



# Relazione Civitanavi Systems S.P.A

Oggetto: Primary Flight Display

31 Maggio 2024

## Autori:

Diego Ciucaloni

Matteo Fabbioni

Luca Niccià

Vittorio Piotti

Indice	
1. Introduzione a. Scopo generale b. I nostri obiettivi da raggiungere c. Dove siamo arrivati d. Architettura del programma	x
2. Analisi del file 'civitanavi_server.py'	x
3. Analisi del file 'civitanavi_client.py'	x
4. Analisi del file 'display.py' a. Analisi del file 'Calc.py' b. Analisi del file 'Horizon.py' c. Analisi del file 'PitchLadder.py' d. Analisi del file 'Viewfinder.py'	x
5. Testing	x
6. Linee guida	x
7. Collegamento a gitHub	

## 1) Introduzione

### a) Scopo generale

In collaborazione con l'azienda Civitanavi Systems, l'istituto Montani di Fermo si è focalizzato nella costruzione di un velivolo ultraleggero partendo dai sensori e dalla strumentazione di bordo.

Il lavoro è stato suddiviso tra i diversi indirizzi della scuola:

Indirizzo	Occupazione
Conduzione del mezzo aereo	Preparazione della struttura dell'aereo con il motore
Telecomunicazioni	Installazione dei sensori e ricezione dei dati
Digital Strategist	Realizzazione del Primary Flight Display

Il progetto si è svolto sotto la supervisione dei docenti della scuola incaricati e dei responsabili dell'azienda Civitanavi tramite meeting online con gli studenti delegati.

### b) I nostri obiettivi da raggiungere

Gli obiettivi prefissati inizialmente:

- Creazione dello schermo diviso tra cielo, orizzonte e terra;
- Movimento dello schermo in funzione del pitch;
- Movimento dello schermo in funzione del roll;
- Creazione delle linee di rappresentazione del pitch (in base ai gradi);
- Creazione delle linee di rappresentazione del roll (in base ai gradi);
- Creazione del mirino con le ali per rappresentare l'aereo;
- Creazione di una schermo di rappresentanza dello yaw.

### c) Dove siamo arrivati

In conclusione del progetto che è stato portato avanti, sono stati raggiunti tali obiettivi:

- Creazione dello schermo diviso tra cielo, orizzonte e terra;
- Movimento dello schermo in funzione del pitch;
- Movimento dello schermo in funzione del roll;
- Creazione delle linee di rappresentazione del pitch (in base ai gradi);
- Creazione del mirino con le ali per rappresentare l'aereo.

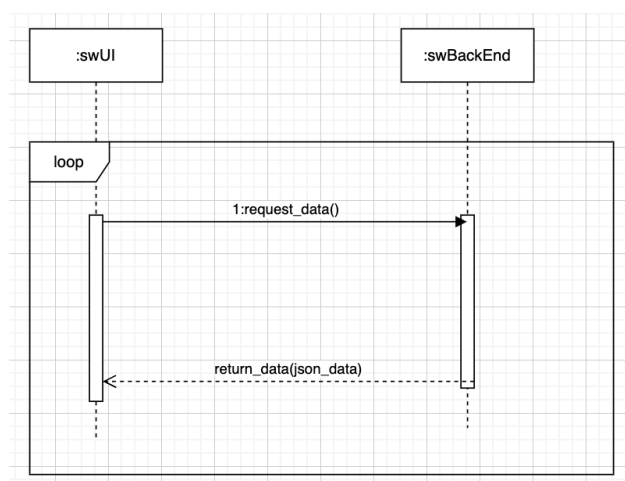
### d) Architettura del programma

L'architettura utilizzata per l'ideazione del progetto propone una parte server che invia i dati ogni 100 millisecondi presi dai sensori al client. I dati verranno mandati con il formato json nel quale il client estrapola i dati che gli serviranno per la progettazione del software.

I ragazzi dell'indirizzo di telecomunicazioni ci passeranno tramite i sensori i seguenti dati:

- pitch (in gradi);
- roll (in gradi);
- yaw (in gradi);
- pressione (in mBar)
- accelerazioni sui tre assi (espresso in g)

In questo progetto per adesso sono stati utilizzati solamente i primi due dati.



## 2) Analisi del file 'civitanavi\_server.py'

Il file 'civitanavi\_server.py' implementa un server che:

- Ascolta le connessioni in entrata su una porta specifica.
- Accetta una connessione e comunica con il client.
- Genera e invia continuamente dati casuali rappresentanti angoli di pitch, roll e yaw.
- Gestisce l'interruzione dell'esecuzione in modo pulito chiudendo i socket.

Andiamo ad esaminare nel dettaglio i componenti del file.

### Importazioni

```
import time
import socket
import json
import random
```

- **time**: per la gestione delle attese temporali.
- **socket**: per la creazione e gestione delle connessioni di rete.
- **json**: per la serializzazione e deserializzazione dei dati in formato JSON.
- **random**: per la generazione di numeri casuali

### Funzione per Inviare Dati

```
# Funzione per inviare i dati al client
def send_data(client_socket, data):
    message = json.dumps(data).encode('utf-8')
    client_socket.send(message)
```

Questa funzione prende un socket del client e un dizionario di dati, lo converte in una stringa JSON, lo codifica in UTF-8 e lo invia al client tramite il socket.

