Self-Guidance Missile System Documentation

By **[Ashvith]**
Date [10/10/2006]

1. Project Title

Self-Guidance Missile

2. Abstract

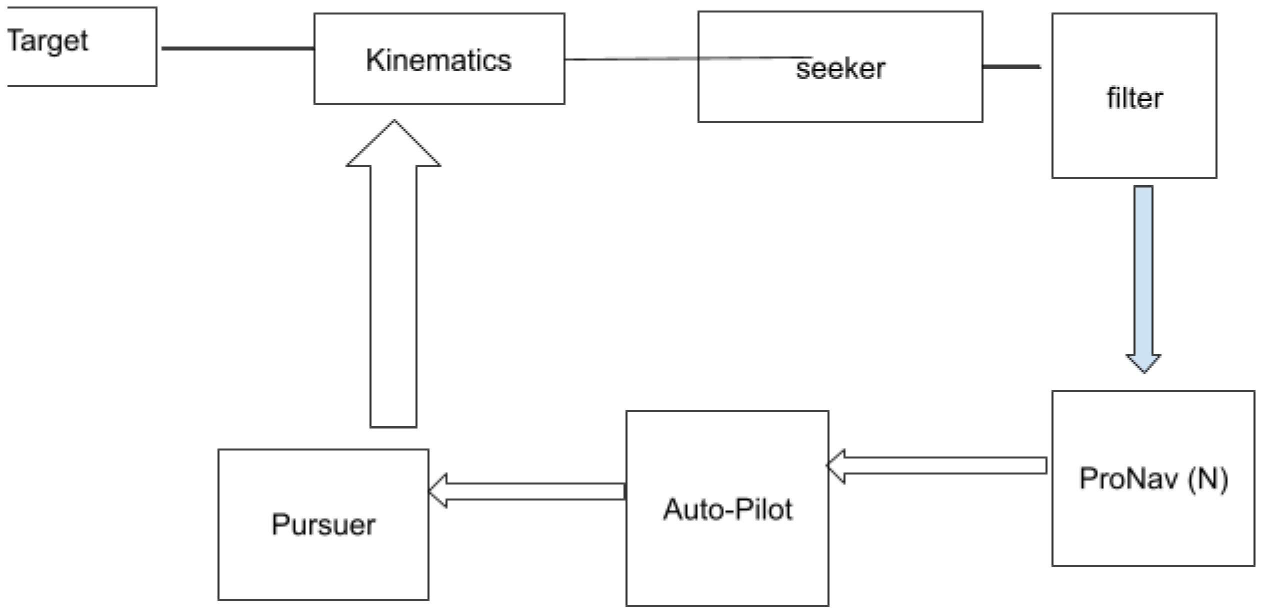
This project explores the design and implementation of a lightweight target-locking system capable of identifying, tracking, and maintaining a lock on a fixed target over long distances. Using a combination of sensor data, real-time processing, and simple guidance logic, the system predicts target movement and adjusts its tracking behavior to stay aligned. The goal of this work is not to create a weaponized platform, but to understand the engineering principles behind guidance, control, and tracking—principles commonly used in aerospace, robotics, and navigation systems. The documentation details the algorithms, hardware choices, testing process, and challenges encountered while building a functional prototype that demonstrates the fundamentals of target acquisition and lock-on behavior. This is just a still-on work project . and its goal is to make a guidance system which can be effective in hostage situations .

3. Introduction

I always wondered how missiles hit their targets precisely , when i was kid i thought " is there any hidden pigeon inside the missile to control its path". Now I know what exactly is the reason, and I want to uncover it and understand it by making a project of my own and experience it for myself .With the budget of my project being low I need to complete my project with the salvage parts from electronics to minimize the cost. This will be a challenge for me . And further I have an idea to make it more precise by detecting particular humans and areas to never let another innocent life be taken amidst a war.

4. System Overview

- Block diagram



5. Hardware Components

- ### Processing And Control Hardware

Microcontroller

- Arduino Uno / Nano / Mega — **[STORE-BOUGHT]**
  (Mobiles don't have directly usable MCUs.)

