PC6432 Project: Relativistic Planetary Projectile Motion with Atmospheric Drag and Mach Speed Aerodynamics

Yun

Projectile motion is often modelled with a simple quadratic curve, or with drag $F_D = \frac{1}{2}\rho v^2 C_D A$. In many cases, C_D is assumed to be fixed at 0.5 (sphere).

This is, of course, not as accurate as the sphere's C_D changes with its Reynolds number, and will also exhibit significant difference during transonic and supersonic airflow. This could be attributed to several reasons such as flow separation, the difference between turbulent flow and laminar flow, and the formation of "air shield" around the sphere etc.

To simulate projectile motion with reasonably high accurate drag force, I have decided to reference the paper **Supersonic and Hypersonic Drag Coefficients for a Sphere** published 8 Aug 2021. It references experimental data and comparisons with previous models to support its drag coefficient at subsonic or supersonic speeds.

Furthermore, a projectile with sufficient range would experience the curvature of Earth or whichever celestial body it was launched on, this would affect its atmospheric altitude, direct and strength of gravity, air density, air temperature, speed of sound etc.

Lastly, with high speeds, notably above 1 percent, the speed of light $0.01c = 2,998 km/s$, relativistic effects would be significant, hence would affect the accuracy of the simulation.

Hence, to have a proper projectile simulator, we have to take into account all of these factors, to model the motion of a smooth rigid spherical sphere with fixed size experiencing atmospheric drag and gravity.

```
# imports
import multiprocessing
from os.path import exists

# used for multiprocessing jobs
PROCESSES = 8

from math import tanh, exp, sqrt, pi, sin, cos, radians, ceil, log2, atan2, degrees, acos
from tqdm import tqdm
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.collections as mcoll
import numpy as np
from numpy import sign

matplotlib.rcParams['figure.figsize'] = (16, 9)


class Vector(object):
    """
    3-D vector implementation.
    Purpose: Provide
    * Core functionality to ray tracing implementation.
    * Examples of using special methods & overriding class attributes
    (which is why packages such as `numpy` or `attrs` are not used).
    ---
    NOTE: Implementation prioritizes readability/simplicity over efficiency.
    Strategies for incremental performance improvements include:
      *  Define the `__slots__` class attribute.
      *  Use the dot operator less frequently.
      *  Support multiprocessing.
    Dramatic performance improvement:
      *  Run using PyPy, instead of CPython (since PyPy implements Python 3.5.3,
         make minor changes like using string formatting in place of f-strings).
    """

    def __init__(self, x: float, y: float, z: float):
        self.x = x
        self.y = y
```

```
        self.z = z

    def __repr__(self):  # for debugging
        return f"Vector3({self.x}, {self.y}, {self.z})"

    def __str__(self):  # for printing
        return f"{self.x}, {self.y}, {self.z}"

    def __eq__(self, other):
        """Overload equality operator."""
        if not isinstance(other, Vector):
            return False
        return (self.x == other.x) and (self.y == other.y) and (self.z == other.z)

    def __add__(self, other):
        """Overload addition operator."""
        if isinstance(other, Vector):
            x = self.x + other.x
            y = self.y + other.y
            z = self.z + other.z
        else:
            x = self.x + other
            y = self.y + other
            z = self.z + other
        return Vector(x, y, z)

    # Support "reverse[d]" addition.
    # Aliasing (w/c?) works because addition is commutative.
    # i.e., a + b == b + a
    __radd__ = __add__

    def __sub__(self, other):
        """Overload subtraction operator."""
        if isinstance(other, Vector):
            x = self.x - other.x
            y = self.y - other.y
            z = self.z - other.z
        else:
            x = self.x - other
            y = self.y - other
            z = self.z - other
        return Vector(x, y, z)

    def __rsub__(self, other):
        """Reversed subtraction. Subtraction is noncommutative."""
        if isinstance(other, Vector):
            x = other.x - self.x
            y = other.y - self.y
            z = other.z - self.z
        else:
            x = other - self.x
            y = other - self.y
            z = other - self.z
        return Vector(x, y, z)

    def __mul__(self, other):
        """Overload multiplication operator."""
        if isinstance(other, Vector):
            x = self.x * other.x
            y = self.y * other.y
            z = self.z * other.z
        else:
            x = self.x * other
            y = self.y * other
            z = self.z * other
        return Vector(x, y, z)

    __rmul__ = __mul__  # "Reverse[d]" multiplication
```

```
    def __truediv__(self, other):
        """Overload division operator."""
        if isinstance(other, Vector):
            x = self.x / other.x
            y = self.y / other.y
            z = self.z / other.z
        else:
            x = self.x / other
            y = self.y / other
            z = self.z / other
        return Vector(x, y, z)

    def __rtruediv__(self, other):
        """Reversed division. Division is noncommutative."""
        if isinstance(other, Vector):
            x = other.x / self.x
            y = other.y / self.y
            z = other.z / self.z
        else:
            x = other / self.x
            y = other / self.y
            z = other / self.z
        return Vector(x, y, z)

    def magnitude(self):
        """Compute magnitude (length)."""
        return np.sqrt(self.magnitude2())

    def magnitude2(self):
        """Compute magnitude (length)."""
        x = self.x ** 2
        y = self.y ** 2
        z = self.z ** 2
        return x + y + z

    def dot(self, other):
        """Compute dot product."""
        a = self.x * other.x
        b = self.y * other.y
        c = self.z * other.z
        return a + b + c

    def cross(self, other):
        """Compute cross product."""
        x = (self.y * other.z) - (self.z * other.y)
        y = (self.z * other.x) - (self.x * other.z)
        z = (self.x * other.y) - (self.y * other.x)
        return Vector(x, y, z)

    def unit_vector(self):
        """Create vector whose magnitude is 1 (but retains self's direction)."""
        return self / self.magnitude()

    def tup(self):
        return self.x, self.y, self.z


def partial(f, *args, **kwargs):
    def wrapped(*args2, **kwargs2):
        return f(*args, *args2, **kwargs, **kwargs2)

    return wrapped
```

Atmosphere

These attributes should only depend on the parameter h, altitude of the projectile, and do not take into account other variables such as

wind, rain, turbulence for those are out of the scope of this project.

Pressure

$$P(h) = \begin{cases} P_b \exp\left[\frac{-g_0 M(h-h_b)}{R^* T_b}\right], & \text{if } L_b = 0 \\ P_b \left[\frac{T_b + (h-h_b)L_b}{T_b}\right]^{\frac{-g_0 M}{R^* L_b}}, & \text{otherwise} \end{cases}$$

The reference pressure P_b has been precalculated for the start of each layer to save computing power.

Temperature

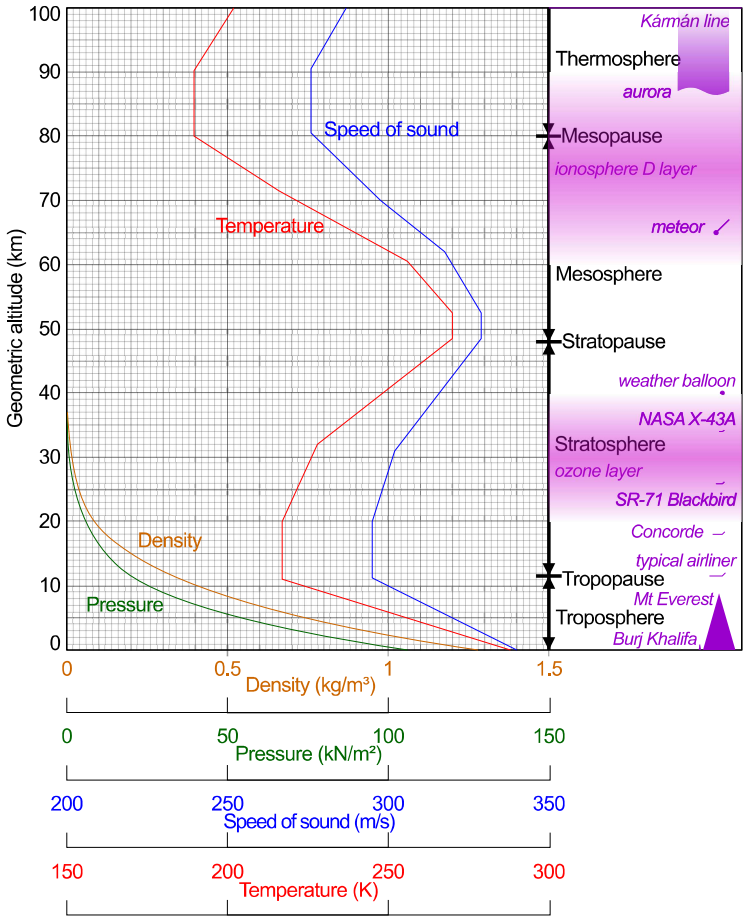
$$T = T_b + L_b \Delta h$$

This follows the simplified temperature model, and is continuous by precalculating the reference temperature for each layer.

Density

$$\rho = \frac{P}{R^* T}$$

Reference Graph



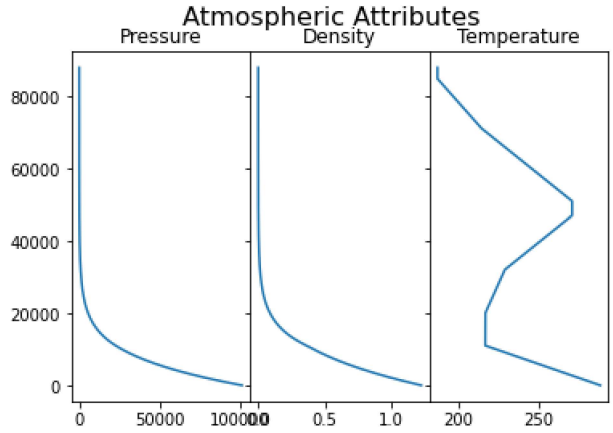

```
'''
Constants used for this simulator

ISA: International Standard Atmosphere
'''
c = 299_792_458
G = 6.6743e-11  # Gravitational Constant
R_E = 6371000  # Radius of Earth
M_E = 5.97219e24  # Mass of Earth
R_u = 8.3144598  # Universal Gas Constant
E_M = 0.0289644  # Earth gas molar mass
```

```
R_s = R_u / E_M  # Specific Gas Constant
ISA_LEVELS = [-610, 11000, 20000, 32000, 47000, 51000, 71000, 84852]
ISA_TEMPERATURES = [292.15, 216.685, 216.685, 228.685, 270.685, 270.685, 214.685, 186.981]
ISA_LAPSE = [-0.0065, 0.0, 0.001, 0.0028, 0, -0.0028, -0.002, 0]
ISA_PRESSURE = [108870.81, 22710.93210827388, 5533.442250839256, 891.5810971148962, 117.0960635813038, 71.197
5245770445,
                4.462120968034988, 0.4466509986473006]
```

```
def g(h: float):
    return G / (h + R_E) * M_E / (h + R_E)


# Which layer am I in?
def layer(h: float):
    i = len(ISA_LEVELS)
    while i > 0:
        i -= 1
        if h > ISA_LEVELS[i]: return i
    return 0


def temperature(h: float, l=None):
    if l is None: l = layer(h)
    return ISA_TEMPERATURES[l] + (h - ISA_LEVELS[l]) * ISA_LAPSE[l]


def pressure(h: float, l=None):
    if l is None: l = layer(h)
    p = ISA_PRESSURE[l]
    if ISA_LAPSE[l] == 0: # zero lapse: constant temperature
        return p * exp(-g(h) * E_M * (h - ISA_LEVELS[l]) / (R_u * ISA_TEMPERATURES[l]))
    else: # variable temperature
        return p * pow((ISA_TEMPERATURES[l] + (h - ISA_LEVELS[l]) * ISA_LAPSE[l]) / ISA_TEMPERATURES[l],
                       -(g(h) * E_M) / (R_u * ISA_LAPSE[l]))


def density(h: float, l=None):
    if l is None: l = layer(h)
    return pressure(h, l) / (R_s * temperature(h, l))
```

```
fig, (a1, a2, a3) = plt.subplots(1, 3, sharey=True)
fig.suptitle("Atmospheric Attributes", fontsize=16)
fig.subplots_adjust(wspace=0, hspace=0)
H = np.linspace(1, 88_000, 1000)

a1.plot(np.vectorize(pressure)(H), H)
a1.set_title("Pressure")
a2.plot(np.vectorize(density)(H), H)
a2.set_title("Density")
a3.plot(np.vectorize(temperature)(H), H)
a3.set_title("Temperature")
```

```
Text(0.5, 1.0, 'Temperature')
```