### Inizializzazione del Server Socket

```
# Inizializzazione del server socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('0.0.0.0', 8888))
server_socket.listen(1)
```

- **socket.socket()**: crea un nuovo socket.
- **socket.AF\_INET**: indica che il socket utilizzerà IPv4.
- **socket.SOCK\_STREAM**: indica che il socket sarà di tipo TCP.

- ***bind(('0.0.0.0', 8888))***: associa il socket all'indirizzo IP '0.0.0.0' e alla porta 8888, permettendo al server di accettare connessioni su qualsiasi interfaccia di rete disponibile.
- ***listen(1)***: pone il server in ascolto, pronto a gestire una connessione in entrata.

### Accettazione della Connessione

```
print("Server in ascolto...")

# Accetta la connessione
client_socket, client_address = server_socket.accept()
print("Connessione accettata da:", client_address)
```

Il server si mette in attesa di una connessione in entrata. Quando un client si connette, il metodo `accept()` restituisce un nuovo socket per la comunicazione con il client e l'indirizzo del client.

### Generazione e Invio dei Dati

```
try:
    pitchInizio = random.randint(-20, 20)
    rollInizio = random.randint(-20, 20)
    yawInizio = random.randint(-20, 20)

    while True:
        x = 1
        while x == 1:
            pitch = random.randint(pitchInizio - 1, pitchInizio + 1)
            roll = random.randint(rollInizio - 1, rollInizio + 1)
            yaw = random.randint(yawInizio - 1, yawInizio + 1)
            if (pitch < 20 and pitch > -20) and (roll < 20 and roll >
-20) and (yaw < 20 and yaw > -20):
                x = 0
                if (roll == 0):
                    roll = 1

            pitchInizio = pitch
            rollInizio = roll
            yawInizio = yaw

        # Invia i dati al client
        data = {'pitch': pitch, 'roll': roll, 'yaw': yaw}
        send_data(client_socket, data)
```

```
        time.sleep(0.1) # Attendi 100 millisecondi tra l'invio di ogni
messaggio

except KeyboardInterrupt:
    print("Connessione interrotta.")
    client_socket.close()
    server_socket.close()
```

- **Inizializzazione dei valori:** Vengono generati casualmente i valori iniziali di pitch, roll e yaw compresi tra -20 e 20.
- **Ciclo infinito:** Viene eseguito un ciclo continuo per generare e inviare dati:
- **Generazione dei dati:** Nuovi valori di pitch, roll e yaw sono generati in modo che ogni valore non differisca di più di 1 dall'ultimo valore inviato. Se i valori generati sono all'interno dell'intervallo da -20 a 20, vengono accettati.
- **Condizione particolare per roll:** Se roll è zero, viene impostato a 1 per evitare di inviare un valore di zero.
- **Invio dei dati:** I dati vengono inviati al client utilizzando la funzione `send_data`.
- **Attesa:** Il server attende 100 millisecondi prima di generare e inviare nuovi dati.

### 3) Analisi del file 'civitanavi\_client.py'

Il file 'civitanavi\_client.py' implementa un client in Python che si connette al server per ricevere dati, li visualizza tramite un'interfaccia grafica Tkinter e li aggiorna in tempo reale.

Ecco cosa implementa in pochi punti il file client:

- Si connette a un server specificato.
- Riceve dati in formato JSON.
- Aggiorna un'interfaccia grafica Tkinter con i dati ricevuti.
- Gestisce eventuali errori di decodifica JSON.
- Chiude il socket in modo corretto in caso di interruzione dell'esecuzione.

Andiamo ad esaminare nel dettaglio i componenti del file.

#### Importazioni

```
import socket
import display
```

```
import json
import tkinter as tk
```

- **socket:** per la creazione e gestione delle connessioni di rete.
- **display:** (ipotizziamo che sia un modulo personalizzato) per la gestione della visualizzazione dei dati.
- **json:** per la serializzazione e deserializzazione dei dati in formato JSON.
- **tkinter:** per creare l'interfaccia grafica del client

### Funzione per Aggiornare i Dati

```
# Funzione per aggiornare i dati nella schermata Tkinter
def update_data(roll,pitch):
    pitch_label.config(text=roll)
    roll_label.config(text=pitch)
```

Questa funzione aggiorna i valori di pitch e roll visualizzati nell'interfaccia grafica Tkinter.

### Connessione al Server

```
# Connessione al server
ipserver = "127.0.0.1"
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((ipserver, 8888))
```

- **ipserver:** l'indirizzo IP del server a cui il client si conatterà (in questo caso, il localhost).
- **socket.socket():** crea un nuovo socket.
- **connect((ipserver, 8888)):** connette il socket al server specificato sull'indirizzo IP e porta indicati.

### Configurazione dell'Interfaccia Grafica Tkinter

```
root = tk.Tk()
root.title("Dati dal server")

pitch_label = tk.Label(root, text="Pitch: ")
pitch_label.pack()

roll_label = tk.Label(root, text="Rooll: ")
roll_label.pack()
```

- **tk.Tk():** crea una nuova finestra Tkinter.
- **root.title("Dati dal server"):** imposta il titolo della finestra.
- **tk.Label:** crea etichette per visualizzare i dati di pitch e roll.
- **pack():** posiziona le etichette nella finestra.