Voltage regulators / power ICs

- Step-down DC/DC regulators (buck converters) — **[can be made on my own]**
- Linear LDO regulators — **[SALVAGED** from old routers, set-top boxes, power banks]

Oscillators

- Crystal oscillators (8–32 MHz range) — **[SALVAGED]** from old phones / motherboards

---

2. Core Sensors

IMU (Accelerometer + Gyroscope)

- MPU-6050
- ICM-20948 (9-axis)

Sensors salvageable from phones:

Phones contain extremely high-quality MEMS sensors, but **they are soldered as tiny BGA chips** and cannot be reused unless you have BGA reflow equipment. But for completeness:

- Bosch BMI160 (IMU) — **[SALVAGED]** (hard to desolder)
- InvenSense MPU series — **[SALVAGED]**
- STMicro LIS3DH accelerometer — **[SALVAGED]**

GPS

- NEO-6M / NEO-M8N GPS modules — **[STORE-BOUGHT]**
  Phones have GNSS chips but they **cannot be reused or interfaced** directly → not salvageable.

Altitude Sensing

- Barometer (BMP280 / BME280 / MS5611) — **[STORE-BOUGHT]**

Boards that contain usable barometer sensors:

- Some **drones**, **smartwatches**, **fitness bands** — **[SALVAGED]**
  (very difficult to extract though)

---

3. Communication / Telemetry

Wireless modules

- HC-05 / HC-06 Bluetooth module — **[STORE-BOUGHT]**
- ESP-01 / ESP8266 / ESP32 — **[STORE-BOUGHT]**
- 433 MHz LoRa SX1278 — **[STORE-BOUGHT]**
- nRF24L01+ -[STORE-BOUGHT]

Salvageable components from devices:

- WiFi antennas — **[SALVAGED]** from routers/phones
- 2.4 GHz RF front-end chips — **[SALVAGED]** (high difficulty)

---

4. Power System

Batteries

- Li-ion 18650 cells — **[SALVAGED]** from laptops / power banks
- Li-Poly pouch cells — **[SALVAGED]** from phones/tablets
  (for the reader and future aspirants , the above should be done only with better experience and)

Battery management / charging

- TP4056 charger module — **[STORE-BOUGHT]**
- BMS boards — **[STORE-BOUGHT]**

Power converters

- DC-DC buck converters (LM2596 etc.) — **[STORE-BOUGHT]**
- Small inductors + switching ICs — **[SALVAGED]** from computer motherboard VRMs

---

5. Logging & Storage

SD card module

- SD card breakout (CS, MOSI, MISO, SCK pins) — **[STORE-BOUGHT]**
- microSD cards — **[SALVAGED]** from old phones, cameras

EEPROM

- 24Cxx series EEPROM chips — **[STORE-BOUGHT]**
- EEPROM chips — **[SALVAGED]** from routers, set-top boxes

---

6. Structural & Support Electronics

Connectors

- JST connectors — **[STORE-BOUGHT]**

- Micro USB/Type-C ports — **[SALVAGED]** from phones/power banks
- FFC/FPC connectors — **[SALVAGED]** from laptops

Switches / Buttons

- Momentary pushbuttons — **[STORE-BOUGHT]**
- Side-buttons (volume/power) — **[SALVAGED]** from phones

Wires

- Silicone flexible wires — **[STORE-BOUGHT]**
- Copper ribbon cable — **[SALVAGED]** from printers and DVD players

LED indicators

- SMD/through-hole LEDs — **[STORE-BOUGHT]**
- RGB / notification LEDs — **[SALVAGED]** from phones

Filters & Analog components

- Capacitors, inductors, resistors — **[SALVAGED]** from motherboards, TV PCBs, chargers

---

- Heat sinks — **[SALVAGED]** (desktop motherboard VRMs)
- Cooling fans — **[SALVAGED]** (laptops, PCs)
-
- Servo motors - to control the fins through arduino

6. Software Components

- OpenCV detection pipeline
- Communication protocol
- Arduino firmware logic

7. Detection Algorithm

```
function detectTarget(sensor_data):
Preprocessing the data that we acquired from the sensor

    clean_data = preprocess(sensor_data)

    features = extractFeatures(clean_data)



    // 3. Generate values
```

```
candidates = detectCandidates(features)


// 4. Validate each values
for c in candidates:
    if isValid(c):
        confidence = computeConfidence(c)
        if confidence > CONF_THRESH:
            return {
                is_detected: True,
                raw_position: c.position,
                confidence: confidence,
                timestamp: now()
            }


// 5. If no values pass validation
return {
    is_detected: False,
    raw_position: None,

    confidence: 0,

    timestamp: now()

}
```

- Frame processing

Still on work

- Target locking Algorithm

1. Target Detection / Initialization

1. Read the target's coordinates **($x_{□}$, $y_{□}$, $z_{□}$)** from the sensor module or mission input.
2. Record the rocket's initial position **(x_0, y_0, z_0)**.