Mach Speed Aerodynamics

$$H(M) = \begin{cases} 0.0239 \cdot M^3 + 0.212 \cdot M^2 - 0.074 \cdot M + 1 & \text{for } M \leq 1 \\ 0.93 + \frac{1}{3.5+M^5} & \text{for } M > 1 \end{cases}$$

$$G(M) = \begin{cases} 166M^3 + 3.29M^2 - 10.9M + 20 & \text{for } M \leq 0.8 \\ 5 + 40M^{-3} & \text{for } M > 0.8 \end{cases}$$

$$C(M) = \begin{cases} 1.65 + 0.65\tanh(4M - 3.4) & \text{for } M \leq 1.5 \\ 2.18 - 0.13\tanh(0.9M - 2.7) & \text{for } M > 1.5 \end{cases}$$

$$C_D\left(R_E, M\right) = \frac{24}{R_E}\left(1 + 0.15{R_E}^{0.687}\right) H\left(M\right) + \frac{0.42C\left(M\right)}{1 + \frac{42{,}500}{R^{1.16C(M)}} + \frac{G(M)}{R^{0.5}}}$$

```
def speed_of_sound(t):
    return sqrt(1.400 * (R_s * t))


def dynamic_viscosity(t):
    S = 1.49355617587983
    return 1.81e-5 * pow(t / 288.15, 3 / 2) * (288.15 + S) / (t - S)


def sonic_h(M):
    if M > 1:
        return 0.93 + (1 / (3.5 + M ** 5))
    else:
        return 0.0239 * M ** 3 + 0.212 * M ** 2 - 0.074 * M + 1


def sonic_g(M):
    if M > 0.8:
        return 5 + 40 * M ** (-3)
    else:
        return 166 * M ** 3 + 3.29 * M ** 2 - 10.9 * M + 20


def sonic_c(M):
    if M > 1.5:
        return 2.18 - 0.13 * tanh(0.9 * M - 2.7)
    else:
        return 1.65 + 0.65 * tanh(4 * M - 3.4)


def CD(R, M):
    return (24 / R) * (1 + 0.15 * R ** 0.687) * sonic_h(M) + \
           ((0.42 * sonic_c(M)) / (1 + (42500 / (R ** (1.16 * sonic_c(M)))) + (sonic_g(M) / (R ** 0.5))))
```

Drag force

Not very sure if this equation is accurate for relativistic speeds.

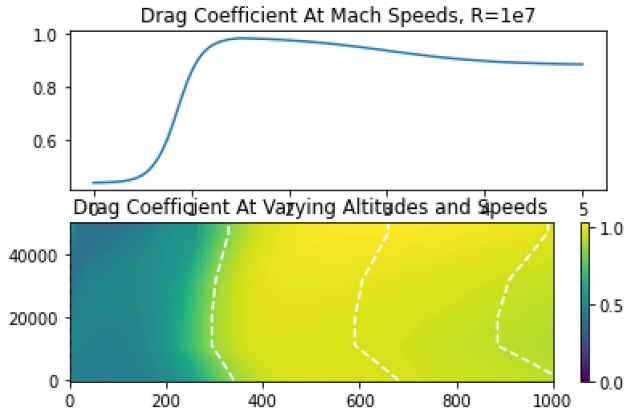
$$\mathrm{Re} = \frac{vL}{\nu} = \frac{\rho v L}{\mu}$$
$$F_D = \tfrac{1}{2}\rho v^2 C_D A$$

Buoyant Force

$$\mathbf{B} = \int \operatorname{div}\sigma \, dV = -\int \mathbf{f} \, dV = -\rho_f \mathbf{g} \int dV = -\rho_f \mathbf{g} V$$

```
def RE(v, d, L, t):
    return d * v * L / dynamic_viscosity(t)


def C_D(v, d, L, t):
    if v == 0: return 0.0
    return CD(float(RE(v, d, L, t)), float(v / speed_of_sound(t)))


def drag(v: Vector, A: float, L: float, h=0):
    if h > 200_000: return 0  # assume no air (exit of atmosphere)
    if h > 170_000: return 1.777e-06 * v.magnitude2() * exp(-(h - 170_000) / 15_000)  # save exec time
    l = layer(h)
    d = density(h, l)
    t = temperature(h, l)
    return -1 * v.unit_vector() * max(0, 0.5 * d * (v.magnitude2()) * C_D(v.magnitude(), d, L, t) * A)
```

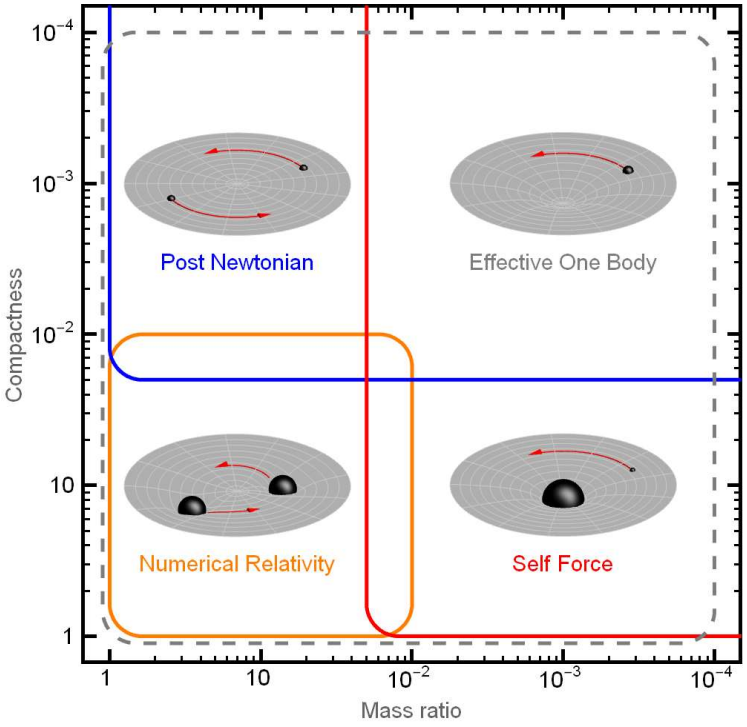
```
fig, (a1, a2) = plt.subplots(2, 1)

M = np.linspace(0, 5, 100)
a1.plot(M, np.vectorize(CD)(1e7, M))
a1.set_title("Drag Coefficient At Mach Speeds, R=1e7")

@np.vectorize
def model_cd_at_height_and_speed(v, h):
    l = layer(h)
    d = density(h, l)
    t = temperature(h, l)
    return C_D(v, d, 1, t)

H = np.linspace(0, 5e4, 100)
Vs, Hs = np.meshgrid(np.linspace(0, 1e3, 200), H)
CDs = model_cd_at_height_and_speed(Vs, Hs)
plt.colorbar(a2.pcolor(Vs, Hs, CDs), ax=a2, fraction=0.05)
a2.set_title("Drag Coefficient At Varying Altitudes and Speeds")
for i in range(10):
    a2.plot(i*np.vectorize(speed_of_sound)(np.vectorize(temperature)(H)), H, linestyle="--", color="white")
a2.set_xlim([0, 1e3])
plt.show()
```



Relativity

Since a projectile could be travelling at near light speed, it would experience relativistic effects such as time dilation.

To determine which model to best estimate these effects, we can refer to the diagram of the parameter space of compact binaries with the various approximation schemes and their regions of validity.



Compactness

Earth does not fall into the category of white dwarfs, neutron stars or black holes, and is hence not a compact body.

Mass ratio

The projectile is very light in comparison to Earth, hence the mass ratio is definitely lower than the threshold for two body questions.

We can hence categorize the question as effectively a one-body problem.

Gravity (general relativity)

$$\mathbf{L} = \mathbf{x} \times \mathbf{p}$$
$$F_G(r) = -\frac{GMm}{r^2} + \frac{L^2}{mr^3} - \frac{3GML^2}{mc^2r^4}$$

Relativistic Mechanics

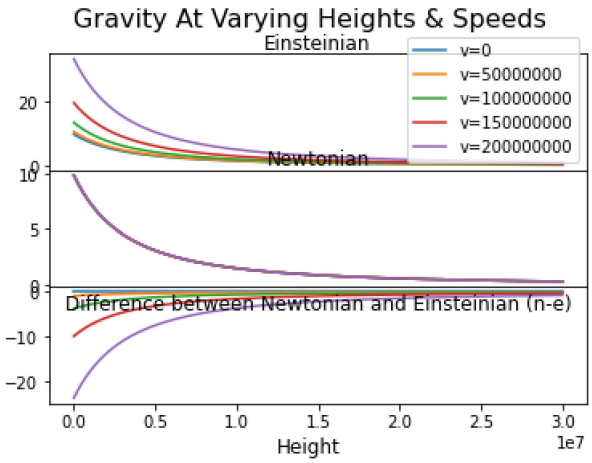
$$E = \gamma(\mathbf{v})m_0c^2$$
$$\mathbf{p} = \gamma(\mathbf{v})m_0\mathbf{v}$$

Since force is the derivative of momentum, $F = ma$ does not apply.

