Authors: Nick Havlicek (nhavlicek@wisc.edu (mailto:nhavlicek@wisc.edu)), Shunzhang Li (sli568@wisc.edu (mailto:sli568@wisc.edu))

Adapted from MATLAB code written by Brandon Wilson for the purpose of altitude prediction given a set of physical parameters. This rewrite aims to add tank fitting and sensitivity analysis.

The original matlab code dependes on trial and error inputs to choose suboptimal, but realistic, tank dimensions. This rewrite looks to optimize these suboptimal dimensions, as well as performing sensitivity analysis on parameters relevant to rocket development.

Introduction

This notebook alters preexisting matlab code originally written to calculate a rough estimate of predicted altitude of a single stage, bi-propellant, liquid rocket. The code was originally written by a member of the UW-Madison club called WI Space Race. The goal of WI Space Race is to launch a rocket as high as possible for the Dollar Per Foot Challenge (https://friendsofamateurrocketry.org/dpf-challenge/ (https://friendsofamateurrocketry.org/dpf-challenge/)). In order to maximize the prize money, the rocket needs to be built to achieve maximum altitude. The goal of this project is to use Julia to optimize certain parameters of the rocket in order to help WI Space Race achieve their goal.

Problem to be solved

For a single stage rocket to reach the highest altitude it can, the rocket engine must produce the highest thrust it can for the longest duration possible. These challenges are achieved by maximizing the amount of propellant in the rocket. Two separate tanks of cryogenic liquid oxygen and pressurized Kerosene (https://en.wikipedia.org/wiki/RP-1 (https://en.wikipedia.org/wiki/RP-1)) must hold an optimal amount of fuel such that when they are ejected out the engine, they both run out at the same time. For this to be accomplished, the flow of fuel out of the tanks must be calculated precicely, and the tanks must be sized accordingly.

At the time of creation of this project, WI Space Race has a pre-existing rocket structure that tanks must be fitted inside. This means that the radius of the tanks is determined, and the length of the chamber for the tanks has a constraint later to be chosen. For more information, see the tank section of the Preliminary Design Report. This is a complex engineering problem, and this project ignores many considerations that will eventually be made considering the tanks.

This problem was recognized by Nick when he came across altitude prediction code written in matlab. This code was run multiple times, changing numbers until a reasonable tank volume was achieved. This immediatly jumped out as an optimization problem where instead of trial and error, Julia could be used to find an optimal solution to this problem. Since matlab is very similar to Julia, the code could be reasonable translated and results could be produced.

This code will be used in the future of WI Space Race to find optimal tank sizes for any future rockets to be built, with the final goal of reaching an altitude of 100,000 km (62 miles), aka the Karman Line. UW-Madison would be the first student group to send a rocket past the Karman Line in history. The club hopes this sparks interest in amature rocket scientists within UW-Madison.

Model Creation Process

In the preliminary project report feedback, Bainian asked for a simplified description. This is an incredibly difficult thing to do, because only the math itself is the only way completely understand this code. Any abstraction must come from an optimization standpoint. Below are descriptions of the process that went into developing the JuMP models, and the relevant variables, constraints, and objectives. The scope that they are described in are at the level that I feel comfortable explaining without error. I understand how important an accurate math model is, but the structure of the code was determined beforehand by somebody that is not in this group project. Every line is justifiable, but for the purpose of this project, should be trusted at face value. The descriptions of the models, and the code itself, should make sense without any of the math that happens inbetween. The output is explained in context and analysis is performed relative to the original objective of the project.

Should this still not be enough, a class such as Rocket Dynamics in Engineering Mechanics and Aeronautics would do it justice. There is so much to digest that this code barely scratches of exactly how tanks are chosen. This code is not meant to perfectly predict the fluid dynamics of the rocket engine, only estimate it. The actual tanks will be sized based on engine performance, which this code will help predict. Therefore only an abstract understanding of what the code predicts is relevant in the future. Going into the math would help you understand some of the decisions that need to be made eventually within WI Space Race, but for this code they are only used to obtain estimates for constraints.

Assuming that is not enough for blind faith, next we will give it a go with a simplified description.

Information was gathered with the help of Lucy Fitzgerald and Finn Roberts, the propulsion team lead and the structures team lead respectively. Two files have also been attached. The Propulsion Theory power point written by John McShane and the Preliminary Design Report written by WI Space Race. The PDR has all the info on how the rocket is being built. This can be used to verify how constraints are identified. d

Be warned, it is long.

The variables

When choosing the variables to represent in this code, it must consider the order of difficulty of measurement during engine testing. By this, we mean what things are easily varied during testing that we can get meaningful data from. The test stand that the engine will be attached to will have sensors on it that measure engine thrust, which we can use to measure the efficiency of the engine design. This is what the Propulsion Team is interested in maximizing. Maximum thrust is theoretically achieved by maximizing mass flow of the propellant at a specific ratio defined by the chemistry of the fuel. This maximization is proven in the Propulsion Theory power point. During testing, engine performance will be analyzed, and the efficiency will be used to choose tank sizes. In the code, this efficiency number is the ox:fuel ratio (theoretical optimum is 2.56:1 ox:kerosene). This number will be changed, and the tank sizing model will be run with structural constraints on the tanks, such as length and radius. This is where the Structures Team becomes interested.

Structures is concerned with, you guessed it, the structure of the rocket. They are responsible for designing the "skeleton" of the rocket that gives it its structural integrity, and ensuring that it can handle the stress of moving faster than the speed of sound. An existing small rocket skeleton already exists, however exactly how the tanks will fit has not been decided. The chamber for the tanks has very real constraints, but these constraints are substituted with simpler values for the sake of demonstration of the validity of the code. Precise measurements will be made further in the design phase. Exact constraints will be decided by taking into consideration many challenges that cannot be addressed by this code. Many challenges involve other areas of engineering: temperature gradients of pressurant and cryogenic oxygen, radial force on propellant due to rotation of the rocket in flight, expansion of tanks due to temperature increases. All of these problems must be addressed as WI Space Race gathers data. After all, the Saturn V wasnt designed in a day.

The constraints

Identifying constraints is a real challenge. It involves a deep understanding of how the parts of the rocket function in practice. Lucy Fitzgerald and Finn Roberts are just two of the many brilliant students trying to make this rocket operational. This project was made possible by Brandons existing code, and the calculations done by him to create this code probably took five times longer than for me to translate it, maybe even 10 times. Optimization constraints were originally identified by asking Brandon himself.

"I chose a mixture ratio and chamber pressure based on combustion performance and chamber temperature and solved for everything else for a given amount of propellant and varied the propellant amount until a reasonable tank volume was achieved."

- Brandon on old matlab code

This was exactly what the project set out to optimize. Realizing that this problem would lend itself perfectly to this class, we set out translating the matlab code to julia for optimization. After translation was complete, we identified existing variables that needed to modified for JuMP solvers, and created models that would complete our objective. The two models that will be explained below are created with this quote in mind. The first model returns optimal tank sizing based on optimum rocket engine performance. The second two models are created for sensitivity analysis of mass flows of the two propellants. It analyzes how small changes in the mass flow out of a tank effects the pressure changes. These pressure changes are important to the performance of the rocket, as well as for structural concerns. The analysis is only based on what this project members assume they mean from our limited understanding of the problem itself.

