proptools Documentation

Release 0.0.0

Matthew Vernacchia

Sep 22, 2019

Contents

Proptools is a Python package for preliminary design of rocket propulsion systems.

Proptools provides implementations of equations for nozzle flow, turbo-machinery and rocket structures. The project aims to cover most of the commonly used equations in *Rocket Propulsion Elements* by George Sutton and Oscar Biblarz and *Modern Engineering for Design of Liquid-Propellant Rocket Engines* by Dieter Huzel and David Huang.

Proptools can be used as a desktop calculator:

```
>> from proptools import nozzle
>> p_c = 10e6; p_e = 100e3; gamma = 1.2; m_molar = 20e-3; T_c = 3000.
>> C_f = nozzle.thrust_coef(p_c, p_e, gamma)
>> c_star = nozzle.c_star(gamma, m_molar, T_c)
>> I_sp = C_f * c_star / nozzle.g
>> print "The engine's ideal sea level specific impulse is {:.1f} seconds.".format(I_
↪sp)
The engine's ideal sea level specific impulse is 288.7 seconds.
```

Proptools can also be used as a library in other propulsion design and analysis software. It is distributed under a MIT License and can be used in commercial projects.

CHAPTER 1

Tutorials

1.1 Nozzle Flow

Nozzle flow theory can predict the thrust and specific impulse of a rocket engine. The following example predicts the performance of an engine which operates at chamber pressure of 10 MPa, chamber temperature of 3000 K, and has 100 mm diameter nozzle throat.

```
"""Estimate specific impulse, thrust and mass flow."""
from math import pi
from proptools import nozzle

# Declare engine design parameters
p_c = 10e6    # Chamber pressure [units: pascal]
p_e = 100e3    # Exit pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]
m_molar = 20e-3    # Exhaust molar mass [units: kilogram mole**1]
T_c = 3000.    # Chamber temperature [units: kelvin]
A_t = pi * (0.1 / 2)**2    # Throat area [units: meter**2]

# Predict engine performance
C_f = nozzle.thrust_coef(p_c, p_e, gamma)    # Thrust coefficient [units:␣
↪dimensionless]
c_star = nozzle.c_star(gamma, m_molar, T_c)    # Characteristic velocity [units:␣
↪meter second**-1]
I_sp = C_f * c_star / nozzle.g    # Specific impulse [units: second]
F = A_t * p_c * C_f    # Thrust [units: newton]
m_dot = A_t * p_c / c_star    # Propellant mass flow [units: kilogram second**-1]

print 'Specific impulse = {:.1f} s'.format(I_sp)
print 'Thrust = {:.1f} kN'.format(F * 1e-3)
print 'Mass flow = {:.1f} kg s**-1'.format(m_dot)
```

Output:

```
Specific impulse = 288.7 s

Thrust = 129.2 kN

Mass flow = 45.6 kg s**-1
```

The rest of this page derives the nozzle flow theory, and demonstrates other features of `proptools.nozzle`.

1.1.1 Ideal Nozzle Flow

The purpose of a rocket is to generate thrust by expelling mass at high velocity. The rocket nozzle is a flow device which accelerates gas to high velocity before it is expelled from the vehicle. The nozzle accelerates the gas by converting some of the gas's thermal energy into kinetic energy.

Ideal nozzle flow is a simplified model of the aero- and thermo-dynamic behavior of fluid in a nozzle. The ideal model allows us to write algebraic relations between an engine's geometry and operating conditions (e.g. throat area, chamber pressure, chamber temperature) and its performance (e.g. thrust and specific impulse). These equations are fundamental tools for the preliminary design of rocket propulsion systems.

The assumptions of the ideal model are:

1. The fluid flowing through the nozzle is gaseous.
2. The gas is homogeneous, obeys the ideal gas law, and is calorically perfect. Its molar mass ($\mathcal{M}$) and heat capacities (c_p, c_v) are constant throughout the fluid, and do not vary with temperature.
3. There is no heat transfer to or from the gas. Therefore, the flow is adiabatic. The specific enthalpy h is constant throughout the nozzle.
4. There are no viscous effects or shocks within the gas or at its boundaries. Therefore, the flow is reversible. If the flow is both adiabatic and reversible, it is isentropic: the specific entropy s is constant throughout the nozzle.
5. The flow is steady; $\frac{d}{dt} = 0$.
6. The flow is quasi one dimensional. The flow state varies only in the axial direction of the nozzle.
7. The flow velocity is axially directed.
8. The flow does not react in the nozzle. The chemical equilibrium established in the combustion chamber does not change as the gas flows through the nozzle. This assumption is known as "frozen flow".

These assumptions are usually acceptably accurate for preliminary design work. Most rocket engines perform within 1% to 6% of the ideal model predictions [RPE].

1.1.2 Isentropic Relations

Under the assumption of isentropic flow and calorically perfect gas, there are several useful relations between fluid states. These relations depend on the heat capacity ratio, $\gamma = c_p/c_v$. Consider two gas states, 1 and 2, which are isentropically related ($s_1 = s_2$). The states' pressure, temperature and density ratios are related:

$$\frac{p_1}{p_2} = \left(\frac{\rho_1}{\rho_2}\right)^{\gamma} = \left(\frac{T_1}{T_2}\right)^{\frac{\gamma}{\gamma-1}}$$

Stagnation state

Now consider the relation between static and stagnation states in a moving fluid. The stagnation state is the state a moving fluid would reach if it were isentropically decelerated to zero velocity. The stagnation enthalpy h_0 is the sum of the static enthalpy and the specific kinetic energy:

$$h_0 = h + \frac{1}{2}v^2$$

For a calorically perfect gas, $T = h/c_p$, and the stagnation temperature is:

$$T_0 = T + \frac{v^2}{2c_p}$$

It is helpful to write the fluid properties in terms of the Mach number M, instead of the velocity. Mach number is the velocity normalized by the local speed of sound, $a = \sqrt{\gamma RT}$. In terms of Mach number, the stagnation temperature is:

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

Because the static and stagnation states are isentropically related, $\frac{p_0}{p} = \left(\frac{T_0}{T}\right)^{\frac{\gamma}{\gamma-1}}$. Therefore, the stagnation pressure is:

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

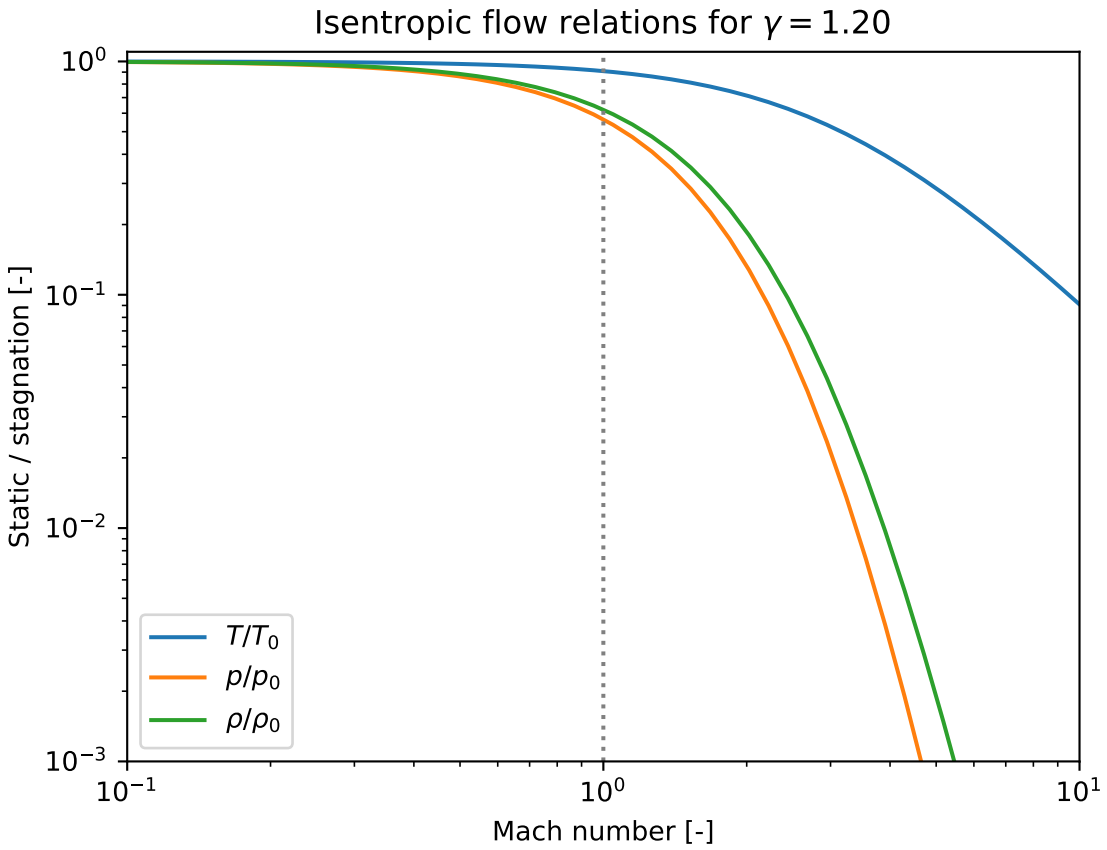
Use `proptools` to plot the stagnation state variables against Mach number:

```
"""Plot isentropic relations."""
import numpy as np
from matplotlib import pyplot as plt
from proptools import isentropic

M = np.logspace(-1, 1)
gamma = 1.2

plt.loglog(M, 1 / isentropic.stag_temperature_ratio(M, gamma), label='$T / T_0$')
plt.loglog(M, 1 / isentropic.stag_pressure_ratio(M, gamma), label='$p / p_0$')
plt.loglog(M, 1 / isentropic.stag_density_ratio(M, gamma), label='$\\rho / \\rho_0$')
plt.xlabel('Mach number [-]')
plt.ylabel('Static / stagnation [-]')
plt.title('Isentropic flow relations for $\gamma={:.2f}$'.format(gamma))
plt.xlim(0.1, 10)
plt.ylim([1e-3, 1.1])
plt.axvline(x=1, color='grey', linestyle=':')
plt.legend(loc='lower left')
plt.show()
```

Exit velocity

The exit velocity of the exhaust gas is the fundamental measure of efficiency for rocket propulsion systems, as the rocket equation shows. We can now show a basic relation between the exit velocity and the combustion conditions of the rocket. First, use the conservation of energy to relate the velocity at any two points in the flow:

$$v_2 = \sqrt{2(h_1 - h_2) + v_1^2}$$


Isentropic flow relations for $\gamma = 1.20$
Static / stagnation [-]
Mach number [-]
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-1}
10^{0}
10^{1}
T/T_0
p/p_0
ρ/ρ_0

We can replace the enthalpy difference with an expression of the pressures and temperatures, using the isentropic relations.

$$v_2 = \sqrt{\frac{2\gamma}{\gamma-1} R T_1 \left(1 - \left(\frac{p_2}{p_1}\right)^{\frac{\gamma-1}{\gamma}}\right) + v_1^2}$$

Set state 1 to be the conditions in the combustion chamber: $T_1 = T_c, p_1 = p_c, v_1 \approx 0$. Set state 2 to be the state at the nozzle exit: $p_2 = p_e, v_2 = v_e$. This gives the exit velocity:

$$\begin{aligned} v_e &= \sqrt{\frac{2\gamma}{\gamma-1} R T_c \left(1 - \left(\frac{p_e}{p_c}\right)^{\frac{\gamma-1}{\gamma}}\right)} \\ &= \sqrt{\frac{2\gamma}{\gamma-1} \mathcal{R} \frac{T_c}{\mathcal{M}} \left(1 - \left(\frac{p_e}{p_c}\right)^{\frac{\gamma-1}{\gamma}}\right)} \end{aligned}$$

where $\mathcal{R} = 8.314$ J mol $^{-1}$ K $^{-1}$ is the universal gas constant and $\mathcal{M}$ is the molar mass of the exhaust gas. Temperature and molar mass have the most significant effect on exit velocity. To maximize exit velocity, a rocket should burn propellants which yield a high flame temperature and low molar mass exhaust. This is why many rockets burn hydrogen and oxygen: they yield a high flame temperature, and the exhaust (mostly H_2 and H_2O) is of low molar mass.

The pressure ratio p_e/p_c is usually quite small. As the pressure ratio goes to zero, the exit vleocity approaches its maximum for a given $T_c, \mathcal{M}$ and γ.

$$\frac{p_e}{p_c} \to 0 \quad \Rightarrow \quad 1 - \left(\frac{p_e}{p_c}\right)^{\frac{\gamma-1}{\gamma}} \to 1$$

If $p_e \ll p_c$, the pressures have a weak effect on exit velocity.

The heat capacity ratio γ has a weak effect on exit velocity. Decreasing γ increases exit velocity.

Use `proptools` to find the exit velocity of the example engine:

```
"""Estimate exit velocity."""
from proptools import isentropic

# Declare engine design parameters
p_c = 10e6    # Chamber pressure [units: pascal]
p_e = 100e3    # Exit pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]
m_molar = 20e-3    # Exhaust molar mass [units: kilogram mole**1]
T_c = 3000.    # Chamber temperature [units: kelvin]

# Compute the exit velocity
v_e = isentropic.velocity(v_1=0, p_1=p_c, T_1=T_c, p_2=p_e, gamma=gamma, m_molar=m_
↪molar)

print 'Exit velocity = {:.0f} m s**-1'.format(v_e)
```

```
Exit velocity = 2832 m s**-1
```

1.1.3 Mach-Area Relation

Using the isentropic relations, we can find how the Mach number of the flow varies with the cross sectional area of the nozzle. This allows the design of a nozzle geometry which will accelerate the flow the high speeds needed for rocket propulsion.

Start with the conservation of mass. Because the flow is quasi- one dimensional, the mass flow through every cross-section of the nozzle must be the same:

$$\dot{m} = Av\rho = \text{const}$$

where A is the cross-sectional area of the nozzle flow passage (normal to the flow axis). To conserve mass, the ratio of areas between any two points along the nozzle axis must be:

$$\frac{A_1}{A_2} = \frac{v_2\rho_2}{v_1\rho_1}$$

Use the isentropic relations to write the velocity and density in terms of Mach number, and simplify:

$$\frac{A_1}{A_2} = \frac{M_2}{M_1}\left(\frac{1+\frac{\gamma-1}{2}M_1^2}{1+\frac{\gamma-1}{2}M_2^2}\right)^{\frac{\gamma+1}{2(\gamma-1)}}$$

We can use `proptools` to plot Mach-Area relation. Let $M_2 = 1$ and plot A_1/A_2 vs M_1:

```
"""Plot the Mach-Area relation."""
import numpy as np
from matplotlib import pyplot as plt
from proptools import nozzle

M_1 = np.linspace(0.1, 10)     # Mach number [units: dimensionless]
gamma = 1.2     # Exhaust heat capacity ratio [units: dimensionless]

area_ratio = nozzle.area_from_mach(M_1, gamma)

plt.loglog(M_1, area_ratio)
plt.xlabel('Mach number $M_1$ [-]')
plt.ylabel('Area ratio $A_1 / A_2$ [-]')
plt.title('Mach-Area relation for $M_2 = 1$')
plt.ylim([1, 1e3])
plt.show()
```

We see that the nozzle area has a minimum at $M = 1$. At subsonic speeds, Mach number increases as the area is decreased. At supersonic speeds ($M > 1$), Mach number increases as area increases. For a flow passage to accelerate gas from subsonic to supersonic speeds, it must first decrease in area, then increase in area. Therefore, most rocket nozzles have a convergent-divergent shape. Larger expansions of the divergent section lead to higher exit Mach numbers.

