Goddard's Rocket Problem

My project on Optimization and Optimal Control

Konrad Malik

The most general problem statement:

This problem concerns controlling a rocket's thrust in such a way that it will archive the greatest height. There is no contraint on time and in the whole case just a one (vertical) direction is taken into consideration.

Formal problem statement:

We assume that our equations of motion are nondimensionalized and normalized. That is they are not realistic but can be changed/transformed to such. Examplary normalisation process can be found in Tsiotras work (in References).

Take:

- t - time
- $v(t)$ - velocity
- $h(t)$ - height
- $g(h)$ - gravitational acceleration
- $m(t)$ - mass
- c - specific impulse
- $D(v,h)$ - drag
- $T(t)$ - thrust (our control)

Initial conditions:

- $h(0) = 1$
- $v(0) = 0$
- $m(0) = 1$

Boundary condition:

- $g(0) = 1$

Additional needed constants parameters, and formulas:

- $T_c = 3.5$
- $H_c = 500$
- $V_c = 620$
- $M_c = 0.6$
- $c = 0.5\sqrt{g(0)h(0)}$
- $D_c = 0.5\frac{V_c m(0)}{g(0)}$
- $T_{max} = T_c g(0)m(0)$

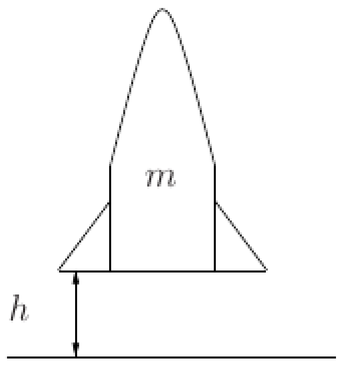
Final condition (empty fuel tank):

- $m(t_f) = M_c m(0)$

As density and the gravitational constant are dependant on altitude, we will use simplified models for $D(v, h)$ and g:

- $D(v, h) = D_c v^2 exp(-H_c\frac{h-h(0)}{h(0)})$
- $g(h) = g(0)(\frac{h(0)}{h})^2$

Given the above, we can now define the rocket's equations of motion (EOM):



$$
\begin{aligned}
\dot{h}(t) &= v(t) \\
\dot{v}(t) &= \frac{1}{m(t)}[T(t) - D(v, h)] - g(h) \\
\dot{m}(t) &= \frac{-T(t)}{c}
\end{aligned}
$$

We define our objective:

We will try to obtain the optimal thrust profile which will give us the highest altitude. That means we want to maximize $h(t_f)$ with no constraints on time t_f, thus we can write:

$$J(x,u) = -h(t_f) \tag{1}$$

where x are our state variables: $[h, v, m]^T$, and u is our control: $T(t)$.

The general formula for cost J is:

$$J(x,u) = \phi(x(t_f)) + \int_{t_0}^{t_f} f(x(t),u(t))dt$$

based on that we conclude:

$$\begin{gathered} \phi(x(t_f)) = -h(t_f) \\ f(x,u) = 0 \end{gathered} \tag{2}$$

Equations derivation

Hamiltonian

Given that Hamiltonian formula is:

$$H(x,u,\lambda) = L(x,u) + \lambda f(x,u)$$

in our case Hamiltonian equals:

$$H(x,u,\lambda) = \lambda_h + \lambda_v(\frac{1}{m(t)}[T(t) - D(v,h)] - g(h)) - \lambda_m \frac{T(t)}{c}$$

Co-state equations of motion

General formula:

$$\dot{\lambda} = -H_x$$

In our case:

$$\begin{aligned} \dot{\lambda_h} &= -H_h = \lambda_v \frac{\partial D}{\partial h}\frac{1}{m} \\ \dot{\lambda_v} &= -H_v = -(\lambda_h - \lambda_v \frac{\partial D}{\partial v}\frac{1}{m}) = \lambda_v \frac{\partial D}{\partial v}\frac{1}{m} - \lambda_h \\ \dot{\lambda_m} &= -H_m = \frac{T-D}{m^2} \end{aligned}$$

Co-state final conditions:

From (2) we have (as $\phi(x(t_f))$ a linear function of $h(t_f)$):

$$\begin{gathered} \lambda_h = -1 \\ \lambda_h = 0 \\ \lambda_h = 0 \end{gathered}$$

Hamiltonian maximalization discussion:

The next step is to find when Hamiltonian achives its extrema based on the Pontryagin's Maximum Principle:

$$H(x,u,\lambda) = \lambda_h + \lambda_v(\frac{1}{m(t)}[T(t) - D(v,h)] - g(h)) - \lambda_m \frac{T(t)}{c}$$

grouping elemets in the above equation gives:

$$H(x,u,\lambda) = (\frac{\lambda_v}{m} - \frac{\lambda_m}{c})T - (\frac{D}{m} - g)\lambda_v + \lambda_h$$

As our control is T, it can be clearly seen that minimizing the above function can be achived in 3 different cases:

$$\begin{cases} T = T_{max}, & \text{if } (\frac{\lambda_v}{m} - \frac{\lambda_m}{c}) < 0 \\ 0 < T < T_{max}, & \text{if } (\frac{\lambda_v}{m} - \frac{\lambda_m}{c}) = 0 \\ T = 0, & \text{if}(\frac{\lambda_v}{m} - \frac{\lambda_m}{c}) > 0 \end{cases} \tag{3}$$

The middle expression corresponds to the so-called "singular arc". Singular arc typically occurs when Hamiltonian is linear in control (as in our case) and the coefficient of the control term equals zero (as in the middle expression).

On a singular arc the following must be staisfied:

$$H_T = \frac{\lambda_v}{m} - \frac{\lambda_m}{c} = 0$$

From that we conclude:

$$\dot{H}_T = \ddot{H}_T = 0$$

From the above equations a formula describing a nonlinear feedback control law for T on a singular arc can be derived, however this is out of scope in this project.

Based on our above reasoning and papers such as *Drag-law Effects in the Goddard Problem* by Tsiotras and Kelley, a solution to the Goddard Problem as we defined it typically consists of a 3 arcs as defined in eq. (3).

Numerical Solution

In order to solve out problem numerically we will use python 3. Required packages: OpenGoddard, numpy, matplotlib

Necessary imports:

- numpy: general scientific computing
- matplotlib: library for plotting
- OpenGoddard: library for solving optimization problems in python

In [15]:
```
import numpy as np
import matplotlib.pyplot as plt
from OpenGoddard.optimize import Problem, Guess, Condition, Dynamics
%matplotlib inline
```

We define a class "Rocket" with initial/boundary conditions as specified in our Formal Problem Statement. This class represents a real rocket object with attributes such as height, velocity, mass, drag, thrust etc.

Coefficients are chosen based on OpenGoddard examples and can be freely changed.

Equations for drag coefficient, maximum thrust etc. were listed in "Formal Problem Statement" under "Additional needed constants parameters".

In [16]:
```
class Rocket:
    g0 = 1.0  # Gravity acceleration at the surface

    def __init__(self):
        self.H0 = 1.0  # Initial height
        self.V0 = 0.0  # Initial velocity
        self.M0 = 1.0  # Initial mass
        self.Tc = 3.5  # Coeff of thrust
        self.Hc = 500  # Coeff of density changes
        self.Vc = 620  # Coeff of velocity
        self.Mc = 0.6  # Fraction of whole rocket mass reserved for fuel
        self.c = 0.5 * np.sqrt(self.g0*self.H0)  # Specific impulse (empi
rical formula)
        self.Mf = self.Mc * self.M0               # Final mass (empty roc
ket)
        self.Dc = 0.5 * self.Vc * self.M0 / self.g0  # Coeff of drag
        self.T_max = self.Tc * self.g0 * self.M0     # Maximum thrust (em
pirical formula)
```

We define the dynamics of our model. The inputs to this function need to be:

- prob : the Problem object, representing the whole optimisation problem that we are calculating
- obj : the object that is controller and which state is optimised. In this case it is a Rocket class
- section : additional parameter needed by this library, represents a given time-section (timestep)

The problem class is not initialized yet. It will be done after defining all neccessary functions and they will take it as an argument.

