

Gas Dynamics — Work 1

Luiz Georg

15/0041390

April 20, 2022

1 Part 1

1.1 Initial Values

The freestream properties at 18.00 km and Mach 3, as taken from website AeroToolbox (<https://aerotoolbox.com/atmcalc/>), are:

Table 1 – Freestream flow properties.

| Property | Value |
|----------|--------------------------|
| P | 7505 Pa |
| M | 3.000 |
| T | 216.7 K |
| ρ | 0.1207 kg/m ³ |

1.2 Equation system

The flow across the planar 3-shockwave diffuser is assumed to be constant in the control volumes ∞ , ①, and ②. Inside control volume ③, the flow is subsonic and isentropic, but we are only interested in the values immediately after the shockwave, so we will not calculate the changes inside this volume.

Across the region boundaries, the flow goes through shockwaves. The flow after a shockwave can be calculated from the flow before the shockwave according to Eqs. (1.1) to (1.7) (ANDERSON JR., 2017), where the properties before and after the shock are represented by the subscripts 1 and 2, respectively. The right side of these equations depend only on the Mach number before the shockwave M_1 and the shockwave angle β . Alternatively, we can calculate β from θ , but that would lead to costlier calculations.

$$M_{n,1} = M_1 \sin \beta \quad (1.1)$$

$$M_{n,2}^2 = \frac{1 + \frac{\gamma-1}{2} M_{n,1}^2}{\gamma M_{n,1}^2 - \frac{\gamma-1}{2}} \quad (1.2)$$

$$\frac{\rho_2}{\rho_1} = \frac{(\gamma + 1) M_{n,1}^2}{2 + (\gamma + 1) M_{n,1}^2} \quad (1.3)$$

$$\frac{p_2}{p_1} = 1 + \frac{2}{\gamma + 1} (M_{n,1}^2 - 1) \quad (1.4)$$

$$\frac{T_2}{T_1} = \frac{p_2 \rho_1}{p_1 \rho_2} \quad (1.5)$$

$$M_2 = \frac{M_{n,2}}{\sin(\beta - \theta)} \quad (1.6)$$

$$\tan \theta = 2 \cot \beta \frac{M_1^2 \sin^2 \beta - 1}{M_1^2 (\gamma + \cos(2\beta)) + 2} \quad (1.7)$$

We are also interested in the total properties (represented with subscript 0) and in the sonic speed a . In any point of the flow, they can be calculated from the static properties using [Eqs. \(1.8\)](#) to [\(1.11\)](#) ([ANDERSON JR., 2017](#)).

$$\frac{T_0}{T} = 1 + \frac{\gamma - 1}{2} M^2 \quad (1.8)$$

$$\frac{p_0}{p} = \left(1 + \frac{\gamma - 1}{2} M^2\right)^{\frac{\gamma}{\gamma-1}} \quad (1.9)$$

$$\frac{\rho_0}{\rho} = \left(1 + \frac{\gamma - 1}{2} M^2\right)^{\frac{1}{\gamma-1}} \quad (1.10)$$

$$a = \sqrt{\gamma RT} \quad (1.11)$$

Using these equations, we can calculate the values of the flow properties at each control volume for given freestream properties and shockwave angles. Then, the efficiency function for an 3-shockwave diffuser is simply $P_{03}/P_{0\infty} = f(M_\infty, \beta_2, \beta_2)$. This function can be calculated easily by repeated application of the shockwave equations, and a descent algorithm can be used to find the local maximum.

1.3 Optimization

Finding the optimal diffuser is equivalent to finding the global maximum of the efficiency function $\eta(\beta_1, \beta_2)$. A script, shown in full in [Appendix A](#), was used to find the optimal diffuser angles, and some important parts of the code will be discussed here.

The script uses a class, called `Flow`, to represent flow state. This class implements total/static properties conversions, and also a shockwave method. The shockwave method is shown in [listing 1](#), though the actual implementation is a class method, not a plain function.

Listing 1: Shockwave function

```
def _shockwave(self, beta) -> Flow:
    """
    Calculates flow properties after an oblique shockwave.
    Uses equations (9.13) through (9.18) @anderson2017.

    Parameters:
    beta: oblique angle of the shockwave [rad]

    Returns:
    The Flow object after the shockwave
    """

    # Checks shockwave is valid (supersonic and positive theta)
    # Maybe subsonic flow could be accepted, and spit a supersonic flow?
    if (self.mach < 1) or ((theta := self.velocity_deflection(beta)) <= 0):
        return Flow(np.nan, np.nan, np.nan, np.nan, self.fluid)

    gamma = self.fluid.gamma

    # Equation (9.13) @anderson2017
    M_n_1 = self.mach * np.sin(beta)
    M_n_1_sq = M_n_1 * M_n_1
    # Equation (9.14) @anderson2017 (modified)
    M_n_2 = np.sqrt((2 + (gamma - 1) * M_n_1_sq) / (2 * gamma * M_n_1_sq - (gamma - 1)))
    # Equation (9.15) @anderson2017
    rho_2 = self.density * (gamma + 1) * M_n_1_sq / (2 + (gamma - 1) * M_n_1_sq)
    # Equation (9.16) @anderson2017
    p_2 = self.pressure * (1 + 2 * gamma * (M_n_1_sq - 1) / (gamma + 1))
    # Equation (9.17) @anderson2017
    T_2 = self.temperature * p_2 * self.density / (self.pressure * rho_2)
    # Equation (9.18) @anderson2017
    M_2 = M_n_2 / np.sin(beta - theta)

    flow_2 = Flow(p_2, M_2, T_2, rho_2, self.fluid)
    return flow_2
```

Using the Flow class, we can create a function that calculates the total pressure efficiency $\eta(M_\infty, \beta_1, \beta_2)$. In fact, the function is easily generalized to n-shockwave diffusers, so the general version was implemented. The implementation used in the script is shown in [listing 2](#).

Listing 2: Total Pressure efficiency in an n-shockwave diffuser

```
def nShock_p0_eff(betas: tuple[float, ...], flow_in: Flow) -> float:
    """
    Calculates the total pressure efficiency in a diffuser with n-1 oblique and 1 normal
    shockwaves.

    Parameters:
    betas: angles of the n-1 oblique shockwaves [rad]
    flow_in: Flow object of the flow in the upstream

    Returns:
    Total pressure efficiency of the diffuser
    """

    flow_after = flow_in
    for beta in betas:
        flow_after = flow_after.shockwave(beta)
    flow_after = flow_after.shockwave(np.pi / 2)
    return flow_after.total_pressure / flow_in.total_pressure
```

Finally, the script uses the SciPy module to implement the optimization algorithm. The algorithm implements minimization on a scalar function of n-dimensional inputs, i.e. it finds a local minimum. Since we want the local maximum, the optimized function was the total pressure loss $1 - \eta$, instead of the efficiency η . The optimization method chosen takes a initial guess, which was chosen to be just above the minimum β angles to ensure existence. The implementation is shown in [listing 3](#).

Listing 3: Optimization algorithm

```
# Finds minimum beta values, aka beta when theta = 0
min_beta_1 = root_scalar(flow_inf.velocity_deflection, x0=0.1, x1=0.11).root
assert 0 < min_beta_1 < np.pi / 2

# Initial guess based on minimum beta angle (scaled to avoid evaluating on the boundary)
initial_betas = (min_beta_1 * 1.01, min_beta_1 * 1.01)

print("Solving...")
# Minimizes the total pressure efficiency loss using a descent algorithm
sol_weak = minimize(
    lambda betas: 1 - nShock_p0_eff(betas, flow_inf),
    initial_betas,
    method="SLSQP",
    options={"disp": True},
)
print("Done!")
print(f"Solution: {sol_weak.x}")
```

```
Solving...
Optimization terminated successfully    (Exit mode 0)
      Current function value: 0.2563475732605983
      Iterations: 7
      Function evaluations: 24
      Gradient evaluations: 7

Done!
Solution: [0.5622804  0.78800818]
```

To investigate whether our optimization was correct and if the maximum is global, we can plot coarse contour graph of η , shown in [Fig. 1](#). As can be seen, the maximum does appear to be correct and global.

We can then use our solution to extract the values we want, i.e. the optimal diffuser angles and the optimal flow properties in each control volume. This is implemented in [listing 4](#), along with some assertions, using the `Flow` class methods.