   Compute the initial line-of-sight (LOS) vector:

   ```
   LOS = target_position - rocket_position
   ```

3.
4. Normalize LOS to get the initial guidance direction.

---

2. Real-Time State Update

During each simulation tick or control loop:

Read the rocket's position and velocity from the IMU/GPS fusion:

```
rocket_pos = (x, y, z)
```

```
rocket_vel = (vx, vy, vz)
```

1.

   Recompute the line-of-sight vector:

   ```
   LOS = target_pos - rocket_pos
   ```

2.
3. If the target is moving, update target_pos with its predicted next position.

---

3. Lock and Maintain LOS Angle

To keep the target “locked,” measure how the LOS angle changes:

```
LOS_unit = LOS / ||LOS||

rocket_heading_unit = rocket_vel / ||rocket_vel||

angle_error = angle_between(rocket_heading_unit, LOS_unit)
```

If **angle_error** stays below a certain threshold (e.g., < 2°),
 the system considers **target locked**.

---

4. Steering Correction

If the rocket deviates from pointing at the target:

   Compute the direction you want to correct toward:

   ```
   correction_vector = LOS_unit - rocket_heading_unit
   ```

1.

   Normalize it:

   ```
   steer_dir = correction_vector / ||correction_vector||
   ```

2.

3. Use `steer_dir` to update the thrust vector or fin positions.

This step gently nudges the rocket’s direction toward the LOS without over-steering.

---

5. Proportional Navigation (PN) – Optional

```
steering_rate = N * (rate_of_change_of_LOS_angle)
```

Where `N` is the navigation constant (usually 3–5).

This naturally drives the rocket to lead the target and reduce miss distance.

---

6. Stability + Filtering

Smoothing the LOS estimates using a low-pass filter:

```
filtered_LOS = α * previous_LOS + (1 - α) * current_LOS
```

1.
2. Apply max steering limits to avoid violent oscillations.
3. Ignore noise spikes from sensors by clamping sudden jumps.

8. Guidance Algorithm

```
BEGIN Loop
  target_state ← updateTargetKinematics()

  measurement ← seekerMeasure(target_state, pursuer_state)
```

```
    estimate ← kalmanFilter(measurement)

    guidance_cmd ← proportionalNavigation(estimate)
    control_input ← autopilotControl(guidance_cmd)
    pursuer_state ← updatePursuerDynamics(control_input)
    pursuer_state ← updatePursuerKinematics(pursuer_state)

END Loop
```

I modified my algorithm to pursue a fixed target

Modified algorithm

```
function target(pursuer_state, target_position):

    //
    pursuer_position = pursuer_state.position
    self_heading = pursuer_state.heading

    // compute the vector towards the target
    direction = target_position - pursuer_position

    // to get the distance between the pursuer and the target
    distance = norm(direction)
    desired_heading = angle(direction)
    heading_error = desired_heading - self_heading

    // to generate commands to autopilot our system
    cmd = autopilot(heading_error, distance)

    // adjusting the pursuer dynamics according to the data we acquired
    pursuer_state = updatePursuer(cmd)
```

```
  return pursuer_state
```

- PID or simple proportional control
- Servo actuation

9. Communication Between Laptop and Missile

- Serial/WiFi communication

Message Structure:

```
{
  "header": {
    "msg_id": 1,
    "protocol_version": 1.0,
    "timestamp_ms": 1733051000,
    "source_id": 100,
    "destination_id": 1
  },

  "navigation": {
    "target_lat": 11.923456,
    "target_lon": 79.820123,
    "target_alt": 100.0,
    "approach_mode": "DIRECT",
    "velocity_command": 20.0,
    "tolerance_radius": 5.0
  },


  "health": {
    "imu_status": "OK",
    "gps_status": "OK",
    "battery_level": 92,
    "cpu_temperature": 46.2,
    "failsafe_flags": 0
  },


  "checksum": 3948572938
}
```