Other succinct explainations of the models are given below.

The objectives

The objectives are easier to understand, and hence are the large focus of the analysis. Optimizing the size of the two tanks, given a made up set of rocket structure parameters and engine performance, is the overall goal of this project. However, multiple JuMP models are created in order to accomplish this. These individual models are explained in the next cell.

The parameters

Rocket parameters are either obtained from the physically existing small-scale rocket that WI Space Races has, or are estimated using physics equations governing rocket dynamics, fluid flow, etc. The combination of the Propulsion Theory power point and the Preliminary Design Report are excellent starting points to understanding why certain parameters exist in the first place. The parameters were chosen for this project report as approximations for real world data. Since this rocket is currently being built, a lot of the data is subject to change. Therefore the exact numbers don't mean that much, other than for creating a resulting output that falls in a realistic range for how the rocket might actually turn out. All of this data is defined in the code as variables, then the models take these variable definitions and use them as constraints. This allows for quick and easy alteration of the code.

Notes on the models

The important thing to note is that mixture ratio, chamber pressure, combustion performance, and chamber temperature are all theoretically choosen based on the chemistry of the fuel. Each of these will be determined as WI Space Race tests their rocket as it is being built. How these numbers are calculated were calculated before the original code was written. Equations in which these numbers are used are justified in the attached Propulsion Theory power point, but understanding is not necessary to understand the optimization model.

As for the model, the amount of propellant is constrained by the tank sizes, so reasonable constraints have been chosen. Tank pressures, mass flows, and burn times are all dependent on the objective and variable values, and are thus recalculated before results are output.

A second model is created for sensitivity analysis of important variables in regards to tank operating pressures, which will be of interest when purchasing fuel tanks that are strong enough to contain the fuel without exploding.

These two models will give very useful information to the engineering team that will be purchacing the fuel tanks and designing the rocket. It will also be useful for engine testing, as it allows us to understand the impact that engine performance has on optimal tank specifications.

This code is so long because calculating all of these parameters requires a very broad knowledge of rocket propulsion and dynamics to fully understand what is happening. The mathematical model is derived from most of the Propulsion Theory powerpoint created by John McShane in 2019. Since the math takes up an entire 44 slide powerpoint, it seems counterproductive to rewrite it. Instead, the justification for each model will be written out in words, and it will substitute math for an understanding of why each model is important to WI Space Race.

Models

Model 1: tank

Variables

```
Amount of propellant: mass of liquid oxygen + mass of Kerosene (the combination of which is called RP-1)
    (https://en.wikipedia.org/wiki/RP-1)
```

Constraint

```
Tank length: length of the oxygen tank + length of the Kerosene tank
```

Objective

```
Maximize amount of propellant: more propellant = longer burn time = higher altitude achieved. This is only true to a
certain extent. "The reason multi-stage rockets are required is the limitation the laws of physics place on the maximum
velocity achievable by a rocket of given fueled-to-dry mass ratio." (https://en.wikipedia.org/wiki/Multistage_rocket)
This is not taken into consideration in this code because we are nowhere near needing a multistage rocket, which is
not proven in this code.
```

This model is capable of returning optimum tank dimensions for specific rocket parameters. Any changes in rocket design only need to be made in the definitions of the specifications in the code in order for this model to return a corrected measuremet. This model is the main feature that will be used immediatly in the design process of the internal structure of the rocket.

Model 2 & 3: OxygenTankPressure, FuelTankPressure

Variables

```
mass flow of oxidizer: the amount of the oxidizer that flows through the engine per time
mass flow of Kerosene: the amount of Kerosene that flows through the engine per time
ox:fuel ratio: ratio of mass flows of the propellant. Since it's the flow rate that matters, the tanks do not share
    this ratio in an obvious manner
```

Constraint

```
Constraints based on fluid dynamics, and thus is formally explained in the Propulsion Theory power point
```

Objective

```
Maximize tank pressure
```

This model allows us to perform sensitivity analysis on the chosen variables. More variables can be added if deemed important by WI Space Race, but things like feed line length are arbitrarily defined because they will not be chosen for a long time. This analysis is mostly to demonstrate the capability and functionality of sensitivity analysis within the code, and not because these specific variables are the most important to rocket design. As engine design moves further along, sensitivity will be performed at the direction of head engineers.

Solution

Without using extensions, the code must be organized in a difficult to follow manner. It is written so that a simple "run all cells" will compile. Each block has a description of what it does, and most are function definitions. If they don't immediatly make sense, follow each function call starting near the bottom. It is a mess, but without extensions there is hardly a better way.

In [275]:

```julia
using Pkg
Pkg.add("JuMP")
Pkg.add("LinearAlgebra")
Pkg.add("DifferentialEquations")
Pkg.add("Gurobi")
Pkg.add("Ipopt")
Pkg.add("PyPlot")
using JuMP, LinearAlgebra, DifferentialEquations, Gurobi, Ipopt, PyPlot
```

```
Resolving package versions...
 Updating `C:\Users\Nick\.julia\environments\v1.3\Project.toml`
[no changes]
 Updating `C:\Users\Nick\.julia\environments\v1.3\Manifest.toml`
[no changes]
Resolving package versions...
 Updating `C:\Users\Nick\.julia\environments\v1.3\Project.toml`
[no changes]
 Updating `C:\Users\Nick\.julia\environments\v1.3\Manifest.toml`
[no changes]
Resolving package versions...
 Updating `C:\Users\Nick\.julia\environments\v1.3\Project.toml`
[no changes]
 Updating `C:\Users\Nick\.julia\environments\v1.3\Manifest.toml`
[no changes]
Resolving package versions...
 Updating `C:\Users\Nick\.julia\environments\v1.3\Project.toml`
[no changes]
 Updating `C:\Users\Nick\.julia\environments\v1.3\Manifest.toml`
[no changes]
Resolving package versions...
 Updating `C:\Users\Nick\.julia\environments\v1.3\Project.toml`
[no changes]
 Updating `C:\Users\Nick\.julia\environments\v1.3\Manifest.toml`
[no changes]
Resolving package versions...
 Updating `C:\Users\Nick\.julia\environments\v1.3\Project.toml`
[no changes]
 Updating `C:\Users\Nick\.julia\environments\v1.3\Manifest.toml`
[no changes]
```

############## Universal Constants ###################

In [276]:

```
g = 9.807;                # gravitational constant [m/s^2]
R_bar = 8314.3;           # universal gas constant [J/kmol-K]
kg2lbm = 2.20462;         # kg to lbm conversion
lbm2kg = 0.453592;        # lbm to kg conversion
Pa2psi = 0.000145038;     # Pascal to psi conversion
psi2Pa = 6894.76;         # psi to Pascal conversion
m2ft = 3.28084;           # meter to foot conversion
ft2m = 0.3048;            # foot to meter conversion
in2m = 0.0254;            # inch to meter conversion
kN2lbf = 224.81;          # kN to lbf conversion
```

newtzero: This function was written by Matt Fig in matlab and has been modified for julia. It is called in GetMassFlowAndNozzleDimensions().