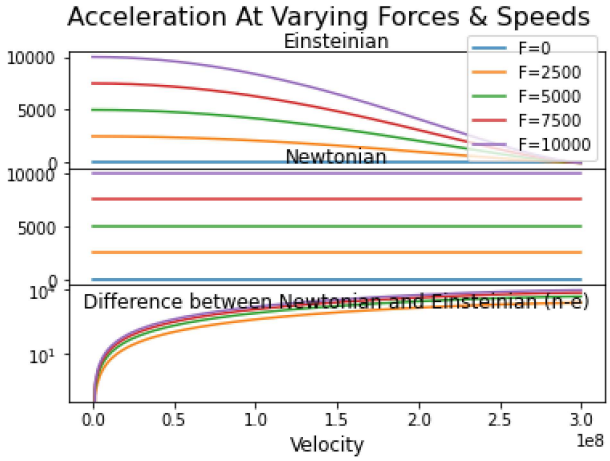
$$\mathbf{F} = \frac{d\mathbf{p}}{dt}$$
$$\mathbf{F} = \gamma(\mathbf{v})^3 m_0 \, \mathbf{a}_{\parallel} + \gamma(\mathbf{v}) m_0 \, \mathbf{a}_{\perp}$$
$$\mathbf{a} = \frac{1}{m_0 \gamma(\mathbf{v})} \left(\mathbf{F} - \frac{(\mathbf{v} \cdot \mathbf{F})\mathbf{v}}{c^2} \right).$$

```
def gravity_orig(pos: Vector, vel: Vector, mass_big: float, mass_small: float, y: float, relativistic=False):
    r = pos.magnitude()
    L = y * mass_small * pos.cross(vel).magnitude()
    F = -G * mass_small * mass_big / (r * r)
    if relativistic:
        F +=  -
 3 * G * mass_big * L * L / (mass_small * c * c * r * r * r * r) #+ L * L / (mass_small * r * r * r)
    return F * pos.unit_vector()


# prevent numerical overflow
def gravity(pos: Vector, vel: Vector, mass_big: float, mass_small: float, y: float, relativistic=False):
    r = pos.magnitude()
    rn = pos.unit_vector()
    L2 = y * mass_small * rn.cross(vel).magnitude2()
    F = -G * mass_small / r * mass_big / r
    if relativistic: F += - 3 * G * mass_big / c / c * L2 / r / r #+ L2 / r
    return F * rn


def acceleration(force: Vector, m_0: float, vel: Vector, y: float, relativistic=False):
    if not relativistic: return force / m_0
    force = force - vel.dot(force) / c * vel / c
    den = m_0 * y
    return force / den


def gamma(v: Vector):
    return 1 / sqrt(1 - v.magnitude2() / c / c)
```

```
H = np.linspace(0, 3e7, 200)
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)
fig.suptitle("Gravity At Varying Heights & Speeds", fontsize=16)
fig.subplots_adjust(wspace=0, hspace=0)
for v in np.linspace(0, 2e8, 5):
    Gn = np.vectorize(lambda h, v: gravity_orig(Vector(0, R_E + h, 0), Vector(v, 0, 0), M_E, 1, gamma(Vector(
v, 0, 0)), False).magnitude())(H, v)
    Go = np.vectorize(lambda h, v: gravity_orig(Vector(0, R_E + h, 0), Vector(v, 0, 0), M_E, 1, gamma(Vector(
v, 0, 0)), True).magnitude())(H, v)
    ax1.plot(H, Go, label=f'v={int(v)}')
    ax2.plot(H, Gn, label=f'v={int(v)}')
    ax3.plot(H, Gn - Go, label=f'v={int(v)}')
ax1.set_title("Einsteinian", pad=-14)
ax2.set_title("Newtonian", pad=-14)
ax3.set_title("Difference between Newtonian and Einsteinian (n-e)", y=1,pad=-14)
fig.supxlabel('Height')
ax1.legend()
plt.show()
```




```
v = np.linspace(0, c-1, 200)
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)
fig.suptitle("Acceleration At Varying Forces & Speeds", fontsize=16)
fig.subplots_adjust(wspace=0, hspace=0)
for F in np.linspace(0, 1e4, 5):
    Gn = np.vectorize(lambda f, v: acceleration(Vector(f, 0, 0), 1, Vector(v, 0, 0), gamma(Vector(v, 0, 0)),
False).magnitude())(F, v)
    Go = np.vectorize(lambda f, v: acceleration(Vector(f, 0, 0), 1, Vector(v, 0, 0), gamma(Vector(v, 0, 0)),
True).magnitude())(F, v)

    ax1.plot(v, Go, label=f'F={int(F)}')
    ax2.plot(v, Gn, label=f'F={int(F)}')
    ax3.plot(v, Gn - Go, label=f'F={int(F)}')
ax1.set_title("Einsteinian", pad=-14)
ax2.set_title("Newtonian", pad=-14)
ax3.set_title("Difference between Newtonian and Einsteinian (n-e)", y=1, pad=-14)
ax3.set_yscale("log")
fig.supxlabel('Velocity')
ax1.legend()
plt.show()
```



One Body Problem

We can see the effect of perihelion precession quite clearly with this example.

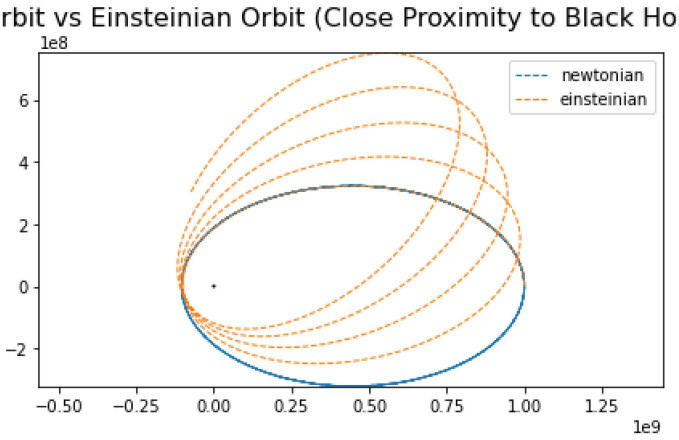
$$\sigma = \frac{24\pi^3 L^2}{T^2 c^2 \left(1 - e^2\right)}$$

```
def one_body(mass_b, mass_s, relativistic=False):
    def dx(t, x, v, y):
        return v
```

```
    def dv(t, x, v, y):
        force = gravity(x, v, mass_b, mass_s, y, relativistic)
        return acceleration(force, mass_s, v, y, relativistic)

    def rk4(t, x, v, dt):
        if relativistic: dt = dt * sqrt(1 - v.magnitude2() / c / c)
        y = gamma(v)
        k0 = dt * dx(t, x, v, y)
        l0 = dt * dv(t, x, v, y)
        k1 = dt * dx(t + dt / 2, x + k0 / 2, v + l0 / 2, y)
        l1 = dt * dv(t + dt / 2, x + k0 / 2, v + l0 / 2, y)
        k2 = dt * dx(t + dt / 2, x + k1 / 2, v + l1 / 2, y)
        l2 = dt * dv(t + dt / 2, x + k1 / 2, v + l1 / 2, y)
        k3 = dt * dx(t + dt, x + k2, v + l2, y)
        l3 = dt * dv(t + dt, x + k2, v + l2, y)
        return (t + dt,
                x + (k0 + 2 * k1 + 2 * k2 + k3) / 6,
                v + (l0 + 2 * l1 + 2 * l2 + l3) / 6)

    def sim(x: Vector, v: Vector, dt, it_count=1e4):
        xy_plot = []
        t = 0
        for i in tqdm(range(int(it_count))):
            t, x, v = rk4(t, x, v, dt)
            xy_plot.append([x.x, x.y])
        return xy_plot

    return sim
```

```
data_n = one_body(1.989e33, 3.285e23, False)(Vector(1_000_000_000, 0, 0), Vector(0, 500_0000, 0), 0.1)
data_r = one_body(1.989e33, 3.285e23, True)(Vector(1_000_000_000, 0, 0), Vector(0, 500_0000, 0), 0.1)
```
```
100%|██████████| 10000/10000 [00:03<00:00, 3099.50it/s]
100%|██████████| 10000/10000 [00:03<00:00, 2667.66it/s]
```

```
fig = plt.figure()
data_n = np.array(data_n)
data_r = np.array(data_r)
x1 = data_n[:, 0]
y1 = data_n[:, 1]
x2 = data_r[:, 0]
y2 = data_r[:, 1]
plt.xlim(min(x1.min(), x2.min()), max(x1.max(), x2.max()))
plt.ylim(min(y1.min(), y2.min()), max(y1.max(), y2.max()))
ax = plt.gca()
ax.add_patch(plt.Circle((0, 0), 2_954_000, color='black'))
ax.plot(x1, y1, ls='--', linewidth=1, label = "newtonian")
ax.plot(x2, y2, ls='--', linewidth=1, label = "einsteinian")
ax.set_aspect('equal', adjustable='datalim')
plt.title("Newtonian Orbit vs Einsteinian Orbit (Close Proximity to Black Hole 0.01667%C)", fontsize=16)
plt.legend()
plt.tight_layout()
```



Relativistic Aerodynamics

I can't find experimental studies on air speeds of relativistic speeds, hence I just have to assume that the drag equation still works.

Simulation

This section will now apply aforementioned models and conduct simulations to investigate projectile motion with atmospheric drag.

- Projectile Launching (from $h = 0$)
  - Flat Land (No curvature of Earth)
  - Curved surface (Earth)
- Projectile reentry (from beyond the atmosphere)
  - Direct impact (high velocity)
  - Angle of Attack comparison
- Cannonball Root Finding
  - From the North Pole to the South Pole
  - From Washington dc to Moscow

Flat Land

A totally flat simulation, achieved by ignoring the x position when dealing with drag and gravity.

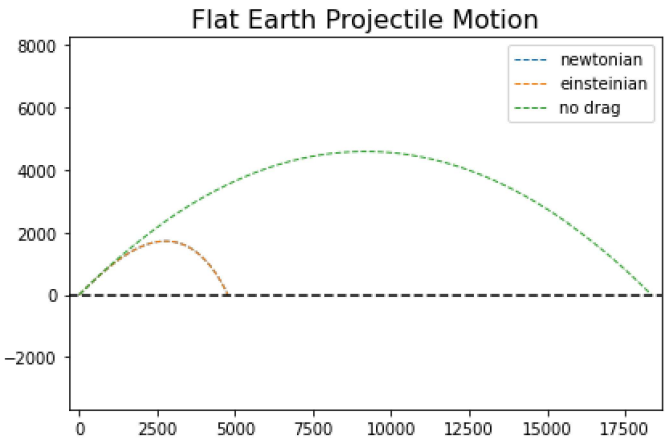
```
def flat_world(mass_b, mass_s, ground_h, radius=0.5, relativistic=False, dragish=True):
    A = pi * radius ** 2
    L = 2 * radius
    V = 4 / 3 * pi * radius ** 3

    def dx(t, x, v, y):
        return v

    def dv(t, x, v, y):
        force = gravity(Vector(0, x.y + ground_h, 0), v, mass_b, mass_s, y, relativistic)  # remove side-
ways gravity
        if dragish:
            force += drag(v, A=A, L=L, h=x.y)
            if x.y < 200_000:
                force += Vector(0, x.y + ground_h, 0).unit_vector() * g(x.y+ground_h) * density(x.y) * V # bu
oyant force
        return acceleration(force, mass_s, v, y, relativistic)

    def rk4(t, x, v, dt):
        if relativistic: dt = dt * sqrt(1 - v.magnitude2() / c / c)
        y = gamma(v)
        k0 = dt * dx(t, x, v, y)
        l0 = dt * dv(t, x, v, y)
        k1 = dt * dx(t + dt / 2, x + k0 / 2, v + l0 / 2, y)
        l1 = dt * dv(t + dt / 2, x + k0 / 2, v + l0 / 2, y)
        k2 = dt * dx(t + dt / 2, x + k1 / 2, v + l1 / 2, y)
        l2 = dt * dv(t + dt / 2, x + k1 / 2, v + l1 / 2, y)
        k3 = dt * dx(t + dt, x + k2, v + l2, y)
        l3 = dt * dv(t + dt, x + k2, v + l2, y)
        return (t + dt,
                x + (k0 + 2 * k1 + 2 * k2 + k3) / 6,
                v + (l0 + 2 * l1 + 2 * l2 + l3) / 6)

    def sim(x: Vector, v: Vector, dt=0.01, it_count=1e6, silence=False):
        data = []
        t = 0
        for _ in range(int(it_count)) if silence else tqdm(range(int(it_count))):
            t, x, v = rk4(t, x, v, dt)
            data.append([t, x.x, x.y, v.x, v.y])
```

```
            if x.y <= 0: break
        return data

    return sim
```

```
data_s = flat_world(M_E, 1000, 6_371_000, radius=0.5, relativistic=False, dragish=False)
(Vector(0, 0, 0), Vector(300, 300, 0), .01)
data_n = flat_world(M_E, 1000, 6_371_000, radius=0.5, relativistic=False)
(Vector(0, 0, 0), Vector(300, 300, 0), .01)
data_r = flat_world(M_E, 1000, 6_371_000, radius=0.5, relativistic=True)
(Vector(0, 0, 0), Vector(300, 300, 0), .01)
```

```
  1%|          | 6115/1000000 [00:02<05:54, 2800.21it/s]
  0%|          | 3682/1000000 [00:02<10:51, 1529.24it/s]
  0%|          | 3682/1000000 [00:03<14:46, 1124.47it/s]
```

```
fig = plt.figure()
data_n = np.array(data_n)
data_r = np.array(data_r)
data_s = np.array(data_s)
x1 = data_n[:, 1]
y1 = data_n[:, 2]
x2 = data_r[:, 1]
y2 = data_r[:, 2]
x3 = data_s[:, 1]
y3 = data_s[:, 2]
plt.xlim(min(x1.min(), x2.min(), x3.min()), max(x1.max(), x2.max(), x3.max()))
plt.ylim(min(y1.min(), y2.min(), y3.min()), max(y1.max(), y2.max(), y3.max()))
ax = plt.gca()
# ax.add_patch(plt.Circle((0, 0), 696_340_000, color='black'))
ax.plot(x1, y1, ls='--', linewidth=1, label = "newtonian")
ax.plot(x2, y2, ls='--', linewidth=1, label = "einsteinian")
ax.plot(x3, y3, ls='--', linewidth=1, label = "no drag")
ax.axhline(0, color='black', ls='--')
ax.set_aspect('equal', adjustable='datalim')
plt.title("Flat Earth Projectile Motion", fontsize=16)
plt.legend()
plt.tight_layout()
```



There does not seem to be much difference when deep in the atmosphere, which is reasonable since the projectile is travelling at $\ll 1\%\, c$, and atmospheric drag is much more significant.