1.1.4 Choked Flow

The station of minimum area in a converging-diverging nozzle is known as the *nozzle throat*. If the pressure ratio across the nozzle is at least:

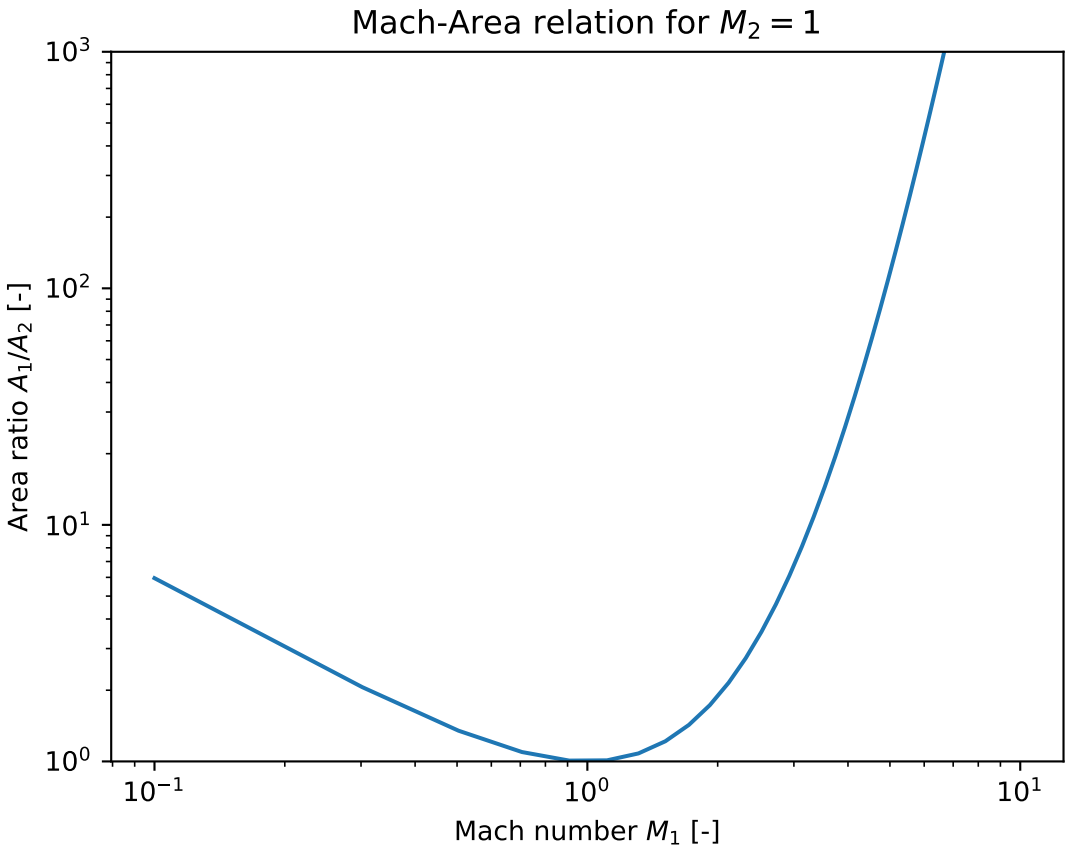
$$\frac{p_c}{p_e} > \left(\frac{\gamma+1}{2}\right)^{\frac{\gamma}{\gamma-1}} \sim 1.8$$

then the flow at the throat will be sonic ($M = 1$) and the flow in the diverging section will be supersonic. The velocity at the throat is:

$$v_t = \sqrt{\gamma R T_t} = \sqrt{\frac{2\gamma}{\gamma+1} R T_c}$$

The mass flow at the sonic throat (i.e. through the nozzle) is:

$$\dot{m} = A_t v_t \rho_t = A_t p_c \frac{\gamma}{\sqrt{\gamma R T_c}}\left(\frac{2}{\gamma+1}\right)^{\frac{\gamma+1}{2(\gamma-1)}}$$


Mach-Area relation for $M_2 = 1$
10^3
10^2
10^1
10^0
Area ratio A_1/A_2 [-]
10^{-1}
10^0
10^1
Mach number M_1 [-]

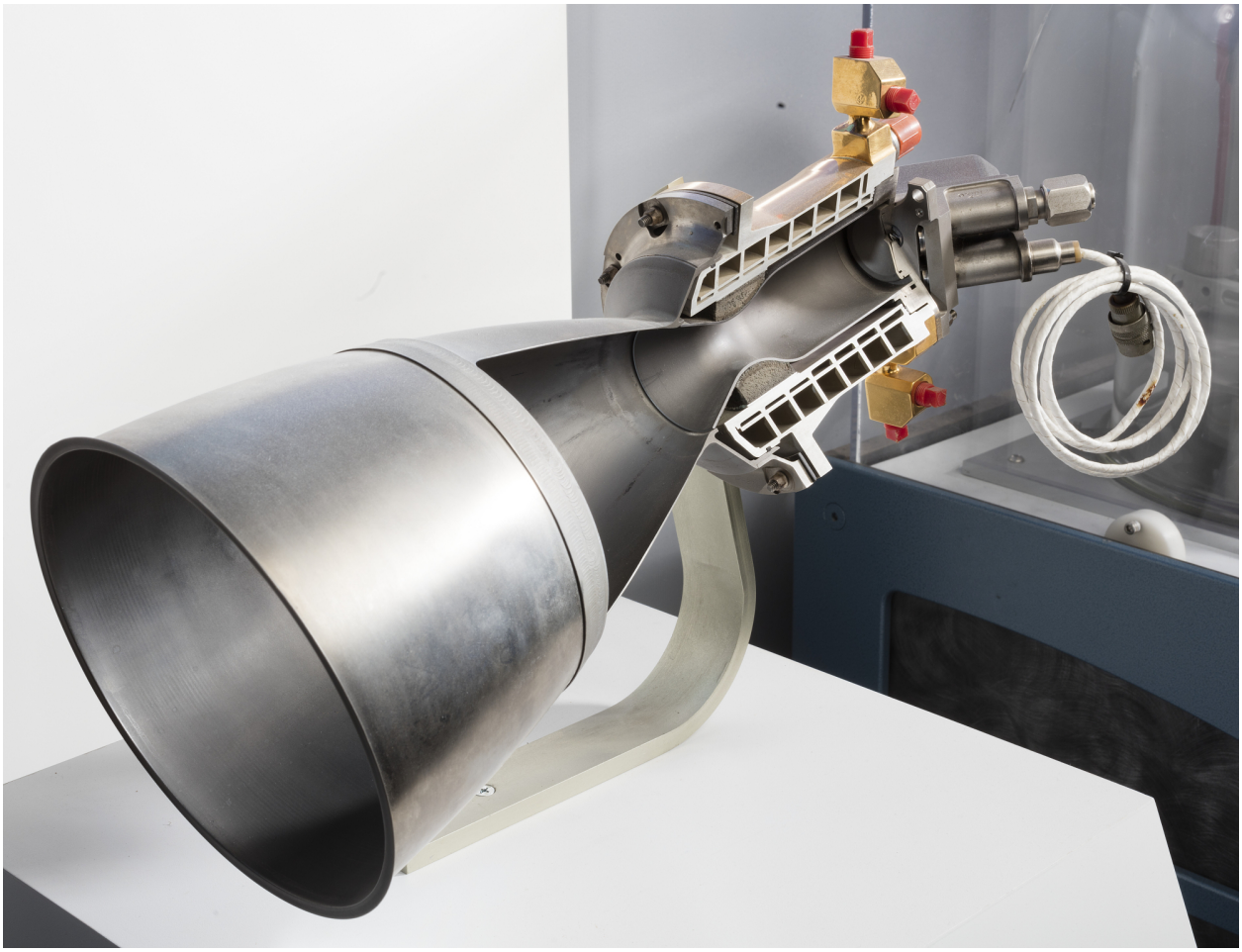


Fig. 1: The typical converging-diverging shape of rocket nozzles is shown in this cutaway of the Thiokol C-1 engine. Image credit: Smithsonian Institution, National Air and Space Museum

Notice that the mass flow does not depend on the exit pressure. If the exit pressure is sufficiently low to produce sonic flow at the throat, the nozzle is *choked* and further decreases in exit pressure will not alter the mass flow. Increasing the chamber pressure increases the density at the throat, and therefore will increase the mass flow which can "fit through" the throat. Increasing the chamber temperature increases the throat velocity but decreases the density by a larger amount; the net effect is to decrease mass flow as $1/\sqrt{T_c}$.

Use `proptools` to compute the mass flow of the example engine:

```
"""Check that the nozzle is choked and find the mass flow."""
from math import pi
from proptools import nozzle

# Declare engine design parameters
p_c = 10e6    # Chamber pressure [units: pascal]
p_e = 100e3    # Exit pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]
m_molar = 20e-3    # Exhaust molar mass [units: kilogram mole**1]
T_c = 3000.    # Chamber temperature [units: kelvin]
A_t = pi * (0.1 / 2)**2    # Throat area [units: meter**2]

# Check choking
if nozzle.is_choked(p_c, p_e, gamma):
    print 'The flow is choked'

# Compute the mass flow [units: kilogram second**-1]
m_dot = nozzle.mass_flow(A_t, p_c, T_c, gamma, m_molar)

print 'Mass flow = {:.1f} kg s**-1'.format(m_dot)
```

```
The flow is choked

Mass flow = 45.6 kg s**-1
```

1.1.5 Thrust

The thrust force of a rocket engine is equal to the momentum flow out of the nozzle plus a pressure force at the nozzle exit:

$$F = \dot{m} v_e + (p_e - p_a) A_e$$

where p_a is the ambient pressure and A_e is the nozzle exit area. We can rewrite this in terms of the chamber pressure:

$$F = A_t p_c \sqrt{\frac{2\gamma^2}{\gamma - 1} \left(\frac{2}{\gamma + 1}\right)^{\frac{\gamma+1}{\gamma-1}} \left(1 - \left(\frac{p_e}{p_c}\right)^{\frac{\gamma-1}{\gamma}}\right)} + (p_e - p_a) A_e$$

Note that thrust depends only on γ and the nozzle pressures and areas; not chamber temperature.

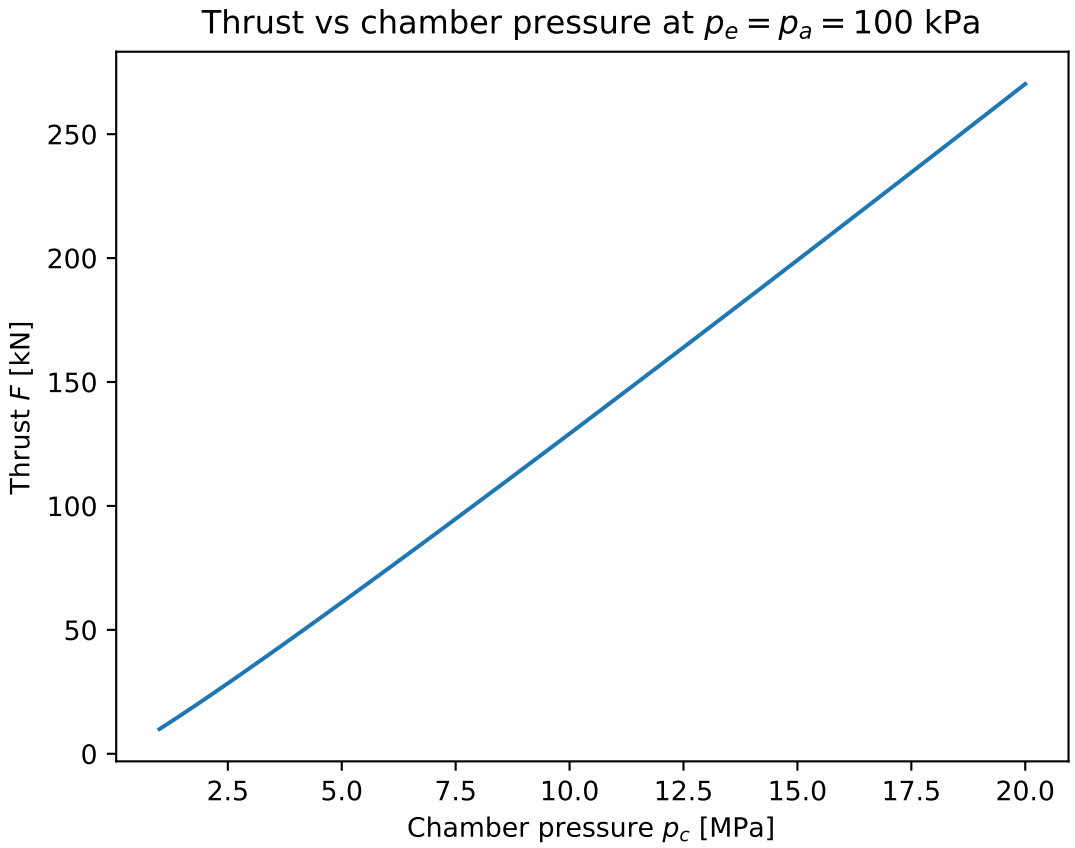
Use `proptools` to plot thrust versus chamber pressure for the example engine:

```
"""Plot thrust vs chmaber pressure."""

import numpy as np
from matplotlib import pyplot as plt
from proptools import nozzle
```

(continues on next page)

(continued from previous page)

```
p_c = np.linspace(1e6, 20e6)     # Chamber pressure [units: pascal]
p_e = 100e3     # Exit pressure [units: pascal]
p_a = 100e3     # Ambient pressure [units: pascal]
gamma = 1.2     # Exhaust heat capacity ratio [units: dimensionless]
A_t = np.pi * (0.1 / 2)**2     # Throat area [units: meter**2]

# Compute thrust [units: newton]
F = nozzle.thrust(A_t, p_c, p_e, gamma)

plt.plot(p_c * 1e-6, F * 1e-3)
plt.xlabel('Chamber pressure $p_c$ [MPa]')
plt.ylabel('Thrust $F$ [kN]')
plt.title('Thrust vs chamber pressure at $p_e = p_a = {:.0f}$ kPa'.format(p_e * 1e-3))
plt.show()
```



Note that thrust is almost linear in chamber pressure.

We can also explore the variation of thrust with ambient pressure for fixed p_c, p_e:

```
"""Plot thrust vs ambient pressure."""
import numpy as np
from matplotlib import pyplot as plt
import skaero.atmosphere.coesa as atmo
```

(continues on next page)

(continued from previous page)

```
from proptools import nozzle

p_c = 10e6   # Chamber pressure [units: pascal]
p_e = 100e3    # Exit pressure [units: pascal]
p_a = np.linspace(0, 100e3)    # Ambient pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]
A_t = np.pi * (0.1 / 2)**2    # Throat area [units: meter**2]

# Compute thrust [units: newton]
F = nozzle.thrust(A_t, p_c, p_e, gamma,
                  p_a=p_a, er=nozzle.er_from_p(p_c, p_e, gamma))

ax1 = plt.subplot(111)
plt.plot(p_a * 1e-3, F * 1e-3)
plt.xlabel('Ambient pressure $p_a$ [kPa]')
plt.ylabel('Thrust $F$ [kN]')
plt.suptitle('Thrust vs ambient pressure at $p_c = {:.0f}$ MPa, $p_e = {:.0f}$ kPa'.
↪format(
    p_c *1e-6, p_e * 1e-3))

# Add altitude on second axis
ylim = plt.ylim()
ax2 = ax1.twiny()
new_tick_locations = np.array([100, 75, 50, 25, 1])
ax2.set_xlim(ax1.get_xlim())
ax2.set_xticks(new_tick_locations)


def tick_function(p):
    """Map atmospheric pressure [units: kilopascal] to altitude [units: kilometer]."""
    h_table = np.linspace(84e3, 0)    # altitude [units: meter]
    p_table = atmo.pressure(h_table)    # atmo pressure [units: pascal]
    return np.interp(p * 1e3, p_table, h_table) * 1e-3


ax2.set_xticklabels(['{:.0f}'.format(h) for h in tick_function(new_tick_locations)])
ax2.set_xlabel('Altitude [km]')
ax2.tick_params(axis='y', direction='in', pad=-25)
plt.subplots_adjust(top=0.8)
plt.show()
```

1.1.6 Thrust coefficient

We can normalize thrust by $A_t p_c$ to give a non-dimensional measure of nozzle efficiency, which is independent of engine size or power level. This is the *thrust coefficient*, C_F:

$$C_F \equiv \frac{F}{A_t p_c}$$

For an ideal nozzle, the thrust coefficient is:

$$C_F = \sqrt{\frac{2\gamma^2}{\gamma-1}\left(\frac{2}{\gamma+1}\right)^{\frac{\gamma+1}{\gamma-1}}\left(1-\left(\frac{p_e}{p_c}\right)^{\frac{\gamma-1}{\gamma}}\right)} + \frac{p_e - p_a}{p_c}\frac{A_e}{A_t}$$

Note that C_F is independent of the combustion temperature or the engine size. It depends only on the heat capacity ratio, nozzle pressures, and expansion ratio (A_e/A_t). Therefore, C_F is a figure of merit for the nozzle expansion

process. It can be used to compare the efficiency of different nozzle designs on different engines. Values of C_F are generally between 0.8 and 2.2, with higher values indicating better nozzle performance.