Firstly the state variables h, v and m are assigned to the "Problem" class as "states". The "Problem" class in OpenGoddard is an abstraction representing the actual problem we want to solve. Thus, it is neccessary to provide state variables.

Then, under the "control" comment, we are assigning control variables to the same Problem class. In this case the only control we have is thrust T.

Next, we take the necessary parameters from the defined earlier Rocket class.

In order to fully describe our dynamic equations, we need to define "additional formulas" for drag, which is dependent on the state variables, and gravity constant, which changes with altitude.

The function returns a Dynamics class with dynamic equations defined below.

In [17]:
```
def dynamics(prob, obj, section):
    # retrieve states from Problem class
    h = prob.states(0, section) # height
    v = prob.states(1, section) # velocity
    m = prob.states(2, section) # mass

    # retrieve control from problem
    T = prob.controls(0, section) # control (thrust)

    # retrieve Rocket class objects
    Dc = obj.Dc # drag coefficient as defined in Rocket class
    c = obj.c # specific impulse as defined in Rocket class
    g0 = obj.g0 # gravity constant as defined in Rocket class
    H0 = obj.H0 # initial height as defined in Rocket class
    Hc = obj.Hc # height constant as defined in Rocket class

    # define additional formulas
    drag = Dc * v ** 2 * np.exp(-Hc * (h - H0) / H0) # drag equation
    g = g0 * (H0 / h)**2 # gravity constant

    # final dynamic equations definition
    dx = Dynamics(prob, section) # Dynamics class initialization
    dx[0] = v # time-derivative of the first state (height)
    dx[1] = (T - drag) / m - g # time-derivative of the second state (vel
ocity)
    dx[2] = - T / c # time-derivative of the third state (mass)
    return dx() # returns the array of dynamic equations
```

Next, we need to define the constraints for our optimisation problem. In OpenGoddard we need to separately define equality, and inequality constraints. The equality function defines equality constraints (initial and boundary conditions) and takes the following arguments:

- prob : the Problem object, representing the whole optimisation problem that we are calculating
- obj : the object that is controller and which state is optimised. In this case it is a Rocket class

We need to retrieve already defined states and controls from the Problem object, initialize the Condition class and add equalities to this class. In particular these are just initial and end conditions.

It returns a Conditions class with equalities that we define below.

In [18]:
```
def equality(prob, obj):
    # retrieve states from Problem class for all sections (all timesteps)
    h = prob.states_all_section(0) # height
    v = prob.states_all_section(1) # velocity
    m = prob.states_all_section(2) # mass

    # retrieve control from Problem class for all sections (all timestep
s)
    T = prob.controls_all_section(0) # thrust

    # initialize condition class
    result = Condition()

    # add initial conditions
    result.equal(h[0], obj.H0) # initial height
    result.equal(v[0], obj.V0) # initial velocity
    result.equal(m[0], obj.M0) # initial mass

    # add final conditions
    result.equal(v[-1], 0.0) # zero velocity
    result.equal(m[-1], obj.Mf) # final mass equal to mass of empty Rocke
t

    return result() # return these conditions
```

An analogous function has to be defined for inequality constraints. It takes the same inputs and returns the Condition class but this time it contains only inequality contraints.

In [19]:
```python
def inequality(prob, obj):
    # retrieve states from Problem class for all sections (all timesteps)
    h = prob.states_all_section(0) # height
    v = prob.states_all_section(1) # velocity
    m = prob.states_all_section(2) # mass

    # retrieve control from Problem class for all sections (all timestep
s)
    T = prob.controls_all_section(0) # thrust

    # retrieve time from the Problem class "prob"
    tf = prob.time_final(-1) # final time

    # initialize condition class
    result = Condition()

    # add lower bound conditions
    result.lower_bound(h, obj.H0) # height cannot be lower than initial
    result.lower_bound(v, 0.0) # velocity must be positive
    result.lower_bound(m, obj.Mf) # mass cannot be lower than the final m
ass of an empty rocket
    result.lower_bound(T, 0.0) # thrust must be positive
    result.lower_bound(tf, 0.1) # assume that final time cannot be lower
 than 0.1 for stability

    # add upper bound conditions
    result.upper_bound(m, obj.M0) # mass cannot be larger than the initia
l mass
    result.upper_bound(T, obj.T_max) # thrust cannot be larger than the m
aximal thrust

    return result() # return these conditions
```

