Homework 4 - Unconstrained Optimization

Mark C. Anderson

March 11, 2021

1 Theory

The goal of our project is to maximize the amount of fuel inside a rocket after it reaches a certain altitude after traveling with a constant acceleration. We decided to use equal step sizes for the altitude and vary the step sizes in the downrange direction. This amounts to effectively just changing the tilt angle of the rocket relative to the ground. Specifying it in terms of height and downrange position step sizes enabled a simpler optimization model where we could set the start and endpoints easier. Our optimization problem is thus

Objective:

- Minimize the mass of the fuel consumed by the rocket when the rocket has arrived at a set position (x_{target}, y_{target}) by varying the distance traveled in the x-direction for each iteration.

Constraints:

- The rocket must start at (x_0, y_0) and end at (x_{target}, y_{target})
- The rocket must accelerate with a constant value equal to three times the natural force of gravity on the surface of the Earth (ie. 3 g's)
- Each successive step size in the x-direction must be no larger than twice the size of the previous step.

An example trajectory and force diagram are shown in Figure 1.

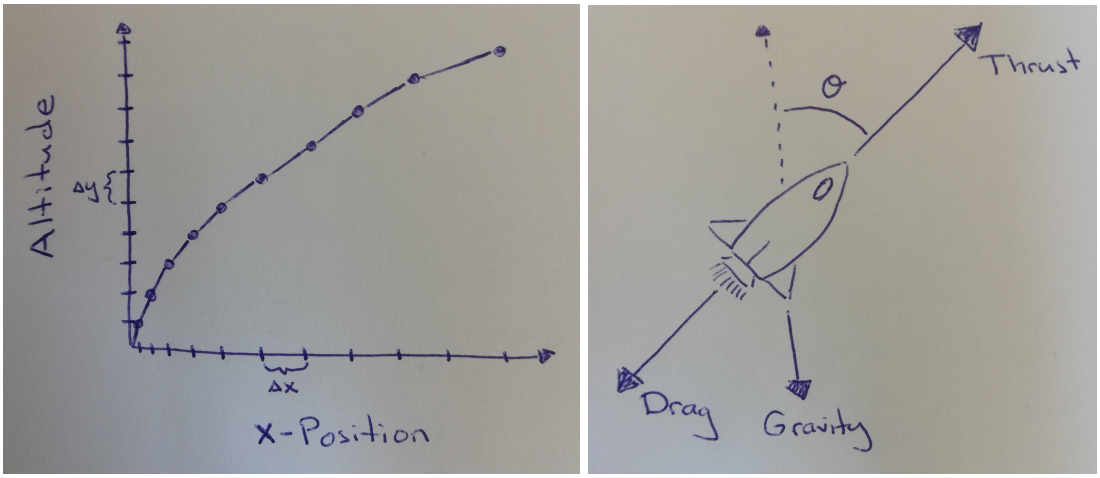

Figure 1: *Left:* An example trajectory showing evenly-spaced vertical steps and unevenly-spaced horizontal steps. *Right:* A force diagram for the rocket at some arbitrary point in its trajectory.

Where again the Δy value is constant, and we are changing the Δx values to reach the top-most location with the minimum amount of fuel consumed.

For each iteration, we use the final values from the previous iteration to calculate the drag as a function of altitude and velocity. The gravitational force is calculated using the mass of the rocket at the end of the previous iteration. The angle θ is calculated using the Δx and Δy values. Then the required thrust to maintain a constant acceleration of 3 g's is calculated from the diagram. The velocity of the gas is set, so we are able to then calculate how much fuel mass must be expended on the current iteration.

1.1 Calculating Drag

1.1.1 Atmosphere Model

The atmosphere is modeled as a function of the height in kilometers as:

$$T(h) = T_{sl} - 71.5 + 2\ln\left[1 + \exp(35.75 - 3.25h) + \exp(-3 + 0.0003h^3)\right]$$

$$p(h) = p_{sl}\exp\left(-0.118h - \frac{0.0015h^2}{1 - 0.018h + 0.0011h^3}\right)$$

where

- h = the altitude in kilometers
- T = the temperature in Kelvin
- T_{sl} = the temperature at sea level
- p = the pressure in Pascals
- p_{sl} = the pressure at sea level

This model is only valid up to 42 kilometers, but we are working on a way to extend that if we can. Results are shown in Figure 2.