1.1.7 Expansion Ratio

The expansion ratio is an important design parameter which affects nozzle efficiency. It is the ratio of exit area to throat area:

$$\epsilon \equiv \frac{A_e}{A_t}$$

The expansion ratio appears directly in the equation for thrust coefficient. The expansion ratio also allows the nozzle designer to set the exit pressure. The relation between expansion ratio and pressure ratio can be found from mass conservation and the isentropic relations:

$$\begin{aligned}\epsilon &= \frac{A_e}{A_t} = \frac{\rho_t v_t}{\rho_e v_e} \\ &= \left(\left(\frac{\gamma+1}{2} \right)^{\frac{1}{\gamma-1}} \left(\frac{p_e}{p_c} \right)^{\frac{1}{\gamma}} \sqrt{\frac{\gamma+1}{\gamma-1} \left(1 - \left(\frac{p_e}{p_c} \right)^{\frac{\gamma-1}{\gamma}} \right)} \right)^{-1}\end{aligned}$$

This relation is implemented in `proptools`:

```
"""Compute the expansion ratio for a given pressure ratio."""
from proptools import nozzle

p_c = 10e6    # Chamber pressure [units: pascal]
p_e = 100e3    # Exit pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]

# Solve for the expansion ratio [units: dimensionless]
exp_ratio = nozzle.er_from_p(p_c, p_e, gamma)

print 'Expansion ratio = {:.1f}'.format(exp_ratio)
```

```
Expansion ratio = 11.9
```

We can also solve the inverse problem:

```
"""Compute the pressure ratio from a given expansion ratio."""
from scipy.optimize import fsolve
from proptools import nozzle

p_c = 10e6    # Chamber pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]
exp_ratio = 11.9    # Expansion ratio [units: dimensionless]

# Solve for the exit pressure [units: pascal].
p_e = p_c * nozzle.pressure_from_er(exp_ratio, gamma)

print 'Exit pressure = {:.0f} kPa'.format(p_e * 1e-3)
```

```
Exit pressure = 100 kPa
```

Let us plot the effect of expansion ratio on thrust coefficient:

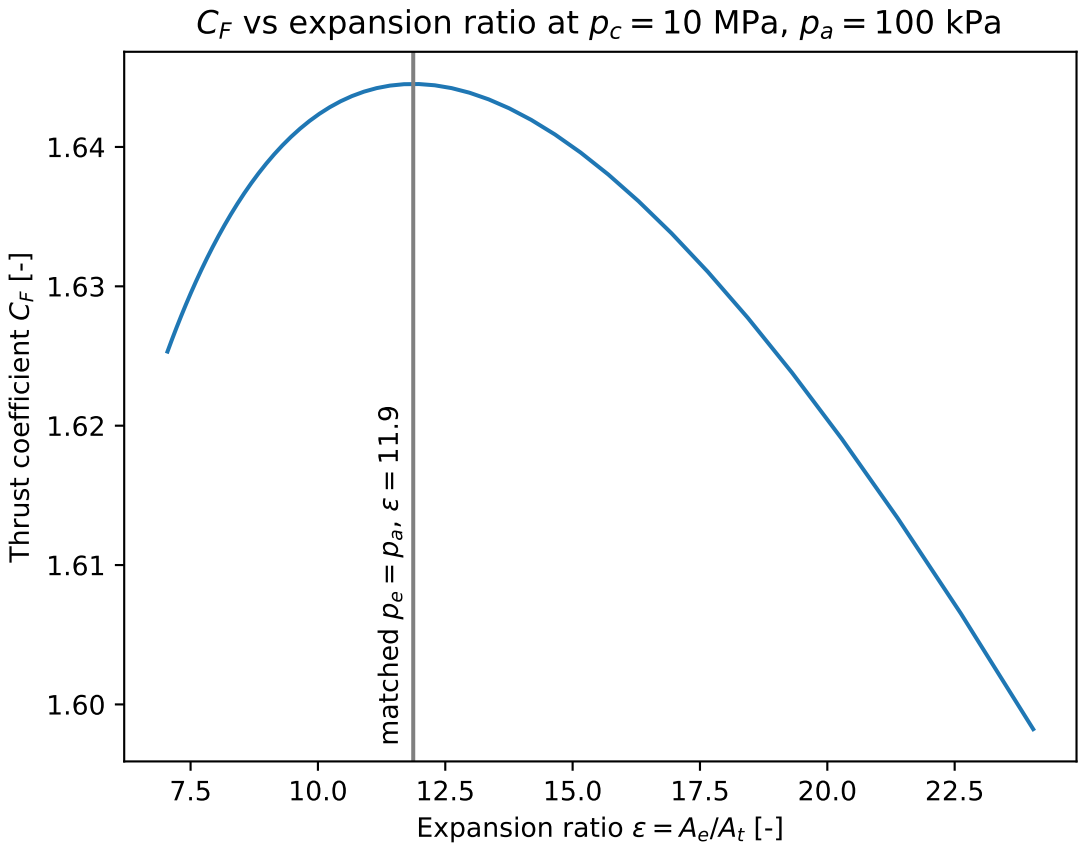
```
"""Effect of expansion ratio on thrust coefficient."""
import numpy as np
from matplotlib import pyplot as plt
from proptools import nozzle

p_c = 10e6    # Chamber pressure [units: pascal]
p_a = 100e3    # Ambient pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]
p_e = np.linspace(0.4 * p_a, 2 * p_a)    # Exit pressure [units: pascal]

# Compute the expansion ratio and thrust coefficient for each p_e
exp_ratio = nozzle.er_from_p(p_c, p_e, gamma)
C_F = nozzle.thrust_coef(p_c, p_e, gamma, p_a=p_a, er=exp_ratio)

# Compute the matched (p_e = p_a) expansion ratio
exp_ratio_matched = nozzle.er_from_p(p_c, p_a, gamma)

plt.plot(exp_ratio, C_F)
plt.axvline(x=exp_ratio_matched, color='grey')
plt.annotate('matched $p_e = p_a$, $\epsilon = {:.1f}$'.format(exp_ratio_matched),
             xy=(exp_ratio_matched - 0.7, 1.62),
             xytext=(exp_ratio_matched - 0.7, 1.62),
             color='black',
             fontsize=10,
             rotation=90
             )
plt.xlabel('Expansion ratio $\\epsilon = A_e / A_t$ [-]')
plt.ylabel('Thrust coefficient $C_F$ [-]')
plt.title('$C_F$ vs expansion ratio at $p_c = {:.0f}$ MPa, $p_a = {:.0f}$ kPa'.format(
    p_c *1e-6, p_a * 1e-3))
plt.show()
```

The thrust coefficient is maximized at the *matched expansion condition*, where $p_e = p_a$. Therefore, nozzle designers select the expansion ratio based on the ambient pressure which the engine is expected to operate in. Small expansion ratios are used for space launch boosters or tactical missiles, which operate at low altitudes (high ambient pressure). Large expansion ratios are used for second stage or orbital maneuvering engines, which operate in the vacuum of space.

The following plot shows C_F vs altitude for our example engine with two different nozzles: a small nozzle suited to a first stage application (blue curve) and a large nozzle for a second stage (orange curve). Compare these curves to the performance of a hypothetical matched nozzle, which expands to $p_e = p_a$ at every altitude. The fixed-expansion nozzles perform well at their design altitude, but have lower C_F than a matched nozzle at all other altitudes.

```
"""Plot C_F vs altitude."""
import numpy as np
from matplotlib import pyplot as plt
import skaero.atmosphere.coesa as atmo
from proptools import nozzle

p_c = 10e6   # Chamber pressure [units: pascal]
gamma = 1.2    # Exhaust heat capacity ratio [units: dimensionless]
p_e_1 = 100e3    # Nozzle exit pressure, 1st stage [units: pascal]
exp_ratio_1 = nozzle.er_from_p(p_c, p_e_1, gamma)    # Nozzle expansion ratio [units:␣
↪dimensionless]
p_e_2 = 15e3    # Nozzle exit pressure, 2nd stage [units: pascal]
exp_ratio_2 = nozzle.er_from_p(p_c, p_e_2, gamma)    # Nozzle expansion ratio [units:␣
↪dimensionless]
```

(continues on next page)


C_F vs expansion ratio at $p_c = 10$ MPa, $p_a = 100$ kPa
Thrust coefficient C_F [-]
1.64
1.63
1.62
1.61
1.60
matched $p_e = p_a$, $\varepsilon = 11.9$
7.5
10.0
12.5
15.0
17.5
20.0
22.5
Expansion ratio $\varepsilon = A_e/A_t$ [-]

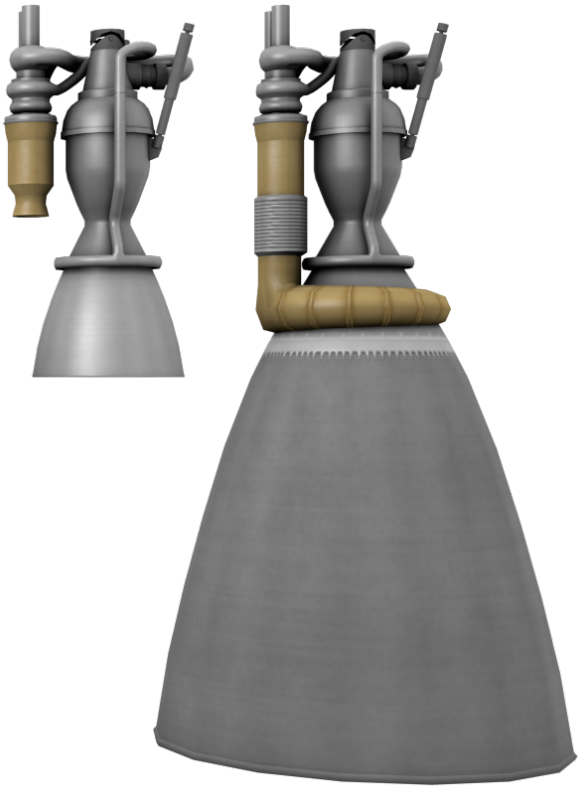


Fig. 2: This illustration shows two variants of an engine family, one designed for a first stage booster (left) and the other for a second stage (right). The first stage (e.g. sea level) engine has a smaller expansion ratio than the second stage (e.g. vacuum) engine. Image credit: shadowmage.

(continued from previous page)

```
alt = np.linspace(0, 84e3)     # Altitude [units: meter]
p_a = atmo.pressure(alt)     # Ambient pressure [units: pascal]

# Compute the thrust coeffieicient of the fixed-area nozzle, 1st stage [units:␣
↪dimensionless]
C_F_fixed_1 = nozzle.thrust_coef(p_c, p_e_1, gamma, p_a=p_a, er=exp_ratio_1)

# Compute the thrust coeffieicient of the fixed-area nozzle, 2nd stage [units:␣
↪dimensionless]
C_F_fixed_2 = nozzle.thrust_coef(p_c, p_e_2, gamma, p_a=p_a, er=exp_ratio_2)

# Compute the thrust coeffieicient of a variable-area matched nozzle [units:␣
↪dimensionless]
C_F_matched = nozzle.thrust_coef(p_c, p_a, gamma)

plt.plot(alt * 1e-3, C_F_fixed_1, label='1st stage $\\epsilon_1 = {:.1f}$'.format(exp_
↪ratio_1))
plt.plot(alt[0.4 * p_a < p_e_2] * 1e-3, C_F_fixed_2[0.4 * p_a < p_e_2],
    label='2nd stage $\\epsilon_2 = {:.1f}$'.format(exp_ratio_2))
plt.plot(alt * 1e-3, C_F_matched, label='matched', color='grey', linestyle=':')
plt.xlabel('Altitude [km]')
plt.ylabel('Thrust coefficient $C_F$ [-]')
plt.title('Effect of altitude on nozzle performance')
plt.legend()
plt.show()
```

1.1.8 Characteristic velocity

We can define another performance parameter which captures the effects of the combustion gas which is supplied to the nozzle. This is the *characteristic velocity*, c^*:

$$c^* \equiv \frac{A_t p_c}{\dot{m}}$$

For an ideal rocket, the characteristic velocity is:

$$c^* = \frac{\sqrt{\gamma R T_c}}{\gamma} \left(\frac{\gamma + 1}{2} \right)^{\frac{\gamma+1}{2(\gamma-1)}}$$

The characteristic velocity depends only on the exhaust properties (γ, R) and the combustion temperature. It is therefore a figure of merit for the combustion process and propellants. c^* is independent of the nozzle expansion process.

The ideal c^* of the example engine is:

```
"""Ideal characteristic velocity."""
from proptools import nozzle

gamma = 1.2     # Exhaust heat capacity ratio [units: dimensionless]
m_molar = 20e-3     # Exhaust molar mass [units: kilogram mole**1]
T_c = 3000.     # Chamber temperature [units: kelvin]

# Compute the characteristic velocity [units: meter second**-1]
c_star = nozzle.c_star(gamma, m_molar, T_c)

print 'Ideal characteristic velocity = {:.0f} m s**-1'.format(c_star)
```

```
Ideal characteristic velocity = 1722 m s**-1
```

1.1.9 Specific Impulse

Finally, we arrive at *specific impulse*, the most important performance parameter of a rocket engine. The specific impulse is the ratio of thrust to the rate of propellant consumption:

$$I_{sp} \equiv \frac{F}{\dot{m} g_0}$$

For historical reasons, specific impulse is normalized by the constant $g_0 = 9.807$ m s $^{-2}$, and has units of seconds. For an ideal rocket at matched exit pressure, $I_{sp} = v_2/g_0$.

The specific impulse measures the "fuel efficiency" of a rocket engine. The specific impulse and propellant mass fraction together determine the delta-v capability of a rocket.

Specific impulse is the product of the thrust coefficient and the characteristic velocity. The overall efficiency of the engine (I_{sp}) depends on both the combustion gas (c^*) and the efficiency of the nozzle expansion process (C_F).

$$I_{sp} = \frac{c^* C_F}{g_0}$$

1.2 Solid-Propellant Rocket Motors

Solid propellant rocket motors store propellant as a solid grain within the combustion chamber. When the motor is ignited, the surfaces of the propellant grain burn and produce hot gas, which is expelled from the chamber through a nozzle to produce thrust.

However, in most solid rocket motors, no mechanism exists to control the chamber pressure and thrust during flight. Rather, the chamber pressure of a solid rocket motor arises from an equilibrium between exhaust generation from combustion and exhaust discharge through the nozzle.

The rest of this page reviews the fundamentals solid propellants, and demonstrates the use of `proptools.solid` to predict the performance of a solid rocket motor.

1.2.1 Applications of Solid Rocket Motors

Compared to liquid-propellant rocket engines, solid propellant motors are mechanically simpler, require less support equipment and time to prepare for launch, and can be stored for long times loaded and ready for launch. Therefore, solid motors are preferred for most military applications, which may need to be fired from mobile launchers (e.g. tactical missiles) or be quickly ready for launch after many years of storage (e.g. strategic missiles). The mechanical simplicity of solid motors is also favors their use in some space-launch applications.

1.2.2 Propellant Ingredients

Chemical requirements of solid propellants

A solid propellant contains both fuel and oxidizer mixed together. This is different from most other combustion systems, where the fuel and oxidizer are only mixed just before combustion (e.g. internal combustion engines, torches, liquid bi-propellant rocket engines). This poses a chemistry challenge: the propellant ingredients must react energetically with each other, but also be safely stored and handled while mixed together. Clearly, a formulation which

spontaneously ignites during mixing has no practical value as a storable solid propellant. A propellant must also not ignite when exposed to mechanical shock, heat or electrostatic discharges during handling. A propellant which is resistant to these accidental ignition sources is said to have low sensitivity. In chemical terms, low sensitivity roughly requires that the combustion reaction have high activation energy.

A further difference between solid rocket motors and most other combustion devices is that the motor contains all its propellant in the combustion chamber, rather than gradually injecting it as it is to be burned. This means that the rate of propellant consumption is not governed by a throttle or injector, but by the chemical dynamics of the combustion reaction. The propellant must burn at a stable and predictable rate. Ingredient sets which react very quickly may be useful as explosives, but not as propellants.

In summary, the choice of ingredients must produce a solid propellant which:

1. stores a large amount of chemical potential energy, and reacts to provide hot gas for propulsion
2. is resistant to accidental ignition during production, storage and handling
3. burns at a stable and predictable rate

Ammonium perchlorate composite propellant

Ammonium perchlorate composite propellant (APCP) is the most-used solid propellant composition in space launch applications (e.g. the Space Shuttle's Reusable Solid Rocket Motor, Orbital ATK's Star motor series). APCP is energetic (up to ~270 seconds of specific impulse), is resistant to accidental ignition, and will burn stably in a properly designed motor.

APCP contains a solid oxidizer (ammonium perchlorate) and (optionally) a powdered metal fuel, held together by a rubber-like binder. Ammonium perchlorate is a crystalline solid, which divided into small particles (10 to 500 μm) and dispersed though the propellant. During combustion, the ammonium perchlorate decomposes to produce a gas rich in oxidizing species. A polymer matrix, the binder, binds the oxidizer particles together, giving the propellant mechanical strength. The binder serves as a fuel, giving off hydrocarbon vapors during combustion. Additional fuel may be added as hot-burning metal powder dispersed in the binder.

1.2.3 Combustion Process

The combustion process of a composite propellant has many steps, and the flame structure is complex. Although the propellant is a solid, important reactions, including combustion of the fuel with the oxidizer, occur in the gas phase. A set of flames hover over the surface of the burning propellant. These flames transfer heat to the propellant surface, causing its solid components to decompose into gases. The gaseous decomposition products contain fuel vapor and oxidizing species, which supply the flames with reactants.

Importantly, the combustion process contains a feedback loop. Heat from the flames vaporizes the surface, and vapor from the surface provides fuel and oxidizer to the flames. The rate at which this process proceeds depends on chemical kinetics, mass transfer, and heat transfer within the combustion zone. Importantly, the feedback rate depends on pressure. As we will see in the next section, the rate of propellant combustion determines the chamber pressure and thrust of a solid rocket motor.