Now we can define the cost function as we depicted in the eq. (1):

In [20]:
```python
def cost(prob, obj):
    # retrieve height from Problem for all sections (all timesteps)
    h = prob.states_all_section(0)
    # define const function (final height should be maximised)
    # also note the minus sign, since this in OpenGoddard cost function i
s being minimised
    cost_function = -h[-1]
    return cost_function # return this cost function
```

Starting point (initialization)

Now with defined all constraints, objects and the cost function, we can proceed to initialization of out problem and defining all the constants specific to our case.

Below we set the time bounds, number of nodes (timesteps, this is a discretisation in time). We also need to set the number of state equations (3), number of controls (1) and the maximum number of iterations, after which calculations will be stopped if we won't find an optimal solution until then.

```
In [21]: # set case specific constants
         time_init = [0.0, 0.5] # initial and final time (arbitrary values)
         n = [100] # number of nodes (timesteps, discretisation in time)
         num_states = [3] # number of states equations (3 - height, velocity, mas
         s)
         num_controls = [1] # number of controls (1 - thrust)
         max_iteration = 40 # maximum number of iterations after which calculation
         s will stop
```

Now we can initialize the OpenGoddard *Problem* class (main entry point to our optimization) passing the values we defined above:

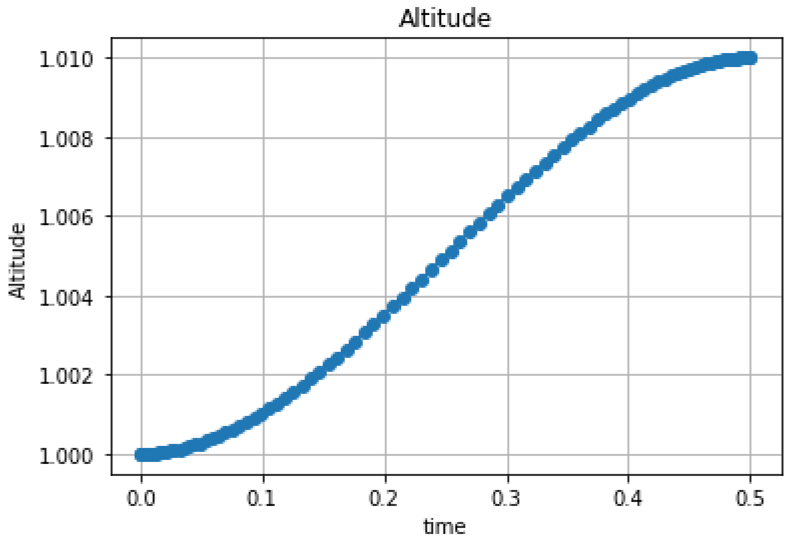
```
In [22]: prob = Problem(time_init, n, num_states, num_controls, max_iteration)
```

The next step is to initialize a *Rocket* object defined earlier:

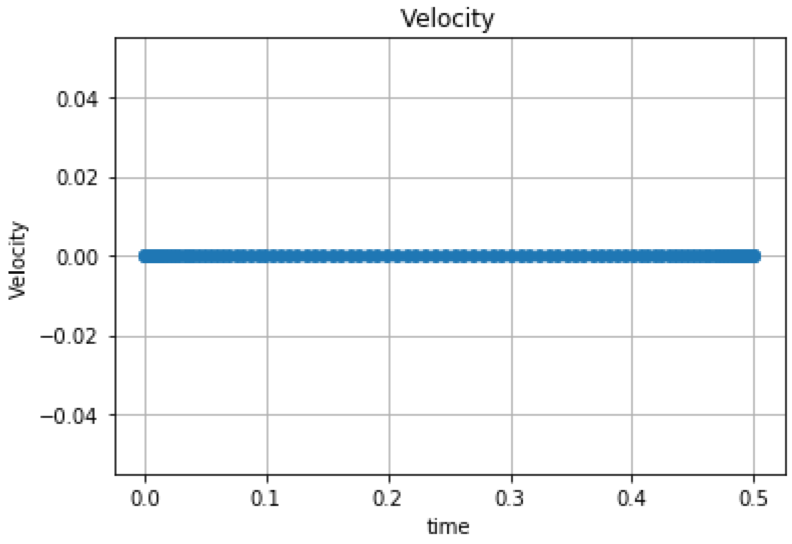
```
In [23]: obj = Rocket()
```

In order to be able to solve this optimization problem we need to initialze our state varables (make a initial guess about their profiles). This is done in the cells below. We also plot these initial profiles just to visualize them.

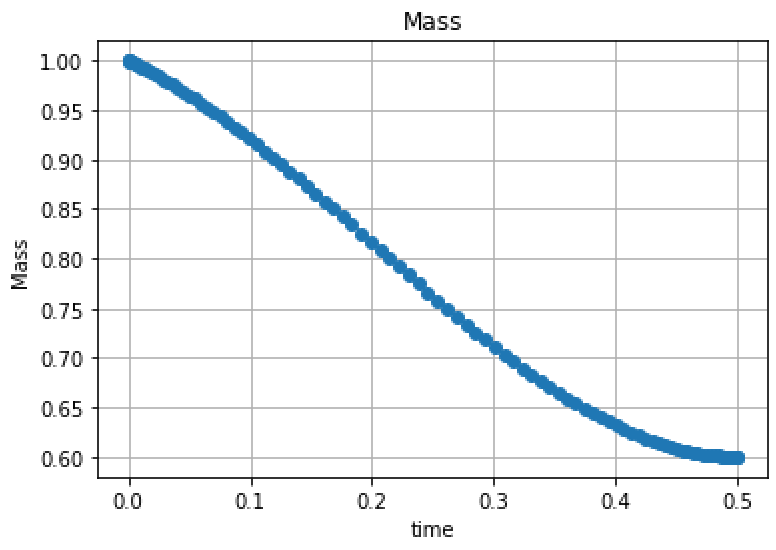
```
In [24]: # initial guess for the altitude profile
         H_init = Guess.cubic(prob.time_all_section, obj.H0, 0.0, 1.010, 0.0)
         Guess.plot(prob.time_all_section, H_init, "Altitude", "time", "Altitude")
```



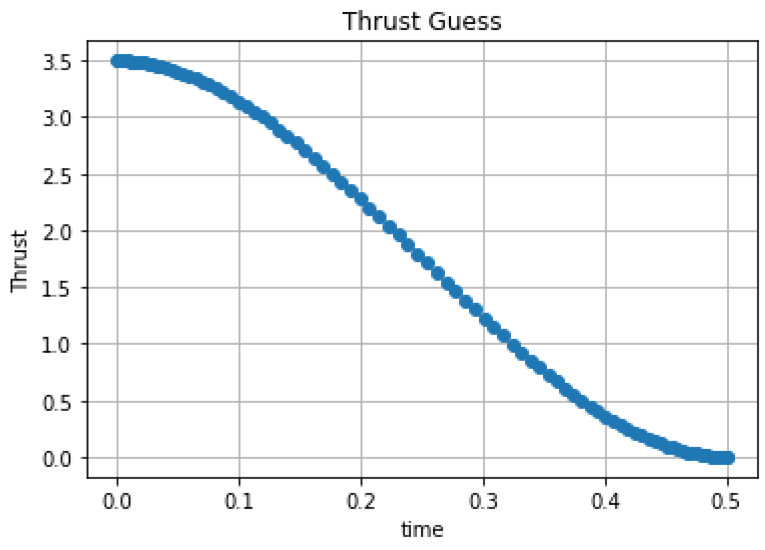
```
In [25]: # initial guess for the velocity profile
         V_init = Guess.linear(prob.time_all_section, 0.0, 0.0)
         Guess.plot(prob.time_all_section, V_init, "Velocity", "time", "Velocity")
```