However, we can clearly see the effect of drag, which is very significant.

Different objects of varying density and size

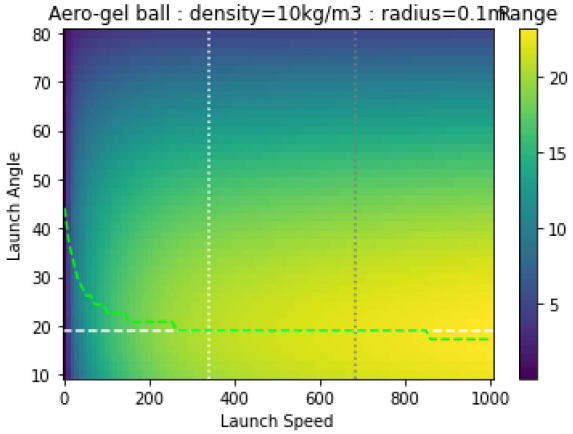
```
PRESETS = [
    ("Aero-gel ball", 10, 0.1, 0.001),
    ("Tennis ball", 377, 0.03329, 0.01),
    ("Cannonball", 7860, 0.11, 0.01),
```

```
    ("Football", 77.13, 0.11, 0.01),
    ("Tungsten 10mm Bullet", 19250, 0.01, 0.1),
    ("Tungsten Football", 19250, 0.11, 0.1)
]

def flat_earth(density, radius, relativistic=False):
    volume = 4 / 3 * pi * radius ** 3
    mass = volume * density
    return flat_world(M_E, mass, ground_h=6_371_000, radius=radius, relativistic=relativistic)
```

```
y_angles = np.linspace(10, 80, 40)
x_velocities = np.linspace(1, 1000, 100)
xv, yv = np.meshgrid(x_velocities, y_angles)
```

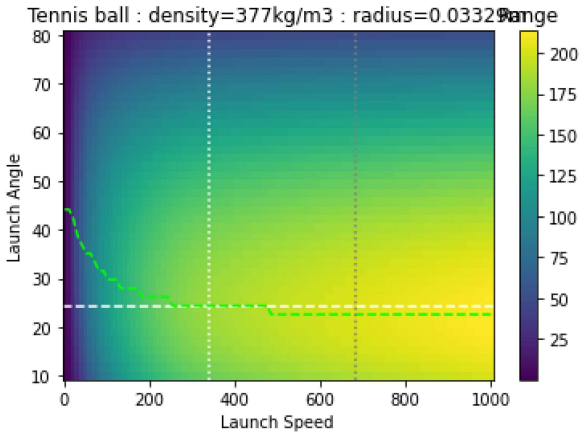
```
def flat_range(v, silence=True):
    ranges = np.zeros(y_angles.size)
    for i in range(y_angles.size):
        a = y_angles[i]
        data = np.array(world(Vector(0, 0, 0), Vector(v * cos(radians(a)), v * sin(radians(a)), 0), dt=preset
_dt, silence=silence))
        ranges[i] = [data[-1, 1], data[:,2].max(), data[-1, 0]]
    return ranges

def plot_heat_range(R, name, density, radius):
    plt.pcolor(xv, yv, R)
    plt.title(f"{name} : density={density}kg/m3 : radius={radius}m")
    plt.ylabel("Launch Angle")
    plt.xlabel("Launch Speed")
    plt.colorbar().ax.set_title('Range')
    best_angle = R.sum(axis=1).argmax()
    plt.axhline(yv[best_angle][0], color='white', ls='--', label=f'{round(yv[best_angle][0], 1)}°')
    plt.plot(x_velocities, yv[R.argmax(axis=0), 0], ls='--', color='lime')
    plt.axvline(speed_of_sound(temperature(0)), color="white", ls=":")
    plt.axvline(2*speed_of_sound(temperature(0)), color="gray", ls=":")
    plt.show()
```

```
preset_name, preset_density, preset_radius, preset_dt = PRESETS[0]
path = f"graphs/graph-d{preset_density}-r{preset_radius}.npy"

world = flat_earth(preset_density, preset_radius)
if exists(path):
    Ds = np.load(path)
else:
    with multiprocessing.Pool(PROCESSES) as pool:
        Ds = np.array(list(tqdm(pool.imap(flat_range, x_velocities), total=x_velocities.size)))
    np.save(path, Ds)

plot_heat_range(Ds[:, :, 0], preset_name, preset_density, preset_radius)
```

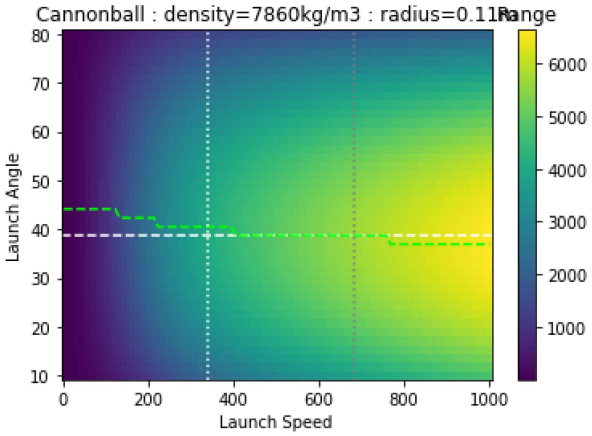



```
preset_name, preset_density, preset_radius, preset_dt = PRESETS[1]
path = f"graphs/graph-d{preset_density}-r{preset_radius}.npy"
```

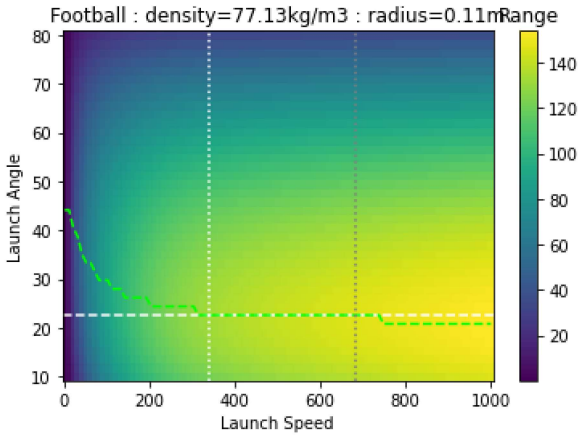
```
world = flat_earth(preset_density, preset_radius)
if exists(path):
    Ds = np.load(path)
else:
    with multiprocessing.Pool(PROCESSES) as pool:
        Ds = np.array(list(tqdm(pool.imap(flat_range, x_velocities), total=x_velocities.size)))
    np.save(path, Ds)

plot_heat_range(Ds[:, :, 0], preset_name, preset_density, preset_radius)
```



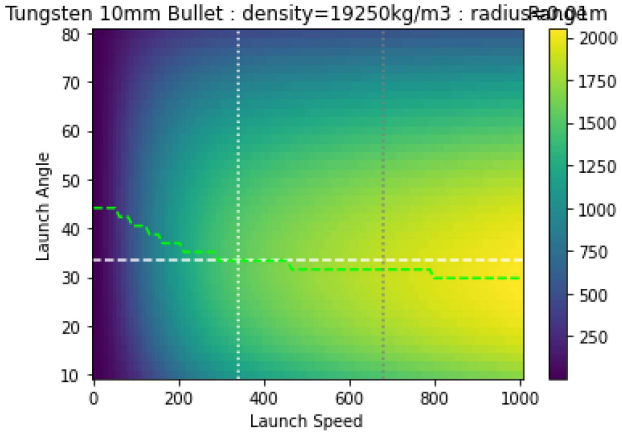

```
preset_name, preset_density, preset_radius, preset_dt = PRESETS[2]
path = f"graphs/graph-d{preset_density}-r{preset_radius}.npy"

world = flat_earth(preset_density, preset_radius)
if exists(path):
    Ds = np.load(path)
else:
    with multiprocessing.Pool(PROCESSES) as pool:
        Ds = np.array(list(tqdm(pool.imap(flat_range, x_velocities), total=x_velocities.size)))
    np.save(path, Ds)

plot_heat_range(Ds[:, :, 0], preset_name, preset_density, preset_radius)
```




```
preset_name, preset_density, preset_radius, preset_dt = PRESETS[3]
path = f"graphs/graph-d{preset_density}-r{preset_radius}.npy"

world = flat_earth(preset_density, preset_radius)
if exists(path):
    Ds = np.load(path)
else:
    with multiprocessing.Pool(PROCESSES) as pool:
        Ds = np.array(list(tqdm(pool.imap(flat_range, x_velocities), total=x_velocities.size)))
    np.save(path, Ds)

plot_heat_range(Ds[:, :, 0], preset_name, preset_density, preset_radius)
```




```
preset_name, preset_density, preset_radius, preset_dt = PRESETS[4]
path = f"graphs/graph-d{preset_density}-r{preset_radius}.npy"

world = flat_earth(preset_density, preset_radius)
if exists(path):
    Ds = np.load(path)
else:
    with multiprocessing.Pool(PROCESSES) as pool:
        Ds = np.array(list(tqdm(pool.imap(flat_range, x_velocities), total=x_velocities.size)))
    np.save(path, Ds)

plot_heat_range(Ds[:, :, 0], preset_name, preset_density, preset_radius)
```



Evaluation

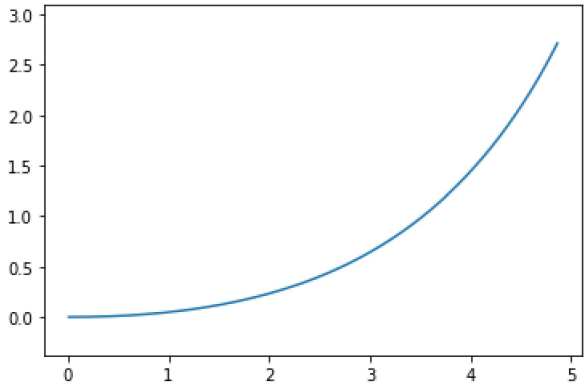
From the simulations, we can clearly see that at lower speeds, the optimal angle of launch is 45°. And as the launch speed increases, the optimal launch angle always decreases.

