**gaussianPuff/scenario\_example.py => da dove si regolano tutte le configurazioni le ho concentrate invece di averle in config.py**

**gaussianPuff/gaussianModel.py**

Immagina di voler capire **come una sostanza chimica si disperde nell’aria** dopo essere stata rilasciata in un punto (una sorgente, tipo un camino o un piccolo spill).  
Il file gaussianModel.py è il cervello che fa proprio questo:  
simula **dove** e **quanto** di quella sostanza si muove e si concentra nello spazio e nel tempo.

💨 Quindi:

* definisce un **rettangolo di territorio** (una griglia con coordinate x e y),
* simula **il vento** (quanto forte e in che direzione soffia),
* definisce **la sorgente** (dove parte la sostanza e con che intensità),
* e poi **calcola la concentrazione** della sostanza in ogni punto della griglia, ora dopo ora.

Il risultato finale è una specie di **“mappa animata” della nuvola di sostanza**, che si sposta e si diluisce nel tempo.  
È esattamente quello che vedi nei software di dispersione atmosferica o nei modelli di contaminazione.

**🌬️ Due modi per simulare la dispersione**

Il modello può lavorare in due modalità:

1. **PLUME (pennacchio continuo)**  
   Immagina un camino che emette fumo *costantemente*. Il modello calcola come quella “scia” si muove e si allarga con il vento e la turbolenza dell’aria.
2. **PUFF (sbuffi)**  
   Immagina invece dei piccoli *puff*, come “soffiate” di gas rilasciate ogni tot minuti.  
   Ogni puff viene spinto dal vento, si allarga un po’, e poi sparisce dopo un certo tempo.  
   Il modello tiene traccia di tutti questi puff e somma i loro effetti.

**🧮 Cosa fa il codice, passo per passo (in parole normali)**

**1️⃣ Prepara la griglia**

Pensa alla griglia come a una scacchiera che copre la zona.  
Ogni quadratino è un punto dove il modello calcola quanta sostanza c’è.

Il codice crea i vettori **x** e **y** (le coordinate dei punti) e, se serve, anche **z** (l’altezza, ma in molti casi resta al suolo).  
Se gli dici, per esempio:

bounds=(-250, -250, 250, 250)

grid\_size=50

significa che stai simulando un quadrato di 500 metri per lato, diviso in 50×50 punti.

**2️⃣ Imposta le condizioni meteorologiche**

Qui decide com’è l’atmosfera:

* **velocità del vento**, per spingere la nuvola;
* **direzione del vento** (fissa o variabile);
* **stabilità atmosferica**, cioè quanto l’aria è “ferma” o “mossa” (nelle giornate calme la sostanza resta più concentrata).

**3️⃣ Simula il rilascio**

Per ogni punto della griglia e per ogni ora del giorno:

* calcola **quanta sostanza arriva lì** partendo dalla sorgente,
* usando la formula di dispersione gaussiana (la classica campana “soffiata” dal vento).

Nel caso PUFF, i “puff” vengono creati a intervalli regolari e “camminano” col vento, quindi la concentrazione è la somma dei puff ancora “in aria”.

**4️⃣ (Facoltativo) Effetto umidità**

Se il parametro humidify=True, il modello applica un effetto chiamato **igroscopia**:  
le particelle assorbono acqua dall’umidità dell’aria e diventano più grandi ⇒ più pesanti ⇒ cambia la concentrazione.  
Questo lo fa con la funzione apply\_hygroscopic\_growth.

**5️⃣ Restituisce i risultati**

Alla fine ti restituisce:

* C1 → un grande blocco di dati (una matrice 3D) con la concentrazione per ogni punto e tempo;
* (x, y, z) → le coordinate (i vettori);
* times → i momenti della simulazione;
* stability e wind\_dir → i parametri meteorologici usati;
* e qualche etichetta o i puff (se usi il modello “puff”).

**gaussianPuff/api\_gaussian.py**

Immagina di avere il “motore” che fa i conti della dispersione (gaussianModel.py).  
**api\_gaussian.py** è la **porta d’ingresso web**: riceve una richiesta HTTP con i parametri della simulazione, fa girare il motore e ti rimanda indietro un **risultato pronto da usare** (numeri e metadati).

**In pratica:**

* espone un endpoint: **POST /start\_simulation**;
* legge e valida il payload (con Pydantic);
* costruisce l’oggetto **ModelConfig** con i parametri meteo/sorgenti/modello;
* chiama **run\_dispersion\_model(config, bounds)** del motore;
* restituisce **metadati** e **range** delle concentrazioni (e, se chiedi esplicitamente, anche il campo 3D completo).

**Cosa riceve dal client**

La richiesta ha due parti:

1. **config**: tutto ciò che serve per la simulazione (giorni, vento, stabilità, tipo NPS, output, stacks, ecc.).
2. **altri campi**:
   * **bounds** = [xmin, ymin, xmax, ymax] → rettangolo di simulazione;
   * **seed** (opzionale) → per rendere la simulazione **riproducibile** quando c’è casualità (es. vento fluttuante);
   * **return\_field** (False di default) → se True, ti restituisce **tutta** la matrice C(x,y,t).