Effect of pressure on burn rate

The flame structure described above causes the propellant to burn faster at higher pressures. At higher pressures, the gas phase is denser, causing reactions and diffusion to proceed more quickly. This moves the flame structure closer to the surface. The closer flames and denser conducing medium enhance heat transfer to the surface, which drives more decomposition, increasing the burn rate.

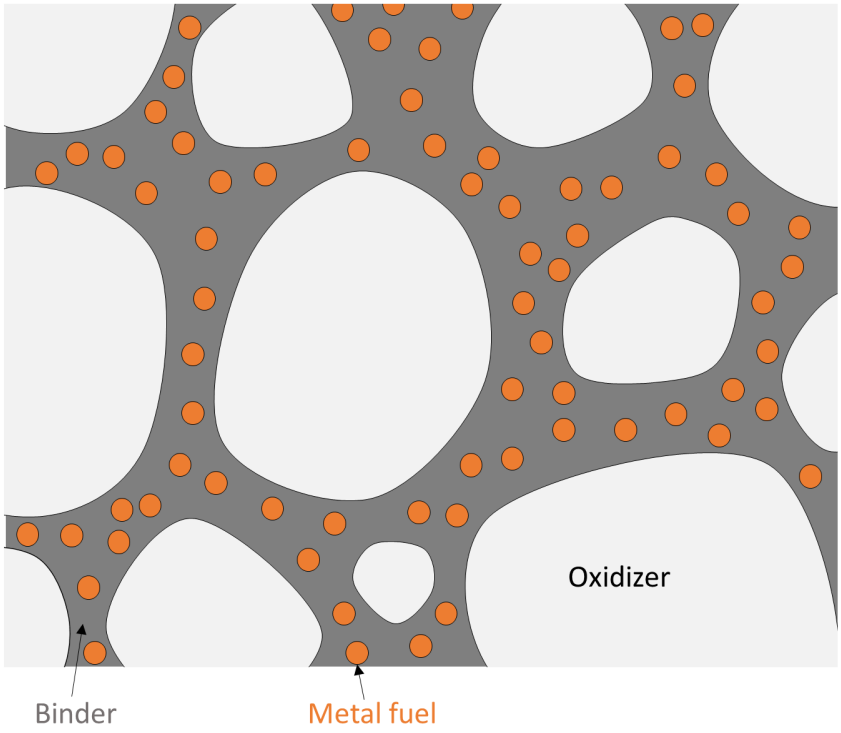


Fig. 3: A composite propellant consists of crystalline oxidizer particles, and possibly a metal fuel powder, dispersed in a polymer binder.

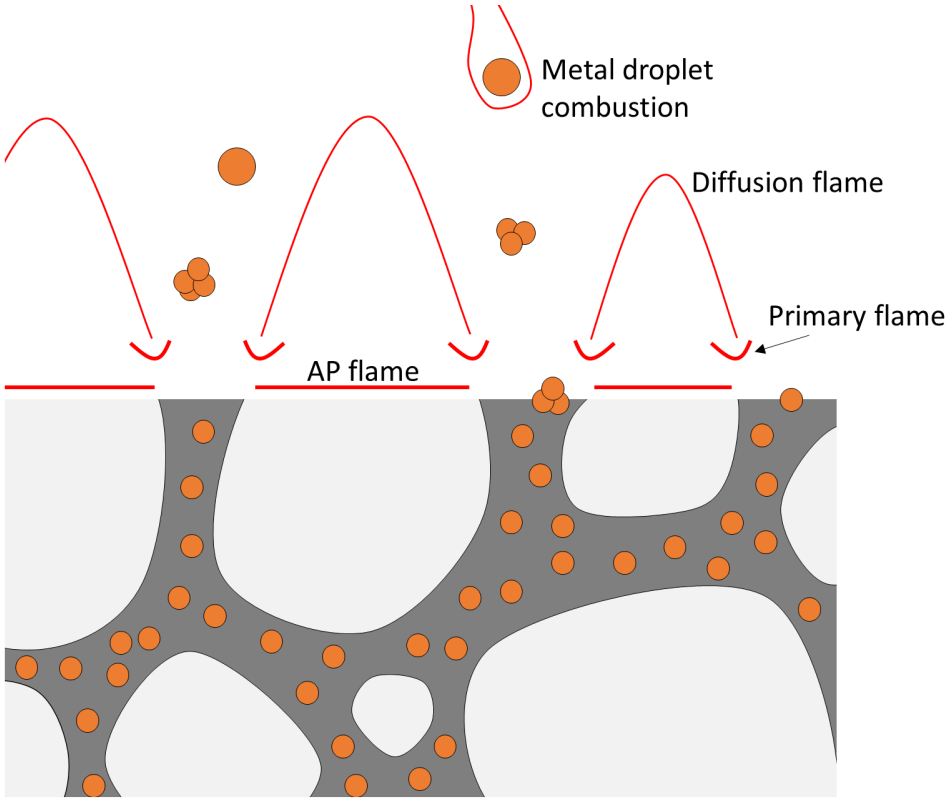


Fig. 4: The typical flame structure of composite propellant combustion. Heat from the flames decomposes the ammonium perchlorate and binder, which in turn supply oxidizing (AP) and fuel (binder) gases to the flames.

Although this dependence is complicated, it can be empirically described by Vieille's Law, which relates the burn rate r to the chamber pressure p_c via two parameters:

$$r = a(p_c)^n$$

r is the rate at which the surface regresses, and has units of velocity. a is the burn rate coefficient, which has units of [velocity (pressure) $^{-n}$]. n is the unitless burn rate exponent. The model parameters a, n must be determined by combustion experiments on the propellant.

1.2.4 Motor Internal Ballistics

The study of propellant combustion and fluid dynamics within a rocket motor is called internal ballistics. Internal ballistics can be used to estimate the chamber pressure and thrust of a motor.

Equilibrium chamber pressure

The operating chamber pressure of a solid motor is set by an equilibrium between exhaust generation from combustion and exhaust discharge through the nozzle. The chamber pressure of a solid rocket motor is related to the mass of combustion gas in the chamber by the Ideal Gas Law:

$$p_c = mRT_c\frac{1}{V_c}$$

where R is the specific gas constant of the combustion gases in the chamber, T_c is their temperature, and V_c is the chamber volume. Gas mass is added to the chamber by burning propellant, and mass flows out of the chamber through the nozzle. The rate of change of the chamber gas mass is:

$$\frac{dm}{dt} = \dot{m}_{combustion} - \dot{m}_{nozzle}$$

Ideal nozzle theory relates the mass flow through the nozzle to the chamber pressure and the *Characteristic velocity*:

$$\dot{m}_{nozzle} = \frac{p_c A_t}{c^*}$$

The rate of gas addition from combustion is:

$$\dot{m}_{combustion} = A_b \rho_s r(p_c)$$

where :math:'A_b' is the burn area of the propellant, ρ_s is the density of the solid propellant, and $r(p_c)$ is the burn rate (a function of chamber pressure).

At the equilibrium chamber pressure where the inflow and outflow rates are equal:

$$\frac{dm}{dt} = \dot{m}_{combustion} - \dot{m}_{nozzle} = 0$$

$$A_b \rho_s r(p_c) = \frac{p_c A_t}{c^*}$$

$$\begin{aligned} p_c &= \frac{A_b}{A_t}\rho_s c^* r(p_c) \\ &= K \rho_s c^* r(p_c) \end{aligned}$$

where $K \equiv \frac{A_b}{A_t}$ is ratio of burn area to throat area. If the propellant burn rate is well-modeled by Vieille's Law, the equilibrium chamber pressure can be solved for in closed form:

$$p_c = (K \rho_s c^* a)^{\frac{1}{1-n}}$$

Consider an example motor. The motor burns a relatively slow-burning propellant with the following properties:

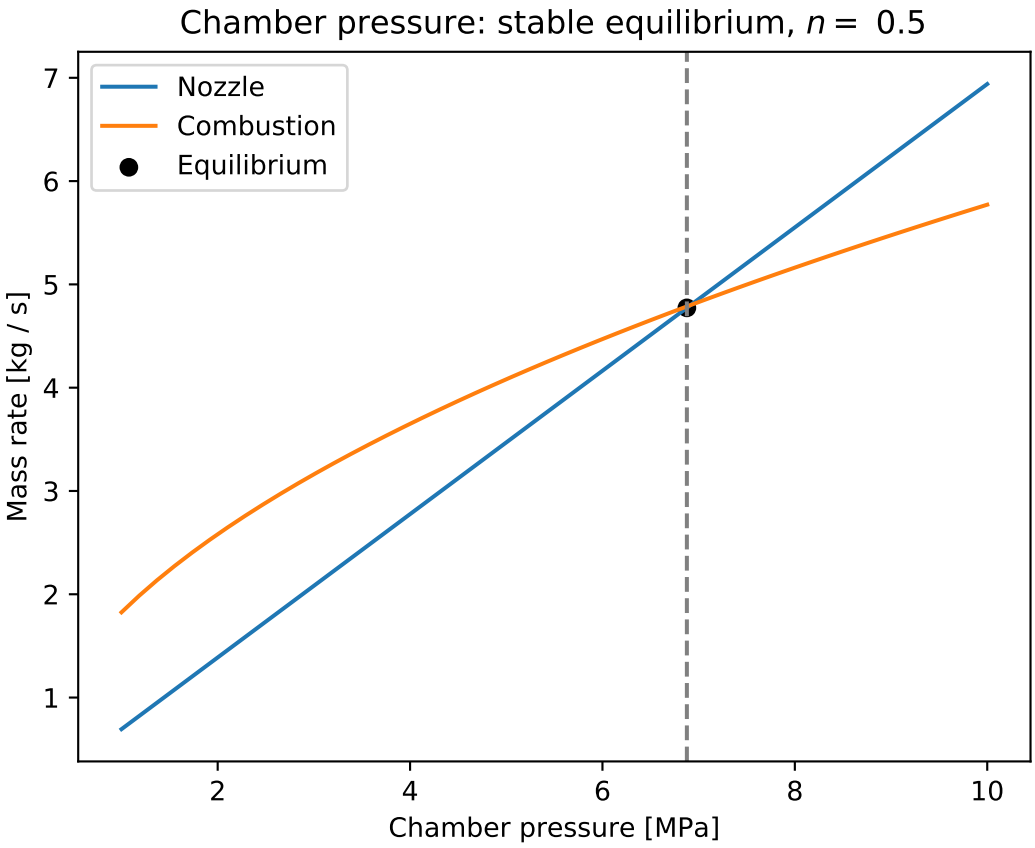
- Burn rate exponent of 0.5, and a burn rate of 2.54 mm s^{-1} at 6.9 MPa
- Exhaust ratio of specific heats of 1.26
- Characteristic velocity of 1209 m s^{-1}
- Solid density of 1510 kg m^{-3}

The motor has a burn area of 1.25 m^2, and a throat area of 839 mm^2 (diameter of 33 mm).

Plot the combustion and nozzle mass flow rates versus pressure:

```
"""Illustrate the chamber pressure equilibrium of a solid rocket motor."""

from matplotlib import pyplot as plt
import numpy as np

p_c = np.linspace(1e6, 10e6)    # Chamber pressure [units: pascal].

# Propellant properties
gamma = 1.26    # Exhaust gas ratio of specific heats [units: dimensionless].
rho_solid = 1510.    # Solid propellant density [units: kilogram meter**-3].
n = 0.5    # Propellant burn rate exponent [units: dimensionless].
a = 2.54e-3 * (6.9e6)**(-n)    # Burn rate coefficient, such that the propellant
# burns at 2.54 mm s**-1 at 6.9 MPa [units: meter second**-1 pascal**-n].
c_star = 1209.    # Characteristic velocity [units: meter second**-1].

# Motor geometry
A_t = 839e-6    # Throat area [units: meter**2].
A_b = 1.25    # Burn area [units: meter**2].

# Compute the nozzle mass flow rate at each chamber pressure.
# [units: kilogram second**-1].
m_dot_nozzle = p_c * A_t / c_star

# Compute the combustion mass addition rate at each chamber pressure.
# [units: kilogram second**-1].
m_dot_combustion = A_b * rho_solid * a * p_c**n

# Plot the mass rates
plt.plot(p_c * 1e-6, m_dot_nozzle, label='Nozzle')
plt.plot(p_c * 1e-6, m_dot_combustion, label='Combustion')
plt.xlabel('Chamber pressure [MPa]')
plt.ylabel('Mass rate [kg / s]')

# Find where the mass rates are equal (e.g. the equilibrium).
i_equil = np.argmin(abs(m_dot_combustion - m_dot_nozzle))
m_dot_equil = m_dot_nozzle[i_equil]
p_c_equil = p_c[i_equil]

# Plot the equilibrium point.
plt.scatter(p_c_equil * 1e-6, m_dot_equil, marker='o', color='black', label=
↪'Equilibrium')
plt.axvline(x=p_c_equil * 1e-6, color='grey', linestyle='--')

plt.title('Chamber pressure: stable equilibrium, $n =$ {:.1f}'.format(n))
plt.legend()
plt.show()
```

The nozzle and combustion mass flow rates are equal at 6.9 MPa: this is the equilibrium pressure of the motor. This


Chamber pressure: stable equilibrium, $n = 0.5$
Nozzle
Combustion
Equilibrium
Mass rate [kg / s]
Chamber pressure [MPa]
1
2
3
4
5
6
7
2
4
6
8
10

equilibrium is stable:

- At lower pressures, the combustion mass addition rate is higher than the nozzle outflow rate, so the mass of gas in the chamber will increase and the pressure will rise to the equilibrium value.
- At higher pressures, the combustion mass addition rate is lower than the nozzle outflow rate, so the mass of gas in the chamber will decrease and the pressure will fall to the equilibrium value.

In general, a stable equilibrium pressure will exist for propellants with $n < 1$ (i.e. burn rate is sub-linear in pressure).

We can use proptools to quickly find the chamber pressure and thrust of the example motor:

```
"""Find the chamber pressure and thrust of a solid rocket motor."""
from proptools import solid, nozzle

# Propellant properties
gamma = 1.26    # Exhaust gas ratio of specific heats [units: dimensionless].
rho_solid = 1510.    # Solid propellant density [units: kilogram meter**-3].
n = 0.5    # Propellant burn rate exponent [units: dimensionless].
a = 2.54e-3 * (6.9e6)**(-n)    # Burn rate coefficient, such that the propellant
# burns at 2.54 mm s**-1 at 6.9 MPa [units: meter second**-1 pascal**-n].
c_star = 1209.    # Characteristic velocity [units: meter second**-1].

# Motor geometry
A_t = 839e-6    # Throat area [units: meter**2].
A_b = 1.25    # Burn area [units: meter**2].

# Nozzle exit pressure [units: pascal].
p_e = 101e3

# Compute the chamber pressure [units: pascal].
p_c = solid.chamber_pressure(A_b / A_t, a, n, rho_solid, c_star)

# Compute the sea level thrust [units: newton].
F = nozzle.thrust(A_t, p_c, p_e, gamma)

print 'Chamber pressure = {:.1f} MPa'.format(p_c * 1e-6)
print 'Thrust (sea level) = {:.1f} kN'.format(F * 1e-3)
```

```
Chamber pressure = 6.9 MPa

Thrust (sea level) = 9.1 kN
```

Burn area evolution and thrust curves

In most propellant grain geometries, the burn area of the propellant grain changes as the flame front advances and propellant is consumed. This change in burn area causes the chamber pressure and thrust to change during the burn. The variation of thrust (or chamber pressure) with time is called a thrust curve. Thrust curves are classified as regressive (decreasing with time), neutral or progressive (increasing with time).

If we know how the burn area A_b varies with the flame front progress distance x, we can use `proptools` to predict the thrust curve. For example, consider a cylindrical propellant with a hollow circular core. The core radius r_{in} is 0.15 m, the outer radius r_{out} is 0.20 m, and the length L is 1.0 m. The burn area is given by:

$$A_b(x) = 2\pi(r_{in} + x)L$$

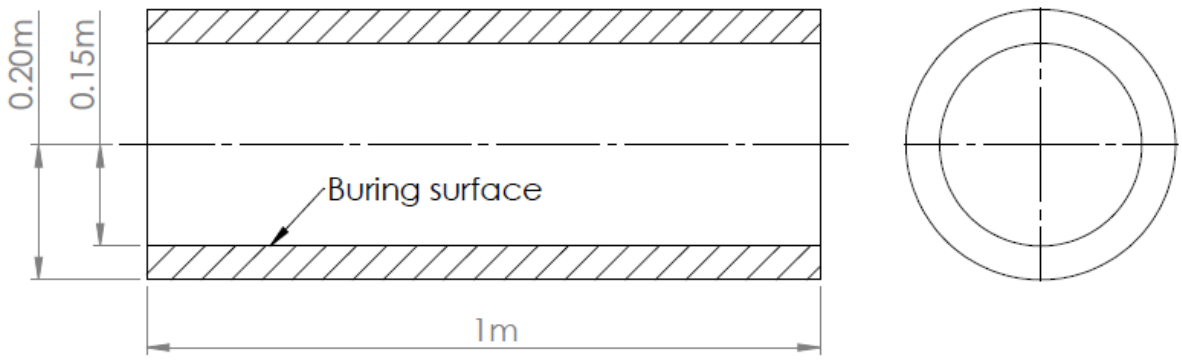


Fig. 5: Dimensions of the example cylindrical propellant grain.