### Ciclo Principale per Ricevere e Visualizzare Dati

```
try:
    while True:
        msg = client_socket.recv(1024)
        msgs = msg.strip().decode('utf-8').split('\n')
        for msg in msgs:
            try:
                data = json.loads(msg)
                print(data)
                roll = float(data['roll'])
                pitch = float(data['pitch'])

                update_data(roll, pitch)

                display.update_components(roll, pitch)

                root.update()
            except json.decoder.JSONDecodeError as e:
                print("Errore di decodifica JSON:", e)

except KeyboardInterrupt:
    client_socket.close()

root.mainloop()
```

- **try/except:** gestisce eventuali interruzioni da tastiera per chiudere correttamente il socket.
- **client\_socket.recv(1024):** riceve fino a 1024 byte di dati dal server.
- **msg.strip().decode('utf-8').split('\n'):** rimuove gli spazi bianchi, decodifica i dati in una stringa UTF-8 e divide i messaggi separati da nuove righe.
- **json.loads(msg):** converte la stringa JSON in un dizionario Python.
- **print(data):** stampa i dati ricevuti per il debug.
- **roll = float(data['roll']):** converte il valore di roll in un float.
- **pitch = float(data['pitch']):** converte il valore di pitch in un float.
- **update\_data(roll, pitch):** aggiorna i dati visualizzati nell'interfaccia grafica.
- **display.update\_components(roll, pitch):** aggiorna i componenti del display personalizzato.
- **root.update():** aggiorna la finestra Tkinter per riflettere i nuovi dati.
- **json.decoder.JSONDecodeError:** gestisce eventuali errori di decodifica JSON.

- **KeyboardInterrupt:** chiude il socket quando l'esecuzione viene interrotta dall'utente.

## 4) Analisi del file 'display.py'

Il file 'display.py' implementa una classe Display per la visualizzazione di un display di volo primario (nel nostro caso per un aereo superleggero) utilizzando il modulo Tkinter per l'interfaccia grafica in Python. La classe utilizza vari componenti grafici per rappresentare elementi come l'orizzonte artificiale, la scala di beccheggio e il mirino.

Ora andremo a visualizzare nel dettaglio i componenti del file:

### Importazione di Moduli Personalizzati

```
# Aggiusta il percorso per trovare i moduli src
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__),
    '..', 'src')))

from src.Viewfinder import Viewfinder
from src.Horizon import Horizon
from src.PitchLadder import PitchLadder
from src.Calc import Calc
```

Questa parte del codice aggiunge la directory src al percorso dei moduli Python e importa le classi Viewfinder, Horizon, PitchLadder e Calc, che sono definite in moduli all'interno della directory src.

La classe **Display** è responsabile della configurazione dell'interfaccia grafica e dell'aggiornamento dei componenti grafici basati sui dati di rollio e beccheggio.

### Inizializzazione della Classe

```
class Display:
    def __init__(self, width, height):
        self.width = width # Larghezza finestra
        self.height = height # Altezza finestra
        self.xFs = 0 # Larghezza minima
        self.xNd = width # Larghezza massima
        self.calc = Calc() # Istanza della classe Calc
        self.finestra = tk.Tk() # Crea la finestra principale
dell'applicazione
```

```

        self.finestra.geometry(f"{width}x{height}") # Imposta le
dimensioni della finestra
        self.finestra.title("Primary Flight Display") # Imposta il
titolo della finestra
        self.canvas = tk.Canvas(self.finestra, width=width,
height=height) # Crea un widget Canvas
        self.canvas.pack() # Aggiunge il Canvas alla finestra
        self.horizon = Horizon(width, height, self.canvas) # Crea
un'istanza della classe Horizon
        self.pitchLadder = PitchLadder(width, height, self.canvas,
self.calc) # Crea un'istanza della classe PitchLadder
        self.viewfinder = Viewfinder(width, height, self.canvas) #
Crea un'istanza della classe Viewfinder

```

- **Dimensioni della Finestra:** width e height specificano le dimensioni della finestra.
- **xFs e xNd:** Variabili per la larghezza minima e massima.
- **Istanza Calc:** Una classe che probabilmente contiene calcoli utili per le trasformazioni grafiche.
- **Finestra Tkinter:** Creazione e configurazione della finestra principale.
- **Canvas:** Un'area di disegno all'interno della finestra per visualizzare i componenti.
- **Componenti Grafici:** Horizon, PitchLadder e Viewfinder sono istanze di classi personalizzate che rappresentano diversi elementi del PFD.

### Aggiornamento dei Componenti

```

def update_components(self, roll, pitch):
    # Converti pitch e roll in radianti
    radPitch = math.radians(pitch)
    radRoll = math.radians(roll)
    sinPitch = math.sin(radPitch)
    tanRoll = math.tan(radRoll)

    # Calcola le coordinate per il centro della linea
dell'orizzonte
    xm = self.width / 2
    ym = (self.height) - ((sinPitch + 1) * self.height)

    # Calcola le coordinate della linea dell'orizzonte
    yFs = ym + (tanRoll * (-1)) * (self.xFs - xm)
    yNd = ym + (tanRoll * (-1)) * (self.xNd - xm)
    self.yFs, self.yNd = yFs, yNd # Memorizza yFs e yNd come
variabili di classe

```

```

        # Disegna tutte le linee della scala di beccheggio
        self.pitchLadder.draw_all_lines(pitch, roll, self.xFs,
self.xNd, self.horizon.lineCoords)

        # Aggiorna le coordinate dell'orizzonte
        self.horizon.newCoords(0, 1, self.horizon.earthCoords, yFs,
yNd, self.horizon.earthPolygon, True)
        self.horizon.newCoords(0, 1, self.horizon.lineCoords, yFs, yNd,
self.horizon.line, True)