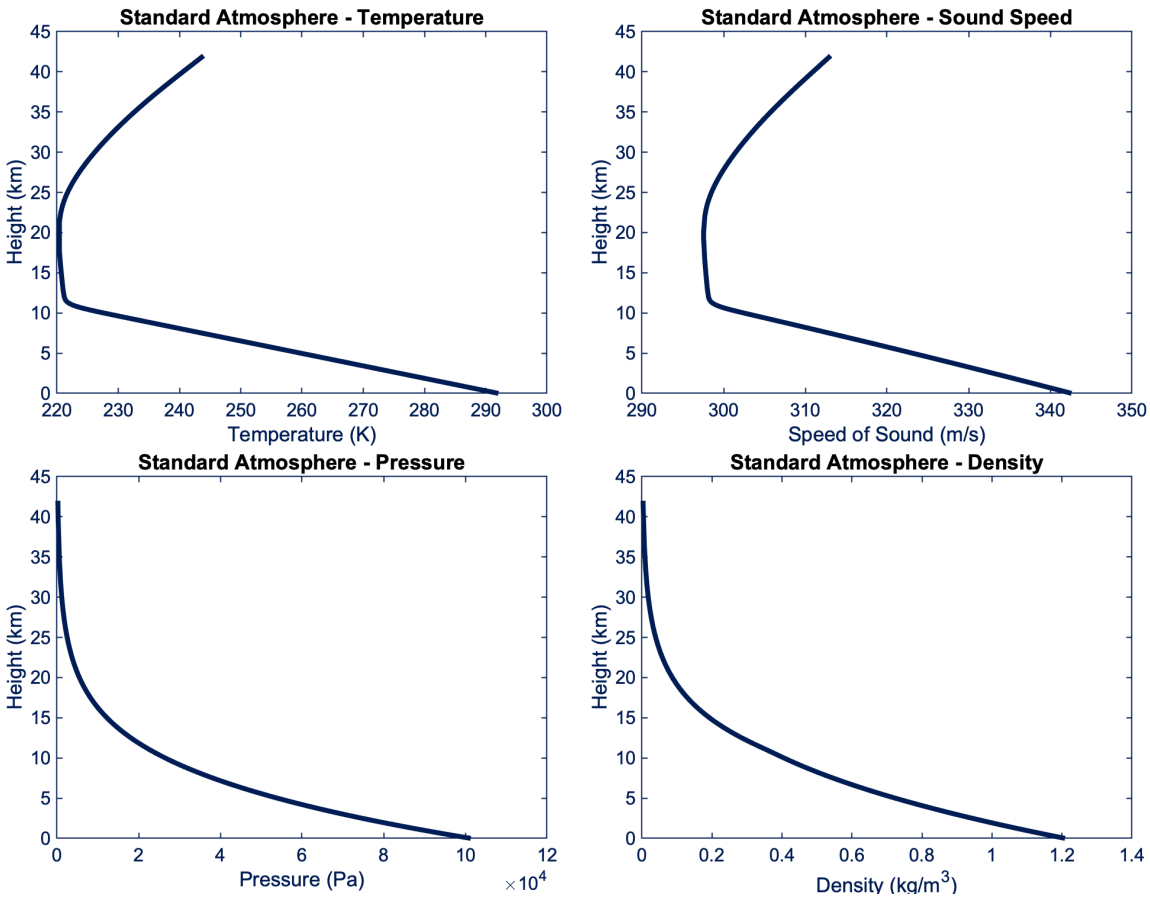

Figure 2: Different values calculated for a standard atmosphere.

1.1.2 Calculating the Drag

Ultimately we hope to be able to use the rocket dimensions to calculate both the subsonic and the supersonic drag on it, but for now we are just using approximate expressions that result in the model shown in Figure 3.

