MAJOR PROJECT # 1

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

ABSTRACT

A transfer orbit from Low Earth Orbit to Medium Earth Orbit is investigated. Four separate maneuvers are investigated and compared. The numerical algorithm used is also presented and described.

CONTENTS

NOMENCLATURE

a_{LEO} Radius of Low Earth Orbit $\approx 8530km$
a_{MEO} Radius of Medium Earth Orbit $\approx 13200km$
$\bar{X}$ Matrix containing r, ν, V_r, V_ν, and m
r Location in the r-direction
ν Location in the ν-direction
V_r Velocity in the r-direction
V_ν Velocity in the ν-direction
m Mass of satellite

$m_{propellant}$ Mass of propellant used to get into final orbit
$m_{propellant}\big|_1$ Amount of propellant used at first thrust
$m_{propellant}\big|_2$ Amount of propellant used at second thrust
ΔV_1 Change in velocity for first thrust to get into transfer orbit
ΔV_2 Change in velocity for second thrust to get into circular final orbit
ΔV_{tot} Change in velocity for combined thrust to get into circular final orbit
dt Time interval for numerical algorithm

1 INTRODUCTION

There were four sections to this problem statement. The overall goal was achieve a simulation that would produce a satellite's trajectory moving from a_{LEO} to a_{MEO}. A Homann Transfer calculation was used to compare to this trajectory. The following two equations (Eq. 1 and 2)

$$\begin{aligned} F_{thrust} &= 10N \\ I_{sp} &= 2000s \end{aligned} \tag{1}$$

$$\begin{aligned} F_{thrust} &= 2000N \\ I_{sp} &= 270s \end{aligned} \tag{2}$$

1.1 Part a

The parameters in Eq. 1 are for the low thrust burn. This burn is used first to get into the transfer orbit. Once the transfer orbit is reached, we then coast until at the apogee and then use an impulsive calculation using the parameters in Eq. 2 to get into the circular orbit in a_{MEO}.

Additionally, both a trapezoidal rule and the Runge-Kutta integration schemes were applied. The time interval was then altered to see where the algorithm blows up.

The resulting ΔV and $m_{propellant}$ were then compared to other parts of this problem.

1.2 Part b

This is similar to part a, except that both maneuvers were performed impulsively the parameters in Eq. 2 was used both times. This was exactly the Homann Transfer calculation using the impulsive motor. We calculated the ΔV and $m_{propellant}$ and compared these results to part a.

1.3 Part c

This analysis a complete continuous large thrust transfer. Meaning both the first and second thrust was calculated using a continuous motor. Eq. 2 describes the parameters for the motor used. It was shown that the dt value was required to be fairly small in order for the numerical algorithm to accurately predict the satellite's position.

1.4 Part d

This part of the problem statement did not require much change from the calculations in part c. The only change was the time that the second thrust began. This was done by manually altering the time that the coasting section ended the second thrust began. It was aimed so that the total ΔV_2 was centered around the when the orbit would reach the required apogee if it was coasting.

2 NUMERICAL ALGORITHM

In order to program this prediction method, several numerical algorithms were used. A matrix of values describing the orbit and satellite location was created. This was titled $\bar{X}$. The values it contained is shown in Eq. 3.

$$\bar{X} = [r, \nu, V_r, V_\nu, m] \tag{3}$$

This matrix had several equations that was know in order to calculate the time derivative of $\bar{X}$. These equations were used along with the Trapezoidal and Runge-Kutta numerical integration scheme. These schemes provided the next step in the satellite's orbit. The four phases of the orbit were then used in parallel with these integration schemes to yield the orbit transfer for each of the parts of the problem stated above.

The trapezoidal algorithm was analyzed in part a. When the dt value was steadily increased we could see the performance of the algorithm. The steps between each point on the plot got larger, and the values got less accurate. It was found that at high dt the trajectory was not calculated correctly. Fig. 1 shows the resulting trajectory. It can be seen that if the $dt = 500$ we find that the coasting process seems to enter an escape velocity. This is impossible and we instantly know that the numerical algorithm has failed to calculate the correct result for each step.