10. Mechanical Design

Still on work

11. Testing and Validation

The code i used to simulate model

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # enables 3D plotting

# ---------- Physical constants ----------
g = 9.80665       # m/s^2
rho = 1.225       # kg/m^3 (sea level)
```

```
# ---------- Rocket / scenario parameters ----------
Cd = 0.5          # drag coefficient
A = 0.03          # cross-sectional area (m^2)
burn_time = 5.0   # seconds
T0 = 200.0        # thrust magnitude (N)
m0 = 5.0          # initial mass (kg)
prop_mass = 1.0   # propellant mass (kg)

# ---------- Target and launch geometry ----------
target_distance = 1300.0   # horizontal distance to target (m)
elevation_deg = 45.0       # launch elevation angle (degrees)
azimuth_deg = 0.0          # launch azimuth (degrees) - 0 = +x direction

# Derived
elev = np.deg2rad(elevation_deg)
azi = np.deg2rad(azimuth_deg)
# Target coordinates on ground (x, y, z=0)
target = np.array([target_distance * np.cos(azi),
                   target_distance * np.sin(azi),
                   0.0])

# ---------- Thrust / mass functions ----------
def thrust(t):
    """Simple constant-thrust burn; replace with curve if you have one."""
    return T0 if 0 <= t <= burn_time else 0.0

def mass(t):
    """Linear mass depletion during burn."""
    if t <= burn_time:
        return m0 - prop_mass * (t / burn_time)
    return m0 - prop_mass

# ---------- Thrust direction (fixed initial direction) ----------
u_thrust = np.array([np.cos(elev) * np.cos(azi),
                     np.cos(elev) * np.sin(azi),
                     np.sin(elev)])  # unit vector

# ---------- Dynamics ----------
def dynamics(t, y):
    # State y = [x, y, z, vx, vy, vz]
    x, y_pos, z, vx, vy, vz = y
    m = mass(t)
    v = np.array([vx, vy, vz])
    speed = np.linalg.norm(v) + 1e-9

    # Quadratic drag (vector form)
    F_drag = -0.5 * rho * Cd * A * speed * v

    # Thrust (applies only during burn)
    T = thrust(t)
    F_thrust = T * u_thrust

    # Gravity
    F_grav = np.array([0.0, 0.0, -m * g])

    # Acceleration
    a = (F_thrust + F_drag + F_grav) / m

    return [vx, vy, vz, a[0], a[1], a[2]]

# ---------- Integration and impact detection ----------
# Initial state: at origin, zero velocity
y0 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# Event to stop integration at impact (z crosses 0 downward)
def hit_event(t, y):
    return y[2]
hit_event.terminal = True
hit_event.direction = -1

t_span = (0, 300)  # max sim time (s)
t_eval = np.linspace(0, t_span[1], 8000)

sol = solve_ivp(dynamics, t_span, y0, events=hit_event, t_eval=t_eval, rtol=1e-6, atol=1e-8)

# Determine impact state
if sol.t_events and len(sol.t_events[0]) > 0:
    t_impact = sol.t_events[0][0]
    state_impact = sol.y_events[0][0]
    x_imp, y_imp, z_imp = state_impact[0], state_impact[1], state_impact[2]
else:
    # use final step if event didn't trigger
    t_impact = sol.t[-1]
    x_imp, y_imp, z_imp = sol.y[0,-1], sol.y[1,-1], sol.y[2,-1]

impact = np.array([x_imp, y_imp, z_imp])
miss_distance = np.linalg.norm(impact - target)
```

```
print(f"Impact time: {t_impact:.3f} s")
print(f"Impact coordinates (x, y, z): ({x_imp:.2f}, {y_imp:.2f}, {z_imp:.2f}) m")
print(f"Target coordinates (x, y, z): ({target[0]:.2f}, {target[1]:.2f}, {target[2]:.2f}) m")
print(f"Miss distance: {miss_distance:.2f} m")

# ---------- Plotting ----------
xs = sol.y[0]
ys = sol.y[1]
zs = sol.y[2]

fig = plt.figure(figsize=(12,6))