In [26]:
```
# initial guess for the mass profile
M_init = Guess.cubic(prob.time_all_section, obj.M0, -obj.Mc, obj.Mf, 0.0)
Guess.plot(prob.time_all_section, M_init, "Mass", "time", "Mass")
```



In [27]:
```
# initial guess for the thrust profile
T_init = Guess.cubic(prob.time_all_section, obj.Tc, 0.0, 0.0, 0.0)
Guess.plot(prob.time_all_section, T_init, "Thrust Guess", "time", "Thrus
t")
```



Now take these guesses and set them as the initial input to our problem:

In [28]:
```
# Set the profiles for all timesteps using our initial guesses
# for states
prob.set_states_all_section(0, H_init) # for height
prob.set_states_all_section(1, V_init) # for velocity
prob.set_states_all_section(2, M_init) # for mass

# for control
prob.set_controls_all_section(0, T_init) # for thrust
```

Now we need to take our initialized Problem class and set up the dynamics, cost, equalities and equalities that we defined as functions earlier.

In [29]:
```
# set the rest of inputs to our Problem class
prob.dynamics = [dynamics] # set dynamics
prob.knot_states_smooth = [] # knot states (intermediate values, we are n
ot using them so set as an empty array)
prob.cost = cost # set the cost function
prob.cost_derivative = None # the derivative of the cost (the cost is sim
ply a final height, so no derivative - set None)
prob.equality = equality # set the equality constraints
prob.inequality = inequality # set the inequality constraints
```

One last step is to define a function to format results during optimization process:

In [30]:
```
def display_func():
    # take height from the Problem class
    h = prob.states_all_section(0)
    # print to console the actual final height
    print("max altitude: {0:.5f}".format(h[-1]))
```

Start the solver

As we now have all the prerequisites, we can start the solver.

This solver allows to solve NLP problems and by default uses the *Sequential Least Squares Programming* optimiser.

To cite the description of this algorithm: "SLSQP optimizer is a sequential least squares programming algorithm which uses the Han–Powell quasi–Newton method with a BFGS update of the B–matrix and an L1–test function in the step–length algorithm. The optimizer uses a slightly modified version of Lawson and Hanson’s NNLS nonlinear least-squares solver"

More information about this optimiser can be found in References.

In [31]:
```
prob.solve(obj, display_func, ftol=1e-10)
```

```
---- iteration : 1 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: -1.0080580256013836
            Iterations: 26
            Function evaluations: 10478
            Gradient evaluations: 26
Iteration limit exceeded
max altitude: 1.00806

.
.
.

---- iteration : 33 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: -1.0128299882354883
            Iterations: 26
            Function evaluations: 10478
            Gradient evaluations: 26
Iteration limit exceeded
max altitude: 1.01283

---- iteration : 34 ----
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -1.0128299883189094
            Iterations: 1
            Function evaluations: 404
            Gradient evaluations: 1
Optimization terminated successfully.
max altitude: 1.01283
```

We see that our optimisation ended after 34 iterations and that the maximal nondimensionalized altitude obtained was 1.01283.

Post processing

Now as our optimisation has ended successfully, we can start to post-process results.

Firstly we need to extract states, control and time from the resolved Problem class:

```
In [33]: # retrieved state variables from the solved case
         h = prob.states_all_section(0) # height
         v = prob.states_all_section(1) # velocity
         m = prob.states_all_section(2) # mass

         # retrieved control variables from the solved case
         T = prob.controls_all_section(0) # thrust

         # retrieved time from the solved case
         time = prob.time_update()
```

We need also to calculate drag as in was not resolved explicitly during optimisation (but taken into consideration) and we want to visualize it along with thrust.