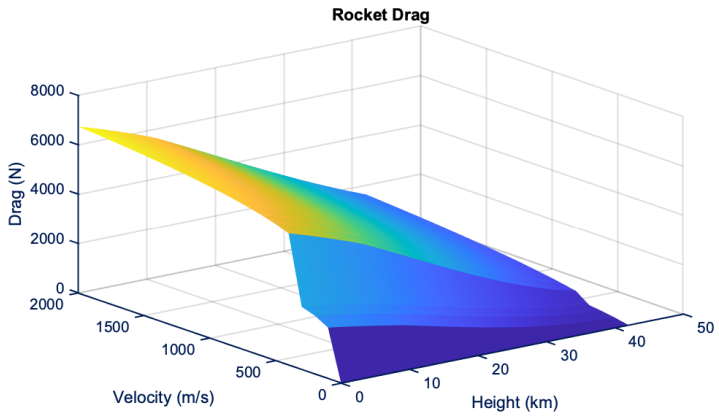

Figure 3: The drag as a function of height and velocity. This uses the standard atmosphere model to provide **very** rough estimates for the drag that for now are likely much different than what the rocket would really face. However, it should serve as a decent model for initial proving of the objective function.

We recognize that the drag values are likely much different than would be expected in reality, but this overall curve uses the atmospheric calculations shown above to help calculate a rough estimate for the overall drag shape as a function of height and velocity. We see that even at high speeds, the drag decreases substantially at higher altitudes, which is as expected.

1.2 Calculating θ

The angle θ represents the tilt angle of the rocket relative to its initial takeoff orientation (ie 0° means it is pointed straight up and 90° means it is oriented horizontal). It is calculated using the Δx and Δy values on the current iteration as:

$$\boxed{\theta = \tan^{-1}\left(\frac{\Delta x}{\Delta y}\right)}$$

1.3 Calculating Gravity

We could calculate the gravitational acceleration as a function of height, but we are going to stick with a contant acceleration due to gravity of $g = 9.8$ m/s^2.

However, the mass of the rocket is changing, and so the force of gravity on the rocket is also changing. We will use the mass at the end of the previous iteration as our estimate of the rocket's current mass. Mathematically, the force of gravity is therefore

$$\boxed{F_g = mg}$$

where m is the mass of the rocket after the previous iteration.

1.4 Calculating Thrust

The thrust must vary such that we maintain a constant acceleration. In order to solve for it we will use Newton's second law in vector form

$$\sum \vec{F} = m\vec{a}$$

$$\vec{F_g} + \vec{F_D} + \vec{F_T} = m\vec{a}$$

where

- $\vec{F_g}$ = the force of gravity
- $\vec{F_D}$ = the drag force
- $\vec{F_T}$ = the thrust force
- m = the mass of the rocket
- $\vec{a}$ = the acceleration of the rocket

Adding in the two-dimensional components of each vector, we arrive at

$$\begin{bmatrix} 0 \\ -mg \end{bmatrix} + \begin{bmatrix} -D\sin\theta \\ -D\cos\theta \end{bmatrix} + \begin{bmatrix} T\sin\theta \\ T\cos\theta \end{bmatrix} = m\begin{bmatrix} a\sin\theta \\ a\cos\theta \end{bmatrix}$$

$$\begin{bmatrix} T\sin\theta - D\sin\theta \\ T\cos\theta - D\cos\theta - mg \end{bmatrix} = m\begin{bmatrix} a\sin\theta \\ a\cos\theta \end{bmatrix}$$

Where D is the total drag force and T is the total thrust force. Substituting $T_x = T\sin\theta$ and $T_y = T\cos\theta$,

$$\begin{bmatrix} T_x - D\sin\theta \\ T_y - D\cos\theta - mg \end{bmatrix} = m \begin{bmatrix} a\sin\theta \\ a\cos\theta \end{bmatrix}$$

Solving for T_x and T_y gives the following solutions

$$\boxed{T_x = (ma + D)\sin\theta}$$

$$\boxed{T_y = (ma + D)\cos\theta + mg}$$

The final expression for the thrust is then

$$\boxed{T = \sqrt{T_x^2 + T_y^2}}$$

where T_x and T_y are defined above.