Listing 4: Extracting the optimal diffuser angles and flow properties

```
# Gets and prints all the solution values
betas = sol_weak.x
assert np.all(0 < betas) and np.all(betas < np.pi / 2)

p0_eff = 1 - sol_weak.fun
assert 0 < p0_eff < 1

print(f"Efficiency = {p0_eff:.4}")
print()

# Stores flow states into a list
flows = [flow_inf]
for n, beta in enumerate(betas):
    print(f"beta_{n+1}: {np.rad2deg(beta)}")
    print(f"theta_{n+1}: {np.rad2deg(flows[-1].velocity_deflection(beta))}")
    flows.append(flows[-1].shockwave(beta))
    print(f"Flow_{n+1}:", flows[-1], "\n")
flows.append(flows[-1].shockwave(np.pi / 2))
print(f"Flow_out: {flows[-1]}")
# Garbage Collection
gc.collect()
```

Efficiency = 0.7437

beta_1: 32.21629364825329

theta_1: 14.976701128986543

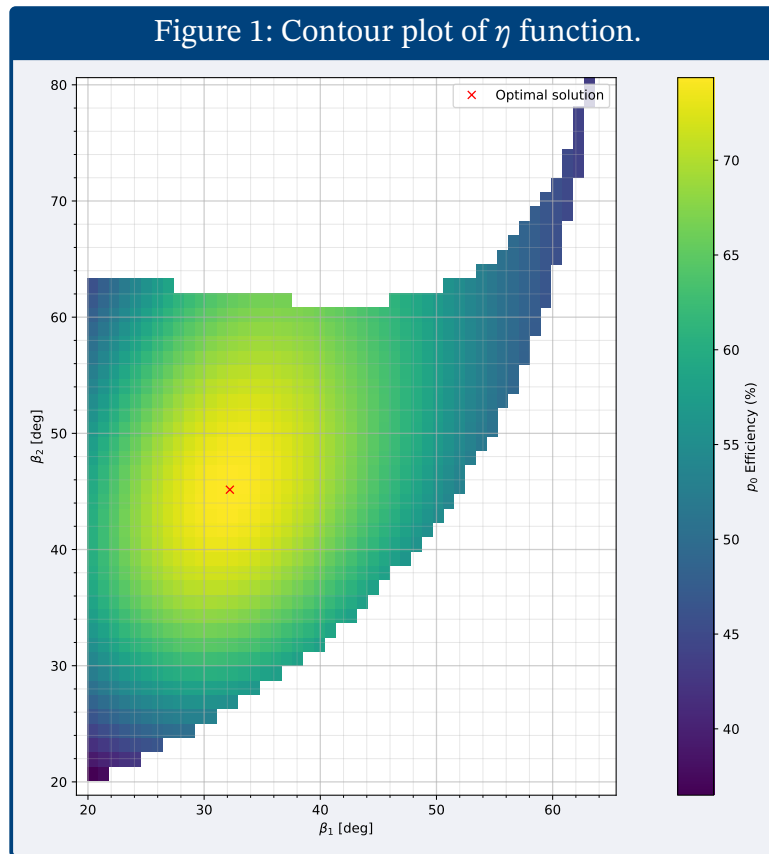
```
Flow_1: Flow(
    Pressure: 2.115e+04
    Mach: 2.256
    Temperature: 300.6
    Density: 0.2451
    fluid: Fluid(R=287.058, gamma=1.4)
)
```

beta_2: 45.14954292171187

theta_2: 18.814742838551123

```
Flow_2: Flow(
    Pressure: 5.959e+04
    Mach: 1.507
    Temperature: 417.1
    Density: 0.4977
    fluid: Fluid(R=287.058, gamma=1.4)
)
```

```
Flow_out: Flow(
    Pressure: 1.48e+05
    Mach: 0.6986
    Temperature: 552.7
    Density: 0.9328
    fluid: Fluid(R=287.058, gamma=1.4)
)
```

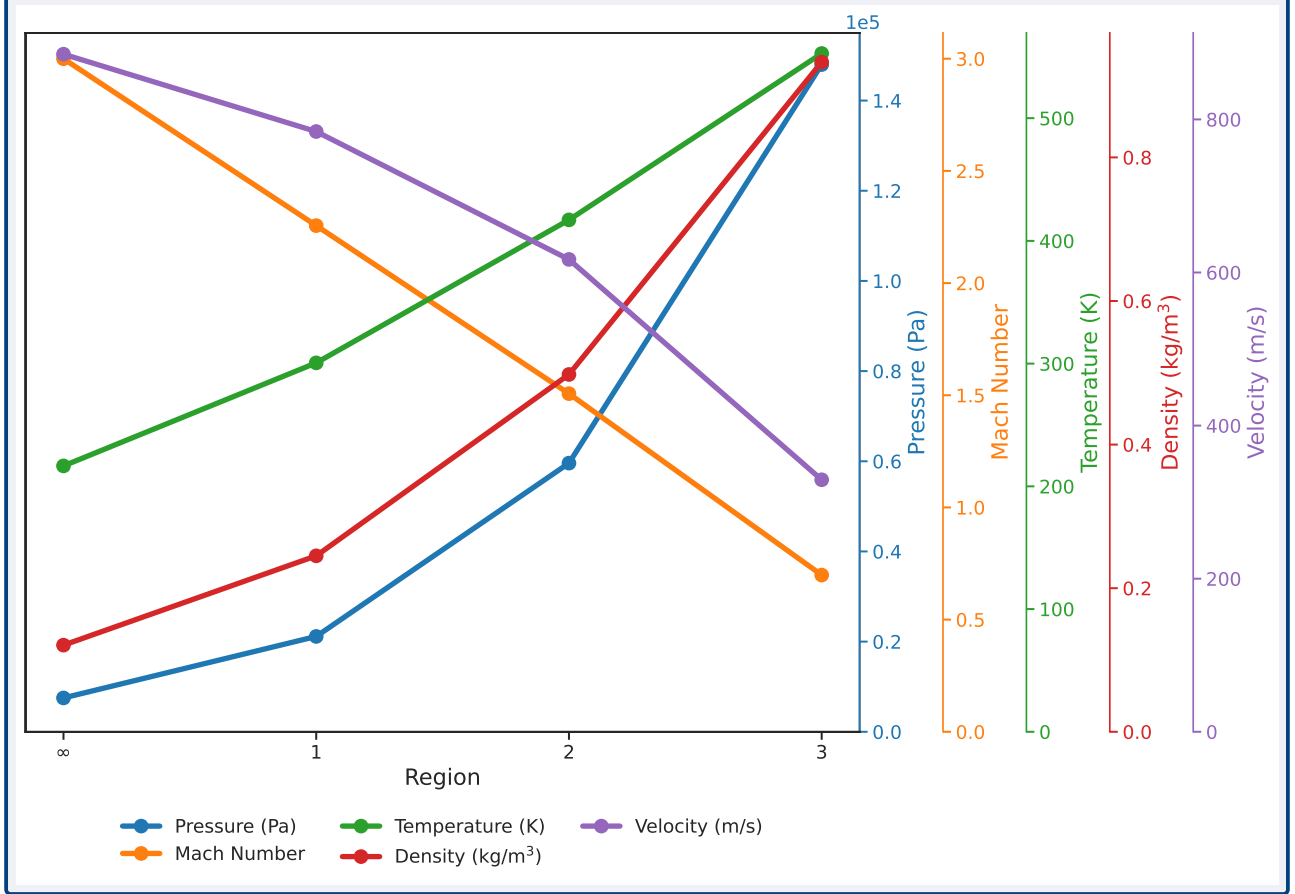


Then, we can plot the properties over each region. To better visualize relative changes in magnitude, we compared Pressure, Mach number, Temperature, Density, and Velocity in the Fig. 2. For easier lookup, the properties were also tabulated in Table 2.