We can also see that the rate at which the optimal launch angle decreases is positively related to density while radius.

Lighter or smaller objects have lower optimal launch angles.

No significant pattern seems to exist about the sound barrier.

```
# Balloon

world = flat_earth(0.167, 5) # Helium
data=np.array(world(Vector(0,0,0),Vector(10,0,0),it_count=1e3,dt=0.001))
plt.plot(data[:,1],data[:,2])
plt.gca().set_aspect('equal', adjustable='datalim')
```




```
100%|██████████| 1000/1000 [00:00<00:00, 1673.13it/s]
```

Curved Land

Accounting for the curvature of Earth. A projectile would see the ground curve away as it flies.

```
def curve_world(mass_s, radius=0.5, relativistic=False, dragish=True):
    A = pi * radius ** 2
    L = 2 * radius
    V = 4 / 3 * pi * radius ** 3

    def dx(t, x, v, y):
        return v

    def dv(t, x, v, y):
        force = gravity(x, v, M_E, mass_s, y, relativistic)  # remove side-ways gravity
        if dragish:
            h = x.magnitude()
            force += drag(v, A=A, L=L, h=h - R_E)
            if h < 200_000 + R_E:
                force += x.unit_vector() * g(h) * density(h - R_E) * V  # buoyant force
        return acceleration(force, mass_s, v, y, relativistic)

    def rk4(t, x, v, dt):
        if relativistic: dt = dt * sqrt(1 - v.magnitude2() / c / c)
        y = gamma(v)
        k0 = dt * dx(t, x, v, y)
        l0 = dt * dv(t, x, v, y)
        k1 = dt * dx(t + dt / 2, x + k0 / 2, v + l0 / 2, y)
        l1 = dt * dv(t + dt / 2, x + k0 / 2, v + l0 / 2, y)
        k2 = dt * dx(t + dt / 2, x + k1 / 2, v + l1 / 2, y)
        l2 = dt * dv(t + dt / 2, x + k1 / 2, v + l1 / 2, y)
        k3 = dt * dx(t + dt, x + k2, v + l2, y)
        l3 = dt * dv(t + dt, x + k2, v + l2, y)
        return (t + dt,
                x + (k0 + 2 * k1 + 2 * k2 + k3) / 6,
                v + (l0 + 2 * l1 + 2 * l2 + l3) / 6)

    def sim(x: Vector, v: Vector, dt=0.01, it_count=1e6, silence=False, max_height=2 * R_E):
        data = []
        t = 0
        for _ in range(int(it_count)) if silence else tqdm(range(int(it_count))):
            t, x, v = rk4(t, x, v, dt)
            data.append([t, x.x, x.y, v.x, v.y])
            h = x.magnitude()
            if h <= R_E or h >= max_height: break
        return data

    return sim
```

```
def curve_earth(density, radius, relativistic=False, dragish=True):
    volume = 4 / 3 * pi * radius ** 3
    mass = volume * density
    return curve_world(mass, radius=radius, relativistic=relativistic, dragish=dragish)
```

```
def make_segments(x, y):
    points = np.array([x, y]).T.reshape(-1, 1, 2)
    segments = np.concatenate([points[:-1], points[1:]], axis=1)
    return segments

def colorline(x, y, z=None, cmap=plt.get_cmap('copper'), norm=plt.Normalize(0.0, 1.0),
              linewidth=3, alpha=1.0):
    if z is None:
        z = np.linspace(0.0, 1.0, len(x))

    # Special case if a single number:
    if not hasattr(z, "__iter__"):  # to check for numerical input -- this is a hack
        z = np.array([z])

    z = np.asarray(z)

    segments = make_segments(x, y)
    lc = mcoll.LineCollection(segments, array=z, cmap=cmap, norm=norm,
                              linewidth=linewidth, alpha=alpha)

    ax = plt.gca()
    ax.add_collection(lc)

    return lc
```

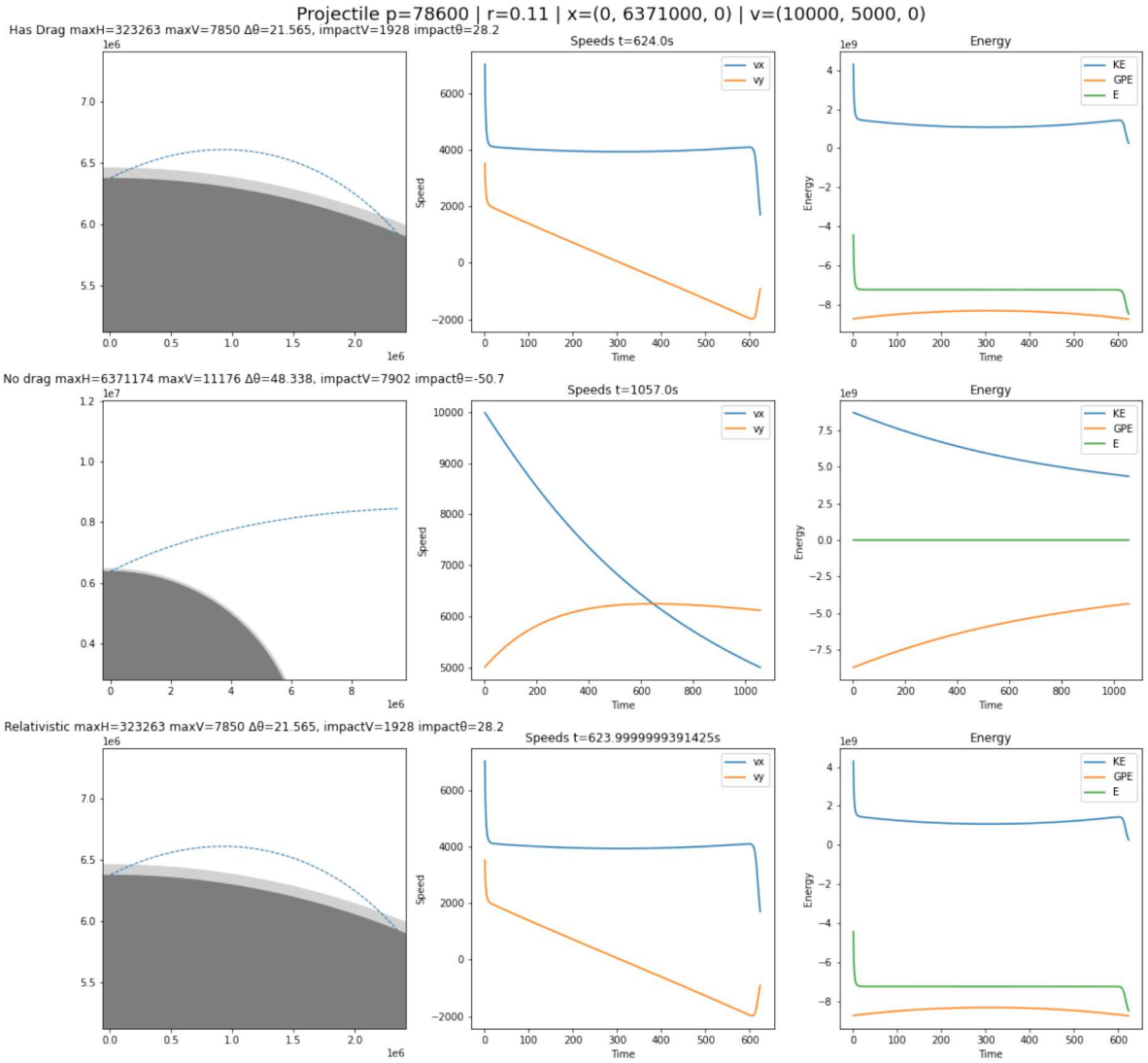
```
def normalize(data):
    return data / np.max(data)


def plot_earth(axi, data, title="Launch", label=""):
    x = data[:, 1]
    y = data[:, 2]
    vx = data[:, 3]
    vy = data[:, 4]

    axi.set_xlim([x.min() - 1000, x.max() + 1000])
    axi.set_ylim([y.min() - 1000, y.max() + 1000])
    axi.set_title(title)
    axi.add_patch(plt.Circle((0, 0), R_E + ISA_LEVELS[7], color='lightgrey'))
    axi.add_patch(plt.Circle((0, 0), R_E, color='grey'))

    axi.plot(x, y, ls='--', linewidth=1, label=label)
    # colorline(x, y,
    #         normalize(np.log(np.hypot(vx, vy))),
    #         cmap=plt.get_cmap('jet'), linewidth=2)
    axi.set_aspect('equal', adjustable='datalim')


def plot_speeds(axi, data, title="Speeds", label=""):
    def normal_speed(x, y, vx, vy):
        r = Vector(x, y, 0).unit_vector()
        v = Vector(vx, vy, 0)
        vy = r.dot(v)
        vx = np.sqrt(v.magnitude2() - vy * vy)
        return vx, vy

    vx, vy = normal_speed(data[:, 1], data[:, 2], data[:, 3], data[:, 4])
    axi.set_title(f"{title} t={data[-1, 0]}s")
    axi.plot(data[:, 0], vx, label="vx")
    axi.plot(data[:, 0], vy, label="vy")
    axi.set_xlabel("Time")
    axi.set_ylabel("Speed")
```

```
    axi.legend()

def plot_energy(axi, data, mass, rel=False):
    x = Vector(data[:, 1], data[:, 2], 0)
    v = Vector(data[:, 3], data[:, 4], 0)

    r = x.magnitude()
    v2 = v.magnitude2()
    ke = 0.5 * mass * v2
    gpe = -G*M_E*mass/r
    axi.plot(data[:, 0], ke, label="KE")
    axi.plot(data[:, 0], gpe, label="GPE")
    axi.plot(data[:, 0], ke+gpe, label="E")
    axi.set_title("Energy")
    axi.set_xlabel("Time")
    axi.set_ylabel("Energy")
    axi.legend()

def get_stat(data):
    x = Vector(data[:, 1], data[:, 2], 0)
    v = Vector(data[:, 3], data[:, 4], 0)
    r = x.magnitude()
    vm = v.magnitude()
    deltaAngle = degrees(atan2(data[0, 2], data[0, 1]) - atan2(data[-1, 2], data[-1, 1]))
    lastX = Vector(data[-1, 1], data[-1, 2], 0)
    lastV = Vector(data[-1, 3], data[-1, 4], 0)
    impactAngle = degrees(pi/2-acos(-lastX.unit_vector().dot(lastV.unit_vector())))
    maxH = r.max() - R_E
    return maxH, vm.max(), deltaAngle, vm[-1], impactAngle

def get_info(data):
    maxH, maxV, delA, impV, impA = get_stat(data)
    return f"maxH={round(maxH)} maxV={round(maxV)} Δθ={round(delA,3)}, impactV={round(impV)} impactθ=
{round(impA,1)}"
```