The code in Appendix A shows the program created to solve this problem.

3 RESULTS

Four separate transfer orbits were described and will be analyzed below.

3.1 Part a

The resultant trajectory is shown in Fig. 1. The large dt value was previously described, but when the code did have a small enough dt value we found the results shown here.

$\Delta V_1 = 1321.034$
$\Delta V_2 = 19.499$
$\Delta V_{tot} = 1340.533$
$m_{propellant} \approx 77.6$

We can see here that the rocket equation does not accurately depict what is happening. This can be seen by comparing these values to the values calculated by the rocket equation in part b. Note that the ΔV_1 and ΔV_2 values are off. But the ΔV_{tot} is fairly accurate still.

3.2 Part b

Doing the Homann Transfer calculations yielded the results shown here.

$\Delta V_1 = 698.829$
$\Delta V_2 = 626.159$
$\Delta V_{tot} = 1324.988$
$m_{propellent} \approx 393.739$

Comparing these results to Part a we find that the impulsive Homann Transfer used more mass propellent, but required less ΔV_{tot}. Thus the Homann Transfer was a more efficient transfer orbit, but the motor used in part a required less mass to achieve the same orbit transfer. No figure is shown here since the equations did not require any numerical algorithm. They only required the used of the Homann Transfer equations.

3.3 Part c

Doing this transfer and the calculations provided the results shown here.

$\Delta V_1 = 705.611$
$\Delta V_2 = 774.191$
$\Delta V_{tot} = 1479.802$
$m_{propellent} \approx 745.531$

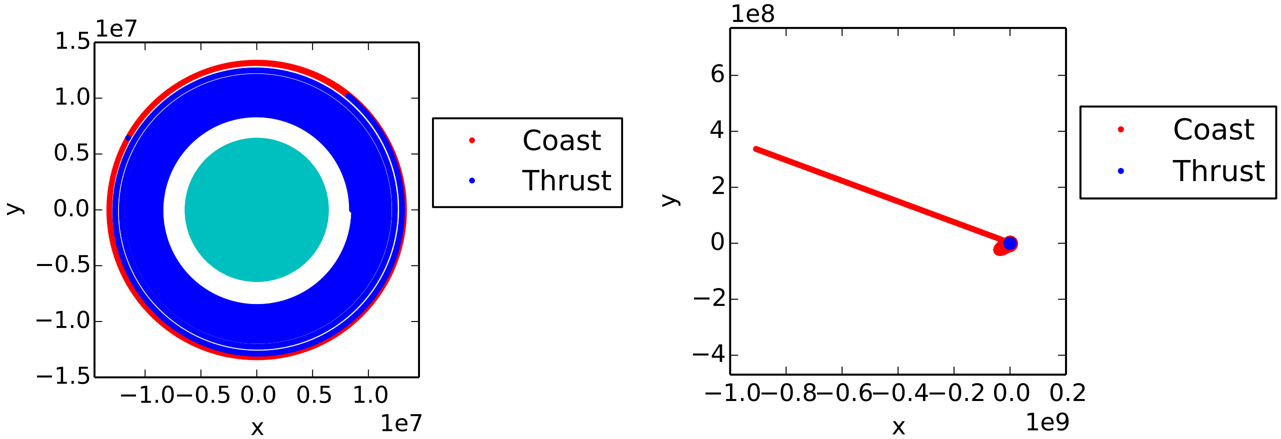

FIGURE 1. **Left:**Plot of trajectory for part a. **Right:** Plot of trajectory for part a when the $dt = 500$.

The trajectory is shown in Fig. 2. We can see here that the trajectory is reached sooner, but it costs more propellant in order to reach the same orbit.

3.4 Part d

Doing this transfer and the calculations provided the results shown here.