In [277]:

```
function newtzero(f, xr)
    mx = 100 # max iterations
    tol = 1e-13 # tolerance for convergance

# NEWTZERO finds roots of function using unique approach to Newton's Method.
# May find more than one root, even if guess is way off.  The function f
# should be an inline object or function handle AND capable of taking
# vector arguments.  The user can also pass in a string function.
# NEWTZERO takes four arguments. The first argument is the function to be
# evaluated, the second is an optional initial guess, the third is an
# optional number of iterations, and the fourth is an absolute tolerance.
# If the optional arguments are not supplied, the initial guess will be set
# to 1, the number of iterations will be set to 30, and the tolerance will
# be set to 1e-13.  If the initial guess is a root, no iterations will take
# place.
#
#
# Author:  Matt Fig
# Contact:  popkenai@yahoo.com

    if abs(f(xr))< tol
        global root = xr; # The user supplied a root as the initial guess.
        return  # Initial guess correct.
    end

    # apparently log ranges are harder than could ever be imagined in julia https://github.com/JuliaLang/julia/pull/25896
        #~NH~
    logrange(x1, x2, n) = (10^y for y in range(log10(x1), log10(x2), length=n))
    density = 220
    LGS = collect(logrange(0.00001,3,density)); # Can be altered for wider range or denser search.

    startvar = 0
    stopvar = 18/19
    LNS = range(startvar, stop=(stopvar), step=((stopvar-startvar)/(density-1))); # Search very close to initial guess too
        #sorry this is so brutal but matlab somehow handles line 81 if the 4 columns are different lengths, idk how.
    xrLGS = xr.*(ones(length(LGS),1))
    xrLNS = xr.*(ones(length(LNS),1))
    xr = transpose([xr.-LGS xr.+LGS xr.-LNS xr.+LNS]);  # Make vector of guesses.
    iter = 1;  # Initialize the counter for the while loop.
    mn1 = .1; # These will store the norms of the converging roots.
    mn2 = 1; # See last comment.
    sqrteps = sqrt(eps(Float64)); # Used to make h.  See loop.

    while iter <= mx && abs(mn2-mn1) > 5*eps(Float64)
        h = sqrteps.*xr; # From numerical recipes, make h = h(xr)
        xr = @. xr.-f(xr)./((f(xr.+h).-f(xr.-h))./(2*h)); # Newton's method.
        xr[isnan.(xr)] = []; # No need for these anymore.
        xr[isinf.(xr)] = []; # No need for these anymore.
        mn1 = mn2; # Store the old value first.
        mn2 = norm(xr); # This could make the loop terminate early!
        iter = iter+1;  # Increment the counter.
    end


    if abs(f(0)) < tol # The above method will tend to send zero root to Inf.
        xr = [xr;0];  # So explicitly check.
    end

    # Filtering.  We want to filter out certain common results.
    idxi = abs.(imag.(xr)) .< 5e-15;  # A very small imag term is zero.
    xr[idxi] = real.(xr[idxi]);  # Discard small imaginary terms.
    idxr = abs.(real.(xr)) .< 5e-15;  # A very small real term is zero.
    xr[idxr] = complex.(0,imag.(xr[idxr])); # Discard small real terms.
    root = xr[abs.(f.(xr)) .< tol]; # Apply the tolerance.

    # Next we are going to delete repeat roots.  unique does not work in
    # this case because many repeats are very close to each other but not
    # equal.  For loops are fast enough here, most root vectors are short(ish).

    if ~isempty(root)
        cnt = 1;  # Counter for while loop.
        rt = []
        while ~isempty(root)
            vct = abs.(root .- root[1]).<5e-6; # Minimum spacing between roots.
            C = root[vct];  # C has roots grouped close together.
            idx = argmin(abs.(f.(C)));  # Pick the best root per group.
            append!(rt,C[idx]);  # #ok<AGROW>  Most root vectors are small.
            root = root[.~vct]; # Deplete the pool of roots.
            cnt = cnt + 1;  # Increment the counter.
        end
    root = transpose(sort(rt));  # return a nice, sorted column vector
    end

    return root
end
;
```

GetMassFlowAndNozzleDimensions Function returns mass flow based on initial thrust

In [278]:

```julia
function GetMassFlowAndNozzleDimensions(F_T_0)

    epsilon = expansionRatio;
    P0 = P0_eng;
    gamma = gamma_prop;
    T0 = T0_eng;
    R = R_bar/M_bar_prop;
    P_atm = 101.3e3; # atmospheric pressure [Pascals]

    # calculate Me from expansion ratio. See pg 28 of Propulsion Theory pp
    Ae_Astar_expr = Me -> 1/Me.*(2/(gamma+1).*(1+(gamma-1)./2*Me.^2)).^((gamma+1)./(2*(gamma-1))) - epsilon;
    global Me = newtzero(Ae_Astar_expr, 1);

    # 2 solutions, only want supersonic
    if(length(Me) == 2)
        Me = Me[2];
    end
    # get exit pressure and temp
    Pe = P0*(1+(gamma-1)/2*Me^2)^(gamma/(1-gamma));
    global Pe_eng = Pe;
    Te = T0*(1+(gamma-1)/2*Me^2)^(-1);

    # solving for throat area and m_dot
    rho0 = P0/(R*T0);
    rho_t = rho0*(1+(gamma-1)/2)^(-1/(gamma-1));
    Tt = T0*(1+(gamma-1)/2)^(-1);
    At = F_T_0/((Pe-P_atm)*epsilon+Me*rho_t*gamma*R*sqrt(Tt*Te));
    global m_dot = rho_t*At*sqrt(gamma*R*Tt);

    # exit area, velocity and ISP
    global Ae = epsilon*At;
    ue = Me*sqrt(gamma*R*Te);
    global Isp = ue/9.807;

    return Pe_eng, ue, m_dot
end
;
```

Get_rho_T_P Function returns temperature, pressure and density from an altitude input

In [279]:

```julia
function Get_rho_T_P(alt)
    # Model based on NASA.gov:
    # https://www.grc.nasa.gov/www/k-12/airplane/atmosmet.html

    #atmospheric definitions
    Tropopause = 11000;     # Altitude of initial Tropopause reg [m]
    lowerStrat = 25000;     # Altitude of Lower stratosphere [m]

    # Temp linearly decreases then becomes constant at tropopause
    if(alt > lowerStrat)
        T = -131.21 + 0.00299*alt; #C
        P = 2.488*((T+273.15)/216.6)^(-11.388); # kPa
    elseif(alt > Tropopause && alt < lowerStrat)
        T = -56.46; # C
        P = 22.65*exp(1.73-0.000157*alt); #kPa
    else
        T = 15.04 - 0.00649*alt;    # C
        P = 101.29*((T+273.15)/288.08)^5.256; # kPa
    end

    # Air density as a function of pressure and temp [kg /m^3]
    rho = P/(0.2869*(T+273.15));

    # convert back to SI
    P = P*1000; #Pa
    T = T + 273.15; # K

    return rho,T,P
end
;
```