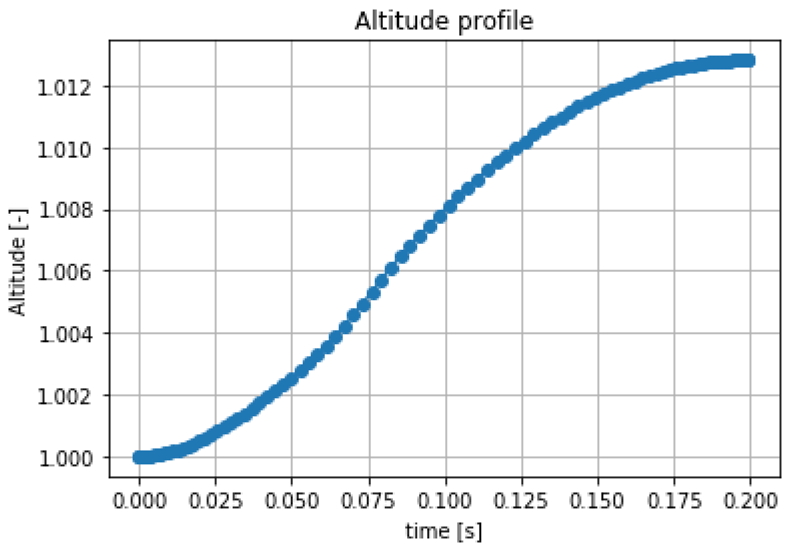
```
In [34]: # calculate drag using the same formula as before
         drag = obj.Dc * v ** 2 * np.exp(-obj.Hc * (h - obj.H0) / obj.H0)
```

Results visualisation

We can now visualize all the results. Code below is needed only to plot the already computed and extracted results.

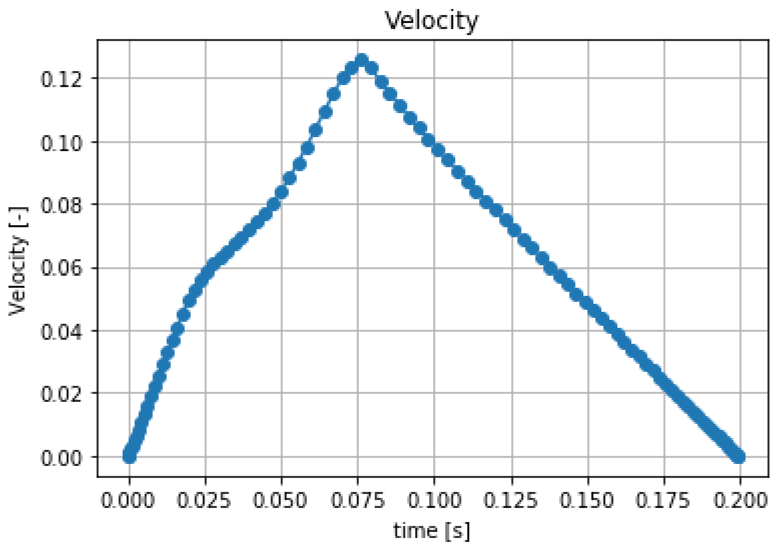
```
In [35]: plt.figure()
         plt.title("Altitude profile")
         plt.plot(time, h, marker="o", label="Altitude")
         plt.grid()
         plt.xlabel("time [s]")
         plt.ylabel("Altitude [-]")

Out[35]: Text(0,0.5,'Altitude [-]')
```



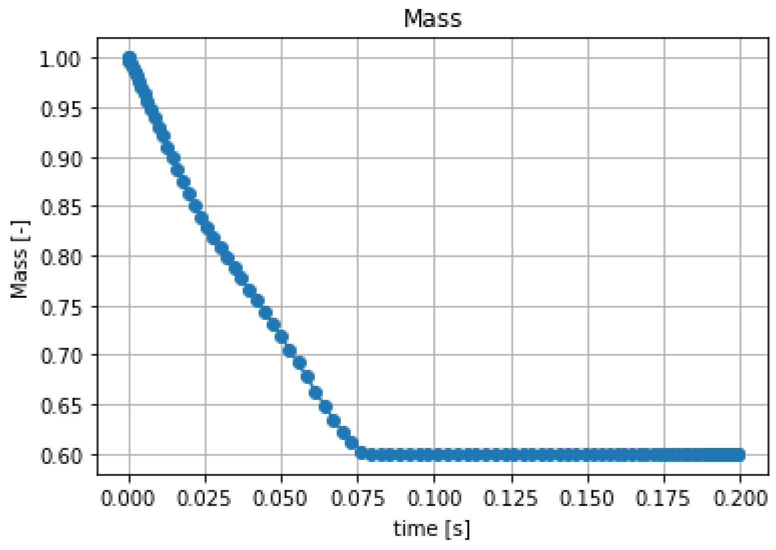
```
In [36]: plt.figure()
         plt.title("Velocity")
         plt.plot(time, v, marker="o", label="Velocity")
         plt.grid()
         plt.xlabel("time [s]")
         plt.ylabel("Velocity [-]")

Out[36]: Text(0,0.5,'Velocity [-]')
```