Table 2 – Flow Properties in the 3-shockwave diffuser

| Region | Pressure [Pa] | Mach Number | Temperature [K] | Density [kg/m ³] | Velocity [m/s] |
|--------|---------------|-------------|-----------------|------------------------------|----------------|
| ∞ | 7505 | 3.000 | 216.7 | 0.1207 | 885.2 |
| 1 | 21 150 | 2.256 | 300.6 | 0.2451 | 784.2 |
| 2 | 59 590 | 1.507 | 417.1 | 0.4977 | 617.1 |
| 3 | 148 000 | 0.6986 | 552.7 | 0.9328 | 329.2 |

Figure 2: Properties in each control volume.



1.4 Optimal Geometry

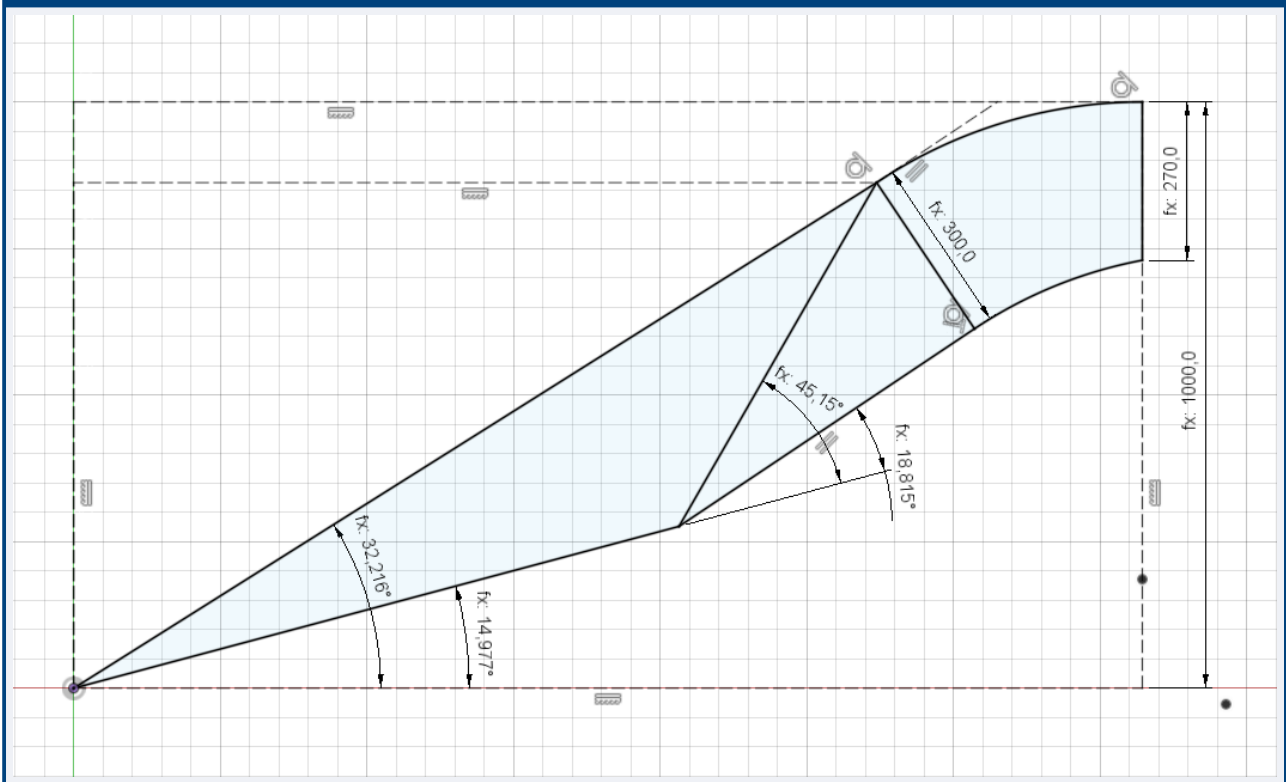
With the optimal angles, the geometry is still under-determined. The external wall size, D can be scaled arbitrarily, and will be set as unitary for a geometry generalization. Scaling this value will scale the geometry uniformly. For the walls to normal shockwave to form adequately, both the inner and outer wall must be (nearly) tangent to the flow in the normal shockwave vicinity. Additionally, a constraint is put in place such that the outer wall ends in a horizontal direction, finishing the rotation of the flow.

The wedge walls can still be scaled (up to a limit) inside these constraints, changing the geometry of entry to the diffuser. To constrain this geometry, the cross-sectional length of the normal shockwave was set to $D_e = 0.3D$.

Lastly, the internal length D_{int} is also arbitrary. [Oswatitsch \(1944\)](#) and [Hermann \(1956\)](#) show some restrictions to the throat formed here, but not a conclusive answer to its value. The critical value (where the normal shockwave happens right at the entrance) depends on the geometry further down the flow. Thus, as an initial approximation, the D_{int} was set such that the diffuser exit length is equal to $D_o = 0.9D_e$.

The final geometry, then, was modeled on the CAD Software Fusion 360, and shown in [Fig. 3](#).

Figure 3: Geometry of the 3-shockwave diffuser.



2 Part 2

The flow inside the ramjet motor can be divided into 3 parts: Expansion, Heat Addition, and Nozzle.

2.1 Expansion

Quasi-one-dimensional, isentropic expansion follows the Area–Mach Number relationship (Eq. (2.1)). Even though a sonic throat is not present, the relationship still stands, and the Mach Number anywhere in the flow can be calculated from the area ratio with a point with known properties. In our case, the point with known properties will be the entrance right after the normal shockwave from Chapter 1. The area in the entrance is D_e , and the area after expansion is D .

$$\left(\frac{A}{A^*}\right)^2 = \frac{1}{M^2} \left[\frac{2}{\gamma + 1} \left(1 + \frac{\gamma - 1}{2} M^2 \right) \right]^{\frac{\gamma + 1}{\gamma - 1}} \quad (2.1)$$

To calculate the Mach Number, we can find A^* and numerically solve Eq. (2.1) for $A = D$. Thus, the implementation in the script can be reduced to the Area–Mach Number relationship implemented in listing 5 and the root search implemented in listing 6. Details of the implementation can be found in Appendix A.

Listing 5: Area–Mach Number relationship function

```
def area_mach(M: float, gamma: float) -> float:
    """
    Calculates the ratio A / A* in isentropic quasi-one-dimensional flow.
    Uses Equation (10.32) @anderson2017.

    Parameters:
    M: Mach number
    gamma: Specific heat ratio
    """
    g_ratio = (gamma + 1) / (gamma - 1)
    # Equation (10.32) @anderson2017 (modified)
    return (2 / (gamma + 1) + M * M / g_ratio) ** (g_ratio / 2) / M
```

Listing 6: Finding Mach number after expansion

```
# Calculates Mach number using the area-Mach relation
# Calculates virtual throat
D_virtual = D_entrance / area_mach(flows2[-1].mach, flows2[-1].fluid.gamma)
# Solves area-Mach relation for the new Mach number
expanded_mach = root_scalar(
    lambda M: area_mach(M, flows2[-1].fluid.gamma) - D / D_virtual,
    bracket=[1, 1e-2],
    method="brentq",
).root

# Appends new Flow to the list
flows2.append(Flow.from_total_values(p0, expanded_mach, T0, rho0, flows2[-1].fluid))
print("Flow after expansion:", flows2[-1])

Flow after expansion: Flow(
    Pressure: 2.013e+05
    Mach: 0.161
    Temperature: 603.5
    Density: 1.162
    fluid: Fluid(R=287.058, gamma=1.4)
)
```

2.2 Heat Addition

The temperature variation ΔT will be considered as a change in static temperature. [Equation \(2.2\)](#) gives the relationship between Mach Number and static temperature ([ANDERSON, 2021](#)). Since we know the temperature before and after heat addition, the new Mach Number can be found by solving this equation.

When the new Mach Number is known, the new pressure and density can be calculated using [Eqs. \(2.3\) and \(2.4\)](#) ([ANDERSON, 2021](#)). Thus, the flow after heat addition is completely defined.

$$\frac{T_2}{T_1} = \left(\frac{1 + \gamma M_1^2}{\gamma M_2^2} \right)^2 \left(\frac{M_2^2}{M_1^2} \right) \quad (2.2)$$

$$\frac{p_2}{p_1} = \frac{1 + \gamma M_1^2}{1 + \gamma M_2^2} \quad (2.3)$$

$$\frac{\rho_2}{\rho_1} = \frac{p_2 T_1}{p_1 T_2} \quad (2.4)$$

Using [Eq. \(2.2\)](#) the script can find the new Mach number as shown in [listing 7](#), while the pressure and density were calculated as shown in [listing 8](#), yielding a complete Flow object.