$\Delta V_1 = 701.576$
$\Delta V_2 = 627.3218$
$\Delta V_{tot} = 1328.8978$
$m_{propellent} \approx 688.01414$
$m_{propellant}\big|_1 = 406.3272$
$m_{propellant}\big|_2 = 281.68694$

The trajectory is shown in Fig. 2. We can see here that the trajectory is similar to part c, but that it took a little less propellant to reach the orbit. Obviously, it is more efficient to begin the final boost before the apogee is reached.

4 CONCLUSION

The most efficient orbit, in terms of $m_{propellant}$ was part a. This transfer took the least amount of propellant overall. Though, it should be noted that it did take a significantly longer time to reach the orbit. If time was of importance, than another orbit may have been desired. Likely, the orbit transfer in part d would be the most realistic choice if speed was of importance.

Ideally part b would be the fastest transfer in orbit. But it takes an unrealistic impulse in speed to accommodate the transfer. This Homann Transfer in part b is a fairly accurate approximation of the true ΔV values. It should be noted though that the rocket equations are inaccurate when applied to a long duration non-impulsive burn such as in part a. We can see that the rocket equations do not accurately depict what is happening.

A Appendix A: Code

Header files

```
#ifndef DEFS_H
#define DEFS_H
#include <iostream>
#include <vector>
#include <cmath>
#include <string>
#include <sstream>
#include <iomanip>
```