Launching from the Surface

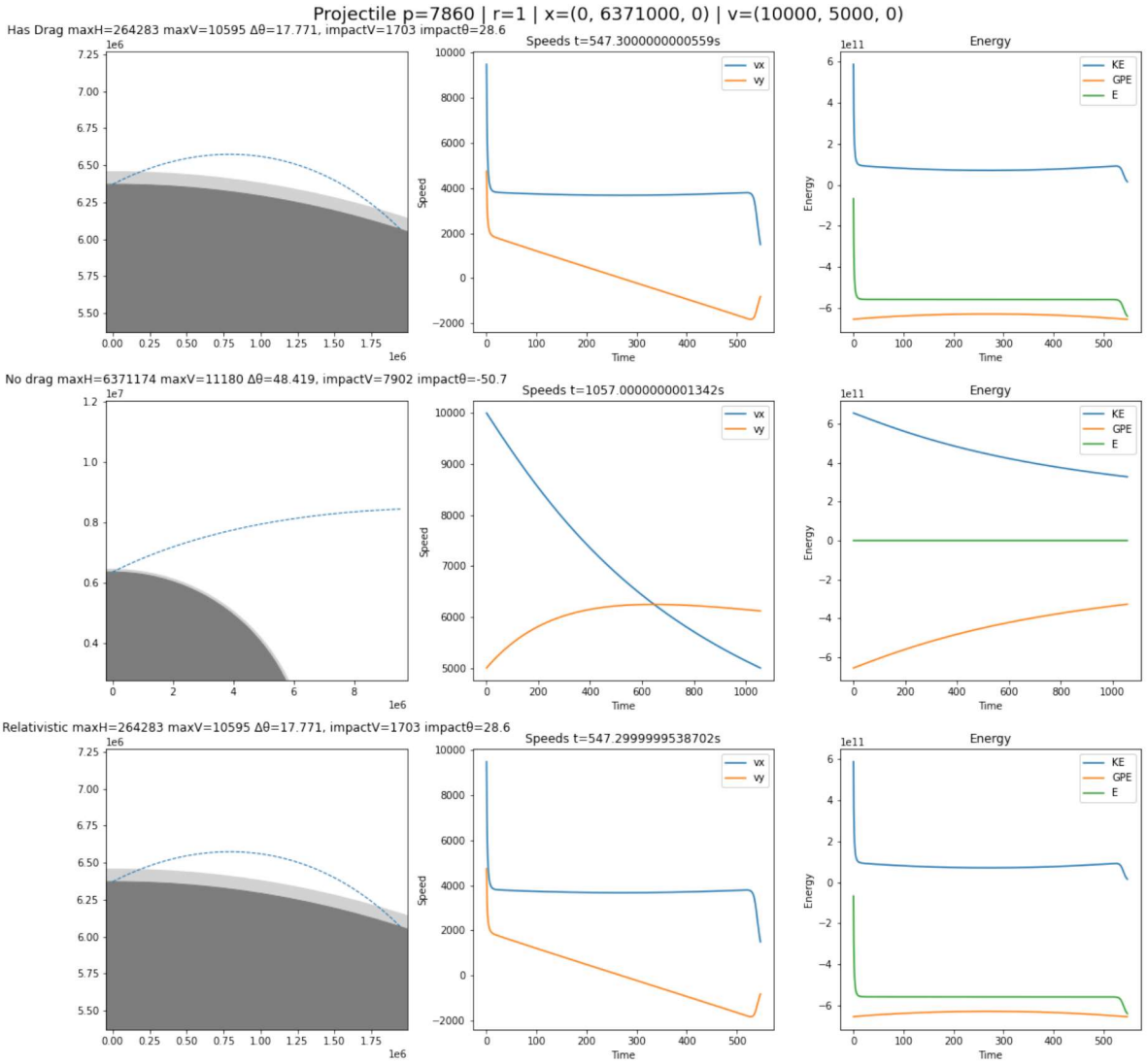
```
def plot_three_cases(obj_density, obj_radius, x, v, dt=1.0, max_height=2 * R_E, max_it=1e6):
    world = curve_earth(obj_density, obj_radius)
    world_no_drag = curve_earth(obj_density, obj_radius, dragish=False)
    world_rela = curve_earth(obj_density, obj_radius, relativistic=True)
    data1 = np.array(world(x, v, dt, max_height=max_height, it_count=max_it))
    data2 = np.array(world_no_drag(x, v, dt, max_height=max_height, it_count=max_it))
    data3 = np.array(world_rela(x, v, dt, max_height=max_height, it_count=max_it))

    mass = 4 / 3 * obj_radius ** 3 * obj_density

    fig, ((ax1, ax2, e1), (ax3, ax4, e2), (ax5, ax6, e3)) = plt.subplots(3, 3)
    fig.set_figheight(15)
    fig.set_figwidth(15)
    fig.suptitle(f"Projectile p={obj_density} | r={obj_radius} | x=({x}) | v=({v})", fontsize=18)
    plot_earth(ax1, data1, title="Has Drag " + get_info(data1))
    plot_speeds(ax2, data1)
    plot_energy(e1, data1, mass)
    plot_earth(ax3, data2, title="No drag " + get_info(data2))
    plot_speeds(ax4, data2)
    plot_energy(e2, data2, mass)
    plot_earth(ax5, data3, title="Relativistic " + get_info(data3))
    plot_speeds(ax6, data3)
    plot_energy(e3, data3, mass)
    fig.tight_layout()
```

```
plot_three_cases(78600, 0.11, Vector(0, R_E, 0), Vector(10000, 5000, 0))
```




```
  0%|          | 623/1000000 [00:00<06:07, 2721.65it/s]
  0%|          | 1056/1000000 [00:00<04:02, 4126.94it/s]
  0%|          | 623/1000000 [00:00<11:30, 1447.20it/s]
```

```
plot_three_cases(7860, 1, Vector(0, R_E, 0), Vector(10000, 5000, 0), dt=0.1)
```




```
  1%|          | 5472/1000000 [00:02<06:31, 2541.80it/s]
  1%|          | 10569/1000000 [00:02<04:32, 3628.47it/s]
  1%|          | 5472/1000000 [00:02<06:44, 2455.75it/s]
```

Evaluation

Relativity does not affect these projectiles much, since they are travelling at low speeds.

Reentry

When projectiles are flying back down from space.

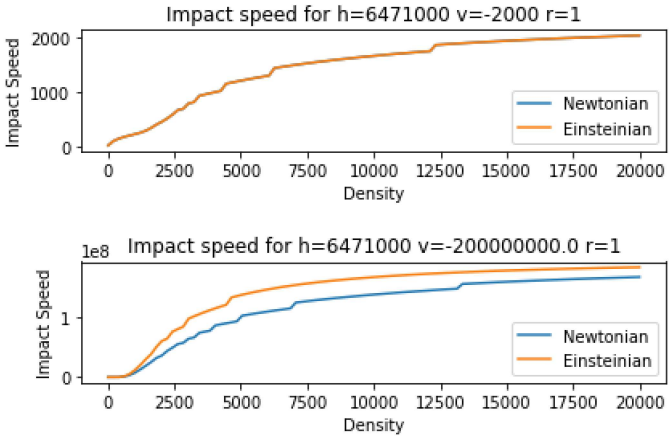
Direct Impact

A projectile shot downwards from space.

```
def plot_density_impact(ax, obj_radius, x, v, dt=1.0, max_it=1e6):
    path = f"graphs/plot_density_impact-{obj_radius}-{x}-{v}.npy"
    D = np.linspace(10, 20000, 100)
    ax.set_title(f"Impact speed for h={x} v={v} r={obj_radius}")
    x = Vector(0, x, 0)
    v = Vector(0, v, 0)
    if exists(path):
        data = np.load(path)
    else:
        data = []
```

```
        for obj_density in tqdm(D):
            world = curve_earth(obj_density, obj_radius)
            world_r = curve_earth(obj_density, obj_radius, relativistic=True)
            data1 = np.array(world(x, v, dt, silence=True, it_count=max_it))
            data2 = np.array(world_r(x, v, dt, silence=True, it_count=max_it))

            _, _, _, v1, _ = get_stat(data1)
            _, _, _, v2, _ = get_stat(data2)
            data.append([v1, v2])
        data = np.array(data)
        np.save(path, data)
    ax.set_xlabel("Density")
    ax.set_ylabel("Impact Speed")
    ax.plot(D, data[:, 0], label="Newtonian")
    ax.plot(D, data[:, 1], label="Einsteinian")
    ax.legend()
```

```
fig, (ax1, ax2) = plt.subplots(2, 1)
plot_density_impact(ax1, 1, R_E + 100_000, -2000)
plot_density_impact(ax2, 1, R_E + 100_000, -2e8, dt=0.00001, max_it=1e4)
fig.tight_layout()
```



There appears to be jumps in impact speed for certain values of density.

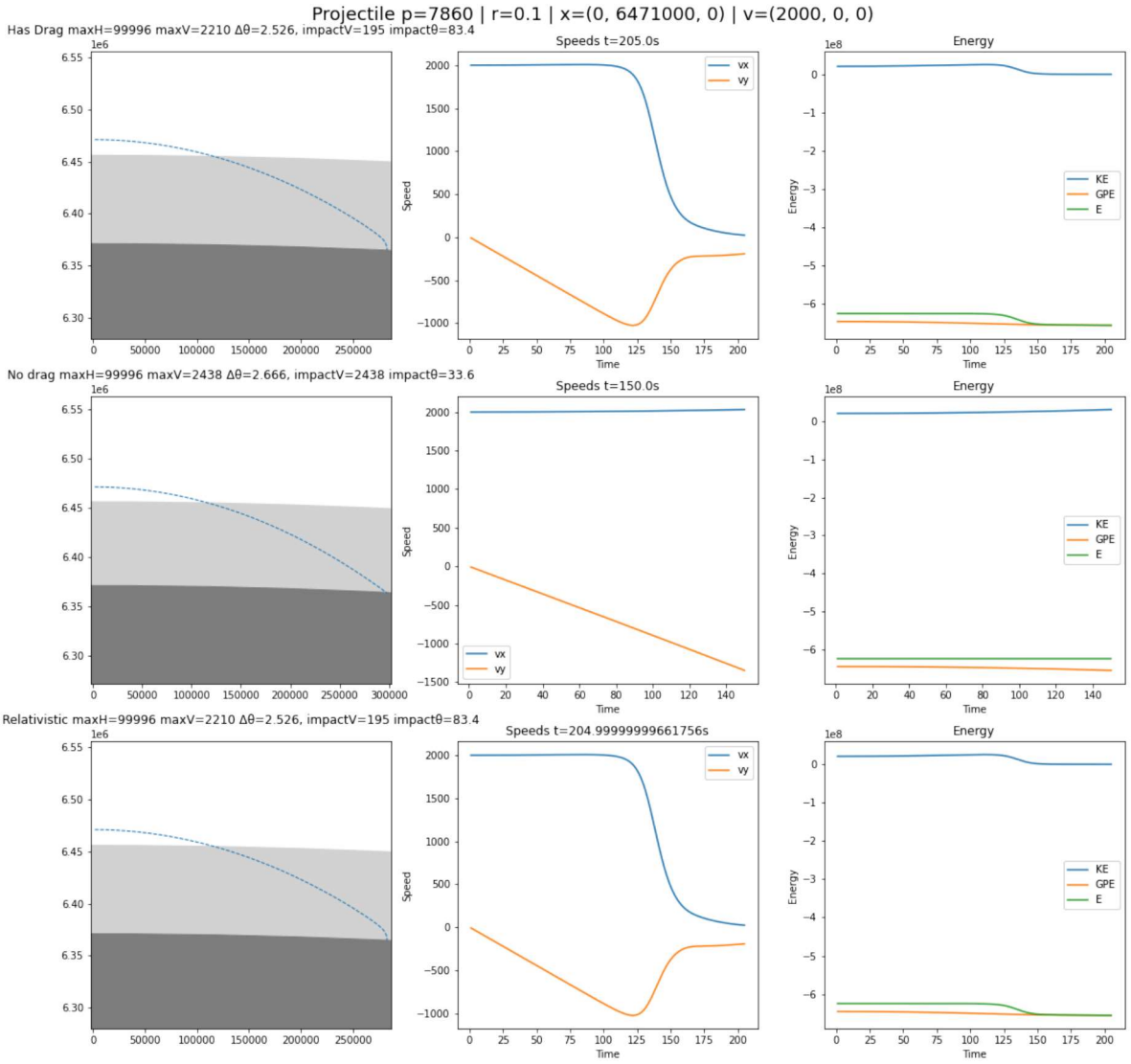
Not sure of the reason, but interesting find.