Listing 7: Finding Mach number after heat addition

```
# Calculates temperature after heat addition
expanded_temp = flows2[-1].temperature
heated_temp = expanded_temp + delta_temp

# Calculates Mach number after heat addition
# Part of Equation (3.81) @anderson2021 that doesn't depend on the new Mach number
const_part = (
    np.sqrt(heated_temp / expanded_temp)
    * expanded_mach
    / (1 + flows2[-1].fluid.gamma * expanded_mach * expanded_mach)
)
# Solves Equation (3.81) @anderson2021 (modified) for the new Mach number
heated_mach = root_scalar(
    lambda M: const_part - M / (1 + flows2[-1].fluid.gamma * M * M),
    bracket=[1, 0],
    method="brentq",
).root
print(f"New Mach number: {heated_mach:.4}")
```

New Mach number: 0.3358

Listing 8: Finding flow properties after heat addition

```
# Calculates remaining Flow properties after heat addition
# Equation (3.78) @anderson2021
heated_pressure = (
    flows2[-1].pressure
    * (1 + flows2[-1].fluid.gamma * expanded_mach * expanded_mach)
    / (1 + flows2[-1].fluid.gamma * heated_mach * heated_mach)
)
# Equation (3.79) @anderson2021
heated_rho = (
    flows2[-1].density
    * heated_pressure
    / flows2[-1].pressure
    * expanded_temp
    / heated_temp
)

# Appends new Flow to the list
flows.append(
    Flow(heated_pressure, heated_mach, heated_temp, heated_rho, flows2[-1].fluid)
)
print("Flow after heat addition:", flows[-1])

Flow after heat addition: Flow(
    Pressure: 1.802e+05
    Mach: 0.3358
    Temperature: 2.103e+03
    Density: 0.2984
    fluid: Fluid(R=287.058, gamma=1.4)
)
```

2.3 Nozzle

Just as in [Section 2.1](#), the nozzle is a quasi-one-dimensional flow. The nozzle is defined by the area ratio, and the Mach Number. Using [Eq. \(2.1\)](#), one can find the critical throat area, implemented in the script as shown in [listing 9](#).

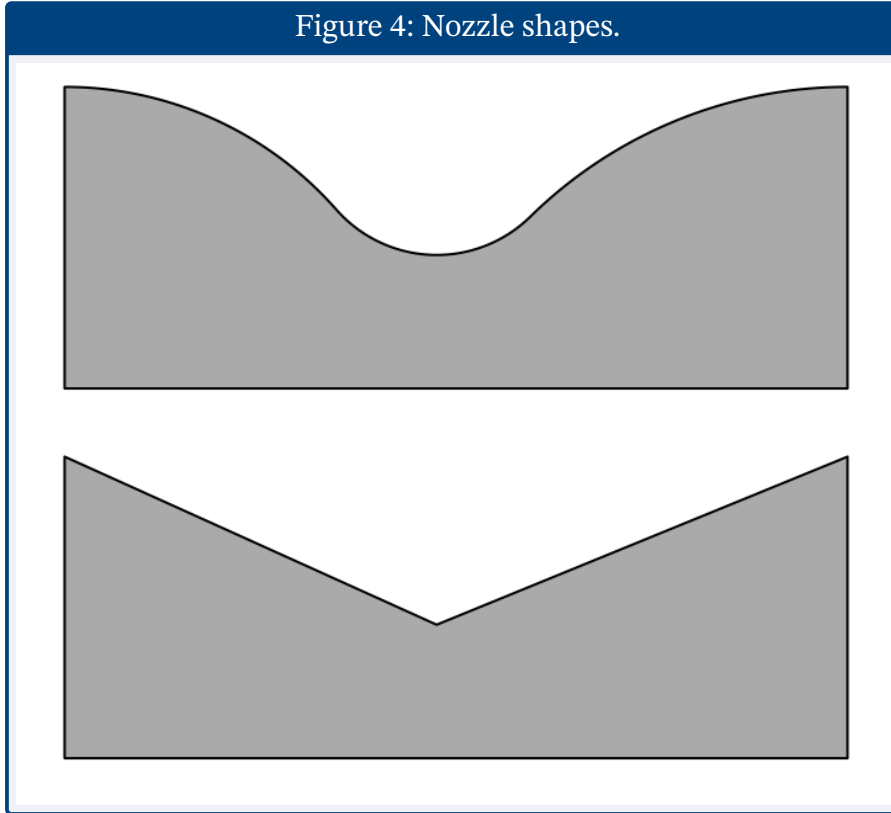
Listing 9: Finding the throat area

```
# Calculates throat semi-area
D_throat = D / area_mach(heated_mach, flows[-1].fluid.gamma)
print(f"Throat area: {D_throat}")

Throat area: 0.542665144954579
```

With the nozzle throat area D_t known, the geometry of the nozzle is fully determined. The two nozzle shapes under study are shown in [Fig. 4](#).

Figure 4: Nozzle shapes.



The walls for each shape were implemented as functions, and flow properties were calculated using [Eq. \(2.1\)](#) and [Eqs. \(1.8\)](#) and [\(1.11\)](#). The code implementing this calculation is shown in [listing 10](#).

Listing 10: Calculating flow properties in the nozzle

```
nozzles_flows = np.empty([len(nozzle_curves), len(x)], Flow)
p0_nozzle, T0_nozzle, rho0_nozzle = flows2[-1].total_values()
for n, curve in enumerate(nozzle_curves):
    # Calculates the Mach number across the nozzle.
    crit_index = np.argmin(curve[1])
    D_crit = curve[1, crit_index]
    crit_x = curve[0, crit_index]
    machs = np.array(
        [
            root_scalar(
                lambda M: area_mach(M, flows2[-1].fluid.gamma) - y_ / D_crit,
                # The sign function selects solution branch according to starting flow
                # and position relative to critical length
                bracket=[1, 100 ** np.sign((1 - flows2[-1].mach) * (x_ - crit_x))],
                method="brentq",
            ).root
            for x_, y_ in curve.T
        ]
    )
    nozzles_flows[n, :] = np.array(
        [
            Flow.from_total_values(
                p0_nozzle, mach, T0_nozzle, rho0_nozzle, flows2[-1].fluid
            )
            for mach in machs
        ]
    )
```

Some flow properties were calculated and plotted in [Fig. 5](#), where one can compare their relative changes as well as the differences between the two nozzle shapes.

Finally, the full ramjet design sketch can be seen in [Fig. 6](#). Some dimensions remained unconstrained (namely the length of the expansion region and of the combustion chamber), and were set to D .

Figure 5: Flow properties in the nozzle. Dotted lines represent the Bell shaped nozzle, while solid lines represent the wedge shaped nozzle.

Figure 6: Complete geometry of the ramjet.