Get_flow Function returns drag coefficient, Mach and Reynolds #

In [280]:

```julia
function GetFlow(T,rho,V)

    gamma = 1.4;            # Ratio of specific heat for air
    M_bar_air = 28.97;      # kg/kmol
    R = R_bar/M_bar_air;    # Ideal gas constant [j/(Kg-K)]
    c = sqrt(gamma*R*T);    # Speed of sound
    Ma = norm(V/c);         # Mach Number
    mu0 = 18.27e-6;         # Dynamic viscosity at 291.15 K [Pa-s]
    T0 = 291.15;
    C = 120;                # Sutherland constant
    mu = mu0*((T0+C)/(T+C))*(T/T0)^(3/2);   # Dynamic viscosity (Sutherland's eq)
    Re = rho*V*r_veh*2/mu; # Reynold's number

    # approximationg drag coefficient from "rocket ans spacecraft propulsion"
    # from Marting Turner (page 150). Esssentiall Cd is roughly constant in
    # subsonic regime, then exponentially increases to mach 1, then
    # exponentiall decreases.
    a = 0.15;
    b = 0.35;
    if(Ma < 1)
        Cd = a + b*Ma^6;
    else
        Cd = a + b/Ma^2;
    end

    return Cd, Ma, Re
end
;
```

GetFeedPress Function returns tank pressure based on pressure drops, losses, and line diam and mass flow, density, and line length [m]

In [281]:

```julia
function GetFeedPress(lineDiam, P0_eng, catBed_pressDrop, inj_PressDrop, m_dot, rho, L_in, model)

    A = pi/4*lineDiam^2; # area [m^2]

    @variable(model, v)
    @NLconstraint(model, v == m_dot/(rho*A)) # velocity [m/s] 100 in denominator for dual constraint step
    mu = 1.003e-3;  # viscosity (using water)
    @variable(model, Re >= 2200) # non-laminar flow if >= 2200
    @NLconstraint(model, Re == rho*v*lineDiam/mu); # reynolds number
    epsilon = 0.005/1000; # guess for pipe roughness
    L= L_in*in2m; # line length [m]

    @variable(model, x >= 0)
    @NLconstraint(model, -2*log(epsilon/lineDiam/3.7+2.51/(Re*sqrt(x)))-1/sqrt(x) == 0) # some fluid dynamic equation
    # Maximum tank_press
    @NLobjective(model, Max, 2*Pa2psi*(( rho*v^2 )
            + ( P0_eng+catBed_pressDrop+inj_PressDrop )
            + ( rho*x*L/lineDiam*v^2/2)) )
    optimize!(model)

    println("obj_val max tank pressure: " ,objective_value(model))
    f = value(x)
    v = value(v)
    deltaP = rho*f*L/lineDiam*v^2/2;

    backPress = P0_eng+catBed_pressDrop+inj_PressDrop; # pressure at inlet
    return tank_press = rho*v^2 + backPress + deltaP; # pressure

end
;
```

GetPropTankVol Function Returns tank volumes

In [282]:

```julia
function GetPropTankVol(OF, propellantMass,rho_ox, rho_f)

    m_ox = (OF * propellantMass) / (1 + OF);  # mass oxidizer [kg]
    m_f = propellantMass - m_ox;              # mass fuel [kg]
    vol_ox = m_ox / rho_ox;                   # Volume of oxidizer [m^3]
    vol_f = m_f / rho_f;                      # Volume of fuel [m^3]
    vol_ox_boilOff = vol_ox * 0.01;           # boil off volume (rough approx)
    vol_f_boilOff = vol_f * 0;                # non cryo, wont boil off
    vol_ox_ullage = (vol_ox_boilOff + vol_ox) * 0.05;
    vol_f_ullage = (vol_f_boilOff + vol_f) * 0.05;

    vol_ox_total = vol_ox + vol_ox_boilOff + vol_ox_ullage;
    vol_f_total = vol_f + vol_f_boilOff + vol_f_ullage;


    return vol_ox_total,vol_f_total,m_ox,m_f
end
;
```

propTankWt_Size function returns overall tank weight, overall length, thickness of cylinder, thickness of hemispherical ends, thickness of junction, and length of tank

In [283]:
```julia
function propTankWt_Size(vol_total, r_tank, operatingPressure)

    # Parameters are:  outer radius of tank [ft], and operating pressure [psi]
    # Tank size and weight

    rho_Al = 170;                        # Density of 6066 t-6 aluminum, lbm/ft^3
    al_yield = 39000;                    # Yield strength of 6066 t-6 al
    al_ultimate = 45000;                 # Ultimate strength of 6066 t-6 al
    K = 1/0.67;                          # Knuckle factor for stress concentrations

    # Max allowable stress per MIL-STD1522A (USAF)
    maxAllowStress = min(al_yield/1.25, al_ultimate/1.5);

    ###############################################################################
    # Tank Thickness Calcs
    # Required wall thickness at the weld [in]

    t_k = K * operatingPressure * r_tank * 12 / maxAllowStress;

    # Required wall thickness of the hemispheircal ends [in]
    t_cr = operatingPressure * r_tank * 12 / (2 * maxAllowStress);

    # Hemispherical end thickness [in]
    t_hemi = (t_k + t_cr) / 2;

    # Required thickness of cylindrical section [in]
    t_cyl = operatingPressure * r_tank * 12 / maxAllowStress;

    ###############################################################################
    # Tank Sizing Calcs
    # Inner radii of hemi and cyl [ft]

    r_inner = r_tank - t_hemi/12;

    # volume of end cap
    vol_hemi = 2/3 * pi * r_inner^3;

    # Volume required for cylinders
    vol_cyl = vol_total - 2 * vol_hemi;

    # oxider tank length cylinder only
    l_tank_cyl = vol_cyl / (pi * r_inner^2);

    # tank overall Length (inner dimensions)
    tank_length = l_tank_cyl + 2 * r_inner;

    ###############################################################################
    # Weight Calcs
    # Weight of components

    weight_tank_cyl = rho_Al * pi * l_tank_cyl * (r_tank^2 - r_inner^2);
    weight_Tank_Hemis = 2/3 * pi * rho_Al * (r_tank^3 - r_inner^3);

    # Overall weight
    tankWt = weight_tank_cyl + 2 * weight_Tank_Hemis;


    return tankWt,tank_length,t_cyl,t_hemi,l_tank_cyl
end
;
```

Rocket Altitude Predictions main function that calls sub functions and does plotting