```

- **Conversione degli Angoli:** I valori di pitch e roll vengono convertiti da gradi a radianti.
- **Calcoli delle Coordinate:** Vengono calcolate le coordinate per disegnare la linea dell'orizzonte in base agli angoli di rollio e beccheggio.
- **Aggiornamento della Scala di Beccheggio:** draw\_all\_lines aggiorna tutte le linee della scala di beccheggio in base ai nuovi valori di pitch e roll.
- **Aggiornamento delle Coordinate dell'Orizzonte:** newCoords aggiorna le coordinate dell'orizzonte terrestre e della linea dell'orizzonte nel display.

### a) Analisi del file 'Calc.py'

Il file 'Calc.py' implementa una classe Calc che fornisce una serie di metodi per effettuare calcoli geometrici. Questi metodi includono il calcolo delle intersezioni tra una retta e una circonferenza, la lunghezza di un segmento, punti equidistanti da un centro di un segmento, l'equazione della retta e il punto di intersezione tra due rette.

Andiamo ad analizzare i diversi metodi del file:

#### Metodo 'intersezione\_retta\_circonferenza'

```

def intersezione_retta_circonferenza(self, m, cx=300,
cy=300):
    x = 1 / math.sqrt(1 + m**2)
    y = m / math.sqrt(1 + m**2)
    P1 = (cx + x, cy + y)
    P2 = (cx - x, cy - y)
    return P1, P2

```

Questo metodo calcola i punti di intersezione tra una retta con coefficiente angolare  $m$  e una circonferenza centrata nel punto (300, 300). Restituisce due punti di intersezione (P1, P2).

#### Metodo 'lunghezza segmento'

```
def lunghezza_segmento(self, P1, P2):  
    return math.sqrt((P2[0] - P1[0])**2 + (P2[1] -  
P1[1])**2)
```

Questo metodo calcola la lunghezza del segmento che collega due punti P1 e P2 utilizzando la distanza euclidea.

#### Metodo 'punti\_equidistanti'

```
def punti_equidistanti(self, P1, P2, distanza):  
    # Calcola il centro del segmento P1P2  
    C = ((P1[0] + P2[0]) / 2, (P1[1] + P2[1]) / 2)  
    # Calcola la direzione del segmento P1P2  
    dx = P2[0] - P1[0]  
    dy = P2[1] - P1[1]  
    # Normalizza la direzione del segmento  
    lunghezza = self.lunghezza_segmento(P1, P2)  
    dx /= lunghezza  
    dy /= lunghezza  
    # Calcola i punti A e B equidistanti dal centro C di una  
    certa distanza  
    A = (C[0] + distanza * dx, C[1] + distanza * dy)  
    B = (C[0] - distanza * dx, C[1] - distanza * dy)  
    return A, B
```

Questo metodo calcola due punti equidistanti dal centro del segmento definito dai punti P1 e P2, a una certa distanza.

#### Metodo 'equazione\_retta'

```
@staticmethod  
def equazione_retta(S, E):  
    x1, y1 = S  
    x2, y2 = E  
    if x2 == x1:  
        return 0  
    m = (y2 - y1) / (x2 - x1)  
    return m
```

Questo metodo statico calcola il coefficiente angolare della retta passante per i punti S ed E.

**Metodo 'coefficiente\_angolare'**

```
@staticmethod
def coefficiente_angolare(S, E):
    x1, y1 = S
    x2, y2 = E
    if x2 == x1:
        return float('inf') # La retta è verticale,
restituisce l'infinito
    return (y2 - y1) / (x2 - x1)
```

Questo metodo statico calcola il coefficiente angolare della retta passante per i punti S ed E, restituendo infinito se la retta è verticale.

**Metodo 'punto\_intersezione'**

```
@staticmethod
def punto_intersezione(r1x, r1y, r2x, r2y, r3x, r3y, r4x,
r4y):
    if r2x - r1x == 0: # Se la retta è verticale, il
coefficiente angolare è indefinito
        m1 = float('inf')
    else:
        m1 = (r2y - r1y) / (r2x - r1x)
    if r4x - r3x == 0: # Se la retta è verticale, il
coefficiente angolare è indefinito
        m2 = float('inf')
    else:
        m2 = (r4y - r3y) / (r4x - r3x)

    b1 = r1y - m1 * r1x if m1 != float('inf') else r1x
    b2 = r3y - m2 * r3x if m2 != float('inf') else r3x

    if m1 == m2: # Se i coefficienti angolari sono uguali,
le rette sono parallele e non si intersecano
        return 0, 0

    if m1 == float('inf'):
        x_intersezione = r1x
        y_intersezione = m2 * x_intersezione + b2
    elif m2 == float('inf'):
        x_intersezione = r3x
        y_intersezione = m1 * x_intersezione + b1
    else:
        x_intersezione = (b2 - b1) / (m1 - m2)
        y_intersezione = m1 * x_intersezione + b1
```

```
return x_intersezione, y_intersezione
```

Questo metodo statico calcola il punto di intersezione tra due rette definite dai punti (r1x, r1y)-(r2x, r2y) e (r3x, r3y)-(r4x, r4y). Se le rette sono parallele, restituisce (0, 0).

