
59             - b*xj*xj - 1.;
60     }
61     else if (isen==4){// prandtl-meyer equations
62         F = (theta + V(M) - V(M2));
63     }
64 }
65 void Prandtl_Meyer::set_dFdM(){
66     if (isen==0){
67         //if isen=0 then assume isentropic nozzle
68         dFdM= (pow(2.,
69             (1.-3.*gamma)/(2.-2.*gamma)))*
70             (M*M-1.)/
71             (M*M*(2.+M*M*(gamma-1.))*
72             pow(((1.+((gamma-1.)/2.)*M*M)/(gamma+1.)),
73                 ((gamma+1.)/(2.*(gamma-1.)))));
74     }
75     //if a=1 then non-isentropic nozzle, ahead of shock wave
76     else if (isen==1){
77         dFdM= -(pow(2.,
78             3.-((2.*gamma)/(gamma-1.)))*
79             gamma*
80             pow(M*M-1.,2)*
81             pow(pow(((gamma+1.)/2.)*M),2)/
82             (1.+((gamma-1.)/2.)*M*M),
83             gamma/(gamma-1.))*
84             pow(0.5 + gamma*(M*M-0.5), -1./(gamma-1.))
85             /
86             ((gamma+1.)*M*(2.+M*M*(gamma-1.))* (1.+gamma*(2*M*
87                 M-1.)));
88     }
89     else if (isen==2){// rayleigh pitot equations
90         if (M>1){
91             dFdM = (
92                 gamma * M * (2.*M*M-1.) * pow((M*M*(gamma
93                     +1.))/2., 1./(gamma-1.))
94                 )
95                 /(
96                     pow(((2.*gamma)/(gamma+1.))*M*M - (
97                         gamma-1.)/(gamma+1.), (gamma/(
98                             gamma-1.)))
99                     );
100             // std::cout<<"M>1"<<std::endl;
101         }
102         else if (M<=1){
103             dFdM = (M*(gamma))
104                 * pow(1.+((gamma-1.)/2.)*M*M, (1./(gamma-1.)))
105                 ;
106             // std::cout<<"M<=1"<<std::endl;
107         }
108     }
109 }
110 //calculate the next iteration Mach number in the newton
111 solver iteration
112 void Prandtl_Meyer::Newton_iter(){ M-=F/dFdM; }
113 // Newton solver for oblique shock waves
114 void Prandtl_Meyer::Newton_iter_oblique(){
115     xnew = F/dFdM;
116     beta = atan(xnew);
117     // cout<<"iter<<" iteration and beta = "<<beta
118         *180./M.PI<<endl;
119 }
120 void Prandtl_Meyer::Newton_iter_PM(){ M2+=F/dFdM; }
121 //calculate the error (and set the new F and dFdM values) on
122 each iteration of Mach number in the newton solver
123 double Prandtl_Meyer::error(){
124     set_F(); // set the F = 0 value for the current
125     iteration
126     set_dFdM(); // also calculate the dF/dM value for the
127     current iteration
128     iter++; // each time this is run count up the
129     iterations by one increment
130     // std::cout<<"iter<<" iteration Ma = "<<M<<std::
131         endl;
132     // return (std::fabs(F)/(A/A_star)); //
133     // alternative iterative solver convergence criteria
134     // cout<<"error = "<<F<<" / "<<dFdM<<endl;
135     if (isen!=3){
136         return (std::abs(F/dFdM));
137     }
138     else if (isen==3){
139         return (std::abs((F/dFdM-xnew)/(F/dFdM)));
140     }
141     else
142         return 999999999.;
143 }
144 //calculate the converged value of Mach number using the
145 Newton iteration solver
146 void Prandtl_Meyer::Newton(){
147     if (isen!=3 && isen!=4){
148         // for loop to converge on the correct M value
149         for (;error()>tol; Newton_iter());
150     }
151     else if (isen==3){
152

```