APPENDIX A – Source Code

```
1  # %% [markdown]
2  # # SETUP
3  #
4
5  # %% [markdown]
6  # ## Imports
7
8  # %%
9  from __future__ import annotations
10 from scipy.optimize import minimize, root_scalar
11 import numpy as np
12 import matplotlib.pyplot as plt
13 from functools import lru_cache
14 import gc
15
16
17 # %% [markdown]
18 # ## Classes
19
20 # %%
21 class Fluid:
22     """
23     Stores the constant properties R and gamma of a fluid
24     """
25
26     def __init__(self, R: float, gamma: float) -> None:
27         self.R = R
28         self.gamma = gamma
29
30     def __str__(self) -> str:
31         return f"Fluid(R={self.R}, gamma={self.gamma})"
32
33
34 # %% [markdown]
35 # Flow class uses a few equations from @anderson2017, repeated here as they appear in the
36 ↪ book:
37
38 # %% [markdown]
39 #
40 # From chapter 8:
41 #
42 # 
$$a = \sqrt{\gamma R T}$$

43 #
```

```

44 #
45 # $$\tag{8.40}
46 # \frac{T_0}{T} = 1 + \frac{\gamma - 1}{2}M^2
47 # $$
48 #
49 # $$\tag{8.42}
50 # \frac{p_0}{p} = \left( 1 + \frac{\gamma - 1}{2}M^2 \right)^{\frac{\gamma}{\gamma - 1}}
51 # $$
52 #
53 # $$\tag{8.43}
54 # \frac{\rho_0}{\rho} = \left( 1 + \frac{\gamma - 1}{2}M^2 \right)^{\frac{1}{\gamma - 1}}
55 # \hookrightarrow 1}
56 # $$
57 #
58 # %% [markdown]
59 # From chapter 9
60 # $$\tag{9.13}
61 # M_{n,1} = M_1 \sin\beta
62 # $$
63 #
64 # $$\tag{9.14}
65 # {M_{n,2}}^2 = \frac{
66 # 1 + \frac{\gamma - 1}{2} {M_{n,1}}^2
67 # }{
68 # \gamma {M_{n,1}}^2 - \frac{\gamma - 1}{2}
69 # }
70 # $$
71 #
72 # $$\tag{9.15}
73 # \frac{\rho_2}{\rho_1} = \frac{(\gamma + 1) {M_{n,1}}^2}{2 + (\gamma + 1) {M_{n,1}}^2}
74 # \hookrightarrow }
75 # $$
76 # $$\tag{9.16}
77 # \frac{p_2}{p_1} = 1 + \frac{2}{\gamma + 1} ( {M_{n,1}}^2 - 1 )
78 # $$
79 #
80 # $$\tag{9.17}
81 # \frac{T_2}{T_1} = \frac{p_2}{p_1} \frac{\rho_1}{\rho_2}
82 # $$
83 #
84 # $$\tag{9.18}
85 # M_2 = \frac{M_{n,2}}{\sin(\beta - \theta)}
86 # $$
87 #
88 # $$\tag{9.23}
89 # \tan\theta = 2 \cot\beta \frac{{M_1}^2 \sin^2\beta - 1}{{M_1}^2 (\gamma +
90 # \hookrightarrow \cos(2\beta)) + 2 }

```

```

90  # $$
91
92  # %%
93  class Flow:
94      """
95      Stores the defining properties of a flow:
96
97      Pressure (p)
98      Mach Number (M)
99      Temperature (T)
100     Density (rho)
101     Fluid (fluid)
102     Speed of sound (a)
103     Velocity (V)
104
105     Care should be taken to use consistent units.
106     """
107
108     def __init__(self, p: float, M: float, T: float, rho: float, fluid: Fluid) -> None:
109         """
110         Constructs a Flow object from the defining properties and calculates some
111         auxiliary properties.
112         Uses Equations (8.25), (8.40), (8.42), (8.43) @anderson2017.
113
114         Parameters:
115         p: pressure [Pa]
116         M: Mach number
117         T: temperature [K]
118         rho: density [kg/m^3]
119         fluid: Fluid object
120         """
121
122         self.pressure = p
123         self.mach = M
124         self.temperature = T
125         self.density = rho
126         self.fluid = fluid
127
128         # Equation (8.25) @anderson2017
129         self.snd_spd = np.sqrt(fluid.gamma * fluid.R * self.temperature)
130         # From the definition of Mach Number
131         self.vel = M * self.snd_spd
132
133         aux_const = 1 + (fluid.gamma - 1) * M * M / 2
134         # Equation (8.40) @anderson2017
135         self.total_temperature = T * aux_const
136         # Equation (8.42) @anderson2017
137         self.total_pressure = p * aux_const ** (fluid.gamma / (fluid.gamma - 1))
138         # Equation (8.43) @anderson2017

```

```

139         self.total_density = rho * aux_const ** (1 / (fluid.gamma - 1))
140
141         self.shockwave = lru_cache(maxsize=8)(self._shockwave)
142
143     @classmethod
144     def from_total_values(
145         cls, p0: float, M: float, T0: float, rho0: float, fluid: Fluid
146     ):
147         f = 1 / Flow(1 / p0, M, 1 / T0, 1 / rho0, fluid).total_values()
148         return Flow(f[0], M, f[1], f[2], fluid)
149
150     def _p_ratio(M, gamma):
151         """
152         Calculates the pressure ratio of a shockwave.
153         Uses equation (9.19) @anderson2017.
154
155         Parameters:
156         M: Mach number
157         gamma: ratio of specific heats
158
159         Returns:
160         Pressure ratio
161         """
162
163         return (1 + (gamma - 1) * M * M / 2) ** (gamma / (gamma - 1))
164
165     def __str__(self) -> str:
166         return (
167             "\n\t".join(
168                 [
169                     "Flow(",
170                     f"Pressure: {self.pressure:.4}",
171                     f"Mach: {self.mach:.4}",
172                     f"Temperature: {self.temperature:.4}",
173                     f"Density: {self.density:.4}",
174                     f"fluid: {self.fluid}",
175                 ]
176             )
177             + "\n)"
178         )
179
180     def static_values(self) -> np.ndarray:
181         """
182         Outputs the defining properties of a flow as an ndarray.
183         Uses same order as __init__, but excludes the fluid property.
184         [p, M, T, rho]
185         """
186         return np.array([self.pressure, self.mach, self.temperature, self.density])
187

```

```

188     def total_values(self) -> np.ndarray:
189         """
190         Outputs the stagnation properties of a flow as an ndarray.
191         [p0, T0, rho0]
192         """
193         return np.array(
194             [self.total_pressure, self.total_temperature, self.total_density]
195         )
196
197     def aux_values(self) -> np.ndarray:
198         """
199         Outputs the auxiliary properties of a flow as an ndarray.
200         [a, V]
201         """
202         return np.array([self.snd_spd, self.vel])
203
204     def _shockwave(self, beta) -> Flow:
205         """
206         Calculates flow properties after an oblique shockwave.
207         Uses equations (9.13) through (9.18) @anderson2017.
208
209         Parameters:
210         beta: oblique angle of the shockwave [rad]
211
212         Returns:
213         The Flow object after the shockwave
214         """
215
216         # Checks shockwave is valid (supersonic and positive theta)
217         # Maybe subsonic flow could be accepted, and spit a supersonic flow?
218         if (self.mach < 1) or ((theta := self.velocity_deflection(beta)) <= 0):
219             return Flow(np.nan, np.nan, np.nan, np.nan, self.fluid)
220
221         gamma = self.fluid.gamma
222
223         # Equation (9.13) @anderson2017
224         M_n_1 = self.mach * np.sin(beta)
225         M_n_1_sq = M_n_1 * M_n_1
226         # Equation (9.14) @anderson2017 (modified)
227         M_n_2 = np.sqrt(
228             (2 + (gamma - 1) * M_n_1_sq) / (2 * gamma * M_n_1_sq - (gamma - 1))
229         )
230         # Equation (9.15) @anderson2017
231         rho_2 = self.density * (gamma + 1) * M_n_1_sq / (2 + (gamma - 1) * M_n_1_sq)
232         # Equation (9.16) @anderson2017
233         p_2 = self.pressure * (1 + 2 * gamma * (M_n_1_sq - 1) / (gamma + 1))
234         # Equation (9.17) @anderson2017
235         T_2 = self.temperature * p_2 * self.density / (self.pressure * rho_2)
236         # Equation (9.18) @anderson2017