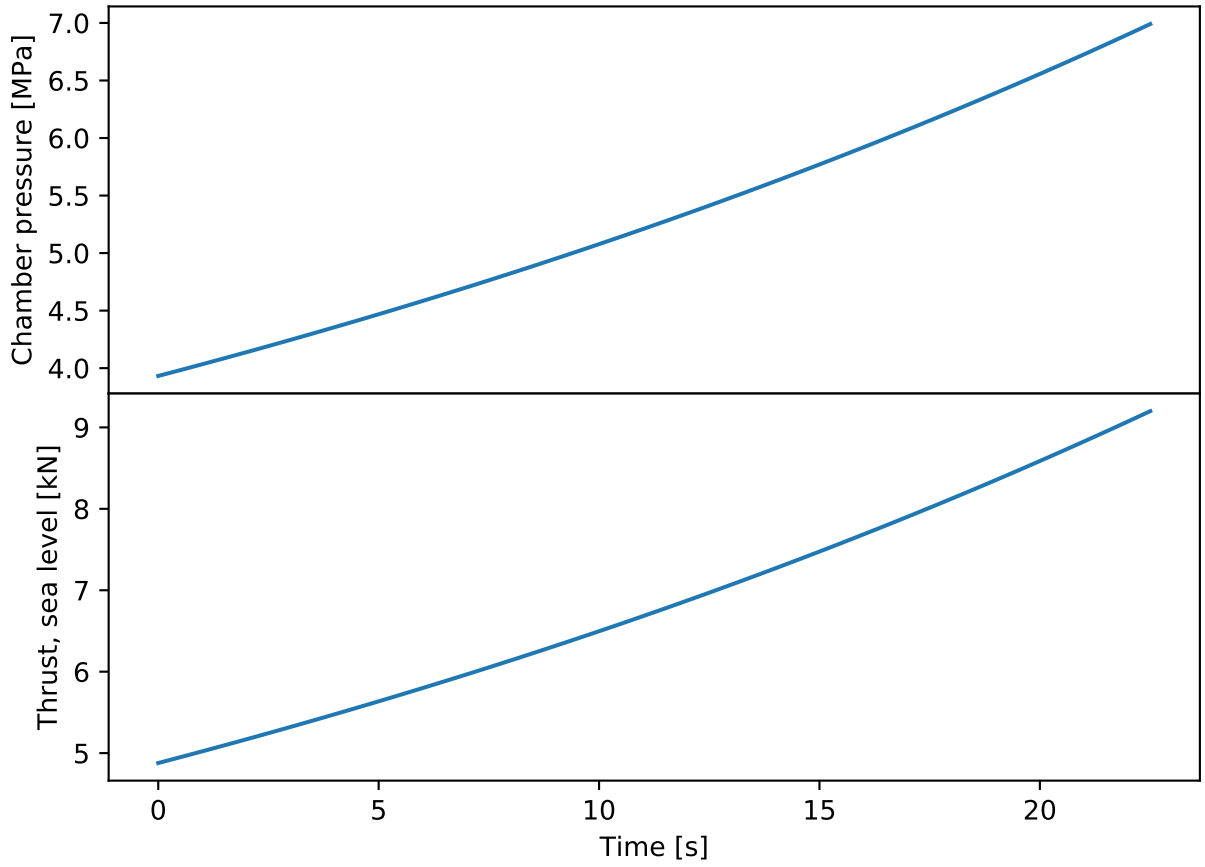
Assume that the propellant properties are the same as in the previous example. The nozzle throat area is still 839 mm ^{2}, and the nozzle expansion area ratio is 8.

```
"""Plot the thrust curve of a solid rocket motor with a cylindrical propellant grain."
↪""

from matplotlib import pyplot as plt
import numpy as np
from proptools import solid


# Grain geometry (Clinder with circular port)
r_in = 0.15    # Grain inner radius [units: meter].
r_out = 0.20    # Grain outer radius [units: meter].
length = 1.0    # Grain length [units: meter].

# Propellant properties
gamma = 1.26    # Exhaust gas ratio of specific heats [units: dimensionless].
rho_solid = 1510.    # Solid propellant density [units: kilogram meter**-3].
n = 0.5    # Propellant burn rate exponent [units: dimensionless].
a = 2.54e-3 * (6.9e6)**(-n)    # Burn rate coefficient, such that the propellant
# burns at 2.54 mm s**-1 at 6.9 MPa [units: meter second**-1 pascal**-n].
c_star = 1209.    # Characteristic velocity [units: meter second**-1].

# Nozzle geometry
A_t = 839e-6    # Throat area [units: meter**2].
A_e = 8 * A_t    # Exit area [units: meter**2].
p_a = 101e3    # Ambeint pressure during motor firing [units: pascal].

# Burning surface evolution
x = np.linspace(0, r_out - r_in)    # Flame front progress steps [units: meter].
A_b = 2 * np.pi * (r_in + x) * length    # Burn area at each flame progress step␣
↪[units: meter**2].

# Compute thrust curve.
t, p_c, F = solid.thrust_curve(A_b, x, A_t, A_e, p_a, a, n, rho_solid, c_star, gamma)

# Plot results.
ax1 = plt.subplot(2, 1, 1)
```

(continues on next page)

(continued from previous page)

```
plt.plot(t, p_c * 1e-6)
plt.ylabel('Chamber pressure [MPa]')

ax2 = plt.subplot(2, 1, 2)
plt.plot(t, F * 1e-3)
plt.ylabel('Thrust, sea level [kN]')
plt.xlabel('Time [s]')
plt.setp(ax1.get_xticklabels(), visible=False)

plt.tight_layout()
plt.subplots_adjust(hspace=0)
plt.show()
```



Note that the pressure and thrust increase with time (the thrust curve is progressive). This grain has a progressive thrust curve because the burn area increases with x as the flame front moves outward from the initial core.

Solid motor designers have devised a wide variety of grain geometries to achieve different thrust curves.

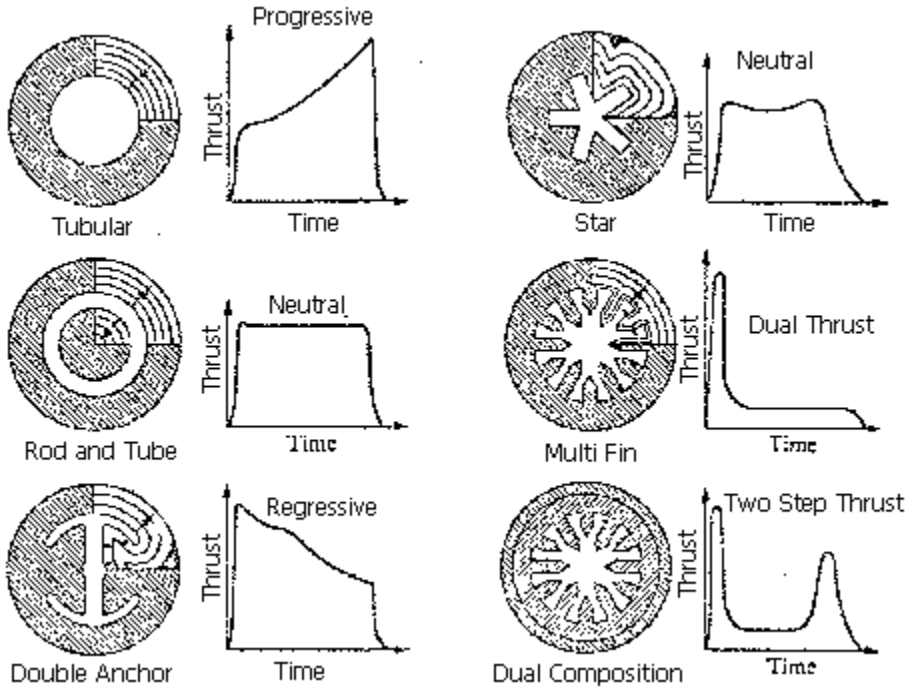


Fig. 6: Various grains and their thrust curves. Reprinted from Richard Nakka's rocketry page.

1.3 Electric Propulsion Basics

1.3.1 Electrostatic Thrust Generation

Like all rockets, electric thrusters generate thrust by expelling mass at high velocity. In electric (electrostatic) propulsion, a high-velocity jet is produced by accelerating charged particles through an electric potential.

The propellant medium in electrostatic thrusters is a plasma consisting of electrons and positively charged ions. The plasma is usually produced by ionizing a gas via electron bombardment. The electric fields within the thruster are configured such that there is a large potential drop between the plasma generation region and the thruster exit. When ions exit the thruster, the are accelerated to high velocities by the electrostatic force. If the ions fall through a beam potential of V_b, they will leave the thruster with a velocity

$$v_e = \sqrt{2V_b \frac{q}{m_i}}$$

where q is the ion charge and m_i is the ion mass. The thrust produced by the ion flow is

$$\begin{aligned} F &= v_e \dot{m} \\ &= \left(\sqrt{2V_b \frac{q}{m_i}}\right)\left(\frac{m_i}{q} I_b\right) \\ &= I_b \sqrt{2V_b \frac{m_i}{q}} \end{aligned}$$

where I_b is the ion beam current.

The ideal specific impulse of an electrostatic thruster is:

$$I_{sp} = \frac{v_e}{g_0} = \frac{1}{g_0}\sqrt{2V_b\frac{q}{m_i}}$$

As an example, use `proptools` to compute the thrust and specific impulse of singly charged Xenon ions with a beam voltage of 1000 V and a beam current of 1 A:

```
"""Example electric propulsion thrust and specific impulse calculation."""
from proptools import electric, constants

V_b = 1000.    # Beam voltage [units: volt].
I_b = 1.    # Beam current [units: ampere].
F = electric.thrust(I_b, V_b, electric.m_Xe)
I_sp = electric.specific_impulse(V_b, electric.m_Xe)
m_dot = F / (I_sp * constants.g)
print 'Thrust = {:.1f} mN'.format(F * 1e3)
print 'Specific Impulse = {:.0f} s'.format(I_sp)
print 'Mass flow = {:.2f} mg s^-1'.format(m_dot * 1e6)
```

```
Thrust = 52.2 mN

Specific Impulse = 3336 s

Mass flow = 1.59 mg s^-1
```

This example illustrates the typical performance of electric propulsion systems: low thrust, high specific impulse, and low mass flow.

1.3.2 Advantages over Chemical Propulsion

Electric propulsion is appealing because it enables higher specific impulse than chemical propulsion. The specific impulse of a chemical rocket depends on velocity to which a nozzle flow can be accelerated. This velocity is limited by the finite energy content of the working gas. In contrast, the particles leaving an electric thruster can be accelerated to very high velocities if sufficient electrical power is available.

In a chemical rocket, the kinetic energy of the exhaust gas is supplied by thermal energy released in a combustion reaction. For example, in the stoichiometric combustion of H_2 and O_2 releases an energy per particle of 4.01×10^{-19} J, or 2.5 eV. If all of the released energy were converted into kinetic energy of the exhaust jet, the maximum possible exhaust velocity would be:

$$v_e = \sqrt{2\frac{E}{m_{H_2O}}} = 5175\,\mathrm{ms}^{-1}$$

corresponding to a maximum specific impulse of about 530 s. In practice, the most efficient flight engines have specific impulses of 450 to 500 s.

With electric propulsion, much higher energies per particle are possible. If we accelerate (singly) charged particles through a potential of 1000 V, the jet kinetic energy will be 1000 eV per ion. This is 400 times more energy per particle than is possible with chemical propulsion (or, for similar particle masses, 20 times higher specific impulse). Specific impulse in excess of 10,000 s is feasible with electric propulsion.

1.3.3 Power and Efficiency

In an electric thruster, the energy to accelerate the particles in the jet must be supplied by an external source. Typically solar panels supply this electrical power, but some future concepts might use nuclear reactors. The available power limits the thrust and specific impulse of an electric thruster.

The kinetic power of the jet is given by:

$$P_{jet} = \frac{F^2}{2\dot{m}}$$

where $\dot{m}$ is the jet mass flow. Increasing thrust with increasing mass flow (i.e. increasing I_{sp}) will increase the kinetic power of the jet.

The power input required by the thruster (P_{in}) is somewhat higher than the jet power. The ratio of the jet and input power is the total efficiency of the thruster, η_T:

$$\eta_T \equiv \frac{P_{jet}}{P_{in}}$$

The total efficiency depends on several factors:

1. Thrust losses due to beam divergence. This loss is proportional to the cosine of the average beam divergence half-angle.
2. Thrust losses due to doubly charged ions in the beam. This loss is a function of the doubles-to-singles current ratio, I^{++}/I^{+}
3. Thrust losses due to propellant gas escaping the thruster without being ionized. The fraction of the propellant mass flow which is ionized and enters the beam is the mass utilization efficiency, η_m.
4. Electrical losses incurred in ion generation, power conversion, and powering auxiliary thruster components. These losses are captured by the electrical efficiency, $\eta_e = \frac{I_b V_b}{P_{in}}$

Use `proptools` to compute the efficiency and required power of the example thruster. Assume that the beam divergence is $\cos(10xb0)$, the double ion current fraction is 10%, the mass utilization efficiency is 90%, and the electrical efficiency is 85%:

```
"""Example electric propulsion power calculation."""
import numpy as np
from proptools import electric

F = 52.2e-3    # Thrust [units: newton].
m_dot = 1.59e-6    # Mass flow [units: kilogram second**-1].

# Compute the jet power [units: watt].
P_jet = electric.jet_power(F, m_dot)

# Compute the total efficiency [units: dimensionless].
eta_T = electric.total_efficiency(
    divergence_correction=np.cos(np.deg2rad(10)),
    double_fraction=0.1,
    mass_utilization=0.9,
    electrical_efficiency=0.85)

# Compute the input power [units: watt].
P_in = P_jet / eta_T

print 'Jet power = {:.0f} W'.format(P_jet)
print 'Total efficiency = {:.3f}'.format(eta_T)
print 'Input power = {:.0f} W'.format(P_in)
```

```
Jet power = 857 W

Total efficiency = 0.703

Input power = 1219 W
```

The overall efficiency of the thruster is about 70%. The required input power could be supplied by a few square meters of solar panels (at 1 AU from the sun).