## b) Analisi del file 'Horizon.py'

Il file 'Horizon.py' implementa la classe **Horizon** che è responsabile della creazione e gestione dell'orizzonte artificiale. Utilizzando la libreria tkinter, è possibile disegnare e aggiornare gli elementi dell'orizzonte in tempo reale, riflettendo i cambiamenti nell'assetto dell'aeromobile. Il metodo 'newCoords' fornisce un meccanismo flessibile per aggiornare le coordinate degli elementi grafici, mantenendo una rappresentazione visiva accurata dell'orizzonte.

Andiamo a vedere nel dettaglio i componenti principali e le funzionalità di questa classe.

### Attributi della classe:

- **width e height:** La larghezza e l'altezza della tela su cui viene disegnato l'orizzonte artificiale.
- **canvas:** La tela di tkinter su cui vengono disegnati gli elementi dell'orizzonte artificiale.
- **skyCoords:** Coordinate dei punti per il poligono che rappresenta il cielo.
- **earthCoords:** Coordinate dei punti per il poligono che rappresenta la terra.
- **lineCoords:** Coordinate dei punti per la linea dell'orizzonte.
- **skyPolygon:** Poligono che rappresenta il cielo.
- **earthPolygon:** Poligono che rappresenta la terra.
- **line:** Linea che rappresenta l'orizzonte.

### Metodo '\_\_init\_\_'

```
def __init__(self, width, height, canvas):  
    self.width = width  
    self.height = height  
    self.canvas = canvas  
    self.skyCoords = [  
        (0, 0),  
        (width, 0),
```

```

        (width, height),
        (0, height)
    ] # Coordinate dei punti per il poligono che
    rappresenta il cielo
    self.earthCoords = [
        (0, height / 2),
        (width, height / 2),
        (width, height),
        (0, height)
    ] # Coordinate dei punti per il poligono che
    rappresenta la terra
    self.lineCoords = [
        (0, height / 2),
        (width, height / 2)
    ] # Coordinate dei punti per la linea dell'orizzonte
    self.skyPolygon =
self.canvas.create_polygon(*self.skyCoords, fill='sky blue',
outline='')
    self.earthPolygon =
self.canvas.create_polygon(*self.earthCoords, fill='saddle
brown', outline='')
    self.line = self.canvas.create_line(*self.lineCoords,
fill='white', width=4)

```

Il costruttore inizializza l'orizzonte artificiale, creando il cielo, la terra e la linea dell'orizzonte sulla tela di tkinter.

### Metodo 'newCoords'

```

def newCoords(self, coordFs, coordNd, coords, left, right,
element, state):
    coords[coordFs] = (coords[coordFs][0],
coords[coordFs][1] + (left if not state else -left))
    coords[coordNd] = (coords[coordNd][0],
coords[coordNd][1] + (right if not state else -right))
    flat_coords = [coord for point in coords for coord in
point]
    self.canvas.coords(element, *flat_coords)
    coords[coordFs] = (coords[coordFs][0],
coords[coordFs][1] - (left if not state else -left))
    coords[coordNd] = (coords[coordNd][0],
coords[coordNd][1] - (right if not state else -right))
    midpoint = ((coords[coordFs][0] + coords[coordNd][0]) /
2, (coords[coordFs][1] + coords[coordNd][1]) / 2)
    dx = coords[coordNd][0] - coords[coordFs][0]

```



```

dy = coords[coordNd][1] - coords[coordFs][1]
if dx == 0:
    perp_coords = [(midpoint[0] - self.width / 2,
midpoint[1]), (midpoint[0] + self.width / 2, midpoint[1])]
else:
    slope = dy / dx
    if dy == 0:
        perp_coords = [(midpoint[0], midpoint[1] -
self.height / 2), (midpoint[0], midpoint[1] + self.height / 2)]
    else:
        perp_slope = -1 / slope
        x1 = midpoint[0] - self.width / 2
        y1 = midpoint[1] + perp_slope * (x1 -
midpoint[0])
        x2 = midpoint[0] + self.width / 2
        y2 = midpoint[1] + perp_slope * (x2 -
midpoint[0])
        perp_coords = [(x1, y1), (x2, y2)]

```

Il metodo newCoords aggiorna le coordinate di un elemento dell'orizzonte artificiale. Accetta vari parametri per determinare i nuovi punti di coordinate e applica questi aggiornamenti alla tela. Viene anche calcolata una nuova coppia di coordinate perpendicolari basate sul centro del segmento e la sua inclinazione.

Andiamo a spiegare più nel dettaglio il funzionamento di questo metodo:

1. **Aggiornamento delle Coordinate:** Il metodo inizia aggiornando le coordinate dei punti in base ai valori di left e right. Questi valori determinano quanto spostare verticalmente i punti.
2. **Piatta Coords:** Le nuove coordinate vengono poi appiattite e applicate all'elemento sulla tela utilizzando il metodo coords di tkinter.
3. **Ripristino delle Coordinate:** Dopo l'aggiornamento sulla tela, le coordinate originali vengono ripristinate.
4. **Calcolo del Punto Medio e Direzione:** Viene calcolato il punto medio del segmento definito dalle nuove coordinate e la differenza dx e dy tra i punti.