# 3D trajectory
ax = fig.add_subplot(121, projection='3d')
ax.plot(xs, ys, zs)
ax.scatter([target[0]], [target[1]], [target[2]], marker='o')  # target
ax.scatter([impact[0]], [impact[1]], [impact[2]], marker='x')  # impact
ax.set_xlabel('X (m)')
ax.set_ylabel('Y (m)')
ax.set_zlabel('Z (m)')
ax.set_title('3D Trajectory (target marked)')

# Plan (ground) view
ax2 = fig.add_subplot(122)
ax2.plot(xs, ys)
ax2.scatter([target[0]], [target[1]], marker='o')
ax2.scatter([impact[0]], [impact[1]], marker='x')
ax2.set_xlabel('X (m)')
ax2.set_ylabel('Y (m)')
ax2.set_title('Ground Track (plan view)')
ax2.axis('equal')

plt.tight_layout()
plt.show()
```

Controlling Fins through Arduino :

normalize and clamp to gimbal/fin limits to produce unit thrust direction

Switch to terminal when horizontal distance < `R_switch` (e.g., 100–300 m) or when predicted impact radius small.

Terminal descent :

Goal: near-zero horizontal velocity at target and controlled vertical descent.

Algorithm I used :

Wind model

Constant wind : $w = [w_x, w_y, 0]$ (m/s)

Gust Model : $w(t) = w_0 + \sum_i A_i \sin(w_i t + \phi_i)$

Spatially varying : $w(x, y, z)$

Drag with wind

$$F_{drag} = -\frac{1}{2}\rho C_d A \|v - w\|(v - w)$$

Clamp a_{des} ($\|a_{des}\| \leq a_{max}$)

commanding the thrust vector

$$u_{raw} = \frac{m a_{des} + mg}{T}$$

Terminal Descent

Goal : near - zero horizontal velocity at target and Controlled vertical descent.

1) Compute horizontal distance

$r_{\perp} = [r_x, r_y, 0]$ and $v_{\perp}$ hori veloci

2) Desired horizontal acceleration

$$a_h = K_{ph} r_{\perp} - K_{dh} v_{\perp}$$

Desired vertical accel:

$v_{des} = -2\ m/s$

vertical accel.

$a_z = K_{pv}(z_{target} - z) - K_{dv}(v_z - v_{des,z})$

Combine $a_{des} = [a_n, a_z]$ and

Compute thrust.

```
function guidanceLoop(t, state, target, parameters):

    # state: r (3), v (3), m
    r = state.position
    v = state.velocity
    m = state.mass

    # 1) compute air-relative velocity (for future use)
    v_air = v - wind(t, r)

    # 2) distance to target
    r_err = target - r
    horiz_dist = norm(r_err[0:2])

    # 3) choose phase
    if horiz_dist > R_switch:
        # MIDCOURSE
        a_des = k_p * r_err - k_d * v
        # optionally reduce vertical aggressiveness:
        a_des[2] = clip(a_des[2], a_z_min, a_z_max)
    else:
        # TERMINAL
    # horizontal control
    r_h = [r_err[0], r_err[1], 0]
    v_h = [v[0], v[1], 0]
    a_h = k_ph * r_h - k_dh * v_h

    # vertical control: go to small descent rate
    a_v = k_pv*(z_target - r[2]) - k_dv*(v[2] - v_descent)

    a_des = [a_h[0], a_h[1], a_v]

# 4) clamp desired acceleration magnitude
if norm(a_des) > a_max:
    a_des = a_des * (a_max / norm(a_des))

# 5) compute required thrust direction:
# T = thrust(t)   (current thrust magnitude)
u_raw = (m * a_des + m * [0,0,-g]) / max(thrust(t), tiny)
u_des = normalize(u_raw)