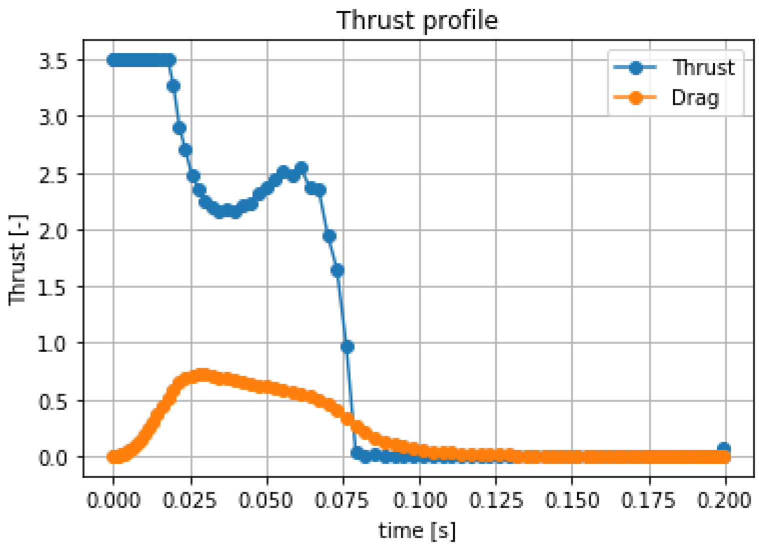
```
In [37]: plt.figure()
         plt.title("Mass")
         plt.plot(time, m, marker="o", label="Mass")
         plt.grid()
         plt.xlabel("time [s]")
         plt.ylabel("Mass [-]")

Out[37]: Text(0,0.5,'Mass [-]')
```



```
In [38]: plt.figure()
         plt.title("Thrust profile")
         plt.plot(time, T, marker="o", label="Thrust")
         plt.plot(time, drag, marker="o", label="Drag")
         plt.grid()
         plt.xlabel("time [s]")
         plt.ylabel("Thrust [-]")
         plt.legend(loc="best")

Out[38]: <matplotlib.legend.Legend at 0x7f6462f4c320>
```



Discussion

We can see that we accurately predicted that our solution should consist of 3 arcs. This scenario is called a *Bang-Singular-Bang scenario*. That is, we use maximum thrust initially to approx. t=0.02. Next, as in eq. (3), we are minimizing H, so we are keeping $\frac{\lambda_v}{m} - \frac{\lambda_m}{c} = 0$ until we ran out of fuel $(m(0) = m(t_f))$. Finally, we are forced to keep $T = 0$ and wait until we reach the maximum altitude.

References

Tsiotras P., Kelley H.J., *Drag-law Effects in the Goddard Problem*, 1991

Principles of Optimal Control, Course notes, MIT, 2008

Nonlinear Systems and Control, Course notes, Automatic Control Laboratory, ETH Zurich, 2015

OpenGoddard library examples and documentation
https://istellartech.github.io/OpenGoddard/apis/OpenGoddard.html
(https://istellartech.github.io/OpenGoddard/apis/OpenGoddard.html)

Sequential Quadratic Programming Methods, P. Gill, E. Wong, 2010
https://www.ccom.ucsd.edu/~peg/papers/sqpReview.pdf
(https://www.ccom.ucsd.edu/~peg/papers/sqpReview.pdf)