In [284]:

```julia
function altitude_prediction(sim_t,R,r_tank,m_struct,m_propellant,
        F_T,epsilon,P0,T0,M_bar,gamma,inj_percentDrop,catBed_pressDrop,
        ox_lineDiameter,f_lineDiameter,ox_fuel_ratio)

############## Defining variables from input ###############
    global m_veh_structure = m_struct;
    global simTime = sim_t;
    global m_prop = m_propellant;
    global P0_eng = P0;
    global T0_eng = T0;
    global M_bar_prop = M_bar;
    global F_T_max = F_T;
    global gamma_prop = gamma;
    global expansionRatio = epsilon;
    global r_veh = R;

############# Initial Calculations ##################

    # mass flow, throat area, and exit area for engine
    massFlowAndNozzleArray = GetMassFlowAndNozzleDimensions(F_T_max)
    global Pe_eng = massFlowAndNozzleArray[1];
    global ue     = massFlowAndNozzleArray[2];
    global m_dot  = massFlowAndNozzleArray[3];

    global eng_time = m_prop/m_dot;  # burn time
    global Pe = Pe_eng

    # Propellant tank calculations
    global rho_ox = 1141; # oxidizer density [kg/m^3]
    global rho_f = 820;   # fuel density [kg/m^3]
    global OF = ox_fuel_ratio; # ox:fuel ratio

    global inj_PressDrop = inj_percentDrop/100*P0_eng; # pressure drop across injector


    ############# Sensitivity analysis ###################

    OxygenTankPressure = Model(with_optimizer(Ipopt.Optimizer, print_level=0))
    @variable(OxygenTankPressure, OFVarOx)
    @constraint(OxygenTankPressure, OxRatioCon, OFVarOx == OF)
    @variable(OxygenTankPressure, m_dot_ox_Var)
    @NLconstraint(OxygenTankPressure, OxVelCon, m_dot_ox_Var == (OFVarOx * m_dot) / (1 + OFVarOx))
    global ox_lineLength = 24; # guess for ox run line length [in]

    # oxidizer tank pressure
    println("\nOxygen Tank")
    global ox_tankPressure = 2*GetFeedPress(ox_lineDiameter*in2m,P0_eng, catBed_pressDrop,inj_PressDrop,m_dot_ox_Var,
    rho_ox,ox_lineLength,OxygenTankPressure)

    println("dual m_dot_ox: ", dual(OxVelCon))
    global m_dot_ox = value(m_dot_ox_Var)
    println("mean ox_tankPressure ", 1/2*ox_tankPressure*Pa2psi)
    println()

    ############

    FuelTankPressure = Model(with_optimizer(Ipopt.Optimizer, print_level=0))
    @variable(FuelTankPressure, OFVarF)
    @constraint(FuelTankPressure, FuelRatioCon, OFVarF == OF)
    @variable(FuelTankPressure, m_dot_f_Var)
    @NLconstraint(FuelTankPressure, FVelCon, m_dot_f_Var == (m_dot - ((OFVarF * m_dot) / (1 + OFVarF))) )
    global f_lineLength = 144; # guess for fuel run line length [in]

    # fuel tank pressure
    println("Fuel Tank")
    global f_tankPressure = 2*GetFeedPress(f_lineDiameter*in2m, P0_eng, catBed_pressDrop*0, inj_PressDrop, m_dot_f_Var,
        rho_f, f_lineLength, FuelTankPressure)
    global m_dot_f = value(m_dot_f_Var)

    println("dual m_dot_f: ", dual(FVelCon))
    println("mean f_tankPressure ", 1/2*f_tankPressure*Pa2psi)
    println()

    ################

    # get tank volumes
    volArray = GetPropTankVol(OF,m_prop,rho_ox,rho_f)
    global vol_ox = volArray[1];
    global vol_f =  volArray[2];
    global m_ox=    volArray[3];
    global m_f =    volArray[4];

    # get ox tank size and weight estimates
    oxArray = propTankWt_Size(vol_ox*m2ft^3,r_tank,ox_tankPressure*Pa2psi)
    global ox_tankWt=        oxArray[1];
    global ox_tank_length =  oxArray[2];
    global ox_t_cyl =        oxArray[3];
    global ox_t_hemi=        oxArray[4];
    global ox_l_tank_cyl =   oxArray[5];
    # get fuel tank size and weight estimates
    fuelArray = propTankWt_Size(vol_f*m2ft^3,r_tank,f_tankPressure*Pa2psi)
    global f_tankWt =        fuelArray[1];
    global f_tank_length =   fuelArray[2];
    global f_t_cyl =         fuelArray[3];
    global f_t_hemi =        fuelArray[4];
    global f_l_tank_cyl =    fuelArray[5];

    @constraint(tank, ox_tank_length + f_tank_length <= 4) # 4 is just a placeholder. Somewhat realistic however.
    @objective(tank, Max, m_prop) # maximize propellant to maximize altitude.
    optimize!(tank)
    global m_prop = objective_value(tank)

######### Redefine everything with m_prop decided #######

    global eng_time = m_prop/m_dot;  # burn time
    global Pe = Pe_eng

    # get tank volumes
    volArray = GetPropTankVol(OF,m_prop,rho_ox,rho_f)
    global vol_ox = volArray[1];
    global vol_f =  volArray[2];
    global m_ox=    volArray[3];
    global m_f =    volArray[4];

    # get ox tank size and weight estimates
    oxArray = propTankWt_Size(vol_ox*m2ft^3,r_tank,ox_tankPressure*Pa2psi)
    global ox_tankWt=        oxArray[1];
    global ox_tank_length =  oxArray[2];
    global ox_t_cyl =        oxArray[3];
    global ox_t_hemi=        oxArray[4];
    global ox_l_tank_cyl =   oxArray[5];
    # get fuel tank size and weight estimates
    fuelArray = propTankWt_Size(vol_f*m2ft^3,r_tank,f_tankPressure*Pa2psi)
    global f_tankWt =        fuelArray[1];
    global f_tank_length =   fuelArray[2];
    global f_t_cyl =         fuelArray[3];
    global f_t_hemi =        fuelArray[4];
    global f_l_tank_cyl =    fuelArray[5];

################################################################


    global tankWt = ox_tankWt + f_tankWt;

    global m_tanks = tankWt*lbm2kg; # propellant tank total

    global m_veh_empty = m_veh_structure+m_tanks;  # empty mass of vehicle [kg]
    global m_veh_wet = m_veh_empty+m_prop;         # wet mass of vehicle [kg]
    global A_veh = pi*r_veh^2;                     # Cross sectional area of vehicle [m^2]
end
;
```

start here for code tracing

Rocket Parameters

```
all numbers may be subject to change, but are chosen realistically by WI Space Race (Brandon Wilson)
```

This cell is what runs all of the above functions and currently outputs any printed optimization results