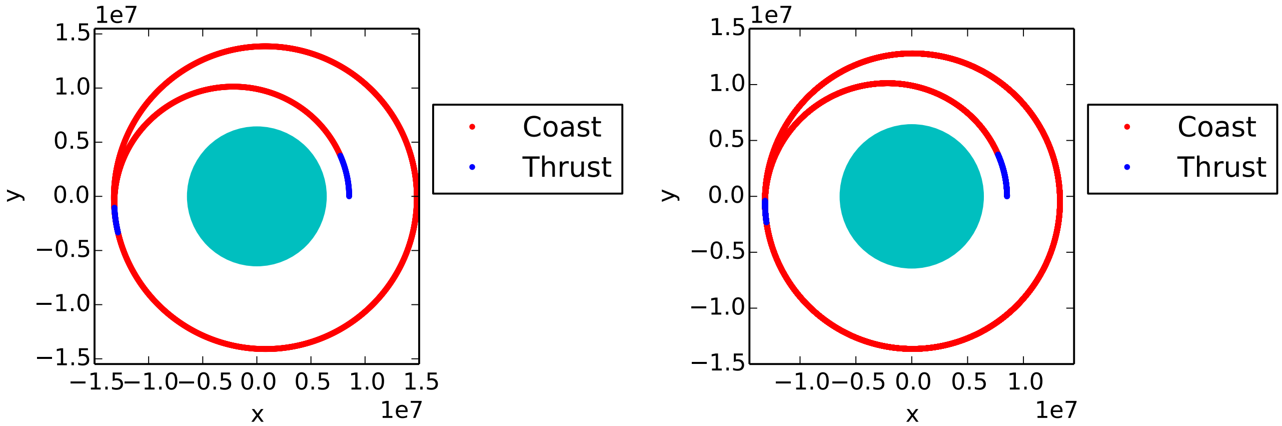

FIGURE 2. **Left:** Plot of trajectory for part c. **Right:** Plot of trajectory for part d.

```cpp
#include <fstream>
using namespace std;
#endif
```

```cpp
#ifndef X_H
#define X_H
#include "DEFS.h"
class X{
    public:
        X(){
            Vr      =   0.;
            Vv      =   0.;
            r       =   0.;
            v       =   0.;
            m       =   0.;
            Vr_dot  =   0.;
            Vv_dot  =   0.;
            r_dot   =   0.;
            v_dot   =   0.;
            m_dot   =   0.;
            t       =   0.;
//          mu      =   3.9860044E14;      // mu for earth
            mu      =   3.9860000E14;      // mu for earth
        }
        // variables
        double Vr, Vv,r,v,m;
        double Vr_dot, Vv_dot,r_dot,v_dot,m_dot;
        double gamma,Isp,mu;
        double Thrust,dt;
        double x,y,t;
        double a,e; // elliptical orbit
        double DeltaV; // accumulated DeltaV value for the whole system

        // functions
        void x_dot(double& F_thrust);
        X Trapz (double& F_thrust,double& dt);
        X RK4   (double& F_thrust,double& dt);
        X function(int a,X& Old_f, double& Thrust_f, double& dt_f);
        double  HomannTransfer(int a,int b,X start, X coast, X final, X Old, X New);
        double  ShootingMethod(int a,int b,X start, X coast, X final, X Old, X New);
        void    HomannTransfer_output(int a,int b,X start, X coast, X final, X Old, X New, ofstream& myfile, ofstream& myfile2);
        void xy();
        void print();
        void output_line(stringstream &a);
};
#endif
```