```

```

237         M_2 = M_n_2 / np.sin(beta - theta)
238
239         flow_2 = Flow(p_2, M_2, T_2, rho_2, self.fluid)
240         return flow_2
241
242     def velocity_deflection(self, beta: float) -> float:
243         """
244         Calculates the velocity deflection angle theta for an oblique shockwave
245         Uses equations (9.13), (9.23) @anderson2017
246
247         Parameters:
248         beta: oblique angle of the shockwave [rad]
249
250         Returns:
251         angle of deflection of velocity [rad]
252         """
253
254         M = self.mach
255         gamma = self.fluid.gamma
256
257         # Equation (9.13) @anderson2017
258         M_n_1 = M * np.sin(beta)
259
260         # Equation (9.23) @anderson2017
261         theta = np.arctan(
262             2
263             / np.tan(beta)
264             * (M_n_1 * M_n_1 - 1)
265             / (M * M * (gamma + np.cos(2 * beta)) + 2)
266         )
267         return theta
268
269
270 # %% [markdown]
271 # ## Functions
272
273 # %% [markdown]
274 # Area-Mach number relation @anderson2017:
275 # 
$$\left(\frac{A}{A^*}\right)^2 = \frac{1}{M^2} \left[ \frac{\gamma + 1}{\gamma - 1} \left( 1 + \frac{\gamma - 1}{2} M^2 \right) \right]^{\frac{\gamma + 1}{\gamma - 1}}$$

276 # 
$$\left(\frac{A}{A^*}\right)^2 = \frac{1}{M^2} \left[ \frac{\gamma + 1}{\gamma - 1} \left( 1 + \frac{\gamma - 1}{2} M^2 \right) \right]^{\frac{\gamma + 1}{\gamma - 1}}$$

277 # 
$$\left(\frac{A}{A^*}\right)^2 = \frac{1}{M^2} \left[ \frac{\gamma + 1}{\gamma - 1} \left( 1 + \frac{\gamma - 1}{2} M^2 \right) \right]^{\frac{\gamma + 1}{\gamma - 1}}$$

278
279 # %%
280 def area_mach(M: float, gamma: float) -> float:
281     """
282     Calculates the ratio A / A* in isentropic quasi-one-dimensional flow.
283     Uses Equation (10.32) @anderson2017.
284

```

```

285     Parameters:
286     M: Mach number
287     gamma: Specific heat ratio
288     """
289     g_ratio = (gamma + 1) / (gamma - 1)
290     # Equation (10.32) @anderson2017 (modified)
291     return (2 / (gamma + 1) + M * M / g_ratio) ** (g_ratio / 2) / M
292
293
294 # %%
295 def nShock_p0_eff(betas: tuple[float, ...], flow_in: Flow) -> float:
296     """
297     Calculates the total pressure efficiency in a diffuser with n-1 oblique and 1 normal
298     shockwaves.
299
300     Parameters:
301     betas: angles of the n-1 oblique shockwaves [rad]
302     flow_in: Flow object of the flown in the upstream
303
304     Returns:
305     Total pressure efficiency of the diffuser
306     """
307
308     flow_after = flow_in
309     for beta in betas:
310         flow_after = flow_after.shockwave(beta)
311     flow_after = flow_after.shockwave(np.pi / 2)
312     return flow_after.total_pressure / flow_in.total_pressure
313
314
315 # %%
316 def bellNozzleGenerator(R_in, R_throat, R_out, L_throat, L_in=1, L_out=None):
317     """
318     Generates a function describing the geometry of a planar nozzle defined by 3 arcs.
319
320     Parameters:
321     R_in: Radius of the inlet arc
322     R_throat: Radius of the throat arc
323     R_out: Radius of the outlet arc
324     L_throat: semi-length of the throat
325     L_in: semi-length of the inlet arc
326     L_out: semi-length of the outlet arc
327
328     Returns:
329     bellCurve: The function describing the geometry of the nozzle
330     crit_len: The length where minimum area occurs
331     max_len: The length of the nozzle
332     """
333     if L_out is None:

```



```

334         L_out = L_in
335     L1 = (
336         np.sqrt((R_in + R_throat) ** 2 - (R_in - L_in + R_throat + L_throat) ** 2)
337         * R_in
338         / (R_in + R_throat)
339     )
340     l1 = L1 * R_throat / R_in
341     L2 = (
342         np.sqrt((R_throat + R_out) ** 2 - (R_out - L_out + R_throat + L_throat) ** 2)
343         * R_out
344         / (R_throat + R_out)
345     )
346     l2 = L2 * R_throat / R_out
347     max_len = L1 + l1 + l2 + L2
348     crit_len = L1 + l1
349
350     def bellCurve(x):
351         if not isinstance(x, np.ndarray):
352             x = np.array(x)
353         y = np.zeros_like(x)
354         # First arc
355         i1 = (0 <= x) & (x <= L1)
356         y[i1] = L_in - R_in + np.sqrt(R_in**2 - (x[i1] - 0) ** 2)
357         # Second arc
358         i2 = (L1 < x) & (x < max_len - L2)
359         y[i2] = L_throat + R_throat - np.sqrt(R_throat**2 - (x[i2] - (L1 + l1)) ** 2)
360         # Third arc
361         i3 = (max_len - L2 <= x) & (x <= max_len)
362         y[i3] = L_out - R_out + np.sqrt(R_out**2 - (x[i3] - max_len) ** 2)
363         return y
364
365     return bellCurve, crit_len, max_len
366
367
368 # %%
369 def coneNozzleGenerator(crit_len, max_len, L_throat, L_in=1, L_out=None):
370     """
371     Generates a function describing the geometry of a planar nozzle defined by 2 slopes.
372
373     Parameters:
374     max_len: Length of the nozzle
375     crit_len: Length where minimum area occurs
376     L_throat: Semi-length of the throat
377     L_in: Semi-length of the inlet
378     L_out: Semi-length of the outlet
379
380     Returns:
381     coneCurve: The function describing the geometry of the nozzle
382     crit_len: The length where minimum area occurs

```

```

383     max_len: The length of the nozzle
384     """
385     if L_out is None:
386         L_out = L_in
387
388     def coneCurve(x):
389         if not isinstance(x, np.ndarray):
390             x = np.array(x)
391         y = np.zeros_like(x)
392         # First slope
393         i1 = (0 <= x) & (x <= crit_len)
394         y[i1] = L_in + x[i1] * (L_throat - L_in) / crit_len
395         # Second slope
396         i2 = (crit_len < x) & (x <= max_len)
397         y[i2] = L_throat + (x[i2] - crit_len) * (L_out - L_throat) / (
398             max_len - crit_len
399         )
400         return y
401
402     return coneCurve, crit_len, max_len
403
404
405 # %%
406 styles = ["default", "fivethirtyeight", "seaborn-white"]
407 colors = plt.rcParams["axes.prop_cycle"].by_key()["color"]
408
409
410 def plot_flow_evolution(flow_data: np.ndarray, pos: np.ndarray, fig_ax=None, cols=None):
411     # Plots the relevant properties of the flow
412     if fig_ax is None:
413         fig = plt.figure(figsize=(10, 10))
414         ax = plt.axes()
415     else:
416         fig, ax = fig_ax
417
418     if cols is None:
419         cols = list(range(flow_data.shape[-1]))
420
421     # Different vertical axes
422     v_axes = [ax.twinx() for _ in cols]
423
424     # Line labels
425     labels = [
426         "Pressure (Pa)",
427         "Mach Number",
428         "Temperature (K)",
429         "Density (kg/m3)",
430         "Speed of Sound (m/s)",
431         "Velocity (m/s)",

```

```

432     "Total Pressure (Pa)",
433     "Total Temperature (K)",
434     "Total Density (kg/m3)",
435 ]
436
437 labels = [labels[c] for c in cols]
438
439 flow_data = flow_data[:, cols]
440
441 # Generates the actual plot lines
442 lines = [
443     v_ax.plot(pos, property, color=color, label=label)[0]
444     for v_ax, property, label, color in zip(v_axes, flow_data.T, labels, colors)
445 ]
446
447 # Tick appearance
448 tick_params = dict(size=6, width=1.5)
449
450 # Configures the scale axes
451 for v_ax, label, line, n in zip(v_axes, labels, lines, range(len(lines))):
452     v_ax.set_ylabel(label)
453     v_ax.yaxis.label.set_color(line.get_color())
454     v_ax.spines.right.set_color(line.get_color())
455     v_ax.tick_params(axis="y", colors=line.get_color(), **tick_params)
456     v_ax.ticklabel_format(axis="y", scilimits=(-3, 3))
457     v_ax.yaxis.offsetText.set_position((1.025 + 0.10 * n, 0))
458     v_ax.spines.right.set_position(("axes", 1 + 0.10 * n))
459     v_ax.set_ylim(bottom=0)
460
461 # Ticks on main ax
462 ax.tick_params(axis="y", which="both", left=False, right=False)
463 ax.tick_params(axis="x", **tick_params)
464
465 # fig.legend(loc="upper left", handles=lines, ncol=3)
466 box = ax.get_position()
467 ax.set_position([box.x0, box.y0 + box.height * 0.1, box.width, box.height * 0.9])
468
469 # Put a legend below current axis
470 ax.legend(
471     handles=lines,
472     loc="upper center",
473     bbox_to_anchor=(0.5, -0.1),
474     fancybox=True,
475     ncol=3,
476 )
477 ax.set_yticks([])
478 return fig, [ax, *v_axes]
479
480