1.5 Calculating Mass

On each iteration, we shoot mass out of the back of the rocket. Now that we know the thrust, we can solve for the amount of mass shot out. We will use a simplified model for thrust as we solve for mass that does not account for nozzle shape:

$$T = \dot{m}V_e$$

where

- T = the thrust (solved for above)
- $\dot{m}$ = the mass flow rate
- V_e = the mass exit velocity

We can now discretize this to use finite time steps when solving for $\dot{m}$:

$$T = \frac{\Delta m}{\Delta t}V_e$$

$$\Delta m = \frac{T}{V_e}\Delta t$$

We can solve for the value of Δt if we say

$$\Delta t = \frac{\Delta r}{v}$$

where

- Δr is the displacement
- v = the velocity of the rocket

Writing this expression in terms of things that we know gives

$$\Delta t = \frac{\sqrt{(\Delta x)^2 + (\Delta y)^2}}{v}$$

Therefore,

$$\boxed{\Delta m = \frac{T}{V_e}\frac{\sqrt{(\Delta x)^2 + (\Delta y)^2}}{v}}$$

1.6 Objective Function Outline

The objective function must take in an input array of Δx values and output a total amount of fuel left in the rocket.

for i = 1:length(x)

- get previous iteration values for velocity, height, and mass
  - should just be stored as variables from one iteration to the next
  - initial values are that the velocity and height are both zero and the mass is the total mass of the rocket when fully fueled up
  - height can be solved for by just multiplying the iteration variable by the value of Δy (ie. height = i * Δy)
- calculate the tilt angle using
  - Δx
  - Δy
- calculate the drag using
  - velocity
  - height
- calculate the thrust using
  - mass
  - acceleration ($a = 3g$)
  - theta
- calculate the change in mass over this current iteration using
  - thrust
  - Δx
  - Δy
  - velocity
    - solved for using the iteration variable, since our acceleration is the same on each iteration (ie. velocity = i * acceleration)

1.7 Turning This Into an Optimization Problem

We have done several things to turn this into an optimization problem, namely

- Using $\ln x$ as our input array. This choice was made because we recognized that our different x-values were going to span several orders of magnitude. That would cause issues for scaling, and taking the natural logarithm of the array puts everything to roughly the same order of magnitude.
- Constrain the array of x-values such that each successive value is larger than the previous value. This is done using the linear inequality constraints in *fmincon*, which was something that I had to go learn how to do. Ultimately it wasn't too difficult to use the linear constraints because they are just a system of equations and the process is roughly similar to discretizing differential equations (which I got to do a lot of in a class last year).
- Derivatives are supplied to the optimizer using a function that I wrote for the previous homework. The method of choice is the complex-step method because finite-differencing was running into round-off limitations and algorithmic differentiation slow using the MATLAB package that we are using.

- For now, we are calculating the drag as a step function with altitude. There is a certain region where there is drag, and outside of it there is no drag. The hope is that the rocket will climb roughly straight through this portion of the atmosphere and make turns outside of it.
- The objective function solves for the amount of fuel used in the flight, which was several orders of magnitude larger than order one. Therefore, we divided the used mass to make it of order one.

2 Performing the Optimization

Performing the optimization was successful, except that the optimizer decided that the most fuel-efficient path was to go straight up and then turn almost 90 degrees at altitude. This is true, of course, if all of the upward velocity could instantaneously be converted to horizontal velocity without any negative side effects. Many attempts were made to try to overcome this, but they have not yet been successful. Results are shown in Figure 4.