Linked c++ file

```cpp
#include "X.h"
```

```
#include "DEFS.h"
// calculate the x_dot values using F_thrust and theta
void X::x_dot(double& F_thrust){
    Thrust = F_thrust;
    gamma   = atan(Vr/Vv);
    Vr_dot  = (Vv*Vv/r) - (mu / (r*r))
        + (
                (Thrust/m)
          )
        * sin(gamma);
    Vv_dot  = -((Vr*Vv)/r)
        + (
                (Thrust/m)
          )
        * cos(gamma);
    r_dot   = Vr;
    v_dot   = Vv/r;
    m_dot   = -Thrust/(9.806 * Isp);
}
X X::Trapz(double& F_thrust,double& dt){
    X p,c;    // predictor and correcotr X value initialization
    this->x_dot(F_thrust);    // calculate the current x_dot values
//    double Ei  = -this->mu/(2.*this->a); // initial orbit energy

    // predictor
    p       = *this;
    p.r     = this->r  + dt * this->r_dot   ;
    p.v     = this->v  + dt * this->v_dot   ;
    p.Vr    = this->Vr + dt * this->Vr_dot  ;
    p.Vv    = this->Vv + dt * this->Vv_dot  ;
    p.m     = this->m  + dt * this->m_dot   ;
    p.x_dot(F_thrust)   ;
    // corrector
    c       = *this;
    c.r     = this->r  + dt/2. * (this->r_dot  + p.r_dot)  ;
    c.v     = this->v  + dt/2. * (this->v_dot  + p.v_dot)  ;
    c.Vr    = this->Vr + dt/2. * (this->Vr_dot + p.Vr_dot) ;
    c.Vv    = this->Vv + dt/2. * (this->Vv_dot + p.Vv_dot) ;
    c.m     = this->m  + dt/2. * (this->m_dot  + p.m_dot)  ;
    c.xy();
    c.t     = this->t + dt  ;
//    double Ef  = -c.mu/(2.*c.a); // final orbit energy
    c.DeltaV  = this->DeltaV + 9.806 * c.Isp * log(this->m/c.m);
//    if (abs(Ef-Ei)<1) c.DeltaV = this->DeltaV; // if small DeltaV then do not count DeltaV
    return c;
}
X X::RK4(double& F_thrust, double& dt){
    X k1,k2,k3,k4,c;
    this->x_dot(F_thrust);
//    double Ei  = -mu/(2.*a); // initial orbit energy

    // k1
    k1      = *this;

    // k2
    k2      = k1;
    k2.r    = this->r   + (dt/2.) * k1.r_dot  ;
    k2.v    = this->v   + (dt/2.) * k1.v_dot  ;
    k2.Vr   = this->Vr  + (dt/2.) * k1.Vr_dot ;
    k2.Vv   = this->Vv  + (dt/2.) * k1.Vv_dot ;
    k2.m    = this->m   + (dt/2.) * k1.m_dot  ;
    k2.x_dot(F_thrust);

    // k3
    k3      = k2;
    k3.r    = this->r   + (dt/2.) * k2.r_dot  ;
    k3.v    = this->v   + (dt/2.) * k2.v_dot  ;
    k3.Vr   = this->Vr  + (dt/2.) * k2.Vr_dot ;
    k3.Vv   = this->Vv  + (dt/2.) * k2.Vv_dot ;
    k3.m    = this->m   + (dt/2.) * k2.m_dot  ;
    k3.x_dot(F_thrust);

    // k4
    k4      = k3;
    k4.r    = this->r   + dt    * k3.r_dot  ;
    k4.v    = this->v   + dt    * k3.v_dot  ;
    k4.Vr   = this->Vr  + dt    * k3.Vr_dot ;
```

```
    k4.Vv    = this->Vv  + dt     * k3.Vv_dot ;
    k4.m     = this->m   + dt     * k3.m_dot  ;
    k4.x_dot(F_thrust);

    // corrected values
    c        = *this;
    c.r      = this->r   + (dt/6.) * (k1.r_dot  + 2.*k2.r_dot  + 2.*k3.r_dot  + k4.r_dot );
    c.v      = this->v   + (dt/6.) * (k1.v_dot  + 2.*k2.v_dot  + 2.*k3.v_dot  + k4.v_dot );
    c.Vr     = this->Vr  + (dt/6.) * (k1.Vr_dot + 2.*k2.Vr_dot + 2.*k3.Vr_dot + k4.Vr_dot);
    c.Vv     = this->Vv  + (dt/6.) * (k1.Vv_dot + 2.*k2.Vv_dot + 2.*k3.Vv_dot + k4.Vv_dot);
    c.m      = this->m   + (dt/6.) * (k1.m_dot  + 2.*k2.m_dot  + 2.*k3.m_dot  + k4.m_dot );
    c.xy();
    c.t      = this->t + dt  ;
//   double Ef  = -c.mu/(2.*c.a); // final orbit energy