```

153 // for loop to converge on the correct beta value (
154 // for oblique shock waves)
155 for (; error() > tol;) Newton_iter_oblique();
156 else if (isen == 4){
157 // for loop to converge on the correct M2 value for
158 // Prandtl-Meyer equations
159 for (; error() > tol;) Newton_iter_PM();
160 }
161 else {
162 cout << "you are in big trouble. Fix your isen value to
163 // Newton solve correctly." << endl;
164 }
165 }
166 // calculate the temperature for an isentropic nozzle
167 void Prandtl_Meyer::Temperature() {
168 T = T0 /
169 (1. + (gamma - 1.) / 2. * M * M);
170 }
171 void Prandtl_Meyer::Pressure() {
172 P = P0 /
173 pow((1. + (gamma - 1.) / 2. * M * M),
174 (gamma / (gamma - 1.)));
175 }
176 // calculate the choked mass flow with the given Gamma, Rg,
177 // P0 and T0
178 void Prandtl_Meyer::calc_m_dot() {
179 m_dot = (P0 / sqrt(T0))
180 * A_star
181 * sqrt((gamma / Rg)
182 * pow(2. / (gamma + 1.),
183 (gamma + 1.) / (gamma - 1.)));
184 }
185 // oblique shock wave M to M2
186 void Prandtl_Meyer::calc_M2_oblique() {
187 double Mn2;
188 Mn2 = sqrt((1. + ((gamma - 1.) / 2.) * pow(M * sin(beta), 2))
189 / (gamma * pow(M * sin(beta), 2) - (gamma - 1.) / 2.));
190 M2 = Mn2 / (sin(beta - theta));
191 }
192 // Prandtl-Meyer equation V(M)
193 double Prandtl_Meyer::V(double Mach) {
194 return sqrt(((gamma + 1.) / (gamma - 1.))
195 * atan(sqrt(((gamma - 1.) / (gamma + 1.) * (Mach * Mach - 1.)))
196 - atan(sqrt(Mach * Mach - 1.)));
197 }
198 void Prandtl_Meyer::Stagnation_Pressure() { // isentropic
199 // relation to get P0 from P
200 P0 = P *
201 pow((1. + (gamma - 1.) / 2. * M * M),
202 (gamma / (gamma - 1.)));
203 }
204 void Prandtl_Meyer::Pressure_Oblique() {
205 P2 = P
206 * (1. + (2. * gamma) / (gamma + 1.))
207 * (pow(M * sin(beta), 2) - 1.);
208 }
209 void Prandtl_Meyer::Temperature_Oblique() {
210 T2 = T * (1. + (2. * gamma) / (gamma + 1.) * (pow(M * sin(beta), 2)
211 - 1.))
212 * ((2. + (gamma - 1.) * (pow(M * sin(beta), 2)))
213 / ((gamma + 1.) * pow(M * sin(beta), 2)));
214 }
215 void Prandtl_Meyer::P0_Oblique() {
216 P02 = P0 * 2.
217 / ((gamma + 1.) *
218 pow(gamma * M * sin(beta) * M * sin(beta) - ((gamma - 1.)
219 / 2.),
220 1. / (gamma - 1.)))
221 * pow(pow(((gamma + 1.) / 2.) * M * sin(beta), 2) /
222 ((1. + ((gamma - 1.) / 2.) * M * sin(beta) * M * sin(beta)),
223 gamma / (gamma - 1.)));
224 }
225 void Prandtl_Meyer::calc_Oblique() {
226 Newton();
227 }
228 calc_M2_oblique();
229 Pressure_Oblique();
230 Temperature_Oblique();
231 Stagnation_Pressure();
232 P0_Oblique();
233 }
234 void Prandtl_Meyer::P2_Prandtl_Meyer() {
235 P2 = P
236 * pow((1. + ((gamma - 1.) / 2.) * M * M)
237 / (1. + ((gamma - 1.) / 2.) * M2 * M2),
238 (gamma) / (gamma - 1.));
239 }
240 void Prandtl_Meyer::T2_Prandtl_Meyer() {
241 T2 = T
242 * (1. + ((gamma - 1.) / 2.) * M * M)
243 / (1. + ((gamma - 1.) / 2.) * M2 * M2);
244 }
245 void Prandtl_Meyer::mu_Prandtl_Meyer() {
246 mu1 = asin(1. / M);
247 mu2 = asin(1. / M2) - theta;
248 }
249 void Prandtl_Meyer::calc_Prandtl_Meyer() {
250 Newton(); // calculate M2 using Prandtl-Meyer
251 // equations and a Newton solver iterative method
252 P2_Prandtl_Meyer(); // Calculate P2 using Prandtl-Meyer
253 // equations
254 T2_Prandtl_Meyer(); // Calculate T2 using Prandtl-Meyer
255 // equations
256 mu_Prandtl_Meyer(); // Calculate mu values before and
257 // after using Prandtl-Meyer equations
258 }
259 double Prandtl_Meyer::Thrust(double conical_nozzle_theta,
260 double Pamb) { // Thrust as a function of conical nozzle
261 // angle (degrees) and pressure at altitude
262 // std::cout << conical_nozzle_theta << " " << Pamb << " " << P0 << "
263 // " << P << " " << gamma << " " << A_star << " " << std::endl;
264 return ((1. + cos(conical_nozzle_theta * M_PI / 180.)) / 2.)
265 *
266 gamma * P0 * A_star
267 *
268 sqrt(
269 (
270 2.
271 /
272 (gamma - 1.)
273 )
274 *
275 (
276 pow(2. / (gamma + 1.),
277 (gamma + 1.) / (gamma - 1.))
278 )
279 )
280 *
281 sqrt(
282 (
283 1.
284 -
285 pow(P / P0,
286 (gamma - 1.) / gamma)
287 )
288 )
289 +
290 A * (P - Pamb)
291 ;
292 }
293 #endif
294
295 #ifndef CYLINDRICALPORT_H
296 #define CYLINDRICALPORT_H
297 #include <cmath>
298 #include "CylindricalPort.h"
299

```