Nota: la validazione Pydantic ti protegge da payload sbagliati (tipi, lunghezze, enum invalidi, ecc.).

**Cosa fa “dietro le quinte” (step-by-step)**

1. **Log & seed**  
   Appena arriva la chiamata, logga la richiesta.  
   Se hai passato seed, lo imposta (anche per eventuali parti random future) **e** lo inoltra nel ModelConfig così il **motore** usa lo stesso seed per il suo RNG interno.
2. **Parsing dei parametri**  
   Converte le stringhe del payload in **enum** interni del modello (tipo vento, stabilità, NPS, output, modello di dispersione).  
   Questo evita errori del tipo “valore non previsto” e garantisce coerenza con il core.
3. **Costruisce ModelConfig**  
   Compone l’oggetto con tutti i campi attesi dal motore (giorni, RH, humidify, vento, stabilità, stacks, grid, ecc.).  
   Se manca qualcosa, usa i **default sensati** (es. grid\_size).
4. **Esegue la simulazione**  
   Chiama run\_dispersion\_model(config, bounds) del **gaussianModel.py**.  
   Ottiene: **C1** (concentrazioni 3D), gli assi **x,y,z**, i **times**, i dettagli meteo (stabilità, direzione del vento), e—se in modalità PUFF—i puff.
5. **Risposta “slim” per default**  
   Per non spedirti in rete un mattone enorme:
   * ritorna **solo metadati**: shape, estremi degli assi, #livelli z, #tempi, etichette (stabilità/vento), **seed** usato, e **min/max** della concentrazione;
   * **solo se** metti return\_field=True, aggiunge concentration = **C1** completo in lista (potenzialmente grande).
6. **Error handling & logging**  
   Se qualcosa va storto, logga lo stack trace e alza l’eccezione (così il client riceve l’errore e puoi debuggare).

**gaussianPuff/gaussianFunction.py**

**A cosa serve (in parole semplici)**

Questo file contiene **le formule fisiche** che trasformano i parametri della scena (sorgente, vento, stabilità dell’atmosfera, tempo) in **concentrazioni nello spazio**. In pratica è il “cuore matematico” che, dato **dove** sei, **quando** osservi e **com’è** l’atmosfera, ti dice **quanta sostanza** c’è lì in quel momento.

Più precisamente:

* prende la **posizione della sorgente** e quella del **punto di osservazione**,
* calcola **quanto vento e turbolenza** agiscono tra i due,
* e restituisce **quanta sostanza arriva lì** (cioè la concentrazione prevista).

In termini fisici è proprio il “motore matematico della dispersione”, cioè quello che trasforma tutti i parametri ambientali in **una mappa di concentrazione**.

Lavora con due scenari classici della dispersione:

1. **Pennacchio continuo (plume)**  
   Pensa a un camino che emette in modo costante. Il codice:
   * orienta la mappa rispetto al **vento** (capisce cos’è “sottovento” e “trasverso al vento”);
   * calcola **quanto si allarga** la nuvola in orizzontale e in verticale in base alla **stabilità atmosferica** e alla distanza dalla sorgente;
   * considera anche l’**immagine riflessa** sotto il suolo (il trucco matematico standard che simula la riflessione al terreno), così la somma di due campane verticali riproduce la concentrazione reale vicino al suolo;
   * restituisce la **mappa 2D** della concentrazione per quel livello di quota e istante.
2. **Sbuffo impulsivo (puff)**  
   Pensa a “palloncini” di gas rilasciati a intervalli. Per ciascun palloncino:
   * usa il **tempo trascorso** e la **velocità del vento** per stimare quanto si è **allargato** (più passa il tempo, più il “puff” si allarga);
   * misura la distanza del punto di interesse dal **centro del palloncino** e applica la classica **campana 3D** (con il termine di riflessione al suolo);
   * restituisce la concentrazione prodotta da **quel** palloncino in **quel** punto e **in quell’istante** (poi il modello più alto livello somma i contributi di tutti i puff attivi).

Elemento chiave in entrambi i casi è il calcolo delle **ampiezze (sigma)** della campana, che dipendono dalla **stabilità** e dalla **distanza/tempo**: atmosfera più “ferma” ⇒ la nuvola resta stretta e alta concentrazione; atmosfera più “turbolenta” ⇒ la nuvola si allarga e la concentrazione si diluisce. Queste “sigma” arrivano da una routine dedicata richiamata qui dentro.

**gaussianPuff/sigmaCalculation.py**

“Data la distanza dalla sorgente e quanto l’atmosfera è stabile, quanto si è già allargata la nube?”

In pratica:

* riceve dal motore (gaussianModel) la **classe di stabilità** (1–6) e le **distanze sottovento**,
* usa formule empiriche (tabellate in letteratura) per restituire **σ\_y e σ\_z**,
* poi il resto del modello (in gaussianFunction.py) li usa per calcolare le **concentrazioni** reali.

💡 Puoi pensarlo così:

* sigmaCalculation.py → calcola **la forma della campana** (quanto è larga e alta);
* gaussianFunction.py → applica quella forma per capire **dove e quanta sostanza c’è**.