```

```

481 # %% [markdown]
482 # # PART 1
483
484 # %% [markdown]
485 # ## Initial data
486
487 # %%
488 # Gets data from problem conditions
489 M_inf = 3.0
490 h = 18e3 # [m]
491
492 # Atmospheric conditions @aerotoolbox
493 p_inf = 7505.00 # [Pa]
494 T_inf = 216.65 # [K]
495 rho_inf = 0.12068 # [kg/m^3]
496
497 # Air constants
498 gamma_air = 1.4
499 R_air = 287.058
500
501
502 # %%
503 # Creates initial Objects
504 air = Fluid(R_air, gamma_air)
505 flow_inf = Flow(p_inf, M_inf, T_inf, rho_inf, air)
506
507 print(f"Freestream flow: {flow_inf}")
508
509
510 # %% [markdown]
511 # ## Minimization
512
513 # %% [markdown]
514 # Minimizes over beta angles instead of over theta angles, as `theta(beta)` is a much less
    ↪ expensive function to calculate than `beta(theta)`
515
516 # %%
517 # Finds minimum beta values, aka beta when theta = 0
518 min_beta_1 = root_scalar(flow_inf.velocity_deflection, x0=0.1, x1=0.11).root
519 assert 0 < min_beta_1 < np.pi / 2
520
521
522 # %%
523 # Solves the minimization problem
524
525 # Initial guess based on minimum beta angle (scaled to avoid evaluating on the boundary)
526 initial_betas = (min_beta_1 * 1.01, min_beta_1 * 1.01)
527
528 print("Solving...")

```

```

529 # Minimizes the total pressure efficiency loss using a descent algorithm
530 sol_weak = minimize(
531     lambda betas: 1 - nShock_p0_eff(betas, flow_inf),
532     initial_betas,
533     method="SLSQP",
534     options={"disp": True},
535 )
536 print("Done!")
537 print(f"Solution: {sol_weak.x}")
538
539
540 # %% [markdown]
541 # ## Results
542
543 # %% [markdown]
544 # ### Optimal geometry and flow
545
546 # %%
547 # Gets and prints all the solution values
548 betas = sol_weak.x
549 assert np.all(0 < betas) and np.all(betas < np.pi / 2)
550
551 p0_eff = 1 - sol_weak.fun
552 assert 0 < p0_eff < 1
553
554 print(f"Efficiency = {p0_eff:.4}")
555 print()
556
557 # Stores flow states into a list
558 flows = [flow_inf]
559 for n, beta in enumerate(betas):
560     print(f"beta_{n+1}: {np.rad2deg(beta)}")
561     print(f"theta_{n+1}: {np.rad2deg(flows[-1].velocity_deflection(beta))}")
562     flows.append(flows[-1].shockwave(beta))
563     print(f"Flow_{n+1}:", flows[-1], "\n")
564 flows.append(flows[-1].shockwave(np.pi / 2))
565 print(f"Flow_out: {flows[-1]}")
566 # Garbage Collection
567 gc.collect()
568
569
570 # %%
571 print(flows[-1].temperature)
572 print(flows[-1].pressure)
573
574 # %% [markdown]
575 # ### Plot minimization locus
576
577 # %%

```

```

578 # Calculates the efficiency over a grid of beta_1 and beta_2
579 b1v = np.linspace(min_beta_1, np.deg2rad(65), 50)
580 b2v = np.linspace(min_beta_1, np.deg2rad(80), 50)
581 d = np.empty((len(b2v), len(b1v)))
582 for j, b1 in enumerate(b1v):
583     for i, b2 in enumerate(b2v):
584         d[i, j] = nShock_p0_eff((b1, b2), flow_inf)
585 assert np.nanmax(d) < p0_eff
586 assert np.nanmin(d) > 0
587 # Garbage collection for all the dereferenced values
588 gc.collect()
589
590
591 # %%
592 # Plots the efficiency distribution
593 plt.figure(figsize=(10, 10))
594 plt.pcolormesh(
595     np.rad2deg(b1v),
596     np.rad2deg(b2v),
597     100 * d,
598     cmap="viridis",
599 )
600 cb = plt.colorbar(label="$p_0$ Efficiency (%)")
601
602 # Plots optimal solution in red X
603 plt.plot(*np.rad2deg(betas[:2]), "rx", label="Optimal solution")
604
605 # plt.title(r"Total Pressure Efficiency as a function of $\beta_1$ and $\beta_2$")
606 plt.xlabel(r"$\beta_1$ [deg]")
607 plt.ylabel(r"$\beta_2$ [deg]")
608 plt.legend()
609
610 # Grid and ticks
611 plt.axis("scaled")
612 plt.minorticks_on()
613 plt.grid(visible=True, axis="both", which="major", linewidth=1, alpha=0.6)
614 plt.grid(visible=True, axis="both", which="minor", linewidth=0.5, alpha=0.3)
615
616 # Color bar
617 plt.savefig("gradient.pdf", bbox_inches="tight")
618 plt.show()
619
620
621 # %% [markdown]
622 # ### Plot optimal flow properties
623
624 # %%
625 # Prepares the data as a matrix
626 data = np.stack(

```

```

627     [(f.static_values(), f.aux_values(), f.total_values()) for f in flows]
628 )
629 # Saves data to CSV file
630 np.savetxt("data.csv", data, fmt="%8.5g", delimiter=",")
631
632
633 # %%
634 # Plots the flow variation over the diffuser
635 xt = np.arange(len(data))
636 with plt.style.context(styles), plt.rc_context(
637     rc={"lines.marker": "o", "lines.markersize": 10}
638 ):
639     fig, axes = plot_flow_evolution(data, xt, cols=[0, 1, 2, 3, 5])
640     # Sets the lower axis
641     axes[0].set_xlabel("Region")
642     axes[0].set_xticks(xt)
643     axes[0].set_xticklabels([r"$\infty$", *np.arange(1, len(data))])
644 plt.savefig("diffuser_properties.pdf", bbox_inches="tight")
645 plt.show()
646
647
648 # %% [markdown]
649 # # PART 2
650
651 # %% [markdown]
652 # ## Initial data
653
654 # %%
655 # Sets problem parameters
656 # The cross-sectional area after the normal shockwave can be arbitrarily small. For this
657 # geometry, it can also be as large as about  $0.3 * D$ .
658
659 # D was chosen to be unitary;
660 D = 1
661 D_inlet = 1 * D
662 D_outlet = 1 * D
663 # The cross sectional area was arbitrarily chosen to be  $0.25 * D$ 
664 D_entrance =  $0.3 * D$ 
665 # Temperature change
666 delta_temp = 1500
667 # Total Values from Part 1
668 p0, T0, rho0 = flows[-1].total_values()
669 # Creates new Flow list for Part 2
670 flows2 = [flows[-1]]
671 print("Flow after diffuser:", flows2[-1])
672
673
674 # %% [markdown]
675 # ## Expansion

```

```

676
677 # %%
678 # Calculates Mach number using the area-Mach relation
679 # Calculates virtual throat
680 D_virtual = D_entrance / area_mach(flows2[-1].mach, flows2[-1].fluid.gamma)
681 # Solves area-Mach relation for the new Mach number
682 expanded_mach = root_scalar(
683     lambda M: area_mach(M, flows2[-1].fluid.gamma) - D / D_virtual,
684     bracket=[1, 1e-2],
685     method="brentq",
686 ).root
687
688 # Appends new Flow to the list
689 flows2.append(Flow.from_total_values(p0, expanded_mach, T0, rho0, flows2[-1].fluid))
690 print("Flow after expansion:", flows2[-1])
691
692
693 # %%
694 print(D_virtual / D)
695
696 # %% [markdown]
697 # ## Heat addition
698
699 # %% [markdown]
700 # To calculate the properties after heat addition, equations (3.78), (3.79), (3.81) from
701     ↪ @anderson2021. These equations are written down here as needed before they are
702     ↪ implemented.
703
704 # %% [markdown]
705 # $$$\tag{3.81}
706 # \frac{T_2}{T_1} = \left( \frac{1 + \gamma M_1^2}{\gamma M_2^2} \right)^2
707 # \left( \frac{M_2^2}{M_1^2} \right)
708 # $$$
709
710 # %%
711 # Calculates temperature after heat addition
712 expanded_temp = flows2[-1].temperature
713 heated_temp = expanded_temp + delta_temp
714
715 # Calculates Mach number after heat addition
716 # Part of Equation (3.81) @anderson2021 that doesn't depend on the new Mach number
717 const_part = (
718     np.sqrt(heated_temp / expanded_temp)
719     * expanded_mach
720     / (1 + flows2[-1].fluid.gamma * expanded_mach * expanded_mach)
721 )
722 # Solves Equation (3.81) @anderson2021 (modified) for the new Mach number
723 heated_mach = root_scalar(
724     lambda M: const_part - M / (1 + flows2[-1].fluid.gamma * M * M),