```

6 CylindricalPort::CylindricalPort()
7 {
8     L0          = .035;      // m
9     D0          = .066;      // m
10    d0          = .03;       // m
11    rho_propellant = 1260.;    // kg/m^3
12    A_star       = .0001887;  // m^2
13    A_exit       = 4.*A_star;  // m^2
14    theta_exit   = 20.*M_PI/180.; // radians
15    gamma        = 1.18;      //
16    Mw           = 23.;       // kg/kg-mol
17    Rg           = 8.314/(Mw/1000.);
18    T0           = 2900.;     // K
19    n            = 0.16;
20    a            = 0.00132/pow(1000.,n); // m
21                // (s*Pa^n)
22    M_crit       = 0.3;
23    k            = 0.2;
24    N            = 3;
25 }
26
27
28
29
30 #endif
31
32 #ifndef RP_H
33 #define RP_H
34 #include <cmath>
35 #include "CylindricalPort.h"
36 #include "RP.h"
37
38 RP::RP() { // default constructor
39     r=0.015; // m
40     r_dot=0.0000001; // dm/ds
41     P0=0.0001; // Pa
42     P0_dot=0.0000001; // dPa/ds
43 }
44 // double RP::A_burn (CylindricalPort a) { return 2.*M_PI*r*
45     a.L0; }
46 double RP::A_burn (CylindricalPort a,int b) {
47     if (b == 0) return 2.*M_PI*r*a.L0;
48     else {
49         // double N=double(b);
50         double s=r-a.d0/2.;
51         return a.N*M_PI*
52             (
53                 (a.D0*a.D0-(a.d0+2.*s)*(a.d0+2.*s))/2.
54                 +
55                 ((a.L0-2.*s)*(a.d0+2.*s))
56             );
57     }
58 }
59 // double RP::V_c (CylindricalPort a) { return M_PI*r*r*a
60     .L0; }
61 double RP::V_c (CylindricalPort a,int b) {
62     if (b == 0) return M_PI*r*r*a.L0;
63     else {
64         // double N=double(b);
65         double s=r-a.d0/2.;
66         return a.N*M_PI/4.
67             * ((a.d0+2.*s)*(a.d0+2.*s) * (a.L0-2.*s)
68               + a.D0*a.D0*2.*s
69             );
70     }
71 }
72
73 #endif
74
75 #ifndef SOLVER_H
76 #define SOLVER_H
77 #include "CylindricalPort.h"
78 #include "Prandtl_Meyer.h"
79 #include "RP.h"
80 #include "Solver.h"

```