5. **Calcolo delle Coordinate Perpendicolari:** A seconda della direzione del segmento (verticale, orizzontale o inclinata), vengono calcolate le nuove coordinate perpendicolari.
6. **Aggiornamento sulla Tela:** Queste coordinate perpendicolari vengono poi utilizzate per aggiornare gli elementi grafici sulla tela.

### c) Analisi del file 'PitchLadder.py'

Il file 'PitchLadder.py' contiene la classe '**PitchLadder**' che permette di visualizzare linee di riferimento per diverse inclinazioni (pitch) e aggiorna queste linee in base alle variazioni di assetto dell'aero.

Andiamo ad esaminare i componenti principali e le funzionalità di questa classe.

#### Attributi della classe

- **width e height:** la larghezza e l'altezza della tela su cui viene disegnata la scala di inclinazione del pitch.
- **canvas:** la tela di tkinter su cui vengono disegnati gli elementi della scala di inclinazione del pitch.
- **lines:** linee di inclinazione di 10°, 20° e 30° sopra e sotto l'orizzonte.
- **half\_lines:** linee di inclinazione di 5°, 15° e 25° sopra e sotto l'orizzonte.
- **quarter\_lines:** linee di inclinazione di 2.5°, 7.5°, 12.5°, 17.5°, 22.5° e 27.5° sopra e sotto l'orizzonte.
- **labels:** etichette per le linee principali (10°, 20° e 30°) a destra e sinistra delle linee.
- **calc:** oggetto che contiene funzioni di calcolo per le equazioni delle rette e intersezioni.

#### Metodo '**\_\_init\_\_**'

```
def __init__(self, width, height, canvas, calc):  
    self.width = width  
    self.height = height  
    self.canvas = canvas  
    self.lines = [] # Linee di 10°, 20° e 30° sopra  
l'orizzonte
```

```
self.half_lines = [] # Linee di 5°, 15° e 25° sopra
l'orizzonte
self.quarter_lines = [] # Linee di 2.5°, 7.5°, 12.5°,
17.5°, 22.5° e 27.5° sopra l'orizzonte
self.labels = [] # Scritta dei gradi 10, 20 e 30 per le
linee sopra l'orizzonte (una a destra della linea e una a
sinistra)
self.init_all_lines()
self.calc = calc
```

Il costruttore inizializza la scala di inclinazione del pitch, creando le linee principali, intermedie e minori, insieme alle etichette delle linee principali. Inoltre, memorizza un oggetto calc per i calcoli matematici.

### Metodi della classe per inizializzare le linee

- **Metodo 'init\_lines':** inizializza le linee principali della scala di inclinazione del pitch.
- **Metodo 'init\_half\_lines':** inizializza le linee intermedie della scala di inclinazione del pitch.
- **Metodo 'init\_quarter\_lines':** inizializza le linee minori della scala di inclinazione del pitch.
- **Metodo 'init\_labels':** inizializza le etichette dei gradi per le linee principali.
- **Metodo 'init\_all\_lines':** inizializza tutte le linee e le etichette della scala di inclinazione del pitch.

### Metodi della classe per disegnare le linee

- **Metodo 'draw\_all\_lines':** disegna tutte le linee della scala di inclinazione del pitch sulla tela.
- **Metodo 'draw\_lines':** disegna le linee principali sulla tela.
- **Metodo 'draw\_half\_lines':** disegna le linee intermedie sulla tela.
- **Metodo 'draw\_quarter\_lines':** disegna le linee minori sulla tela.
- **Metodo 'draw\_line':** questo metodo disegna una singola linea sulla tela in base all'angolo di inclinazione, pitch, roll, e altre coordinate. Calcola le posizioni delle linee utilizzando funzioni trigonometriche e aggiorna le coordinate delle linee e delle etichette.

## d) Analisi del file 'Viewfinder.py'

Il file 'Viewfinder.py' contiene la classe **Viewfinder** che permette di visualizzare un mirino centrale e due indicatori di vento (destro e sinistro) sulla tela.

Andiamo ad esaminare i componenti principali e le funzionalità di questa classe.

### Attributi della classe

- **width e height:** La larghezza e l'altezza della tela su cui viene disegnato il mirino.
- **canvas:** La tela di tkinter su cui vengono disegnati gli elementi del mirino.
- **viewfinder:** Il poligono che rappresenta il mirino centrale.
- **rightWind e leftWind:** I poligoni che rappresentano rispettivamente gli indicatori di vento destro e sinistro.

### Metodo '\_\_init\_\_'

```
def __init__(self, width, height, canvas):
    self.width = width
    self.height = height
    self.canvas = canvas
    self.viewfinder =
self.canvas.create_polygon(self.draw_viewfinder(),
outline='white', fill='black', width=2)
    self.rightWind = self.canvas.create_polygon(
self.draw_wind(True), outline='white', fill='black', width=2)
    self.leftWind =
self.canvas.create_polygon(self.draw_wind(False), outline='white
', fill='black', width=2 )
```

Il costruttore inizializza il mirino centrale e gli indicatori di vento destro e sinistro, disegnandoli sulla tela.