    c.DeltaV  = this->DeltaV + 9.806 * c.Isp * log(this->m/c.m);
//    if (abs(Ef-Ei)<1) c.DeltaV =this->DeltaV; // if small DeltaV then do not count DeltaV

    return c;
}
X X::function(int a, X& Old_f, double& Thrust_f, double& dt_f) {
    X stuff;
    if      (a == 0) {  return Old_f.Trapz  (Thrust_f, dt_f); }
    else if (a == 1) {  return Old_f.RK4    (Thrust_f, dt_f); }
    else                return stuff;
}
double X::HomannTransfer(int a,int b,X start, X coast, X final, X Old, X New){
    double m; // output mass value
    Old = start;
    Old.x_dot(start.Thrust);
    // continuous thrust
    for (int i=0; i<20000000; ++i){
        New=function(a,Old,start.Thrust,start.dt);
        Old=New;     // reset for next iteration
        if (New.a*(1. + New.e)>final.r) break; // if apogee is the max desired radius
    }
    // coast
    for (int i=0; i<20000000; ++i){
        New= function(a,Old,coast.Thrust,coast.dt);
        Old=New;     // reset for next iteration
        if (New.r>=final.r) break;
    }
    if (b == 0) {
        // if impulsive final burn
        double FinalDV = sqrt(final.mu/final.r) - sqrt(2.*(final.mu/(final.r) - New.mu/(2.*New.a)));
        Old.Vv  += FinalDV;
        Old.m   -= final.m - final.m/(exp(FinalDV/(9.806*final.Isp)));
    }
    else if (b==1){
        // final thrust kick
        for (int i=0; i<2000000; ++i){
            New= function(a,Old,final.Thrust,final.dt);
            Old=New;     // reset for next iteration
            if (New.a*(1. - New.e)>=final.r) break; // if perigee is the max desired radius
        }
    }
    // coast after
    double temp_r = New.v+2.*M_PI;
    double error=10.;
    for (int i=0; i<20000; ++i){
        New= function(a,Old,coast.Thrust,coast.dt);
        error = sqrt(pow(New.v - temp_r,2));// rms error
        if (error < 0.01) {break;}
        Old=New;     // reset for next iteration
    }
    m = Old.m;

    return m;
}

double  X::ShootingMethod(int a,int b,X start, X coast, X final, X Old, X New){
    double mass1,mass2,mass_next;
    X start2;
    mass1   = start.HomannTransfer(a,b,start,coast,final,Old,New);
    start2  = start;
    start2.m += 10.; // add 10 kg for start of this shooting method
    mass2 = start2.HomannTransfer(a,b,start2,coast,final,Old,New);
    mass_next = start.m  + ( (start2.m - start.m) / (mass2 - mass1)) * (final.m - start.m);
```

```
    cout<<"mass guess = "<<start2.m<<" to get final mass = "<<mass2<<endl;
    cout.precision(14);
    double error = 10.;
    for (; error >4.1;){
        start = start2;
        start2.m = mass_next;
        mass1 = mass2;
        mass2 = start2.HomannTransfer(a,b,start2,coast,final,Old,New);
        mass_next = start.m + ((start2.m - start.m)/(mass2 - mass1)) * (final.m - mass1);
        error = abs(final.m - mass2);
        cout<<"mass guess = "<<start2.m<<" to get final mass = "<<mass2<<endl;
    }
    return start2.m;
}
void X::HomannTransfer_output(int a,int b,X start, X coast, X final, X Old, X New, ofstream &myfile, ofstream &myfile2){
    stringstream line;
    double error,temp_r;
    Old = start;
    Old.x_dot(start.Thrust);
    Old.output_line(line);
    myfile<<line.str()<<endl;
    // continuous thrust
    for (int i=0; i<200000000; ++i){
        New=function(a,Old,start.Thrust,start.dt);