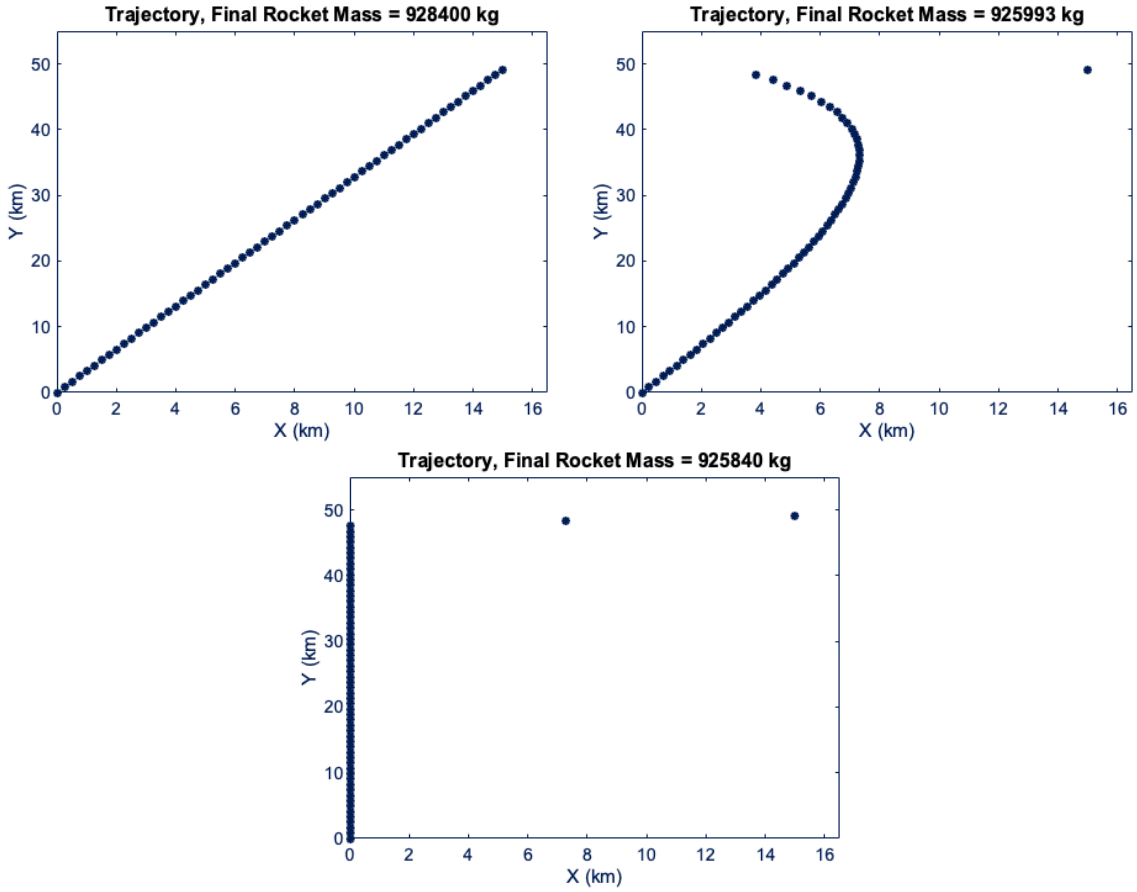

Figure 4: The optimization as shown at three different points. *Top Left:* The initial starting point. *Top Right:* The trajectory after a few iterations. *Bottom:* The final point. The first-order optimality did not fully converge but the optimizer stopped because the step tolerance threshold was reached and the constraint optimality threshold was also reached.

I've tried (unsuccessfully) to correct this behavior by:

- Constrain each step to be no larger than twice the size of the previous step.
  - This was intended to discourage a sudden large jump at the end. However, the smaller values all collapsed down to zero, forcing the successive steps to also become zero. Ultimately, the optimizer produced similar results.
- Constrain the tilt angle at the end of the trajectory.
  - This would then primarily impact the penultimate point in the trajectory, and the others still did not converge properly.
- Constrain the maximum difference in tilt angle between steps.

– This was also not successful. Usually the trajectory would stay roughly the same as the initial guess. The optimizer would then quit after the step tolerance threshold was reached. I'm not sure why this didn't work.

3 My Own Constrained Optimizer

I decided to write a penalty method constrained optimizer. The logical flow is:

1. Start with an initial guess and $\mu = 1$
2. Run *fminunc* on the equation

$$\text{obj}(x) = f(x) + \frac{\mu}{2} * \sum_j c_j(x)^2$$

   where

   - $f(x)$ is the function you are minimizing
   - $c(x)$ is the constraint function

3. Set the output of *fminunc* to be the new initial guess
4. Increment μ by one and
5. Repeat.

3.1 Test Problem

The test problem used in this case is the Rosenbrock function in two dimensions with constraints:

$$x_1 + x_2 \leq 4$$

$$2x_1 - 6x_2 \leq -10$$

Some of the results agree well with *fmincon*, as shown in Figures 5 and 7. Starting from some points can lead to unsuccessful convergence, as shown in Figure 6. Interestingly, for the two cases where they converge to the same point have different final solutions.