# 6) limit tilt angle
if angle_between(u_des, z_axis) > theta_max:
    u_des = rotate_toward(z_axis, u_des, theta_max)

# 7) send u_des to attitude controller (PID -> fins/gimbal)
```

```
    send_to_attitude_controller(u_des)


    return u_des
```

12. Limitations

This is just a working model which is to be further developed in the future , which can only hit a fixed target which is 1.3 Km away

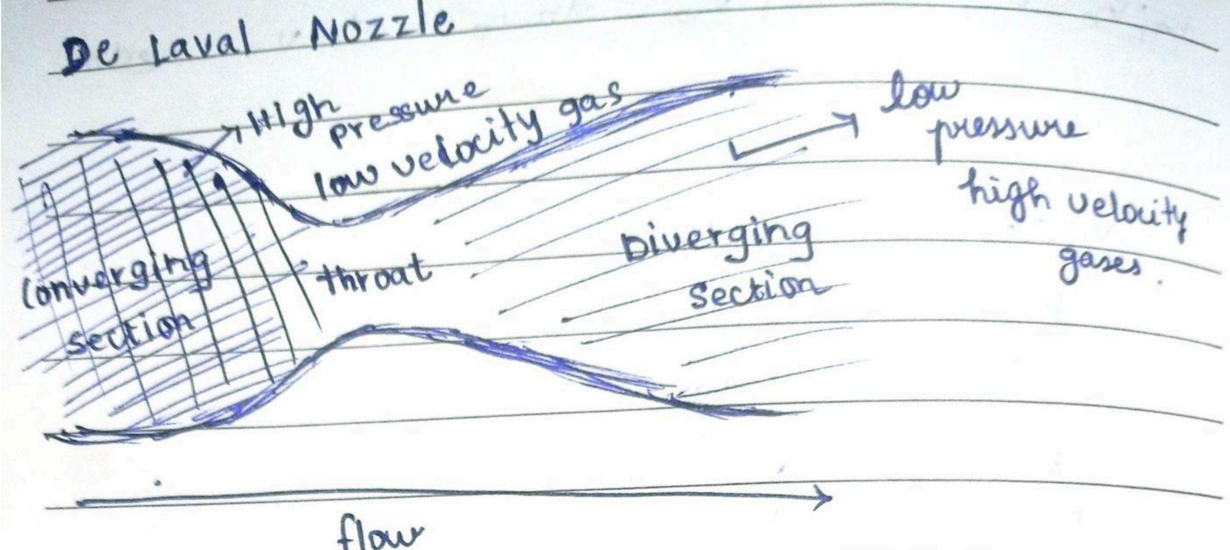
13. Future Improvements

I just have this blue-print for this project , further ill get my parts and I want to make it functional and test it. And further I have ideas to contribute to the Military by inventing new models of guidance systems. For example, I have an idea to develop a guidance system in-case of hostage systems and to