In [285]:
```julia
tank = Model(with_optimizer(Gurobi.Optimizer, OutputFlag = 0))

m_veh_structure = 40; # dry mass of rocket [kg]

@variable(tank, m_prop)
#m_prop = 20;      # Initial propellant mass, kg
r_veh = 4/12*ft2m; # radius of vehicle [m]
epsilon = 4.5;     # expansion ratio of nozzle
F_T_max = 4e3;     # Thrust when exit pressure = atm pressure [Newtons]
r_tank = 3/12;     # tank radius [ft]

# feed pressure parameters
inj_percentDrop = 20;            # pressure drop across injector [% of chamber press]
catBed_pressDrop = 0*psi2Pa;     # pressure drop across catalyst bed [Pa]
ox_lineDiameter = 0.75-2*0.083; # ox feed line diameter [in]
f_lineDiameter = 0.5-2*0.035;   # fuel feed line diameter [in]

# from NASA Chemical Equilibrium with Applications
# The NASA Computer program CEA (Chemical Equilibrium with Applications) calculates chemical
# equilibrium compositions and properties of complex mixtures.
P0_eng = 2.06e6;     # Initial pressure of engine [Pascals]
T0_eng = 3141;       # Initial temperature of engine [Kelvin]
M_bar_prop = 22.15;  # Molar mass of propellant [kg/kmol]
gamma_prop = 1.14;   # specific heat ratio of propellants (frozen-flow)

rho_ox = 1141;  # oxidizer density [kg/m^3]
rho_f = 820;    # fuel density [kg/m^3]
OF = 2.56;      # ox:fuel ratio
# according to wikipedia, RP-1 (https://en.wikipedia.org/wiki/RP-1) has an optimal ratio of 2.56.


# call altitude prediction model
# calculates variables for the equations of motion
altitude_prediction(simTime,r_veh,r_tank,m_veh_structure,
    m_prop,F_T_max,epsilon,P0_eng,T0_eng,M_bar_prop,gamma_prop,
    inj_percentDrop,catBed_pressDrop,ox_lineDiameter,f_lineDiameter, OF);



# Since this cell calls the functions defined above, this cell will have all output of the functions.
# Any analysis performed will output here.
```

```
--------------------------------------------
Warning: your license will expire in 13 days
--------------------------------------------

Academic license - for non-commercial use only

Oxygen Tank
obj_val max tank pressure: 730.0045862074338
dual m_dot_ox: -21.719609524383966
mean ox_tankPressure 365.0022931037169

Fuel Tank
obj_val max tank pressure: 732.7141674678437
dual m_dot_f: -64.53984652858551
mean f_tankPressure 366.35708373392185


--------------------------------------------
Warning: your license will expire in 13 days
--------------------------------------------

Academic license - for non-commercial use only
```

Sensitivity Analysis

This sensitivity analysis was performed as an example of future analysis that can be performed as seen fit. In the future, this code will likely be modified multiple times to analyze the sensitivity of specific parameters individually. This analysis serves as a demonstration of the capabilities of the language. The analysis presented here may or may not be useful immediatly to WI Space Race.

In this model, we analyze how a change in the mass flow of the propellant changes the operating pressure of the tanks. This might be relevant because tanks must be choosen with appropriate maximum pressure ratings. Tanks with higher pressure ratings might be more expensive or able to contain less fuel. Both of these are prohibitive to the end goal of altitude maximization. The numbers that are computed make sense relative to the parameters chosen. Mass flow only accounts for a portion of the total pressure on the tanks, so a large change in mass flow should not have a very large change on the overall tank pressure. Since the mass flow of the fuel is smaller than the oxidizer, the change in pressure corresponding to the same change in flow has a larger overall effect on the tank pressure. These results make analytic sense, showing that the model is not behaving incorrectly.

Since mass flows are in the single digit kg/s range, the dual variable outputs are large perturbations in the mass flow, on the order of 1 kg/s differences, thus causing large pressure swings shown by dual m_dot_ox and dual m_dot_f.

Adding more constraints and variables will increase the complexity of the model, and would allow for additional sensitivity analysis. However, this will be left up to the club to perform in the future as they see fit.

Altitude Prediction

From here on, the code is not used for optimization. It takes all of the calculations performed above and uses rocket equations to give altitude vs time and velocity vs time data. This data is then graphed below. Were this data to not make sense, code can be rewritten to correct it. No analysis is performed as part of this project because rocket motion is not what is being optimized, it is simply a consequence of optimization.

eom_rocket 2nd order non-linear differential equation for vertical ascent and constant thrust

In [286]:

```julia
function eom_rocket(y,p,t)

    #println(t,", ",y)

    # if instability detected in ODE solver below, uncoment this line and check what number
    # t approaches and choose something with an altitude that is close to zero. While not necessary for the
    # program to complete, it is possible to choose a simulation time that ends before altitude approaches
    # negative infinity. This is only due to the method used to solve the Ordinary Differential Equation


    massFlow = m_dot;
    Pe = Pe_eng;

    # altitude, temp and pressure as a function of altitude
    rhoArray = Get_rho_T_P(y[1])
    rho = rhoArray[1];
    T   = rhoArray[2];
    P   = rhoArray[3];

    # Get Cd, Ma and Re
    flowArray =  GetFlow(T,rho,y[2])
    Cd = flowArray[1];
    Ma = flowArray[2];
    Re = flowArray[3];



    # Force due to drag [N]
    F_d = 0.5* Cd * A_veh * rho * y[2]^2;

    # Thrust is constant piecwise
    if t < eng_time

        # thrust
        F_T = massFlow*ue+(Pe-P)*Ae;

        # rocket mass decreases linearly with m_dot (assumed ~constant m_dot)
        m_veh = m_veh_wet-massFlow*t;

    else
        F_T = 0;                # Thrust = 0 after motor burnout
        m_veh = m_veh_empty;    # Mass is constant after burnout
        massFlow = 0;
    end

    # returned ydot values
    ret = [y[2],F_T/m_veh-F_d/m_veh-massFlow.*y[2]/m_veh-g]

    return ret
end
;
```

Backout performance Uses generated eom_rocket ydot array to compute relevant parameters for output

In [287]:

```
function BackoutPerformance(t,y1,y2)
    # 2nd order non-linear differential equation for vertical ascent and
    # constant thrust

    massFlow = m_dot;
    Pe = Pe_eng;

    y=[y1,y2];      # y1 is altitude, y2 is velocity

    # altitude, temp and pressure as a function of altitude
    rhoArray = Get_rho_T_P(y1)
    rho = rhoArray[1];
    T = rhoArray[2];
    P = rhoArray[3];

    # Get Cd, Ma and Re
    flowArray = GetFlow(T,rho,y[2])
    Cd = flowArray[1];
    Ma = flowArray[2];
    Re = flowArray[3];

    # Force due to drag [N]
    F_d = 0.5* Cd * A_veh * rho * y[2]^2;

    # Thrust is constant piecwise
    if t < eng_time

        # thrust
        F_T = massFlow*ue+(Pe-P)*Ae;


        # rocket mass decreases linearly with m_dot (assumed ~constant m_dot)
        m_veh = m_veh_wet-massFlow*t;

    else
        F_T = 0;                # Thrust = 0 after motor burnout
        m_veh = m_veh_empty;    # Mass is constant after burnout
        massFlow = 0;
    end

    # should it be the top one? mirrors the cell above.
    acc = F_T/m_veh-F_d/m_veh-massFlow.*y[2]/m_veh-g
    # returned acceleration
    #acc = F_T/m_veh-F_d/m_veh-g;

    return acc, F_T, F_d, Ma, Re, rho, Cd

end
;
```