The power, thrust, and specific impulse of a thruster are related by:

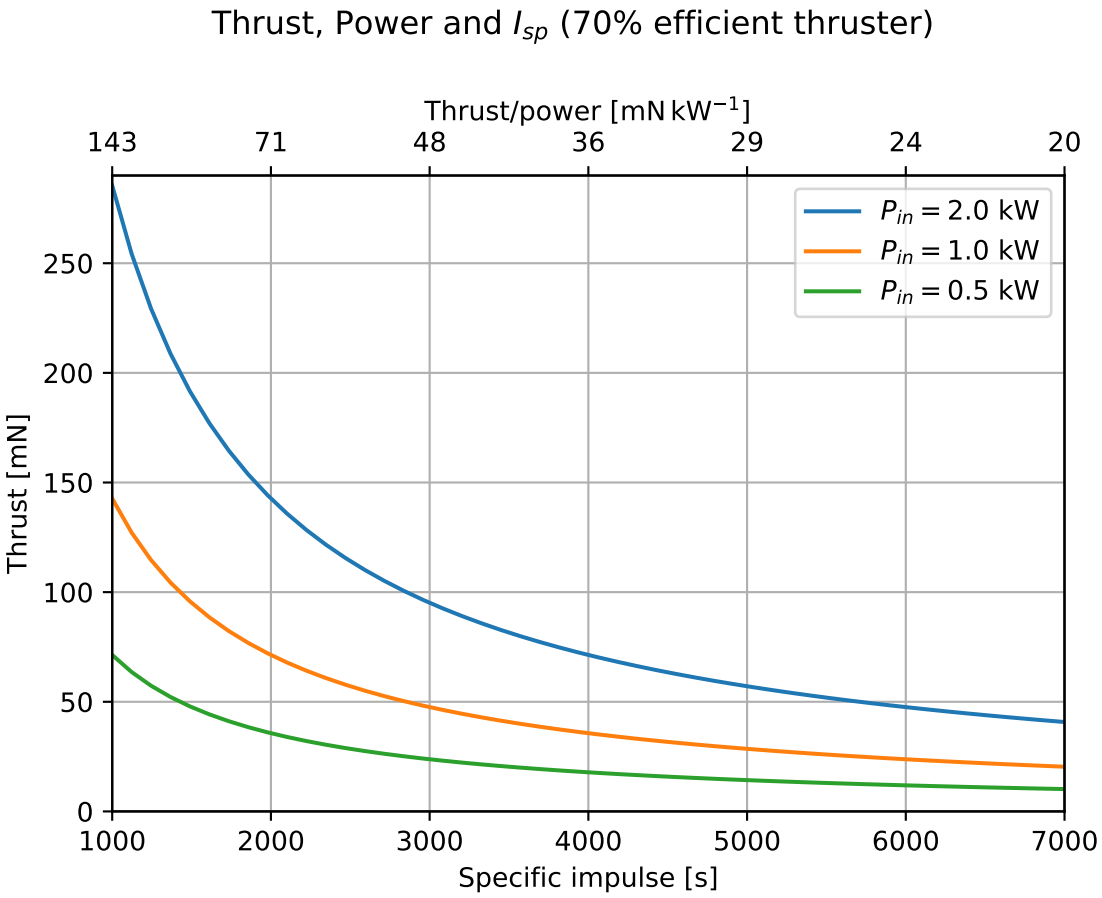
$$\frac{F}{P_{in}} = \frac{2\eta_T}{g_0 I_{sp}}$$

Thus, for a power-constrained system the propulsion designer faces a trade-off between thrust and specific impulse.

```
"""Plot the thrust, power, Isp trade-off."""
from matplotlib import pyplot as plt
import numpy as np
from proptools import electric

eta_T = 0.7    # Total efficiency [units: dimensionless].
I_sp = np.linspace(1e3, 7e3)    # Specific impulse [units: second].

ax1 = plt.subplot(111)
for P_in in [2e3, 1e3, 500]:
    T = P_in * electric.thrust_per_power(I_sp, eta_T)
    plt.plot(I_sp, T * 1e3, label='$P_{{in}} = {:.1f}$ kW'.format(P_in * 1e-3))

plt.xlim([1e3, 7e3])
plt.ylim([0, 290])
plt.xlabel('Specific impulse [s]')
plt.ylabel('Thrust [mN]')
plt.legend()
plt.suptitle('Thrust, Power and $I_{sp}$ (70% efficient thruster)')
plt.grid(True)

# Add thrust/power on second axis.
ax2 = ax1.twiny()
new_tick_locations = ax1.get_xticks()
ax2.set_xlim(ax1.get_xlim())
ax2.set_xticks(new_tick_locations)
ax2.set_xticklabels(['{:.0f}'.format(tp * 1e6)
                     for tp in electric.thrust_per_power(new_tick_locations, eta_T)])
ax2.set_xlabel('Thrust/power [$\\mathrm{mN \\, kW^{-1}}$]')
ax2.tick_params(axis='y', direction='in', pad=-25)
plt.subplots_adjust(top=0.8)

plt.show()
```

1.3.4 Optimal Specific Impulse

For electrically propelled spacecraft, there is an optimal specific impulse which will maximize the payload mass fraction of a given mission. While increasing specific impulse decreases the required propellant mass, it also increases the required power at a particular thrust level, which increases the mass of the power supply. The optimal specific impulse minimizes the combined mass of the propellant and power supply.


Thrust, Power and I_{sp} (70% efficient thruster)
Thrust/power [mN kW^{-1}]
143
71
48
36
29
24
20
P_{in} = 2.0 kW
P_{in} = 1.0 kW
P_{in} = 0.5 kW
Thrust [mN]
0
50
100
150
200
250
1000
2000
3000
4000
5000
6000
7000
Specific impulse [s]

The optimal specific impulse depends on several factors:

1. The mission thrust duration, t_m. Longer thrust durations reduce the required thrust (if the same total impulse or Δv is to be delivered), and therefore reduce the power and power supply mass at a given I_{sp}. Therefore, longer thrust durations increase the optimal I_{sp}.
2. The specific mass of the power supply, α. This is the ratio of power supply mass to power, and is typically 20 to 200 kg kW $^{-1}$ for solar-electric systems. The specific impulse optimization assumes that power supply mass is linear with respect to power. Increasing the specific mass reduces the optimal I_{sp}.
3. The total efficiency of the thruster.
4. The Δv of the mission. Higher Δv (in a fixed time window) requires more thrust, and therefore leads to a lower optimal I_{sp}.

Consider an example mission to circularize the orbit of a geostationary satellite launched onto a elliptical transfer orbit. Assume that the low-thrust circularization maneuver requires a Δv of 2 km s $^{-1}$ over 100 days. The thruster is 70% efficient and the power supply specific mass is 50 kg kW $^{-1}$:

```
"""Example calculation of optimal specific impulse for an electrically propelled␣
↪spacecraft."""
from proptools import electric

dv = 2e3    # Delta-v [units: meter second**-1].
t_m = 100 * 24 * 60 * 60    # Thrust duration [units: second].
eta_T = 0.7    # Total efficiency [units: dimensionless].
specific_mass = 50e-3    # Specific mass of poower supply [units: kilogram watt**-1].
I_sp = electric.optimal_isp_delta_v(dv, eta_T, t_m, specific_mass)

print 'Optimal specific impulse = {:.0f} s'.format(I_sp)
```

```
Optimal specific impulse = 1482 s
```

For the mathematical details of specific impulse optimization, see [Lozano].

CHAPTER 2

Package contents

2.1 proptools

2.1.1 proptools package

Submodules

proptools.constants module

Physical Constants.

Matt Vernacchia proptools 2016 Apr 15

proptools.isentropic module

Isentropic relations.

See *Isentropic Relations* for a physical explaination of the isentropic relations.

stag_temperature_ratio(M, gamma)	Stagnation temperature / static temperature ratio.
stag_pressure_ratio(M, gamma)	Stagnation pressure / static pressure ratio.
stag_density_ratio(M, gamma)	Stagnation density / static density ratio.
velocity(v_1, p_1, T_1, p_2, gamma, m_molar)	Velocity relation between two points in an isentropic flow.

proptools.isentropic.**stag_density_ratio**(*M*, *gamma*)
Stagnation density / static density ratio.

Parameters

- **M** (*scalar*) – Mach number [units: dimensionless].

- **gamma** (*scalar*) – Heat capacity ratio [units: dimensionless].

Returns the stagnation density ratio ρ_0/ρ [units: dimensionless].

Return type scalar

proptools.isentropic.**stag_pressure_ratio**(*M*, *gamma*)
Stagnation pressure / static pressure ratio.

Parameters

- **M** (*scalar*) – Mach number [units: dimensionless].
- **gamma** (*scalar*) – Heat capacity ratio [units: dimensionless].

Returns the stagnation pressure ratio p_0/p [units: dimensionless].

Return type scalar

proptools.isentropic.**stag_temperature_ratio**(*M*, *gamma*)
Stagnation temperature / static temperature ratio.

Parameters

- **M** (*scalar*) – Mach number [units: dimensionless].
- **gamma** (*scalar*) – Heat capacity ratio [units: dimensionless].

Returns the stagnation temperature ratio T_0/T [units: dimensionless].

Return type scalar

proptools.isentropic.**velocity**(*v_1*, *p_1*, *T_1*, *p_2*, *gamma*, *m_molar*)
Velocity relation between two points in an isentropic flow.

Given the velocity, pressure, and temperature at station 1 and the pressure at station 2, find the velocity at station 2. See Rocket Propulsion Elements, 8th edition, equation 3-15b.

Parameters

- **v_1** (*scalar*) – Velocity at station 1 [units: meter second**-1].
- **p_1** (*scalar*) – Pressure at station 1 [units: pascal].
- **T_1** (*scalar*) – Temperature at station 1 [units kelvin].
- **p_2** (*scalar*) – Pressure at station 2 [units: pascal].
- **gamma** (*scalar*) – Gas ratio of specific heats [units: dimensionless].
- **m_molar** (*scalar*) – Gas mean molar mass [units: kilogram mole**-1].

Returns velocity at station 2 [units: meter second**-1].

Return type scalar

proptools.electric package

Electric propulsion design tools.

m_Xe	float(x) -> floating point number
m_Kr	float(x) -> floating point number
thrust(I_b, V_b, m_ion)	Thrust of an electric thruster.
jet_power(F, m_dot)	Jet power of a rocket propulsion device.

Continued on next page

Table 2 – continued from previous page

double_ion_thrust_correction(double_fraction)	Doubly-charged ion thrust correction factor.
specific_impulse(V_b, m_ion[, ...])	Specific impulse of an electric thruster.
total_efficiency([divergence_correction, ...])	Total efficiency of an electric thruster.
thrust_per_power(I_sp[, total_efficiency])	Thrust/power ratio of an electric thruster.
stuhlinger_velocity(total_efficiency, t_m, ...)	Stuhlinger velocity, a characteristic velocity for electric propulsion missions.
optimal_isp_thrust_time(total_efficiency, ...)	Optimal specific impulse for constant thrust / fixed time mission.
optimal_isp_delta_v(dv, total_efficiency, ...)	Optimal specific impulse for a fixed Δv mission.

proptools.electric.**double_ion_thrust_correction**(*double_fraction*)

Doubly-charged ion thrust correction factor.

Compute the thrust correction factor for the presence of doubly charged ions in the beam. This factor is denoted as α in Goebel and Katz.

Reference: Goebel and Katz, equation 2.3-14.

Parameters **double_fraction** (*scalar in [0, 1]*) – The doubly-charged ion current over the singly-charged ion current, I^{++}/I^{+} [units: dimensionless].

Returns The thrust correction factor, α [units: dimensionless].

Return type scalar in (0, 1]

proptools.electric.**jet_power**(*F*, *m_dot*)

Jet power of a rocket propulsion device.

Compute the kinetic power of the jet for a given thrust level and mass flow.

Reference: Goebel and Katz, equation 2.3-4.

Parameters

- **F** (*scalar*) – Thrust force [units: newton].
- **m_dot** (*scalar*) – Jet mass flow rate [units: kilogram second**-1].

Returns jet power [units: watt].

Return type scalar

proptools.electric.**optimal_isp_delta_v**(*dv*, *total_efficiency*, *t_m*, *specific_mass*, *discharge_loss=None*, *m_ion=None*)

Optimal specific impulse for a fixed Δv mission.

The dependence of efficiency on specific impulse can optionally be included in the optimization. If `discharge_loss` and `m_ion` are not provided, then the total efficiency η_T has a constant value of the given `total_efficiency`. If `discharge_loss` and `m_ion` are provided, η_T is assumed to vary with specific impulse as:

$\eta_T = \frac{\eta_{T,0}}{1+(v_L/I_{sp}g)^2}$

where $\eta_{T,0}$ is the given value of `total_efficiency`, and v_L is the loss velocity:

$v_L = \sqrt{2qV_d/m_i}$

where V_d is the discharge loss and m_i is the ion mass.

Reference: Lozano 16.522 notes, equation 3-13.

Parameters

- **total_efficiency** (*scalar in (0, 1]*) – The total efficiency of the thruster [units: dimensionless].
- **t_m** – mission thrust duration [units: second].
- **specific_mass** – specific mass (per unit power) of the thruster and power system [units: kilogram watt**1].
- **discharge_loss** (*scalar, optional*) – The discharge loss per ion [units: volt].
- **m_ion** (*scalar, optional*) – Ion mass [units: kilogram].

Returns Optimal specific impulse [units: second].

Return type scalar

proptools.electric.**optimal_isp_thrust_time**(*total_efficiency*, *t_m*, *specific_mass*)

Optimal specific impulse for constant thrust / fixed time mission.

Reference: Lozano 16.522 notes, equation 3-5.

Parameters

- **total_efficiency** (*scalar in (0, 1]*) – The total efficiency of the thruster [units: dimensionless].
- **t_m** – mission thrust duration [units: second].
- **specific_mass** (*scalar*) – specific mass (per unit power) of the thruster and power system [units: kilogram watt**1].

Returns Optimal specific impulse [units: second].

Return type scalar

proptools.electric.**specific_impulse**(*V_b*, *m_ion*, *divergence_correction=1*, *double_fraction=1*, *mass_utilization=1*)

Specific impulse of an electric thruster.

If only `V_b` and `m_ion` are provided, the ideal specific impulse will be computed. If `divergence_correction`, `double_fraction`, or `mass_utilization` are provided, the specific impulse will be reduced by the corresponding efficiency factors.

Reference: Goebel and Katz, equation 2.4-8.

Parameters

- **V_b** (*scalar*) – Beam voltage [units: volt].
- **m_ion** (*scalar*) – Ion mass [units: kilogram].
- **divergence_correction** (*scalar in (0, 1])*) – Thrust correction factor for beam divergence [units: dimensionless].
- **double_fraction** (*scalar in [0, 1]*) – The doubly-charged ion current over the singly-charged ion current, I^{++}/I^{+} [units: dimensionless].
- **mass_utilization** (*scalar in (0, 1])*) – Mass utilization efficiency [units: dimensionless].

Returns the specific impulse [units: second].

Return type scalar

proptools.electric.**stuhlinger_velocity**(*total_efficiency*, *t_m*, *specific_mass*)

Stuhlinger velocity, a characteristic velocity for electric propulsion missions.

The Stuhlinger velocity is a characteristic velocity for electric propulsion missions, and is used in calculation of optimal specific impulse. It is defined as:

$v_{ch} \equiv \sqrt{\frac{2\eta_T t_m}{\alpha}}$

where η_T is the thruster total efficiency, t_m is the mission thrust duration, and α is the propulsion and power system specific mass.

The Stuhlinger velocity is also an approximate upper limit for the Δv capability of an electric propulsion spacecraft.

Reference: Lozano 16.522 notes, equation 3-7.

Parameters

- **total_efficiency** (*scalar in (0, 1]*) – The total efficiency of the thruster [units: dimensionless].
- **t_m** – mission thrust duration [units: second].
- **specific_mass** – specific mass (per unit power) of the thruster and power system [units: kilogram watt**1].

Returns Stuhlinger velocity [units: meter second**-1].

Return type scalar

proptools.electric.**thrust**(*I_b*, *V_b*, *m_ion*)

Thrust of an electric thruster.

Compute the ideal thrust of an electric thruster from the beam current and voltage, assuming singly charged ions and no beam divergence.

Reference: Goebel and Katz, equation 2.3-8.

Parameters

- **I_b** (*scalar*) – Beam current [units: ampere].
- **V_b** (*scalar*) – Beam voltage [units: volt].
- **m_ion** (*scalar*) – Ion mass [units: kilogram].

Returns Thrust force [units: newton].

Return type scalar

proptools.electric.**thrust_per_power**(*I_sp*, *total_efficiency=1*)

Thrust/power ratio of an electric thruster.

Reference: Goebel and Katz, equation 2.5-9.

Parameters

- **I_sp** (*scalar*) – Specific impulse [units:second].
- **total_efficiency** (*scalar in (0, 1]*) – The total efficiency of the thruster [units: dimensionless].

Returns Thrust force per unit power input [units: newton watt**-1].

Return type scalar

proptools.electric.**total_efficiency**(*divergence_correction=1*, *double_fraction=1*, *mass_utilization=1*, *electrical_efficiency=1*)

Total efficiency of an electric thruster.

The total efficiency is defined as the ratio of jet power to input power:

$\eta_T \equiv \frac{P_{jet}}{P_{in}}$

Reference: Goebel and Katz, equation 2.5-7.

Parameters

- **divergence_correction** (*scalar in (0, 1])*) – Thrust correction factor for beam divergence [units: dimensionless].
- **double_fraction** (*scalar in [0, 1]*) – The doubly-charged ion current over the singly-charged ion current, I^{++}/I^{+} [units: dimensionless].
- **mass_utilization** (*scalar in (0, 1])*) – Mass utilization efficiency [units: dimensionless].
- **electrical_efficiency** (*scalar in (0, 1])*) – Electrical efficiency [units: dimensionless].

Returns Total efficiency [units: dimensionless].

Return type scalar

proptools.isentropic_test module

Unit test for isentropic relations.

class proptools.isentropic_test.**TestStagDesnityRatio**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for isentropic.stag_density_ratio

test_sonic()
Check the ratio when Mach=1.

test_still()
Check that the ratio is 1 when Mach=0.

class proptools.isentropic_test.**TestStagPressureRatio**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for isentropic.stag_pressure_ratio

test_sonic()
Check the ratio when Mach=1.

test_still()
Check that the ratio is 1 when Mach=0.

class proptools.isentropic_test.**TestStagTemperatureRatio**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for isentropic.stag_temperature_ratio

test_sonic()
Check the ratio when Mach=1.

test_still()
Check that the ratio is 1 when Mach=0.

class proptools.isentropic_test.**TestVelocity**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for isentropic.velocity.

test_equal_pressure()
Test that the velocity is the same if the pressure is the same.

test_rpe_3_2()
Test against example problem 3-2 from Rocket Propulsion Elements.

proptools.nonsimple_comp_flow module

Non-simple compressible flow.

Calculate quasi-1D compressible flow properties with varying area, friction, and heat addition. "One-dimensional compressible flows of calorically perfect gases in which only a single driving potential is present are called simple flows" [1]. This module implements a numerical solution for non-simple flows, i.e. flows with multiple driving potentials.