### Metodo 'draw\_wind'

```
def draw_wind(self, inverse):
    CC = (self.width // 2, self.height // 2)
    horizontal_offset = 120
    vertical_offset = 5
    lower_vertical_offset = 30
    side_offset = 15
    center_offset = 70
    A = (CC[0] - center_offset, CC[1] - vertical_offset)
```

```

        B = (CC[0] - center_offset - horizontal_offset, CC[1] -
vertical_offset)
        C = (CC[0] - center_offset - horizontal_offset, CC[1] +
10)
        D = (CC[0] - center_offset - side_offset, CC[1] + 10)
        E = (CC[0] - center_offset - side_offset, CC[1] +
lower_vertical_offset)
        F = (CC[0] - center_offset, CC[1] +
lower_vertical_offset)
        points = [A, B, C, D, E, F]
        if inverse:
            points = [(self.width - x, y) for x, y in points]
        return points

```

Questo metodo calcola i punti del poligono per gli indicatori di vento. Se *inverse* è True, inverte le coordinate orizzontali per disegnare l'indicatore di vento destro; altrimenti, disegna l'indicatore di vento sinistro.

#### Metodo 'draw\_viewfinder'

```

def draw_viewfinder(self):
    CC = (self.width // 2, self.height // 2)
    side_length = 17
    points = [
        (CC[0] - side_length // 2, CC[1] - side_length //
2),
        (CC[0] + side_length // 2, CC[1] - side_length //
2),
        (CC[0] + side_length // 2, CC[1] + side_length //
2),
        (CC[0] - side_length // 2, CC[1] + side_length // 2)
    ]
    return points

```

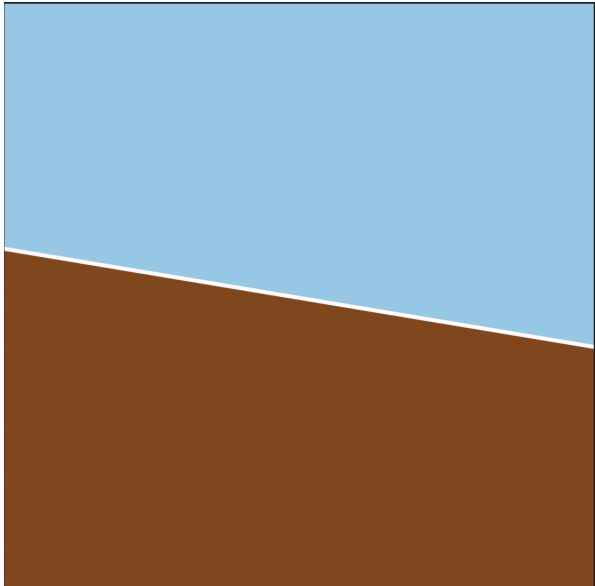
Questo metodo calcola i punti del poligono per il mirino centrale. Il mirino è un quadrato di lato *side\_length*, centrato sul punto centrale della tela.

## 5) Testing

Nel progetto sono presenti dei file all'interno della sottocartella 'test' che rappresentano ogni componente separato del progetto, in modo tale da avere una visione organizzata del medesimo.

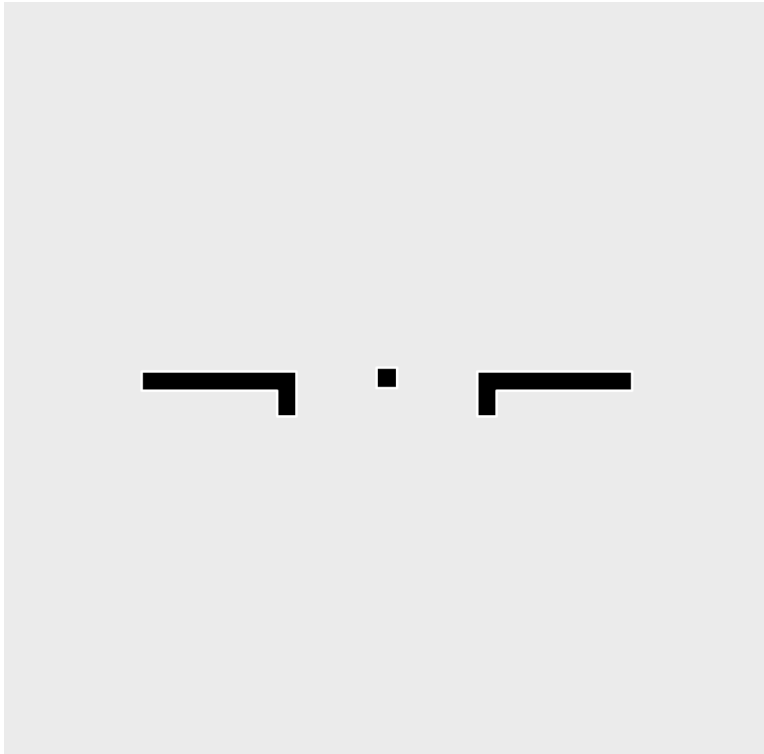
Nella seguente tabella verrà descritta in maniera dettagliata il funzionamento dei file all'interno la sottocartella 'test':

File	Descrizione
<i>'Test_Calc.py'</i>	<p>Il codice crea un'interfaccia grafica utilizzando tkinter per testare i metodi della classe Calc, che viene importata da un modulo situato in una directory specifica. L'interfaccia consiste in una finestra con un widget di testo che mostra i risultati dei test. Quando viene eseguito, il programma:</p> <ul style="list-style-type: none"><li>- Crea una finestra di 600 x 600 pixel con il titolo "Test Calc".</li><li>- Crea un'istanza della classe Calc.</li><li>- Definisce una serie di test che includono chiamate a diversi metodi della classe Calc.</li><li>- Visualizza i risultati di questi test nel widget di testo all'interno della finestra.</li></ul> <p>In poche parole visualizza a schermo i vari dati</p>
Immagine	