Solving EOM

In [288]:

```julia
x0 = 0.0;
v0 = 0.0;
IC = [x0, v0];

simTime = 100 #CHANGE ME IF TOO LOW

# feel free to mess with simTime to prevent error messages. The solver just keeps going negative.
# Basically the solver never quite reaches a larger simTime because the different iteration values
# have very large changes in yout[1][:] (altitude) for very small time steps
# so the solver tries to compensate by taking even smaller time steps. This is obvious
# if you print t for each iteration in eom_rocket() method
# there is probably a solution for this in the DifferentialEquations package, but this isnt NASA.


t_span = (0.0, simTime) # used in Differential Equations problem statement

# Problem statement
prob = ODEProblem(eom_rocket,IC,t_span);

# Problem Solver using Eulers method
yout = DifferentialEquations.solve(prob,Euler(),dt = 0.1);


# only need data while rocket altitude is greater than or equal to zero
alt =  [] # array for storing altitude of rocket
vel =  [] # array for storing y velocity of rocket
time = [] # time at each index of alt and vel arrays

#For graphing purposes collect data until rocket crashes into ground
altGraph =  [] # array for storing altitude of rocket
velGraph =  [] # array for storing y velocity of rocket
timeGraph = [] # time at each index of alt and vel arrays
for i = 1:length(yout) # all altitude and velocity measurements from diffeq solver
    if(yout[2,i] >= 0) # before apex of trajectory
        # keep track
        append!(alt,  yout[i][1]);
        append!(vel,  yout[i][2]);
        append!(time, yout.t[i]);
        append!(altGraph, yout[i][1]);
        append!(velGraph, yout[i][2]);
        append!(timeGraph, yout.t[i]);

        elseif(yout[1,i] >= 0) # after apex of trajectory
        append!(altGraph, yout[i][1]);
        append!(velGraph, yout[i][2]);
        append!(timeGraph, yout.t[i]);
    else
        # dont keep track
        break;
    end
end

# Back Solving for accleration, Force of Drag, Cd, Ma, Re

# storage arrays
acc_num = zeros(length(time),1);
F_d =     zeros(length(time),1);
F_T =     zeros(length(time),1);
Ma =      zeros(length(time),1);
Re =      zeros(length(time),1);
rho =     zeros(length(time),1);
Cd =      zeros(length(time),1);

# back out acceleration, drag, Mach and Reynolds
for i = 1:length(time)
    # solves for important rocket statistics
    arr = BackoutPerformance(time[i],alt[i],vel[i])
    Cd[i] =  arr[1];
    F_T[i] = arr[2];
    F_d[i] = arr[3];
    Ma[i] =  arr[4];
    Re[i] =  arr[5];
    rho[i] = arr[6];
    Cd[i] =  arr[7];
end
;
```

```
┌ Warning: Instability detected. Aborting
└ @ DiffEqBase C:\Users\Nick\.julia\packages\DiffEqBase\XoVg5\src\integrator_interface.jl:349
```

Results and discussion

```
In [289]:   ###### Results output #######↔
```

```
------------------------ Rocket Parameters ------------------------
Vehicle dry weight: 94.6079940806868 [lb]
Vehicle wet weight: 136.321547714638 [lb]
Vehicle diameter: 0.666666688 [ft]
Propellant mass (total): 18.920972155723522 [kg], 41.713553633951186, [lb]
Ox:fuel mass ratio: 2.56[-]
Oxidizer mass: 13.606092336700062 [kg], 29.996263287335687 [lb]
Fuel mass: 5.31487981902346 [kg], 11.7172903466155 [lb]
Propellant mass fraction: 0.3059938383421981 [-]
Propellant tank Weight : 6.42320695895983 [lb]
Oxidizer tank Wall thickness : 0.07300045862074338 [in]
Kerosene tank Wall thickness : 0.07327141674678438 [in]
Oxidizer tank length (inner dim): 2.551537814734738 [ft]
Kerosene tank length (inner dim): 1.4484621852652624 [ft]
Oxidizer tank volume: 0.012646153306810182 [m^3]
Kerosene tank volume: 0.006805638792651992 [m^3]
Oxidizer tank max operating pressure: 730.0045862074338 [psi]
Kerosene tank max operating pressure: 732.7141674678437 [psi]
Oxidizer tank mean operating pressure: 365.0022931037169 [psi]
Kerosene tank mean operating pressure: 366.35708373392185 [psi]

----------------------- Propulsion Parameters ---------------------
Sea level thrust: 3.998815152337936 [kN], 898.9736343970914 [lbf]
Expansion ratio: 4.5 [-]
Chamber Pressure: 2.06 [MPa], 298.77828 [psia]
Exit Pressure: 0.08603909378670767 [MPa], 12.478938084636507 [psia]
Chamber temperature 3141 [K]
Molar mass of propellant: 22.15 [kg/kmol]
Specific heat ratio of propellant (frozen-flow) 1.14 [-]
Pressure at injector: 358.533936 [psia]
Mass flow rate (total): 1.6438736010303263 [kg/s]
Mass flow rate (oxidizer): 1.1821113535498977 [kg/s], 2.6061063322631752 [lb/s]
Mass flow rate (fuel): 0.4617622474804286 [kg/s], 1.0180102860403024 [lb/s]
Oxidizer feed line diameter 0.584 [in]
Fuel feed line diameter 0.43 [in]
Burn time 11.509992096633509 [s]

----------------------------- Results -----------------------------
Max Altitude: 10930.074497958574 [m], 35850.644353304124 [ft]
Max Altitude Time: 48.0 [s]
Freefall Time (no parachute): 41.7 [s]
Max Velocity: 543.0143089333413 [m/s]
Max Mach #: 1.659994878257993 [-]
Max Thrust: 4.207039393439311 [kN], 945.7845260390916 [lbf]
Specific Impulse: 253.91428652166857 [s]
Max Drag Force: 1.161948013164014 [kN], 261.217532839402 [lbf]
Max dynamic pressure: 129.34401438907042 [kPa]
Total Impulse: 10886.463617066614 [lb-s]
```

Above is a printout of all of the relevant calculated rocket data. A lot of this data is useful to WI Space Race, but is not necessary to understand for this project. The numbers that this project is interested in is the Propellant mass, ox/fuel mass, both tank lengths and both volumes, operating pressure of tanks, mass flow rates, and rocket motion outputs at the very bottom.

The numbers generated by the optimized code are very similar to the original output estimated by Brandon at the beginning of this project, confirming that none of the translation is broken.

The sensitivity analysis performed above has some of the same results printed below.

As for the main objective of this project, the optimal tank dimensions were used to generate all of this code, so theoretically the maximum altitude is achieved for the chosen constraints. It is easy to see that the length of the oxidizer tank plus the length of the kerosene tank adds up to the length constraint given previously in the code. And, The mass flow ratios of ox:fuel comes out to the given ratio defined as optimal for RP-1.