As far as efficiency goes, it appears that the penalty method is faster at getting into the right ballpark for the answer, as it tends to jump almost immediately to the close vicinity of the final point. However, I would be more inclined to trust the convergence from *fmincon* because I believe it to be a more robust solver that is less likely to get trapped by the pitfalls of particular functions or constraints.

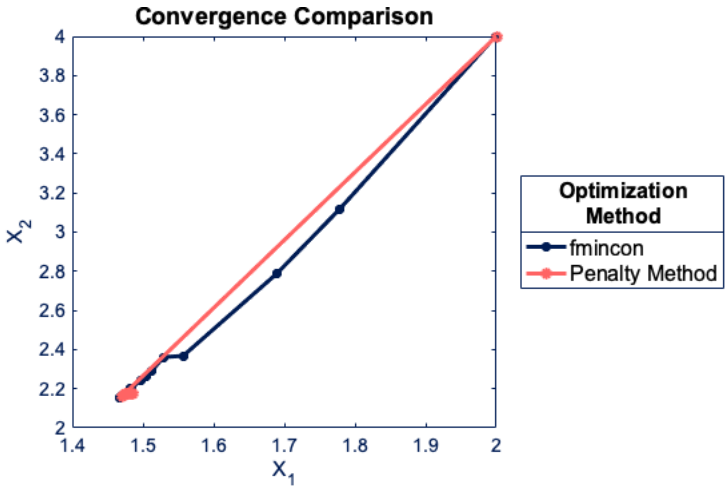

Figure 5: A case where the convergence agrees well between the two methods.

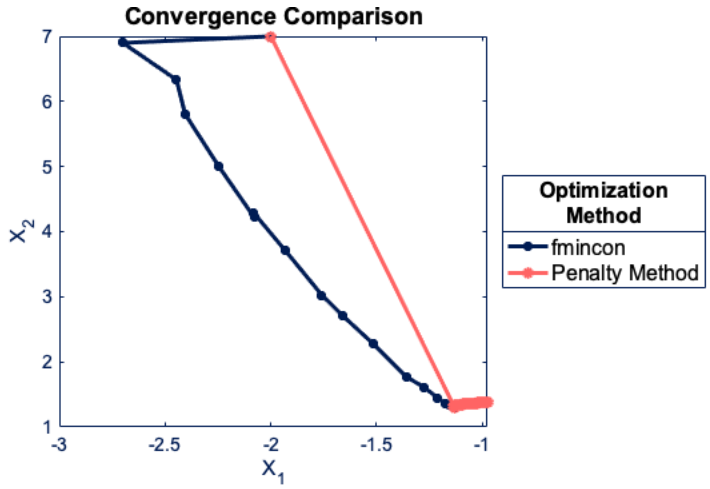

Figure 6: Another case where the convergence agrees well between the two methods

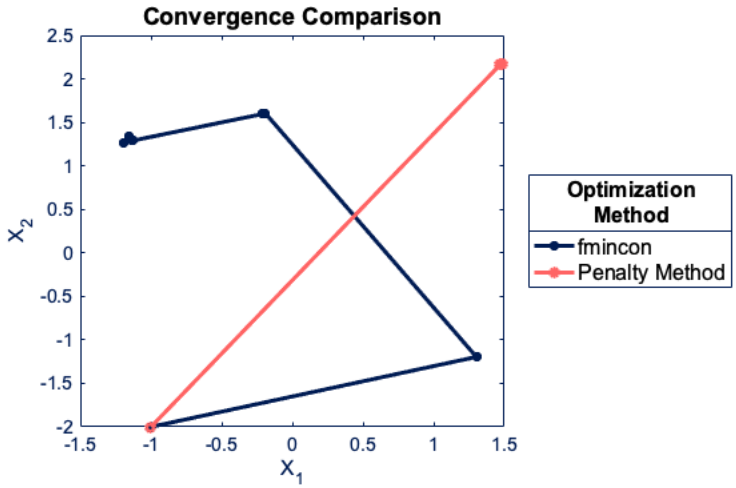

Figure 7: A case where the two methods diverge and do not converge to the same point.