        New.output_line(line);
        myfile<<line.str()<<endl;
        Old=New;    // reset for next iteration
        if (New.a*(1. + New.e)>final.r) break; // if apogee is the max desired radius
    }
    // coast
    for (int i=0; i<20000000; ++i){
        New= function(a,Old,coast.Thrust,coast.dt);
        New.output_line(line);
        myfile2<<line.str()<<endl;
        Old=New;    // reset for next iteration
        if (New.r>=final.r) break;
    }
    if (b == 0) {
        // if impulsive final burn
        double FinalDV = sqrt(final.mu/final.r) - sqrt(2.*(final.mu/(final.r) - New.mu/(2.*New.a)));
        Old.Vv  += FinalDV;
        Old.m   -= final.m - final.m/(exp(FinalDV/(9.806*final.Isp)));
        cout<<"final time = "<<New.t<<endl;
        cout<<"final mass = "<<Old.m<<endl;
        cout<<"Final DV = "<<FinalDV<<endl;
        cout<<"a and e = "<<New.a<<" "<<New.e<<endl;
        Old.output_line(line);
        myfile<<line.str()<<endl;
        // end if impulsive final burn
    }
    else if (b == 1) {
        // final non-impulsive thrust kick
        for (int i=0; i<20000000; ++i){
            New= function(a,Old,final.Thrust,final.dt);
            New.output_line(line);
            myfile<<line.str()<<endl;
            Old=New;    // reset for next iteration
            if (New.a*(1. - New.e)>=final.r) break; // if perigee is the max desired radius
        }
    }
    // coast after
    temp_r = New.v+2.*M_PI;
    for (int i=0; i<20000; ++i){
        New= function(a,Old,coast.Thrust,coast.dt);
        error = sqrt(pow(New.v - temp_r,2)); // rms error
        cout<<'\r'<<"Coasting iteration = "<<i<<" now at "<<New.r<<" "<<New.v;
        if (error < 0.01) {break;}
        New.output_line(line);
        myfile2<<line.str()<<endl;
        Old=New;    // reset for next iteration
    }
    cout<<endl;
    cout<<"Delta V = "<<Old.DeltaV<<endl;
    cout<<"Now V vs start.vv = "<<Old.Vr<<" "<<Old.Vv<<" "<<start.Vv<<endl;
    cout<<"supposed to be =    "<<sqrt(Old.mu/Old.r)<<endl;

}
```

```
void X::xy(){
    x = r*cos(v);
    y = r*sin(v);
    a = mu/((2.*mu/r) - (Vr*Vr + Vv*Vv));
    e = (r/mu) * sqrt( pow(Vv*Vv - (mu/r),2) + pow(Vr*Vv,2));

}
void X::output_line(stringstream &a){
    a.str(string());    // clear contents
    a<<fixed<<setprecision(12);
    a<<"   "<<t        ;
    a<<"   "<<x        ;
    a<<"   "<<y        ;
    a<<"   "<<this->a;
    a<<"   "<<e        ;
    a<<"   "<<m        ;
    a<<"   "<<DeltaV  ;
}
// print out values
void X::print(){
    cout<<""<<endl;
    cout<<"x       = "<<x       <<endl;
    cout<<"y       = "<<y       <<endl;
    cout<<"a       = "<<a       <<endl;
    cout<<"e       = "<<e       <<endl;
    cout<<"Vr      = "<<Vr      <<endl;
    cout<<"Vv      = "<<Vv      <<endl;
    cout<<"r       = "<<r       <<endl;
    cout<<"v       = "<<v       <<endl;
    cout<<"m       = "<<m       <<endl;
    cout<<"Vr_dot  = "<<Vr_dot  <<endl;
    cout<<"Vv_dot  = "<<Vv_dot  <<endl;
    cout<<"r_dot   = "<<r_dot   <<endl;
    cout<<"v_dot   = "<<v_dot   <<endl;
    cout<<"m_dot   = "<<m_dot   <<endl;
    cout<<"gamma   = "<<gamma   <<endl;
    cout<<"Isp     = "<<Isp     <<endl;
    cout<<"mu      = "<<mu      <<endl;
    cout<<"t       = "<<t       <<endl;
    cout<<""<<endl;
}
```