Angle of attack

The angle of attack is the angle at which our projectile hits the atmosphere.

We define the absolute edge of space at $h = 100,000m$.

```
plot_three_cases(7860, 0.1, Vector(0, R_E+100_000, 0), Vector(2000, 0, 0), dt=1)
```




```
  0%|          | 204/1000000 [00:00<06:04, 2746.02it/s]
  0%|          | 149/1000000 [00:00<02:30, 6656.92it/s]
  0%|          | 204/1000000 [00:00<06:32, 2544.91it/s]
```

```
angles = np.linspace(0, 90, 50)
def plot_diff_angle(obj_density, obj_radius, x, v, dt=1.0):
    world = curve_earth(obj_density, obj_radius)
    A = np.linspace(0, 90, 100)
    path = f"graphs/plot_diff_angle{obj_density}-{obj_radius}-{v}-{x}.npy"
    if exists(path):
        D = np.load(path)
    else:
        D = np.array([list(get_stat(np.array(world(x, Vector(cos(radians(-a)), sin(radians(-
a)), 0) * v, dt, silence=True)))) for a in tqdm(A)])
        np.save(path, D)
    maxH, maxV, delA, impV, impA = np.split(D, 5, 1)

    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)
    fig.subplots_adjust(wspace=0, hspace=0)
    fig.suptitle(f"Varying attack angles, p={obj_density} r={obj_radius}, v={v} x=({x})", fontsize=18)
    ax1.plot(A, delA)
    ax1.set_title("Delta θ (Range)", pad=-14)

    ax2.plot(A, impA)
    ax2.set_title("Impact θ", pad=-14)
    ax2.axvline(A[np.argmin(impA)], ls='--', color='black', label=round(A[np.argmin(impA)], 2))
    ax2.legend()

    ax3.plot(A, impV)
```

```
    ax3.set_title("Impact v", y=1, pad=-14)
    ax3.axvline(A[np.argmin(impV)], ls='--', color='black', label=round(A[np.argmin(impV)], 2))
    ax3.legend()
    fig.supxlabel('Angle of Attack')
    plt.show()
```

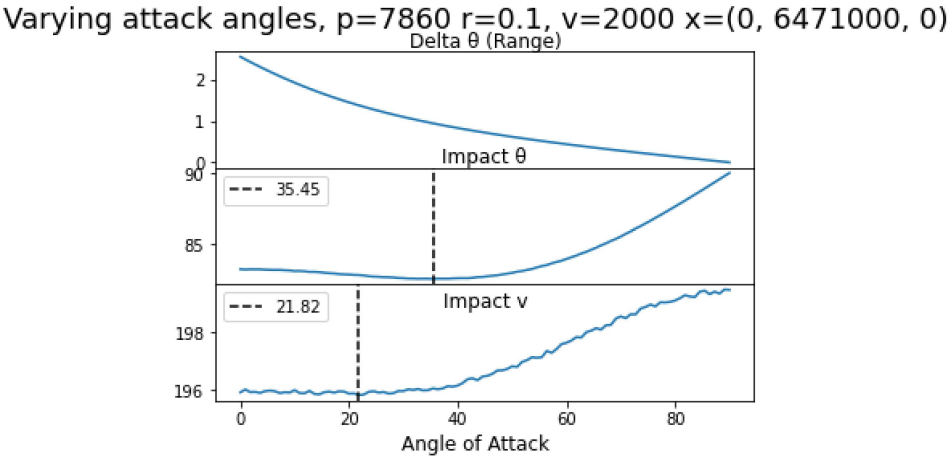
Light, small, low-speed

We can observe that the object impacts the ground at about 90°.

The decrease rate of final range decreases as the angle of attack increases.

While the impact speed increases with the angle of attack.

```
plot_diff_angle(7860, 0.1, Vector(0, R_E+100_000, 0), 2000, dt=0.1)
```

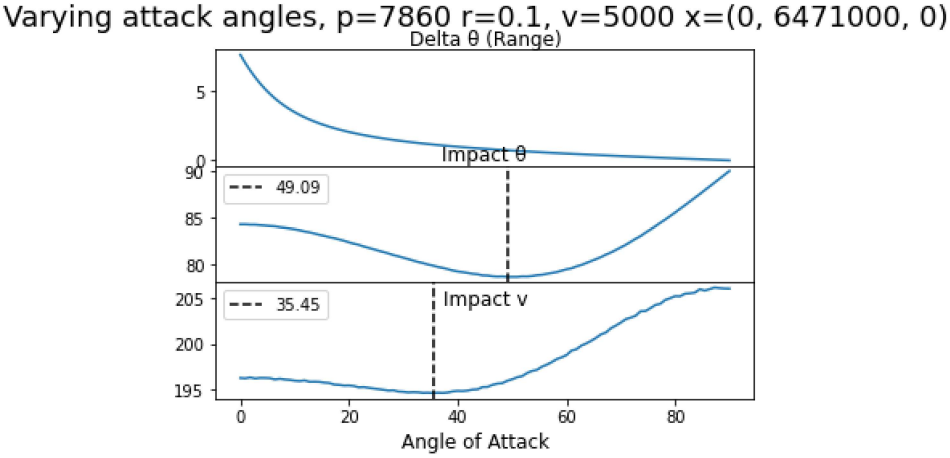


Light, small, high-speed

About the same as low-speed, but we can see a local minimum for the impact angle at an angle of attack 49°.

We can also see a lowest impact speed at an angle of attack 35°.

```
plot_diff_angle(7860, 0.1, Vector(0, R_E+100_000, 0), 5000, dt=0.1)
```

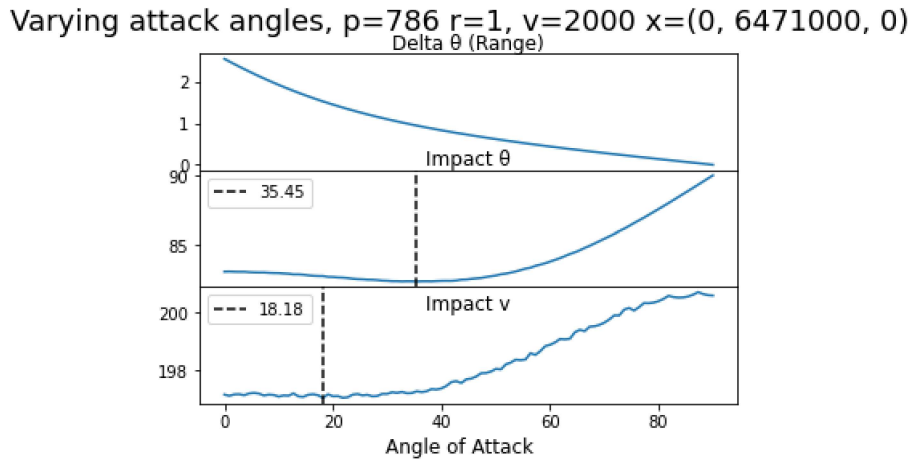


Light, big, low-speed

We can see a local minimum for the impact angle at an angle of attack 35°.

Like the previous two, the final impact speed increases with angle of attack.

```
plot_diff_angle(786, 1, Vector(0, R_E+100_000, 0), 2000, dt=0.1)
```

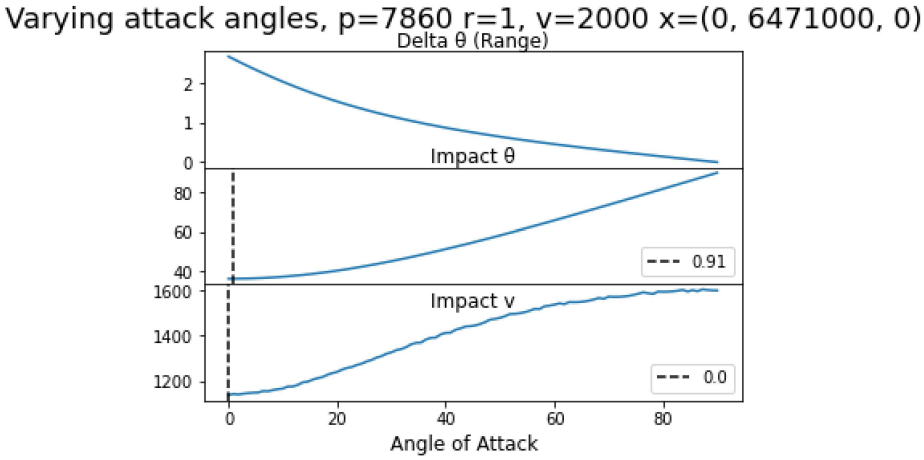


Heavy, big, low-speed

There are no local minima for the impact angle; it appears that a lower angle of attack means a lower impact angle.

Like the previous examples, the final impact speed increases with the angle of attack.

```
plot_diff_angle(7860, 1, Vector(0, R_E+100_000, 0), 2000, dt=0.1)
```

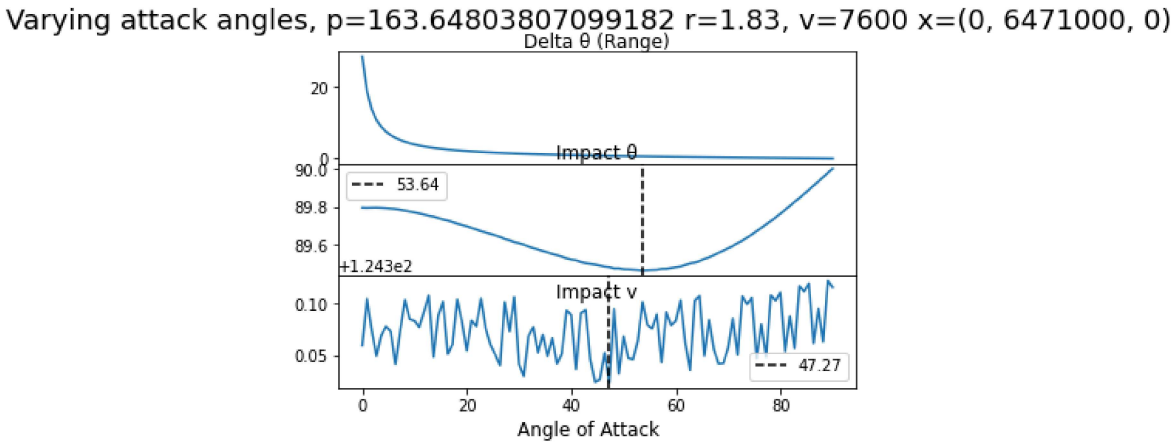


Simplified dragon capsule

This is not at all representative of the real dragon capsule, but it has the size weight and about the same size.

As we can see the impact speed is about $124m/s$, it is very stable and deadly, but should be slow enough for parachutes.

```
plot_diff_angle(4201/(4/3*pi*1.83**3), 1.83, Vector(0, R_E+100_000, 0), 7600, dt=0.1)
```



Evaluation

For less dense objects, there exists a minimum for the impact angle.

And for fast moving objects, there exists a minimum for the impact velocity.

However, for most objects, a larger angle of attack means a larger impact angle and a larger impact velocity.

This is interesting as it suggests that $\theta = 0$ may not be the optimal reentry angle for faster objects.