prevent casualties in the war . It became my aim to invent guidance systems with full precision like that after I witnessed innocents being killed at war. And i pledge that it will be developed in that way .

Sketches :

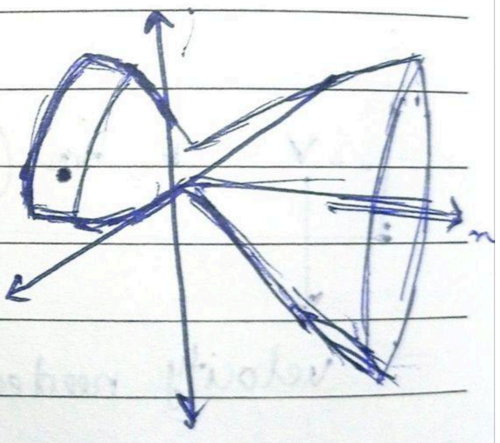
DATE / /

De Laval Nozzle



Mass flow rate

$\boxed{dV = AVdt}$ → take a small element of gas



$dm = \rho dV$

$dm = \rho AVdt$

$\frac{dm}{dt} = \rho AV \Rightarrow \dot{m} = \rho AV$

(or)

$\rho \rightarrow$ density

$A \rightarrow$ Area of c·sect

$V \rightarrow$ velocity of gas

m depends on 3 variable, which will change w.r.t time

$m = f(\rho, v, A)$ $\frac{\quad}{\rho}$ DATE / /

~~dm~~ $\frac{d\dot{m}}{dt} = \frac{\partial \dot{m}}{\partial \rho} \times \frac{d\rho}{dt} + \frac{\partial \dot{m}}{\partial v} \times \frac{dv}{dt} + \frac{\partial \dot{m}}{\partial A} \times \frac{dA}{dt}$