```



```

723     bracket=[1, 0],
724     method="brentq",
725 ).root
726 print(f"New Mach number: {heated_mach:.4}")
727
728
729 # %% [markdown]
730 # $$\tag{3.78}
731 # \frac{p_2}{p_1} = \frac{1 + \gamma M_1^2}{1 + \gamma M_2^2}
732 # $$
733 #
734 # $$\tag{3.79}
735 # \frac{\rho_2}{\rho_1} = \frac{p_2}{p_1} \frac{T_1}{T_2}
736 # $$
737
738 # %%
739 # Calculates remaining Flow properties after heat addition
740 # Equation (3.78) @anderson2021
741 heated_pressure = (
742     flows2[-1].pressure
743     * (1 + flows2[-1].fluid.gamma * expanded_mach * expanded_mach)
744     / (1 + flows2[-1].fluid.gamma * heated_mach * heated_mach)
745 )
746 # Equation (3.79) @anderson2021
747 heated_rho = (
748     flows2[-1].density
749     * heated_pressure
750     / flows2[-1].pressure
751     * expanded_temp
752     / heated_temp
753 )
754
755 # Appends new Flow to the list
756 flows.append(
757     Flow(heated_pressure, heated_mach, heated_temp, heated_rho, flows2[-1].fluid)
758 )
759 print("Flow after heat addition:", flows[-1])
760
761
762 # %% [markdown]
763 # ## Nozzle
764
765 # %% [markdown]
766 # ### Geometry
767
768 # %%
769 # Calculates throat semi-area
770 D_throat = D / area_mach(heated_mach, flows[-1].fluid.gamma)
771 print(f"Throat area: {D_throat}")

```

```

772
773
774 # %%
775 # Creates the 2 nozzle geometries
776 bellFunc, crit_len, max_len = bellNozzleGenerator(
777     1.2 * D_inlet, D_throat, 1.5 * D_outlet, D_throat, D_inlet, D_outlet
778 )
779 coneFunc, _, _ = coneNozzleGenerator(crit_len, max_len, D_throat, D_inlet, D_outlet)
780
781 x = np.hstack(
782     [
783         np.linspace(0, crit_len, 200, endpoint=False),
784         np.linspace(crit_len, max_len, 200),
785     ],
786 )
787 bell = np.array([x, bellFunc(x)])
788 cone = np.array([x, coneFunc(x)])
789 nozzle_curves = [bell, cone]
790
791
792 # %%
793 for x, y in nozzle_curves:
794     plt.figure(figsize=(10, 10))
795     plt.fill_between(x, y, 0, facecolor="#aaa", edgecolor="black", linewidth=2)
796     plt.axis("scaled")
797     plt.axis("off")
798     plt.tight_layout()
799 plt.show()
800
801
802 # %% [markdown]
803 # ### Flow properties
804
805 # %%
806 nozzles_flows = np.empty([len(nozzle_curves), len(x)], Flow)
807 p0_nozzle, T0_nozzle, rho0_nozzle = flows2[-1].total_values()
808 for n, curve in enumerate(nozzle_curves):
809     # Calculates the Mach number across the nozzle.
810     crit_index = np.argmin(curve[1])
811     D_crit = curve[1, crit_index]
812     crit_x = curve[0, crit_index]
813     machs = np.array(
814         [
815             root_scalar(
816                 lambda M: area_mach(M, flows2[-1].fluid.gamma) - y_ / D_crit,
817                 # The sign function selects solution branch according to starting flow
818                 # and position relative to critical length
819                 bracket=[1, 100 * np.sign((1 - flows2[-1].mach) * (x_ - crit_x))],
820                 method="brentq",

```

```

821         ).root
822         for x_, y_ in curve.T
823     ]
824 )
825 nozzles_flows[n, :] = np.array(
826     [
827         Flow.from_total_values(
828             p0_nozzle, mach, T0_nozzle, rho0_nozzle, flows2[-1].fluid
829         )
830         for mach in machs
831     ]
832 )
833
834
835 # %% [markdown]
836 # ### Contour plots
837
838 # %%
839 def plot_nozzle_prop(prop_name, flow_list, nozzle_curve, ax=None):
840     from mpl_toolkits.axes_grid1.inset_locator import inset_axes
841
842     if ax is None:
843         ax = plt.gca()
844     prop = np.array([flow.__getattribute__(prop_name) for flow in flow_list])
845     x, y = nozzle_curve
846     plot = plt.contourf([x, x], [np.zeros_like(y), y], [prop, prop], levels=10)
847     plt.axis("scaled")
848     # cb = plt.colorbar(label=prop_name, orientation="vertical")
849     axins = inset_axes(
850         plt.gca(),
851         width="5%",
852         height="100%",
853         loc="lower left",
854         bbox_to_anchor=(1.02, 0.0, 1, 1),
855         bbox_transform=ax.transAxes,
856         borderpad=0,
857     )
858     cb = plt.gcf().colorbar(plot, cax=axins, label=prop_name)
859     cb.formatter.set_powerlimits((-3, 3))
860     cb.formatter.set_scientific(True)
861
862
863 def plot_nozzles_props(prop_list, nozzle_flow_list, curve_list):
864     for property in prop_list:
865         for nozzle_flow, curve in zip([*nozzle_flow_list], curve_list):
866             plt.figure()
867             plot_nozzle_prop(property, nozzle_flow, curve)
868 plt.show()
869

```

```

870
871 # %%
872 with plt.rc_context({"figure.figsize": (10, 10)}):
873     plot_nozzles_props(
874         ["pressure", "mach", "temperature", "density", "vel"],
875         nozzles_flows,
876         nozzle_curves,
877     )
878
879
880 # %% [markdown]
881 # ### Line plot
882
883 # %%
884 with plt.style.context(styles):
885     fig, ax = plt.subplots(figsize=(10, 10))
886     for nozzle_flows, line_style in zip(nozzles_flows, ["dashed", "solid", "-."]):
887         with plt.rc_context(
888             {"lines.linestyle": line_style, "lines.marker": "", "lines.linewidth": 2}
889         ):
890             data2 = np.stack(
891                 [
892                     (*f.static_values(), *f.aux_values(), *f.total_values())
893                     for f in nozzle_flows
894                 ]
895             )
896             _, axes = plot_flow_evolution(data2, x, (fig, ax), cols=[0, 1, 2, 3, 5])
897             plt.axvline(crit_len, color="black", linestyle="dotted", linewidth=2, alpha=0.5)
898             plt.savefig("nozzle_evolution.pdf", bbox_inches="tight")
899             plt.show()
900
901
902 # %%
903 with plt.style.context("default"):
904     plt.figure(figsize=(10, 10))
905     for prop, color in zip(
906         ["pressure", "mach", "temperature", "density", "vel"], colors
907     ):
908         plt.title(prop)
909         for nozzle_flows, nozzle_names, line_style in zip(
910             nozzles_flows, ["Bell", "Cone"], ["-", "--"]
911         ):
912             y = [f.__getattr__(prop) for f in nozzle_flows]
913             y = y / y[crit_index]
914             plt.plot(
915                 x,
916                 y,
917                 linestyle=line_style,
918                 color=color,

```

```
919         label=f"{prop} {nozzle_names}",
920     )
921     plt.ticklabel_format(scilimits=(-3, 3))
922     plt.autoscale()
923     plt.ylim(bottom=0)
924     plt.legend()
```