```

7 #include <cmath>
8 #include <iostream>
9
10 Solver::Solver() {
11     t0=0.;
12     t1=0.;
13     bates = 0;
14 }
15 Solver::Solver(CylindricalPort a, RP b, Prandtl_Meyer c,
16     Prandtl_Meyer d) {
17     CP = a;
18     RP0= b;
19     RP1= b;
20     Port = c;
21     Nozzle = d;
22
23     t0=0.;
24     t1=0.;
25
26     bates = 0;
27 }
28
29 void Solver::calc_RP1_dot() {
30     // RP1.r_dot = CP.a * pow(RP0.P0,CP.n)/pow(1000.,CP.n);
31     Port.A = RP0.V_c(CP,bates)/CP.L0;
32     RP1.r_dot = CP.a * pow(RP0.P0,CP.n)
33         * ((
34             1.+CP.k*(Port.M/CP.M_crit))
35           / (1.+CP.k));
36     RP1.P0_dot = (RP0.A_burn(CP,bates) * RP1.r_dot / RP0.V_c(
37         CP,bates) )
38     // RP1.P0_dot = (2. * RP1.r_dot / RP0.r )
39     * (CP.rho_propellant*CP.Rg*CP.T0-RP0.P0)
40     - ((CP.A_star/RP0.V_c(CP,bates))
41       * (RP0.P0)
42       * sqrt(CP.gamma*CP.Rg*CP.T0
43             * pow(2./(CP.gamma+1.), (CP.gamma+1.)/(CP.
44                 gamma-1.))));
45 }
46 void Solver::calc_RP1() {
47     RP1.P0 = RP0.P0 + (t1-t0) * RP1.P0_dot;
48     RP1.r = RP0.r + (t1-t0) * RP1.r_dot;
49 }
50 void Solver::writeout() {
51     // std::cout<<std::endl;
52     if (isnan(RP1.P0)) {
53         std::cerr<<"nan on RP1.P0"<<std::endl;
54         return;
55     }
56     std::cout<<t1<<" "<<RP1.P0_dot<<" "<<RP1.r_dot<<" "<<
57         RP1.P0<<" "<<RP1.r<<" "<<RP1.A_burn(CP,bates)<<" "<<
58         ((CP.D0*CP.D0*M_PI/4.*CP.L0)-RP1.V_c(CP,bates))*CP.
59         rho_propellant<<" "<<((RP1.V_c(CP,bates)-RP0.V_c(CP,
60         bates))*CP.rho_propellant)/(t1-t0)<<" "<<Port.m_dot
61         <<" "<<Nozzle.Thrust(20.,0.)<<" "<<Nozzle.Thrust
62         (20.,0.)/(Port.m_dot*9.81)<<" "<<Port.M<<" "<<" "<<
63         Nozzle.M<<std::endl;
64 }
65 void Solver::reset() {
66     RP0 = RP1;
67     t0 = t1;
68 }
69 void Solver::next_step() {
70     calc_RP1_dot();
71     calc_RP1();
72     Port.P0=RP1.P0;
73     Nozzle.P0=RP1.P0;
74     Port.Newton();
75     Port.Pressure();
76     Port.calc_m_dot();
77     Nozzle.Newton();
78     Nozzle.Pressure();
79     Nozzle.calc_m_dot();
80     writeout();
81     reset();
82 }

```

```

74
75 #endif

Main Program

1 #include "DEFS.h"
2 #include "Prandtl_Meyer.h"
3 #include "CylindricalPort.h"
4 #include "RP.h"
5 #include "Solver.h"
6 int main() {
7
8 /*
9     RP a;
10     CylindricalPort b;
11     Prandtl_Meyer PM, PM2;
12     b.Mw=197.231; // g/mol
13     b.Rg = 8.314/(b.Mw/1000.);
14     b.A_exit=3.8*3.8*M.PI/4.;
15     b.A_star=b.A_exit/7.78;
16     b.gamma=1.02;
17     b.L0=34.57;
18     b.D0=3.66;
19     b.d0=1.7;
20     b.n=0.172;
21     b.a=0.00192/pow(1000., b.n); // m/(s*Pa^n)
22     b.rho_propellant=1760.;
23     a.r=b.d0/2.;
24     b.T0=24000.;
25     b.k=0;
26     cout<<"A_star = "<<b.A_star<<endl;
27     cout<<"A_exit = "<<b.A_exit<<endl;
28     cout<<"launch mass = "<<(b.D0*b.D0*M.PI/4.*b.L0-a.V_c(b
    ))*b.rho_propellant<<endl;
29     PM.A = a.V_c(b,0)/b.L0;
30     PM.A_star=b.A_star;
31     PM.gamma=b.gamma;
32     PM.P0 = a.P0;
33     PM.T0 = b.T0;
34     PM.Rg = b.Rg;
35     PM.M = 0.15; // guess subsonic region
36     PM.set_isentropic(0);
37     PM2=PM;
38     PM2.A= b.A_exit;
39     PM2.M= 2.;
40     Solver S(b,a,PM,PM2);
41     cout<<"launch mass = "<<(b.D0*b.D0*M.PI/4.*b.L0-a.V_c(b
    ,S.bates))*b.rho_propellant<<endl;
42     cout<<"Burn area = "<<(S.RP0.A_burn(S.CP,S.bates))<<
    endl;
43
44
45
46
47     cout.precision(16);
48     S.writeout();
49     // while (S.tl < 125.) {
50     while (1) {
51         S.tl=S.t0+.01;
52         S.next_step();
53         if (S.RP0.r>=S.CP.D0/2.) { break; }
54     }
55
56 /*
57
58 /*
59     // non-erosive vs erosive burning
60     RP a;
61     CylindricalPort b;
62     b.L0=0.35;
63     b.D0=0.076;
64     b.d0=0.03;
65     b.rho_propellant=1314.;
66     b.A_star=0.0001887;
67     b.A_exit=4.*b.A_star;
68     b.gamma=1.2;
69     b.Mw=24.26;