References

[1] L. Pekker, "One-Dimensional Compressible Flow in Variable Area Duct with Heat Addition," Air Force Research Laboratory, Edwards, CA, Rep. AFRL-RZ-ED-JA-2010-303, 2010. Online: http://www.dtic.mil/dtic/tr/fulltext/u2/a524450.pdf.

[2] A. Bandyopadhyay and A. Majumdar, "Modeling of Compressible Flow with Friction and Heat Transfer using the Generalized Fluid System Simulation Program (GFSSP)," Thermal Fluid Analysis Workshop, Cleveland, OH, 2007. Online: https://tfaws.nasa.gov/TFAWS07/Proceedings/TFAWS07-1016.pdf

[3] J. D. Anderson, Modern Compressible Flow with Historical Perspective, 2nd ed. New York, NY: McGraw-Hill, 1990.

Matt Vernacchia proptools 2016 Oct 3

proptools.nonsimple_comp_flow.**differential**(*x*, *state*, *mdot*, *c_p*, *gamma*, *f_f*, *f_q*, *f_A*)
Differential equation for Mach number in non-simple duct flow.

Note: This method will not be accurate (and may divide by zero) for flows which contain a region at Mach 1, e.g. a choked convergent-divergent nozzle.

Parameters

- **state** (*2-vector*) – Stagnation temperature [units: kelvin], Mach number [units: none].
- **x** (*scalar*) – Distance from the duct inlet [units: meter].
- **mdot** (*scalar*) – The mass flow through the duct [units: kilogram second**-1].
- **c_p** (*scalar*) – Fluid heat capacity at constant pressure [units: joule kilogram**-1 kelvin**-1].
- **gamma** (*scalar*) – Fluid ratio of specific heats [units: none].
- **f_f** (*function mapping scalar->scalar*) – The Fanning friction factor as a function of distance from the inlet [units: none].
- **f_q** (*function mapping scalar->scalar*) – The heat transfer into the fluid per unit wall area as a function of distance from the inlet [units: joule meter**-2].
- **f_A** (*function mapping scalar->scalar*) – The duct area as a function of distance from the inlet [units: meter**2].

Returns d state / dx

proptools.nonsimple_comp_flow.**main**()

proptools.nonsimple_comp_flow.**solve_nonsimple**(*x*, *M_in*, *T_o_in*, *mdot*, *c_p*, *gamma*, *f_f*, *f_q*, *f_A*)

Solve a non-simple flow case

Parameters

- **state** (*2-vector*) – Stagnation temperature [units: kelvin], Mach number [units: none].
- **x** (*array*) – Distances from the duct inlet at which to return solution [units: meter].
- **T_o_in** (*scalar*) – Inlet stagnation temperature [units: kelvin].
- **M_in** (*scalar*) – Inlet Mach number [units: none].
- **mdot** (*scalar*) – The mass flow through the duct [units: kilogram second**-1].
- **c_p** (*scalar*) – Fluid heat capacity at constant pressure [units: joule kilogram**-1 kelvin**-1].
- **gamma** (*scalar*) – Fluid ratio of specific heats [units: none].
- **f_f** (*function mapping scalar->scalar*) – The Fanning friction factor as a function of distance from the inlet [units: none].
- **f_q** (*function mapping scalar->scalar*) – The heat transfer into the fluid per unit wall area as a function of distance from the inlet [units: joule meter**-2].
- **f_A** (*function mapping scalar->scalar*) – The duct area as a function of distance from the inlet [units: meter**2].

Returns

The stagnation temperature at each station in x [units: none]. M (array of length len(x)): The Mach number at each station in x [units: none]. choked (boolean): True if the flow chokes at M=1 in the duct. M and T_o for x past the

choke point will be nan. Choking can cause shocks or upstream effects which this model does not capture; therefore results for choked scenarios may not be accurate.

Return type T_o (array of length len(x))

proptools.nozzle module

Nozzle flow calculations.

thrust_coef(p_c, p_e, gamma[, p_a, er])	Nozzle thrust coefficient, C_F.
c_star(gamma, m_molar, T_c)	Characteristic velocity, c^*.
er_from_p(p_c, p_e, gamma)	Find the nozzle expansion ratio from the chamber and exit pressures.
throat_area(m_dot, p_c, T_c, gamma, m_molar)	Find the nozzle throat area.
mass_flow(A_t, p_c, T_c, gamma, m_molar)	Find the mass flow through a choked nozzle.
thrust(A_t, p_c, p_e, gamma[, p_a, er])	Nozzle thrust force.
mach_from_er(er, gamma)	Find the exit Mach number from the area expansion ratio.
mach_from_pr(p_c, p_e, gamma)	Find the exit Mach number from the pressure ratio.
is_choked(p_c, p_e, gamma)	Determine whether the nozzle flow is choked.
mach_from_area_subsonic(area_ratio, gamma)	Find the Mach number as a function of area ratio for subsonic flow.
area_from_mach(M, gamma)	Find the area ratio for a given Mach number.

Continued on next page

Table 3 – continued from previous page

pressure_from_er(er, gamma)	Find the exit/chamber pressure ratio from the nozzle expansion ratio.

proptools.nozzle.**area_from_mach**(*M*, *gamma*)

Find the area ratio for a given Mach number.

For isentropic nozzle flow, a station where the Mach number is M will have an area A. This function returns that area, normalized by the area of the nozzle throat A_t. See *Mach-Area Relation* for a physical description of the Mach-Area relation.

Reference: Rocket Propulsion Elements, 8th Edition, Equation 3-14.

Parameters

- **M** (*scalar*) – Mach number [units: dimensionless].
- **gamma** (*scalar*) – Ratio of specific heats [units: dimensionless].

Returns Area ratio A/A_t.

Return type scalar

proptools.nozzle.**c_star**(*gamma*, *m_molar*, *T_c*)

Characteristic velocity, c^*.

The characteristic velocity is a figure of merit for the propellants and combustion process. See *Characteristic velocity* for a description of the physical meaning of the characteristic velocity.

Reference: Equation 1-32a in Huzel and Huang.

Parameters

- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].
- **m_molar** (*scalar*) – Exhaust gas mean molar mass [units: kilogram mole**-1].
- **T_c** (*scalar*) – Nozzle stagnation temperature [units: kelvin].

Returns The characteristic velocity [units: meter second**-1].

Return type scalar

proptools.nozzle.**er_from_p**(*p_c*, *p_e*, *gamma*)

Find the nozzle expansion ratio from the chamber and exit pressures.

See *Expansion Ratio* for a physical description of the expansion ratio.

Reference: Rocket Propulsion Elements, 8th Edition, Equation 3-25

Parameters

- **p_c** (*scalar*) – Nozzle stagnation chamber pressure [units: pascal].
- **p_e** (*scalar*) – Nozzle exit static pressure [units: pascal].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].

Returns Expansion ratio $\epsilon = A_e/A_t$ [units: dimensionless]

Return type scalar

proptools.nozzle.**is_choked**(*p_c*, *p_e*, *gamma*)

Determine whether the nozzle flow is choked.

See *Choked Flow* for details.

Reference: Rocket Propulsion Elements, 8th Edition, Equation 3-20.

Parameters

- **p_c** (*scalar*) – Nozzle stagnation chamber pressure [units: pascal].
- **p_e** (*scalar*) – Nozzle exit static pressure [units: pascal].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].

Returns True if flow is choked, false otherwise.

Return type bool

proptools.nozzle.**mach_from_area_subsonic**(*area_ratio*, *gamma*)
Find the Mach number as a function of area ratio for subsonic flow.

Parameters

- **area_ratio** (*scalar*) – Area / throat area [units: dimensionless].
- **gamma** (*scalar*) – Ratio of specific heats [units: dimensionless].

Returns Mach number of the flow in a passage with `area = area_ratio * (throat area)`.

Return type scalar

proptools.nozzle.**mach_from_er**(*er*, *gamma*)
Find the exit Mach number from the area expansion ratio.

Reference: J. Majdalani and B. A. Maickie, http://maji.utsi.edu/publications/pdf/HT02_11.pdf

Parameters

- **er** (*scalar*) – Nozzle area expansion ratio, A_e / A_t [units: dimensionless].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].

Returns The exit Mach number [units: dimensionless].

Return type scalar

proptools.nozzle.**mach_from_pr**(*p_c*, *p_e*, *gamma*)
Find the exit Mach number from the pressure ratio.

Parameters

- **p_c** (*scalar*) – Nozzle stagnation chamber pressure [units: pascal].
- **p_e** (*scalar*) – Nozzle exit static pressure [units: pascal].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].

Returns Exit Mach number [units: dimensionless].

Return type scalar

proptools.nozzle.**mass_flow**(*A_t*, *p_c*, *T_c*, *gamma*, *m_molar*)
Find the mass flow through a choked nozzle.

Given gas stagnation conditions and a throat area, find the mass flow through a choked nozzle. See *Choked Flow* for details.

Reference: Rocket Propulsion Elements, 8th Edition, Equation 3-24.

Parameters

- **A_t** (*scalar*) – Nozzle throat area [units: meter**2].
- **p_c** (*scalar*) – Nozzle stagnation chamber pressure [units: pascal].

- **T_c** (*scalar*) – Nozzle stagnation temperature [units: kelvin].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].
- **m_molar** (*scalar*) – Exhaust gas mean molar mass [units: kilogram mole**-1].

Returns Mass flow rate $\dot{m}$ [units: kilogram second**-1].

Return type scalar

proptools.nozzle.**pressure_from_er**(*er*, *gamma*)

Find the exit/chamber pressure ratio from the nozzle expansion ratio.

See *Expansion Ratio* for a physical description of the expansion ratio.

Reference: Rocket Propulsion Elements, 8th Edition, Equation 3-25

Parameters

- **er** (*scalar*) – Expansion ratio $\epsilon = A_e/A_t$ [units: dimensionless].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].

Returns Pressure ratio p_e/p_c [units: dimensionless].

Return type scalar

proptools.nozzle.**throat_area**(*m_dot*, *p_c*, *T_c*, *gamma*, *m_molar*)

Find the nozzle throat area.

Given gas stagnation conditions and a mass flow rate, find the required throat area of a choked nozzle. See *Choked Flow* for details.

Reference: Rocket Propulsion Elements, 8th Edition, Equation 3-24

Parameters

- **m_dot** (*scalar*) – Propellant mass flow rate [units: kilogram second**-1].
- **p_c** (*scalar*) – Nozzle stagnation chamber pressure [units: pascal].
- **T_c** (*scalar*) – Nozzle stagnation temperature [units: kelvin].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].
- **m_molar** (*scalar*) – Exhaust gas mean molar mass [units: kilogram mole**-1].

Returns Throat area [units: meter**2].

Return type scalar

proptools.nozzle.**thrust**(*A_t*, *p_c*, *p_e*, *gamma*, *p_a=None*, *er=None*)

Nozzle thrust force.

Parameters

- **A_t** (*scalar*) – Nozzle throat area [units: meter**2].
- **p_c** (*scalar*) – Nozzle stagnation chamber pressure [units: pascal].
- **p_e** (*scalar*) – Nozzle exit static pressure [units: pascal].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].
- **p_a** (*scalar, optional*) – Ambient pressure [units: pascal]. If None, then p_a = p_e.
- **er** (*scalar, optional*) – Nozzle area expansion ratio [units: dimensionless]. If None, then p_a = p_e.

Returns Thrust force [units: newton].

Return type scalar

proptools.nozzle.**thrust_coef**(*p_c*, *p_e*, *gamma*, *p_a=None*, *er=None*)
Nozzle thrust coefficient, C_F.

The thrust coefficient is a figure of merit for the nozzle expansion process. See *Thrust coefficient* for a description of the physical meaning of the thrust coefficient.

Reference: Equation 1-33a in Huzel and Huang.

Parameters

- **p_c** (*scalar*) – Nozzle stagnation chamber pressure [units: pascal].
- **p_e** (*scalar*) – Nozzle exit static pressure [units: pascal].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].
- **p_a** (*scalar, optional*) – Ambient pressure [units: pascal]. If None, then p_a = p_e.
- **er** (*scalar, optional*) – Nozzle area expansion ratio [units: dimensionless]. If None, then p_a = p_e.

Returns The thrust coefficient, C_F [units: dimensionless].

Return type scalar

proptools.nozzle_test module

Unit tests for nozzle flow.

class proptools.nozzle_test.**PressureFromEr**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.pressure_from_er and nozzle.er_from_p

test_inverse()
Test that pressure_from_er is the inverse of area_from_mach.

test_rpe_3_3()
Test against example problem 3-3 from Rocket Propulsion Elements.

class proptools.nozzle_test.**TestCStar**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.c_star.

test_hh_1_3()
Test against example problem 1-3 from Huzel and Huang.

class proptools.nozzle_test.**TestMachArea**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.area_from_mach.

test_gamma_sonic()
Test that the sonic area ratio is 1, across range of gamma.

test_inverse()
Test that mach_from_area_subsonic is the inverse of area_from_mach.

class proptools.nozzle_test.**TestMachFormEr**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.mach_from_er.

test_hh_1_12()
Test mach_from_er against H&H figure 1-12

class proptools.nozzle_test.**TestMachFromPr**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.mach_from_pr.

test_rpe_3_1()
Test mach_from_pr against RPE figure 3-1.

class proptools.nozzle_test.**TestMassFlow**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.mass_flow.

test_rpe_3_3()
Test against example problem 3-3 from Rocket Propulsion Elements.

class proptools.nozzle_test.**TestThrust**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.thrust.

test_rpe_3_3()
Test against example problem 3-3 from Rocket Propulsion Elements.

class proptools.nozzle_test.**TestThrustCoef**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for nozzle.thrust_coef.

test_hh_1_3()
Test against example problem 1-3 from Huzel and Huang.

proptools.solid module

Solid rocket motor equations.

chamber_pressure(K, a, n, rho_solid, c_star)	Chamber pressure due to solid propellant combustion.
burn_area_ratio(p_c, a, n, rho_solid, c_star)	Get the burn area ratio, given chamber pressure and propellant properties.
burn_and_throat_area(F, p_c, p_e, a, n, ...)	Given thrust and chamber pressure, and propellant properties, find the burn area and throat area.
thrust_curve(A_b, x, A_t, A_e, p_a, a, n, ...)	Thrust vs time curve for a solid rocket motor.

proptools.solid.**burn_and_throat_area**(*F*, *p_c*, *p_e*, *a*, *n*, *rho_solid*, *c_star*, *gamma*)
Given thrust and chamber pressure, and propellant properties, find the burn area and throat area.

Assumes that the exit pressure is matched ($p_e = p_a$).

Parameters

- **F** (*scalar*) – Thrust force [units: newton].
- **p_c** (*scalar*) – Chamber pressure [units: pascal].
- **p_e** (*scalar*) – Nozzle exit pressure [units: pascal].
- **a** (*scalar*) – Propellant burn rate coefficient [units: meter second**-1 pascal**-n].
- **n** (*scalar*) – Propellant burn rate exponent [units: none].

- **rho_solid** (*scalar*) – Solid propellant density [units: kilogram meter**-3].
- **c_star** (*scalar*) – Propellant combustion characteristic velocity [units: meter second**-1].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].

Returns

tuple containing:

A_b (scalar): Burn area [units: meter**2].

A_t (scalar): Throat area [units: meter**2].

Return type tuple

proptools.solid.**burn_area_ratio**(*p_c*, *a*, *n*, *rho_solid*, *c_star*)

Get the burn area ratio, given chamber pressure and propellant properties.

Reference: Equation 12-6 in Rocket Propulsion Elements, 8th edition.

Parameters

- **p_c** (*scalar*) – Chamber pressure [units: pascal].
- **a** (*scalar*) – Propellant burn rate coefficient [units: meter second**-1 pascal**-n].
- **n** (*scalar*) – Propellant burn rate exponent [units: none].
- **rho_solid** (*scalar*) – Solid propellant density [units: kilogram meter**-3].
- **c_star** (*scalar*) – Propellant combustion characteristic velocity [units: meter second**-1].

Returns Ratio of burning area to throat area, $K = A_b/A_t$ [units: dimensionless].

Return type scalar

proptools.solid.**chamber_pressure**(*K*, *a*, *n*, *rho_solid*, *c_star*)

Chamber pressure due to solid propellant combustion.

Reference: Equation 12-6 in Rocket Propulsion Elements, 8th edition.

Parameters

- **K** (*scalar*) – Ratio of burning area to throat area, A_b/A_t [units: dimensionless].
- **a** (*scalar*) – Propellant burn rate coefficient [units: meter second**-1 pascal**-n].
- **n** (*scalar*) – Propellant burn rate exponent [units: dimensionless].
- **rho_solid** (*scalar*) – Solid propellant density [units: kilogram meter**-3].
- **c_star** (*scalar*) – Propellant combustion characteristic velocity [units: meter second**-1].

Returns Chamber pressure [units: pascal].