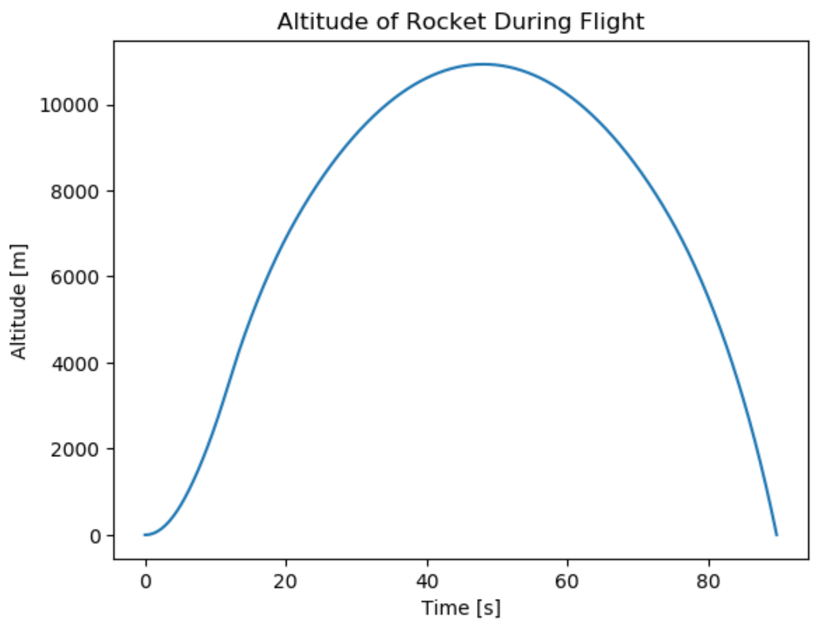
These results are perfect for what we originally set out to do. The code functions well, and sensitivity analysis can be performed. It is easy to change rocket parameters in the future and ideal estimates will be returned. None of this could have been done easily without the existing matlab code. Further modifications to the code are possible, and might be necessary as more complex constraints are ready to be assessed. This code can be used for years to come by WI Space Race, and will likely be maintained.

This project taught us a lot about how to optimize existing code. These optimum values may be plugged back into Matlab for other math that is more suited to matlab.

As an interesting demonstration of how the data is useful to predict the actual flight of the rocket, the equations of motion were used to generate altitude vs time and velocity vs time data that can be graphed using PyPlot.

In [290]:

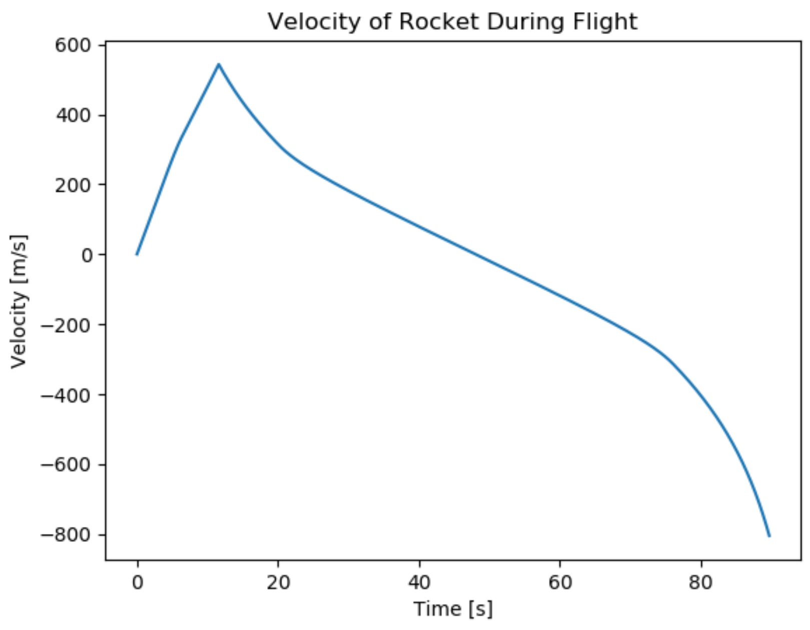
```
plot(timeGraph,altGraph)
xlabel("Time [s]")
ylabel("Altitude [m]")
title("Altitude of Rocket During Flight")
;
```



This altitude vs time graph may be important later because parachutes will be deployed to slow the descent of the empty rocket. It will most likely be deployed with sensors, but knowing about how long it will take might be a fail safe to code into the mechanism to release anyways in case a sensor malfunctions.

In [291]:

```
plot(timeGraph,velGraph)
xlabel("Time [s]")
ylabel("Velocity [m/s]")
title("Velocity of Rocket During Flight")
;
```



The final velocity the rocket would smash into the ground with is approximated by the furthest right datapoint. Quick analysis can show how much energy it would have. This assumes that terminal velocity is not reached before time of impact.

Probably needs a parachute.

In [292]:
```
KE = 0.5*m_veh_empty*(velGraph[end])^2
println("The rocket will have ",KE/(1e6), " Megajoules of energy.")
println("Which is ", KE*907.185*1000/(4.184*10e9), " grams of TNT!")
println("https://www.youtube.com/watch?v=a3KmcuCrkfg")
println("So a parachute is a pretty good idea if WI Space Race doesn't want to clean scrap metal off the ground")
```

```
The rocket will have 13.912386090997988 Megajoules of energy.
Which is 301.65172026677845 grams of TNT!
https://www.youtube.com/watch?v=a3KmcuCrkfg (https://www.youtube.com/watch?v=a3KmcuCrkfg)
So a parachute is a pretty good idea if WI Space Race doesn't want to clean scrap metal off the ground
```

Conclusion

While optimizing two tank lenghts given a maximum combined length seems like a very simple problem at face value, there are a lot of considerations that goes into designing a propulsion system. The added complexity of mass flows and engine parameters complicate things considerably. The math that must be done as an intermediate step is incredibly complicated and requires a few classes in mechanical engineering to truly be able to approach the problem. The input variables and constraints were relatively simple, but the amount of work that goes into getting realistic results is very complicated. So while the problem may seem easy on the outside, it is definitely not.

We are very happy with how this project turned out. We accomplished exactly what we set out to do, and the results are as good as we could have asked for. The code itself is ready to alter for further analysis, and it can be added to for future reference. We predict that the clients at WI Space Race will be satisfied with the results.

Finally, while these results were great for all of the constraints that are considered, the actual tank dimensions have many other considerations that must be given thought before purchasing. These considerations may or may not be possible to add to this code, and the limitations may need to be estimated within a certain tolerance. This tolerance may be able to be estimated by this code. This code will likely be another tool in the toolbox, and not something that will give a final result to anything. With some luck, this code goes down in the history books as the most useful group project ever.

Author Contributions

1. Modelling

Nick Havlicek: 80%
Shunzhang Li: 20%

Shunzhang helped decide what would be realistic to accomplish for this project, and helped reign Nick in a bit.

2. Analysis

Nick Havlicek: 60%
Shunzhang Li: 40%

Both of us checked the validity of the numbers that came out of analysis. Nick did a bit more work on confirming that nothing was going wrong within the code.

3. Data Gathering

Nick Havlicek: 100%
Shunzhang Li: 0%

Nick is a member of WI Space Race and gathered all of the literature and code from WI Space Race.

4. Software Implementation

Nick Havlicek: 60 %
Shunzhang Li: 40 %

Nick learned how the code worked and translated it to Julia, Shunzhang helped model and create the optimization.

5. Report writing and poster presentation

Nick Havlicek: 75%
Shunzhang Li: 25 %

Nick was responsible for the first draft of the description of the project, Shunzhang and Nick were responsible for editing.

note

While Nick did more work in every category, this was anticipated as Nick was the one to come up with the idea for the project and was more comitted to ensuring this project was up to his standards. Both students participated an equal amount towards the completion of the project.