Quando disegni una “campana gaussiana” della nuvola (plume/puff), ti servono due numeri fondamentali:

* **σ\_y**: quanto si allarga la nuvola **in orizzontale** (trasverso al vento);
* **σ\_z**: quanto si allarga la nuvola **in verticale**.

Questi due “sigma” dicono **quanto è larga** e **quanto è alta** (o spessa) la nuvola a una certa distanza dalla sorgente.  
Più sono grandi, più la concentrazione si spalma e diminuisce.

**sigmaCalculation.py** fa proprio questo: dati

* la **classe di stabilità atmosferica** (da molto instabile a molto stabile) e
* la **distanza sottovento** dalla sorgente (x),

calcola i valori di **σ\_y** e **σ\_z** da usare nelle formule della dispersione.

**Come ragiona (step-by-step)**

1. **Prende la distanza** x (in metri) e ne usa il valore assoluto (la dispersione dipende da “quanto sei lontano”, non dal segno).
2. **Sceglie i coefficienti** in base alla **stabilità**:
   * La stabilità è una categoria (numeri 1→6: molto instabile → molto stabile).
   * Per ogni categoria, ci sono **coefficienti tabulati** (a, b per σ\_z; c, d per σ\_y).
   * In alcune categorie (es. molto instabile o neutrale) i coefficienti **cambiano** a tratti in funzione della distanza (spezzate per range di x). Questo rispecchia le tabelle empiriche usate nei modelli classici (tipo Pasquill–Gifford).
3. **Calcola σ\_z (verticale)** con una legge di potenza:

quindi la dispersione verticale **cresce con la distanza**. C’è anche un **tetto massimo** (cap) a 5000 m per evitare valori fisicamente esagerati nei calcoli.

1. **Calcola σ\_y (orizzontale)** con una relazione che usa un **angolo** θ:
   * prima calcola (in radianti),
   * poi .  
     In pratica: la larghezza orizzontale cresce con la distanza ma in modo modulato dalla stabilità (tramite c, d e la funzione tan).
2. **Restituisce (σ\_y, σ\_z)** con la **stessa forma** dell’array x in input, così il resto del modello può usarli punto-per-punto sulla griglia.

**Perché è importante**

* La **stabilità atmosferica** governa **quanto rapidamente** la nuvola si allarga:
  + **instabile** ⇒ tanta turbolenza ⇒ σ grandi ⇒ la nuvola si “spalma” in fretta;
  + **stabile** ⇒ aria “ferma” ⇒ σ piccoli ⇒ la nuvola resta più concentrata e stretta.
* Le **spezzate** (coefficienti diversi per diversi range di distanza) sono un modo standard di catturare l’andamento reale osservato in campo.
* A valle, questi σ finiscono nelle formule di **gaussianFunction.py**, che trasformano σ\_y/σ\_z in **concentrazioni** su mappa e nel tempo.

**gaussianPuff/Sensor.py**

Nel progetto, i **sensori** sono il modo in cui “leggiamo” quello che il modello ha simulato.

* **SensorSubstance** è un singolo sensore chimico posizionato in (x, y, z):
  + prende il **campo di concentrazione** simulato (la mappa 3D C(x,y,t)) e lo **campiona** dove si trova il sensore → ottiene così una **serie temporale**;
  + può aggiungere **rumore realistico** alla misura (per simulare sensori veri);
  + può anche generare **dati sintetici** (una nuvoletta finta e uno **spettro di massa** finto) quando vuoi testare pipeline senza lanciare tutta la simulazione;
  + ha una logica di **guasto** (fault): se il sensore è “rotto”, restituisce NaN o dati placeholder.
* **SensorAir** è un “sensore meteo”: genera condizioni meteorologiche **verosimili** (tipo di vento, classe di stabilità, umidità, ecc.) pescandole in modo casuale ma coerente. Serve quando vuoi simulare rapidamente uno scenario senza un meteo reale.

In pratica:

il modello fisico crea la realtà “vera” (la nube). I sensori, con il loro rumore e possibili guasti, producono i **dati osservati** che poi userai per localizzare la sorgente, classificare le sostanze, addestrare modelli, ecc.

**gaussianPuff/plot\_utils.py**

Questo file contiene **solo funzioni di visualizzazione** dei risultati del modello, per guardare la nube in modi diversi:

* **plan view**: la vista dall’alto integrata o istantanea (mappe 2D con colore = concentrazione);
* **time series**: grafici nel tempo (es. concentrazione in un punto/indice di griglia);
* **slice verticali/3D**: sezioni e superfici 3D per “vedere” la nuvola nello spazio;
* **animazioni**: una gif/animazione che fa scorrere il tempo;
* **mappe geografiche (Folium)**: per proiettare una heatmap su una mappa reale (lat/lon);
* **marker su mappa**: sorgente e sensori sopra la mappa.

In pratica: il motore fa i conti ⇒ plot\_utils.py ti **fa vedere** cosa è successo, come si è mossa la nube, quanto è grande, dove è più concentrata, ecc.