70     b.Rg = 8.314/(b.Mw/1000.);
71     b.T0=2000.;
72     b.M_crit=0.11;
73     b.k=2.25;
74     b.n=0.188;
75     b.a=0.00178/pow(1000., b.n); // m/(s*Pa^n)
76     Prandtl_Meyer Port, Nozzle;
77     Port.A = a.V_c(b,0)/b.L0;
78     Port.A_star=b.A_star;
79     Port.gamma=b.gamma;
80     Port.P0 = a.P0;
81     Port.T0 = b.T0;
82     Port.Rg = b.Rg;
83     Port.M = 0.15; // guess subsonic region
84     Port.set_isentropic(0);
85     Nozzle=Port;
86     Nozzle.A= b.A_exit;
87     Nozzle.M= 2.6;
88     Solver S(b,a,Port,Nozzle);
89
90     cout.precision(16);
91     S.writeout();
92     // S.CP.k=0.;
93
94     while (1) {
95         S.tl=S.t0+.001;
96         S.next_step();
97         if (S.RP0.r>=S.CP.D0/2.) { break; }
98     }
99     */
100
101 /**
102     // Part 1 cylindrical port
103     // initialize parameters
104     RP a;
105     CylindricalPort b;
106     Prandtl_Meyer Port, Nozzle;
107     Port.A = a.V_c(b,0)/b.L0;
108     Port.A_star=b.A_star;
109     Port.gamma=b.gamma;
110     Port.P0 = a.P0;
111     Port.T0 = b.T0;
112     Port.Rg = b.Rg;
113     Port.M = 0.16; // guess subsonic region
114     Port.set_isentropic(0);
115     Nozzle=Port;
116     Nozzle.A= b.A_exit;
117     Nozzle.M= 2.6;
118     Solver S(b,a,Port,Nozzle);
119     S.bates=0; // cylindrical port
120     S.Port.Newton();
121     // solve and output
122     cout.precision(16);
123     // S.CP.k=0.; // non-erosive burn
124     S.writeout();
125     while (1) {
126         S.tl=S.t0+.0001;
127         S.next_step();
128         if (S.RP0.r>=S.CP.D0/2.) { break; }
129     }
130     /**
131     /*
132     // Part 2 bates grain
133     // initialize parameters
134     RP a;
135     CylindricalPort b;
136     Prandtl_Meyer Port, Nozzle;
137     Port.A = a.V_c(b,0)/b.L0;
138     Port.A_star=b.A_star;
139     Port.gamma=b.gamma;
140     Port.P0 = a.P0;
141     Port.T0 = b.T0;
142     Port.Rg = b.Rg;
143     Port.M = 0.16; // guess subsonic region
144     Port.set_isentropic(0);
145     Nozzle=Port;
146     Nozzle.A= b.A_exit;

```

```

147     Nozzle.M= 2.6;
148     Solver S(b,a,Port,Nozzle);
149     S.bates=l; // solve using bates grain
150     S.Port.Newton();
151     // solve and output
152     cout.precision(16);
153     S.CP.k=0.; // non-erosive burn
154     S.writeout();
155     while (1){
156         S.tl=S.t0+.0001;
157         S.next_step();
158         if (S.RP0.r>=S.CP.D0/2.){ break; }
159     }
160 */
161
162 /*
163 // Part 2 bates
164 // initialize parameters
165 RP a;
166 CylindricalPort b;
167 b.T0 = 3300.; // higher values of flame temp
168 Prandtl_Meyer Port,Nozzle;
169 Port.A = a.V_c(b,0)/b.L0;
170 Port.A_star=b.A_star;
171 Port.gamma=b.gamma;
172 Port.P0 = a.P0;
173 Port.T0 = b.T0;
174 Port.Rg = b.Rg;
175 Port.M = 0.16; // guess subsonic region
176 Port.set_isentropic(0);
177 Nozzle=Port;
178 Nozzle.A= b.A_exit;
179 Nozzle.M= 2.6;
180 Solver S(b,a,Port,Nozzle);
181 S.bates=l;
182 S.Port.Newton();
183 // solve and output
184 cout.precision(16);
185 S.CP.k=0.; // non-erosive burn
186 S.writeout();
187 while (1){
188     S.tl=S.t0+.0001;
189     S.next_step();
190     if (S.RP0.r>=S.CP.D0/2.){ break; }
191 }
192 */
193
194
195
196
197 return 0;
198 }

```