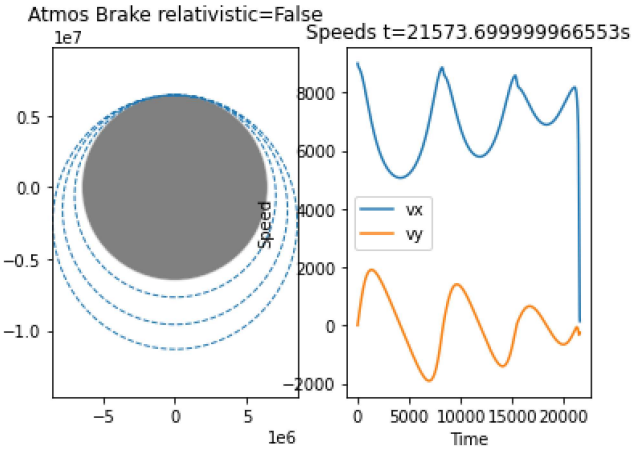
And $\theta = 0$ could actually result in a higher angle of impact than larger angles of attack for less dense objects.

High aphelion and low perihelion

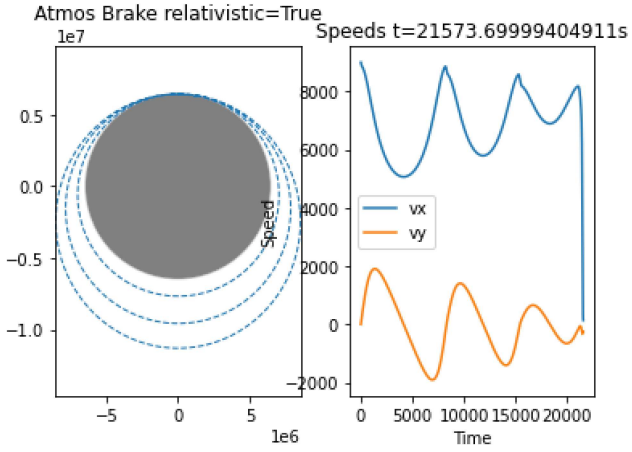
If a spacecraft has an high aphelion and a low perihelion that just touches the atmosphere, the spacecraft could make use of drag to slow itself down, hence lowering its aphelion.

```
def sim_atmos_brake(density, radius, v, relativistic=False, dt=0.1, h=65_000, silence=False):
    world = curve_earth(density, radius, relativistic=relativistic)
    path = f"graphs/sim_atmos_brake{density}-{radius}-{v}-{relativistic}-{dt}-{h}.npy"
    if exists(path):
        data = np.load(path)
    else:
        data = np.array(world(Vector(0, R_E + h, 0), Vector(v, 0, 0), dt, max_height=8 * R_E, silence=silence
))
        np.save(path, data)
    fig, (ax1, ax2) = plt.subplots(1, 2)
    plot_earth(ax1, data, title=f"Atmos Brake relativistic={relativistic}")
    plot_speeds(ax2, data)
    fig.show()
```

```
sim_atmos_brake(5000, 0.5, 9000)
```




```
sim_atmos_brake(5000, 0.5, 9000, relativistic=True)
```



Intercontinental Cannon Ball (ICCB)

If a cannon ball is shot from a position on earth, what should be the velocity to hit a location pre-specified?

```
def calc_diff(world, v, target, dt=0.1, ax=None, silence=True, den=7860, rad=0.23, rel=False):
    if not silence:
        print(f"attempt v={round(v, 2)}")
    path = f"graphs/iccb_diff/{den}-{rad}-{v}-{dt}{'r' if rel else ''}.npy"

    if exists(path):
        data = np.load(path)
    else:
        data = np.array(world(Vector(0, R_E, 0), v * Vector(cos(pi / 4), sin(pi / 4), 0), dt=dt, silence=sile
nce, max_height=6 * R_E))
        np.save(path, data)

    maxH, maxV, delA, impV, impA = get_stat(data)
    if ax is not None:
        plot_earth(ax, data, title=f"Attempt v={round(v)} Δθ={target - delA}")
    return target - delA
```

ITP Method

Short for *Interpolate Truncate and Project*, is the first root-finding algorithm that achieves the super-linear convergence of the secant method while retaining the optimal worst-case performance of the bisection method.

https://en.wikipedia.org/wiki/ITP_method

```
"""
ITP Implementation in Python based on Wikipedia pseudo-code
@author Yun
"""


def itp(f, a: float, b: float, epsilon: float, n0: float, k1=None, k2=2, max_it=20):
    if k1 is None: k1 = 0.2 / (b - a)
    n12 = ceil(log2((b - a) / (2 * epsilon)))
    nmax = n12 + n0
    j = 0

    ya = f(a)
    yb = f(b)

    it = 0

    while b - a > 2 * epsilon:
        # calc params
        x12 = (a + b) / 2
        r = epsilon * 2 ** (nmax - j) - (b - a) / 2
        delta = k1 * (b - a) ** k2

        # interpolation
        xf = (yb * a - ya * b) / (yb - ya)
        # truncation
        sigma = sign(x12 - xf)
        if delta <= abs(x12 - xf):
            xt = xf + sigma * delta
        else:
            xt = x12
        # projection
        if abs(xt - x12) <= r:
            xitp = xt
        else:
            xitp = x12 - sigma * r
```

```
        # update interval
        yitp = f(xitp)
        if yitp > 0:
            b = xitp
            yb = yitp
        elif yitp < 0:
            a = xitp
            ya = yitp
        else:
            a = xitp
            b = xitp
        it += 1
        if it > max_it:
            a = xitp
            b = xitp
            break
    print(f"total iterations {it}")
    return (a + b) / 2
```

```
itp(lambda x: x**3+2*x**2-4*x+20, -10, 10, 0.00001, 1)
```

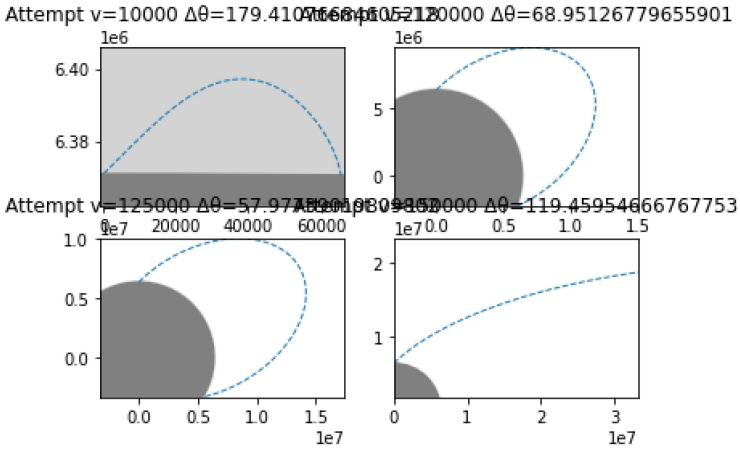
```
total iterations 11
-4.136146987933472
```

~~North Pole to South Pole~~

It seems rather unrealistic, hence we will change this to 2 shots. Hence the difference would be 90°.

```
world = curve_earth(7860, 0.23) # 460mm cannon / iron bullet
def wrapper(v, ax=None):
    return calc_diff(world, v, 180, ax=ax, dt=0.01, silence=False, den=7860, rad=0.23)
fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2)
wrapper(10000, ax1)
wrapper(120000, ax2)
wrapper(125000, ax3)
wrapper(150000, ax4)
```

```
attempt v=10000
attempt v=120000
attempt v=125000
attempt v=150000
119.45954666767753
```

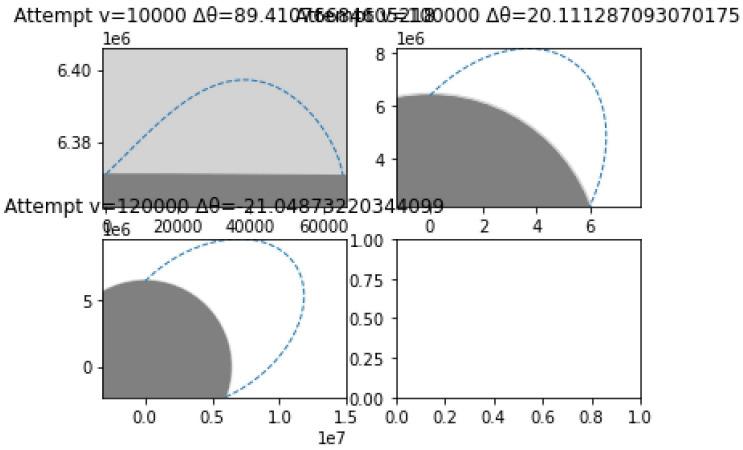


90 Degree Cannon Ball

We obtain a launching velocity of $109393.98m/s$ for a 45° cannon ball with $\rho = 7860kg/m^3$ and $r = 0.23m$.

```
def wrapper(v, ax=None):
    return calc_diff(world, v, 90, ax=ax, dt=0.01, silence=True, den=7860, rad=0.23)
fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2)
```

```
wrapper(10000, ax1)
wrapper(100000, ax2)
wrapper(120000, ax3)
-21.04873220344099
```




```
itp(wrapper, 100000, 120000, 1, 1)
total iterations 21
109393.98109347766
```

Tungsten Ball (England to France)

> The total straight line distance between London and Paris is 343 KM (kilometers) and 600 meters

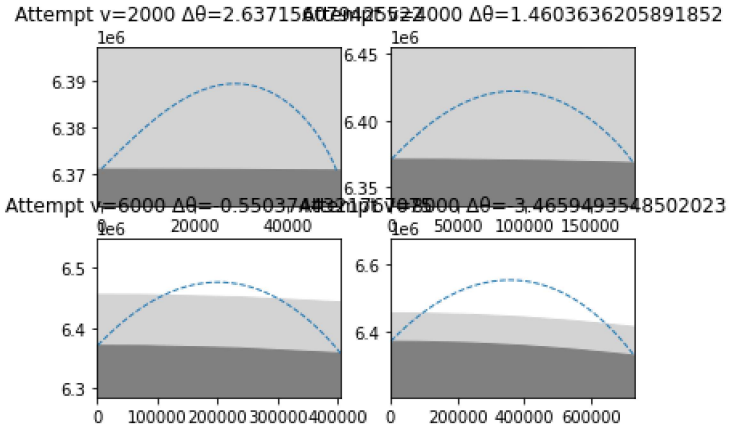
With $\frac{\theta}{360} \times 2\pi \times R_E = 343,600$, we can make use of $R_E = 6,371,000m$ and calculate θ.

$$\theta = \frac{343600 \times 360}{2\pi 6371000} = 3.090069^\circ$$

To shot a tungsten ball of radius 0.22m from London to France, we need a speed of $5368.45m/s$ for a launch angle of 45°.

```
world = curve_earth(19250, 0.22, relativistic=True) # 0.22 tungsten ball
def wrapper(v, ax=None):
    return calc_diff(world, v, 3.090069, ax=ax, dt=0.1, silence=True, den=19250, rad=0.22)
```

```
fig, ((ax1, ax2),(ax3,ax4)) = plt.subplots(2, 2)
wrapper(2000, ax1)
wrapper(4000, ax2)
wrapper(6000, ax3)
wrapper(8000, ax4)
-3.4659493548502023
```




```
itp(wrapper, 4000, 6000, 0.1, 1)
total iterations 21
```

5368.445410068342

Conclusion

By analysing mutliple examples with numerical method we have arrived at many of the aforementioned evaluated conclusions.

It appears that in the presence of atmospheric drag, relativity effects are not very significant. This is reasonale since relativistic effects come into play when speeds approahce light speed.

It also appears quite useful to model atmospheric drag with numerical method, as it could be used in many use cases.

Limitations / Assumptions

- Aerodynamics model's validity at relativistic speeds
- No weather / turbulence / clouds etc
- Limited computing power = limited accuracy

Improvements

- Other shapes than sphere
- Heat
- More accurate relativity (currenting using corrected newton model)