Return type scalar

proptools.solid.**thrust_curve**(*A_b*, *x*, *A_t*, *A_e*, *p_a*, *a*, *n*, *rho_solid*, *c_star*, *gamma*)

Thrust vs time curve for a solid rocket motor.

Given information about the evolution of the burning surface of the propellant grain, this function predicts the time-varying thrust of a solid rocket motor.

The evolution of the burning surface is described by two lists, `A_b` and `x`. Each element in the lists describes a step in the (discretized) evolution of the burning surface. `x[i]` is the distance which the flame front must progress (normal to the burning surface) to reach step `i`. `A_b[i]` is the burn area at step `i`.

Parameters

- **A_b** (*list*) – Burn area at each step [units: meter**-2].
- **x** (*list*) – flame front progress distance at each step [units: meter].
- **A_t** (*scalar*) – Nozzle throat area [units: meter**2].
- **A_e** (*scalar*) – Nozzle exit area [units: meter**2].
- **p_a** (*scalar*) – Ambient pressure during motor firing [units: pascal].
- **a** (*scalar*) – Propellant burn rate coefficient [units: meter second**-1 pascal**-n].
- **n** (*scalar*) – Propellant burn rate exponent [units: none].
- **rho_solid** (*scalar*) – Solid propellant density [units: kilogram meter**-3].
- **c_star** (*scalar*) – Propellant combustion characteristic velocity [units: meter second**-1].
- **gamma** (*scalar*) – Exhaust gas ratio of specific heats [units: dimensionless].

Returns

tuple containing:

t (list): time at each step [units: second].

p_c (list): Chamber pressure at each step [units: pascal].

F (list): Thrust at each step [units: newton].

Return type tuple

proptools.tank_structure module

Tank structure calculations.

Matt Vernacchia proptools 2015 April 17

`proptools.tank_structure.`**cr_ex_press_cylinder**(*a*, *t_c*, *l_c*, *E*, *v*)

Critical external pressure difference to buckle a cylindrical tank section.

Implements eqn 8-12 in Huzel and Huang.

Parameters

- **a** – Tank radius [units: meter].
- **t_c** – Tank wall thickness [units: meter].
- **l_c** – Cylinder length [units: meter].
- **E** – Wall material modulus of elasticity [units: pascal].
- **v** – Wall material Poisson's ratio [units: none].

Returns pascal].

Return type Critical external pressure for buckling [units

proptools.tank_structure.**cr_ex_press_ellipse_end**(*a*, *b*, *t*, *E*, *C_b=0.05*)
Critical external pressure difference to buckle a ellipsoidal tank end.

Implements eqn 8-25 in Huzel and Huang.

Parameters

- **a** – Tank radius [units: meter].
- **a** – Semiminor axis [units: meter].
- **t** – Wall thickness [units: meter].
- **E** – Wall material modulus of elasticity [units: pascal].
- **Cb** – Buckling coefficient [units: none].

Returns pascal].

Return type Critical external pressure for buckling [units

proptools.tank_structure.**cr_ex_press_sphere**(*a*, *t*, *E*, *v*)
Critical external pressure difference to buckle a spherical tank.

Implements eqn 8-12 in Huzel and Huang.

Parameters

- **a** – Tank radius [units: meter].
- **t** – Wall thickness [units: meter].
- **E** – Wall material modulus of elasticity [units: pascal].
- **v** – Wall material Poisson's ratio [units: none].

Returns pascal].

Return type Critical external pressure for buckling [units

proptools.tank_structure.**cr_ex_press_sphere_end**(*a*, *t*, *E*)
Critical external pressure difference to buckle a spherical tank end.

Implements eqn 8-26 in Huzel and Huang.

Parameters

- **a** – Tank radius [units: meter].
- **t** – Wall thickness [units: meter].
- **E** – Wall material modulus of elasticity [units: pascal].

Returns pascal].

Return type Critical external pressure for buckling [units

proptools.tank_structure.**crown_thickness**(*p_t*, *R*, *stress*, *weld_eff*)
Crown thickness of a spherical or ellipsoidal tank end.

Implements eqn 8-16 from Huzel and Huang. The crown is the center of the tank end, see figure 8-6 in Huzel and Huang.

Parameters

- **p_t** – Tank internal pressure (less the ambient pressure) [units: pascal].
- **R** – Crown radius [units: meter].
- **stress** – Max allowable stress in thank wall material [units: pascal].

- **weld_eff** – Weld efficiency, scalar in [0, 1] [units: none].

Returns meter].

Return type crown thickness [units

proptools.tank_structure.**cylinder_mass**(*a*, *t_c*, *l_c*, *rho*)
Mass of a cylindrical tank section.

Parameters

- **a** – Tank radius [units: meter].
- **t_c** – Tank wall thickness [units: meter].
- **l_c** – Cylinder length [units: meter].
- **rho** – Tank material density [units: kilogram meter**-3].

Returns mass of spherical tank [kilogram].

proptools.tank_structure.**cylinder_thickness**(*p_t*, *a*, *stress*, *weld_eff*)
Thickness of a cylindrical tank section.

Implements eqn 8-28 from Huzel and Huang.

Parameters

- **p_t** – Tank internal pressure (less the ambient pressure) [units: pascal].
- **a** – Tank radius [units: meter].
- **stress** – Max allowable stress in thank wall material [units: pascal].
- **weld_eff** – Weld efficiency, scalar in [0, 1] [units: none].

Returns meter].

Return type cylinder thickness [units

proptools.tank_structure.**cylinder_volume**(*a*, *l_c*)
Volume enclosed by a cylindrical tank section.

Parameters

- **a** – Tank radius [units: meter].
- **l_c** – Cylinder length [units: meter].

Returns meter**3].

Return type tank volume [units

proptools.tank_structure.**ellipse_design_factor**(*ellipse_ratio*)
Get the ellipse design factor K for an ellipse ratio.

Implements eqn bs-16 in Huzel and Huang.

Parameters **ellipse_ratio** – Ratio of major and minor axes of ellipse end [units: none].

Returns none].

Return type ellipse design factor K for ellipsoidal end stress calculations [units

proptools.tank_structure.**ellipse_mass**(*a*, *b*, *t*, *rho*)
Mass of a ellipsoidal tank.

Parameters

- **a** – Tank radius [units: meter].

- **b** – Tank semimajor axis [units: meter].
- **t** – Tank wall thickness [units: meter].
- **rho** – Tank material density [units: kilogram meter**-3].

Returns mass of ellipsoidal tank [kilogram].

proptools.tank_structure.**ellipse_volume**(*a*, *b*)
Volume enclosed by a ellipsoidal tank.

Parameters

- **a** – Tank radius (semimajor axis) [units: meter].
- **a** – Semiminor axis [units: meter].

Returns meter**3].

Return type tank volume [units

proptools.tank_structure.**knuckle_factor**(*ellipse_ratio*)
Get the knuckle factor K for an ellipse ratio.

Implements the "Envelope Curve for K for Combined Stress" curve from figure 8-7 in Huzel and Huang.

Parameters **ellipse_ratio** – Ratio of major and minor axes of ellipse end [units: none].

Returns none].

Return type knuckle factor K for ellipsoidal end stress calculations [units

proptools.tank_structure.**knuckle_thickness**(*p_t*, *a*, *b*, *stress*, *weld_eff*)
Knuckle thickness of a ellipsoidal tank end.

Implements eqn 8-15 from Huzel and Huang. The knuckle is the transition from the cylindrical section to the tank end, see figure 8-6 in Huzel and Huang.

Parameters

- **p_t** – Tank internal pressure (less the ambient pressure) [units: pascal].
- **a** – Tank radius [units: meter].
- **a** – Semiminor axis [units: meter].
- **stress** – Max allowable stress in thank wall material [units: pascal].
- **weld_eff** – Weld efficiency, scalar in [0, 1] [units: none].

Returns meter].

Return type knuckle thickness [units

proptools.tank_structure.**max_axial_load**(*p_t*, *a*, *t_c*, *l_c*, *E*)
Maximum compressive axial load that a cylindrical section can support.

Implements eqn 8-33 from Huzel and Huang.

Parameters

- **p_t** – Tank internal pressure (less the ambient pressure) [units: pascal].
- **a** – Tank radius [units: meter].
- **t_c** – Cylinder wall thickness [units: meter].
- **l_c** – Cylinder length [units: meter].
- **E** – Wall material modulus of elasticity [units: pascal].

Returns newtons].

Return type Critical compressive axial load [units

proptools.tank_structure.**sphere_mass**(*a*, *t*, *rho*)
Mass of a spherical tank.

Parameters

- **a** – Tank radius [units: meter].
- **t** – Tank wall thickness [units: meter].
- **rho** – Tank material density [units: kilogram meter**-3].

Returns mass of spherical tank [kilogram].

proptools.tank_structure.**sphere_thickness**(*p_t*, *a*, *stress*, *weld_eff*)
Thickness of a spherical tank.

Implements eqn 8-9 from Huzel and Huang.

Parameters

- **p_t** – Tank internal pressure (less the ambient pressure) [units: pascal].
- **a** – Tank radius [units: meter].
- **stress** – Max allowable stress in thank wall material [units: pascal].
- **weld_eff** – Weld efficiency, scalar in [0, 1] [units: none].

Returns meter].

Return type sphere thickness [units

proptools.tank_structure.**sphere_volume**(*a*)
Volume enclosed by a spherical tank.

Parameters **a** – Tank radius [units: meter].

Returns meter**3].

Return type tank volume [units

proptools.tank_structure_test module

class proptools.tank_structure_test.**TestStringMethods**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_sample_8_3()

test_sample_8_4()

proptools.turbopump module

Nozzle flow calculations.

Matt Vernacchia proptools 2016 Apr 3

proptools.turbopump.**dp2head**(*dp*, *rho*)
Convert pump pressure rise to US-units head.

Parameters

- **dp** – Pump pressure rise [units: pascal].
- **rho** – density [units: kilogram meter**-3].

Returns feet].

Return type pump head [units

proptools.turbopump.**gg_dump_isp**(*p_o*, *p_te*, *p_ne*, *T_o*, *eta*, *gamma*, *c_p*, *m_molar*)
Get the specific impulse of a Gas Generator turbine exhaust dump.

Parameters

- **p_o** – turbine inlet stagnation pressure [units: pascal].
- **p_te** – turbine exit pressure [units: pascal].
- **p_ne** – Dump nozzle exit pressure [units: pascal].
- **T_o** – turbine inlet stagnation temperature [units: kelvin].
- **eta** – turbine efficiency.
- **gamma** – working gas ratio of specific heats [units: none].
- **c_p** – working gas heat capacity at const pressure [units: joule kilogram**-1 kelvin**-1].
- **m_molar** – working gas molar mass [units: kilogram mole**-1].

proptools.turbopump.**gpm2m_dot**(*gpm*, *rho*)
Convert gallons per minute to mass flow.

Parameters

- **gpm** – Volume flow [gallon minute**-1].
- **rho** – density [units: kilogram meter**-3].

Returns mass flow [units: kilogram second**-1].

Return type m_dot

proptools.turbopump.**m_dot2gpm**(*m_dot*, *rho*)
Convert mass flow to gallons per minute.

Parameters

- **m_dot** – mass flow [units: kilogram second**-1].
- **rho** – density [units: kilogram meter**-3].

Returns Volume flow [gallon minute**-1].

proptools.turbopump.**pump_efficiency**(*dp*, *m_dot*, *rho*, *N*)
Pump efficiency estimate.

Based on figure 6-23 in Huzel and Huang.

Parameters

- **dp** – Pump pressure rise [units: pascal].
- **m_dot** – Pump mass flow [units: kilogram second**-1].
- **rho** – Density of pumped fluid [units: kilogram meter**-3].
- **N** – Pump rotation speed [radian second**-1].

Returns none].

Return type pump efficiency [units

proptools.turbopump.**pump_efficiency_demo**()

proptools.turbopump.**pump_power**(*dp*, *m_dot*, *rho*, *eta*)
Get the input drive power for a pump.

Parameters

- **dp** – Pump pressure rise [units: pascal].
- **m_dot** – Pump mass flow [units: kilogram second**-1].
- **rho** – Density of pumped fluid [units: kilogram meter**-3].
- **eta** – Pump efficiency [units: none].

Returns watt].

Return type The shaft power required to drive the pump [units

proptools.turbopump.**pump_specific_speed_us**(*dp*, *m_dot*, *rho*, *N*)
Pump specific speed N_s in US units.

Parameters

- **dp** – Pump pressure rise [units: pascal].
- **m_dot** – Pump mass flow [units: kilogram second**-1].
- **rho** – Density of pumped fluid [units: kilogram meter**-3].
- **N** – Pump rotation speed [radian second**-1].

Returns rpm gallon**0.5 minute**-0.5 feet**-0.75)].

Return type N_s [units

proptools.turbopump.**radsec2rpm**(*radsec*)
Convert radian second**-1 to rpm.

proptools.turbopump.**rpm2radsec**(*rpm*)
Convert rpm to radian second**-1.

proptools.turbopump.**ssi_turbine_efficiency**(*uco*)
Efficiency of a single-stage impulse turbine.

Data from 10-9 in Rocket Propulsion Elements.

Parameters **uco** – Velocity ratio u / c_o [units: none].

Returns none].

Return type turbine efficiency [units

proptools.turbopump.**trubine_power**(*p_o*, *p_e*, *m_dot*, *T_o*, *eta*, *gamma*, *c_p*)
Get the output drive power for a turbine.

Parameters

- **p_o** – Turbine inlet stagnation pressure [units: same as p_e].
- **p_e** – Turbine exit pressure [units: same as p_o].
- **m_dot** – Turbine working gas mass flow [units: kilogram second**-1].
- **T_o** – Turbine inlet stagnation temperature [units: kelvin].
- **gamma** – Turbine working gas ratio of specific heats [units: none].

- **c_p** – working gas heat capacity at const pressure [units: joule kilogram**-1 kelvin**-1].

Returns watt].

Return type The shaft power produced by the turbine [units

proptools.turbopump.**turbine_efficiency_demo**()

proptools.turbopump.**turbine_enthalpy**(*p_o*, *p_e*, *T_o*, *gamma*, *c_p*)

Get the specific enthalpy drop for a turbine.

Parameters

- **p_o** – Turbine inlet stagnation pressure [units: same as p_e].
- **p_e** – Turbine exit pressure [units: same as p_o].
- **T_o** – Turbine inlet stagnation temperature [units: kelvin].
- **gamma** – Turbine working gas ratio of specific heats [units: none].
- **c_p** – working gas heat capacity at const pressure [units: joule kilogram**-1 kelvin**-1].

Returns joule kilogram**-1].

Return type The specific enthalpy drop across the turbine [units

proptools.turbopump.**turbine_exit_temperature**(*p_o*, *p_te*, *T_o*, *eta*, *gamma*, *c_p*)

Get the turbine exit temperature.

Parameters

- **p_o** – turbine inlet stagnation pressure [units: pascal].
- **p_te** – turbine exit pressure [units: pascal].
- **T_o** – turbine inlet stagnation temperature [units: kelvin].
- **eta** – turbine efficiency.
- **gamma** – working gas ratio of specific heats [units: none].
- **c_p** – working gas heat capacity at const pressure [units: joule kilogram**-1 kelvin**-1].

proptools.turbopump.**turbine_spout_velocity**(*p_o*, *p_e*, *T_o*, *gamma*, *c_p*)

Get the theoretical spouting velocity for a turbine.

Parameters

- **p_o** – Turbine inlet stagnation pressure [units: same as p_e].
- **p_e** – Turbine exit pressure [units: same as p_o].
- **T_o** – Turbine inlet stagnation temperature [units: kelvin].
- **gamma** – Turbine working gas ratio of specific heats [units: none].
- **c_p** – working gas heat capacity at const pressure [units: joule kilogram**-1 kelvin**-1].

Returns mater second**-1].

Return type The theoretical spouting velocity c_o of the turbine [units

proptools.turbopump_test module

class proptools.turbopump_test.**TestStringMethods**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_gpm_m_dot_inverse()

test_m_dot2gpm()

test_rpm_radsec_inverse()

test_sample61()

proptools.units module

Unit conversions.

proptools.units.**inch2meter**(*x*)

proptools.units.**kilogram2lbm**(*x*)

proptools.units.**lbf2newton**(*x*)

proptools.units.**lbm2kilogram**(*x*)

proptools.units.**meter2inch**(*x*)

proptools.units.**newton2lbf**(*x*)

proptools.units.**pascal2psi**(*x*)

proptools.units.**psi2pascal**(*x*)

Module contents

- genindex

Bibliography

[RPE] G. P. Sutton and O. Biblarz, *Rocket Propulsion Elements*, Hoboken: John Wiley & Sons, 2010.

[Lozano] P. Lozano, *16.522 Lecture Notes*, Lecture 3-4 Mission Analysis for Electric Propulsion. Online: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-522-space-propulsion-spring-2015/lecture-notes/MIT16_522S15_Lecture3-4.pdf

Python Module Index

p

Index

V