Solving these

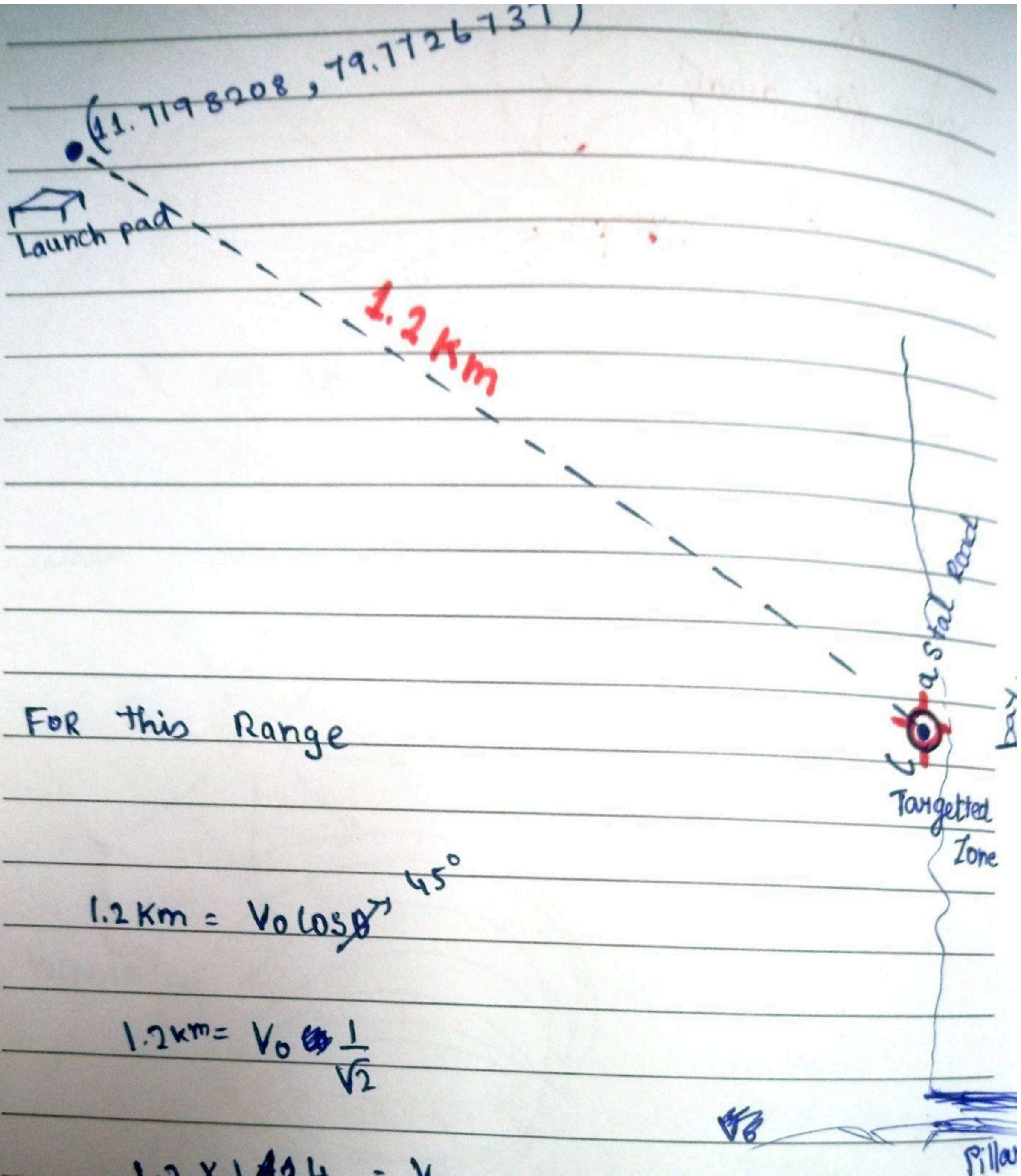
$$0 = \frac{d\rho}{dt} \cdot Av + \frac{dv}{dt} \rho \cdot A + \frac{dA}{dt} \times \rho v$$

$$0 = d\rho \cdot A \cdot v + dv \cdot \rho \cdot A + dA \cdot \rho \cdot v$$

$$\boxed{0 = \frac{d\rho}{\rho} + \frac{dv}{v} + \frac{dA}{A}} \longrightarrow (1)$$

↳ what does this ~~tell~~ says

how am I gonna constructs my nozzle ?



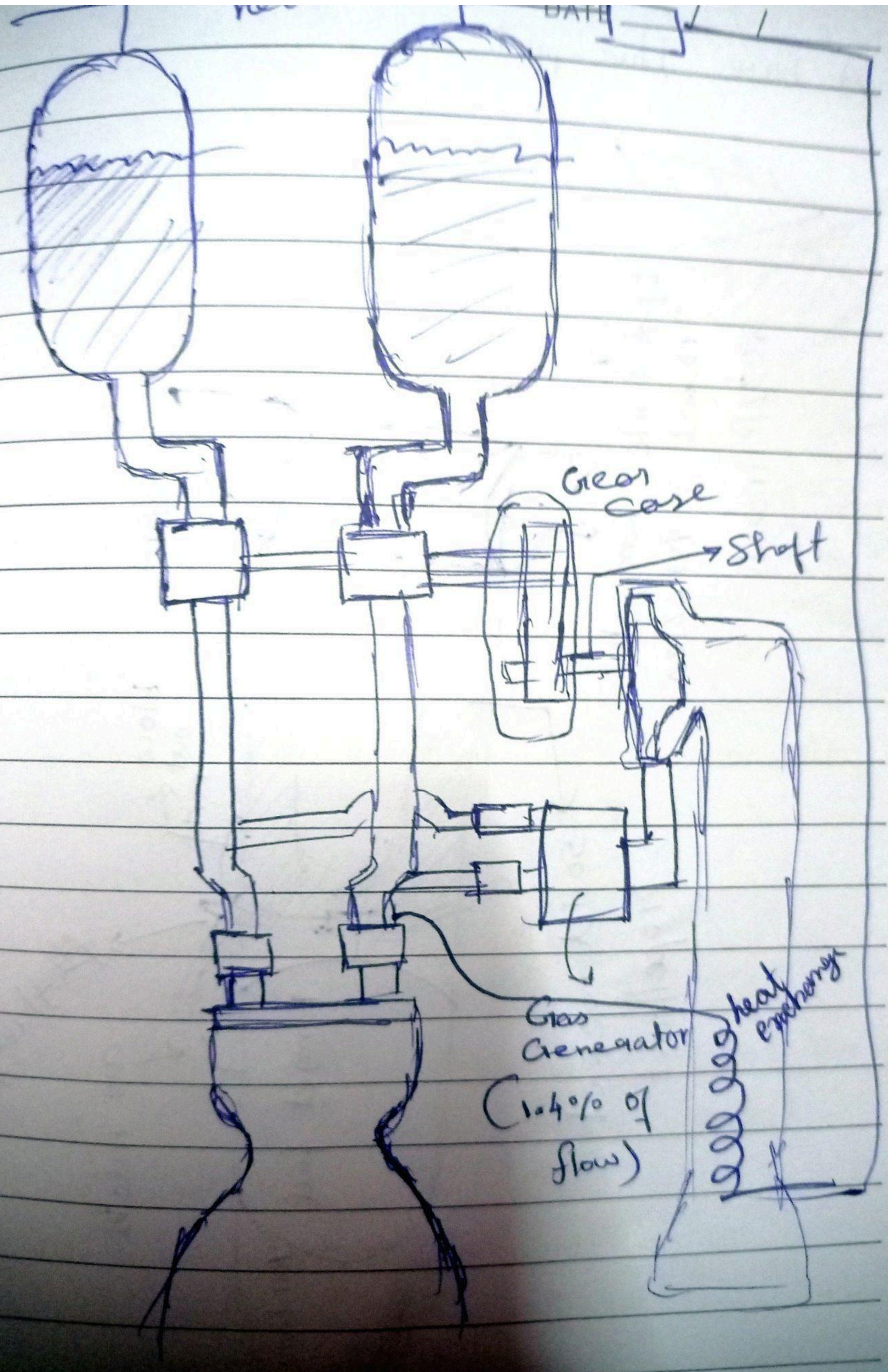
For this Range

$$1.2\,\text{Km} = V_0 \cos \cancel{\theta} \rightarrow 45^\circ$$

$$1.2\,\text{km} = V_0 \frac{1}{\sqrt{2}}$$

$$1.2 \times 1.414 = V_0$$

Required $\Delta V = 1.697056275$ km/hr


Gear case
Shaft
Gas Generator (1.4% of flow)
heat exchanger