<pre>intersezione_retta_circonferenza Parametri: m=1.0, cx=300, cy=300 Risultato: ((300.70710678118655, 300.70710678118655), (299.29289321881345, 299.29289321881345))  lunghezza_segmento Parametri: P1=(100, 100), P2=(200, 200) Risultato: 141.4213562373095  punti_equidistanti Parametri: P1=(100, 100), P2=(200, 200), distanza=50 Risultato: ((185.35533905932738, 185.35533905932738), (114.64466094067262, 114.64466094067262))  equazione_retta Parametri: S=(100, 100), E=(200, 200) Risultato: 1.0  coefficiente_angolare Parametri: S=(100, 100), E=(200, 200) Risultato: 1.0  punto_intersezione Parametri: r1x=0, r1y=0, r2x=1, r2y=1, r3x=0, r3y=1, r4x=1, r4y=0 Risultato: (0.5, 0.5)</pre>	
<i>'Test_Horizon.py'</i>	Questo codice crea un'interfaccia grafica utilizzando il modulo tkinter per testare la visualizzazione dell'orizzonte del display (Primary Flight Display). Importa la classe Horizon da un modulo esterno e utilizza un canvas per disegnare l'orizzonte e la terra. L'interfaccia viene configurata con una finestra di dimensioni specificate, e la classe Horizon viene usata per aggiornare le coordinate grafiche dell'orizzonte e della terra. Quando eseguito, il programma apre una finestra che mostra l'orizzonte in tempo reale.
Immagine	
	

<code>'Test_PitchLadder.py'</code>	<p>Questo codice crea un'interfaccia grafica utilizzando tkinter per testare e visualizzare la scala di beccheggio (pitch ladder) del display (PFD). In particolare, il programma:</p> <ol style="list-style-type: none"><li>1. <b>Configura l'ambiente:</b> Importa i moduli necessari e imposta il percorso per trovare i moduli Calc e PitchLadder.</li><li>2. <b>Definisce la finestra principale:</b> Crea una finestra principale con dimensioni specificate e imposta lo sfondo nero.</li><li>3. <b>Crea un'area di disegno:</b> Utilizza un widget Canvas per disegnare gli elementi grafici della scala di beccheggio.</li><li>4. <b>Inizializza le classi di supporto:</b> istanzia le classi Calc per i calcoli matematici e PitchLadder per disegnare la scala di beccheggio.</li><li>5. <b>Disegna la scala di beccheggio:</b> Utilizza i metodi della classe PitchLadder per disegnare le linee di beccheggio sulla tela in base agli angoli di pitch e rollio forniti.</li></ol> <p>Quando eseguito, il programma mostra una finestra che simula la scala di beccheggio del display di volo primario, visualizzando la posizione della linea dell'orizzonte e le linee di beccheggio relative.</p>
Immagine	



<code>'Test_Viewfinder.py'</code>	<p>Questo codice crea un'applicazione GUI utilizzando tkinter per testare e visualizzare il mirino (viewfinder) di un display di volo primario (PFD). Ecco una descrizione sintetica delle sue funzionalità:</p> <ol style="list-style-type: none"><li>1. Configura l'ambiente importando i moduli necessari e aggiungendo il percorso per trovare i moduli personalizzati.</li><li>2. Definisce una finestra principale con dimensioni specificate e un titolo appropriato.</li><li>3. Crea un'area di disegno (Canvas) all'interno della finestra.</li><li>4. Istanza un oggetto della classe Viewfinder, che disegna il mirino sul Canvas.</li><li>5. Avvia il ciclo principale di tkinter, rendendo la finestra interattiva e visibile.</li></ol> <p>In sostanza, il codice visualizza un'interfaccia grafica che mostra il mirino del PFD, utilizzando la classe Viewfinder per gestire il disegno degli elementi grafici.</p>
<b>Immagine</b>	
	

## 6)Linee guida

In seguito verranno mostrati i passaggi da seguire per avviare il programma:

- **Distribuzione Locale**

1. Configura Python v.3.12 ([link](#))

2. Configura ambiente virtuale:

- 2.1. Crea ambiente virtuale

```
-m venv myenv
```

- 2.2. attiva ambiente virtuale Mac:

```
source myenv/bin/activate
```

- 2.3. attiva ambiente virtuale Windows:

```
.\myenv\Scripts\Activate
```

3. scarica tkinter v.8.6 ([link](#)) in ambiente virtuale:

```
pip install tk
```

- **Passaggi per eseguire il programma**

1. Far partire come primo file 'Civitanavi\_server.py'

- 1.1 Per Mac IOS

```
python3 civitanavi_server.py
```

- 1.2 Per Windows

```
python civitanavi_server.py
```

2. Fai partire poi come secondo file 'Civitanavi\_client.py'

- 2.1 Per Mac IOS

```
python3 civitanavi_client.py
```

2.2 Per Windows

*python civitanavi\_client.py*

## 7) Collegamento alle repository di GitHub

- **PIOTTI VITTORIO**  
<https://github.com/vittorioPiotti>
- **FABBIONI MATTEO**  
<https://github.com/MatteoFabbioni>
- **NICCIÀ LUCA**  
<https://github.com/lucaniccia>
- **CIUCALONI DIEGO**  
<https://github.com/Diego-Ciuck>