Main Program

```
#include "DEFS.h"
#include "X.h"
int main(){
    X start,coast,final,Old,New; // initialize data classes
    stringstream line;

    // read input file
    ifstream input("input",ios::in);
    string LINE,NAME;
    double MASS,RADIUS,THRUST,ISP,DT;
    while ( getline(input,LINE)){
        if (LINE[0] != '#'){
            istringstream ss(LINE) ;
            ss>> NAME >> MASS >> RADIUS >> THRUST >> ISP >> DT;
            if (NAME == "start") {
                start.m      = MASS;
                start.r      = RADIUS;
                start.Thrust= THRUST;
                start.Isp    = ISP;
                start.dt     = DT;
            }
            else if (NAME == "coast"){
                coast.m      = MASS;
                coast.r      = RADIUS;
                coast.Thrust= THRUST;
                coast.Isp    = ISP;
                coast.dt     = DT;
            }
            else if (NAME == "final"){
                final.m      = MASS;
                final.r      = RADIUS;
                final.Thrust= THRUST;
                final.Isp    = ISP;
```

```
                    final.dt     = DT;
                }
            }
        }
        start.Vv     = sqrt(start.mu/start.r); // initial circular orbit velocity
        start.xy();                // calc xy coordinates from r and nu

/* example
        ofstream myfile,myfile2; // output to file
        myfile.open ("continuous.txt", ios::out | ios::trunc);
        myfile2.open ("coast.txt", ios::out | ios::trunc);
        start.HomannTransfer_output(0,0,start,coast,final,Old,New,myfile,myfile2);
        myfile.close();
        myfile2.close();
*/
/*
        // part a
        cout<<"start.V = "<<start.Vr<<" "<<start.Vv<<endl;
        start.m = start.ShootingMethod(0,0,start,coast,final,Old,New);
        // now make output files
        ofstream myfile,myfile2; // output to file
        myfile.open ("continuous.txt", ios::out | ios::trunc);
        myfile2.open ("coast.txt", ios::out | ios::trunc);
        // output to files
        start.HomannTransfer_output(0,0,start,coast,final,Old,New,myfile,myfile2);
        // close files
        myfile.close();
        myfile2.close();
*/
/*
        // part b
        // impulsive Dv
        double DV1,DV2;
        DV1 = sqrt(start.mu/start.r)*(sqrt(2.*(1.-(1./((final.r/start.r)+1.))))-1.);
        DV2 = sqrt(start.mu/start.r)*(sqrt(start.r/final.r) - sqrt(2.*(start.r/final.r - 1./(final.r/start.r + 1.))));
        start.m    -= final.m - final.m/(exp((DV1+DV2)/(9.806*final.Isp)));
        cout<<"DV1 = "<<DV1<<endl;
        cout<<"DV2 = "<<DV2<<endl;
        cout<<"End mass and propellent = "<<start.m<<" "<< final.m - final.m/(exp((DV1+DV2)/(9.806*final.Isp)))<<endl;
*/
/*
        // part c
        start.m = start.ShootingMethod(1,1,start,coast,final,Old,New);
        // now make output files
        ofstream myfile,myfile2; // output to file
        myfile.open ("continuous.txt", ios::out | ios::trunc);
        myfile2.open ("coast.txt", ios::out | ios::trunc);
        // output to files
        start.HomannTransfer_output(1,1,start,coast,final,Old,New,myfile,myfile2);
        // close files
        myfile.close();
        myfile2.close();
*/
// /*
        // part d
        // custom HomannTransfer equation from X.cpp in order to get customized orbit transfer
        ofstream myfile,myfile2; // output to file
        myfile.open ("continuous.txt", ios::out | ios::trunc);
        myfile2.open ("coast.txt", ios::out | ios::trunc);
        double error,temp_r;
        Old = start;
        Old.x_dot(start.Thrust);
        Old.output_line(line);
        myfile<<line.str()<<endl;
        // continuous thrust
        for (int i=0; i<200000000; ++i){
            New=New.function(1,Old,start.Thrust,start.dt);
            New.output_line(line);
            myfile<<line.str()<<endl;
            Old=New;     // reset for next iteration
            if (New.a*(1. + New.e)>final.r) break; // if apogee is the max desired radius
        }
        // coast
        for (int i=0; i<20000000; ++i){
            New= New.function(1,Old,coast.Thrust,coast.dt);
            New.output_line(line);
            myfile2<<line.str()<<endl;
```

```
        Old=New;      // reset for next iteration
        if (New.r>=final.r) break;
        if (New.t>=5400.) break;
    }
    // final non-impulsive thrust kick
    for (int i=0; i<20000000; ++i){
        New= New.function(1,Old,final.Thrust,final.dt);
        New.output_line(line);
        myfile<<line.str()<<endl;
        if (Old.e - New.e < 0.) break;  // break if increasing e
        Old=New;      // reset for next iteration
//          if (New.a*(1. - New.e)>=final.r) break; // if perigee is the max desired radius
    }
    // coast after
    temp_r = New.v+2.*M_PI;
    for (int i=0; i<20000; ++i){
        New= New.function(1,Old,coast.Thrust,coast.dt);
        error = sqrt(pow(New.v - temp_r,2));// rms error
        cout<<'\r'<<"Coasting iteration = "<<i<<" now at "<<New.r<<" "<<New.v;
        if (error < 0.01) {break;}
        New.output_line(line);
        myfile2<<line.str()<<endl;
        Old=New;      // reset for next iteration
    }
    cout<<endl;
    cout<<"Delta V = "<<Old.DeltaV<<endl;
    cout<<"Now V vs start.vv = "<<Old.Vr<<" "<<Old.Vv<<" "<<start.Vv<<endl;
    cout<<"supposed to be =     "<<sqrt(Old.mu/Old.r)<<endl;
// */
    return 0;
}
```